ECE1003 (Digital Logic Design) Lab Report

VIT-AP UNIVERSITY

Name : Khan Mohd. Owais Raza
Reg. No. : 20BCD7138
Submitted to : Prof. Suseela Vappangi
Course : Digital Logic Design
Course Code : ECE1003
Slot : L57+L58

Experiment-1	Basic Logic Gates
Experiment-2	Half Adder, Full Adder, Half Subtractor, Full Subtractor
Experiment-3	4 Bit Binary Adder & 4 Bit Binary Subtractor
Experiment-4	MUX and DE-MUX
Experiment-5	Encoders & Decoders
Experiment-6	1, 2 ,4 Bit Comparators
Experiment-7	Flip Flops
Experiment-8	Shift Registers
Experiment-9	3 Bit Synchronous counter
Experiment-10	3 Bit Asynchronous counter

ECE1003_Digital Logic Design_L57+L58_Experiment-1

Name : Khan Mohd. Owais Raza
Reg. No. : 20BCD7138

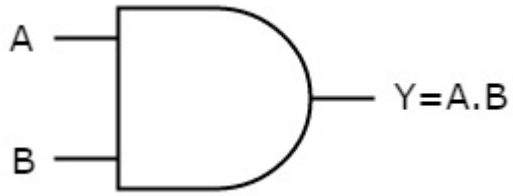
Aim : Design and verify basic logic gates in verilog

Software used : Xilinx Vivado 2014.2

Theory :

1. *AND Gate* :- This gate works in the same way as the logical operator **"AND"**. The AND gate is a circuit that performs the AND operation of the inputs. This gate has a minimum of 2 input values and an output value.

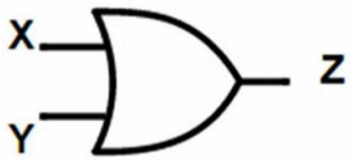
Y=A AND B AND C AND D……N
Y=A.B.C.D……N
Y=ABCD……N



x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

2. *OR Gate* :- This gate works in the same way as the logical operator **"or"**. The OR gate is a circuit which performs the OR operation of the inputs. This gate also has a minimum of 2 input values and an output value.

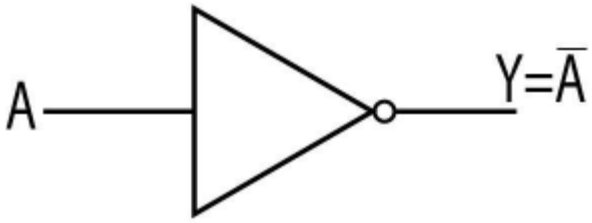
Y=A OR B OR C OR D……N
Y=A+B+C+D……N



Inputs		Output
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

3. *NOT Gate* :- The NOT gate is also called an inverter. This gate gives the inverse value of the input value as a result. This gate has only one input and one output value.

Y=NOT A
Y=A'

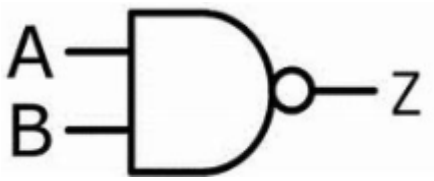


Input	Output
A	B
0	1
1	0

4. *NAND Gate* :- The NAND gate is the combination of AND gate and NOT gate. This gate gives the same result as a NOT-AND operation. This gate can have two or more than two input values and only one output value.

Y=A NOT AND B NOT AND C NOT AND D……N
Y=A NAND B NAND C NAND D……N

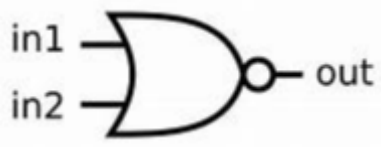
Inputs		Output
A	B	(AB)'
0	0	1
0	1	1
1	0	1
1	1	0



5. *NOR Gate* :- The NOR gate is the combination of an OR gate and NOT gate. This gate gives the same result as the NOT-OR operation. This gate can have two or more than two input values and only one output value.

Y=A NOT OR B NOT OR C NOT OR D……N
Y=A NOR B NOR C NOR D……N

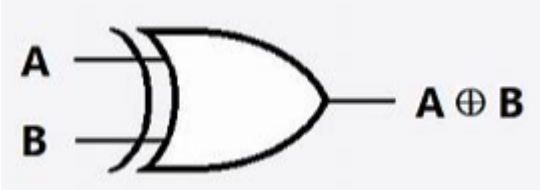
Inputs		Output
A	B	(AB)'
0	0	1
0	1	0
1	0	0
1	1	0



6. *XOR Gate* :- The XOR gate is also known as the Ex-OR gate. The XOR gate is used in half and full adder and subtractor. The exclusive-OR gate is sometimes called as EX-OR and X-OR gate. This gate can have two or more than two input values and only one output value.

Y=A XOR B XOR C XOR D……N
Y=A⊕B⊕C⊕D……N
Y=AB'+A'B

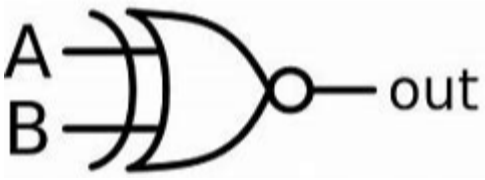
Inputs		Output
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0



7. *XNOR Gate* :- The XNOR gate is also known as the Ex-NOR gate. The XNOR gate is used in half and full adder and subtractor. The exclusive-NOR gate is sometimes called as EX-NOR and XNOR gate. This gate can have two or more than two input values and only one output value.

Y=A XNOR B XNOR C XNOR D……N
Y=A⊖B⊖C⊖D……N
Y=A'B'+AB

Inputs		Output
A	B	A⊖B
0	0	1
0	1	0
1	0	0
1	1	1



I] OR Gate:

Theory : The OR gate is a digital logic gate that implements logical disjunction – it behaves according to the truth table to the right. A HIGH output (1) results if one or both the inputs to the gate are HIGH (1). If neither input is high, a LOW output (0) results.

1. Data Flow
2. Structural
3. Behavioral

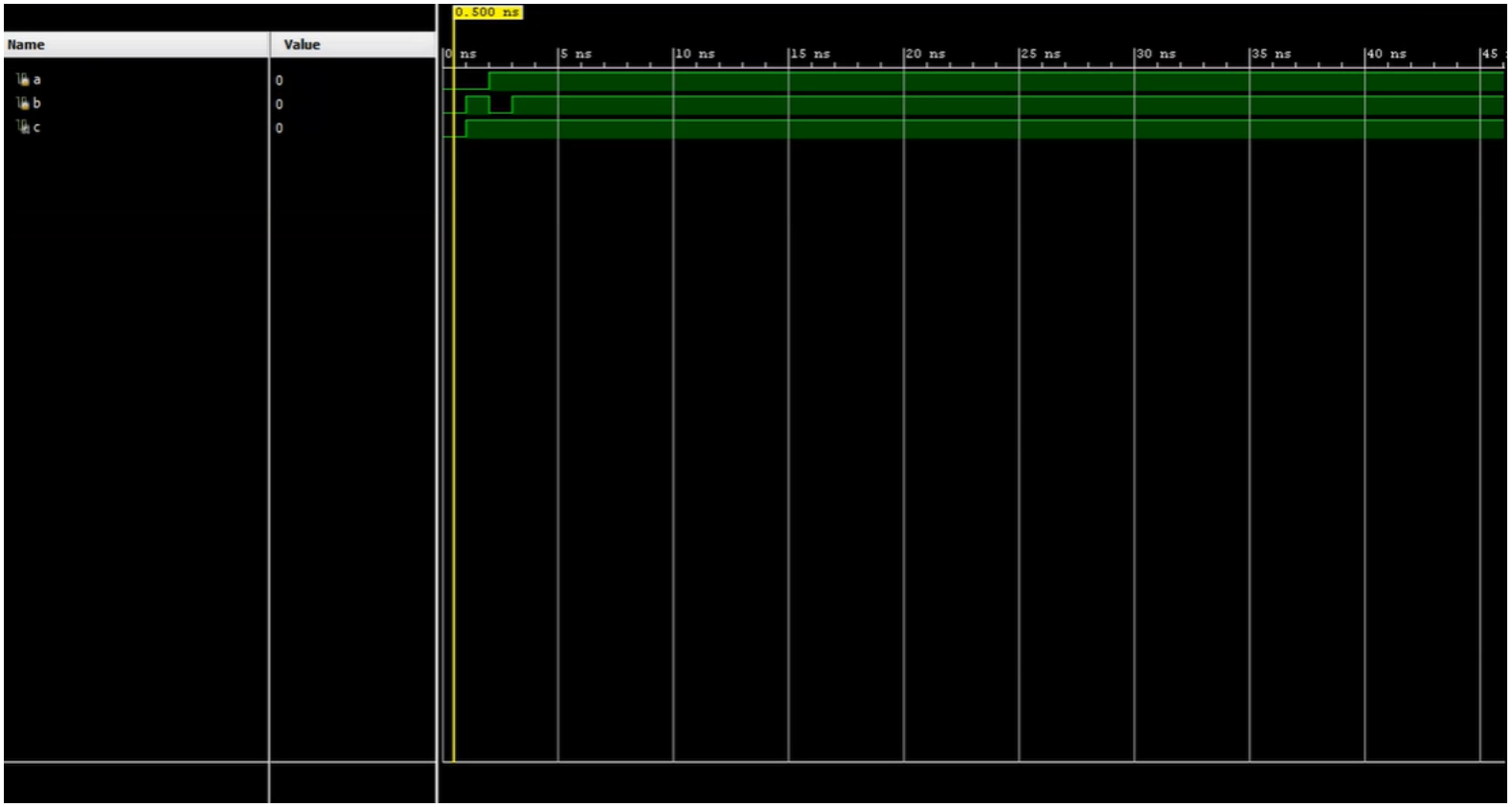
x	y	z
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

Data Flow

Input:

```
module OR_GATE_VITAPSTUDENTSWINTER(x, y, z);
reg a, b;
wire c;
OR_GATE_VITAPSTUDENTS a1(a, b, c);
initial
begin
a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
end
endmodule
```

Output:



Structural

```
module OR_GATE_VITAPSTUDENTS(x, y, z);
input x, y;
output z;
always @(x, z)
begin
if (x==1’b0 && y==1’b0)
z=1’b0;
else
z=1’b1
end
endmodule
```

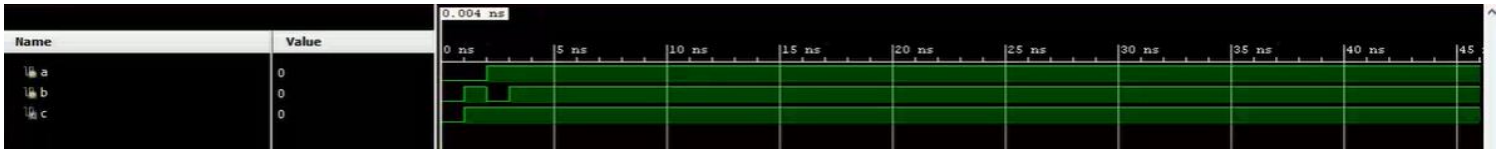
Testbench :

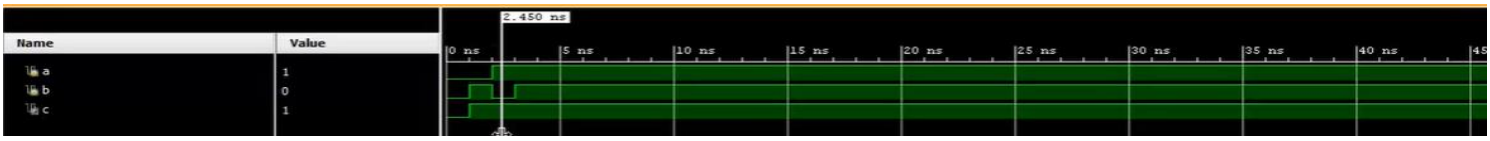
tb_OR_GATE.v *

C:/Users/admin/OR_GATE_VITAPSTUDENTSWINTER/OR_GATE_VITAPSTUDENTSWINTER.srcs/sim_1/

```
22
23 module tb_OR_GATE;
24 reg A, B;
25 wire Y;
26 OR_2_behavioral Indtance0 (Y, A, B);
27 initial begin
28     A = 0; B = 0;
29  #1 A = 0; B = 1;
30  #1 A = 1; B = 0;
31  #1 A = 1; B = 1;
32 end
33 initial begin
34     $monitor ("%t | A = %d| B = %d| Y = %d", $time, A, B, Y);
35     $dumpfile("dump.vcd");
36     $dumpvars();
37 end
38 endmodule
39
```

Output :





II] XOR Gate:

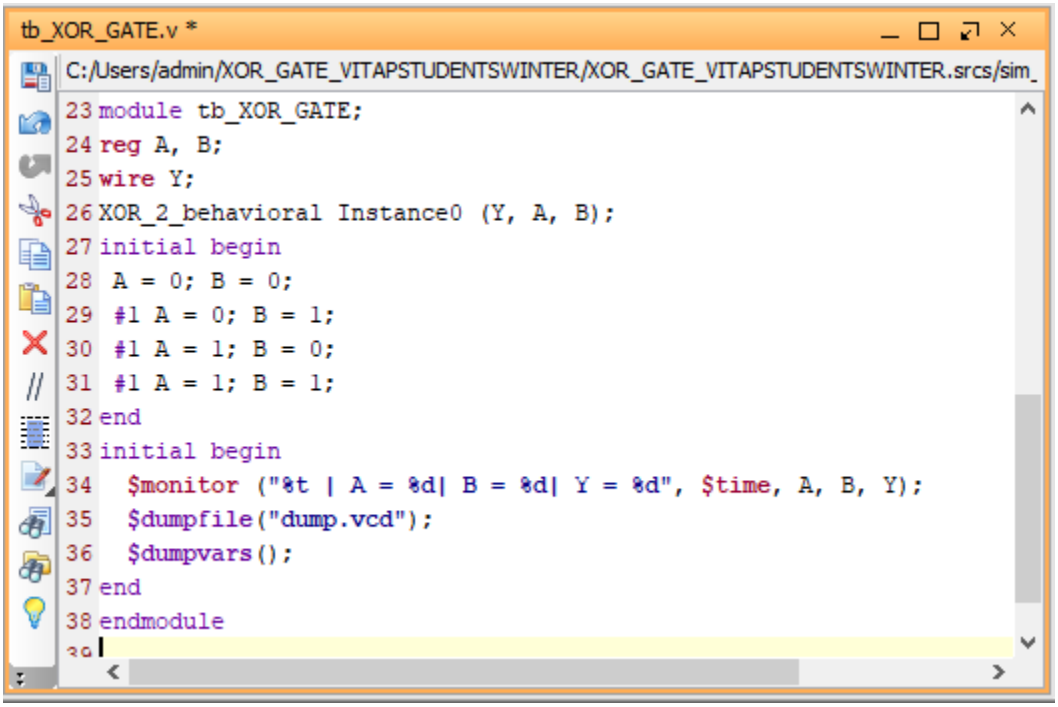
Theory : XOR gate (sometimes EOR, or EXOR and pronounced as Exclusive OR) is a digital logic gate that gives a true (1 or HIGH) output when the number of true inputs is odd. An XOR gate implements an exclusive or; that is, a true output results if one, and only one, of the inputs to the gate is true. If both inputs are false (0/LOW) or both are true, a false output results.

```
module XOR_GATE_VITAPSTUDENTSWINTER(x, y, z);
input x, y;
output reg z;
//DATA FLOW
//assign z=x^y;
//BEHAVIOURAL MODELING
always @(x, y)
begin
if (x==1’b0 && y==1’b0)
z=1’b0;
if (x==1’b1 && y==1’b1)
z=1’b0;
else
z=1’b1;
```

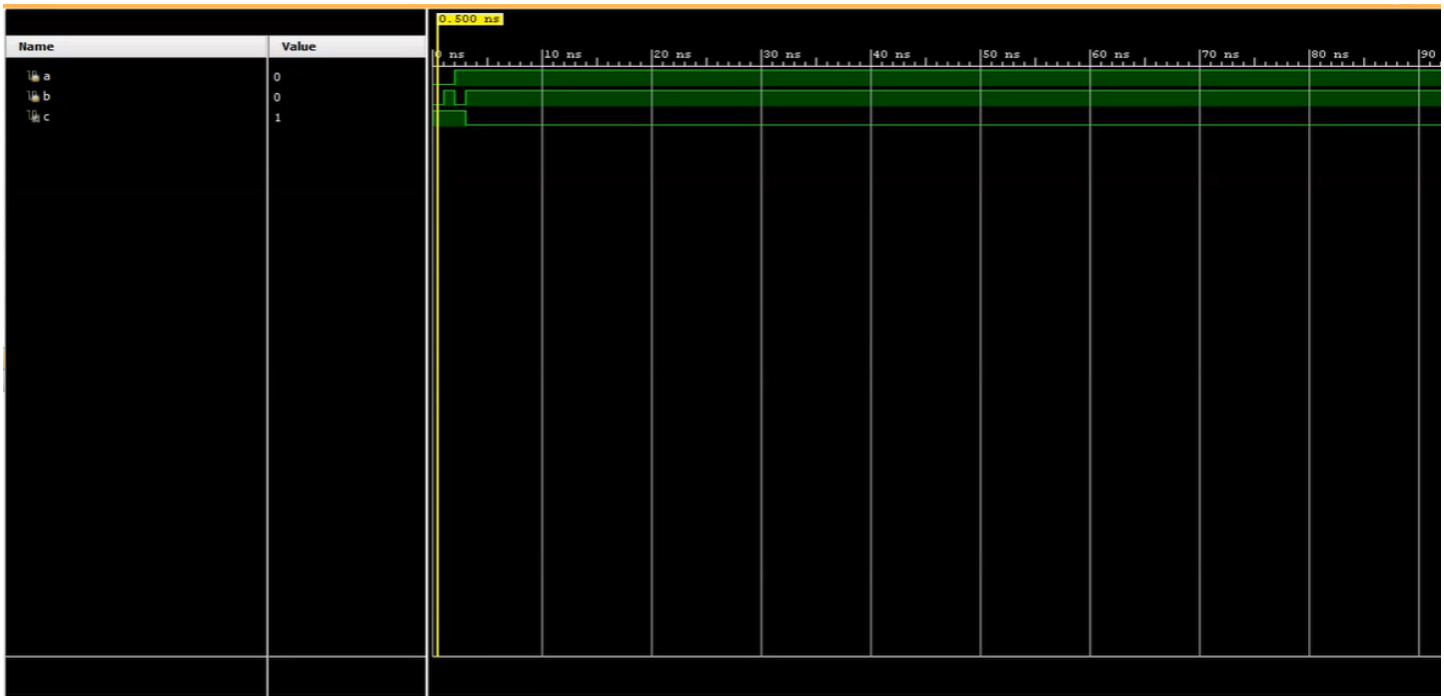
```
end
endmodule
```

```
module tb_XOR_GATE;
reg a, b;
XOR_GATE_WINTERVITAP2021ST dl(a, b, c);
initial
begin
a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
end
endmodule
```

Testbench for XOR gate and output :



Output :



NOR Gate :

Theory : The NOR gate is a digital logic gate that implements logical NOR - it behaves according to the truth table to the right. A HIGH output (1) results if both the inputs to the gate are LOW (0); if one or both input is HIGH (1), a LOW output (0) results. NOR is the result of the negation of the OR operator. It can also in some senses be seen as the inverse of an AND gate.

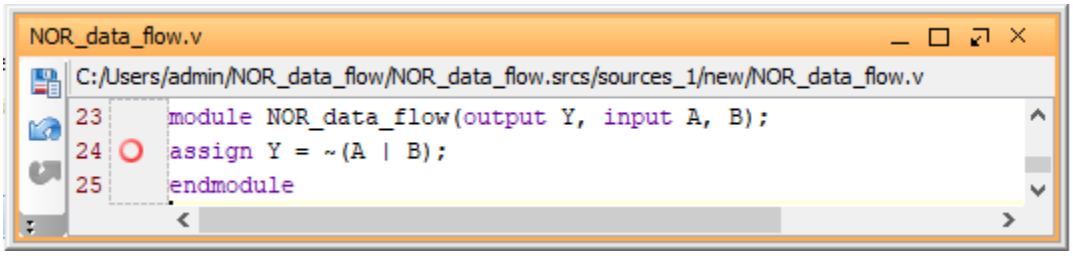
x	y	z
0	0	1
0	1	0
1	0	0
1	1	0

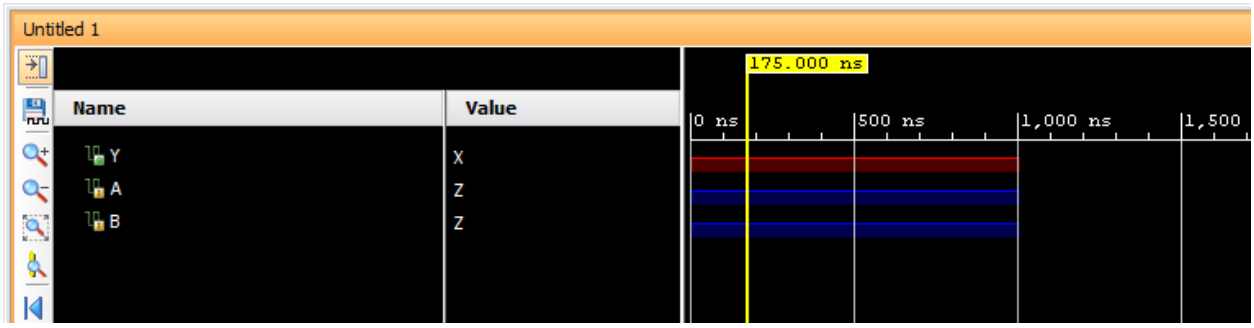
Input :

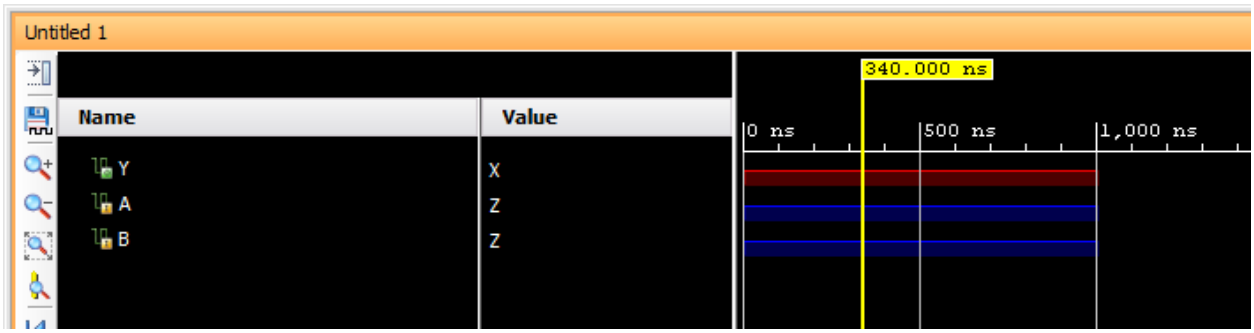
```
module tb_NOR_GATE;
reg a, b;
NOR_GATE_WINTERVITAP2021ST dl(a, b, c);
initial
begin
a=0; b=0;
#1 a=0; b=1;
```

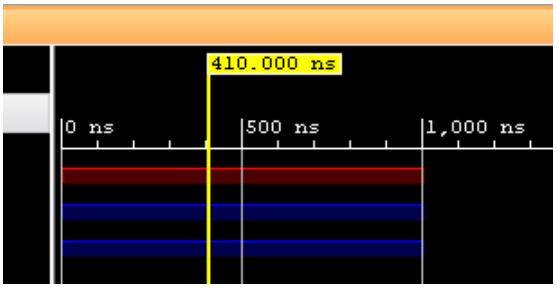
```
#1 a=1; b=0;
#1 a=1; b=1;
end
endmodule
```

```
module NOR_GATE_VITAPSTUDENTS(x, y, z);
entity nor1 is
    Port ( x : in std_logic;
           y : in std_logic;
           z : out std_logic);
end nor1;
architecture nor11 of nor1 is
begin
 y<= x nor y;
end
endmodule
```


NOR_data_flow.v
C:/Users/admin/NOR_data_flow/NOR_data_flow.srcs/sources_1/new/NOR_data_flow.v
23 module NOR_data_flow(output Y, input A, B);
24 assign Y = ~(A | B);
25 endmodule


Untitled 1
Name
Value
Y
A
B
X
Z
Z
175.000 ns
0 ns
500 ns
1,000 ns
1,500


Untitled 1
Name
Value
Y
A
B
X
Z
Z
340.000 ns
0 ns
500 ns
1,000 ns


410.000 ns
0 ns
500 ns
1,000 ns


610.000 ns
0 ns
500 ns
1,000 ns

```
1  module NOR_Gate_behavioral_tb;
2  reg A, B;
3  wire Y;
4  NOR_2_behavioral Instance0 (Y, A, B);
5  initial begin
6      A = 0; B = 0;
7   #1 A = 0; B = 1;
8   #1 A = 1; B = 0;
9   #1 A = 1; B = 1;
10 end
11 initial begin
12     $monitor ("%t | A = %d| B = %d| Y = %d", $time, A, B, Y);
13     $dumpfile("dump.vcd");
14     $dumpvars();
15 end
16 endmodule
```

III] XNOR Gate :

Theory : The XNOR gate (sometimes ENOR, EXNOR or NXOR and pronounced as Exclusive NOR) is a digital logic gate whose function is the logical complement of the Exclusive OR (XOR) gate. The two-input version implements logical equality, behaving according to the truth table to the right, and hence the gate is sometimes called an "equivalence gate".

XNOR_data_flow.v *

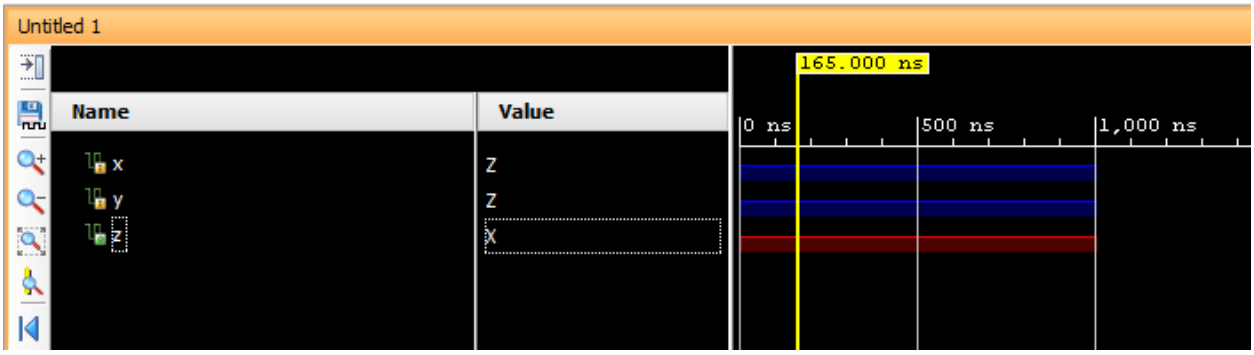
C:/Users/admin/XNOR_data_flow/XNOR_data_flow.srcs/sources_1/ne

```
23 module XNOR_data_flow(output Y, input A, B);
24 assign Y = ~(A ^ B);
25 endmodule
26
```

testbench_XNOR_gate.v *

C:/Users/admin/testbench_XNOR_gate/testbench_XNOR_gate.srcs/sim_1/new/testbench_XNOR_ga

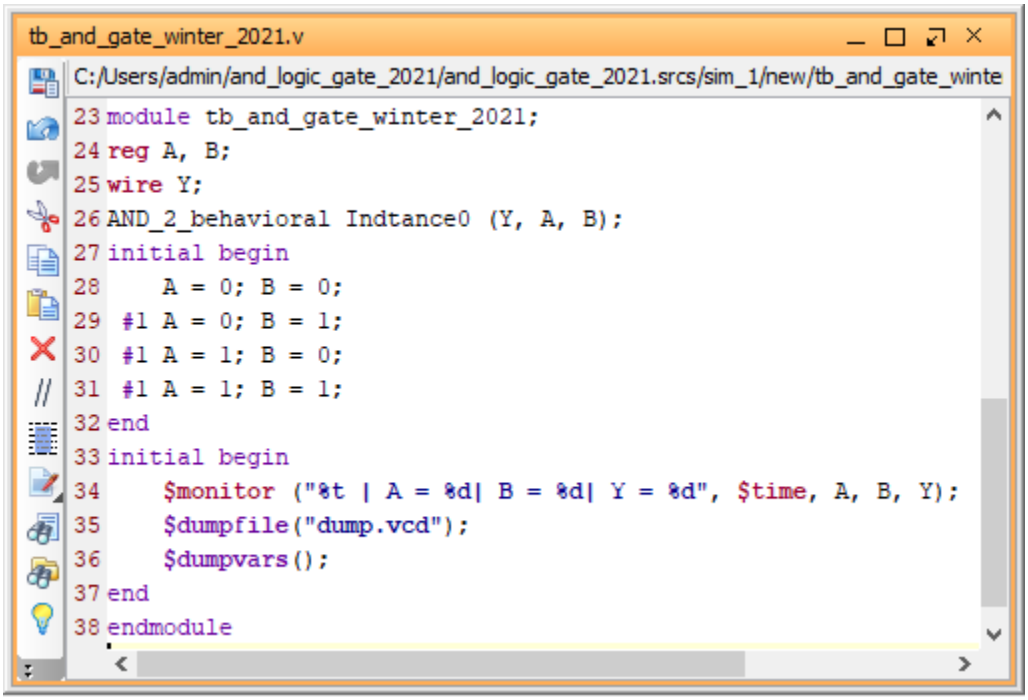
```
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module testbench_XNOR_gate;
24 reg A, B;wire Y;
25 XNOR_2_behavioral Instance0 (Y, A, B);
26 initial begin
27     A = 0; B = 0;
28  #1 A = 0; B = 1;
29  #1 A = 1; B = 0;
30  #1 A = 1; B = 1;
31    end
32 initial begin
33     $monitor ("%t | A = %d| B = %d| Y = %d", $time, A, B, Y);
34     $dumpfile("dump.vcd");
35     $dumpvars();
36 end
37 endmodule
38
```

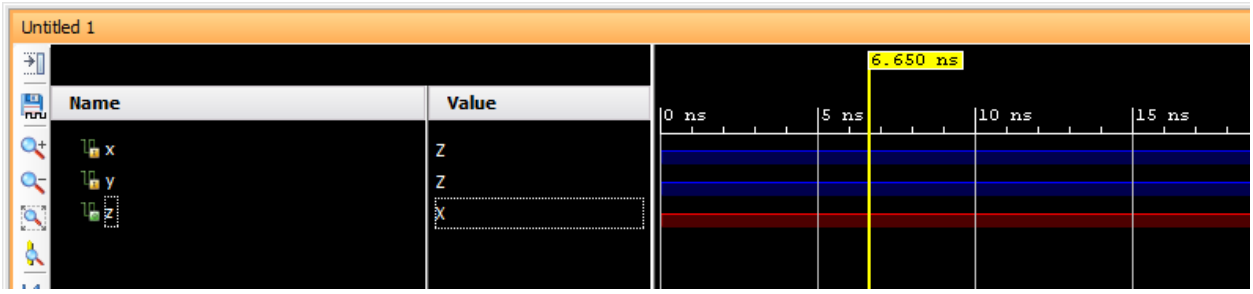


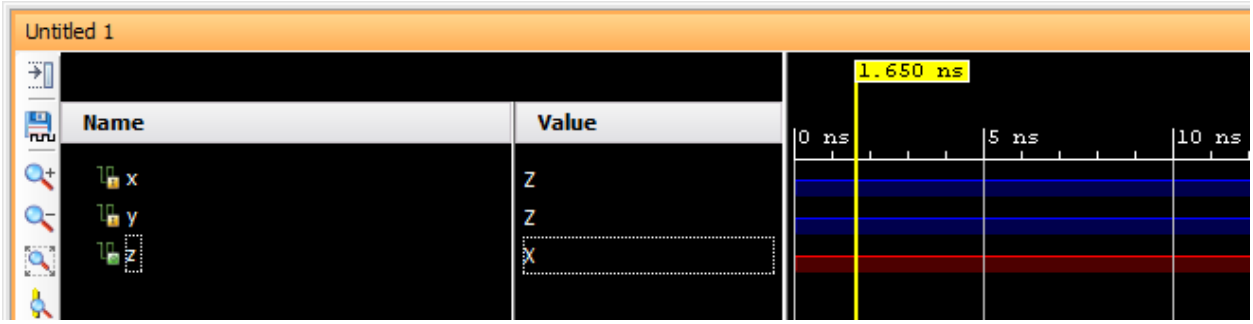
IV] AND Gate :

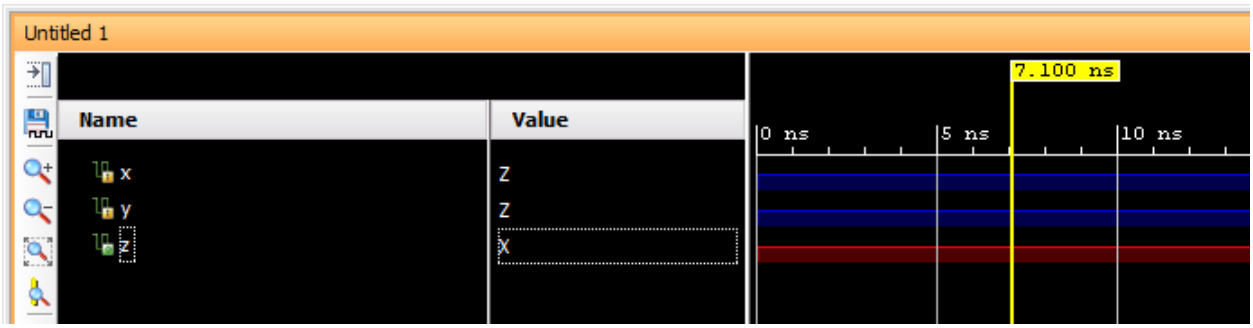
Theory : The AND gate is a basic digital logic gate that implements logical conjunction - it behaves according to the truth table to the right. A HIGH output (1) results only if all the inputs to the AND gate are HIGH (1). If none or not all inputs to the AND gate are HIGH, LOW output results. The function can be extended to any number of inputs.

```
module tb_AND_GATE;
reg a, b
wire c;
AND_GATE_WINTERVITAP2021ST dl(a, b, c);
initial
begin
a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
end
endmodule
```

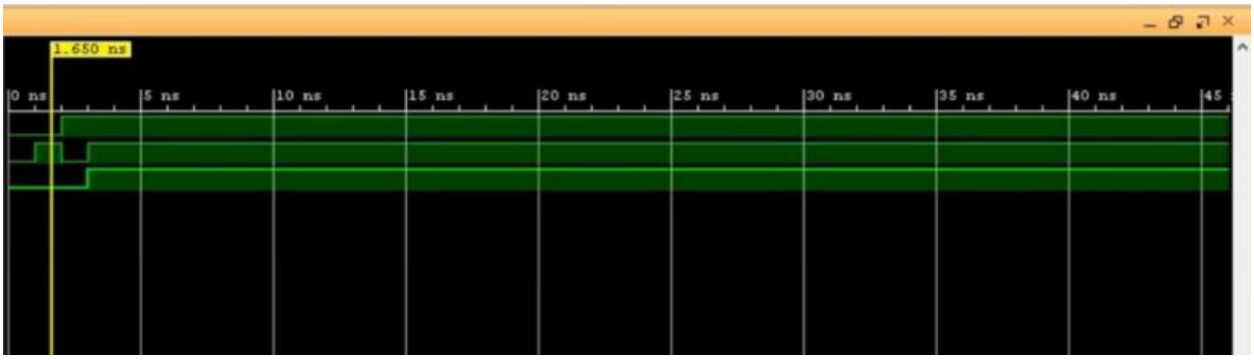


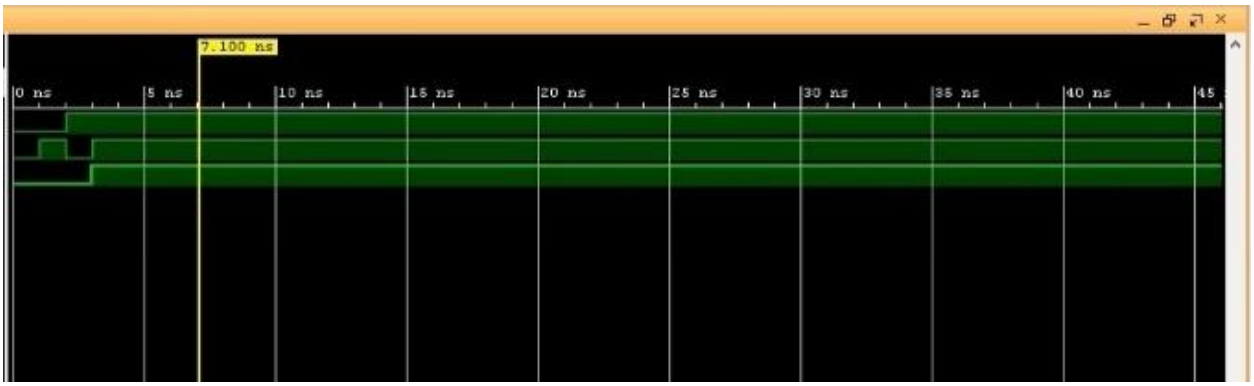

Untitled 1
Name
Value
x
y
z
Z
Z
X
6.650 ns
0 ns
5 ns
10 ns
15 ns


Untitled 1
Name
Value
x
y
z
Z
Z
X
1.650 ns
0 ns
5 ns
10 ns


Untitled 1
Name
Value
x
y
z
Z
Z
X
7.100 ns
0 ns
5 ns
10 ns

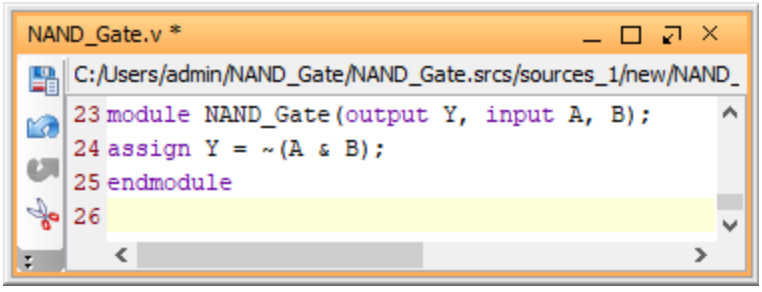

Name
Value
x
y
z
1
1
1
6.650 ns
0 ns
5 ns
10 ns





V] NAND Gate :

Theory : NAND gate (NOT-AND) is a logic gate which produces an output which is false only if all its inputs are true; thus its output is complement to that of an AND gate. A LOW (0) output results only if all the inputs to the gate are HIGH (1); if any input is LOW (0), a HIGH (1) output results.



testbench_NAND_Gate.v *

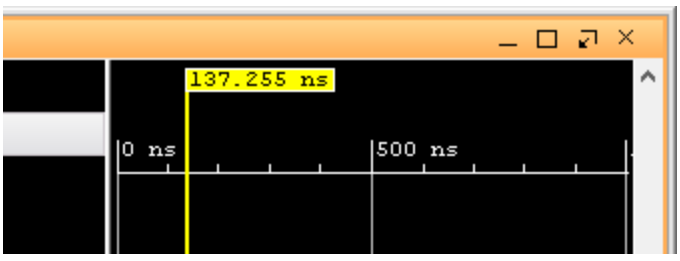
C:/Users/admin/NAND_Gate/NAND_Gate.srcs/sim_1/new/testbench_NAND_Gate.v

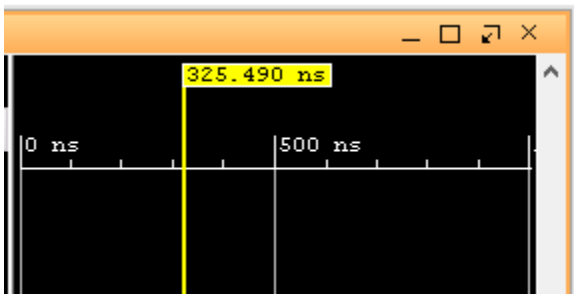
```
22
23 module testbench_NAND_Gate;
24 reg A, B;
25 wire Y;
26 NAND_2_behavioral Indtance0 (Y, A, B);
27 initial begin
28     A = 0; B = 0;
29  #1 A = 0; B = 1;
30  #1 A = 1; B = 0;
31  #1 A = 1; B = 1;
32 end
33 initial begin
34     $monitor ("%t | A = %d| B = %d| Y = %d", $time, A, B, Y);
35     $dumpfile("dump.vcd");
36     $dumpvars();
37 end
38 NAND_Gate
39 endmodule
40
```

NAND_GATE_Behavioural.v

C:/Users/admin/NAND_GATE_Behavioural/NAND_GATE_Behavioural.srcs/sim_1/new/NAND_GATE_Be

```
23 module NAND_GATE_Behavioural(output reg Y, input A, B);
24 always @ (A or B) begin
25     if (A == 1'b1 & B == 1'b1) begin
26         Y = 1'b0;
27     end
28     else
29         Y = 1'b1;
30 end
31 endmodule
32
```





VI] NOT Gate :

NOT_Gate_2021.v *

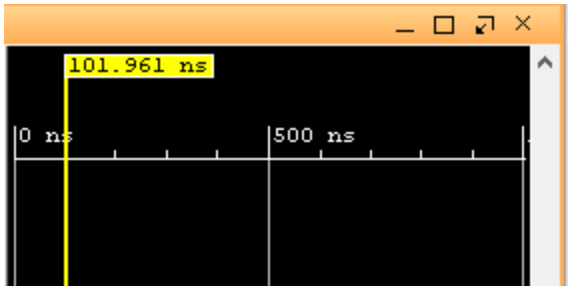
C:/Users/admin/NOT_Gate_2021/NOT_Gate_2021.srcs/sources_1

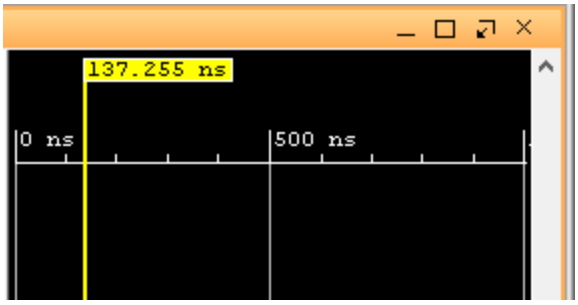
```
23 module NOT_Gate_2021(output Y, input A);
24 assign Y = ~A;
25 endmodule
26
```

testbench_NOT_Gate_2021.v

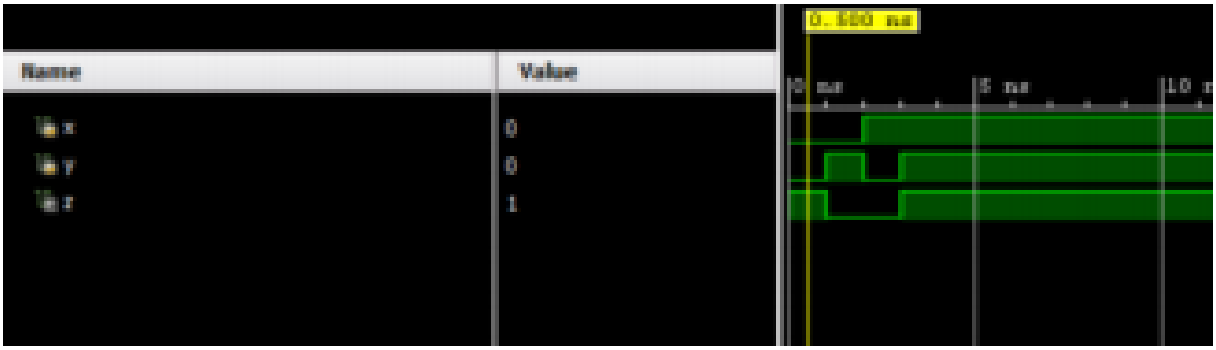
C:/Users/admin/NOT_Gate_2021/NOT_Gate_2021.srcs/sim_1/new/testbench_N

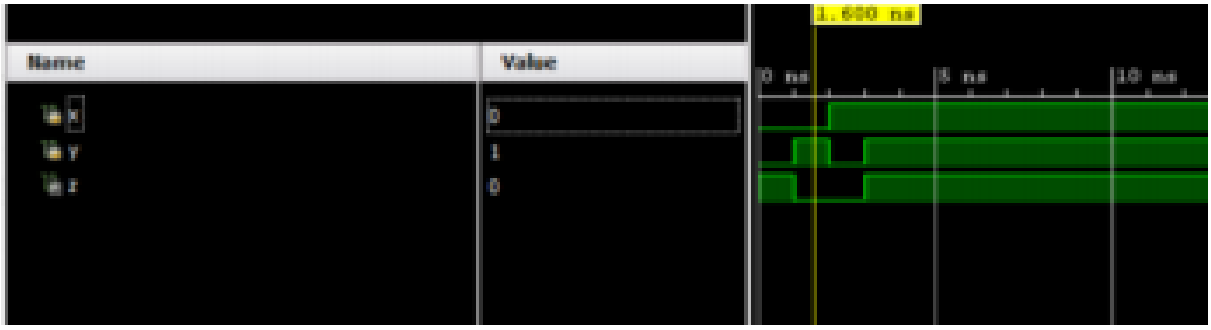
```
23 module testbench_NOT_Gate_2021;
24 reg A;wire Y;
25 NOT_behavioral Instance0 (Y, A);
26 initial begin
27      A = 0;
28   #1 A = 1;
29   #1 A = 0;
30  endinitial begin
31     $monitor ("%t | A = %d| Y = %d", $time, A, Y);
32     $dumpfile("dump.vcd");
33     $dumpvars();
34 end
35 NOT_Gate_2021
36 endmodule
37
```

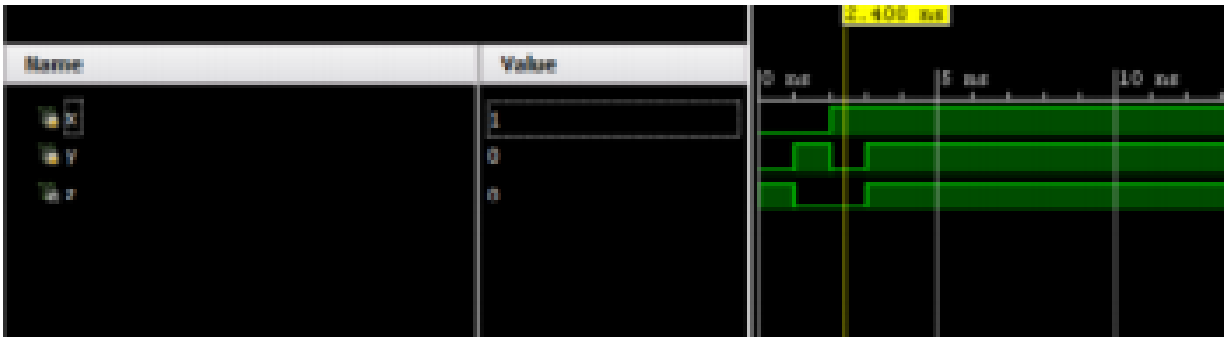


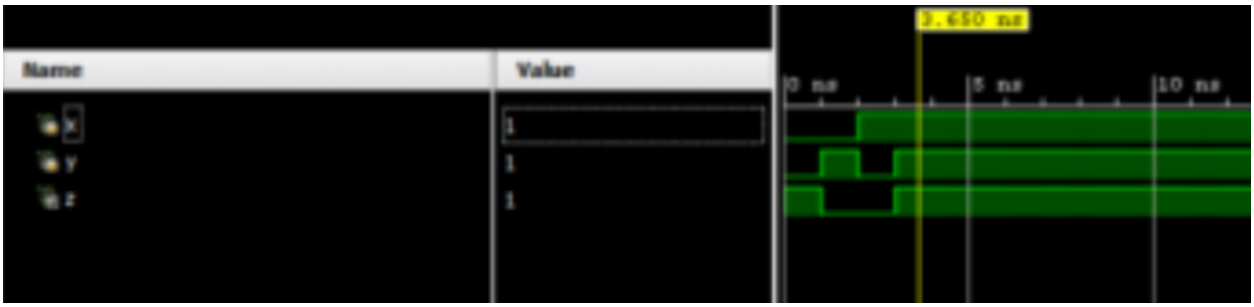


OUTPUT (XNOR gate):

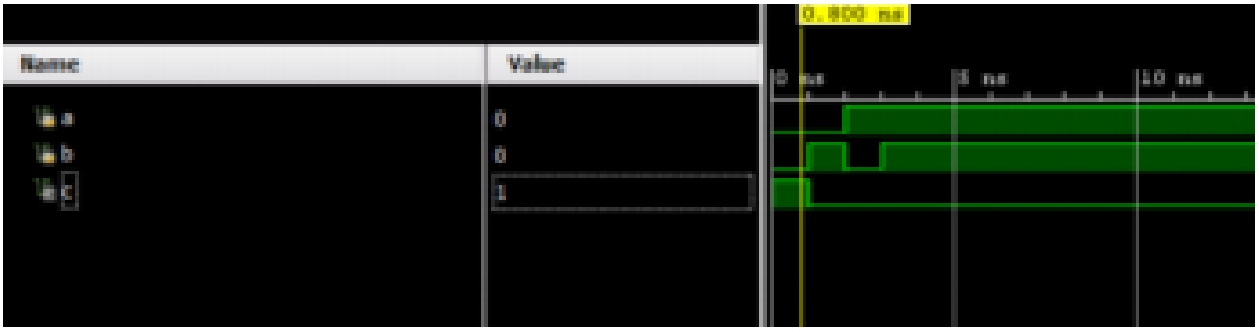


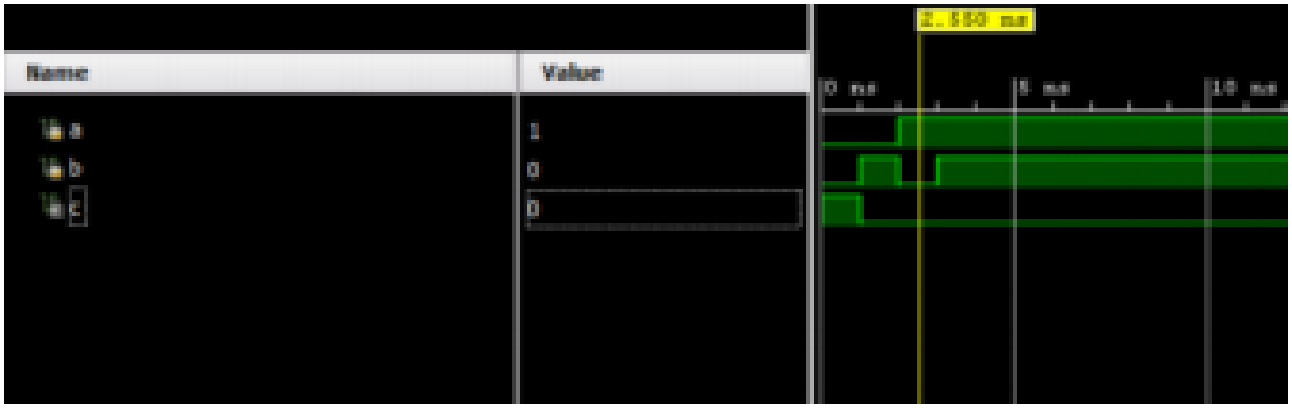




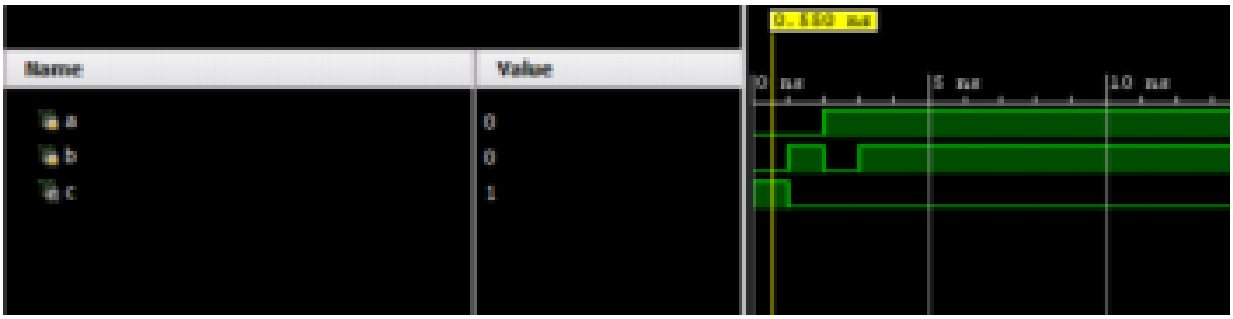


OUTPUT (NOT gate_Testbench, Data Flow, Behavioural) :

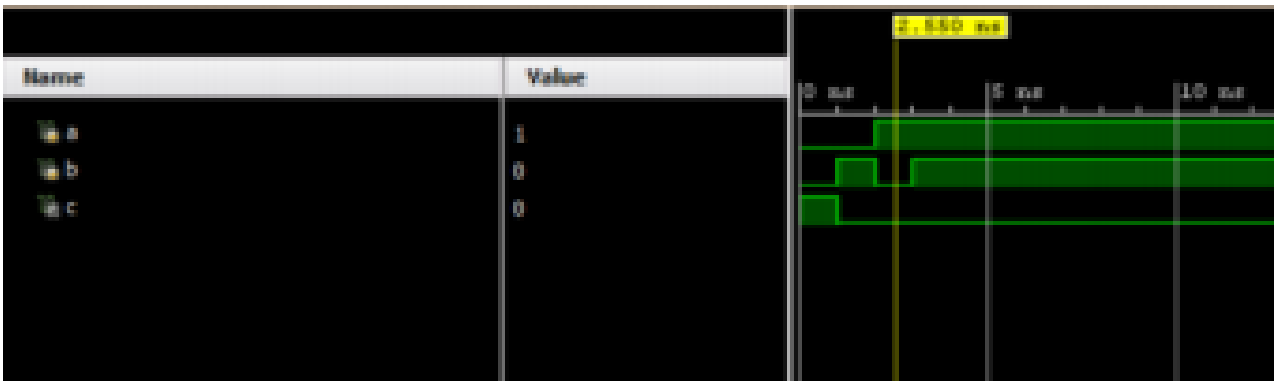



Name
Value
a
b
c
1
0
0
2.550 ns
0 ns
5 ns
10 ns


Name
Value
a
b
c
1
1
0
3.650 ns
0 ns
5 ns
10 ns

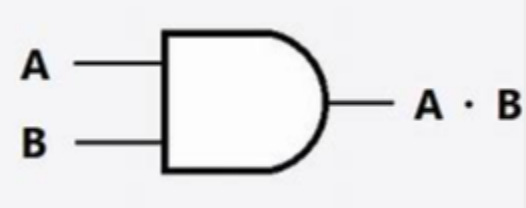

Name
Value
a
b
c
0
0
1
0.550 ns
0 ns
5 ns
10 ns


Name
Value
a
b
c
0
1
0
0 ns
5 ns
10 ns


Name
Value
a
b
c
1
0
0
2,550 ns
0 ns
5 ns
10 ns

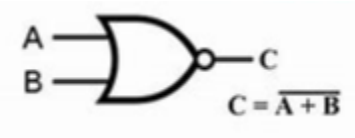

Name
Value
a
b
c
1
1
0
3,600 ns
0 ns
5 ns
10 ns

AND Gate (Circuit diagram):

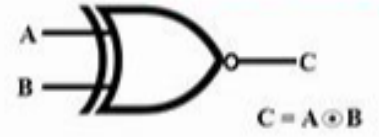


OR Gate (Circuit diagram):

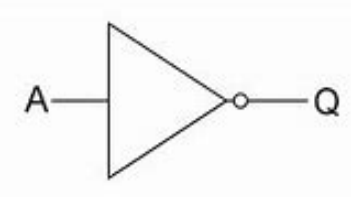
NOR Gate (Circuit diagram):



XNOR Gate (Circuit diagram):

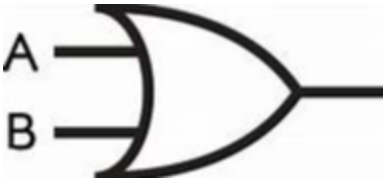


NOT Gate (Circuit diagram):

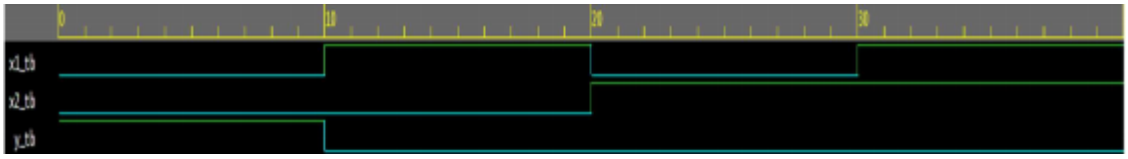


NAND Gate (Circuit diagram):

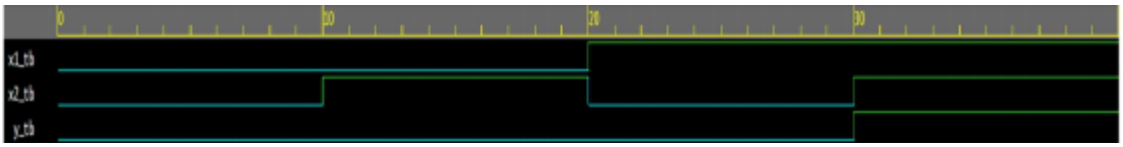
XOR Gate (Circuit diagram):



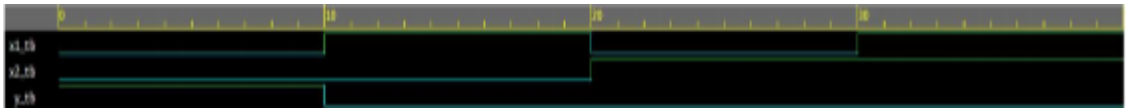
Output for OR gate (Using EDA playground) :



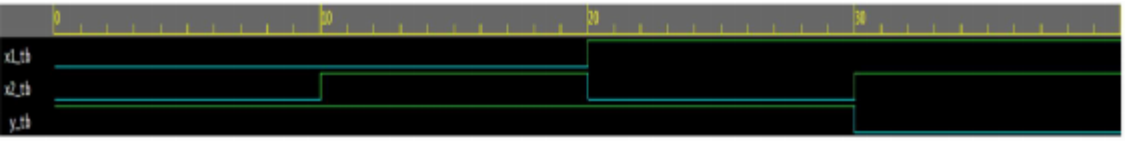
Output for AND gate (Using EDA playground) :



Output for NOR gate (Using EDA playground) :

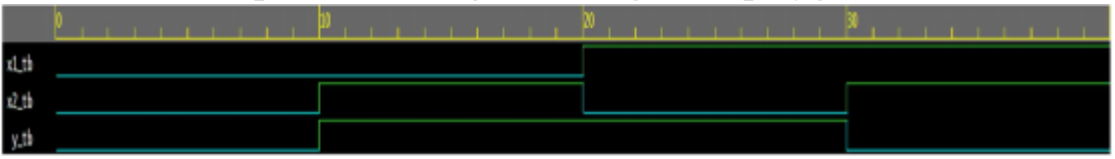


Output for NAND gate (Using EDA playground) :



Output for NOT gate (Using EDA playground) :

Output for XOR gate (Using EDA playground) :



ECE1003_Digital Logic Design_L57+L58_Experiment-2

Name : Khan Mohd. Owais Raza
Registration No. : 20BCD7138

Aim: Design Half Adder, Half Subtractor, Full Adder, Full Subtractor
Software Used : Xilinx Vivado 2014.2

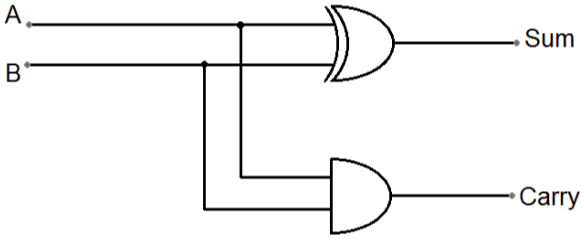
Theory :

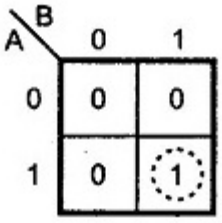
Half Adder :- The Half-Adder is a basic building block of adding two numbers as two inputs and produce out two outputs. The adder is used to perform OR operation of two single bit binary numbers. The **augent** and **addent** bits are two input states, and **'carry'** and **'sum** 'are two output states of the half adder.

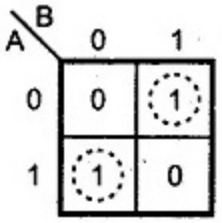
Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1. 'A' and' B' are the input variables. These variables represent the two significant bits which are going to be added
2. 'C_{in}' is the third input which represents the carry. From the previous lower significant position, the carry bit is fetched.
3. The 'Sum' and 'Carry' are the output variables that define the output values.
4. The eight rows under the input variable designate all possible combinations of 0 and 1 that can occur in these variables.

Sum = x' y' z+x' yz+xy' z'+xyz
Carry = xy+xz+yz



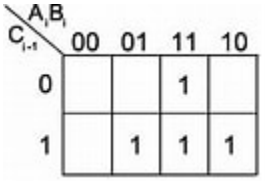


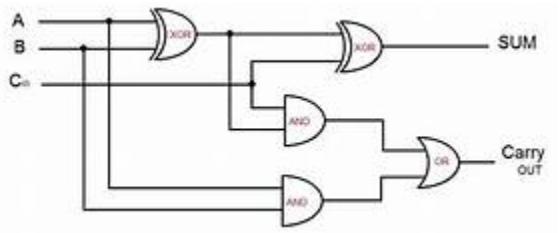


Full Adder :- Full Adder is the adder which adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM.
A full adder logic is designed in such a manner that can take eight inputs together to create a byte-wide adder and cascade the carry bit from one adder to the another.

Logical Expression for SUM:
= A' B' (C-IN) + A' B (C-IN)' + A B' (C-IN)' + A B (C-IN)
= (C-IN) (A' B' + A B) + (C-IN)' (A' B + A B')
= (C-IN) XOR (A XOR B)

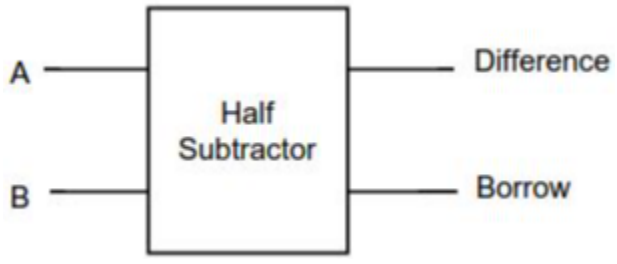
Logical Expression for C-OUT**:**
A' B (C-IN) + A B' (C-IN) + A B (C-IN)' + A B (C-IN)
= A B + B (C-IN) + A (C-IN)





Half Subtractor :- The half subtractor is also a building block for subtracting two binary numbers. It has two inputs and two outputs. This circuit is used to subtract two single bit binary numbers A and B. The '**diff**' and '**borrow**' are two output states of the half subtractor.

The SOP form of the Diff and Borrow is as follows:
Diff= A'B+AB'
Borrow = A'B



Inputs		Outputs	
A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

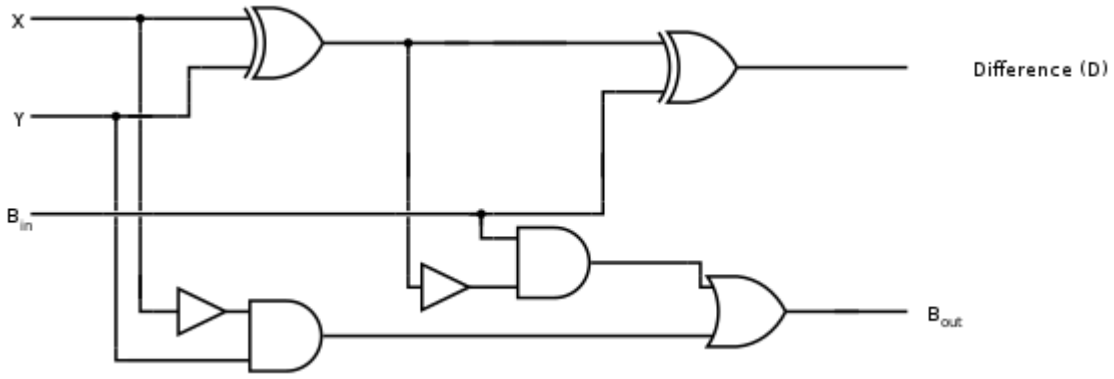
- A' and 'B' are the input variables whose values are going to be subtracted.
- The 'Diff' and 'Borrow' are the variables whose values define the subtraction result, i.e., difference and borrow.
- The first two rows and the last row, the difference is 1, but the 'Borrow' variable is 0.
- The third row is different from the remaining one. When we subtract the bit 1 from the bit 0, the borrow bit is produced.

Half Subtractor :- A full subtractor is a combinational circuit that performs subtraction of two bits, one is minuend and other is subtrahend, taking into account borrow of the previous adjacent lower minuend bit. This circuit has three inputs and two outputs. The three inputs A, B and Bin, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and Bout represent the difference and output borrow, respectively.

D = A'B'Bin + A'BBin' + AB'Bin' + ABBin
= Bin(A'B' + AB) + Bin'(AB' + A'B)
= Bin(A XNOR B) + Bin'(A XOR B)
= Bin (A XOR B)' + Bin'(A XOR B)
= Bin XOR (A XOR B)
= (A XOR B) XOR Bin

Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
= A'B'Bin +A'BBin' + A'BBin + A'BBin + A'BBin + ABBin
= A'Bin(B + B') + A'B(Bin + Bin') + BBin(A + A')
= A'Bin + A'B + BBin

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



Name : Khan Mohd. Owais Raza
Registration No. : 20BCD7138

Aim: Design Half Adder, Half Subtractor, Full Adder, Full Subtractor
Software Used : Xilinx Vivado 2014.2

HALF ADDER :

Theory : The addition of 2 bits is done using a combination circuit called Half adder. The input variables are augend and addend bits and output variables are sum & carry bits. A and B are the two input bits.

Truth Table:-

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Input:-

```
module HalfAdder_ECE1003_DLD(a, b, sum, carry);
input a, b;
output sum, carry;
reg sum, carry;
always @ (a,b)
begin
if (a==1'b0 && b==1'b0)
begin
sum=0;
carry=0;
end
else if ((a==1'b0 && b==1'b1) || (a==1'b1 && b==1'b0))
begin
sum=1'b0;
carry=1'b1;
end
endmodule
```

HalfAdder_ECE1003_Lab.v

C:/Users/admin/HalfAdder_ECE1003_Lab/HalfAdder_ECE1003_Lab.srcs/sources_1/new/Ha

```
23 module HalfAdder_ECE1003_Lab(i_bit1,i_bit2,o_sum,o_carry);
24 nput  i_bit1;
25   input  i_bit2;
26   output o_sum;
27   output o_carry;
28
29   assign o_sum   = i_bit1 ^ i_bit2;
30   assign o_carry = i_bit1 & i_bit2;
31
32 endmodule
```

Testbench :

testbench_HalfAdder_ECE1003.v

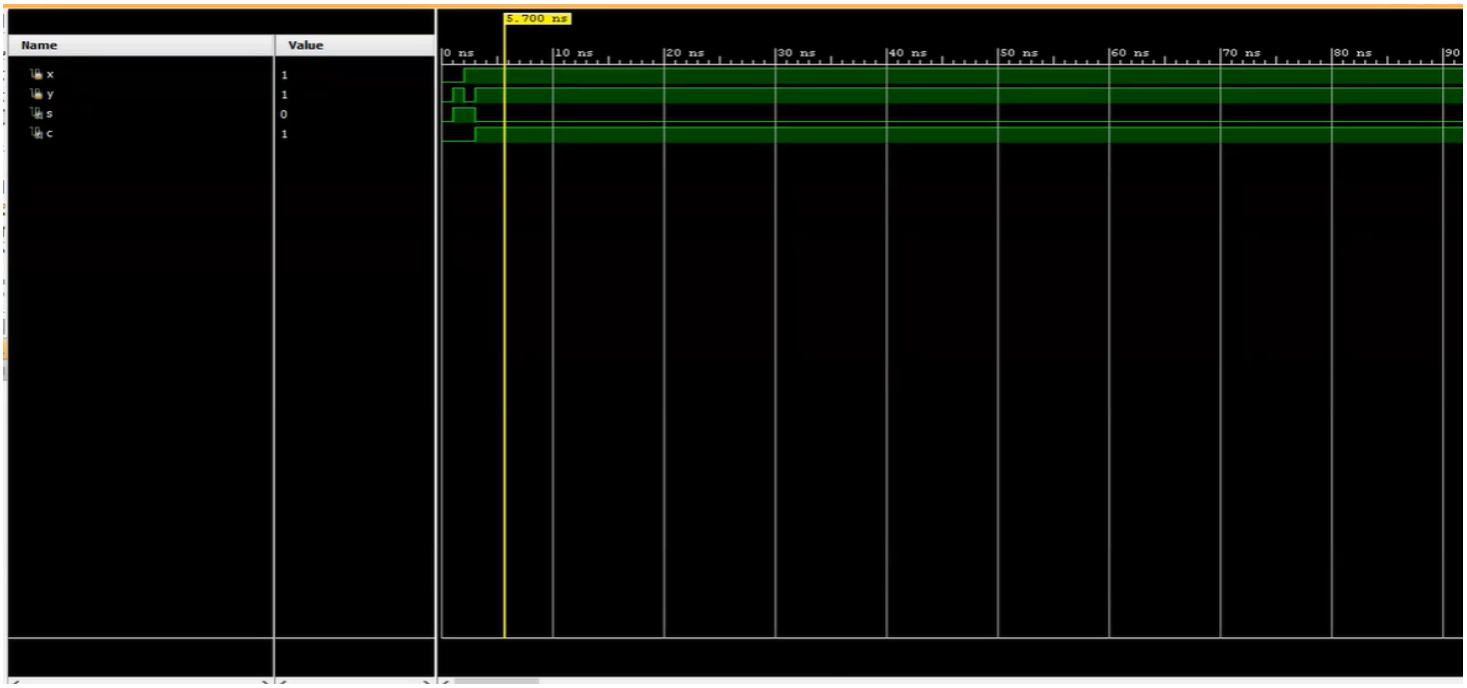
C:/Users/admin/HalfAdder_ECE1003_Lab/HalfAdder_ECE1

```
23 module testbench_HalfAdder_ECE1003;
24
25   reg r_bit1 = 0;
26   reg r_bit2 = 0;
27   wire w_sum;
28   wire w_carry;
29
30   half_adder half_adder_inst
31     (
32      .i_bit1(r_BIT1),
33      .i_bit2(r_BIT2),
34      .o_sum(w_SUM),
35      .o_carry(w_CARRY)
36      );
37
38   initial
39     begin
40       r_bit1 = 1'b0;
41       r_bit2 = 1'b0;
42       #10;
43       r_bit1 = 1'b0;
44       r_bit2 = 1'b1;
45       #10;
46       r_bit1 = 1'b1;
47       r_bit2 = 1'b0;
48       #10;
49       r_bit1 = 1'b1;
50       r_bit2 = 1'b1;
51       #10;
52     end
53 endmodule
```

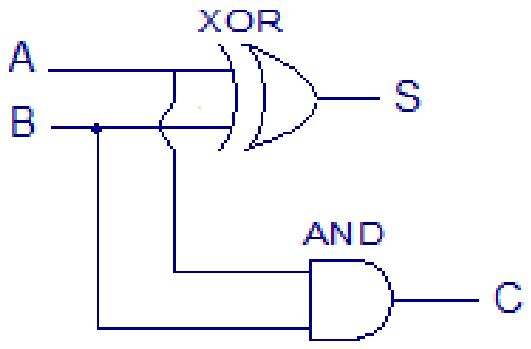
Output :-


Name
Value
x
y
s
c
0
0
0
0
0.550 ns
0 ns
5 ns
10 ns


Name
Value
x
y
s
c
0
1
1
0
1.650 ns
0 ns
5 ns
10 ns
15 ns
20 ns
25 ns
30 ns
35 ns
40 ns



Circuit diagram for half adder :



FULL ADDER :

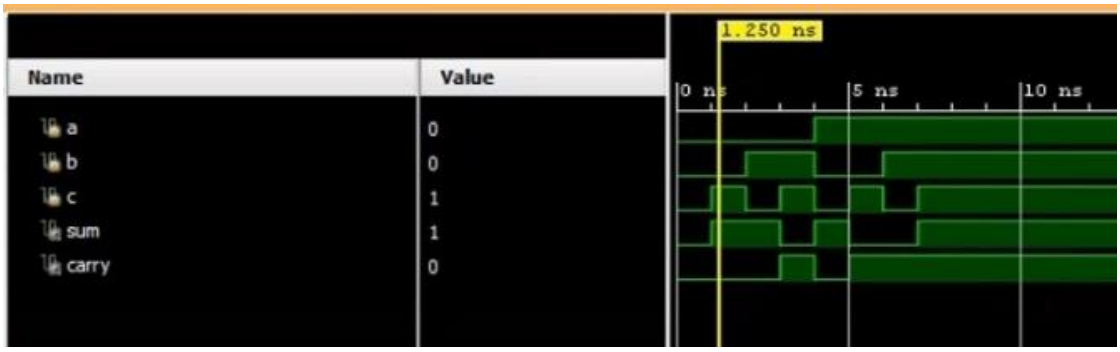
Theory : A full adder gives the number of 1s in the input in binary representation**.** A full adder adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers, often written as A, B, and Cin; A and B are the operands, and Cin is a bit carried in from the previous less-significant stage.

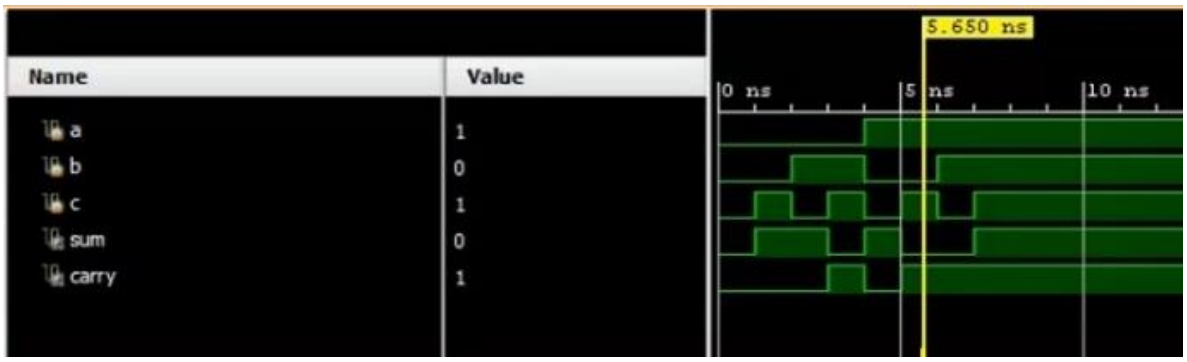
a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
module FullAdder_ECE1003_DLD(a, b, sum, carry);
input a, b, c;
output sum, carry;
assign sum=(a^b)^c;
assign carry= ((a & b)|(b & c)|(a & c))
endmodule


module testbench_FullAdder_ECE1003;
reg a, b, c;
wire sum, carry;
FullAdder_ECE1003 bl(a, b, c, sum, carry);
initial
begin
a= 1'b0; b=1'b0; c=1'b0;
#1 a= 1'b0; b=1'b0; c=1'b1;
#1 a= 1'b0; b=1'b1; c=1'b0;
#1 a= 1'b0; b=1'b1; c=1'b1;
#1 a= 1'b1; b=1'b0; c=1'b0;
#1 a= 1'b1; b=1'b0; c=1'b1;
```

```
#1 a= 1'b1; b=1'b1; c=1'b0;
#1 a= 1'b1; b=1'b1; c=1'b1;
end
endmodule
```

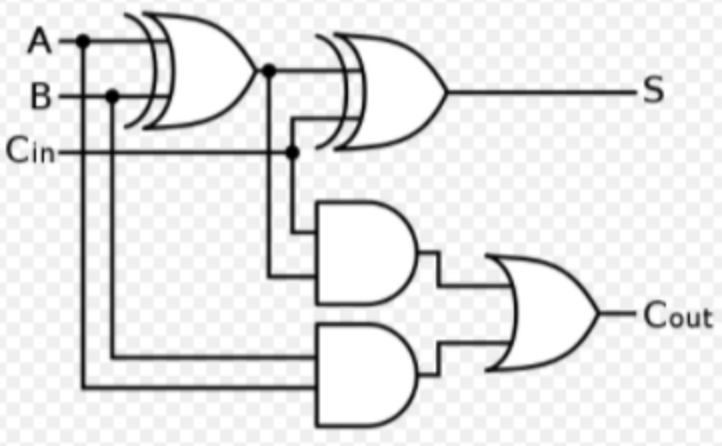
Output :-




5.650 ns
Name
Value
0 ns
5 ns
10 ns
a
1
b
0
c
1
sum
0
carry
1


6.500 ns
Name
Value
0 ns
5 ns
10 ns
a
1
b
1
c
0
sum
0
carry
1

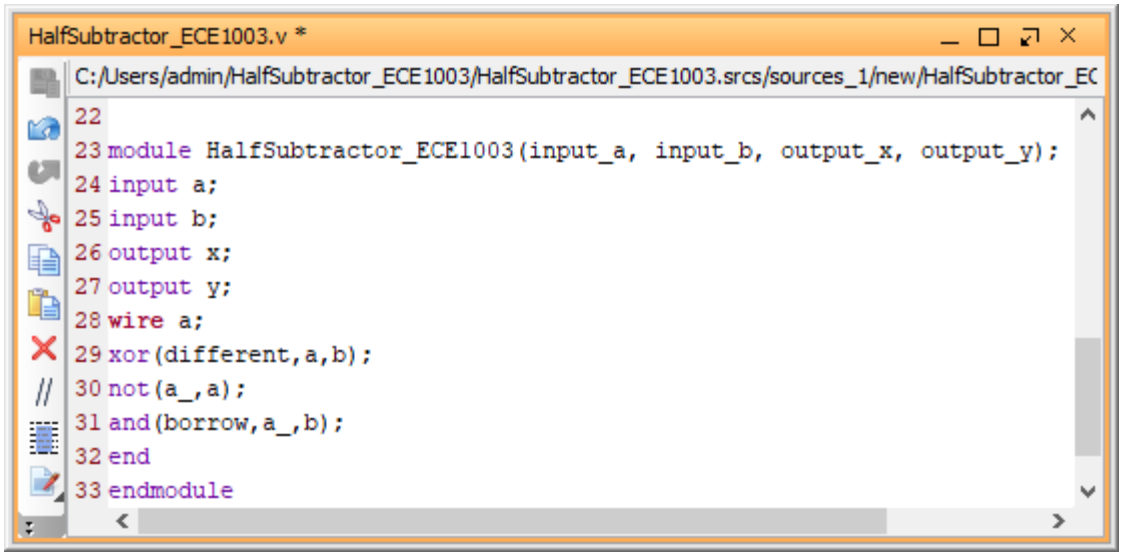

7.650 ns
Name
Value
0 ns
5 ns
10 ns
15 ns
a
1
c
1
sum
1
carry
1



HALF SUBTRACTOR :

Theory : The Half Subtractor is digital circuit which processes the subtraction of two 1-bit numbers**.** In this, the two numbers involved are termed as subtrahend and minuend.

a	**b**	**x**	**y**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



Testbench for Half Subtractor :

testbench_HalfSubtractor_ECE1003.v *

C:/Users/admin/HalfSubtractor_ECE1003/HalfSubtractor_ECE1003.srcs/sim_1/new/testbench_HalfSubtractor

```
23 module testbench_HalfSubtractor_ECE1003;
24 wire D, B;
25 reg X, Y;
26 Half_Subtractor_2 Instance0 (D, B, X, Y);
27 initial begin
28     X = 0; Y = 0;
29 #1  X = 0; Y = 1;
30 #1  X = 1; Y = 0;
31 #1  X = 1; Y = 1;
32 end
33 initial begin
34     $monitor ("%t, X = %d| Y = %d| B = %d| D = %d", $time, X, Y, B, D);
35     $dumpfile("dump.vcd");
36     $dumpvars();
37 end
38 endmodule
```

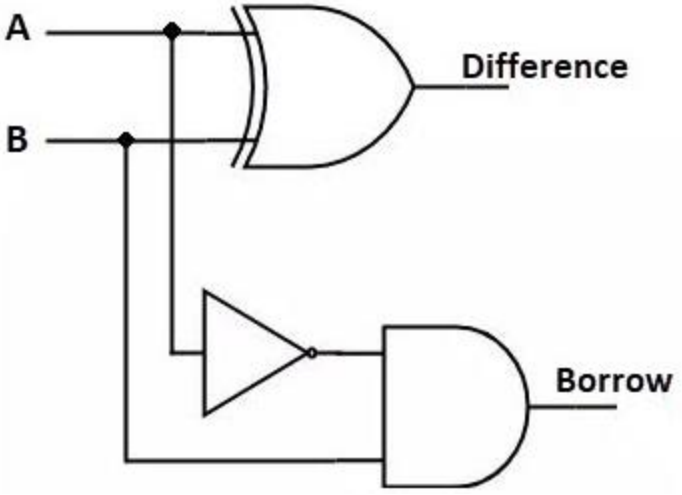
Data flow modelling :

HalfSubtractor_ECE1003.v

C:/Users/admin/HalfSubtractor_ECE1003/HalfSubtractor_ECE1003.srcs/sources_1/new/HalfSubtractor_ECE1003.v

```
22
23 module HalfSubtractor_ECE1003(input_a, input_b, output_diff, output_borrow);
24 output diff ;
25 output borrow ;
26
27 input a ;
28 input b ;
29
30 assign diff = a ^ b;
31 assign borrow = (~a) & b;
32
33 endmodule
```

Circuit diagram :



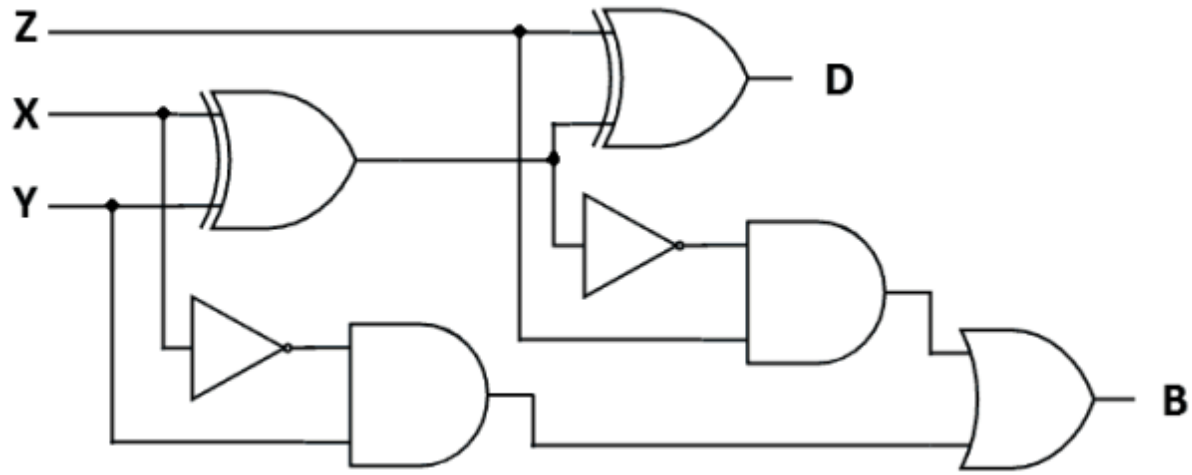
FULL SUBTRACTOR :

a	b	c	x	y
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

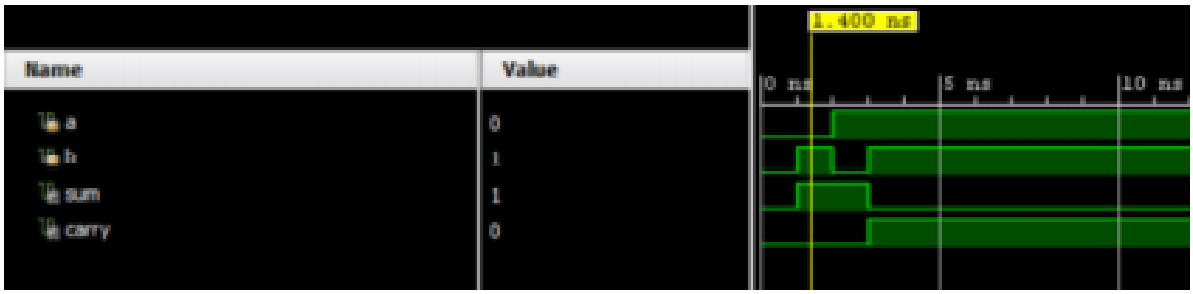
```
module ECE1003_FullSubtractor(a, b, c, x, y);
input a;
input b;
input c;
output x;
output y;
wire d,e,f;
xor(difference,a,b,c);
and(d,~a,b);
and(e,b,c);
and(f,~a,c);
or(borrow,d,e,f);
endmodule
```

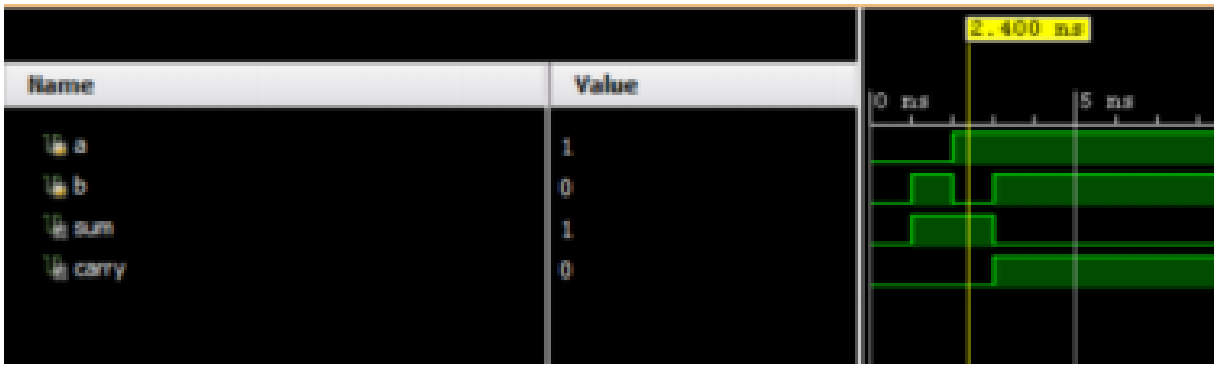
```
module testbench_FullSubtractor_ECE1003;
reg a;
reg b;
reg c;
wire borrow;
wire difference;
initial
begin
#10 a=1'b0;b=1'b0;c=1'b0;
#10 a=1'b0;b=1'b0;c=1'b1;
#10 a=1'b0;b=1'b1;c=1'b0;
#10 a=1'b0;b=1'b1;c=1'b1;
#10 a=1'b1;b=1'b0;c=1'b0;
#10 a=1'b1;b=1'b0;c=1'b1;
#10 a=1'b1;b=1'b1;c=1'b0;
#10 a=1'b1;b=1'b1;c=1'b1;
end
endmodule
```

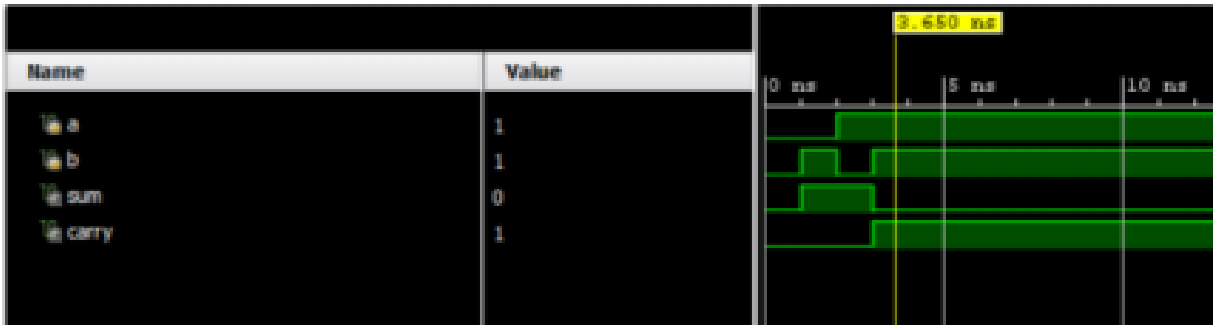
Circuit diagram for full subtractor :

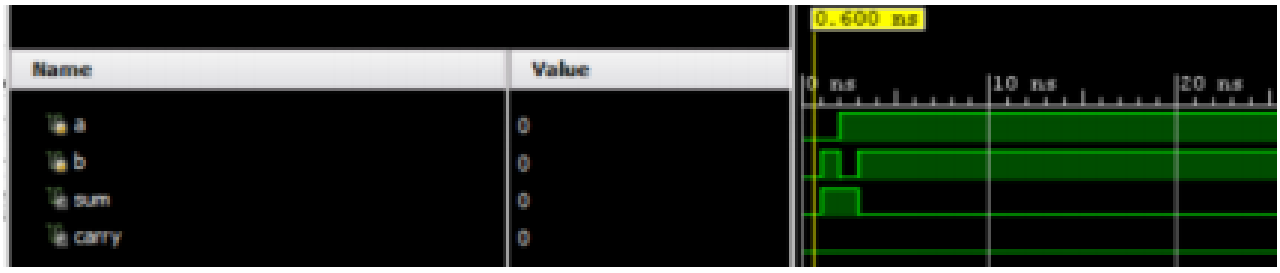


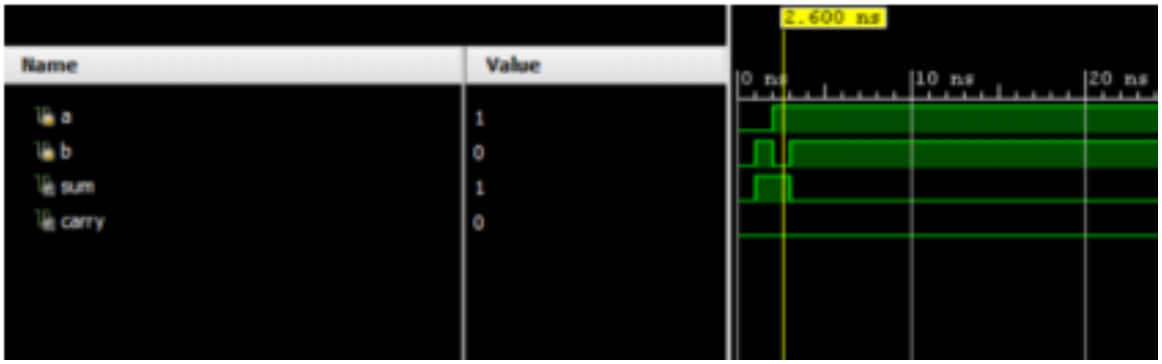
Output (Half Adder) :

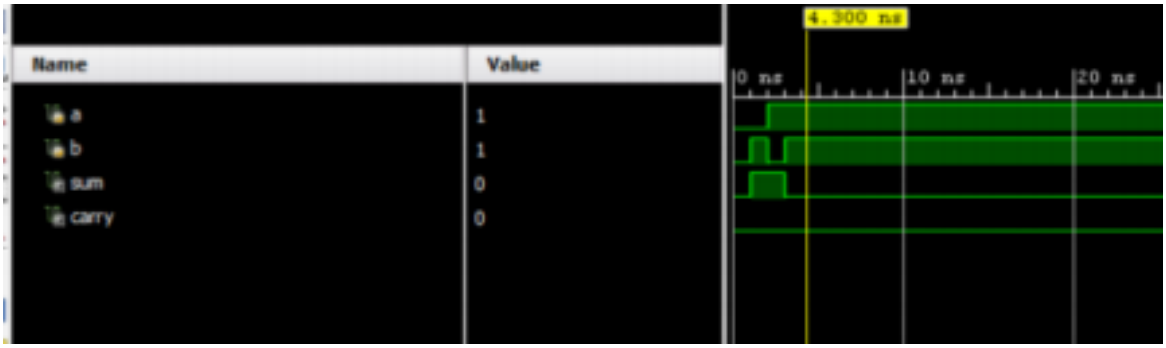


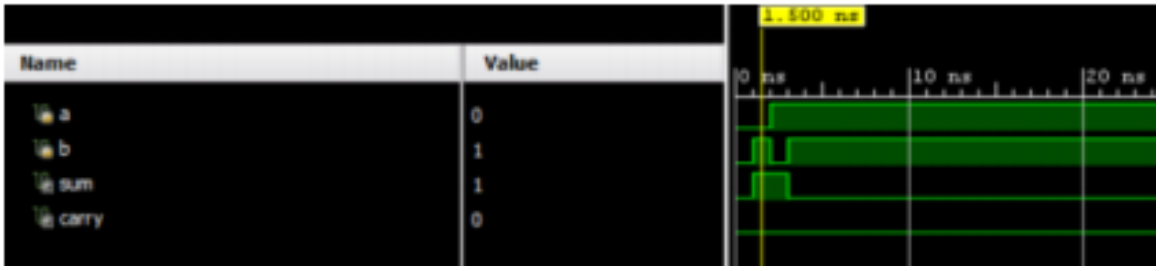




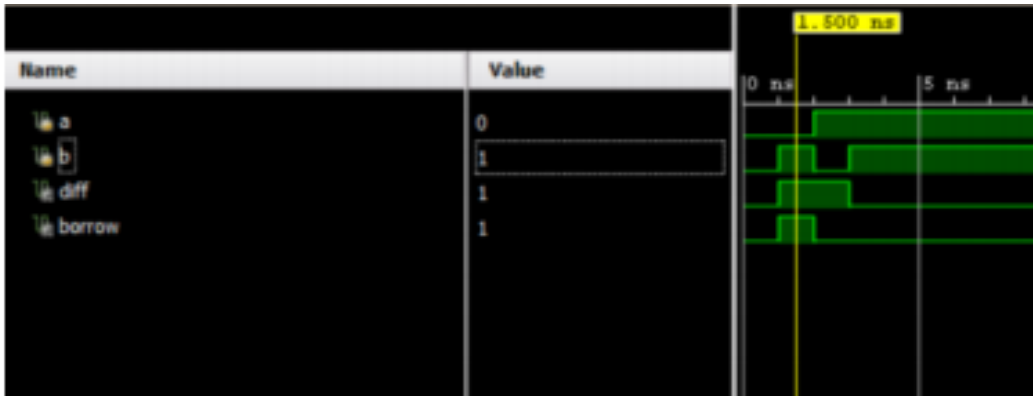


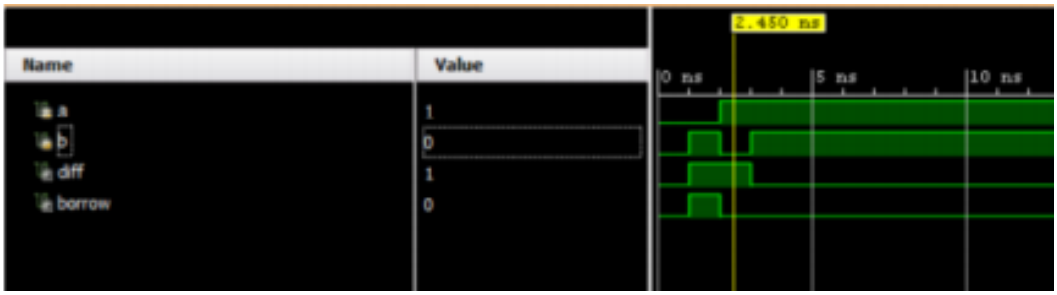

Name
Value
a
b
sum
carry
1
0
1
0
2,600 ns
0 ns
10 ns
20 ns

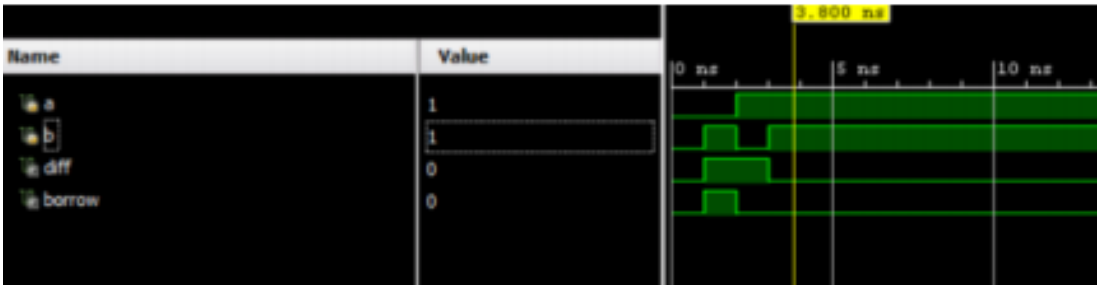

Name
Value
a
b
sum
carry
1
1
0
0
4,300 ns
0 ns
10 ns
20 ns


Name
Value
a
b
sum
carry
0
1
1
0
1,500 ns
0 ns
10 ns
20 ns

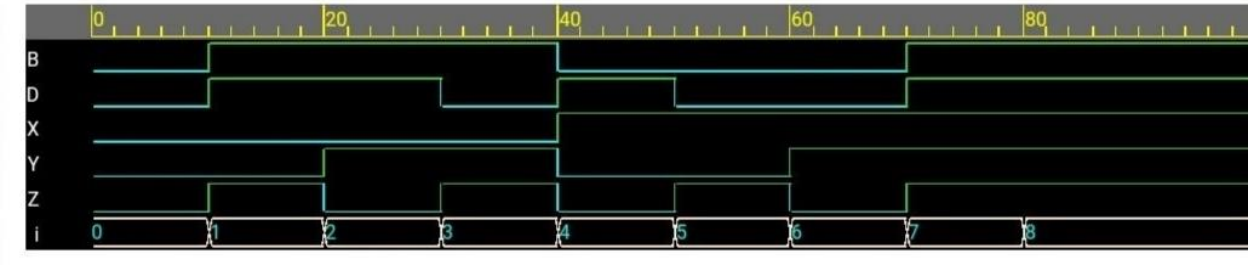
Output (Half Subtractor):

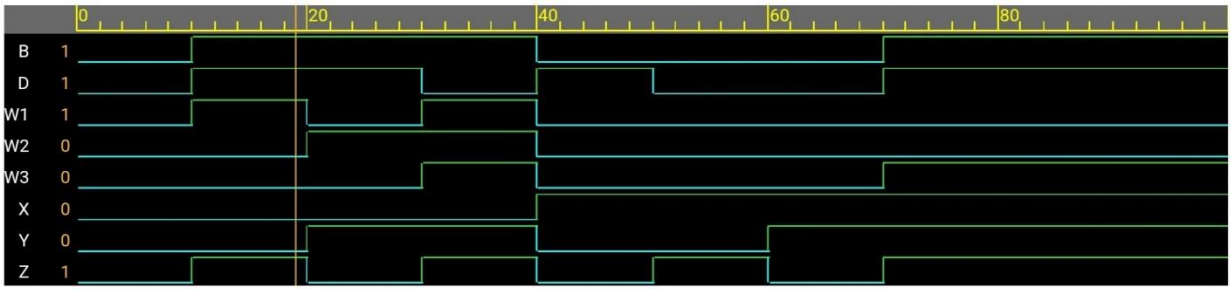






Output Full Subtractor (Using EDA playground) :





ECE1003 Experiment No. 3 **21/03/2021**

NAME : KHAN MOHD. OWAIS RAZA
REGISTRATION NO. : 20BCD7138

Aim: Designing binary adders and subtractors
Software used : Xilinx Vivado 2014.2

(I) 4-Bit Binary Adder:

Structural :-

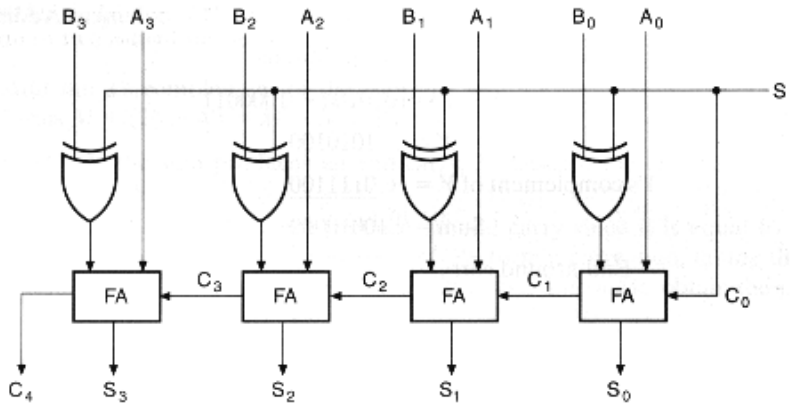
```
module BinaryAdder_ECE1003(a, b, cin, sum, cout);
input [3:0]a, b;
input cin;
output [3:0]sum;
output cout;
wire c0,c1,c2;
FullAdder_ ECE1003 fa1(a[0], b[0], cin, s[0], c0);
FullAdder_ ECE1003 fa2(a[1], b[1], c0, s[1], c1);
FullAdder_ ECE1003 fa3(a[2], b[2], c1, s[2], c2);
FullAdder_ ECE1003 fa4(a[3], b[3], c3, s[3],cout);
endmodule


module testbench_BinaryAdder_ECE1003;
reg [3:0]a, b;
input cin;
output [3:0]sum;
output cout;
wire c0,c1,c2;
FullAdder_ ECE1003 fa1(a[0],b[0],cin,sum[0],c0);
FullAdder_ ECE1003 fa2(a[1],b[1],c0,sum[1],c1);
FullAdder_ ECE1003 fa3(a[2],b[2],c1,sum[2],c2);
FullAdder_ ECE1003 fa4(a[3],b[3],c2,sum[3],cout);
endmodule
```

```
module testbench_BinaryAdder_ECE1003;
reg [3:0]a, b;
reg cin;
wire [3:0]sum;
wire cout;
BinaryAdder_ECE1003 a1(a, b, sum, cout);
initial
begin
a=4'b0000; b=4'b0000; cin=1'b0;
#10 a=4'b0001; b=4'b0001; cin=1'b0;
#10 a=4'b0010; b=4'b0010; cin=1'b0;
#10 a=4'b0011; b=4'b0011; cin=1'b0;
#10 a=4'b0101; b=4'b0101; cin=1'b0;
#10 a=4'b1111; b=4'b1111; cin=1'b0;
end
endmodule
```

Behavioural :-

```
module FullAdder_ ECE1003(a,b,cin,sum,cout);
input [3:0]a,b;
input cin;
output wire [3:0]sum;
output wire cout;
wire [4:0]temp;
assign temp=a+b+cin;
assign sum=temp[3:0];
assign cout=temp[4];
endmodule
```

Circuit Diagram for 4-Bit binary adder:



Truth table for 4-Bit binary adder:

a	b	cin	sum	cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

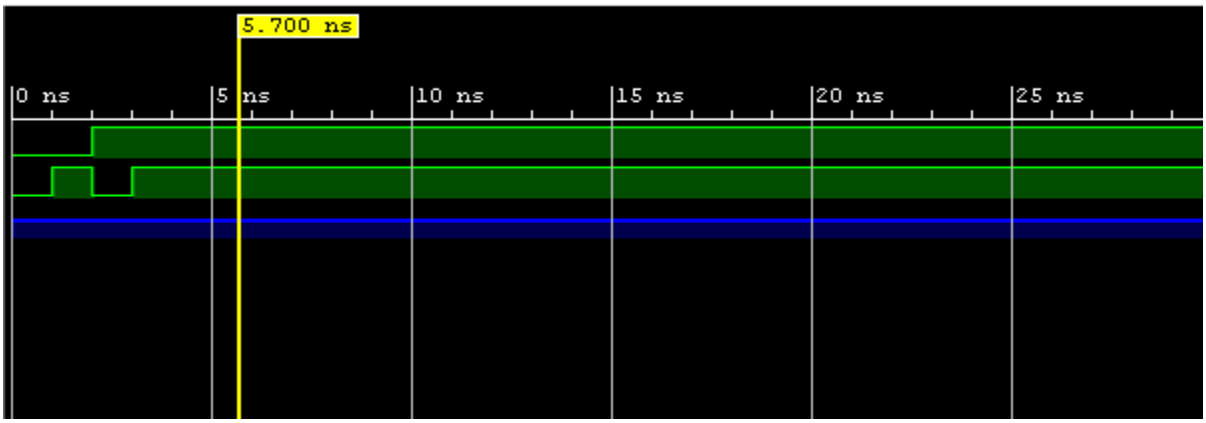
(II) 4-Bit Binary Subtractor:

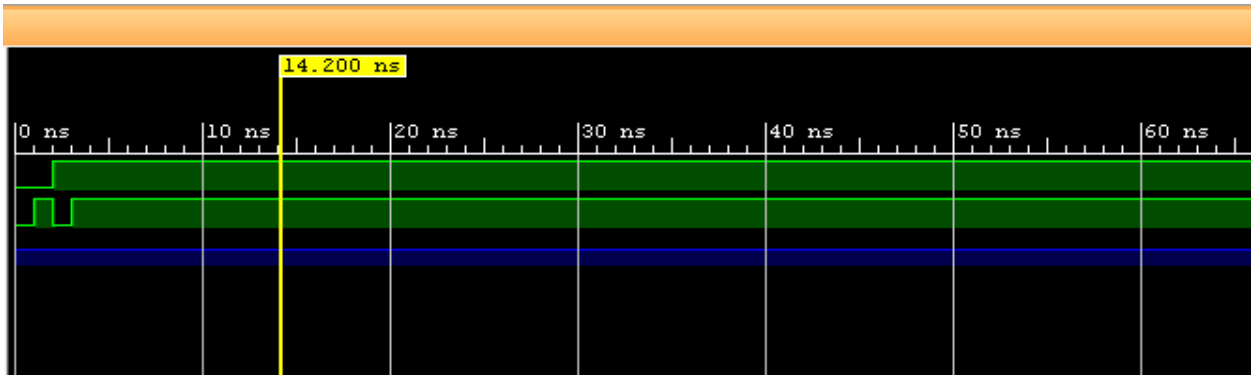
A	B	Difference	Borrow
0	**0**	**0**	**0**
0	**1**	**1**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**0**

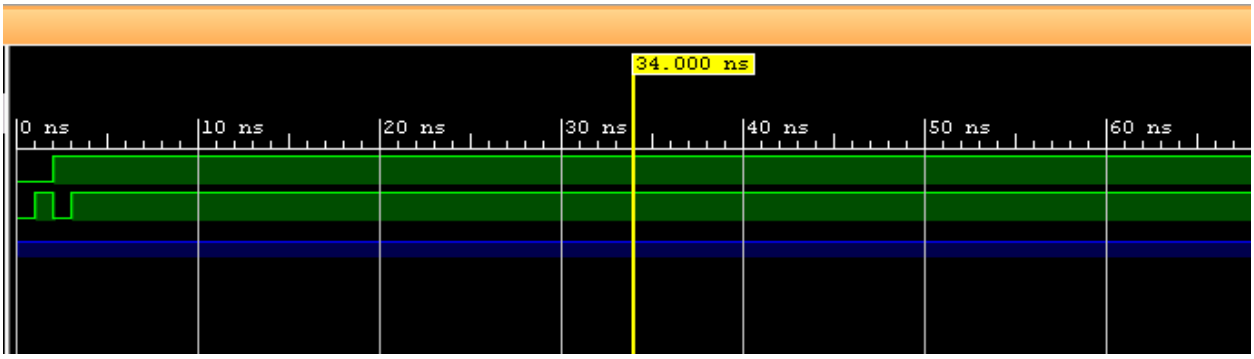
A	B	Bin	Difference	Borrow
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**1**
0	**1**	**0**	**1**	**1**
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**0**
1	**1**	**0**	**0**	**0**
1	**1**	**1**	**1**	**1**

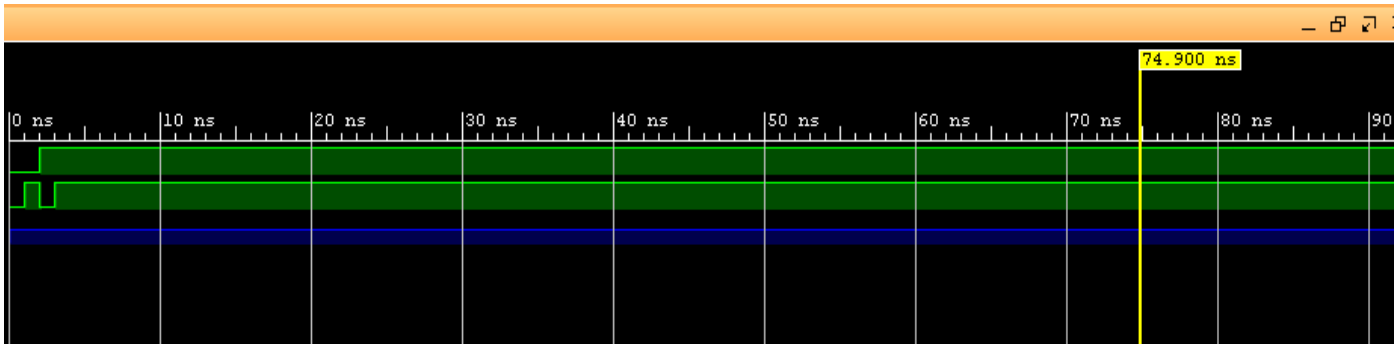
```
module BinarySubtractor_ECE1003(a, b, cin, sum, cout);
input [3:0]a, b;
input cin;
output [3:0]sum;
output cout;
wire c0,c1,c2;
FullSubtractor_ECE1003 fa1(a[0],~b[0], cin, s[0], c0);
FullSubtractor_ECE1003 fa2(a[1],~b[1], c0, s[1], c1);
FullSubtractor_ECE1003 fa3(a[2],~b[2], c1, s[2], c2);
FullSubtractor_ECE1003 fa4(a[3],~b[3], c3, s[3],cout);
endmodule


module testbench_ FullSubtractor_ECE1003;
reg [3:0]a, b;
input cin;
output [3:0]sum;
output cout;
wire c0,c1,c2;
FullSubtractor_ECE1003 fa1(a[0],~b[0],cin,sum[0],c0);
FullSubtractor_ECE1003 fa2(a[1],~b[1],c0,sum[1],c1);
FullSubtractor_ECE1003 fa3(a[2],~b[2],c1,sum[2],c2);
FullSubtractor_ECE1003 fa4(a[3],~b[3],c2,sum[3],cout);
endmodule
```

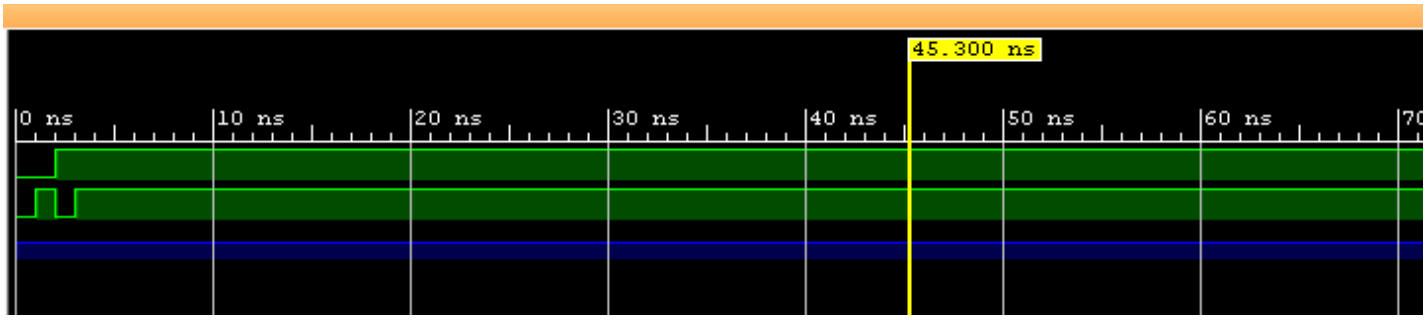
Output (For 4 bit binary adder) :



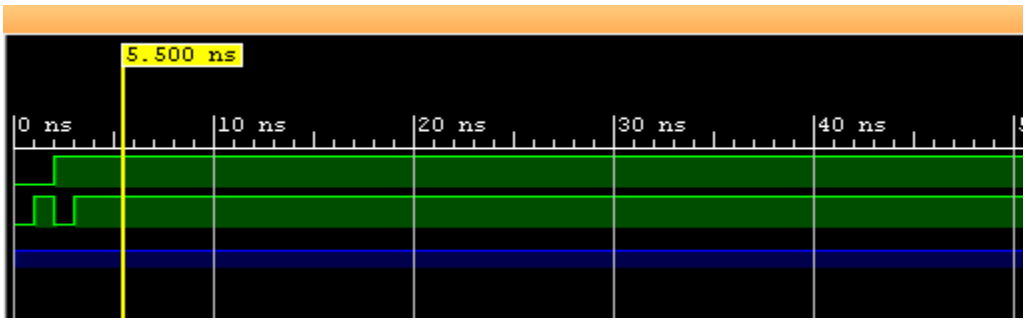


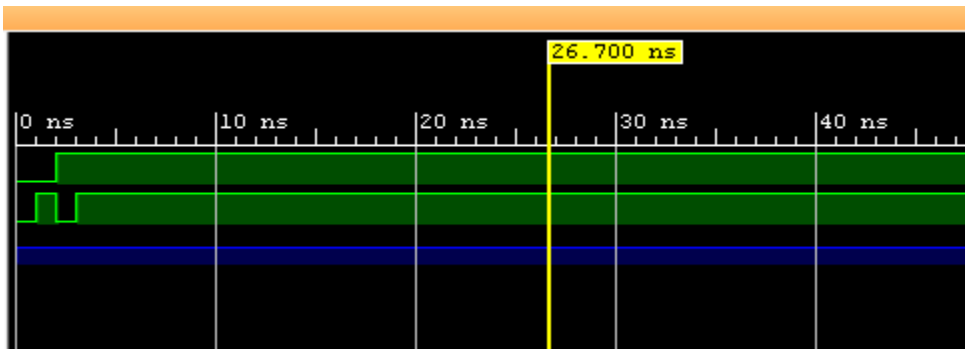


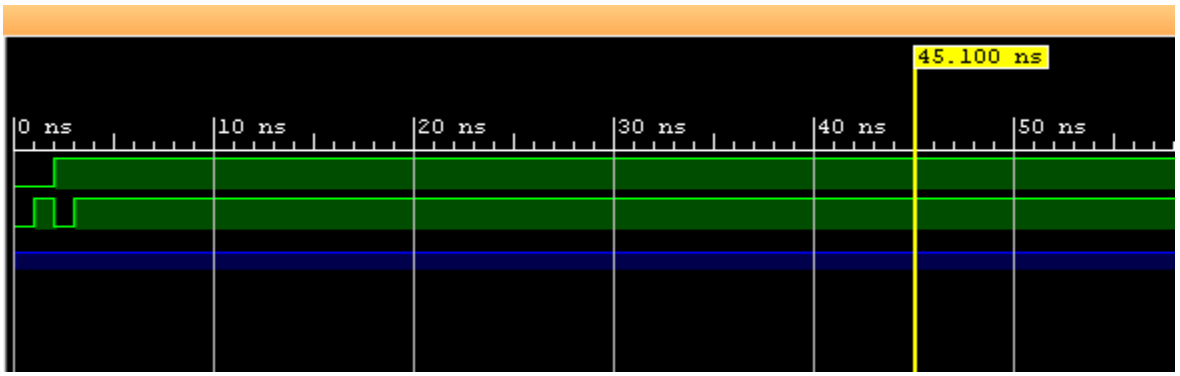


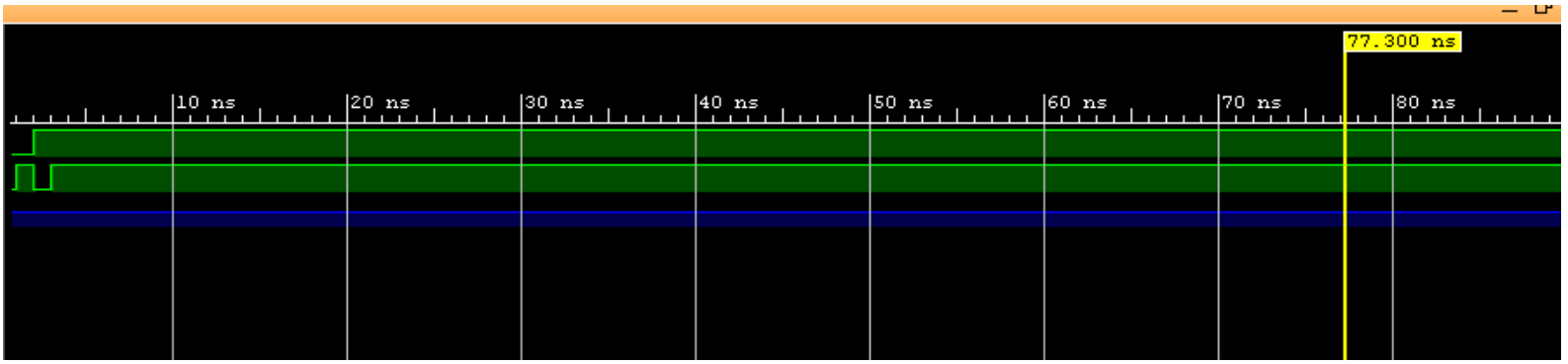


Output (For 4 bit binary subtractor) :

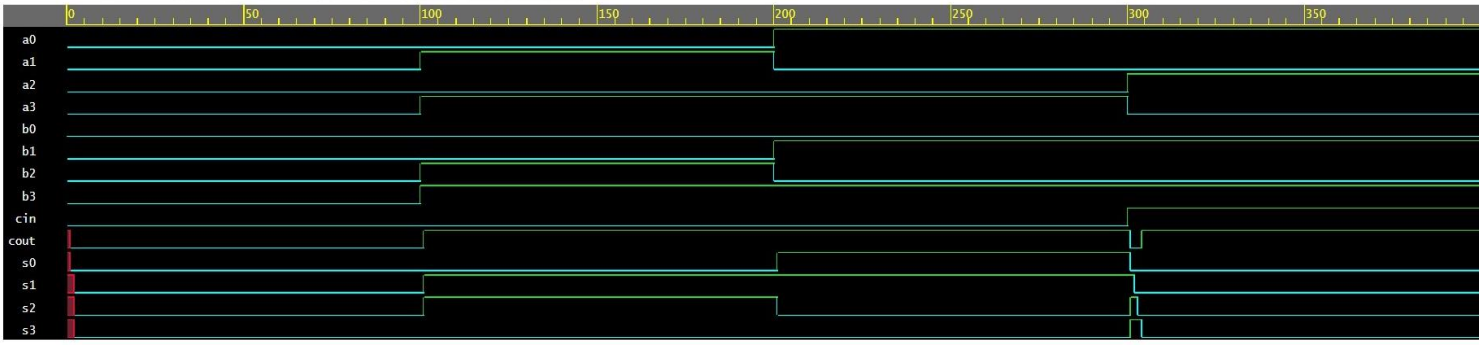




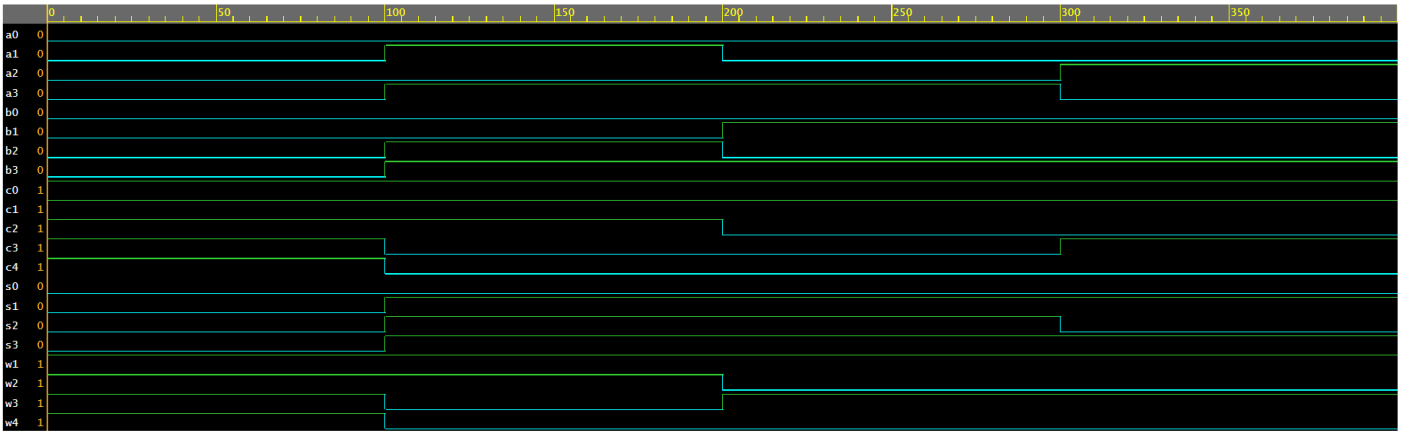




Output for 4 bit adder (Using EDA playground) :



Output for 4 bit subtractor (Using EDA playground) :



ECE1003_Digital Logic Design_L57+L58_Experiment-4

Name : Khan Mohd. Owais Raza
Reg. No. : 20BCD7138

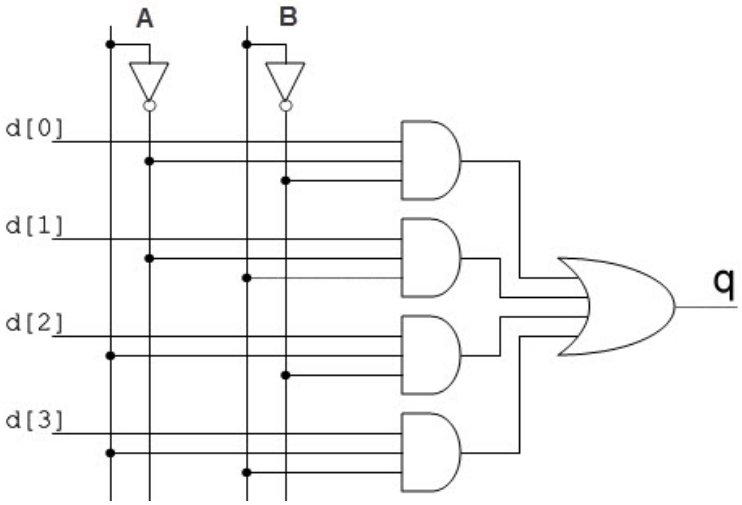
Aim : Design 1:4 Multiplexers (MUX) and 1:4 De-Multiplexers (DEMUX)

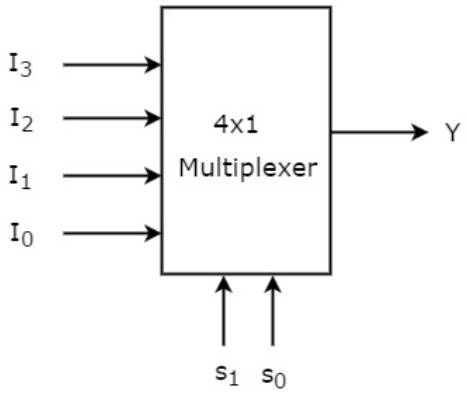
Software used : Xilinx Vivado 2014.2

Theory : Multiplexer means many into one. A multiplexer is a circuit used to select and route any one of the several input signals to a single output. A simple example of an non-electronic circuit of a multiplexer is a single pole multi-position switch. Multi-position switches are widely used in many electronics circuits. However, circuits that operate at high speed require the multiplexer to be automatically selected. A mechanical switch cannot perform this task efficiently. Therefore, multiplexer is used to perform high speed switching are constructed of electronic components. Multiplexers can handle two type of data i.e., analog and digital. For analog application, multiplexer are built using relays and transistor switches. For digital application, they are built from standard logic gates.

4:1 MUX :- It has four data inputs I_3, I_2, I_1 & I_0, two selection lines s_1 & s_0 and one output Y.

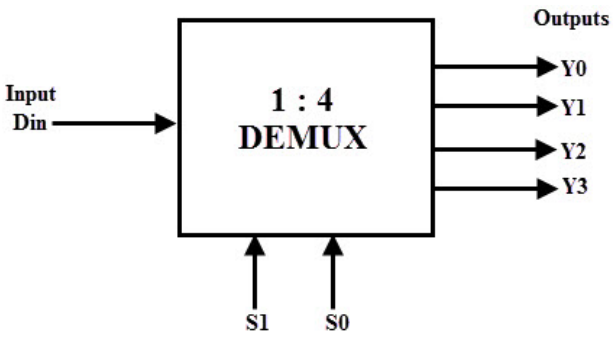
$$Y=S_1'S_0'I_0+S_1'S_0I_1+S_1S_0'I_2+S_1S_0I_3$$

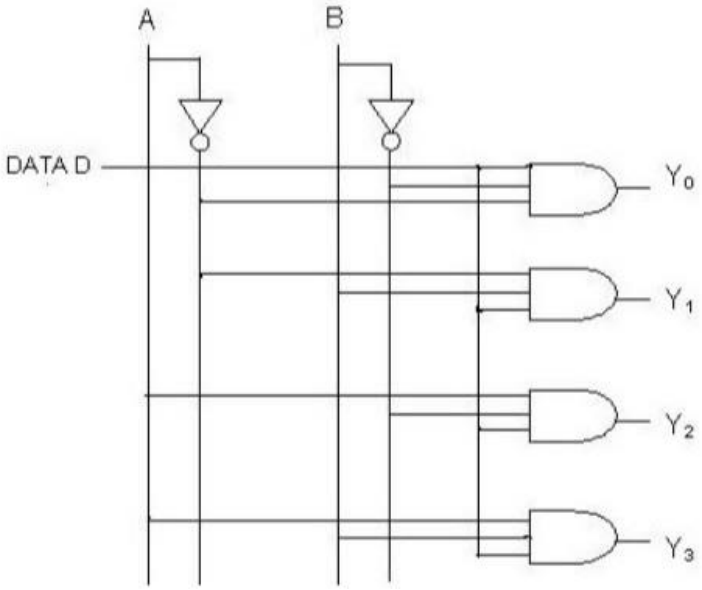




1:4 DE-MUX :- Demultiplexer performs the reverse operation of the multiplexer i.e. it takes a single output and can guide that single output through many outputs. The output to which the input signal is to be passed is decided by the control logic. The control logic can be manipulated by changing the value of the control signal. Thus, it is termed as the demultiplexer as it takes one input and generates several outputs.

The 1:4 Demultiplexer consists of 1 input signal, 2 control signals and 4 output signals. The number of the output signal is always decided by the number of the control signal and vice versa. There are 1 NOT gates through which control signals are passed, and 4 AND gates, which decides or control the output. The combination of the input signal along with control signals will decide that the output through which input signal will pass through.





ECE1003 Experiment No. 4 **(27/03/2021)**

NAME: Khan Mohd. OWAIS RAZA
REGISTRATION NO. : 20BCD7138

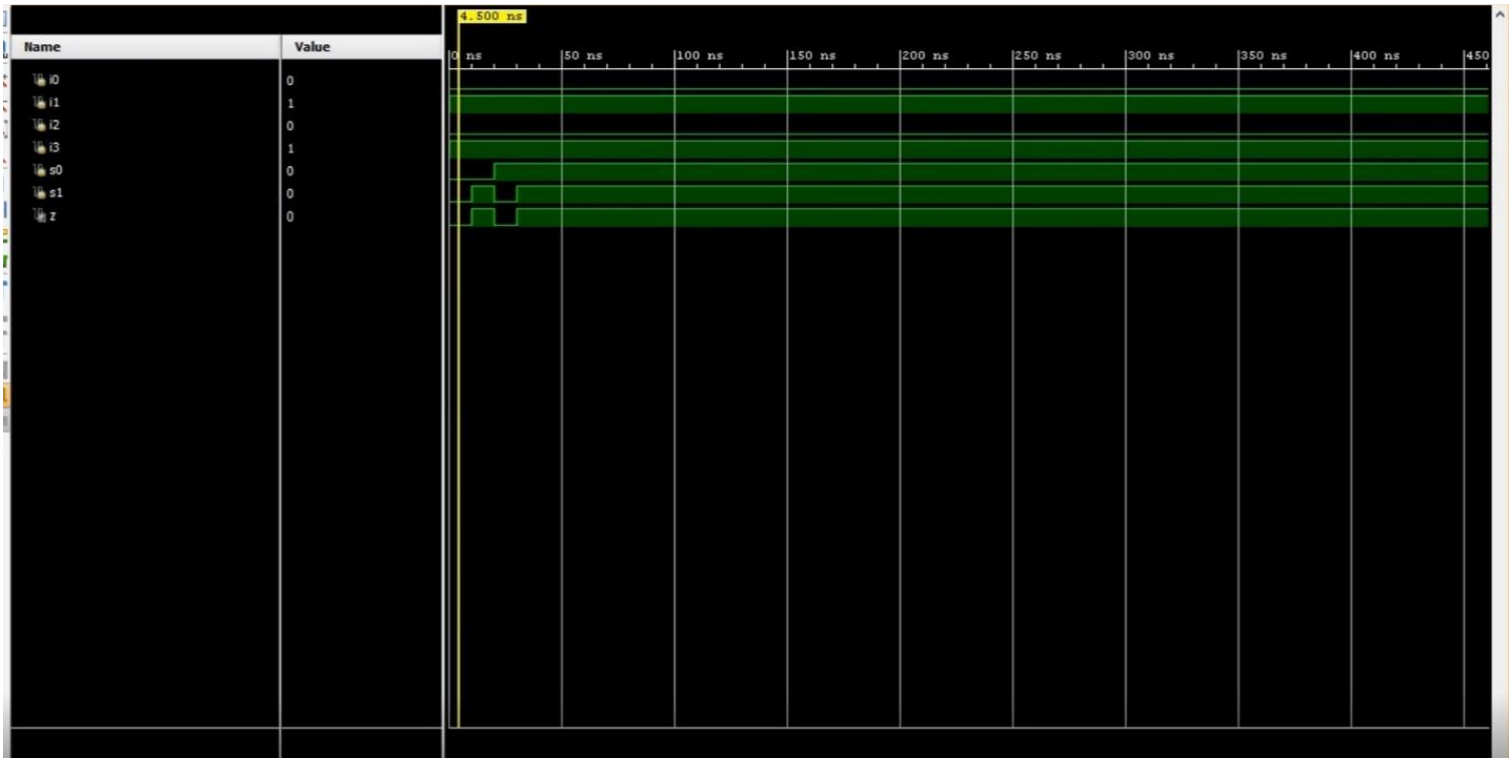
Aim: Design 1:4 Multiplexers and 1:4 De-Multiplexers
Software used : Xilinx Vivado 2014.2

Multiplexer (MUX):

I) Testbench :-

```
module tb_mu;
reg i0, i1, i3, s0, s1;
wire z;
mu_vi a1(i0, i1, i2, i3, s0, s1, z);
initial
begin
i0=1’b0; i1=1’b1; i2=1’b0; i3=1’b1;
s0=1’b0; s1=1’b0;
#10 s0=1’b0; s1=1’b1;
#10 s0=1’b1; s1=1’b0;
#10 s0=1’b1; s1=1’b1;
end
endmodule
```

Output :



II) Data Flow Modelling :

```
module multiplexer_ECE1003(a,b,c,d,s0,s1,y);
input a,b,c,d,s0,s1;
output y;
wire i1,i2,i3,i4,i5,i6;
assign i1=~s0;
assign i2=~s1;
assign i3=i1 & i2 & a;
assign i4=i1 & s1 & b;
assign i5=s0 & i2 & c;
assign i6=s0 & s1 & d;
assign y=i3|i4|i5|i6;
endmodule
```

III) Behavioral Modelling :

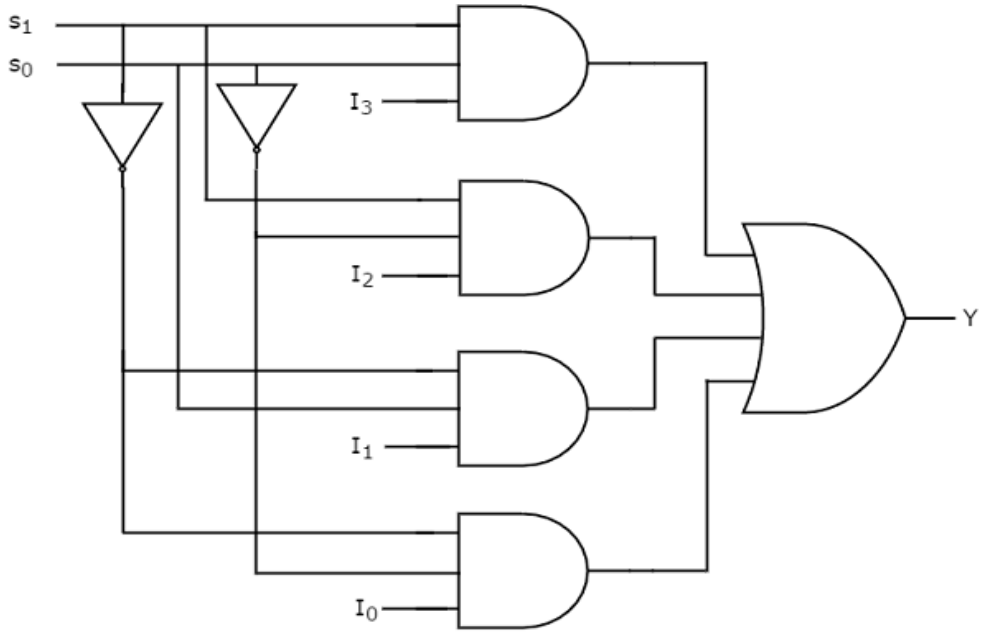
```
module multiplexer_ECE1003(a,b,c,d,s0,s1,y);
input a,b,c,d,s0,s1;
output y;
reg y;
always @(a or b or c or d or s0 or s1);

begin
if(s0==0 & s1==0)
y=a;
else if(s0==0 & s1==1)
y=b;
else if(s0==1 & s1==0)
y=c;
else
begin
y=d;
end
endmodule
```

IV) Structural Modelling :

```
module Multiplexer_Structural_ECE1003(a,b,c,d,s0,s1,y);
input a,b,c,d,s0,s1;
output y;
wire i1,i2,i3,i4,i5,i6;
not(i1,s0);
not(i2,s1);
and(i3,i1,i2,a);
and(i4,i1,s1,b);
and(i5,s0,i2,c);
and(i6,s0,s1,d);
or(y,i3,i4,i5,i6);
end
endmodule
```

Circuit Diagram & Truth table:

S0	S1	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3



Demultiplexer (DEMUX) :

I) Testbench

```
module testbench_Demux_ECE1003(input_a, sel, output_y);
input a_in;
input [1:0] sel;
output [3:0] y_out;
reg [3:0] y_out;
always @(a_in, sel)
begin
case (sel)
2'b00:begin y_out[0]=a_in; y_out[1]= 1'b0;
y_out[2]= 1'b0;y_out[3]=1'b0; end
2'b01: begin y_out[0]= 1'b0;y_out[1]=a_in;
y_out[2]= 1'b0;y_out[3]=1'b0; end
2'b10: begin y_out[0]= 1'b0;y_out[1]=1'b0;
y_out[2]=a_in; y_out[3]=1'b0; end
2'b11: begin y_out[0]= 1'b0; y_out[1]= 1'b0;
y_out[2]=1'b0;y_out[3]=a_in; end
default: y_out=3'b000;
end
endmodule
```

II) Behavioural Modeling

```
module demux_ECE1003_output_Y, input_A, din);
always @(Y, A) begin
case (A)
2'b00 : begin Y[0] = din; Y[3:1] = 0;
2'b01 : begin Y[1] = din; Y[0] = 0;
2'b10 : begin Y[2] = din; Y[1:0] = 0;
2'b11 : begin Y[3] = din; Y[2:0] = 0;
end
endmodule
```

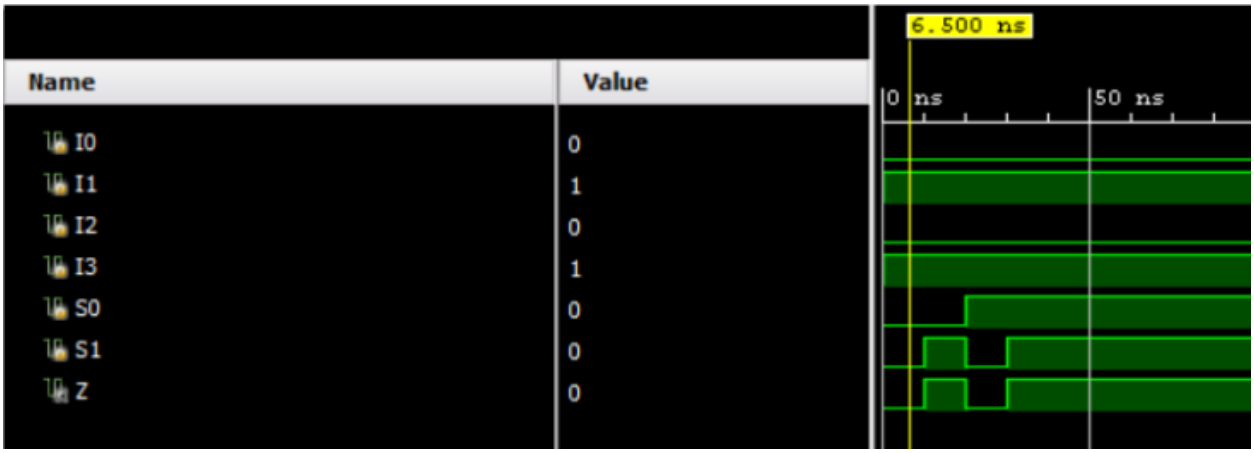
III) Data Flow Modelling

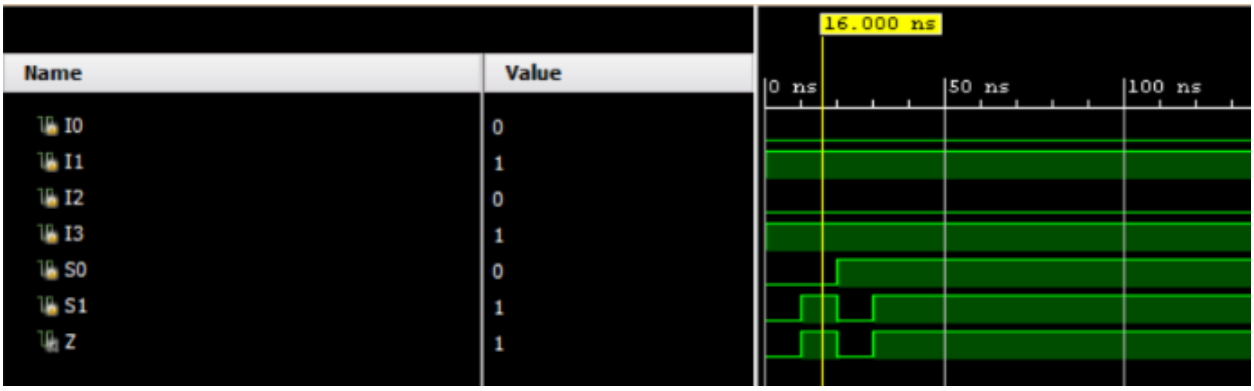
```
module demux_ECE1003_DataFlow( din ,x ,y ,a ,b ,c ,d );

output a ;
output b ;
output c ;
output d ;

input din ;
input x ;
input y ;

assign a = din & (~x) & (~y);
assign b = din & (~x) & y;
assign c = din & x & (~y);
assign d = din & x & y;

end
endmodule
```

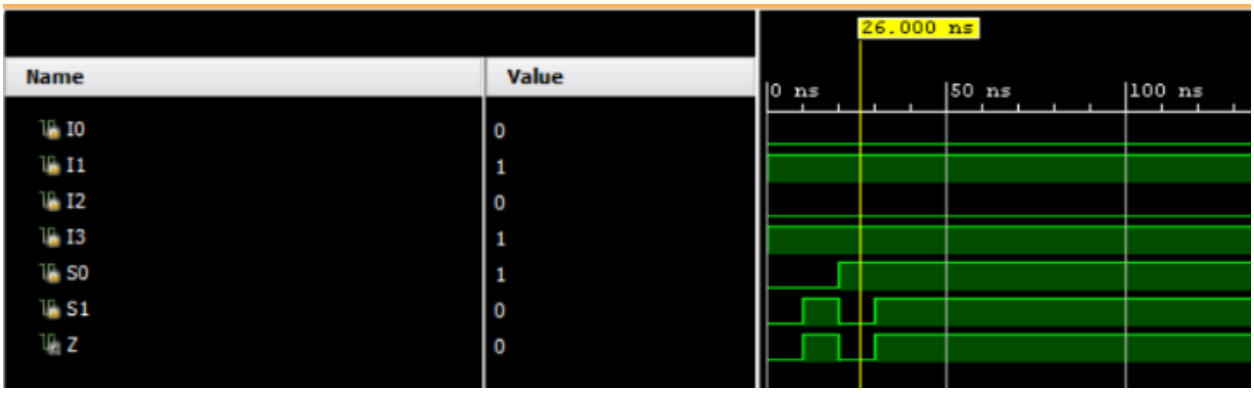
Truth Table:

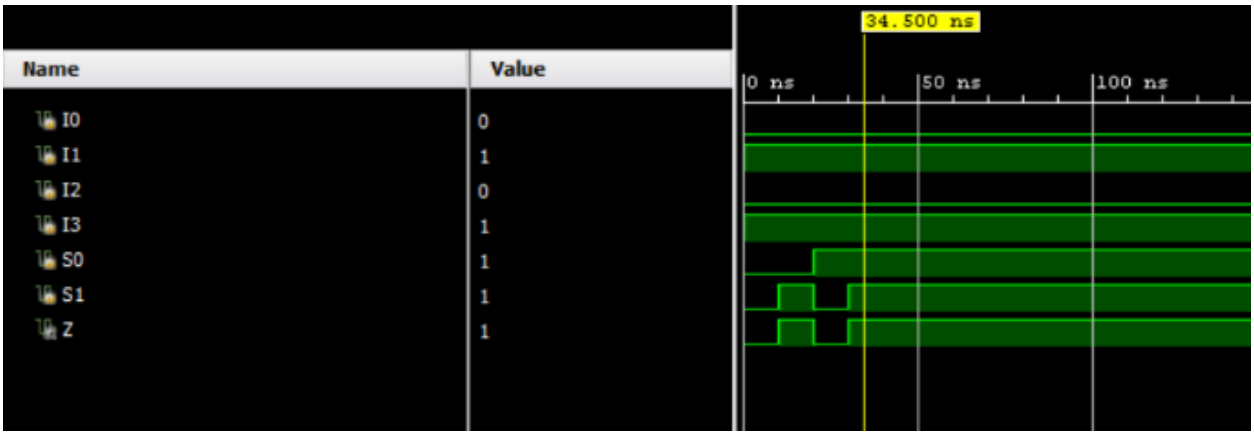
S0	S1	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

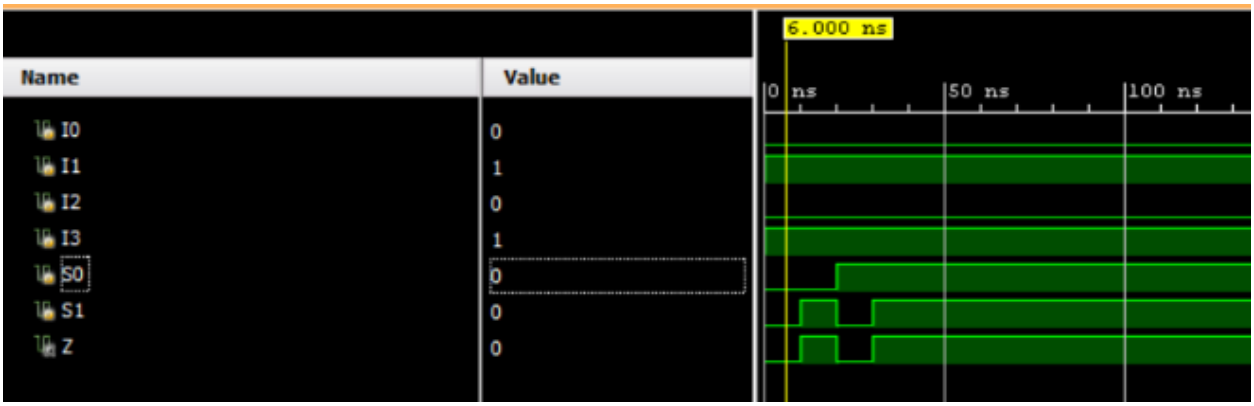
Output (For 1:4 MUX) :



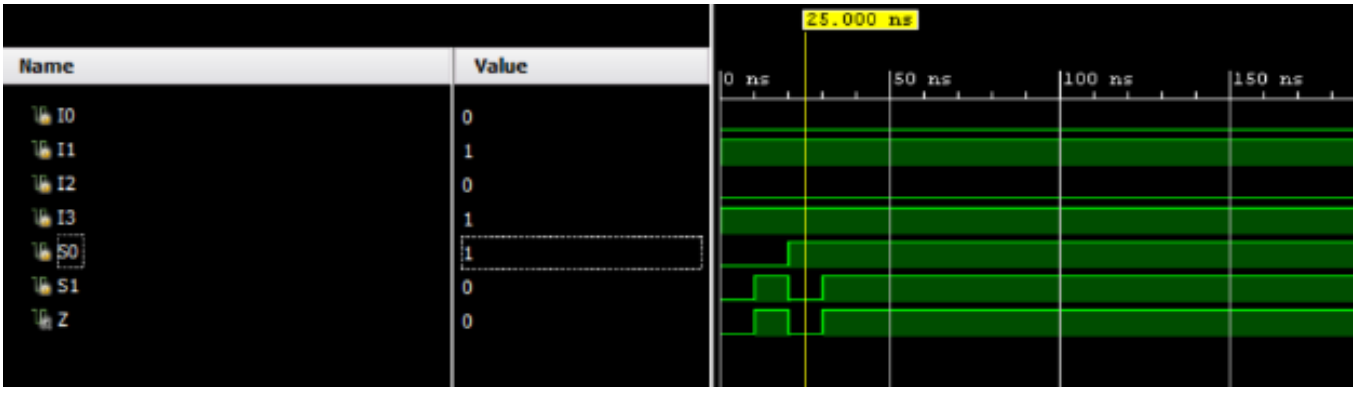


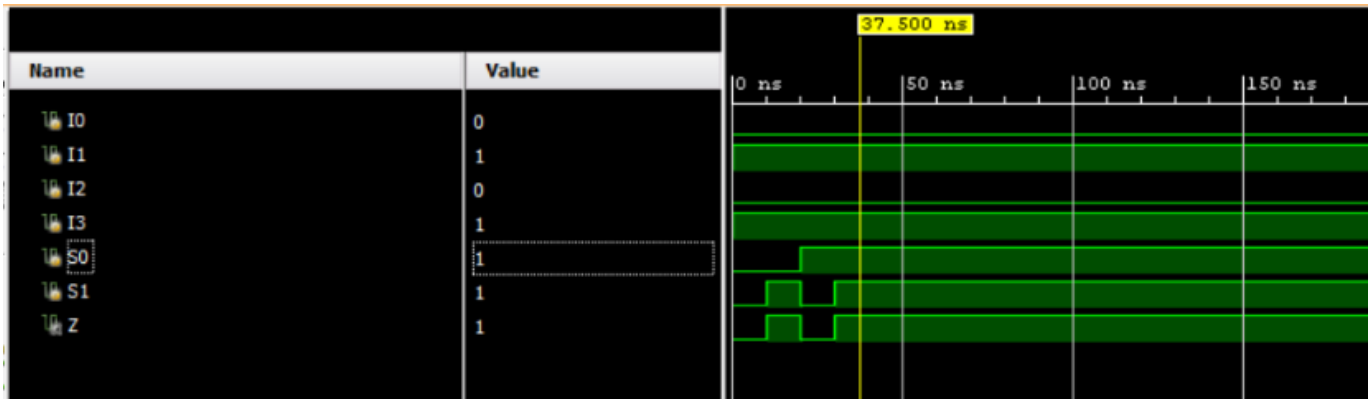



Name
Value
I0
I1
I2
I3
S0
S1
Z
0
1
0
1
1
1
1
34.500 ns
0 ns
50 ns
100 ns

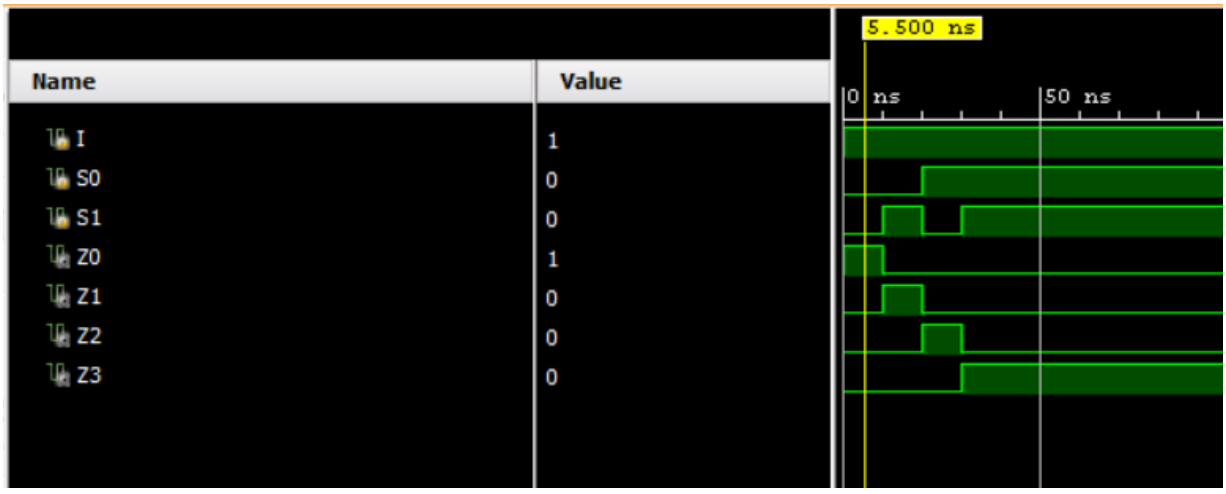

Name
Value
I0
I1
I2
I3
S0
S1
Z
0
1
0
1
0
0
0
6.000 ns
0 ns
50 ns
100 ns

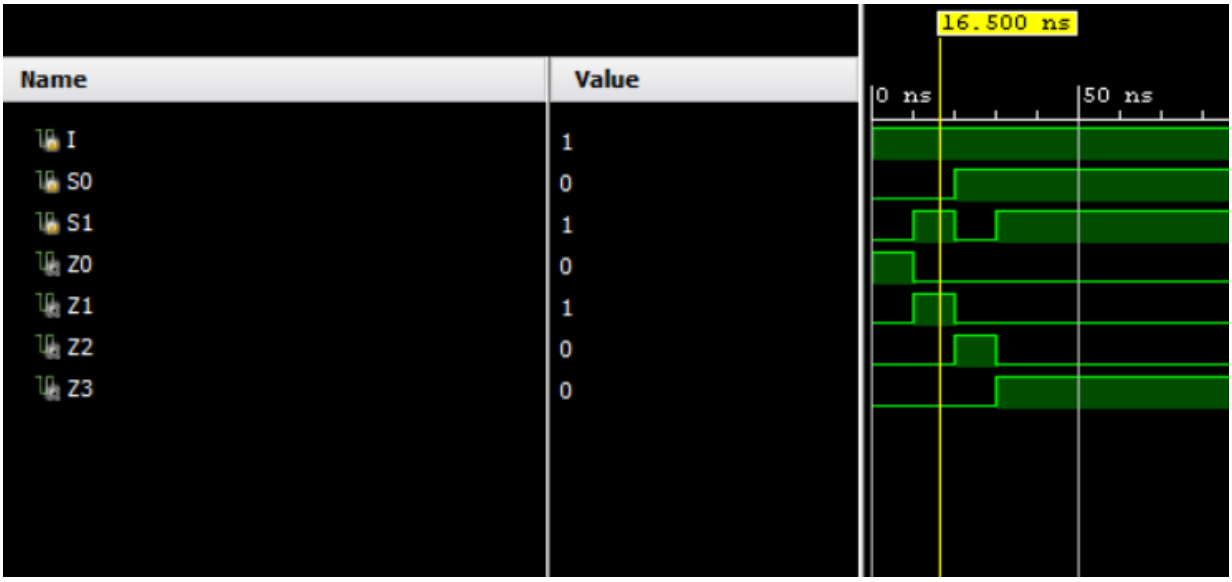

Name
Value
I0
I1
I2
I3
S0
S1
Z
0
1
0
1
0
1
1
14.500 ns
0 ns
50 ns
100 ns


Name
Value
I0
I1
I2
I3
S0
S1
Z
0
1
0
1
1
0
0
25.000 ns
0 ns
50 ns
100 ns
150 ns

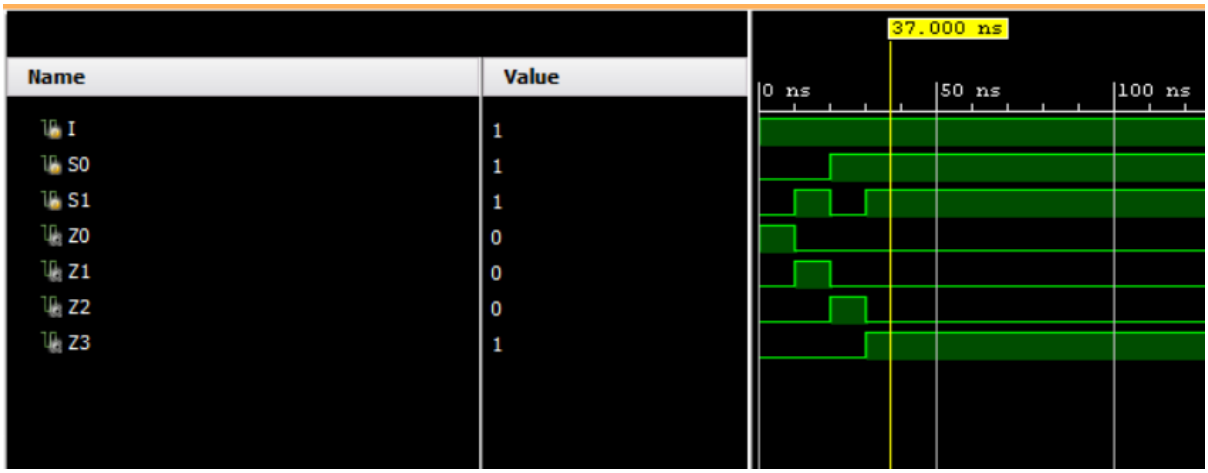


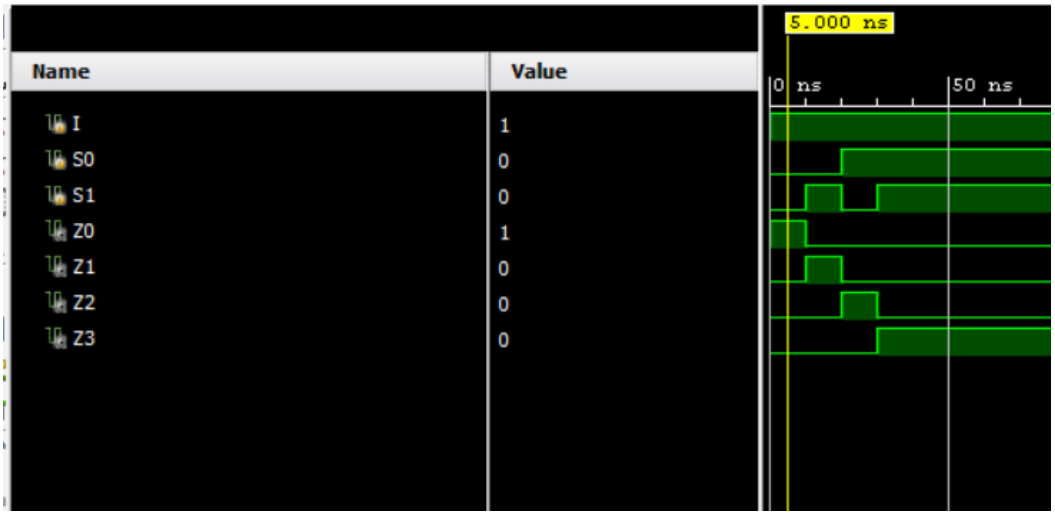
Output (For 4:1 DEMUX) :

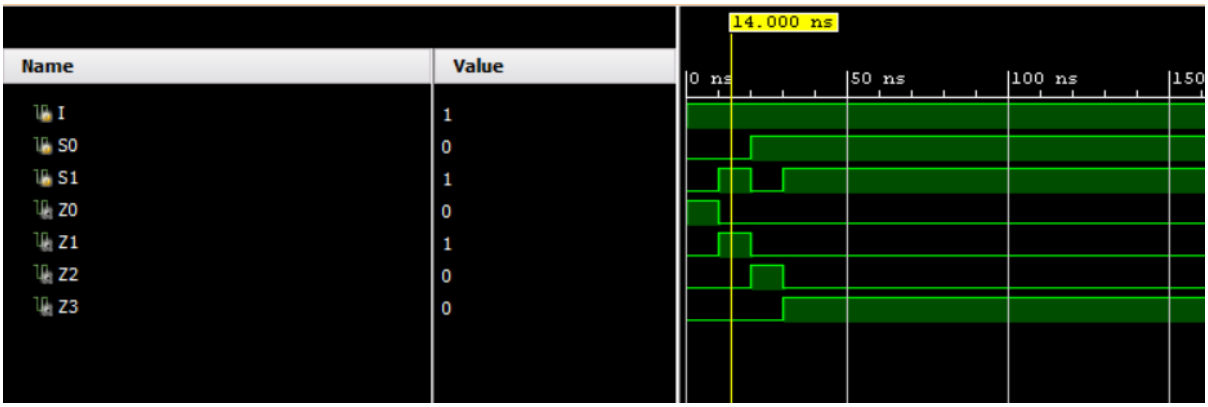


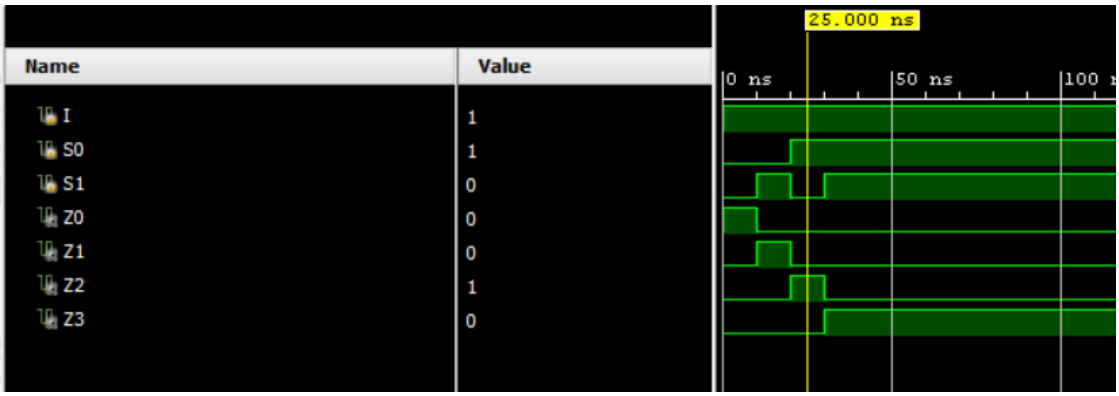


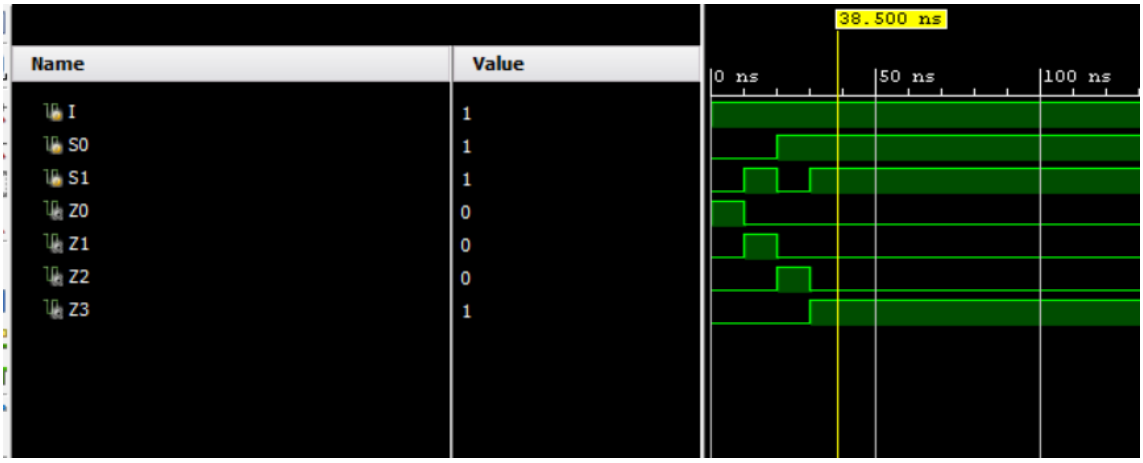

Name
Value
I
S0
S1
Z0
Z1
Z2
Z3
1
1
0
0
0
1
0
26.000 ns
0 ns
50 ns
100 ns


Name
Value
I
S0
S1
Z0
Z1
Z2
Z3
1
1
1
0
0
0
1
37.000 ns
0 ns
50 ns
100 ns


Name
Value
I
S0
S1
Z0
Z1
Z2
Z3
1
0
0
1
0
0
0
5.000 ns
0 ns
50 ns


14.000 ns
Name
Value
I
1
S0
0
S1
1
Z0
0
Z1
1
Z2
0
Z3
0
0 ns
50 ns
100 ns


25.000 ns
Name
Value
I
1
S0
1
S1
0
Z0
0
Z1
0
Z2
1
Z3
0
0 ns
50 ns


38.500 ns
Name
Value
I
1
S0
1
S1
1
Z0
0
Z1
0
Z2
0
Z3
1
0 ns
50 ns
100 ns

ECE1003_Digital Logic Design_L57+L58_Experiment-5

Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

Aim : Design and verify encoders and decoders in verilog

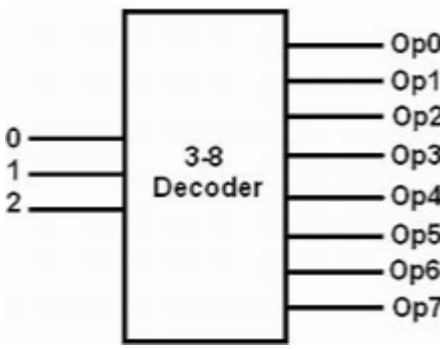
Software used : Xilinx Vivado 2014.2

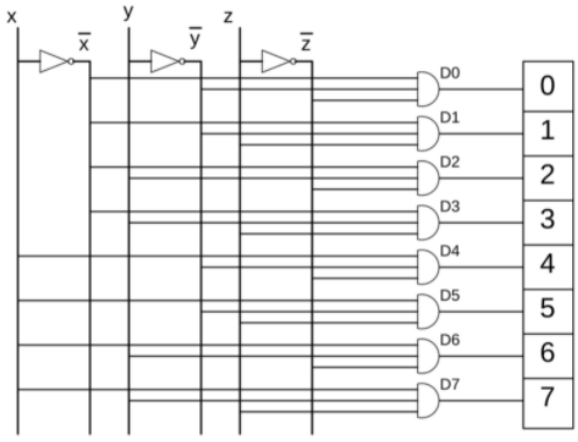
Theory :

Decoder:- The 3 to 8 line decoder is also known as **Binary to Octal Decoder**. In a 3 to 8 line decoder, there is a total of eight outputs, i.e., Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7 and three outputs, i.e., A_0, A1, and A_2. This circuit has an enable input 'E'. Just like 2 to 4 line decoder, when enable 'E' is set to 1, one of these four outputs will be 1.

Enable	INPUTS			Outputs							
E	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

$Y_0=A_0'.A_1'.A_2'$; $Y_1=A_0.A_1'.A_2'$; $Y_2=A_0'.A_1.A_2'$; $Y_3=A_0.A_1.A_2'$; $Y_4=A_0'.A_1'.A_2$; $Y_5=A_0.A_1'.A_2$; $Y_6=A_0'.A_1.A_2$; $Y_7=A_0.A_1.A_2$

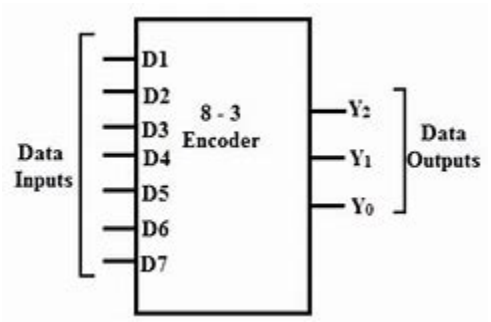


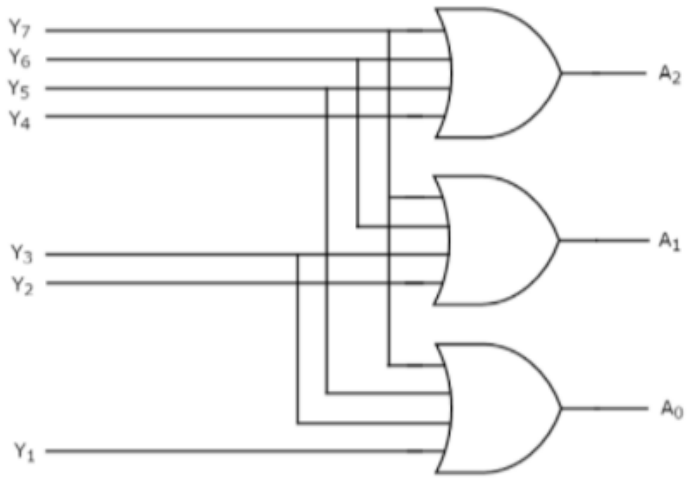


Encoder :- The 8 to 3 line Encoder is also known as Octal to Binary Encoder. In 8 to 3 line encoder, there is a total of eight inputs, i.e., Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7 and three outputs, i.e., A_0, A1, and A_2. In 8-input lines, one input-line is set to true at a time to get the respective binary code in the output side.

INPUTS								OUTPUTS		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
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

$A_2=Y_4+Y_5+Y_6+Y_7$; $A_1=Y_2+Y_3+Y_6+Y_7$; $A_0=Y_7+Y_5+Y_3+Y_1$;





ECE1003 Experiment No. 5 **12/04/2021**

Name : Khan Mohd. Owais Raza
Registration No. : 20BCD7138

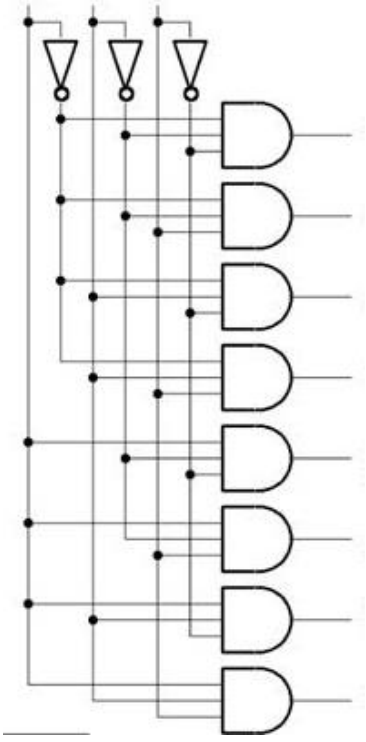
Aim: Design and Verify Decoders and Encoders
Software used : Xilinx Vivado 2014.2

I] 3*8 Decoder

1) Truth Table :

I0	I1	I2	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

2) Circuit Diagram:



3) Data Flow:

```
module_decoder(i0,i1,i2,z0,z1,z2,z3,z4,z5,z6,z7)
input i2,i1,i0;
output z0,z1,z2,z3,z4,z5,z6,z7;
z0=(~i[2])&(~i[1])&(~i[0]);
assign z[1]=(~i[2])&(~i[1])&(i[0]);
assign z[2]=(~i[2])&(i[1])&(~i[0]);
assign z[3]=(~i[2])&(i[1])&(i[0]);
assign z[4]=(i[2])&(~i[1])&(~i[0]);
assign z[5]=(i[2])&(~i[1])&(i[0]);
assign z[6]=(i[2])&(i[1])&(~i[0]);
assign z[7]=(i[2])&(i[1])&(i[0]);
endmodule
```

4) Testbench:

```
module_vitapstudents_tb_decoder;
reg i2,i1,i1
wire z0,z1,z2,z3,z4,z5,z6,z7;
decoder a1(i2,i1,i1,z0,z1,z2,z3,z4,z5,z6,z7);
initial
begin
i2=1'b0;i1=1'b0;i0=1'b0;
#10 i2=1'b0;i1=1'b0;i0=1'b1;
#10 i2=1'b0;i1=1'b1;i0=1'b0;
#10 i2=1'b0;i1=1'b1;i0=1'b1;
#10 i2=1'b1;i1=1'b0;i0=1'b0;
#10 i2=1'b1;i1=1'b0;i0=1'b1;
#10 i2=1'b1;i1=1'b1;i0=1'b0;
#10 i2=1'b1;i1=1'b1;i0=1'b1;
end
endmodule
```

II] 8*3 Encoder

1) Truth table

I0	I1	I2	I3	I4	I5	I6	I7	Z2	Z1	Z0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

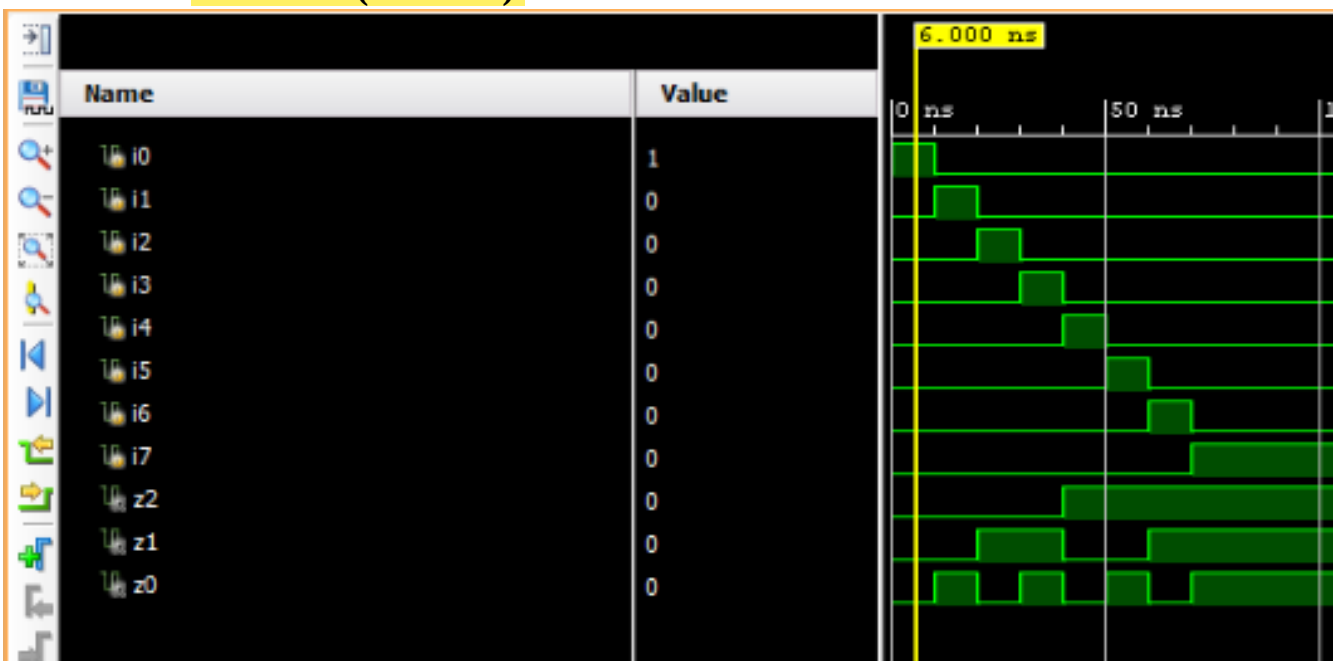
2) Testbench:

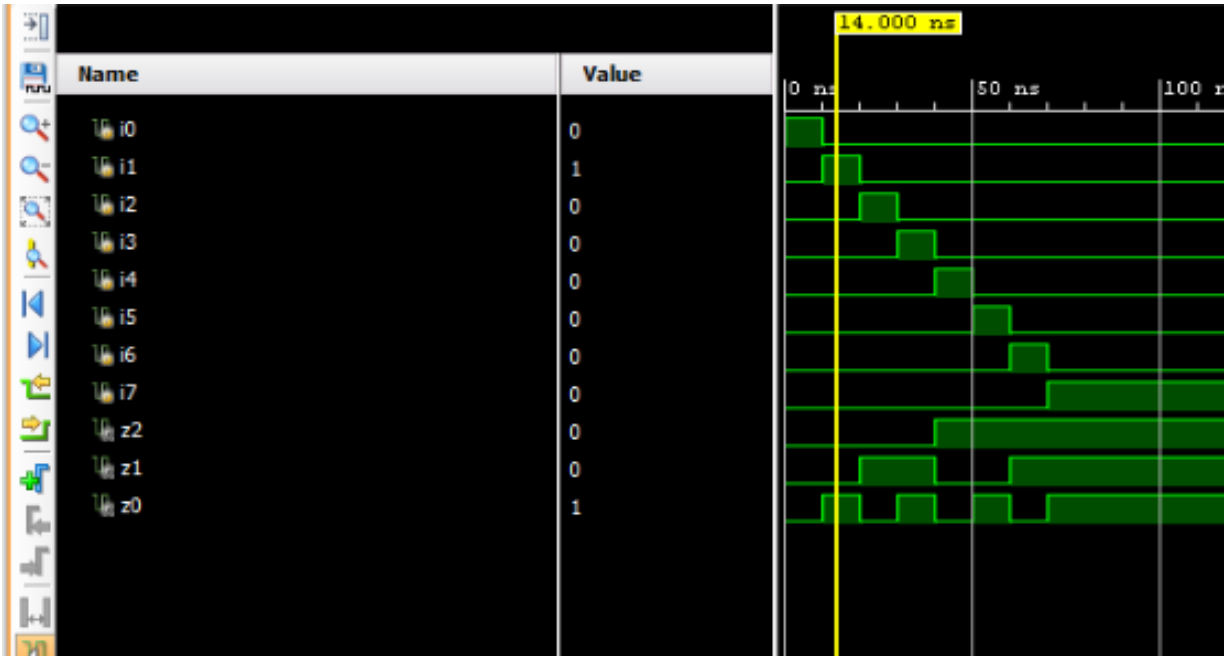
```
module vitapstudents_tb_encoder
reg i1,i2,i3,i4,i5,i6,i7;
wire z0,z1,z2;
encoder a1(i1,i2,i3,i4,i5,i6,i7,z0,z1,z2);
initial
begin
i1=1'b1; i2=1'b0; i3=1'b0; i4=1'b0; i5=1'b0; i6=1'b0;
i7=1'b0;

#10 i1=1'b0; i2=1'b1; i3=1'b0; i4=1'b0; i5=1'b0;
i6=1'b0; i7=1'b0;

#10 i1=1'b0; i2=1'b0; i3=1'b1; i4=1'b0; i5=1'b0;
i6=1'b0; i7=1'b0;

#10 i1=1'b0; i2=1'b0; i3=1'b0; i4=1'b1; i5=1'b0;
i6=1'b0; i7=1'b0;

end
endmodule
```

3) Data Flow :

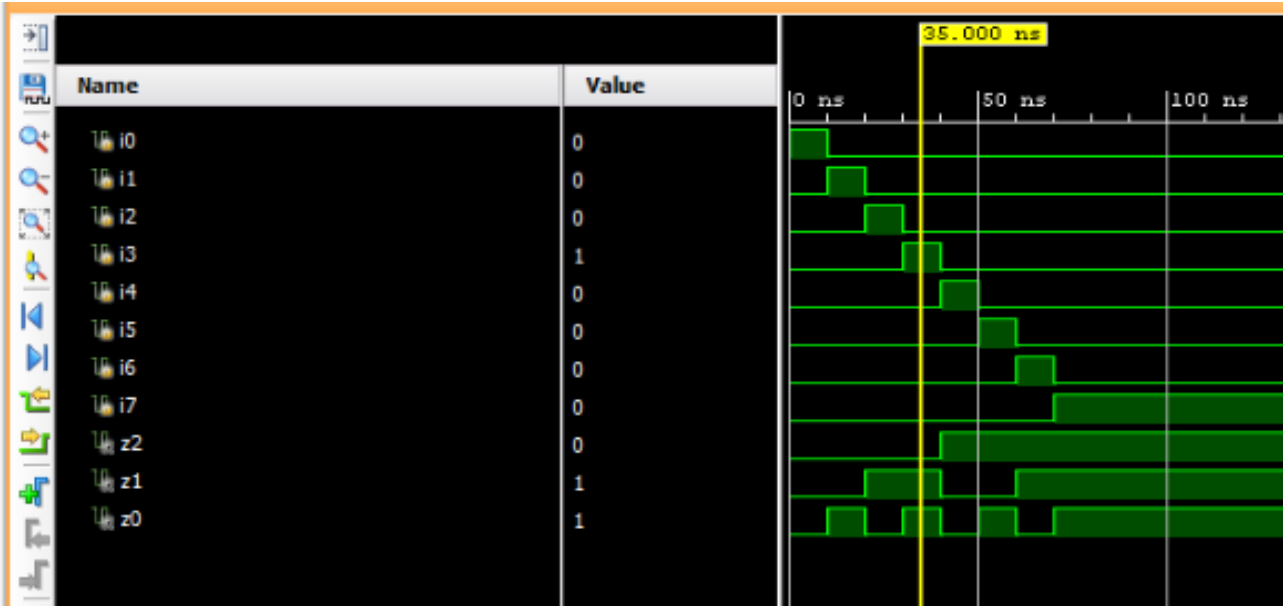
```
module encoder(i1,i2,i3,i4,i5,i6,i7,z0,z1,z2);
input i1,i2,i3,i4,i5,i6,i7;
output z0,z1,z2;
assign z2=i4|i5|i6|i7;
assign z3=i2|i3|i6|i7;
assign z3=i1|i3|i6|i7;
end
endmodule
```

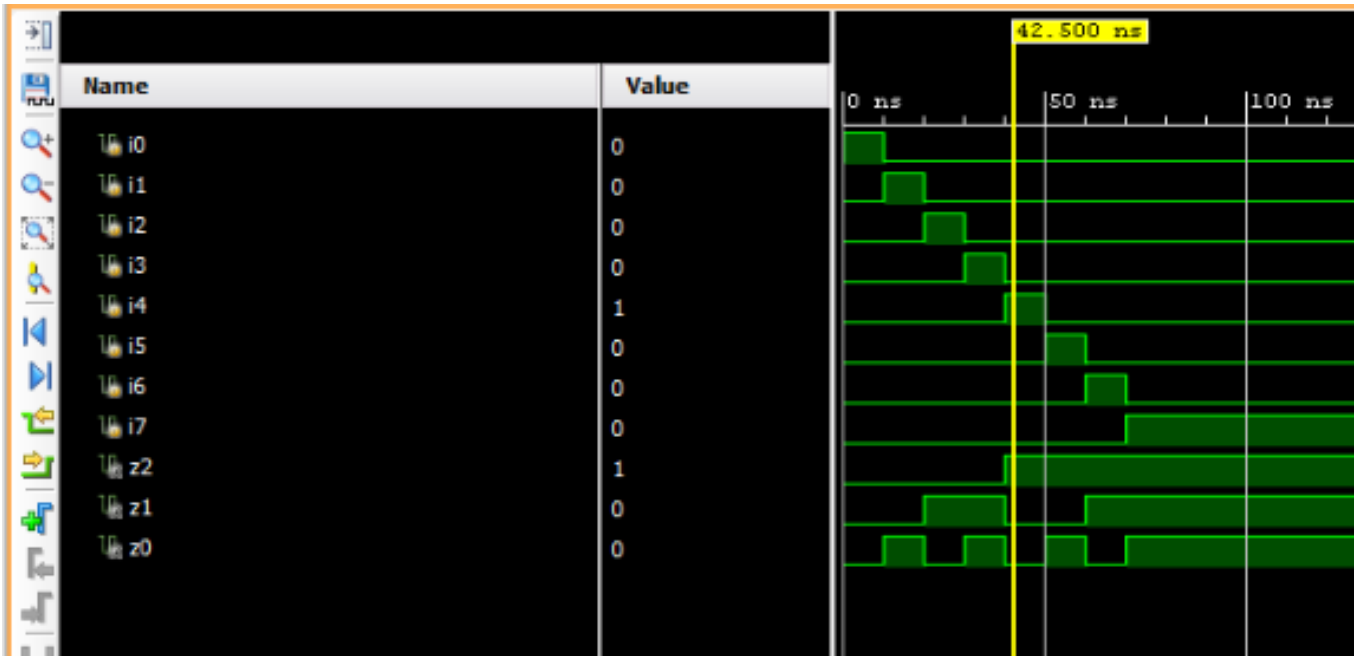
OUTPUT (Encoder) :

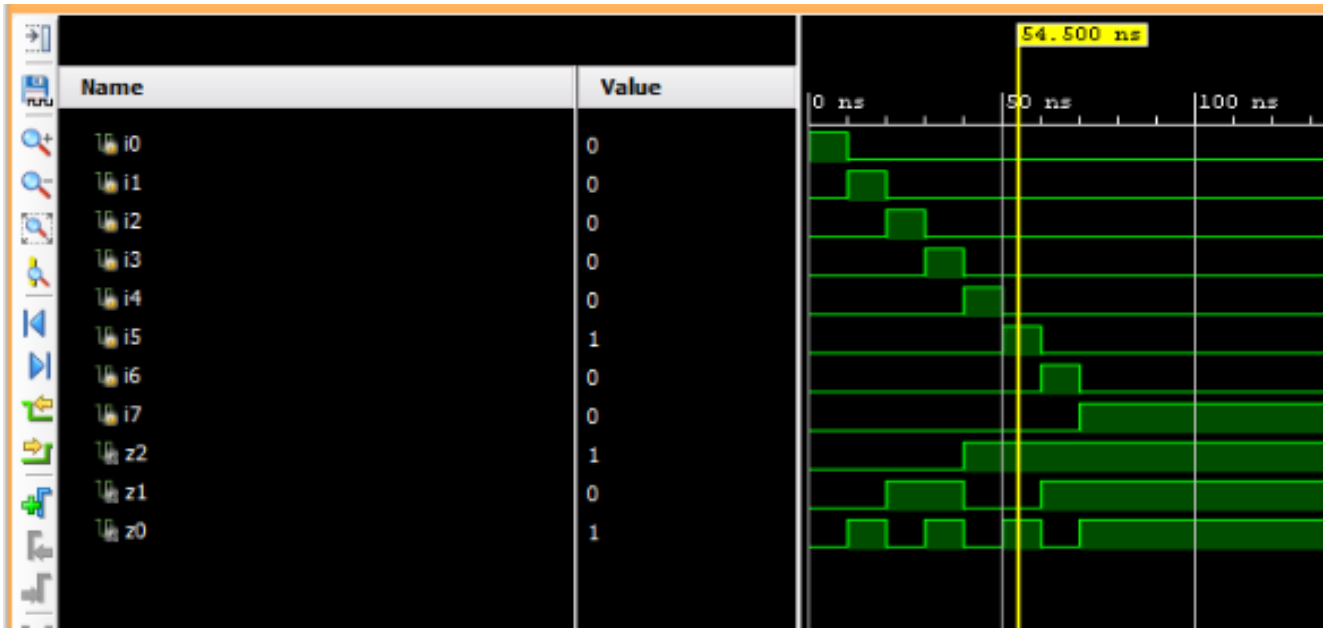


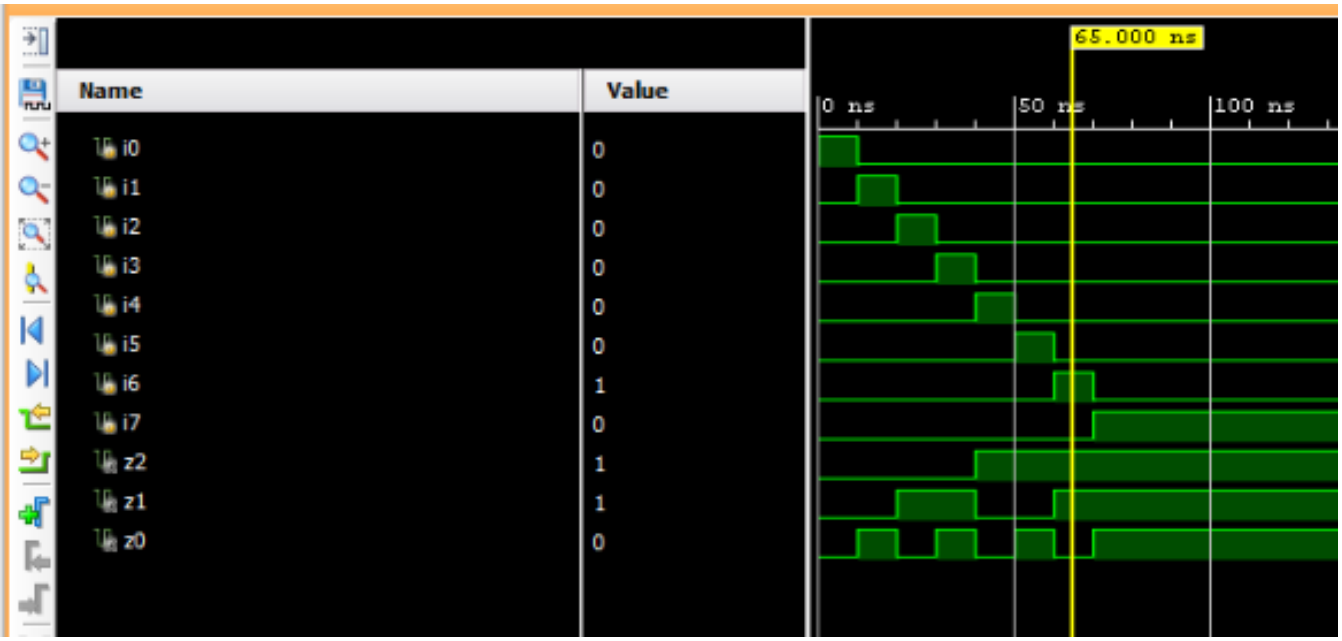

14.000 ns
Name
Value
i0 0
i1 1
i2 0
i3 0
i4 0
i5 0
i6 0
i7 0
z2 0
z1 0
z0 1
0 ns
50 ns
100 ns

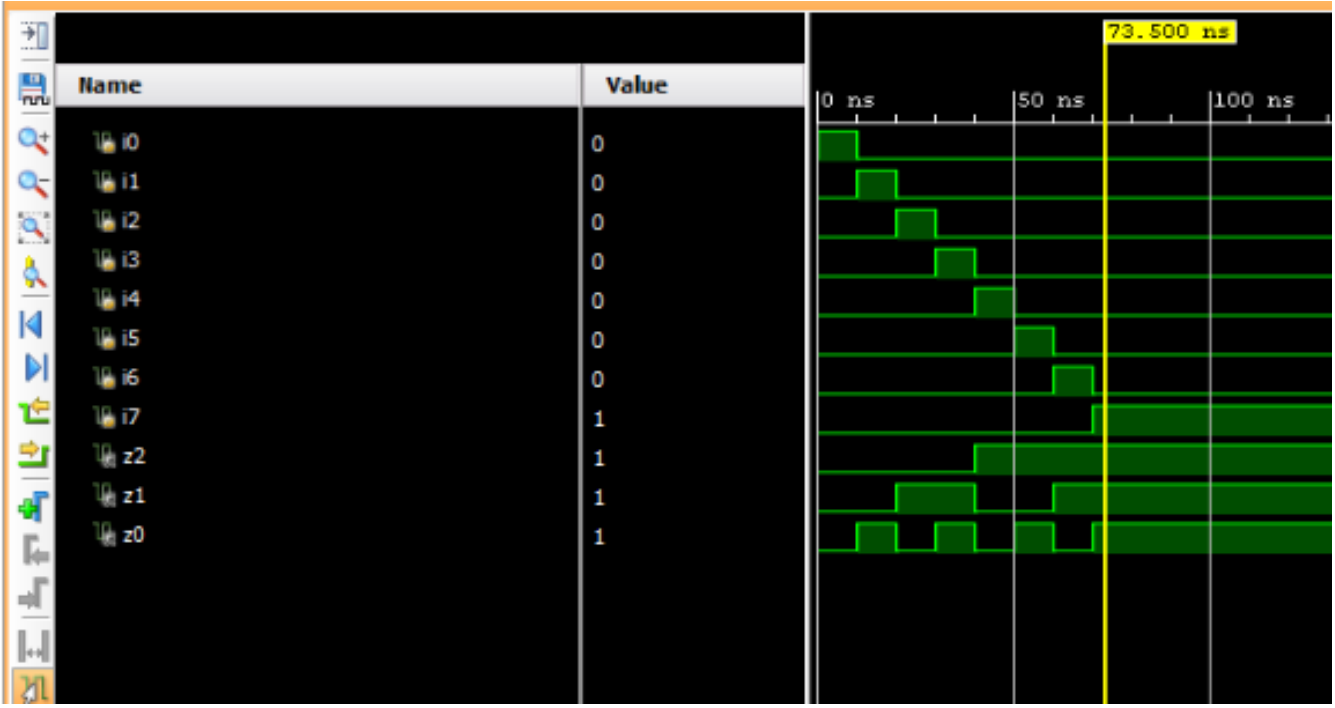

25.000 ns
Name
Value
i0 0
i1 0
i2 1
i3 0
i4 0
i5 0
i6 0
i7 0
z2 0
z1 1
z0 0
0 ns
50 ns
100 ns


35.000 ns
Name
Value
0 ns
50 ns
100 ns
i0 0
i1 0
i2 0
i3 1
i4 0
i5 0
i6 0
i7 0
z2 0
z1 1
z0 1

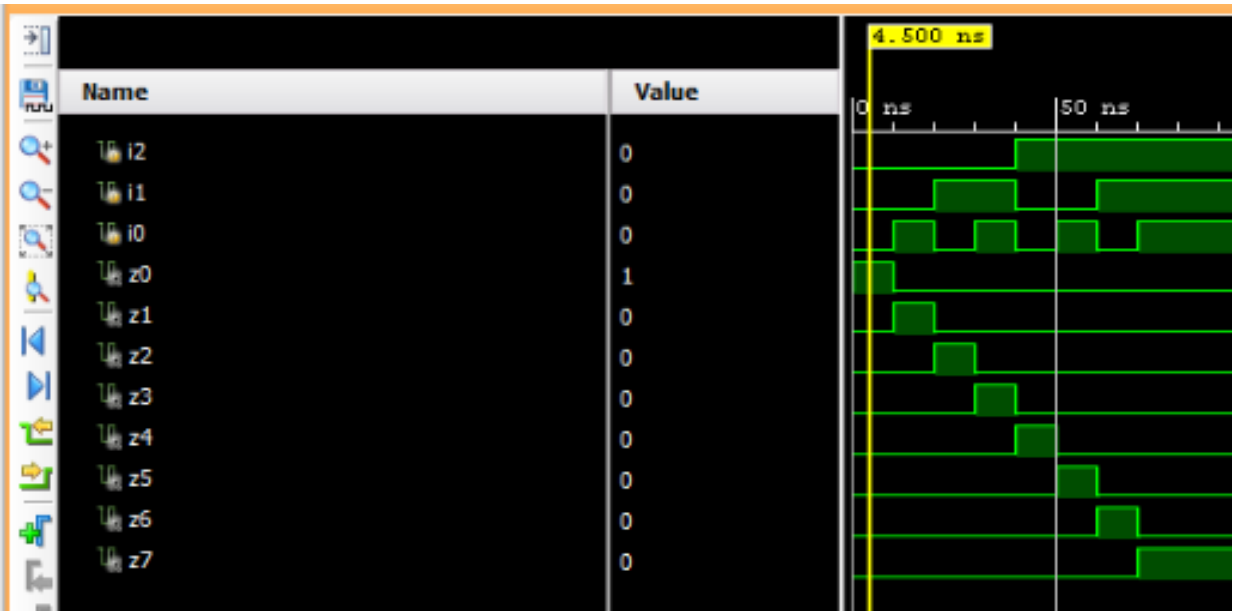

42.500 ns
Name
Value
0 ns
50 ns
100 ns
i0 0
i1 0
i2 0
i3 0
i4 1
i5 0
i6 0
i7 0
z2 1
z1 0
z0 0

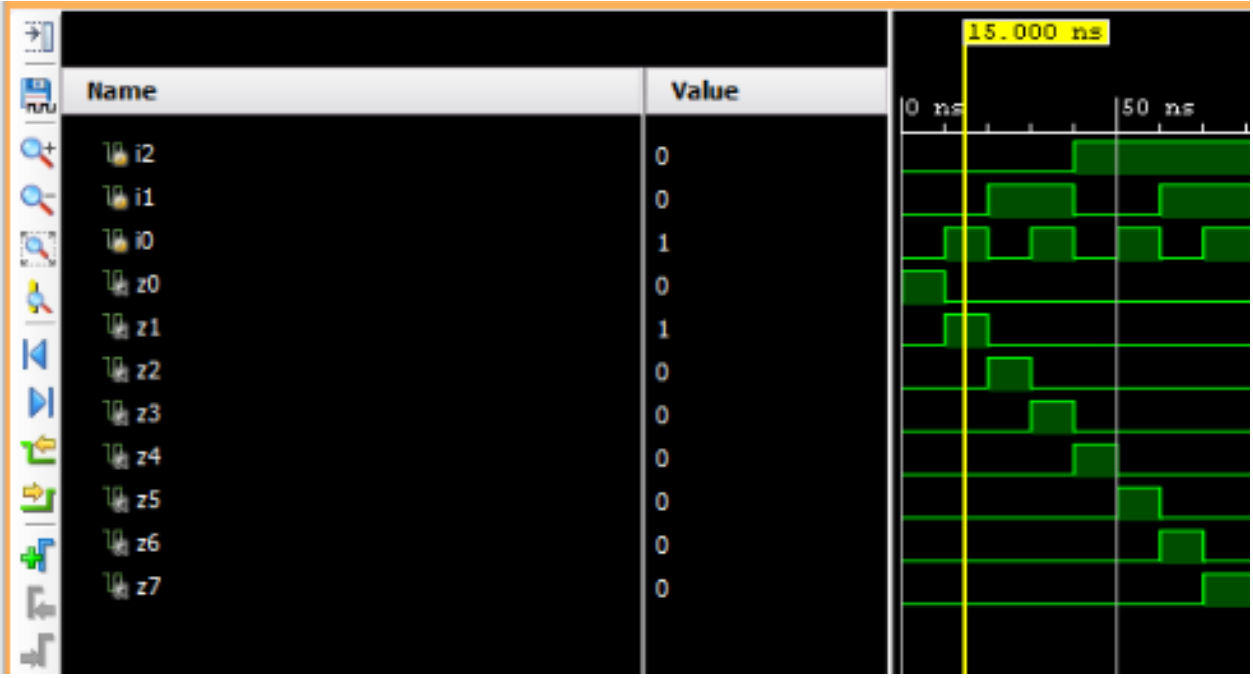

Name
Value
i0
0
i1
0
i2
0
i3
0
i4
0
i5
1
i6
0
i7
0
z2
1
z1
0
z0
1
54.500 ns
0 ns
50 ns
100 ns

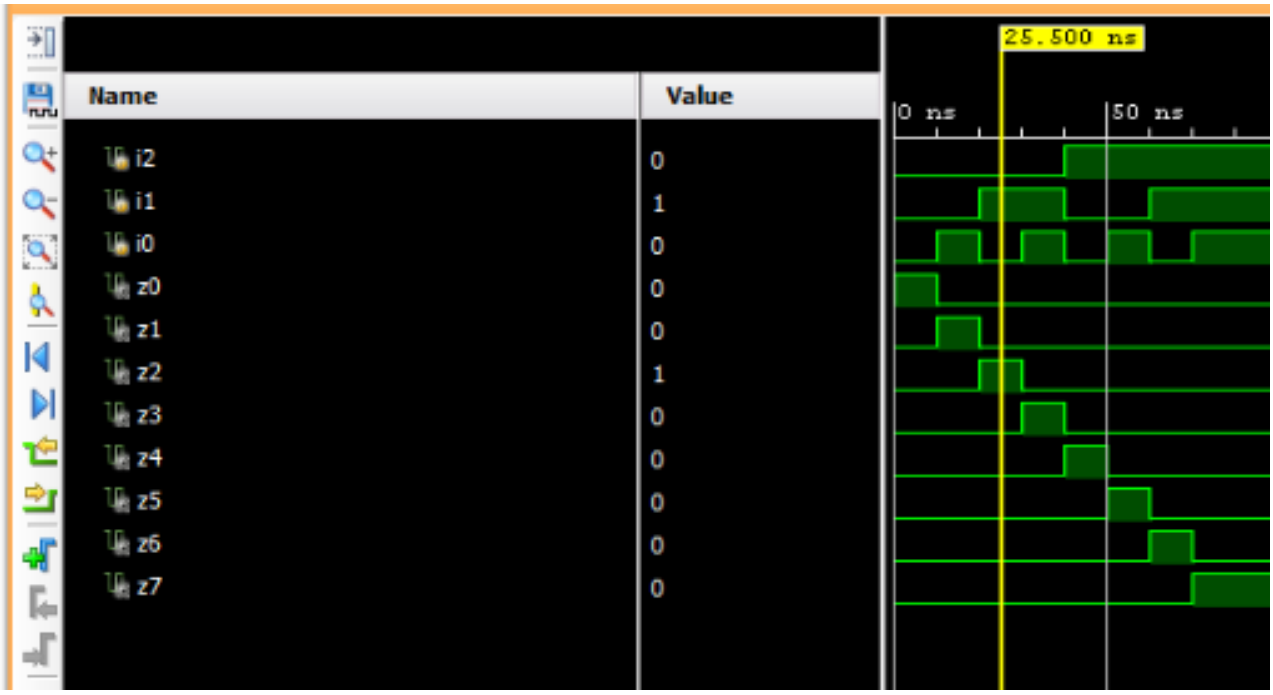

Name
Value
i0
0
i1
0
i2
0
i3
0
i4
0
i5
0
i6
1
i7
0
z2
1
z1
1
z0
0
65.000 ns
0 ns
50 ns
100 ns

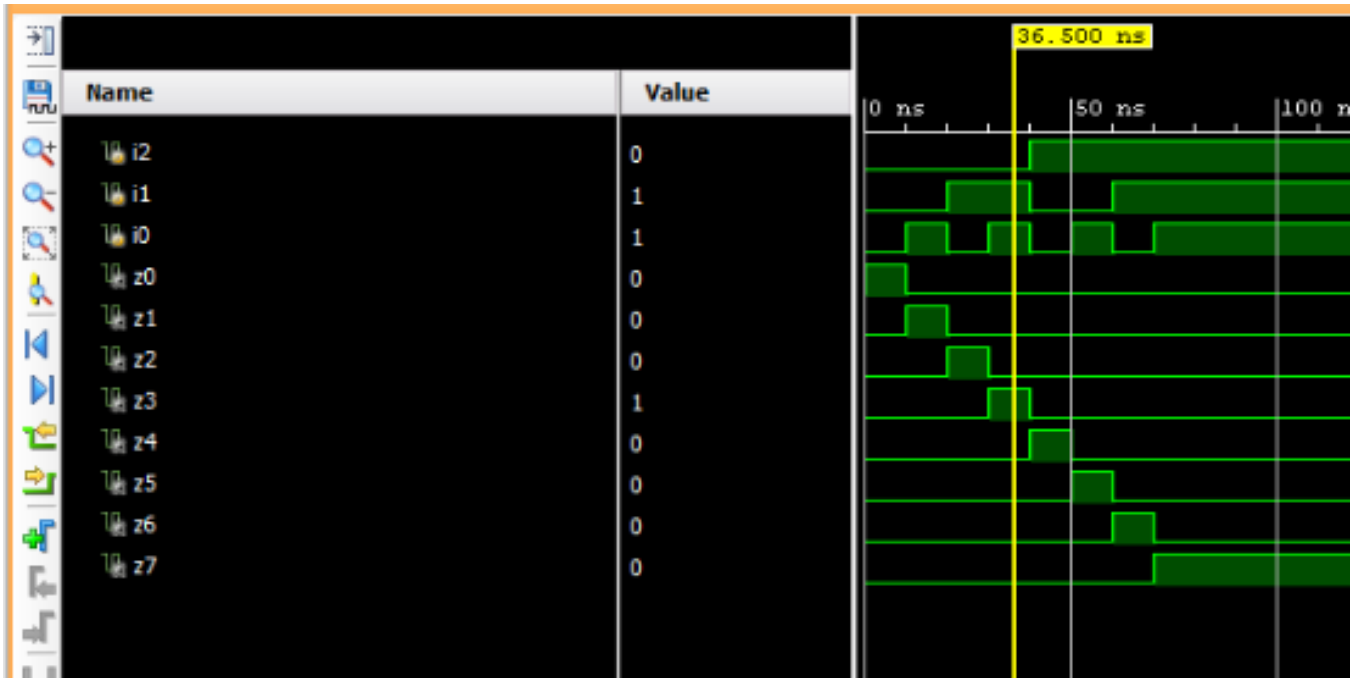


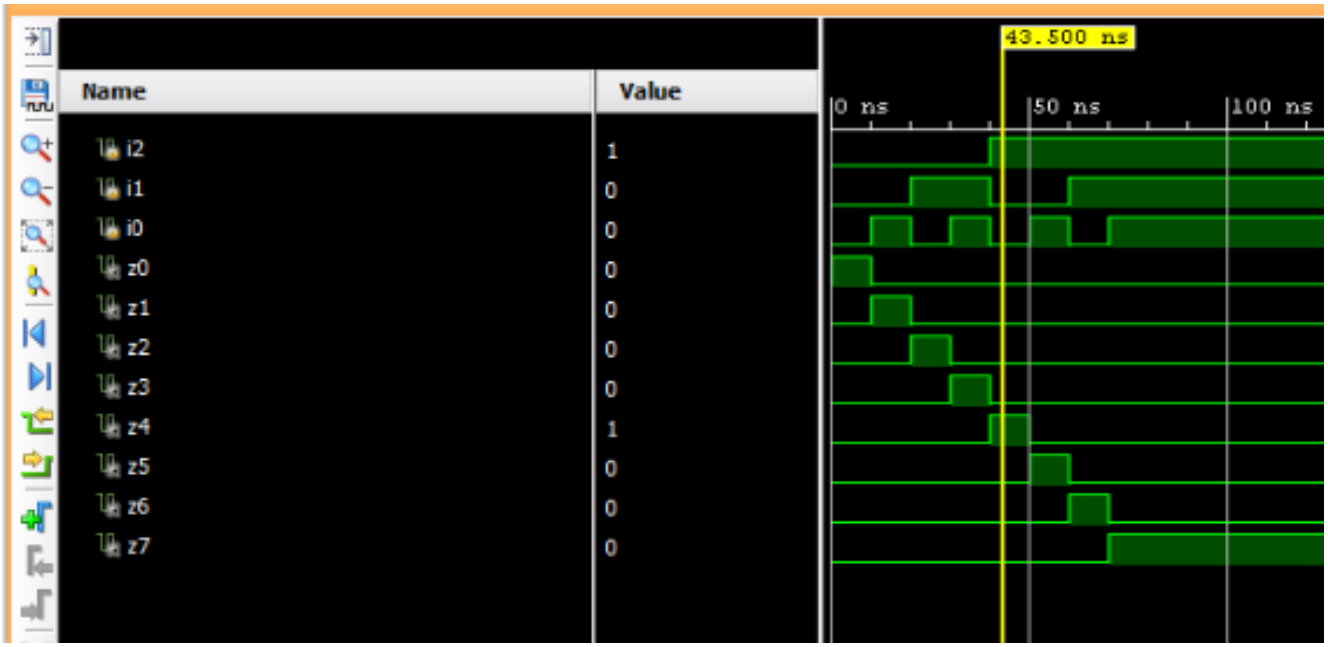
OUTPUT (Decoder) :

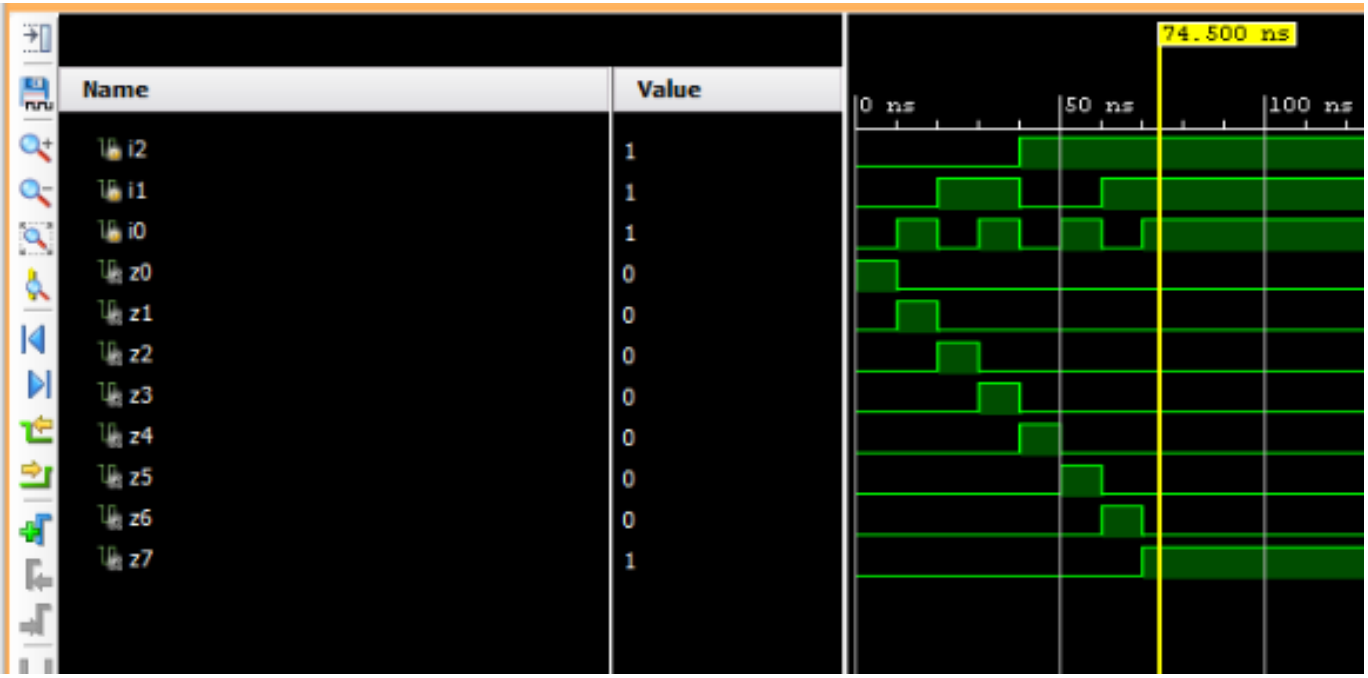


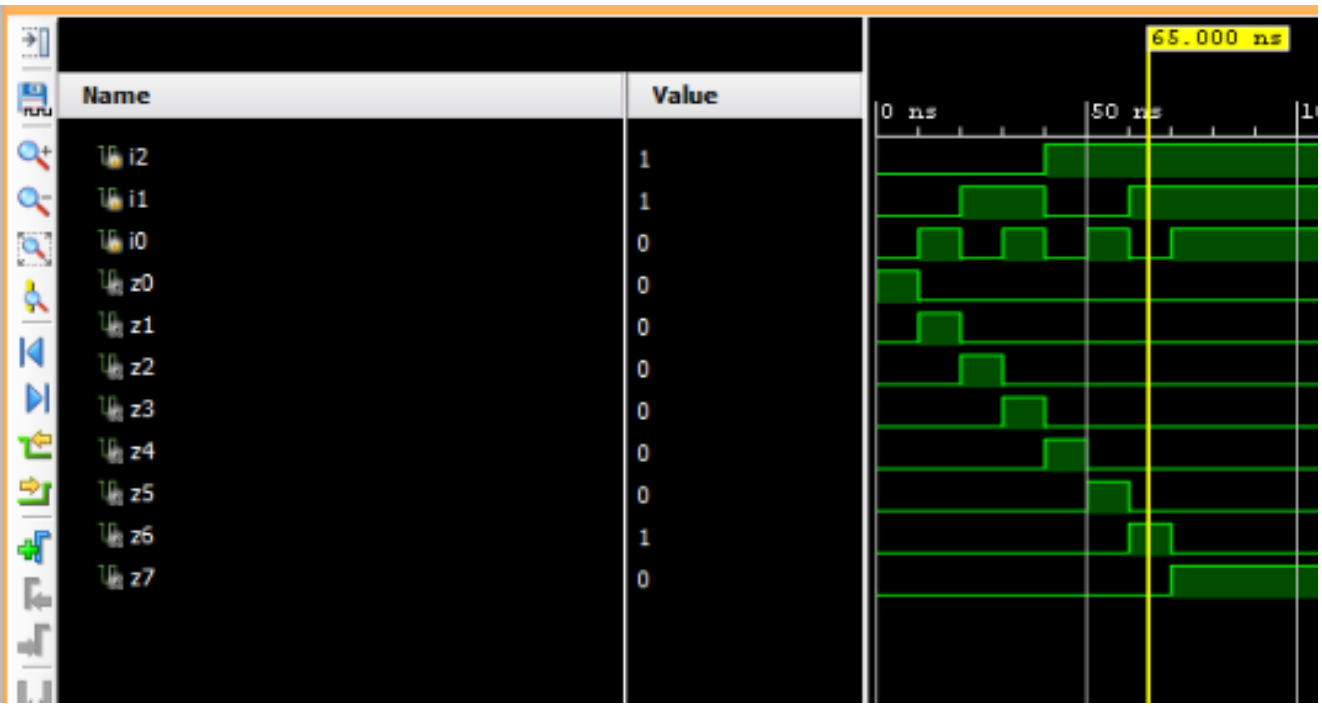

Name
Value
i2
0
i1
0
i0
1
z0
0
z1
1
z2
0
z3
0
z4
0
z5
0
z6
0
z7
0
15.000 ns
0 ns
50 ns

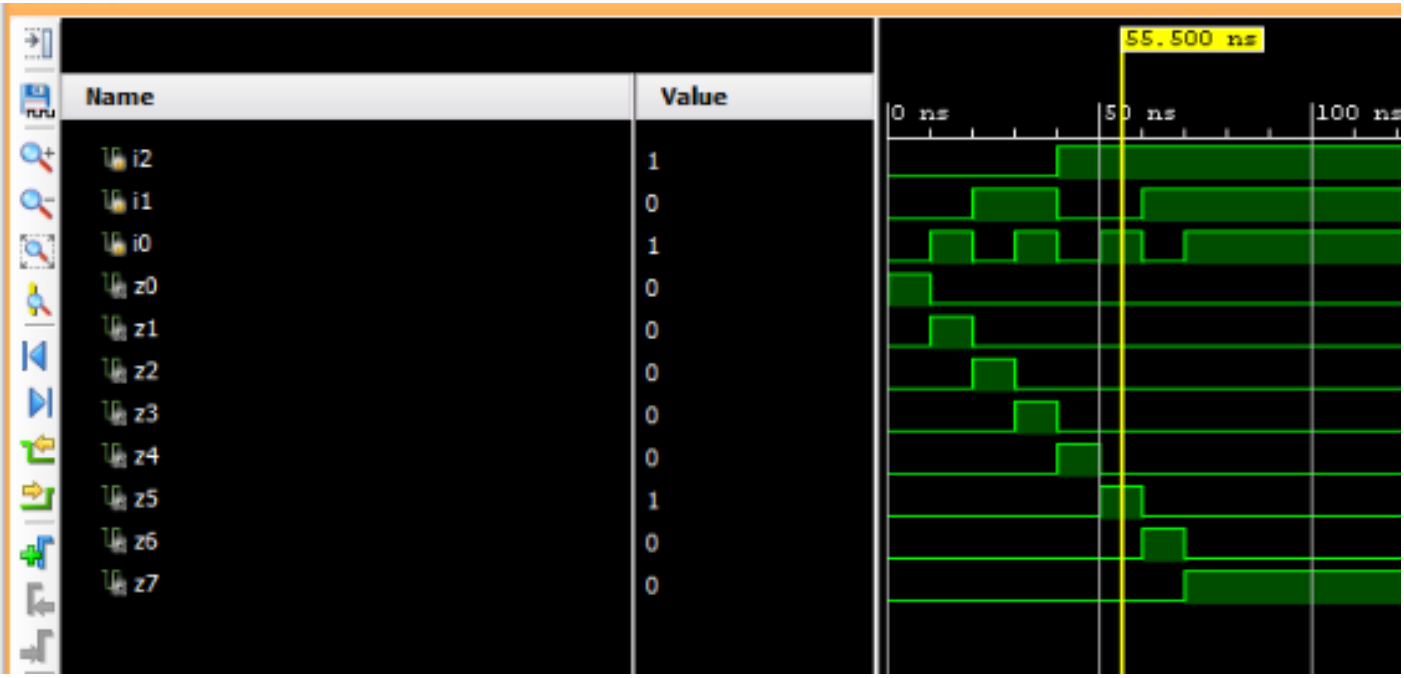

Name
Value
i2
0
i1
1
i0
0
z0
0
z1
0
z2
1
z3
0
z4
0
z5
0
z6
0
z7
0
25.500 ns
0 ns
50 ns


Name
Value
i2
i1
i0
z0
z1
z2
z3
z4
z5
z6
z7
0
1
1
0
0
0
1
0
0
0
36.500 ns
0 ns
50 ns
100 n


Name
Value
i2
i1
i0
z0
z1
z2
z3
z4
z5
z6
z7
1
0
0
0
0
0
0
1
0
0
0
43.500 ns
0 ns
50 ns
100 ns


Name
Value
i2
1
i1
1
i0
1
z0
0
z1
0
z2
0
z3
0
z4
0
z5
0
z6
0
z7
1
74.500 ns
0 ns
50 ns
100 ns


Name
Value
i2
1
i1
1
i0
0
z0
0
z1
0
z2
0
z3
0
z4
0
z5
0
z6
1
z7
0
65.000 ns
0 ns
50 ns


Name
Value
i2
1
i1
0
i0
1
z0
0
z1
0
z2
0
z3
0
z4
0
z5
1
z6
0
z7
0
55.500 ns
0 ns
50 ns
100 ns

ECE1003 Experiment – 6 **19/04/2021**

Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

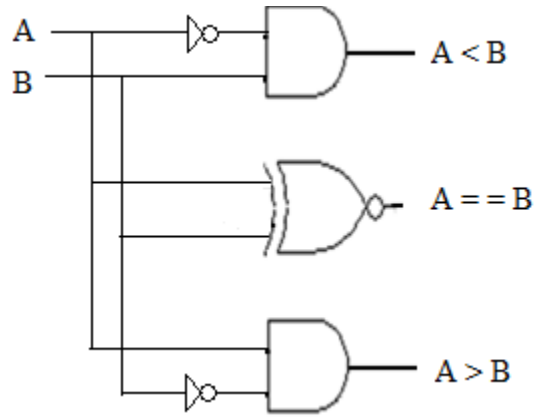
Aim : Design comparators in verilog
Software used : Xilinx Vivado 2014.2

I] 1-Bit Comparator :

1) Truth Table :

a	b	a > b	a = b	a < b
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

2) Circuit Diagram :



3) Structural:

```
module vitapstudents_1BitComparator;
input(a, b); output (lower, equal, greater);
wire s1, s2;
not x1(s1, a);
not x2(s2, b);
and x3(lower,s1, b);
and x4(greater,s2, a);
xnor x5(equal, a, b);
endmodule
```

4) Data Flow :

```
module comparator_1bit ( a ,b ,equal ,greater ,lower );

equal ;
output greater ;
output lower ;
input a ;
input b ;
assign equal = a ^(~b);
assign lower = (~a) & b;
assign greater = a & (~b);
```

5) Testbench:

```
module tb_1BitComparator(a, b, greater, equal, lower);

initial begin  // Initialize Inputs
a = 0;b = 0;   // Wait 100 ns to finish
#100;
#100; a=0;b=1;
#100; a=1;b=0;
#100; a=1;b=1;
end
endmodule
```

6) Behavioural :

```
module 1BitComparator_(a,b,equal,greater,lower);
input a, b;
output reg equal,greater,lower;
always @(a,b)
begin
if(a==b)
(equal,greater,lower)=3'b100;
else if(a>b)
(equal,greater,lower)=3'b010;
else if(a<b)
(equal,greater,lower)=3'b001;
else
(equal,greater,lower)=3'b000;
```

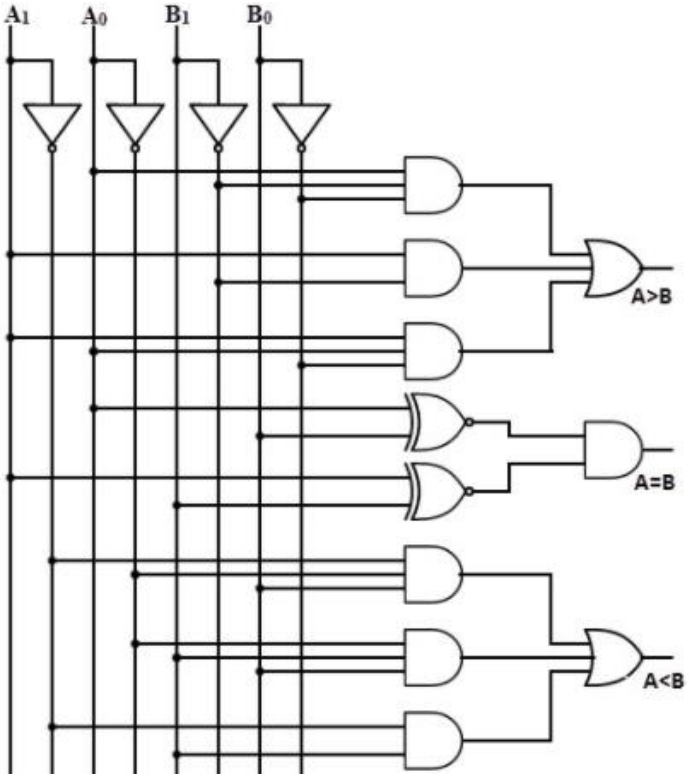
```
end
endmodule
```

II] 2-Bit Comparator :

1) Truth Table :

a1	a0	b1	b0	A<B	A=B	A>B
0	0	0	0	0	0	1
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	0	1	0	0	1
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	0	1	1	1	0	0
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	0	1	1

2) Circuit diagram:



3) Data Flow :

```
module 2BitComparator(a,b,x,y,z);
input(a,b);
output(x,y,z);
assign x=(~a)&b;
assign y=a&(~b);
assign z=(~a)&(~b)|(a&b);
end
endmodule
```

4) Gate level :

```
module 2BitComparator_(a0,a1,b0,b1,f0,f1,f2);
input a0,a1,b0,b1;
output f0,f1,f2;
wire x,y,u,v,p,q,r,j,k,c,f,g;
not(x,a0);
not(y,a1);
not(u,b0);
not(v,b1);
and(p,x,y,b0);
and(q,x,b0);
and(r,b0,b1,y);
or(f0,p,q,r);
and(j,a1,b1);
and(k,y,v);
or(f1,j,k);
and(c,a1,u,v);
and(f,a0,u);
and(g,v,x,y);
or(f2,c,f,g);
end
endmodule
```

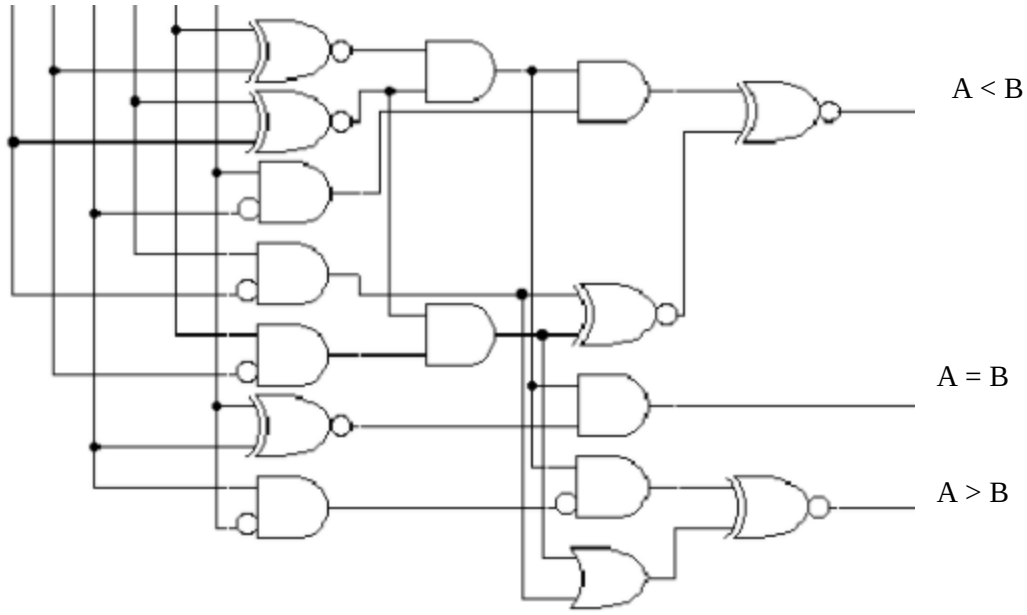
5) Testbench:

```
module 2BitComparator(a,b,x,y,x_greater_y, x_equal_y,
x_lesser_y);

 output x_greater_y, x_equal_y, x_lesser_y;

  input  a, b;

  input [1:0] a,b;

  assign x_greater_y = a | (~b & ((x[1] > y[1]) |
((x[1] == y[1]) & (x[0] > y[0]))));

  assign x_lesser_y = b | (~b & ((x[1] < y[1]) | ((x[1]
== y[1]) & (x[0] < y[0]))));

  assign x_equal_y = ~(b | a);

endmodule
```

III] 4-Bit Comparator

1) Circuit Diagram :

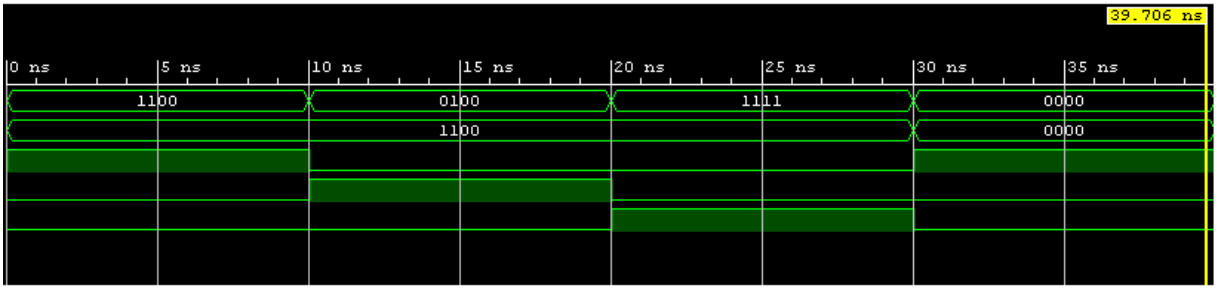
A2 A1 A0 B2 B1 B0

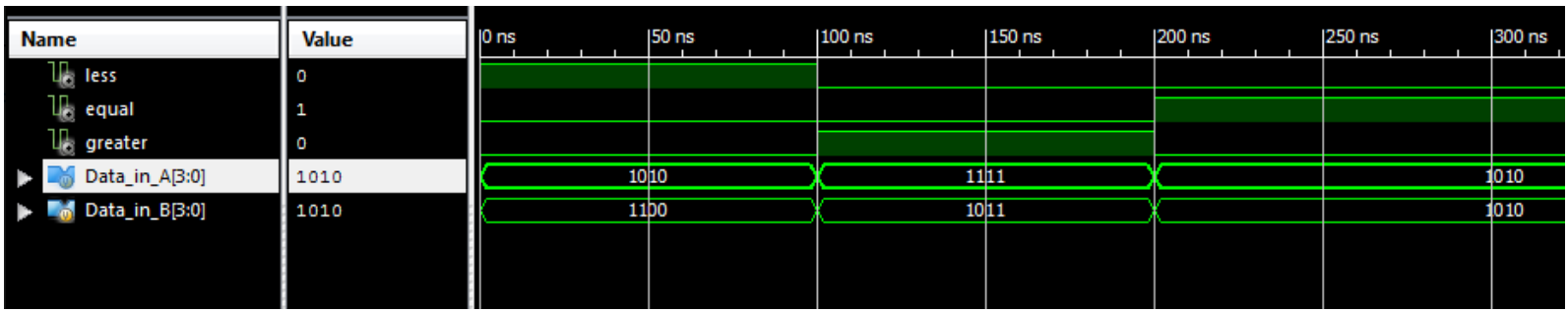


2) Testbench :

```
module tb_4BitComparator (a,b,equal,lower,greater);

reg [3:0] a,b;
wire equal,lower,greater;

input a,b;
output equal,lower,greater;

initial
begin
a = 4'b1100;
b = 4'b1100;
#10;

a = 4'b0100;
b = 4'b1100;
#10;

a = 4'b1111;
b = 4'b1100;
#10;

a = 4'b0000;
b = 4'b0000;
#10;

end
endmodule
```

OUTPUT (Testbench code) :





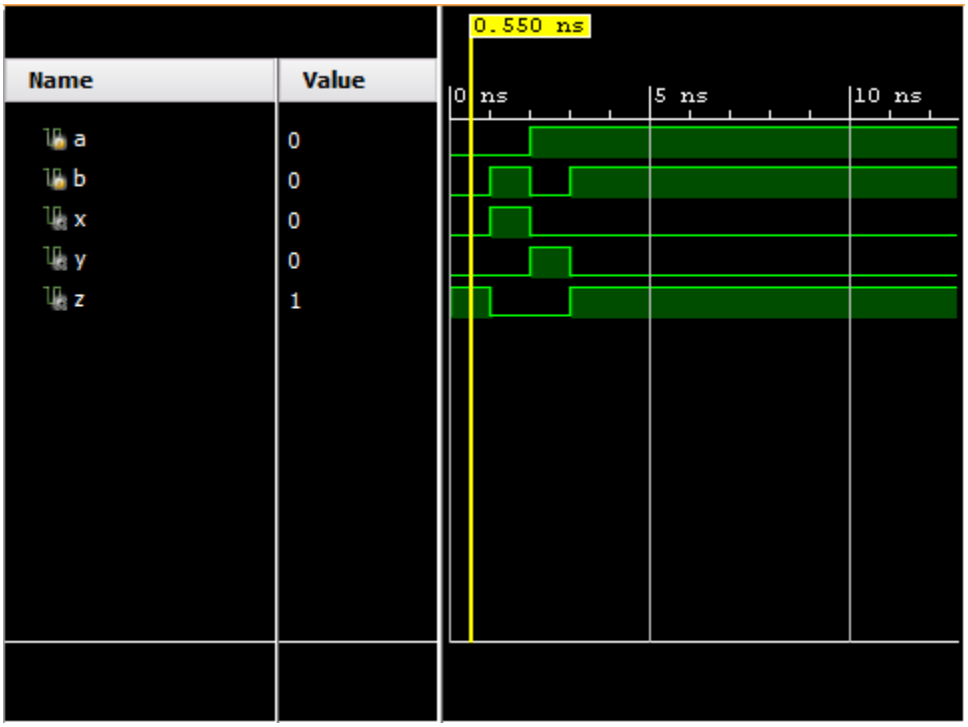
```
module tb_4BitComparator(a, b, x_greater_y, x_equal_y,
x_lower_y, greater, equal,lower);

input [3:0] a;
input [3:0] b;
input greater = 2'b00;
input equal = 2'b01;
input lower = 2'b00;
output x, y, z;
wire x, y, z;

assign x_greater_y = ~(a[3] & ~b[3] | ~a[3] & b[3]) & a[2] &
~b[2] | a[3] & ~b[3];
assign x_equal_y = ~(a[2] & ~b[2] | ~a[2] & b[2]) & ~(a[3] &
~b[3] | ~a[3] & b[3]);
assign x_lesser_y = ~(a[3] & ~b[3] | ~a[3] & b[3]) & ~a[2] &
b[2] | ~a[3] & b[3];

endmodule
```

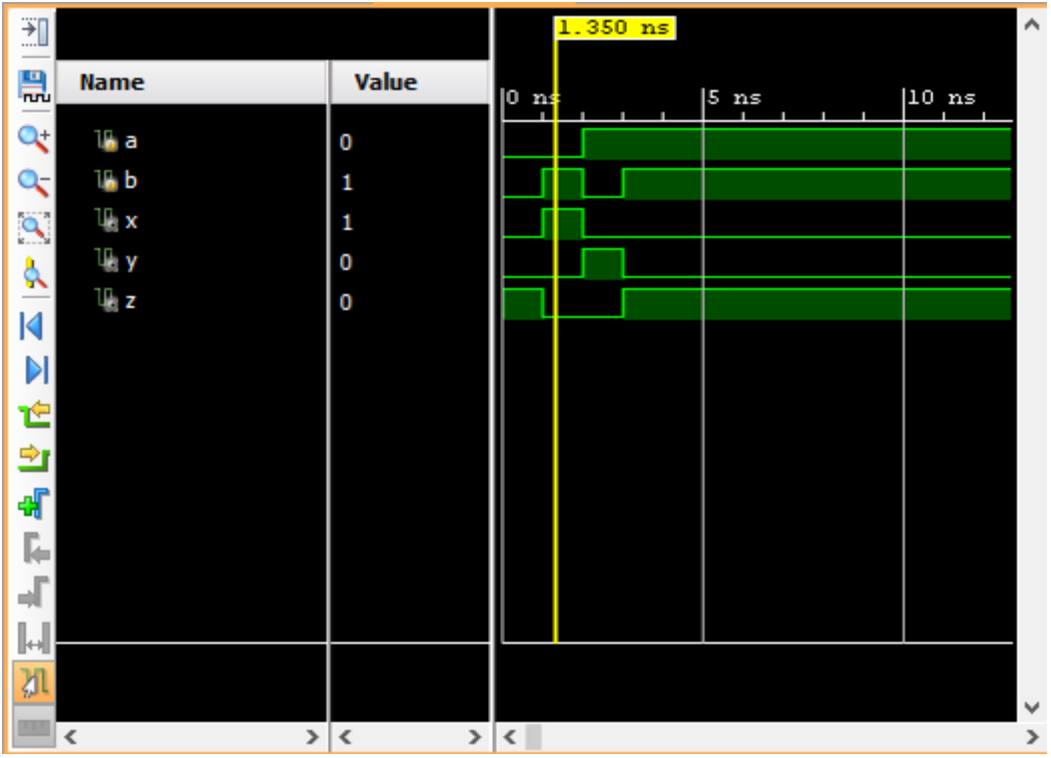
3) Behavioural :

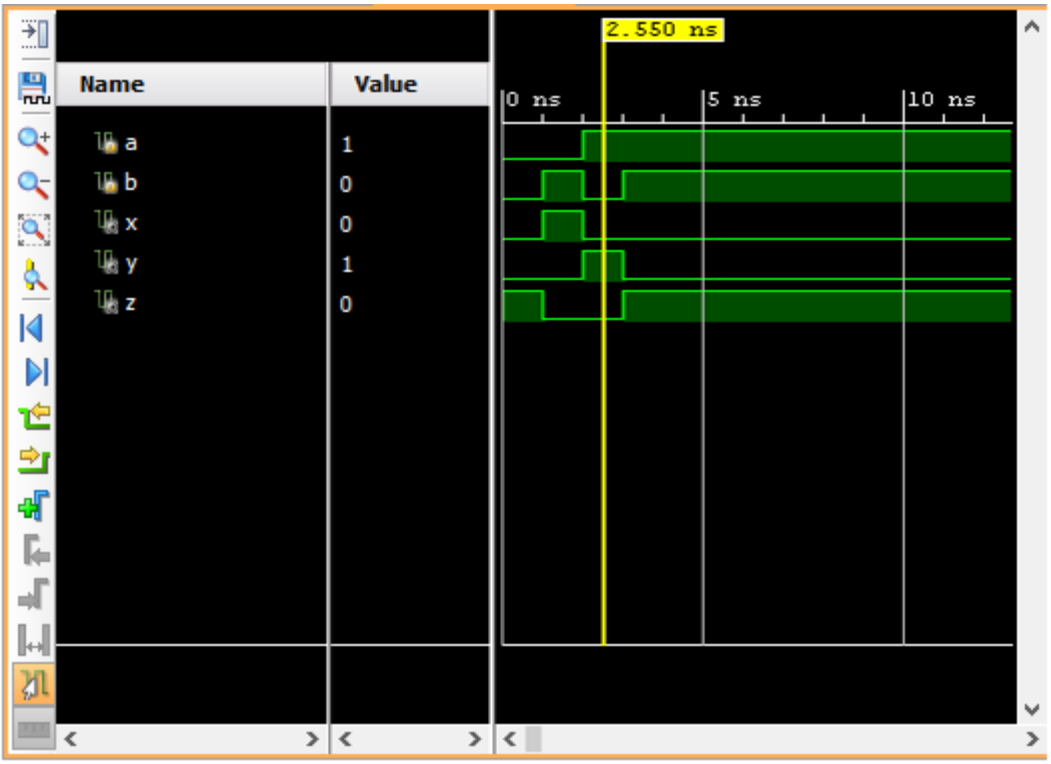
```
module 4BitComparator(a,b,equal,lesser,greater)

input [3:0] a;
input [3:0] b;
output less;
output equal;
output greater;

reg lesser;
reg equal;
reg greater;

always @(a or b)
begin
if(a > b)
begin            //(a > b)
less = 0;
equal = 0;
greater = 1;
```

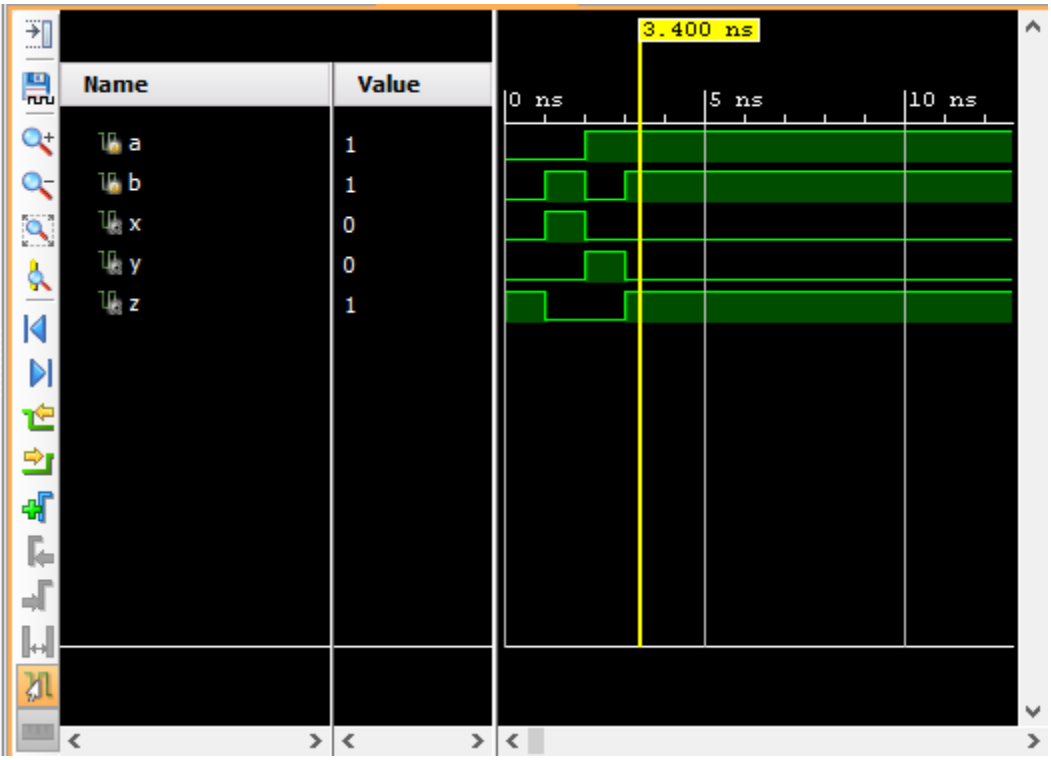
```
else if(a == b)
begin            //(a==b)
less = 0;
equal = 1;
greater = 0;

else
begin                //(a < b)
less = 1;
equal = 0;
greater =0;

end
endmodule
```

OUTPUT (1 Bit Comparator) :

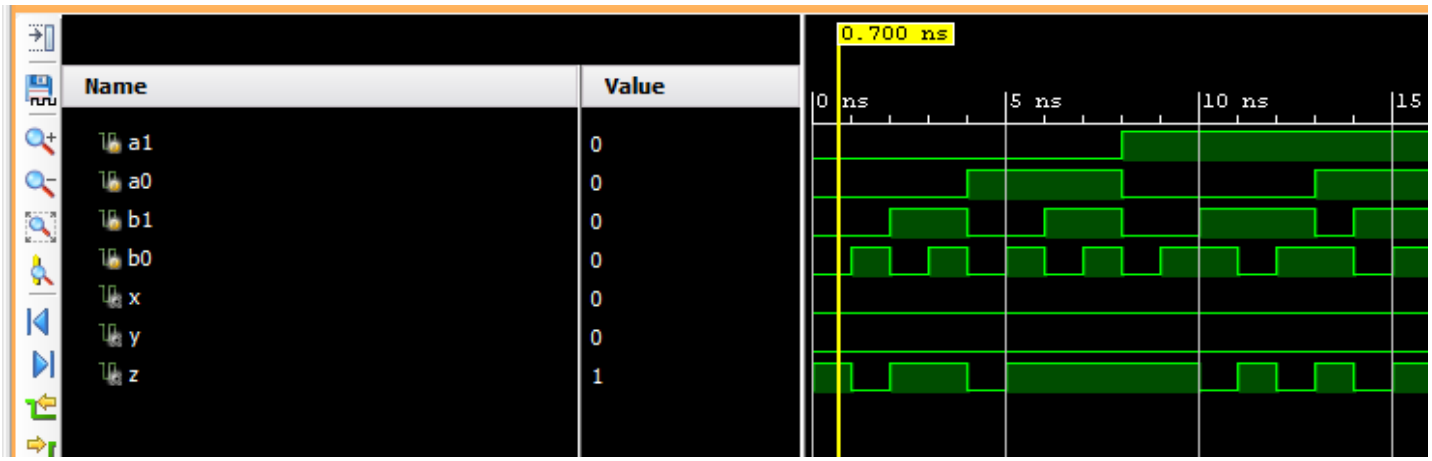


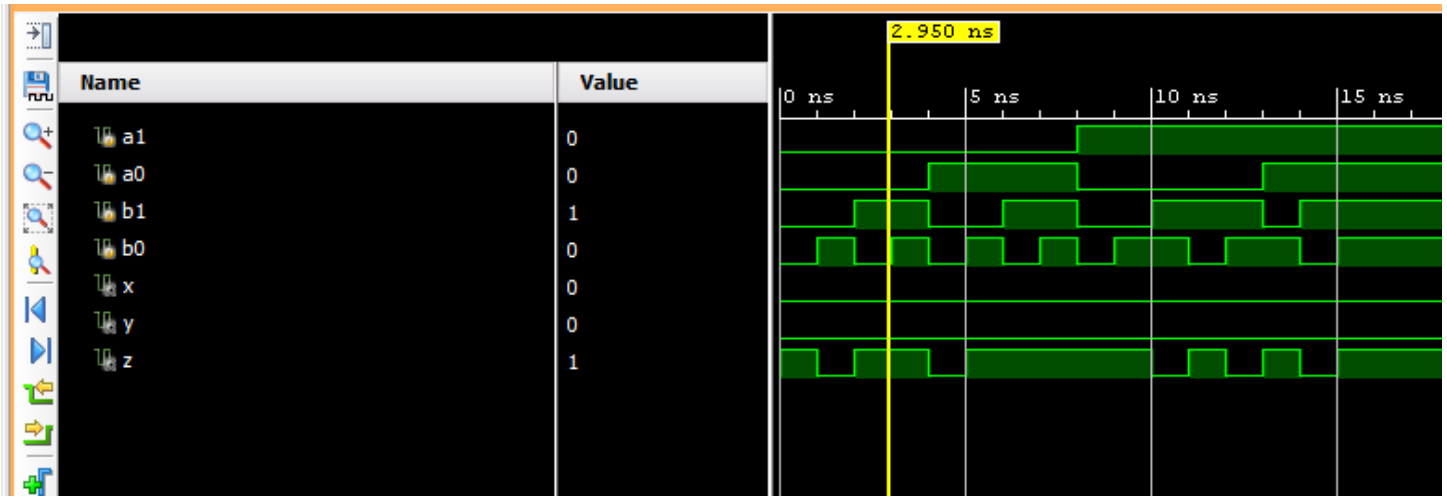

Name
Value
a
0
b
1
x
1
y
0
z
0
1.350 ns
0 ns
5 ns
10 ns

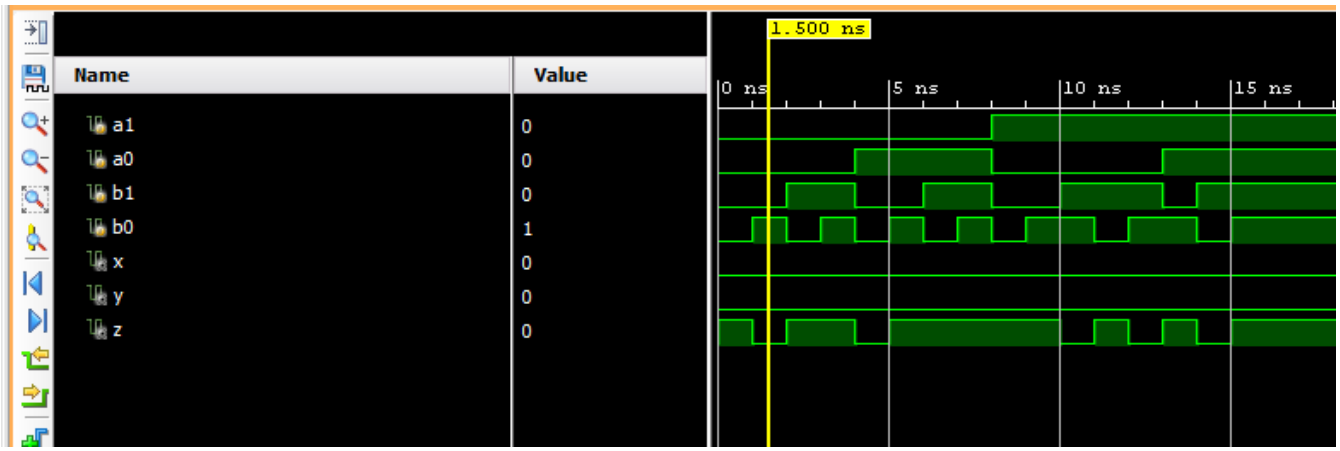

Name
Value
a
1
b
0
x
0
y
1
z
0
2.550 ns
0 ns
5 ns
10 ns

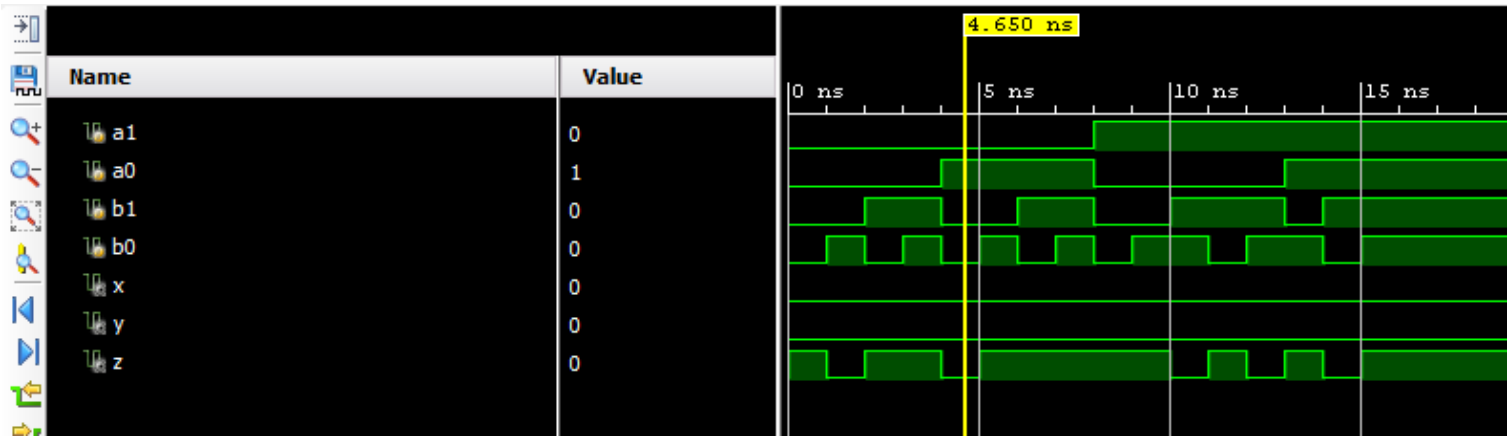


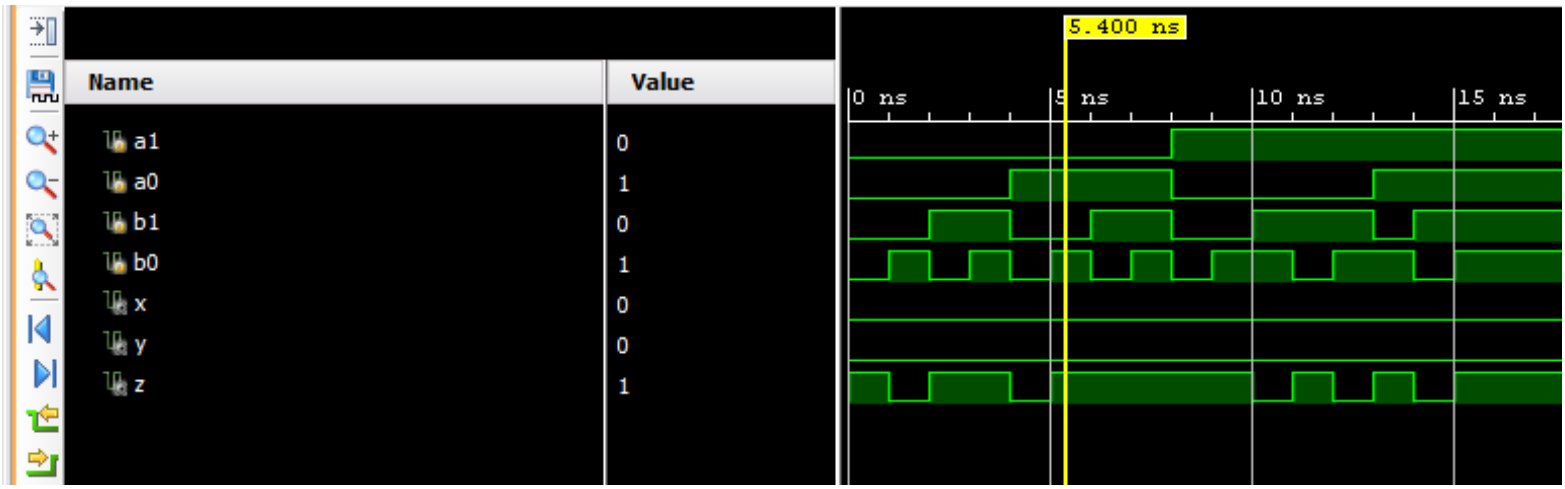
OUTPUT (2 Bit Comparator) :



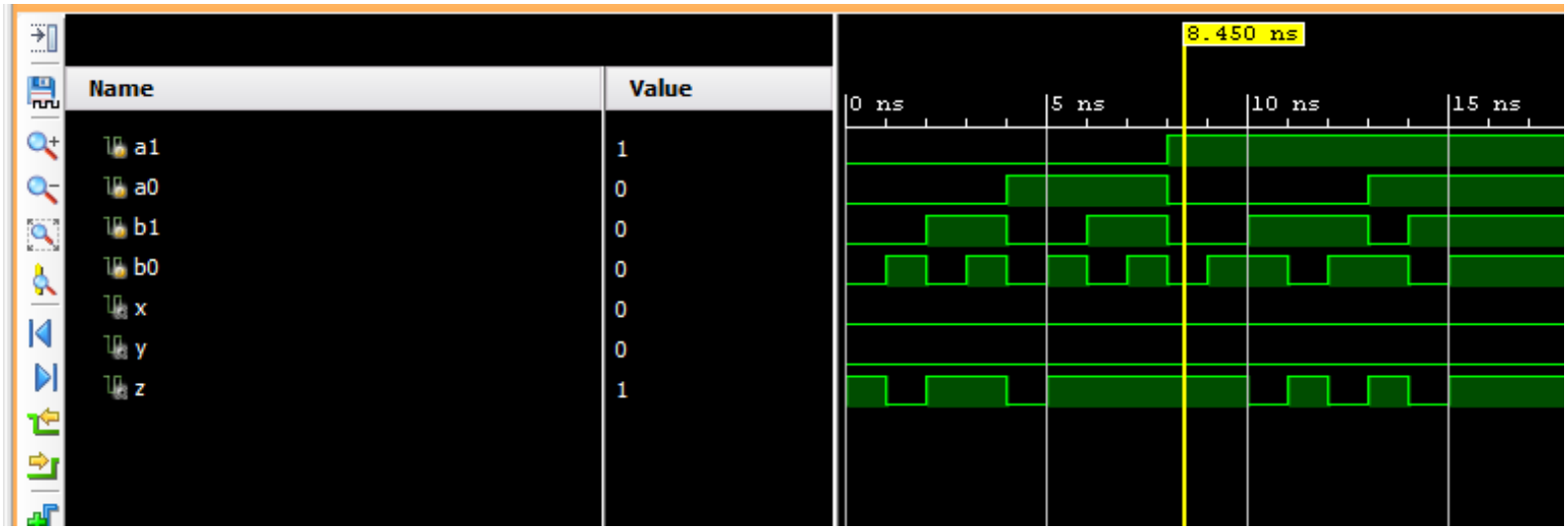


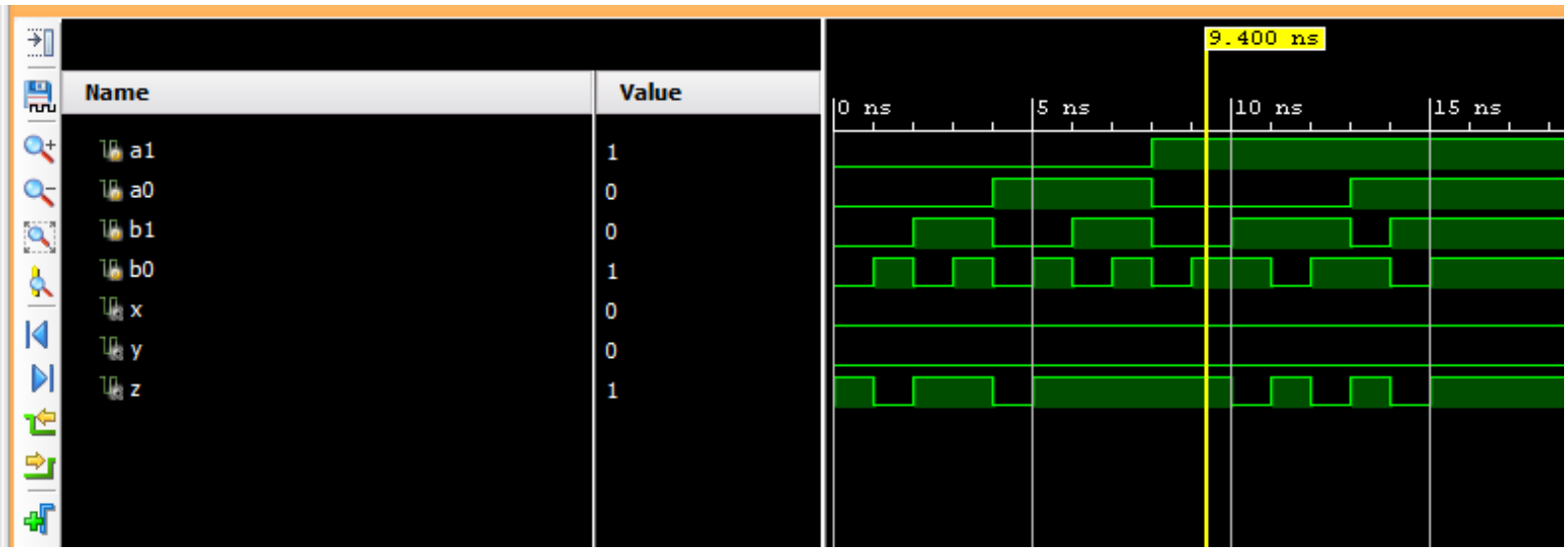

Name
Value
a1
a0
b1
b0
x
y
z
0
0
0
1
0
0
0
1.500 ns
0 ns
5 ns
10 ns
15 ns

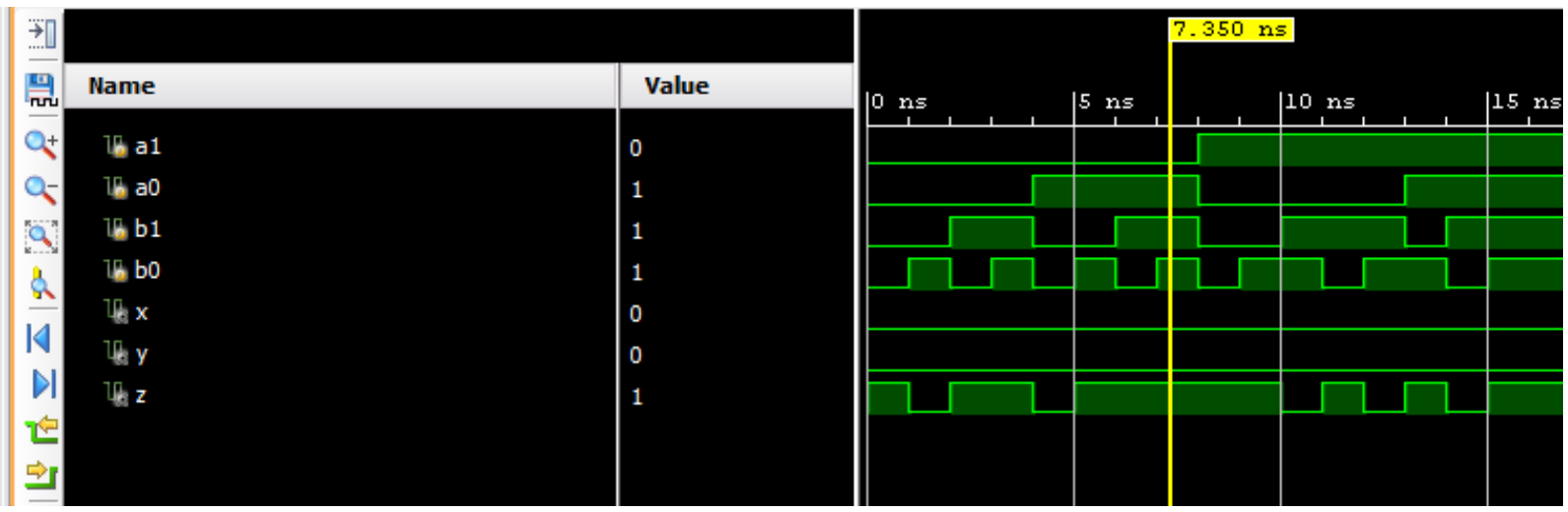

Name
Value
a1
a0
b1
b0
x
y
z
0
1
0
0
0
0
0
4.650 ns
0 ns
5 ns
10 ns
15 ns

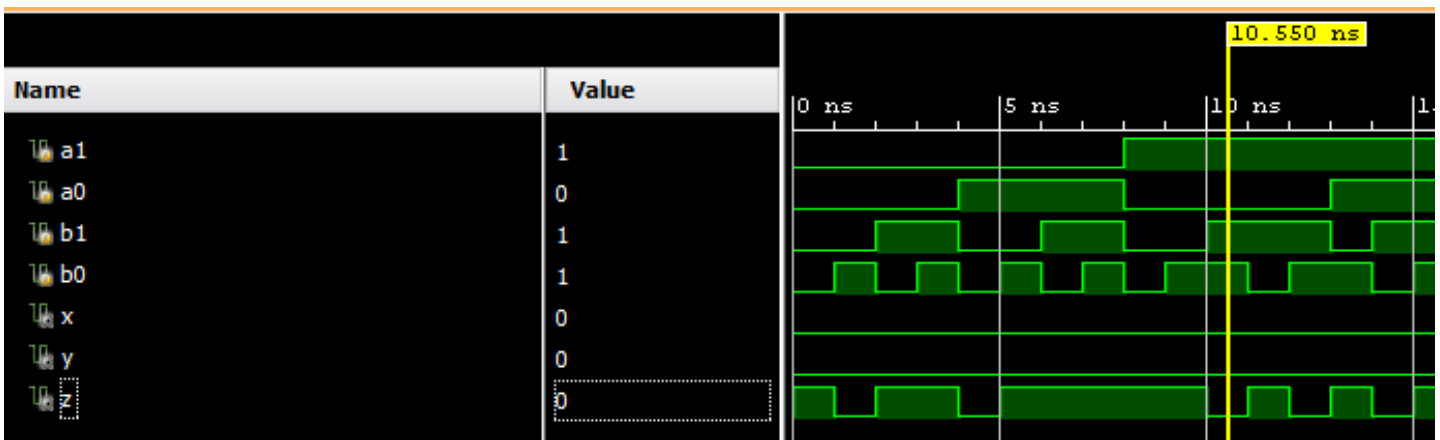

Name
Value
a1
a0
b1
b0
x
y
z
0
1
0
1
0
0
1
5.400 ns
0 ns
10 ns
15 ns

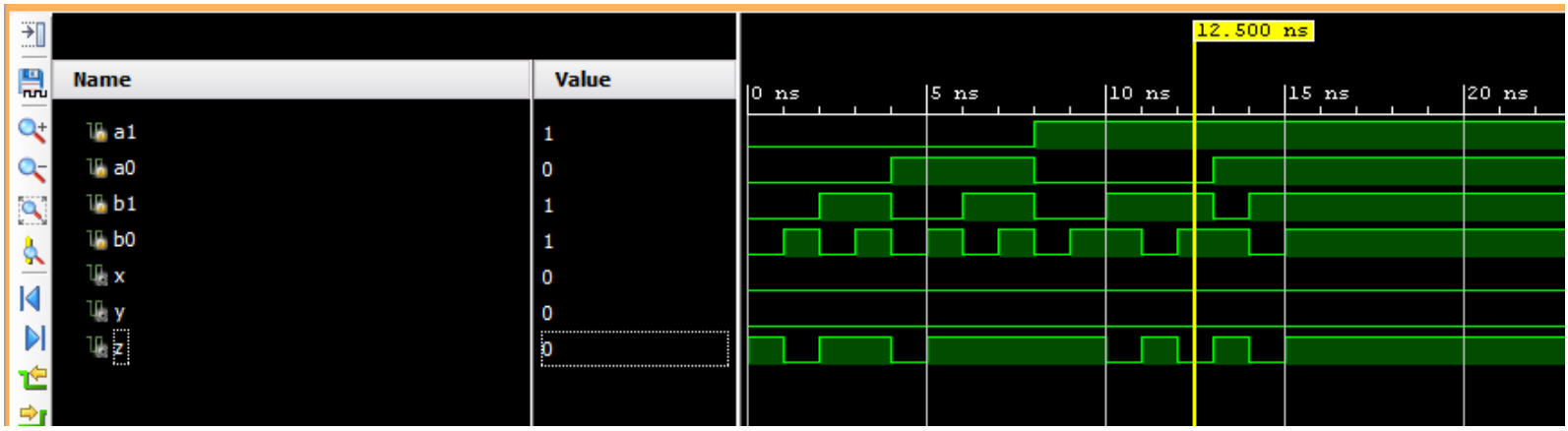

Name
Value
a1
a0
b1
b0
x
y
z
0
1
1
0
0
0
1
6.400 ns
0 ns
10 ns
15 ns


Name
Value
a1
1
a0
0
b1
0
b0
0
x
0
y
0
z
1
8.450 ns
0 ns
5 ns
10 ns
15 ns

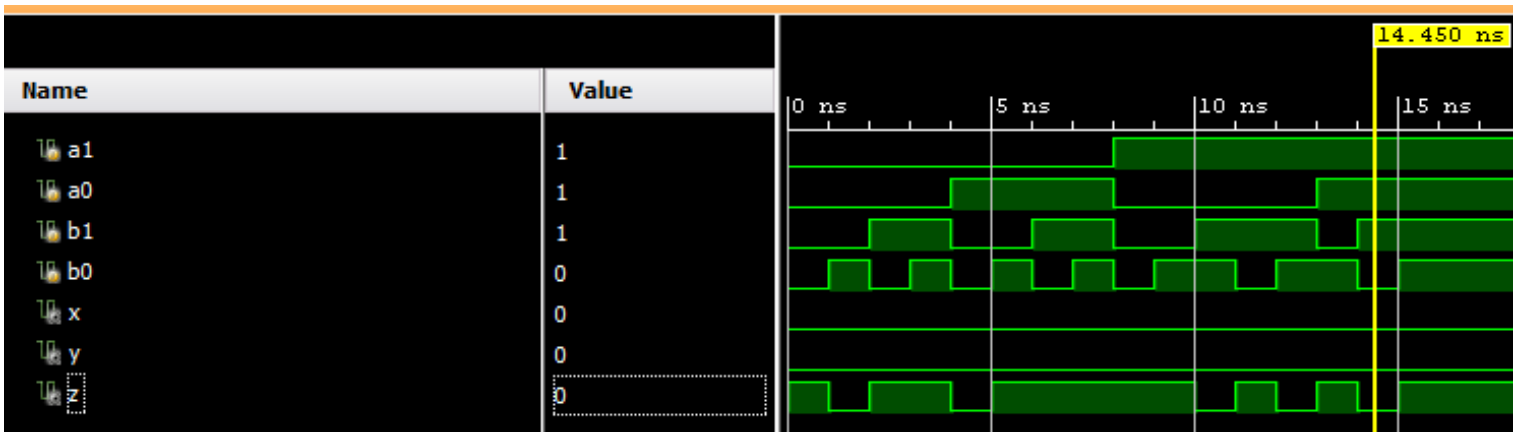

Name
Value
a1
1
a0
0
b1
0
b0
1
x
0
y
0
z
1
9.400 ns
0 ns
5 ns
10 ns
15 ns

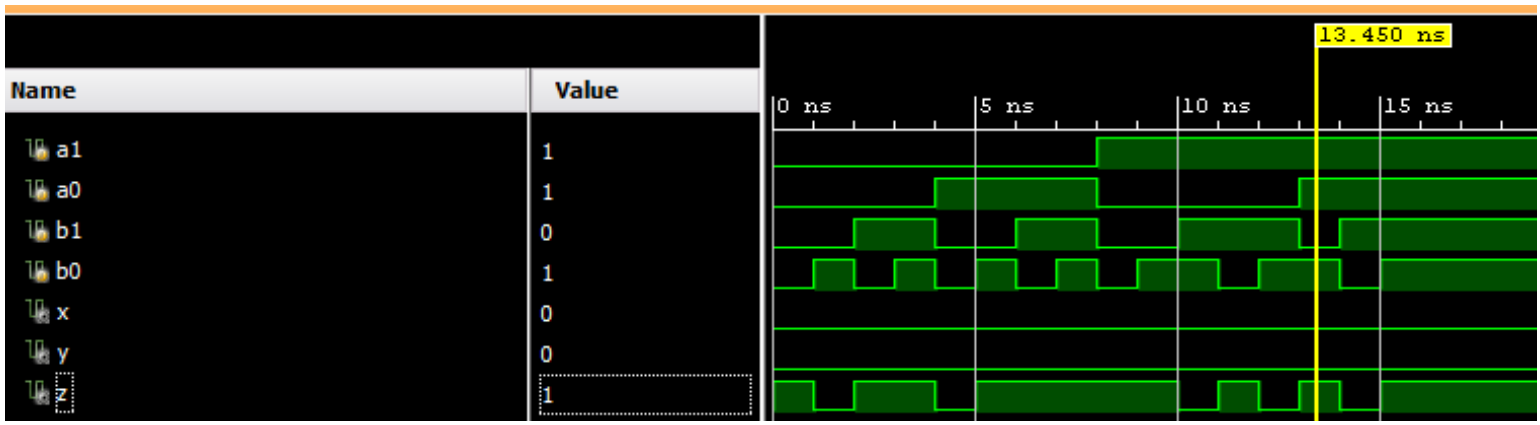

Name
Value
a1
0
a0
1
b1
1
b0
1
x
0
y
0
z
1
7.350 ns
0 ns
5 ns
10 ns
15 ns

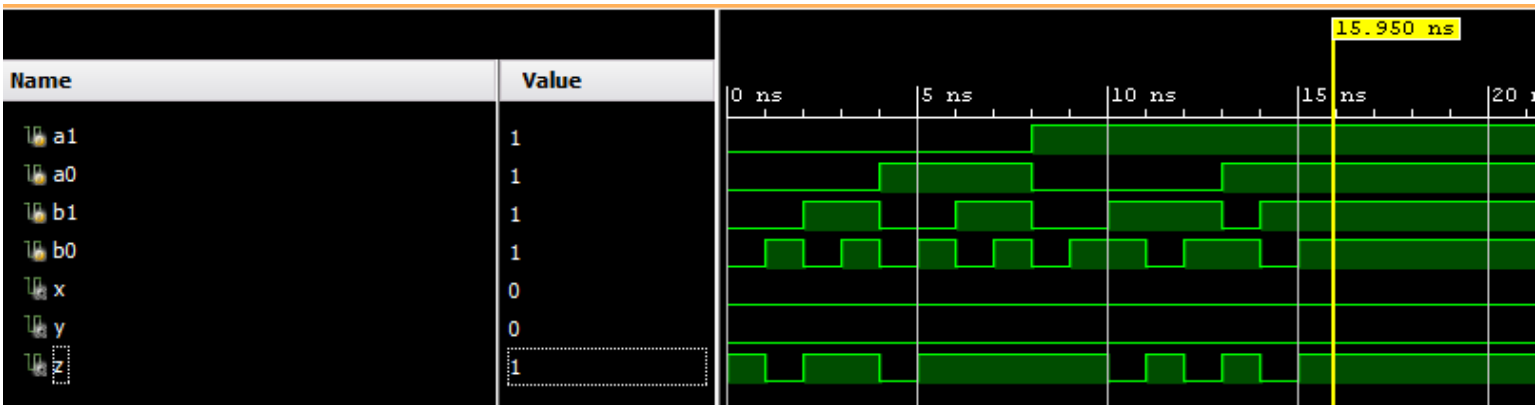

Name
Value
a1
1
a0
0
b1
1
b0
1
x
0
y
0
z
0
10.550 ns
0 ns
5 ns
10 ns


Name
Value
a1
a0
b1
b0
x
y
z
1
0
1
1
0
0
0
12.500 ns
0 ns
5 ns
10 ns
15 ns
20 ns

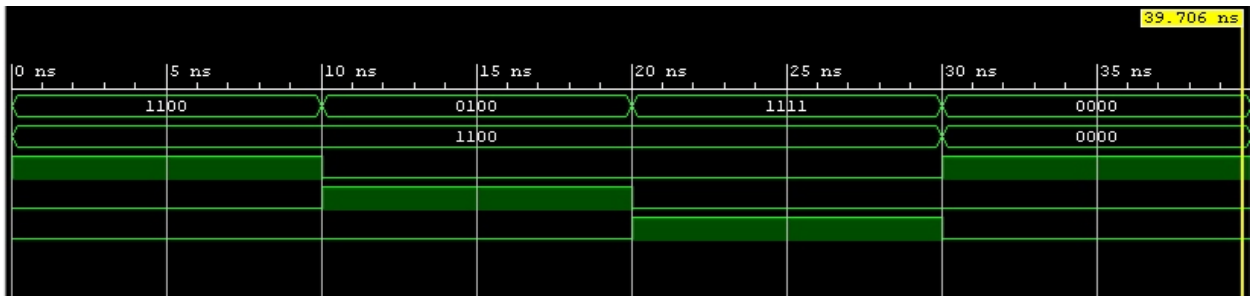

Name
Value
a1
a0
b1
b0
x
y
z
1
0
1
0
0
0
1
11.600 ns
0 ns
5 ns
10 ns
15 ns


Name
Value
a1
a0
b1
b0
x
y
z
1
1
1
0
0
0
0
14.450 ns
0 ns
5 ns
10 ns
15 ns


Name
Value
a1
a0
b1
b0
x
y
z
1
1
0
1
0
0
1
13.450 ns
0 ns
5 ns
10 ns
15 ns



OUTPUT (4 Bit Compartor) :



4 Bit Comparator and Output (Using EDA playground) :

```
module ECE1003_4BitComp_tb;
  reg x;
  reg y;
  wire e;
  wire g ;
  wire l;

  comparator inst(.x(x),
.y(y), .e(e), .g(g) , .l(l));
   initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #400 $finish;
  end
  initial
    begin
      x = 0 ;
      y = 0 ;
      #100 ;
      x = 0 ;
      y = 1 ;
      #100 ;
      x = 1 ;
      y = 0 ;
      #100 ;
      x = 1 ;
      y = 1 ;
      #100 ;
  end
endmodule
```

ECE1003_Digital Logic Design_Experiment No. 7

Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

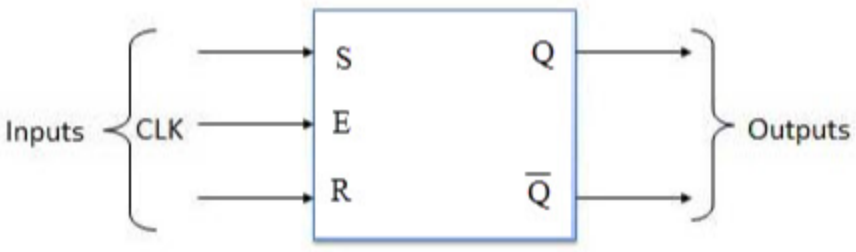
Aim : Design and verify Flip Flops using verilog

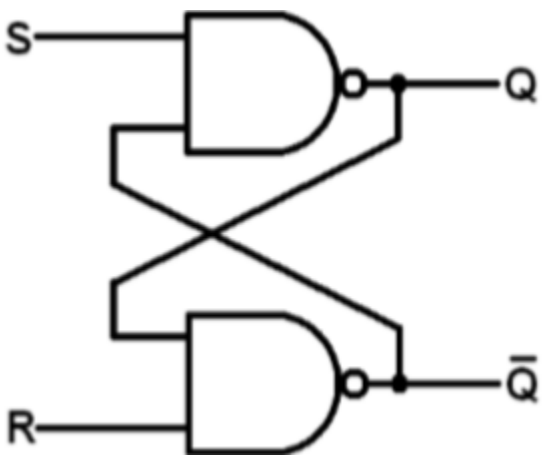
Theory :

SR Flip Flop :- The SR flip flop is a 1-bit memory bistable device having two inputs, i.e., SET and RESET. The SET input 'S' set the device or produce the output 1, and the RESET input 'R' reset the device or produce the output 0. The SET and RESET inputs are labeled as **S** and **R**, respectively. The SR flip flop stands for "Set-Reset" flip flop. The reset input is used to get back the flip flop to its original state from the current state with an output 'Q'. This output depends on the set and reset conditions, which is either at the logic level "0" or "1". The NAND gate SR flip flop is a basic flip flop which provides feedback from both of its outputs back to its opposing input. This circuit is used to store the single data bit in the memory circuit. So, the SR flip flop has a total of three inputs, i.e., 'S' and 'R', and current output 'Q'. This output 'Q' is related to the current history or state. The term "flip-flop" relates to the actual operation of the device, as it can be "flipped" to a logic set state or "flopped" back to the opposing logic reset state.

S	R	Q	Q'	Description
1	0	0	1	Set Q'>>1
1	1	0	1	No change
0	1	1	0	Reset Q'>>0
1	1	1	0	No change
0	0	1	1	Invalid Condition

Circuit & Block diagram :-



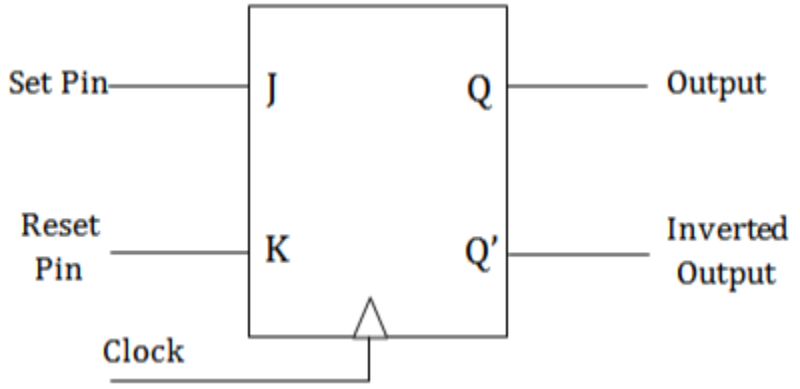


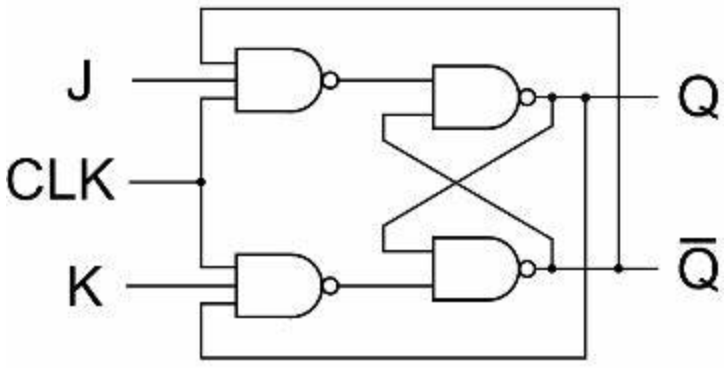
JK Flip Flop :- The JK flip flop is one of the most used flip flops in digital circuits. The JK flip flop is a universal flip flop having two inputs 'J' and 'K'. In SR flip flop, the 'S' and 'R' are the shortened abbreviated letters for Set and Reset, but J and K are not. The J and K are themselves autonomous letters which are chosen to distinguish the flip flop design from other types.

The JK flip flop work in the same way as the SR flip flop work. The JK flip flop has 'J' and 'K' flip flop instead of 'S' and 'R'. The only difference between JK flip flop and SR flip flop is that when both inputs of SR flip flop is set to 1, the circuit produces the invalid states as outputs, but in case of JK flip flop, there are no invalid states even if both 'J' and 'K' flip flops are set to 1.

The JK Flip Flop is a gated SR flip-flop having the addition of a clock input circuitry. The invalid or illegal output condition occurs when both of the inputs are set to 1 and are prevented by the addition of a clock input circuit. So, the JK flip-flop has four possible input combinations, i.e., 1, 0, "no change" and "toggle". The symbol of JK flip flop is the same as SR Bistable Latch except for the addition of a clock input.

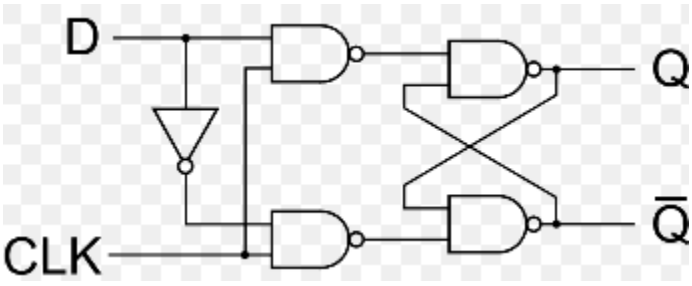
Input		Output		Description
J	K	Q	Q'	
0	0	1	0	Memory no change
0	0	0	1	
0	1	1	0	Reset Q>>0
0	1	0	1	
1	0	0	1	Set Q>>1
1	0	1	0	
1	1	0	1	Toggle
1	1	1	0	





D Flip Flop :- The D flip flop is the most important flip flop from other clocked types. It ensures that at the same time, both the inputs, i.e., S and R, are never equal to 1. The Delay flip-flop is designed using a gated SR flip-flop with an inverter connected between the inputs allowing for a single input D(Data).
This single data input, which is labeled as "D" used in place of the "Set" input and for the complementary "Reset" input, the inverter is used. Thus, the level-sensitive D-type or D flip flop is constructed from a level-sensitive SR flip flop.
In D flip flop, the single input "D" is referred to as the "Data" input. When the data input is set to 1, the flip flop would be set, and when it is set to 0, the flip flop would change and become reset. However, this would be pointless since the output of the flip flop would always change on every pulse applied to this data input.
The "CLOCK" or "ENABLE" input is used to avoid this for isolating the data input from the flip flop's latching circuitry. When the clock input is set to true, the D input condition is only copied to the output Q. This forms the basis of another sequential device referred to as D Flip Flop**.**
When the clock input is set to 1, the "set" and "reset" inputs of the flip-flop are both set to 1. So it will not change the state and store the data present on its output before the clock transition occurred. In simple words, the output is "latched" at either 0 or 1.

D	Q	Q'	Description
X	Q	Q'	Memory no change
0	0	1	Reset Q » 0
1	1	0	Set Q » 1

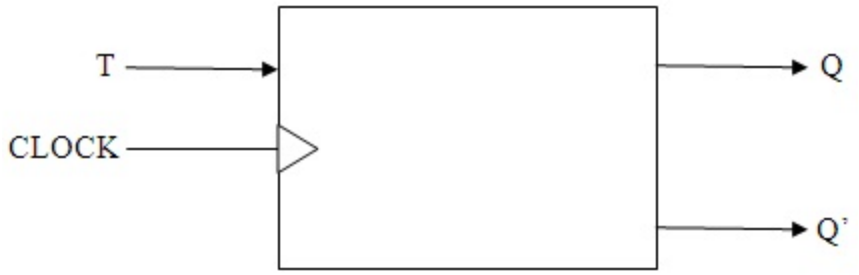


T Flip Flop :- A T flip flop is known as a toggle flip flop because of its toggling operation. It is a modified form of the JK flip flop. A T flip flop is constructed by connecting J and K inputs, creating a single input called T. Hence why a T flip flop is also known as a single input JK flip flop.
The defining characteristic of T flip flop is that it can change its output state. You can change the output signal from one state (on/off) to another state (off/on).
The clock signal must set high to toggle the output. When the clock is set low, the output remains as it is whether the input signal is set high or low. So, to change the output condition, the clock signal has to be high.

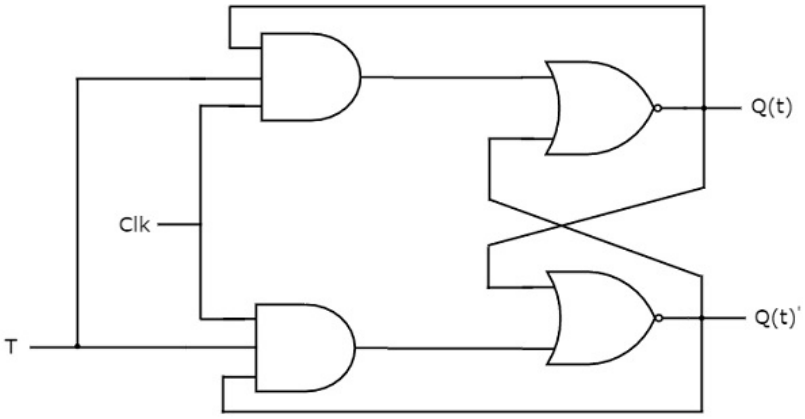
	Previous		Next	
T	Q	Q'	Q	Q'
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

The "T Flip Flop" is toggled when the set and reset inputs alternatively changed by the incoming trigger. The "T Flip Flop" requires two triggers to complete a full cycle of the output waveform. The frequency of the output produced by the "T Flip Flop" is half of the input frequency. The "T Flip Flop" works as the "Frequency Divider Circuit." In "T Flip Flop", the state at an applied trigger pulse is defined only when the previous state is defined. It is the main drawback of the "T Flip Flop". The "T flip flop" can be designed from "JK Flip Flop", "SR Flip Flop", and "D Flip Flop" because the "T Flip Flop" is not available as ICs.

The block diagram of "T Flip Flop" is given below:



The circuit diagram of "T Flip Flop" is given below:



Boolean Expressions :

SR Flip Flop :- $Q_{n+1} = (SR + SR') (Q_n + Q'_n) + Q_n (S'R' + SR') = S + Q_nR'$

JK Flip Flop :- $Q_{n+1} = Q'_n (JK + JK') + Q_n (J'K' + JK') = Q'_nJ + Q_nK'$

D Flip Flop :- $Q(n+1) = D$

T Flip Flop :- $D = T'Q_n + TQ'_n$

ECE1003 Experiment – 7 **26/04/2021**

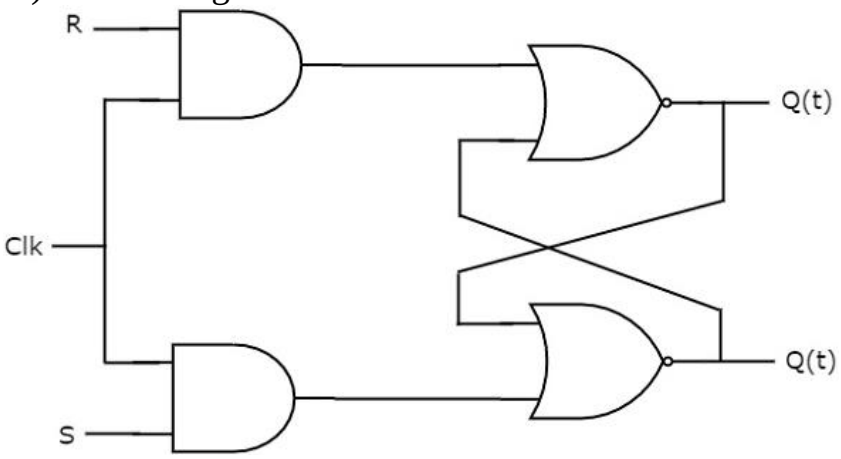
Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

Aim : Design and verify flip flops in verilog

Software used : Xilinx Vivado 2014.2

I] S-R flip flop

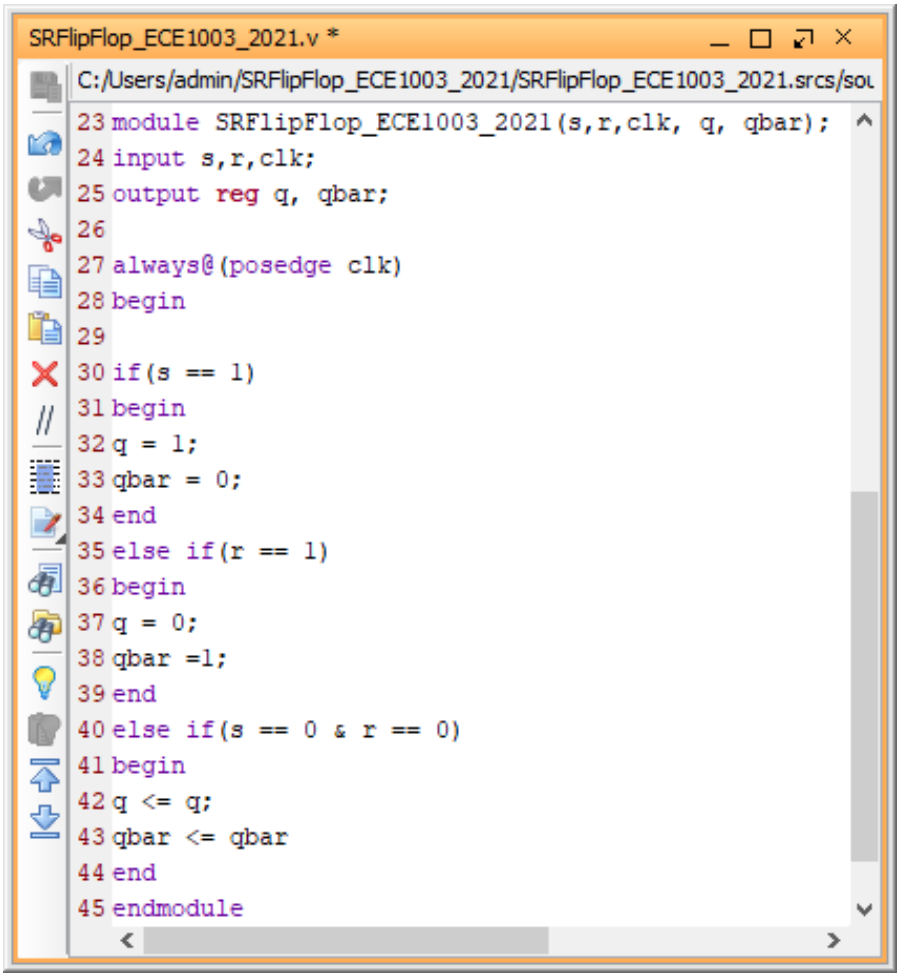
1.) Circuit diagram :



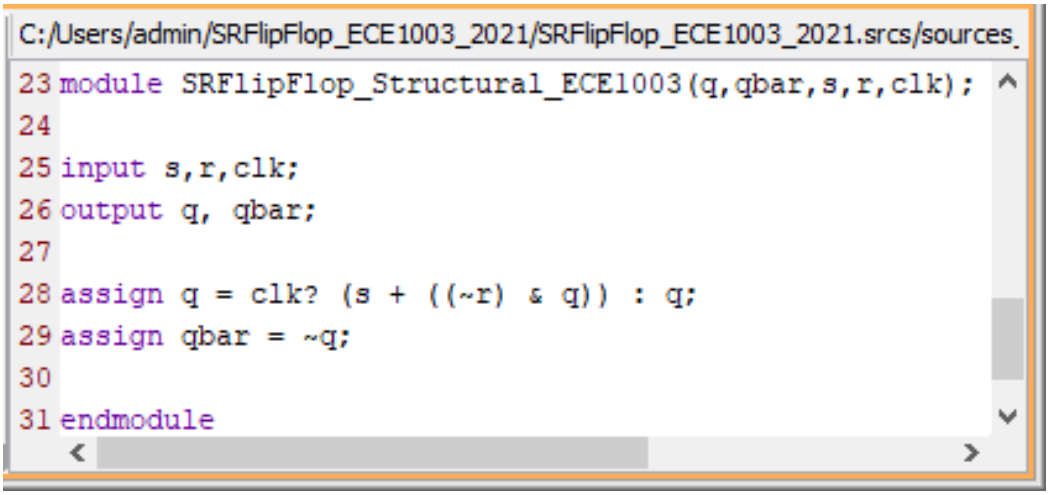
2.) Truth table :

S	R	Q(n)	Q(n+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

3.) Behavioural Modelling :



4.) Data Flow Modelling :



5.) Testbench code :

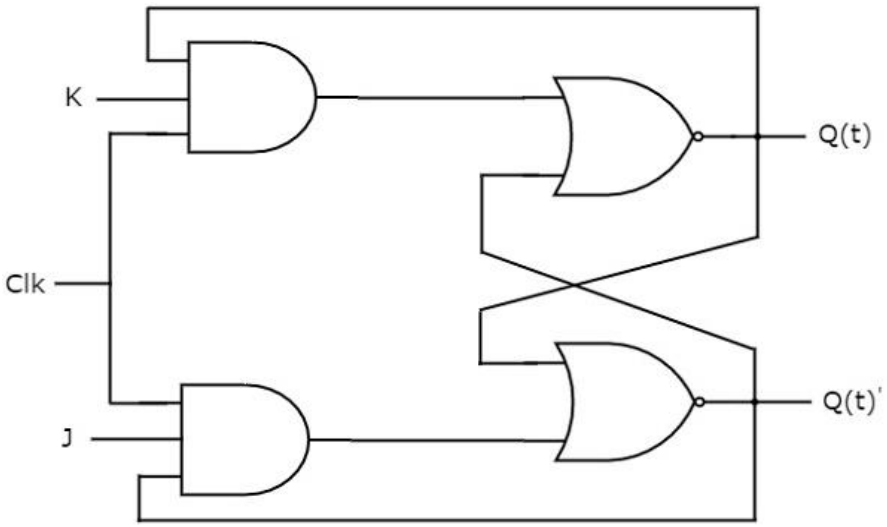
SRFlipFlop_testbench_ECE1003.v

C:/Users/admin/SRFlipFlop_ECE1003_2021/SRFlipFlop_ECE1003_2021.srcs/sim_1/new/SRFlipFlop_testbench_ECE1003.v

```
23 module SRFlipFlop_testbench_ECE1003;
24 reg S,R, CLK;
25 wire Q, QBAR;
26
27 srff_behavior dut(.q(Q), .qbar(QBAR), .s(S), .r(R), .clk(CLK));
28
29 $monitor("simtime = %g, CLK = %b, S = %b, R = %b, Q = %b, QBAR = %b", $time, CLK, S, R, Q, QBAR);
30
31 initial begin
32   clk=0;
33      forever #10 clk = ~clk;
34 end
35 initial begin
36  S=1; R=0;
37  #100; S=0; R=1;
38  #100; S=0; R=0;
39  #100; S=1; R=1;
40 end
41 endmodule
```

II] J-K Flip Flop

1.) Circuit diagram :



2.) Truth Table :

Clk	J	K	Q(n+1)
0	X	X	Q(n)
1	0	0	Q(n)
1	0	1	0
1	1	0	1
1	1	1	Q(n)

3.) Behavioural Modelling :

JK_FipFlop_Behavioural_ECE1003.v

C:/Users/admin/JK_FlipFlop_ECE1003/JK_FlipFlop_ECE1003.srcs/sources_1/new/JK_I

```
23 module JK_FipFlop_Behavioural_ECE1003(clk,j,knq,qbar);
24 input clk,j,k;
25 output reg q,qbar;
26
27 always@(posedge clk)
28 begin
29 if(k = 0)
30 begin
31 q <= 0;
32 qbar <= 1;
33 end
34 always@(posedge clk)
35 begin
36   if(k = 0)
37    begin
38     q <= 0;
39     qbar <= 1;
40    end
41   else if(j = 1)
42    begin
43     q <= 0;
44     qbar <= 0;
45    end
46   else if(j = 0 & k = 0)
47    begin
48     q <= q;
49     qbar <= qbar;
50    end
51   else if(j = 1 & k = 1)
52    begin
53     q <= ~q;
54     qbar <= ~qbar;
55    end
56 end
57 endmodule
```

4.) Structural :

JK_FlipFlop_Structural_ECE1003.v *

C:/Users/admin/JK_FlipFlop_ECE1003/JK_FlipFlop_ECE1003.srcs/sources_1/new/JK_

```
23 module JK_FlipFlop_Structural_ECE1003(q,qbar,clk,j,k);
24
25 input j,k,clk;
26 output q,qbar;
27
28 wire nand1_out; // output from nand1
29 wire nand2_out; // output from nand2
30
31 nand(nand1_out, j,clk,qbar);
32 nand(nand2_out, k,clk,q);
33 nand(q,qbar,nand1_out);
34 nand(qbar,q,nand2_out);
35
36 endmodule
```

5.) Testbench :

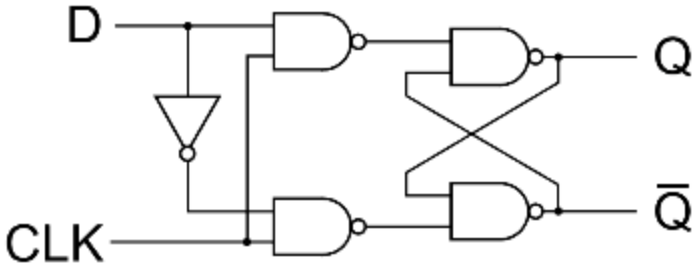
JK_FlipFlop_testbench_ECE1003.v

C:/Users/admin/JK_FlipFlop_ECE1003/JK_FlipFlop_ECE1003.srcs/sim_1/new/JK_FlipFlop_testbench_ECE1003.v

```
23 module JK_FlipFlop_testbench_ECE1003;
24 reg J,K, CLK;
25 wire Q, QBAR;
26
27 jkff_behavior dut(.q(Q), .qbar(QBAR), .j(J), .k(K), .clk(CLK));
28
29 $monitor("simtime = %g, CLK = %b, J = %b, K = %b, Q = %b, QBAR = %b", $time, CLK, J, K, Q, QBAR);
30
31 initial begin
32   clk=0;
33      forever #10 clk = ~clk;
34 end
35 initial begin
36  J= 1; K= 0;
37  #100; J= 0; K= 1;
38  #100; J= 0; K= 0;
39  #100; J= 1; K=1;
40 end
41 endmodule
```

III] D-Flip Flop

1.) Circuit diagram :



2.) Truth table :

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

3.) Coding :

D_FlipFlop_ECE1003.v *

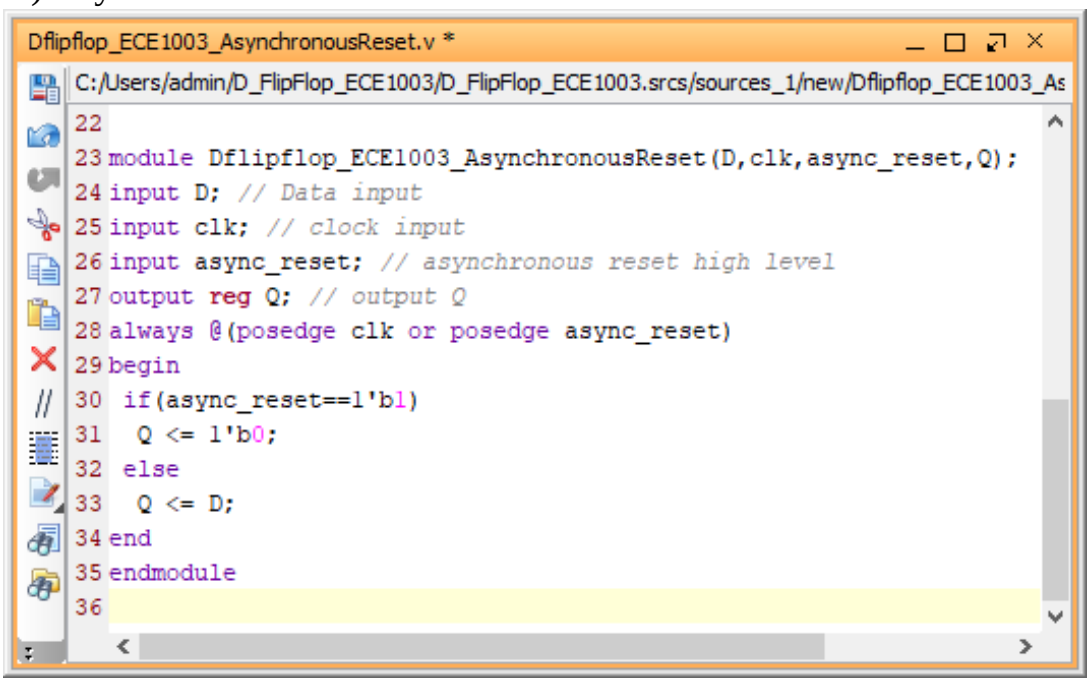
C:/Users/admin/D_FlipFlop_ECE1003/D_FlipFlop_ECE1003.srcs/sources_1,

```
22 // Verilog code for rising edge D flip flop
23 module D_FlipFlop_ECE1003(D,clk,Q);
24 input D; // Data input
25 input clk; // clock input
26 output Q; // output Q
27 always @(posedge clk)
28 begin
29  Q <= D;
30 end
31 endmodule
```

4.) Synchronous reset

```
module D_FlipFlop_ECE1003(D,clk,sync_reset,Q);

input D; // Data input
input clk; // clock input
input sync_reset; // synchronous reset

output reg Q; // output Q

always @(posedge clk)

begin
if(sync_reset==1'b1)
Q <= 1'b0;
else
Q <= D;
end
endmodule
```

5.) Asynchronous reset



6.) Asynchronous reset (Low Value)

DFlipFlop_ECE1003_Asynchronous_Reset_Low.v *

C:/Users/admin/D_FlipFlop_ECE1003/D_FlipFlop_ECE1003.srcs/sources_1/new/DFlipFlop_ECE1003_Asynch

```
23 module DFlipFlop_ECE1003_Asynchronous_Reset_Low(D,clk,async_reset,Q);
24 input D; // Data input
25 input clk; // clock input
26 input async_reset; // asynchronous reset low level
27 output reg Q; // output Q
28 always @(posedge clk or negedge async_reset)
29 begin
30  if(async_reset==1'b0)
31   Q <= 1'b0;
32  else
33   Q <= D;
34 end
35 endmodule
```

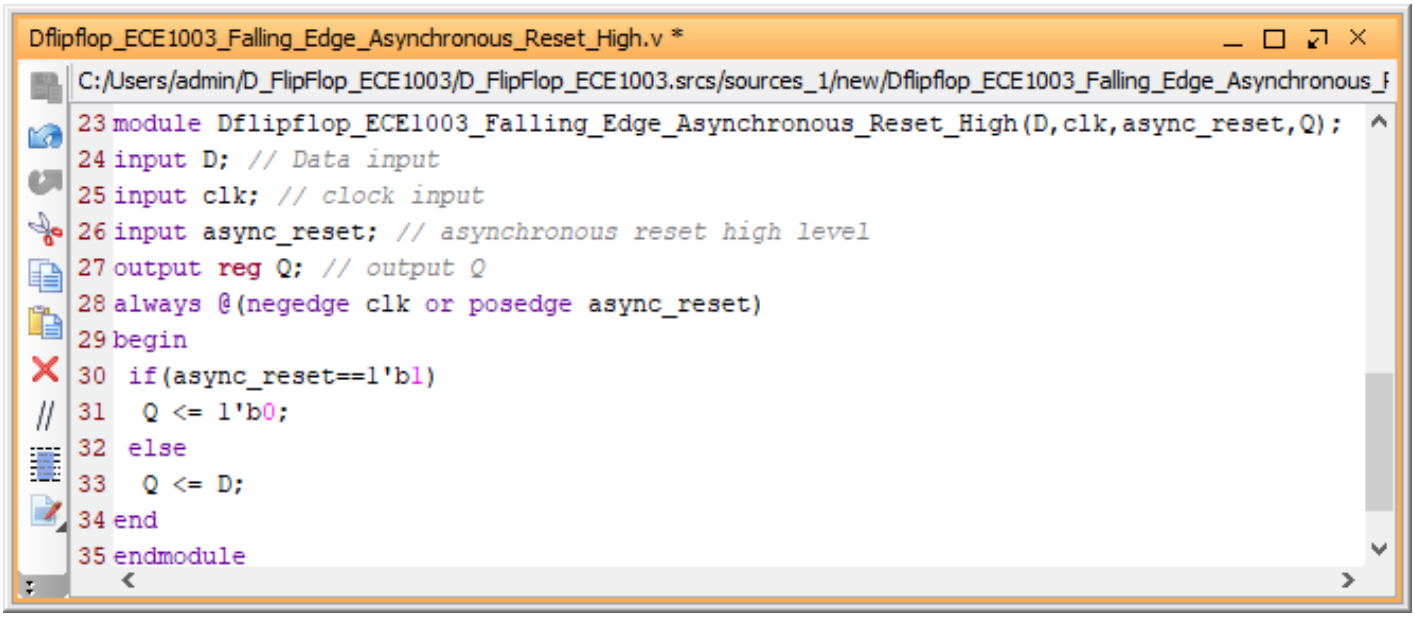
7.) Falling edge

DFlipFlop_ECE1003_Falling_Edge.v

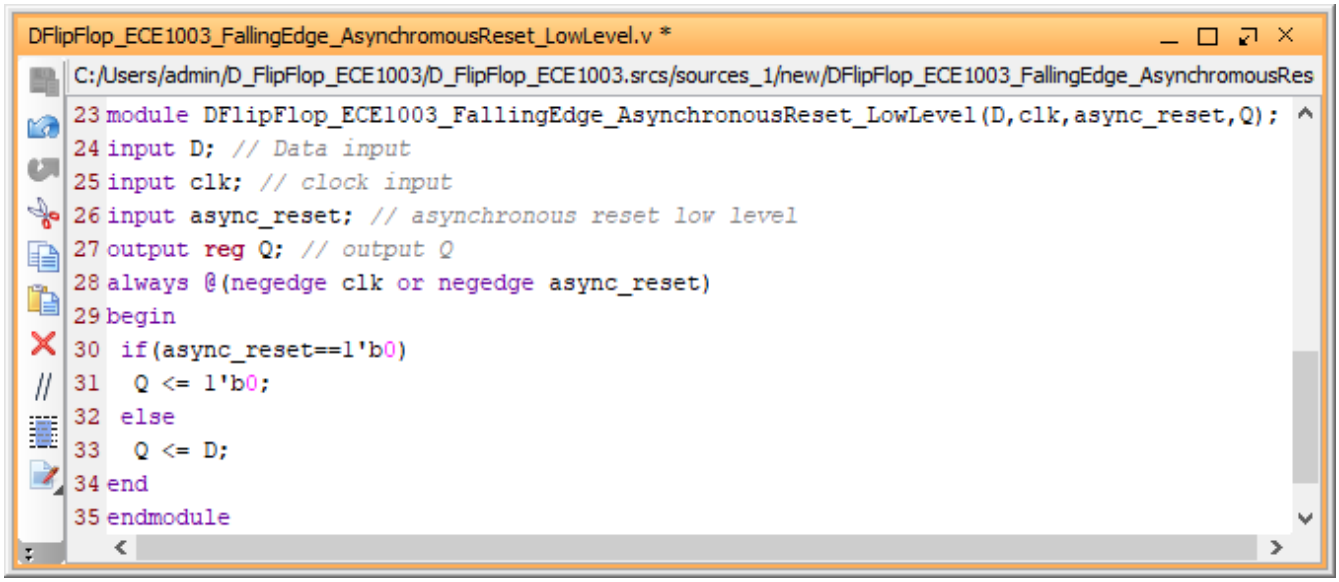
C:/Users/admin/D_FlipFlop_ECE1003/D_FlipFlop_ECE1003.srcs/sources_1/ne

```
23 module DFlipFlop_ECE1003_Falling_Edge(D,clk,Q);
24 input D; // Data input
25 input clk; // clock input
26 output reg Q; // output Q
27 always @(negedge clk)
28 begin
29  Q <= D;
30 end
31 endmodule
```

8.) Falling edge with asynchronous reset (high level)



9.) Falling edge with asynchronous reset (low level) :



10.) Testbench :

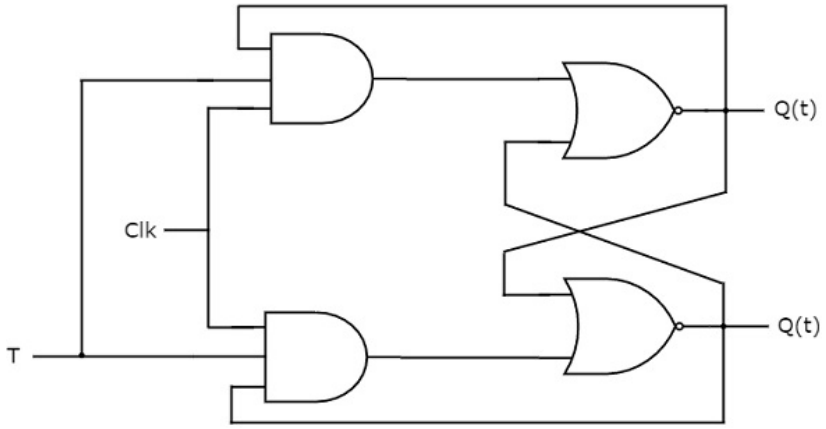
DFlipFlop_testbench_ECE1003.v

C:/Users/admin/D_FlipFlop_ECE1003/D_FlipFlop_ECE1003.srcs/sim_1/new/DF

```
23 module DFlipFlop_testbench_ECE1003;
24 reg D;
25 reg clk;
26 reg reset;
27 wire Q;);
28
29 initial begin
30   clk=0;
31      forever #10 clk = ~clk;
32 end
33 initial begin
34  reset=1;
35  D <= 0;
36  #100;
37  reset=0;
38  D <= 1;
39  #100;
40  D <= 0;
41  #100;
42  D <= 1;
43 end
44 endmodule
```

IV] T Flip Flop

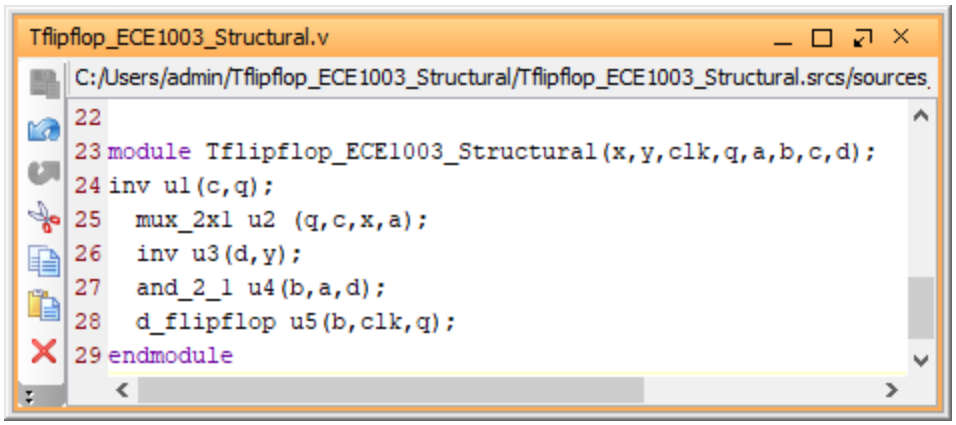
1.) Circuit diagram :



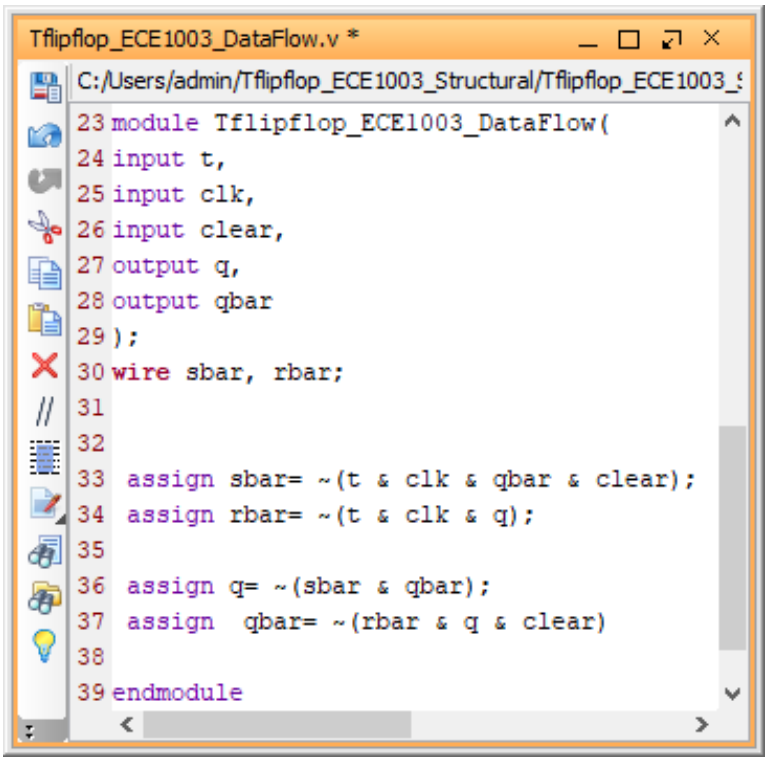
2.) Truth table :

T	Q(t)	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

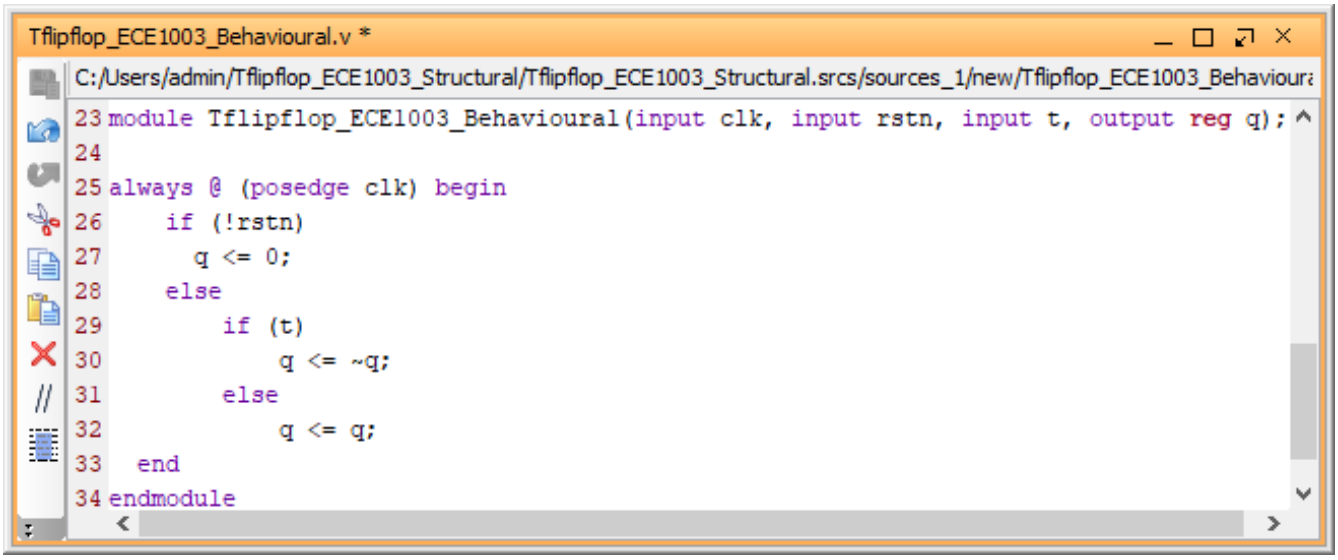
3.) Structural :



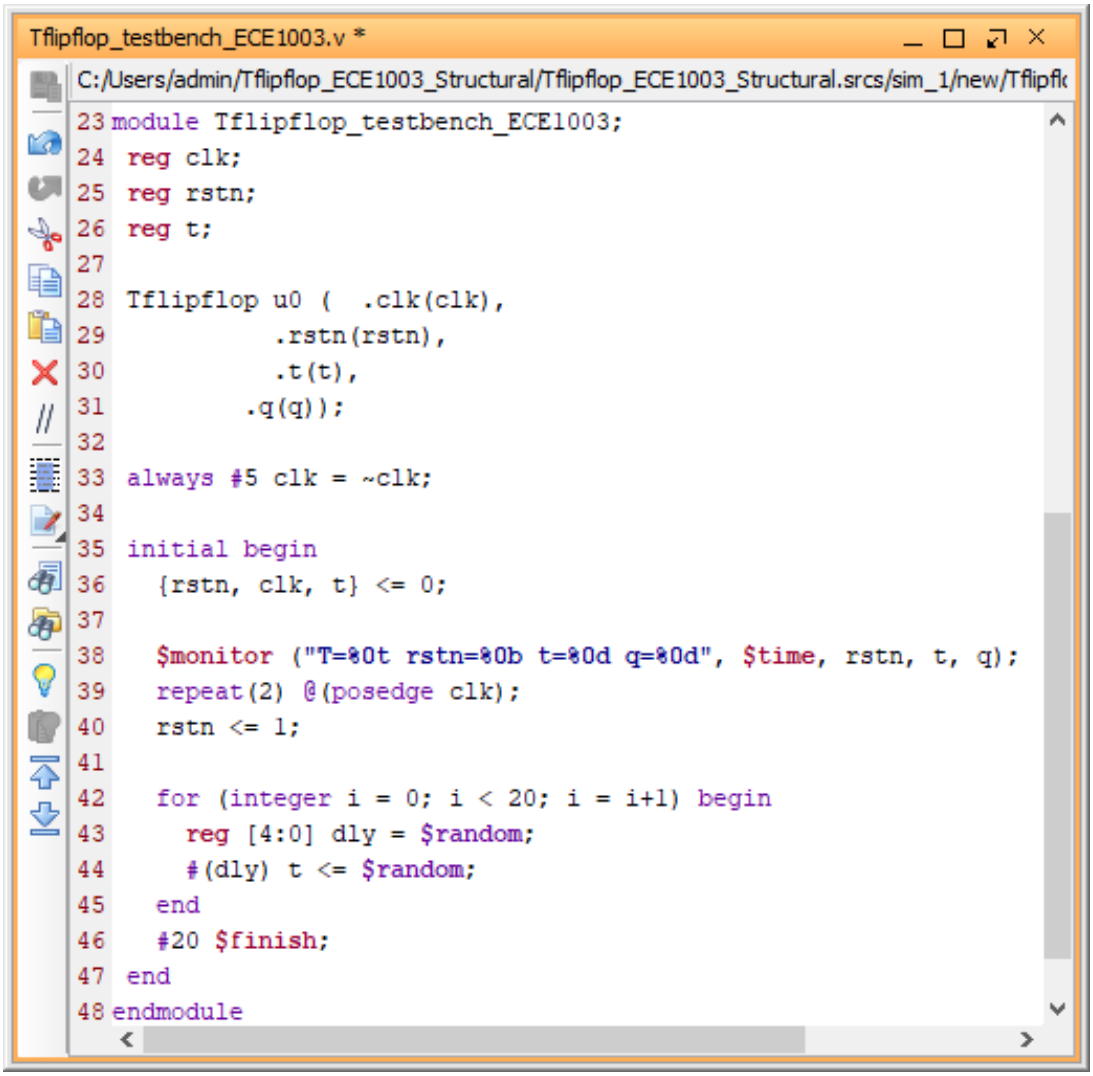
4.) Data Flow:



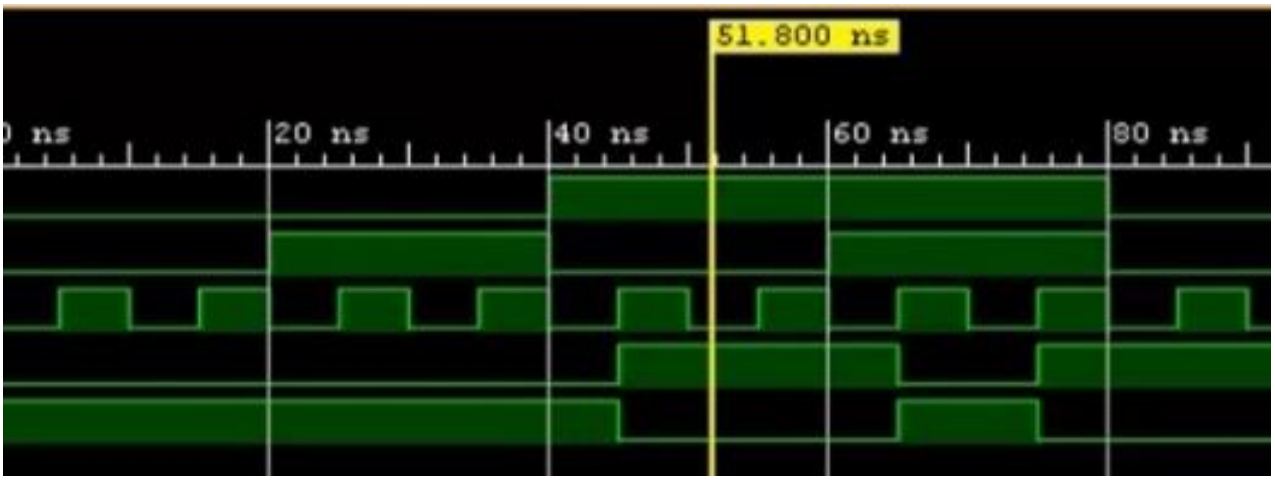
5.) Behavioural :

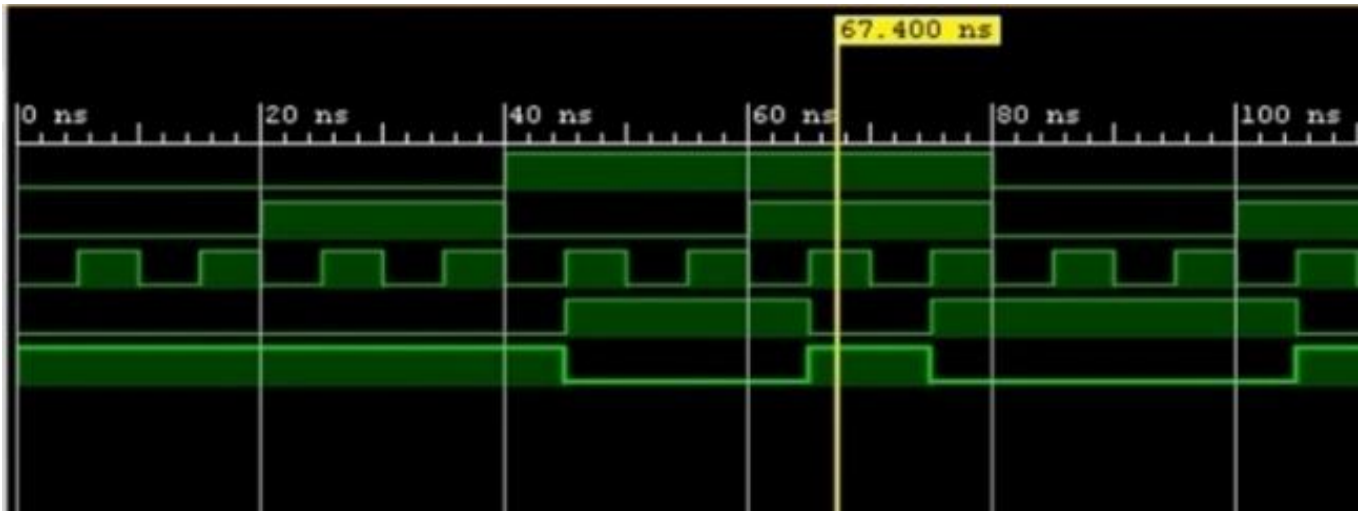


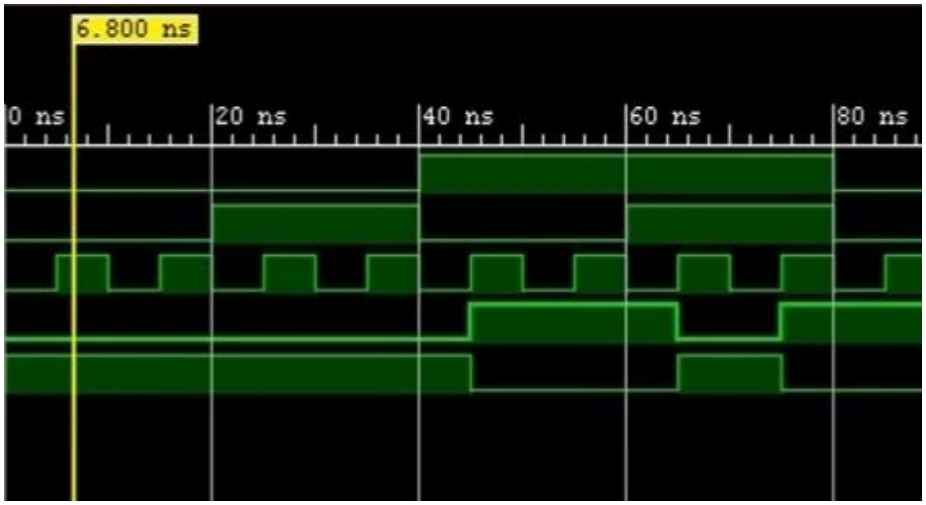
6.) Testbench :

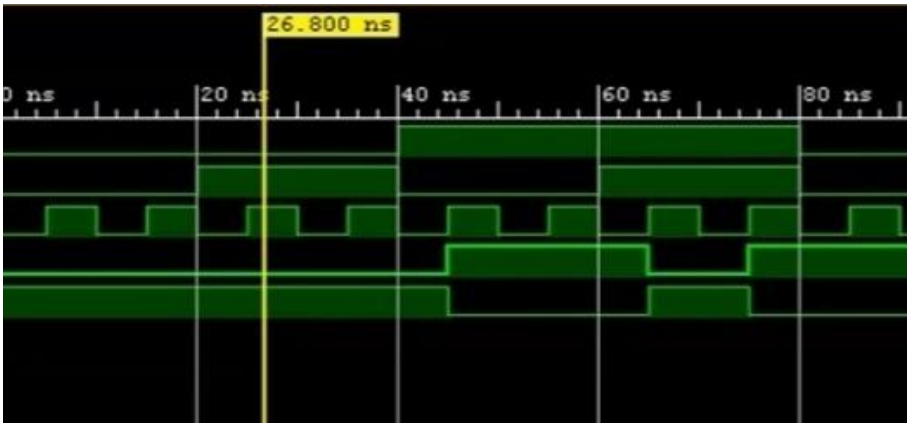


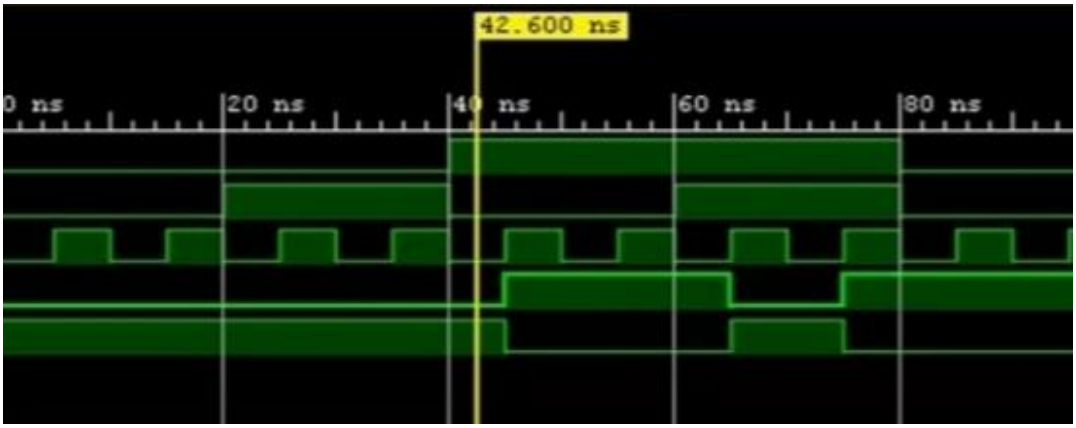
OUTPUT (J-K Flip Flop) :

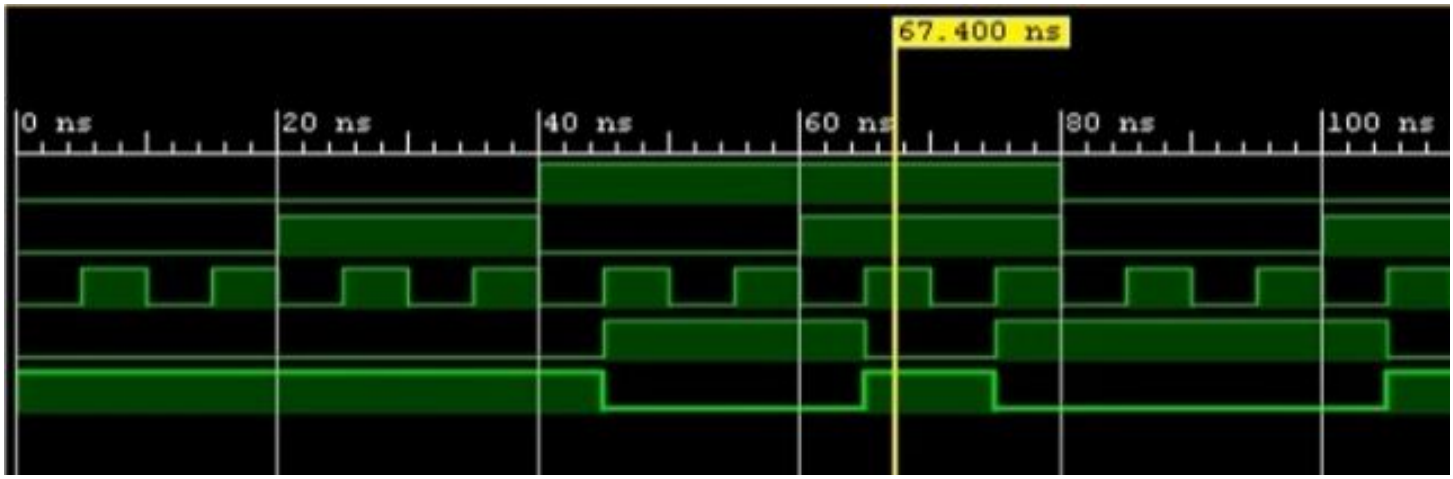




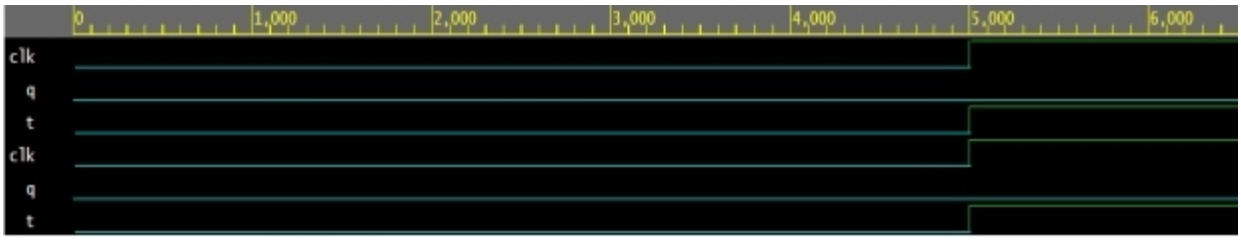




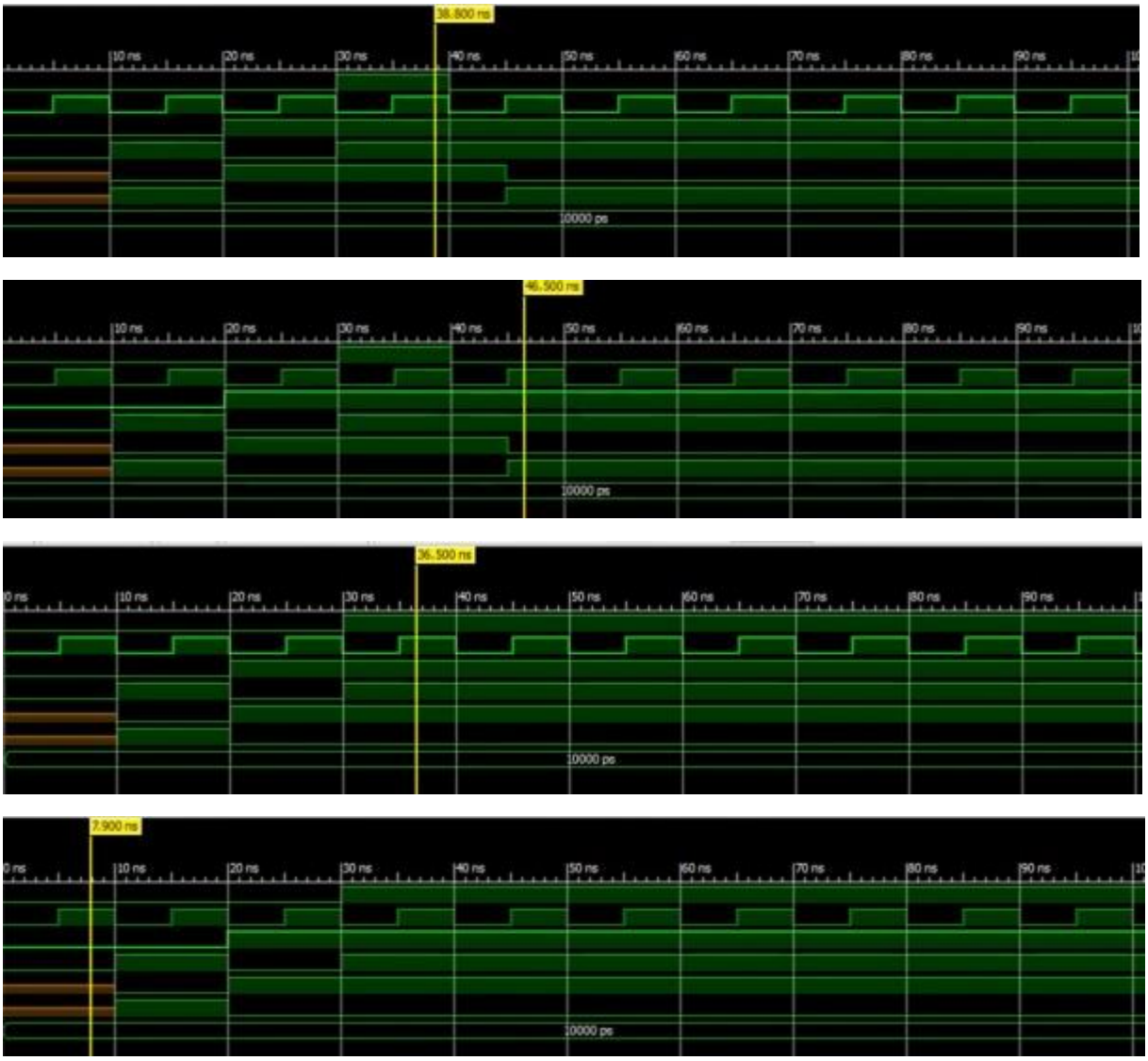




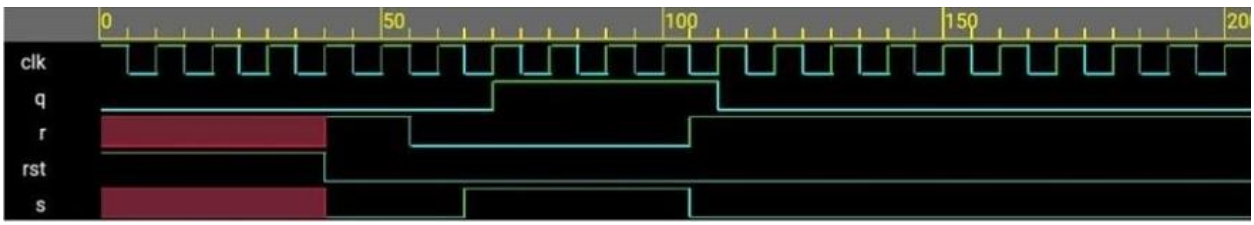
OUTPUT (T FLIP FLOP using EDA playground) :



OUTPUT (D FLIP FLOP) :



OUTPUT (SR FLIP FLOP):



ECE1003_Digital Logic Design_Experiment No. 8

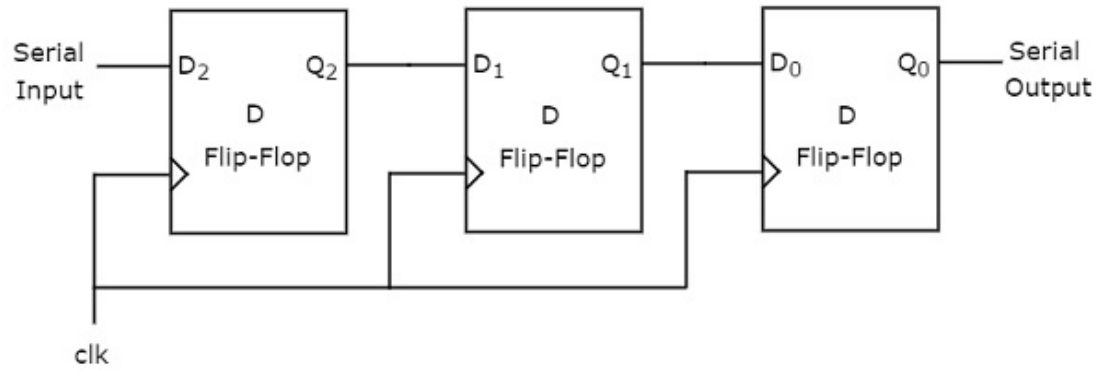
Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

Aim : Design and verify shift registers using verilog

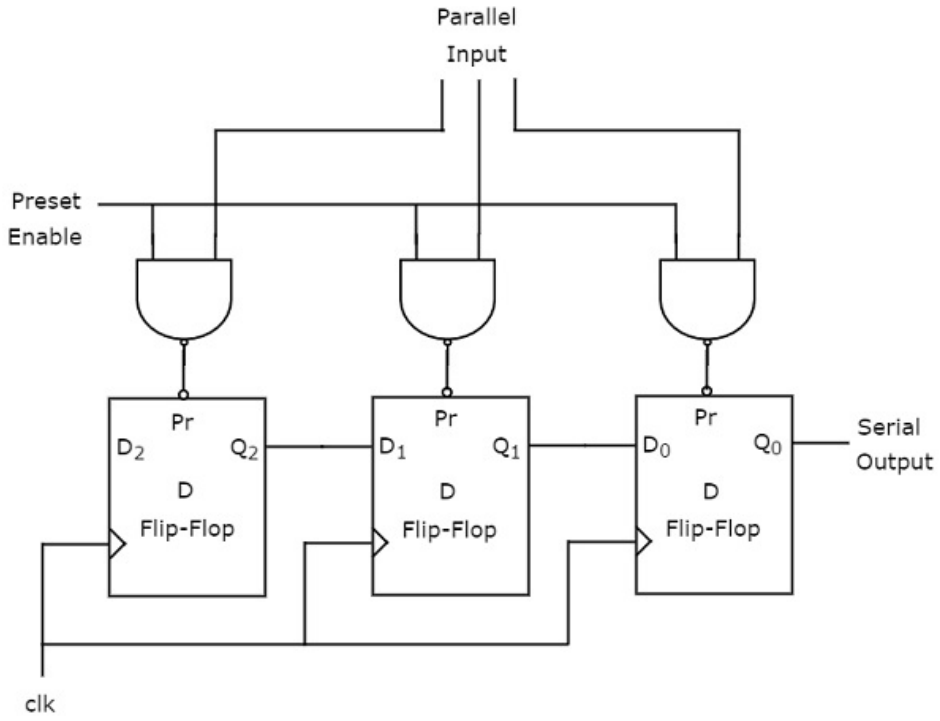
Theory :

A Register is a device which is used to store such information. It is a group of flip flops connected in series used to store multiple bits of data.
The information stored within these registers can be transferred with the help of shift registers. Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data.
The registers which will shift the bits to left are called “Shift left registers”.
The registers which will shift the bits to right are called “Shift right registers”.

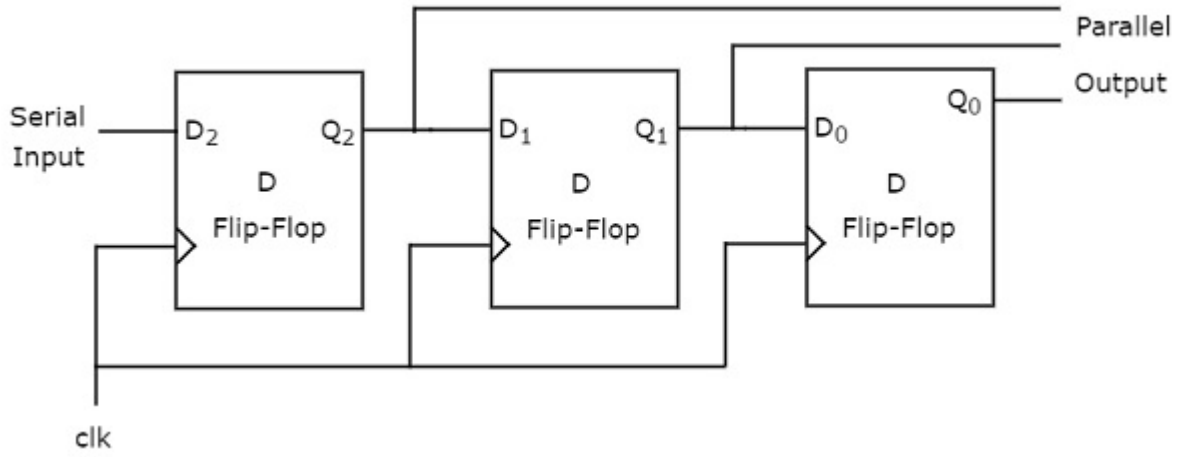
Serial-In Serial-Out Shift Register (SISO):- The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output is known as Serial-In Serial-Out shift register. Since there is only one output, the data leaves the shift register one bit at a time in a serial pattern, thus the name Serial-In Serial-Out Shift Register.The logic circuit given below shows a serial-in serial-out shift register. The circuit consists of four D flip-flops which are connected in a serial manner. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.



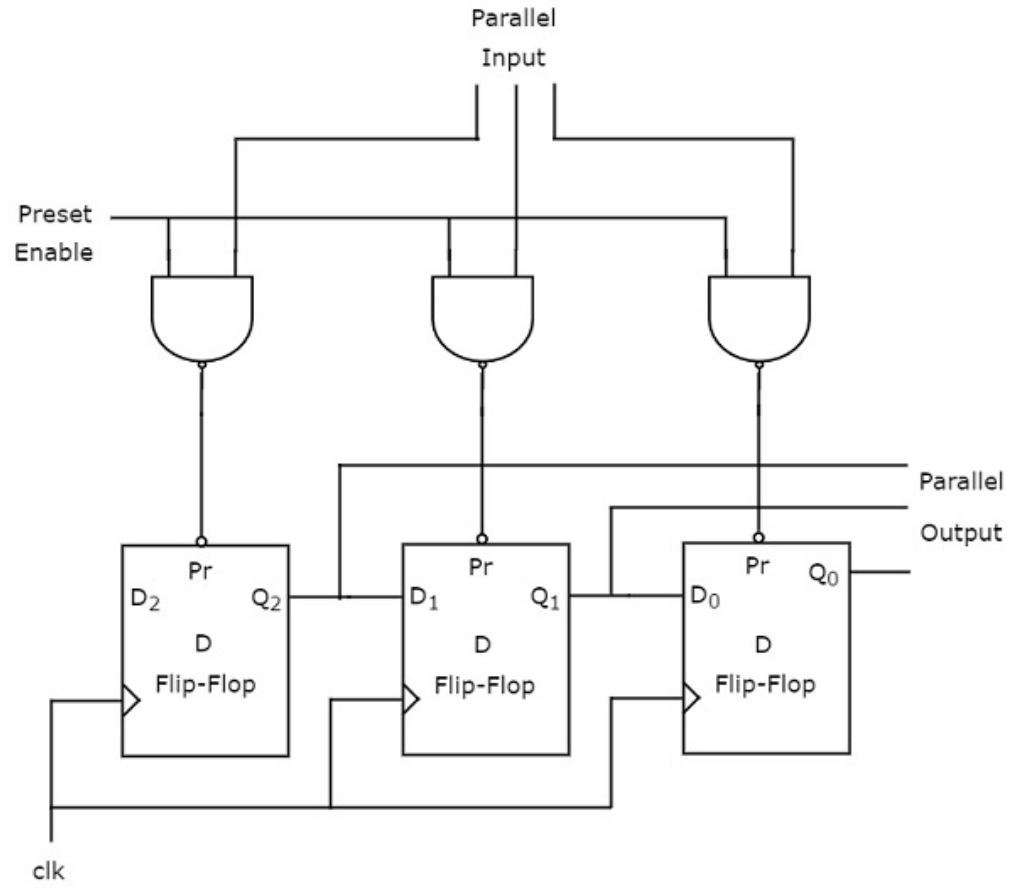
Parallel-In Serial-Out Shift Register (PISO):- The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and produces a serial output is known as Parallel-In Serial-Out shift register.The logic circuit given below shows a parallel-in-serial-out shift register. The circuit consists of four D flip-flops which are connected. The clock input is directly connected to all the flip flops but the input data is connected individually to each flip flop through a multiplexer at the input of every flip flop. The output of the previous flip flop and parallel data input are connected to the input of the MUX and the output of MUX is connected to the next flip flop. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.



Serial-In Parallel-Out shift Register (SIPO):- The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output is known as Serial-In Parallel-Out shift register.The logic circuit given below shows a serial-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal is connected in addition to the clock signal to all the 4 flip flops in order to RESET them. The output of the first flip flop is connected to the input of the next flip flop and so on. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.



Parallel-In Parallel-Out Shift Register (PIPO) :- The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output is known as Parallel-In parallel-Out shift register. The logic circuit given below shows a parallel-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal and clock signals are connected to all the 4 flip flops. In this type of register, there are no interconnections between the individual flip-flops since no serial shifting of the data is required. Data is given as input separately for each flip flop and in the same way, output also collected individually from each flip flop.



ECE1003_Digital Logic Design_Experiment No. 8 **23/05/2021**

Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

Aim : Design 4 bit shift registers using verilog code

Software used : Xilinx Vivado 2014.2

Theory : Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data.

1-BIT SHIFT REGISTER :

Testbench code :-

testbench_1BitRegister.v

C:/Users/admin/testbench_1BitRegister/testbench_1BitRegister.

```
23 module testbench_1BitRegister;
24 reg clk, in, clr, set;
25 wire out;
26 1BitRegister a1(clk, in, clr, set, out);
27 initial
28 begin
29 clk=1'b0; in=1'b0; clr=1'b1; set=1'b0;
30 #40 in=1'b1; clr=1'b0; set=1'b0;
31 #20 in=1'b0; clr=1'b0; set=1'b0;
32 #20 in=1'b1; clr=1'b1; set=1'b0;
33 #20 in=1'b0; clr=1'b1; set=1'b1;
34 end
35 always
36 begin
37 #5 clk=~clk;
38 end
39 endmodule
```

Output :-

4-BIT SHIFT REGISTER :

ECE1003_4BitRegister.v

C:/Users/admin/ECE1003_4BitRegister/ECE1003_4BitRegister.srcs/source

```
23 module ECE1003_4BitRegister(R, L, W, Clk, Q);
24 input [3:0] R;
25 input L, W, Clk;
26 output [3:0] Q;
27 reg [3:0] Q;
28 always @(posedge Clk)
29 if (L)
30 Q <= R;
31 else
32 begin
33 Q[0] <= Q[1];
34 Q[1] <= Q[2];
35 Q[2] <= Q[3];
36 Q[3] <= W;
37 end
38 endmodule
```

Testbench :-

testbench_4BitRegister_ECE1003.v

C:/Users/admin/ECE1003_4BitRegister/ECE1003_4BitRegister.srcs/s

```
23 module testbench_4BitRegister_ECE1003;
24  // Inputs
25 reg [3:0] a;
26 reg [1:0] shift;
27 reg clk;
28
29 // Outputs
30 wire [3:0] y;
31
32 shift_register uut (
33  .a(a),
34  .y(y),
35  .shift(shift),
36  .clk(clk)
37 );
38
39 initial begin
40  // Initialize Inputs
41 a = 3'b0110; shift = 2'b00; clk = 1; #100;
42 a = 3'b0110; shift = 2'b01; #100;
43 a = 3'b0110; shift = 2'b10; #100;
44 a = 3'b0110; shift = 2'b11; #100;
45 end
46 always
47 #50 clk=~clk;
48 endmodule
```

4BitRegister_ECE1003.v

C:/Users/admin/4BitRegister_ECE1003/4BitRegister_ECE1003.srcs/sources_1/new/4BitRegister_ECE1003.v

```
23 module 4BitRegister_ECE1003(clk, d, ce, pre, q);
24 input clk, ce, pre;
25 input [3:0] d;
26 output [3:0] q;
27 reg [3:0] q;
28 always @(posedge clk or posedge pre)
29 begin
30 if (pre)
31 q <= 4'b1111;
32 else if (ce)
33 q <= d;
34 end
35 endmodule
```

8-BIT SHIFT REGISTER :

i) Behavioral Model :

8BitReg_ECE1003.v

C:/Users/admin/8BitReg_ECE1003/8BitReg_ECE1003.srcs/sources_1/new/8BitReg_ECE1003.v

```
21
22
23 module 8BitReg_ECE1003(reset, CLK, D, Q);
24 input reset;
25 input CLK;
26 input [7:0] D;
27 output [7:0] Q;
28 reg [7:0] Q;
29 always @(posedge CLK)
30 if (reset)
31 Q = 0;
32 else
33 Q = D;
34 endmodule
```

ii) Data Flow Modelling :

8BitReg_DataFlow_ECE1003.v

C:/Users/admin/8BitReg_ECE1003/8BitReg_ECE1003.srcs/sources_1/new/8BitReg_Dat

```
 1 `timescale 10ns / 1ps
 2 module 8BitReg_DataFlow_ECE1003(out,in,load,clk);
 3 output [7:0]out;
 4 input [7:0]in;
 5 input clk;
 6 input load;
 7 reg [7:0]out;
 8 wire [7:0]in;
 9 always@(posedge clk)
10 begin
11 if(load)
12 begin
13 out<=in;
14 end
15 else
16 begin
17 out[0]<=out[7];
18 out[1]<=out[0];
19 out[2]<=out[1];
20 out[3]<=out[2];
21 out[4]<=out[3];
22 out[5]<=out[4];
23 out[6]<=out[5];
24 out[7]<=out[6];
25 end
26 endmodule
```

iii) Testbench :

testbench_8BitReg_ECE1003.v

C:/Users/admin/8BitReg_ECE1003/8BitReg_ECE1003.srcs/sim_1/new/testbench_8BitReg_ECE1003.v

```
 3 module testbench_8BitReg_ECE1003;
 4 reg clktb;
 5 reg resettb;
 6 wire [7:0] counttb;
 7 // create an instance of the module
 8 testbench_8BitReg_ECE1003 mycounter(.clk (clktb),
 9  .reset (resettb),
10 .count (counttb));
11 // setup the initial state of the system
12 initial
13 begin
14 $display ("Time\tClk\tReset\tCount");
15 $monitor("%d\t%b\t%b\t%b", $time, clktb, resettb,
16 counttb);
17 clktb = 0;
18 resettb = 0;
19 counttb = 0;
20 #20 // delay for 20 units
21 // test reset condition
22 #20 resettb = 1;
23 #20 resettb = 0;
24 // stop testing at 200 time units
25 #200 $finish;
26 end
27 // clock generation
28 always begin
29  #10 clktb = !clktb; // every 10 time units invert
30 clocks value
31 end
32 endmodule
33 // counter module
34 module listing6 (clk, reset, count);
35 //input clk;
36 input reset;
37 input clk;
38 // outputs
39 output [7:0] count;
40 reg [7:0] count;
41 always @(posedge clk, posedge reset)
42 begin
43 if(reset) begin
44 count <= 8'b0;
45 end
46 else begin
47 count <= count + 1;
48  end
49 end
50 endmodule
```

4 BIT SHIFT REGISTER (Using EDA playground) :

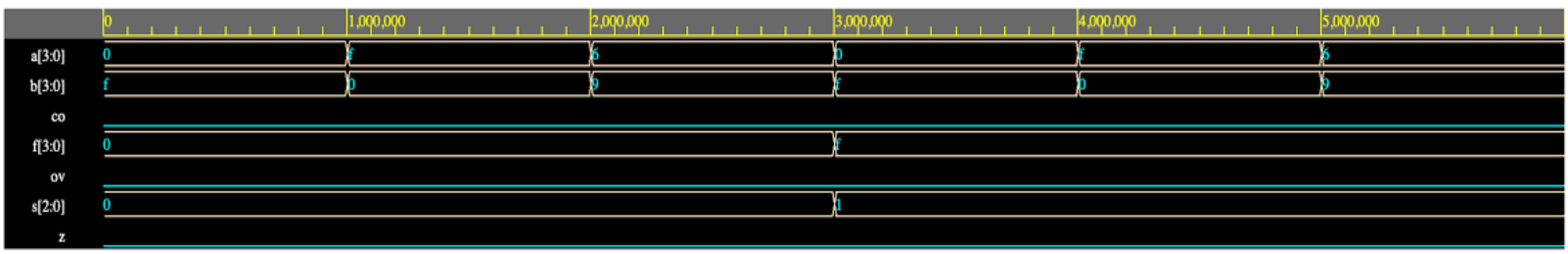
testbench.sv

```
// KHAN MOHD. OWAIS RAZA_20BCD7138_EXP8_SHIFT REGISTER //
module ECE1003_4BitShiftReg_tb;
  wire [3:0] Q;
  reg clock, reset;

  ECE1003_4BitShiftReg(Q, clock, reset);

  initial
  begin
    clock =0;
    reset = 1; #10; reset = 0; #10;
    #200;
    $finish;

  end
  always #5 clock = ~clock;
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end

endmodule
```

design.sv

```
// KHAN MOHD. OWAIS RAZA_20BCD7138_EXP8_SHIFT REGISTER //
module ECE1003_4BitShiftReg(Q, clock, reset);
  output reg [3:0] Q;
  input clock, reset;

  always @(posedge clock)
  begin
    if(reset)
      Q <= 4'b1111;
    else
      Q <= { Q[2:0], (Q[3] ^ Q[0]) };
  end

endmodule
```

Output :



8 BIT SHIFT REGISTER (Using EDA Playground) :

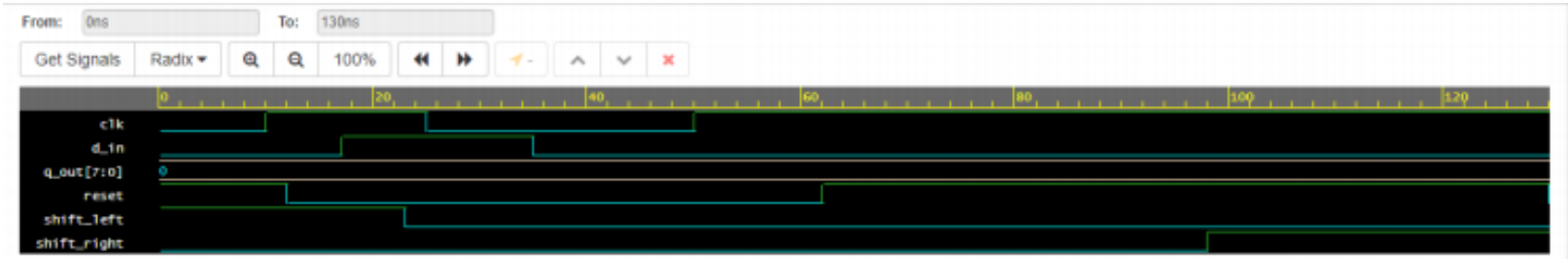
testbench.sv

```
// KHAN MOHD. OWAIS RAZA_20BCD7138_EXP8_SHIFT REGISTER //
module ECE1003_8BitShiftReg_tb;
  reg clk_tb, reset_tb, d_in_tb, shift_left_tb,
shift_right_tb;
 wire [7:0] q_tb;

 shift_Reg m1(
 .shift_left(shift_left_tb), .reset(reset_tb),
.shift_right(shift_right_tb),
 .d_in(d_in_tb), .clk(clk_tb),.q_out(q_tb)
 );
 initial begin
 reset_tb = 1; #12
 reset_tb = 0; #50
 reset_tb = 1; #68
 reset_tb = 0;
end
initial begin
 clk_tb<=0; #10
 clk_tb <= 1; #15
 clk_tb <= 0; #25
 clk_tb <= 1;
 end

initial begin
 d_in_tb = 0; #17
 d_in_tb = 1; #18 d_in_tb = 0;
end

initial begin
 shift_left_tb = 1;
 shift_right_tb = 0; #23
 shift_left_tb = 0; #75
 shift_right_tb = 1;
end
initial begin
 $dumpfile("dump. vcd");
 $dumpvars;
end
endmodule
```

design.sv

```
// KHAN MOHD. OWAIS RAZA_20BCD7138_EXP8_SHIFT REGISTER //
module ECE1003_8BitShiftReg(
  shift_left,shift_right,clk,reset,q_out,d_in);
   input shift_left,shift_right,d_in,clk,reset;

 output reg[7:0] q_out;

 always @(posedge clk)
begin
 if (~reset) begin
 if (shift_left)
 begin
 q_out <= #7 {q_out [6:0],d_in};
 end
 else if (shift_right)
 begin
 q_out <= #7 {d_in, q_out[7:1] };
 end
 end
 end
 always @ (posedge reset)
 begin
 q_out<= 8'b00000000;
 end
endmodule
```

Output :



ECE1003_Digital Logic Design_Experiment No. 9_7th June 2021

Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

Aim : Design and verify counter using verilog

Software used : Vivado 2014.2 / EDA playground

Theory :

Counters :- A special type of sequential circuit used to count the pulse is known as a counter, or a collection of flip flops where the clock signal is applied is known as counters. The counter is one of the widest applications of the flip flop. Based on the clock pulse, the output of the counter contains a predefined state. The number of the pulse can be counted using the output of the counter.

3-bit asynchronous up counter :- The 3-bit asynchronous or ripple up counter is similar to the 2-bit ripple up counter. Here for a 3-bit counter, an additional flip-flop is added. Thus for the 3-bit asynchronous counter, 3 T-flip-flops are used.
This counter consists of $2^3 = 8$ count states (000, 001, 010, 011, 100, 101, 110, 111). The counter counts the incoming pulses starting from 0 to 7.
The clock pulse count is noted at the output of each flip-flop($Q_CQ_BQ_A$), where Q_A is the LSB and Q_C is the MSB.

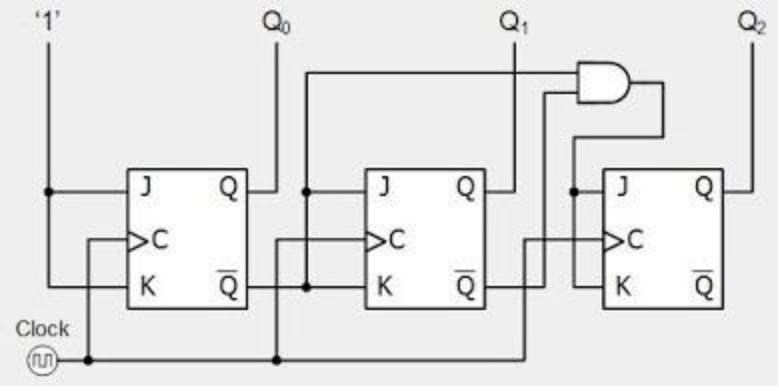
3-bit asynchronous down counter **:-** The down counter will count the clock pulses from maximum value to zero. In other words, for each clock pulse, the count value is decremented.
The below diagram shows the 3-bit asynchronous down counter. Since it is a 3-bit counter, 3 negative edge-triggered flip-flops are used. The clock pulse input is given only to the first flip-flop. The clock input of the remaining flip-flops is triggered by the Q output of the previous flip-flop.

3-bit synchronous up counter :- Synchronous up Counter counts the number of clock pulses at its input from minimum to maximum. A 3-bit counter consists of 3 flip-flops and has $2^3 = 8$ states from 000 to 111.
The circuit of the 3-bit synchronous up counter is shown below. The clock pulse is given for all the flip-flops. The T_A input for the first T-flip-flop TFF1 is always maintained at logic HIGH. The output of TFF1 is fed as an input for TFF2. The Q_A and Q_B output of TFF1 and TFF2 are ANDed together and its output is given to the TFF3. The output states of the counter can be observed at $Q_CQ_BQ_A$**.**

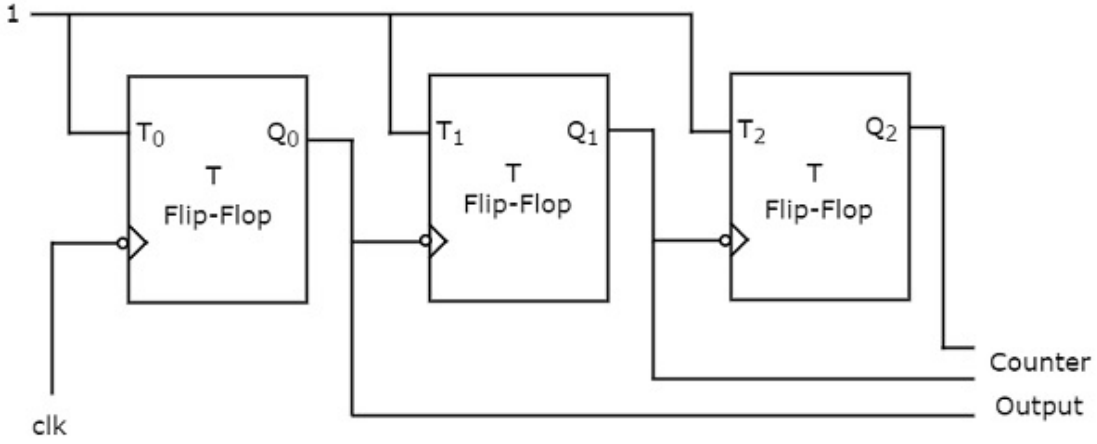
During the falling edge of the first clock pulse input, the output of TFF 1 toggles and produces $Q_A = 1$. For the remaining flip flops, there will be no change in the state.
The output becomes $Q_CQ_BQ_A = 001$. Thus the counter is incremented by 1(000 -> 001). Now, the input for TFF 1 is $T_A = 1$. As Q_A becomes 1, the input for TFF 2 becomes $T_B = 1$. and the input for TFF 3 has no change $T_C = 0$.
During the negative edge of the second clock pulse, the output of TFF 1 toggles and produce $Q_A = 0$. At the same time, TFF 2 also toggles and produces $Q_B = 1$, whereas there will not be a change in the state of TFF3.
Thus the output becomes $Q_CQ_BQ_A = 010$. So the counter increases its value to 2(001 -> 010).Now, the input for TFF 1 is $T_A = 1$. As Q_A and Q_B output are 0 and 1 respectively, the input $T_B = 0$ and the input will be $T_C = 0$.
During the negative edge of the third clock pulse, the TFF 1 will toggle to produce $Q_A = 1$. Since $T_B = 0$, there will not be any change in the state of TFF 2, so its output will be $Q_B = 1$. Since $T_C = 0$, there will not be a change in the output of TFF3, $Q_C = 0$.Thus the output now becomes $Q_CQ_BQ_A = 011$. So the counter increments its value to 3(010 -> 011). Now, the inputs of the flip-flops will become, $T_A = 1$, $T_B = 1$, $T_C = 1$.During the negative edge of the fourth clock pulse, the output of TFF1 will toggle, so QA = 0, Since T_B and T_C inputs are HIGH, both the flip-flops will toggle its output, so $Q_B = 0$, $Q_C = 1$. Now the output becomes $Q_CQ_BQ_A = 100$ and so the counter value increases to 4(011 -> 100).The operation continues and counts the clock pulses until it reaches the final state $Q_CQ_BQ_A = 111$. At the next clock pulse, the counter resets to 000 and starts to count from the first.

Circuit Diagram :

For 3 bit synchronous counter :-



For 3 bit asynchronous counter :-



Truth Table :

Q_C	Q_B	Q_A	Q_{C+}	Q_{B+}	Q_{A+}	T_C	T_B	T_A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

Count[A]	Count[B]	Count[C]
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Verilog For 3 Bit Counter (Using EDA playground) :

testbench.sv

```
// Khan Mohd. Owais Raza 20BCD7138 //
// ECE1003_Digital Logic Design_Experiment-9 //
module tb_3BitSync_ECE1003;
 reg clk,in;
 wire [2:0] count;

3BitSync_ECE1003 sc1(clk,in,count);

initial
begin
clk = 1'b0; in = 1'b0;
#20 in = 1'b1;
end

always
begin
#5 clk = ~clk;
end

endmodule
```

design.sv

```
// Khan Mohd. Owais Raza (20BCD7138) //
// ECE1003_Digital Logic Design_Lab Exp-9 //
module 3BitCountSync(clk,in,count);
  input clk,in;
  output [2:0] count;

  wire qa,qb,qc,z,qa_bar,qb_bar,qc_bar;

  t_ff t1(clk,in,qa,qa_bar);
  t_ff t2(clk,qa,qb,qb_bar);
  and_gate a1 (qa,qb,z);
  t_ff t3(clk,z,qc,qc_bar);
  send s1(qa,qb,qc,count);

endmodule
```

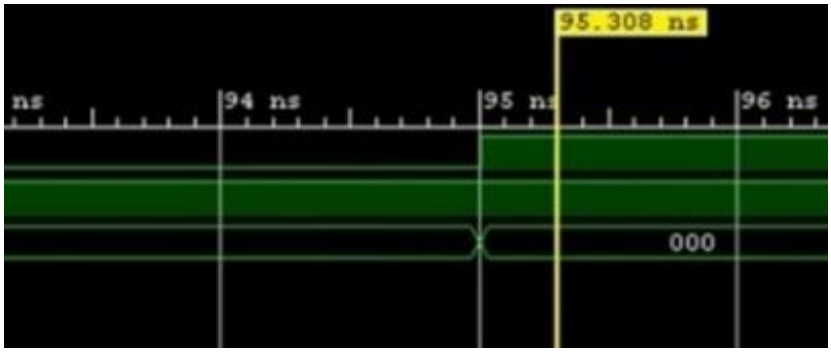
design.sv | ECE1003_3BitCountSync

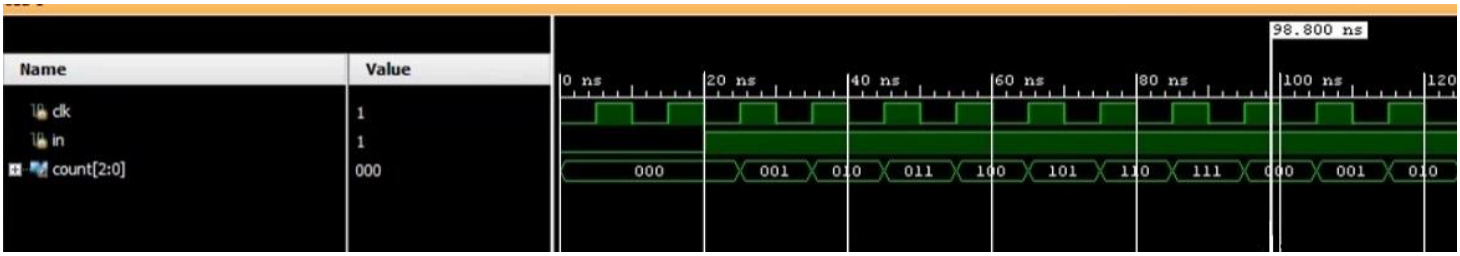
```
// Khan Mohd. Owais Raza (20BCD7138) //
// ECE1003_Digital Logic Design_Lab Exp-9 //

module ECE1003_3BitCountSync(clk,in,out,out_bar);

  input clk,in;
  output out,out_bar;

  reg out,out_bar;

  initial
    out = 0;
  always@(posedge clk)
    begin
      if(in==1)
        begin
          out=~out;
          out_bar=~out;
        end
      else
     begin
       out=out;
       out_bar=out;
     end
    end
endmodule
```

Verilog code (in Xilinx Vivado) :

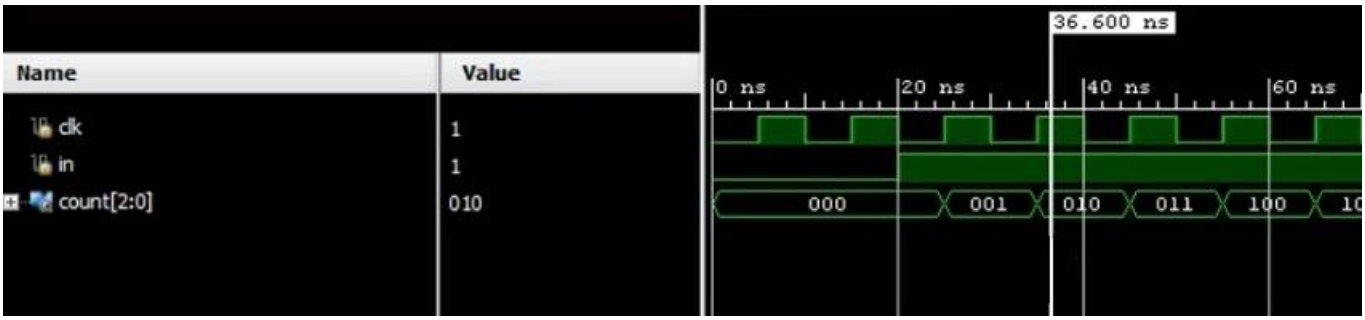
```
module 3BitSyncCounter(clk,reset,out);

input clk,reset;

output [2:0] out;

reg [2:0] out;

always @(posedge clk)

begin

 if (reset == 1'b1)

   out <= 3'b000;

    else

    out <= out+1'b1;

 end
endmodule
```

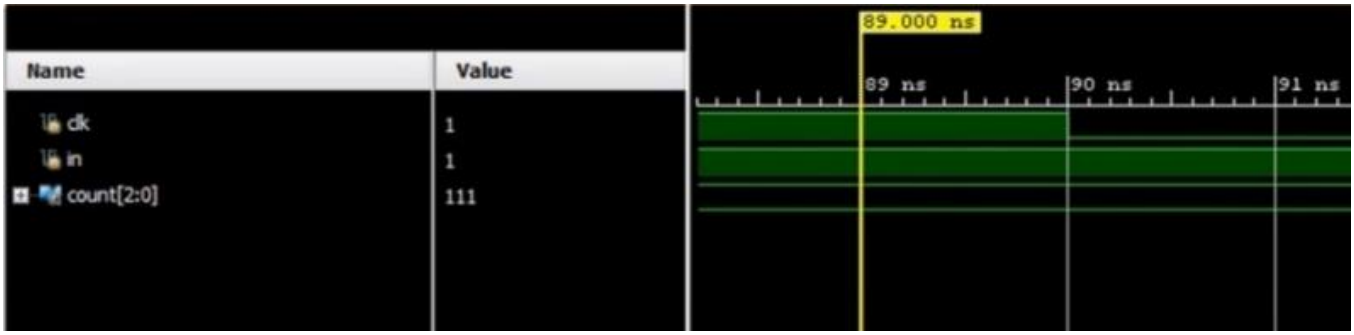
Output using Xilinx Vivado (taught in lab class) :

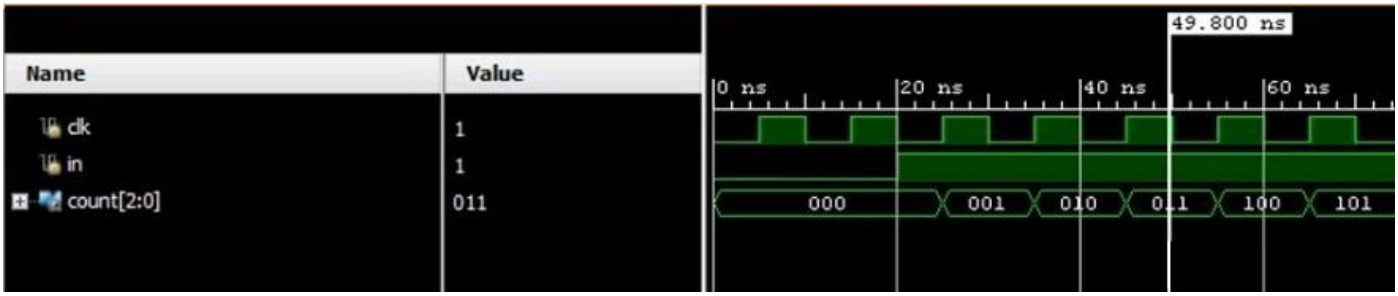


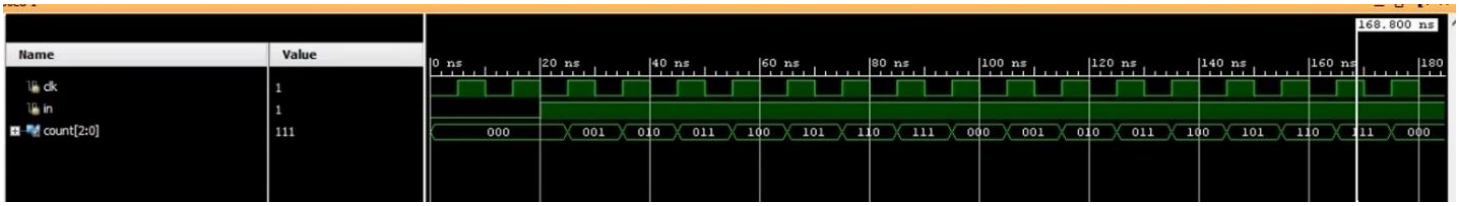

Name
Value
clk
1
in
1
count[2:0]
000
98.800 ns
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
000
001
010
011
100
101
110
111
000
001
010

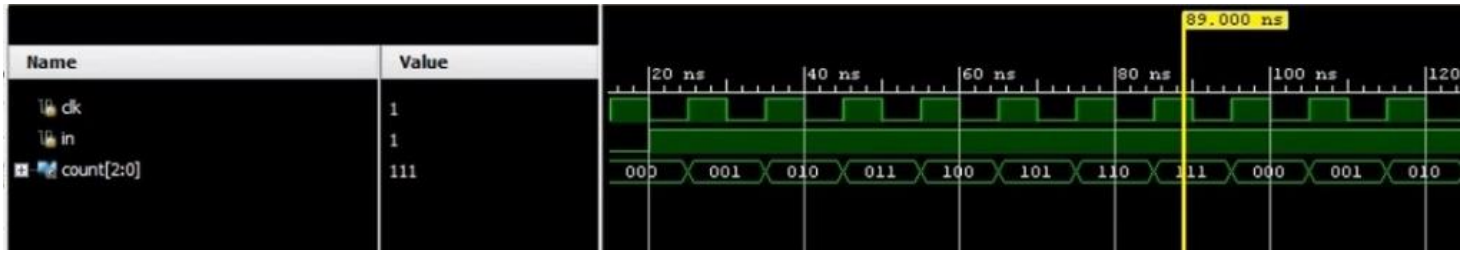

Name
Value
clk
0
in
1
count[2:0]
101
71.200 ns
0 ns
20 ns
40 ns
60 ns
80 ns
000
001
010
011
100
101
110
111


Name
Value
clk
1
in
1
count[2:0]
010
36.600 ns
0 ns
20 ns
40 ns
60 ns
000
001
010
011
100

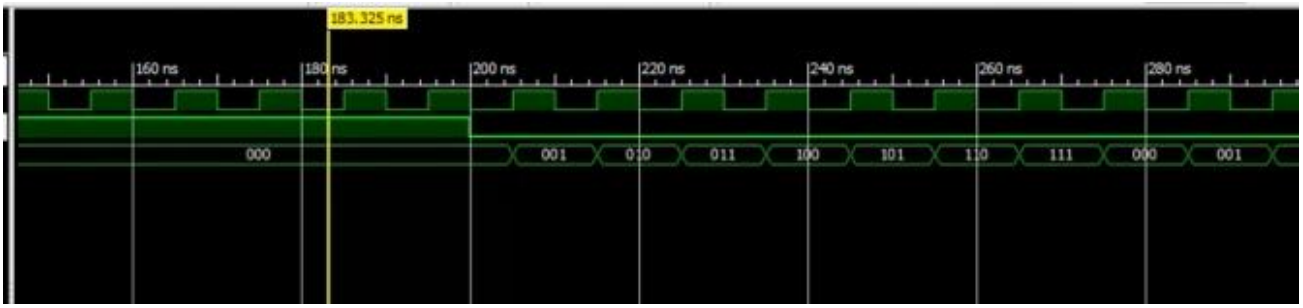

Name
Value
clk
1
in
1
count[2:0]
111
89.000 ns
89 ns
90 ns
91 ns

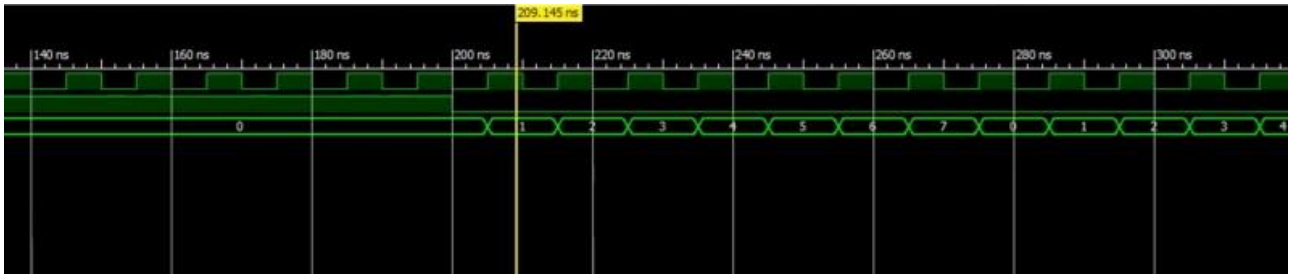

Name
Value
clk
1
in
1
count[2:0]
011
49.800 ns
0 ns
20 ns
40 ns
60 ns
000
001
010
011
100
101

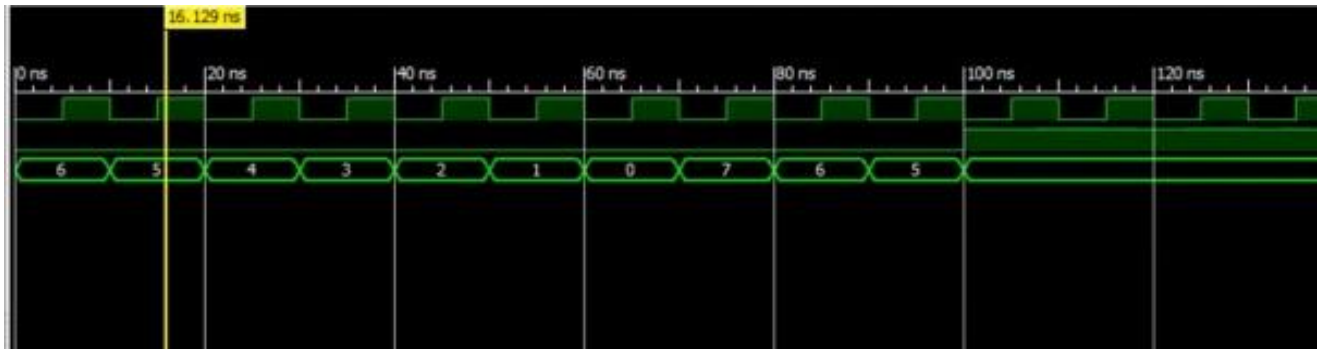

Name
Value
clk
1
in
1
count[2:0]
111
168.800 ns
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
000
001
010
011
100
101
110
111
000
001
010
011
100
101
110
111
000

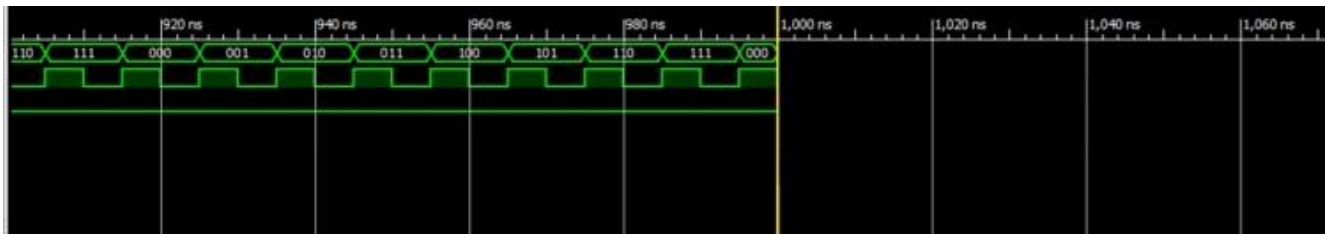


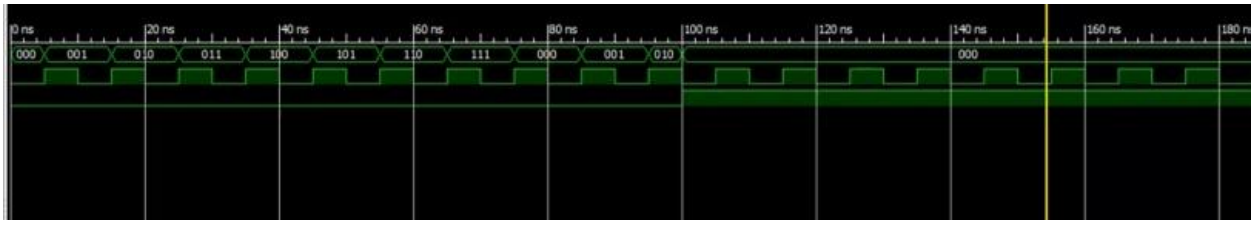
OUTPUT using Xilinx Vivado (self evaluated) :

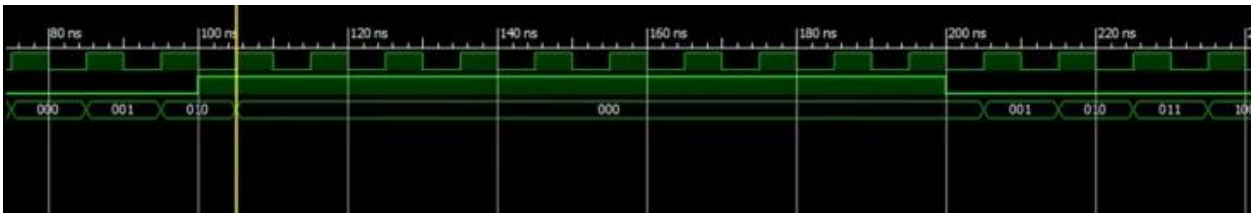


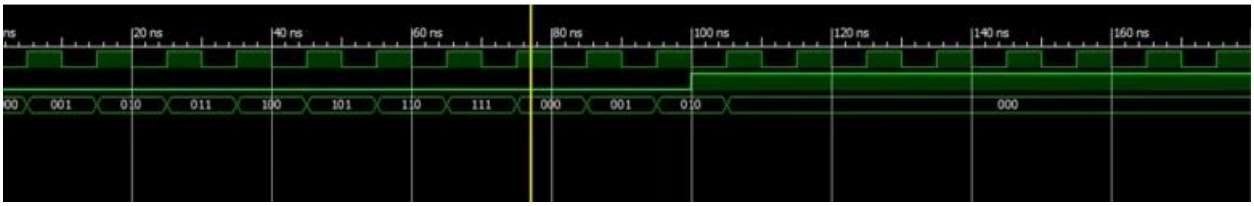


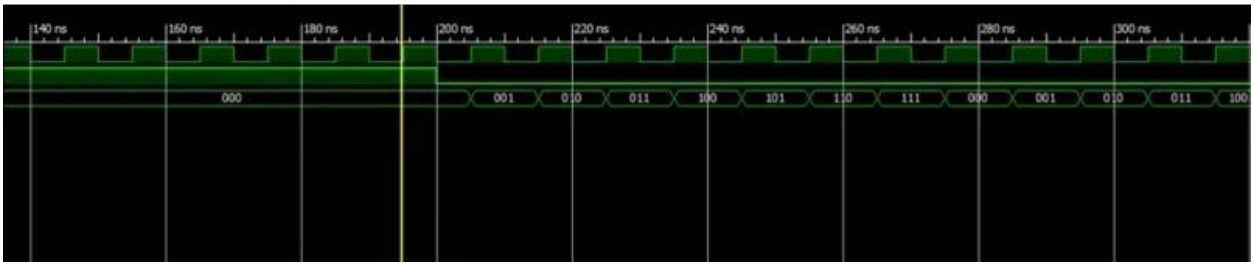







80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
220 ns
000
001
010
000
001
010
011


20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
001
010
011
100
101
110
111
000
001
010
000


140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
000
001
010
011
100
101
110
111
000
001
010
011

ECE1003_Digital Logic Design_Experiment No. : 10_26th June 2021

Name : KHAN MOHD. OWAIS RAZA
Reg. No. : 20BCD7138

Aim : Design and verify 3 bit asynchronous counter in verilog

Software used : Xilinx Vivado 2014.2

Verilog codes :

```
// KHAN MOHD. OWAIS RAZA_20BCD7138_ECE1003 //
// 3 bit asynchrnous counter using T flip flop //
module 3BitAsyncCounter_ECE1003(q,qbar,t,clk,rst);
input t,clk,rst;
 output reg q=0,qbar=1;

 always@(posedge clk,posedge rst)
 begin
 if(rst)begin
 q=0;
 qbar=1;
 end

 else begin
 case(t)
 1'b0:begin
      q<=q;
      qbar<=qbar;
      end

 1'b1:begin
      q<=~q;
      qbar<=~qbar;
      end

  endcase
  end
  end
  endmodule

  module 3BitAsyncCounter_ECE1003(a,abar,b,bbar,clk);
  input clk;
  output a,b,abar,bbar;
  wire rst;

  assign rst= a & b;

  t_ff one(a,abar,1'b1,clk,rst);
  t_ff two(b,bbar,1'b1,abar,rst);
endmodule

```

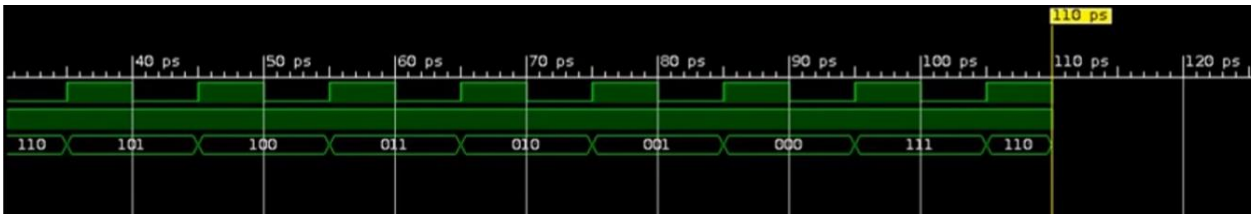
Testbench :

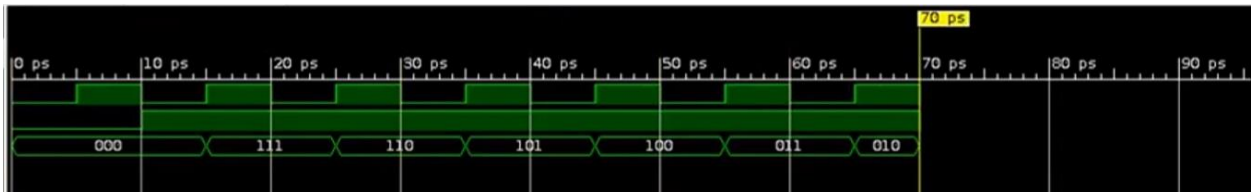
tb_3BitAsyncCounter_ECE1003.v

C:/Users/admin/3BitAsyncCounter_ECE1003/3BitAsyncCounter_ECE1003.srcs/sim_1/new/tb_3BitAsyncCounter_ECE1003.v

```
23 module tb_3BitAsyncCounter_ECE1003;
24 reg clk;
25 wire a,b,abar,bbar;
26
27 3BitAsyncCounter_ECE1003(a,abar,b,bbar,clk);
28
29 initial begin clk=0;
30 end
31 always begin #5 clk=~clk;
32 end
33 initial begin
34 $display ("\t\ttime\t clk\t\t {b,a}\t");
35 $monitor("%d\t %b\t\t %b\t",$time,clk,{b,a});
36 end
37
38 initial begin #100 $finish;
39 end
40 endmodule
```

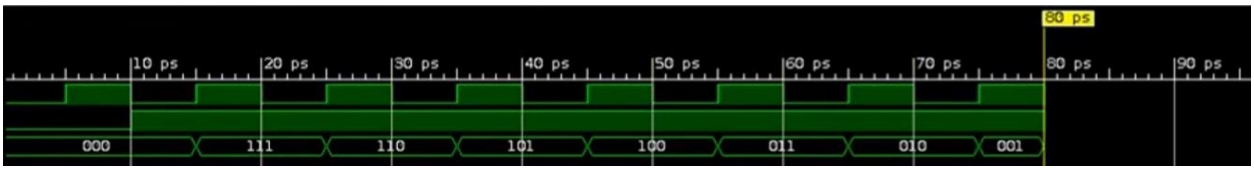
OUTPUT :

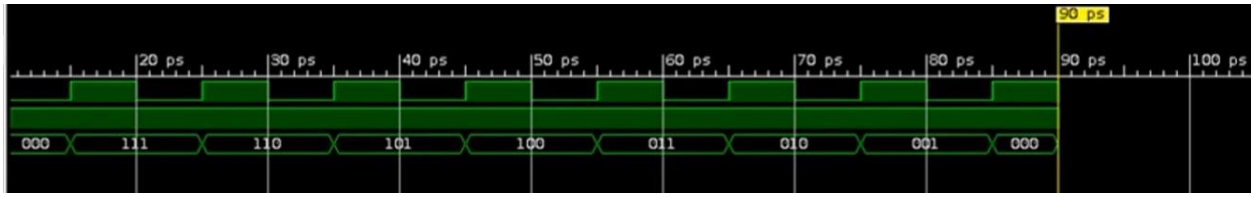





40 ps
0 ps
10 ps
20 ps
30 ps
40 ps
50 ps
60 ps
70 ps
80 ps
90 ps
000
111
110
101


20 ps
0 ps
10 ps
20 ps
30 ps
40 ps
50 ps
60 ps
70 ps
80 ps
90 ps
000
111


80 ps
10 ps
20 ps
30 ps
40 ps
50 ps
60 ps
70 ps
80 ps
90 ps
000
111
110
101
100
011
010
001


90 ps
20 ps
30 ps
40 ps
50 ps
60 ps
70 ps
80 ps
90 ps
100 ps
000
111
110
101
100
011
010
001
000