NATIONAL INSTITUTE OF TECHNOLOGY (NIT) DELHI

EXPERIMENT – 1

Basic Logic Gates

Objective: Implementation of all basic gates using Xilinx'sVivado.

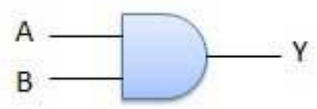
Requirements: Vivado Software

Introduction: Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a certain logic.

1. AND GATE- A circuit which performs an AND operation is shown in figure. It has n input (n >= 2) and one output.

Y	=	A AND B AND C ..... N
Y	=	A.B.C ..... N
Y	=	ABC ..... N

LOGIC DIAGRAM



TRUTH TABLE

Inputs		Output
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

Verilog Design-

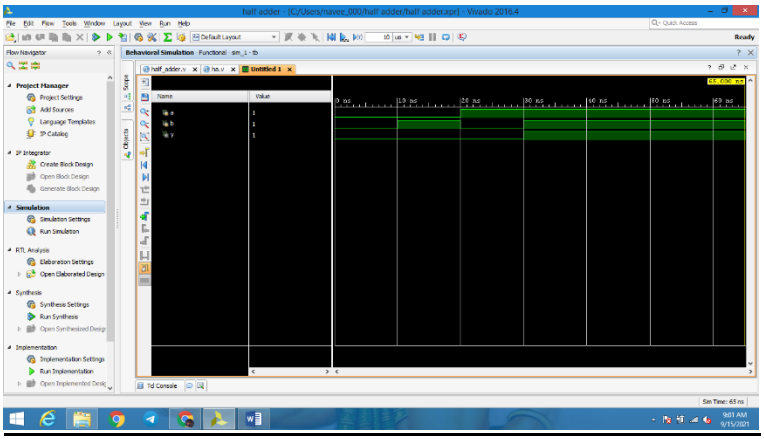
```
timescale 1ns / 1ps
module and_gate(a,b,y);
  input a,b;
  output y;
 assign y= a&b;
endmodule
```

TestBench-

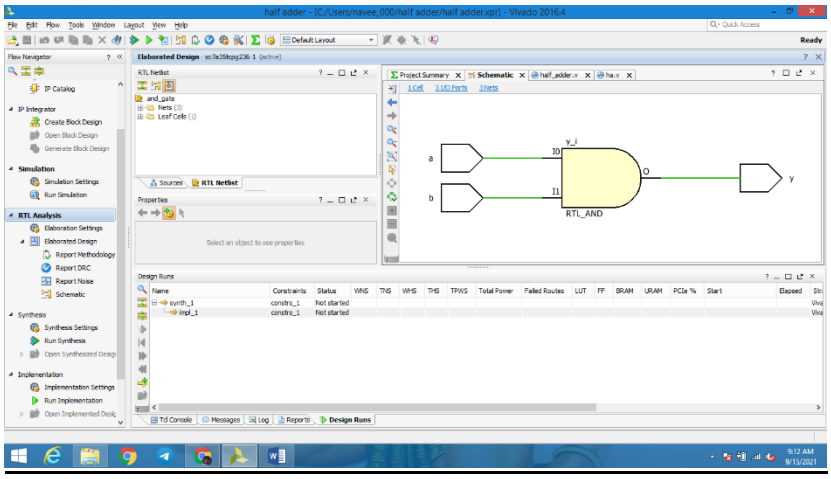
```
timescale 1ns / 1ps
module tb;
reg a,b;
wire y;
and_gate a1(a,b,y);
initial begin
a=0; b=0; #10;
a=0; b=1; #10;
a=1; b=0; #10;
a=1; b=1; #10;
#25 $finish;
end
endmodule
```

Results:-

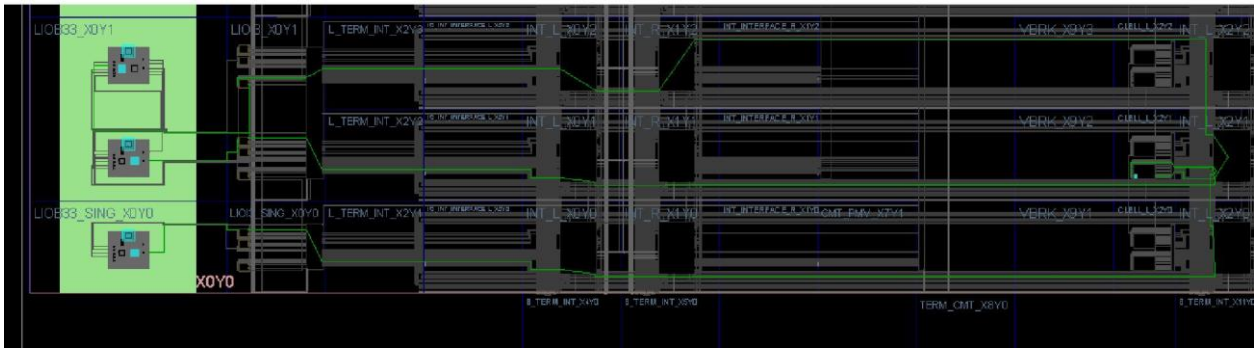
Simulation:-



RTL Analysis:-



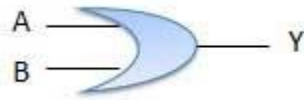
Implementation: -



2. Or Gate:-

A circuit which performs an OR operation is shown in figure. It has n input(n >= 2) and one output.

LOGIC DIAGRAM



TRUTH TABLE

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

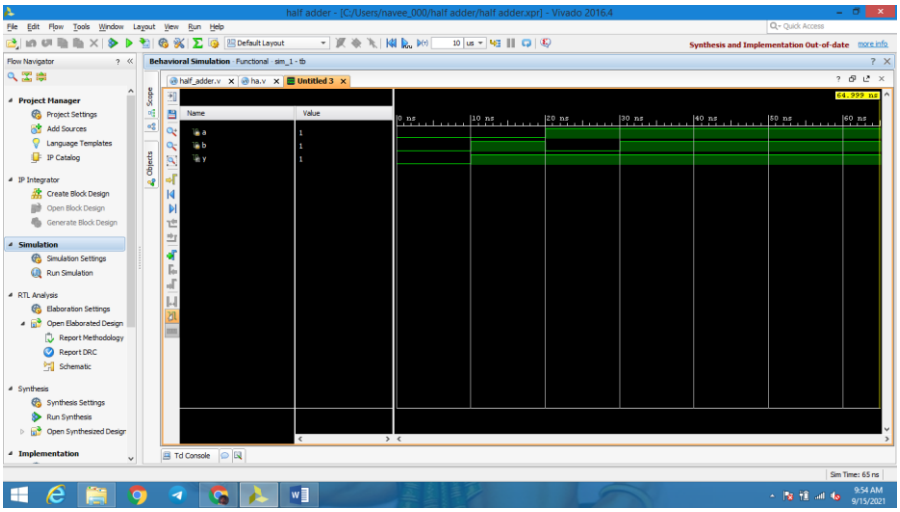
Code:-

```
`timescale 1ns / 1ps
module or_gate(a,b,y);
    input a,b;
    output y;
    assign y=a|b;
    endmodule
```

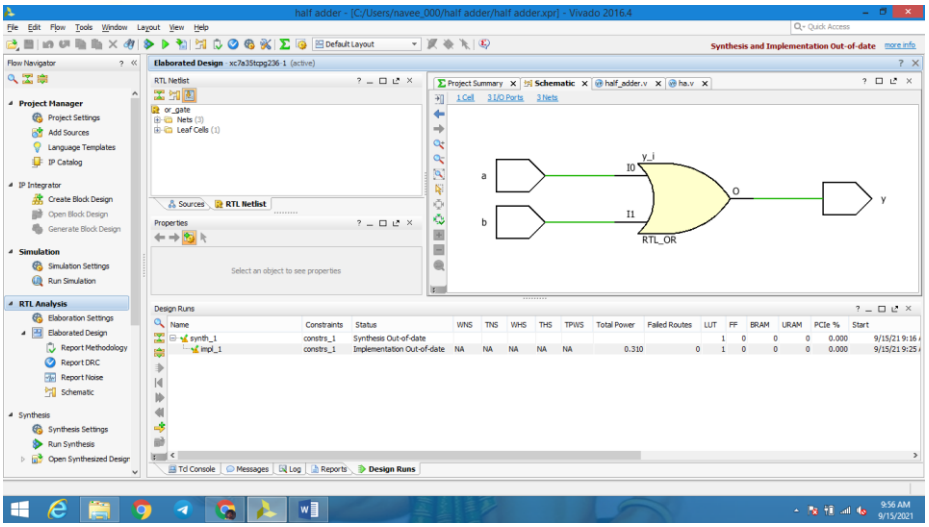
Testbench:-

```
`timescale 1ns / 1ps
module tb;
 reg a,b;
 wire y;
 or_gate a1(a,b,y);
 initial begin
   a=0; b=0; #10;
   a=0; b=1; #10;
   a=1; b=0; #10;
   a=1; b=1; #10;
   #25 $finish;
 end
endmodule
```

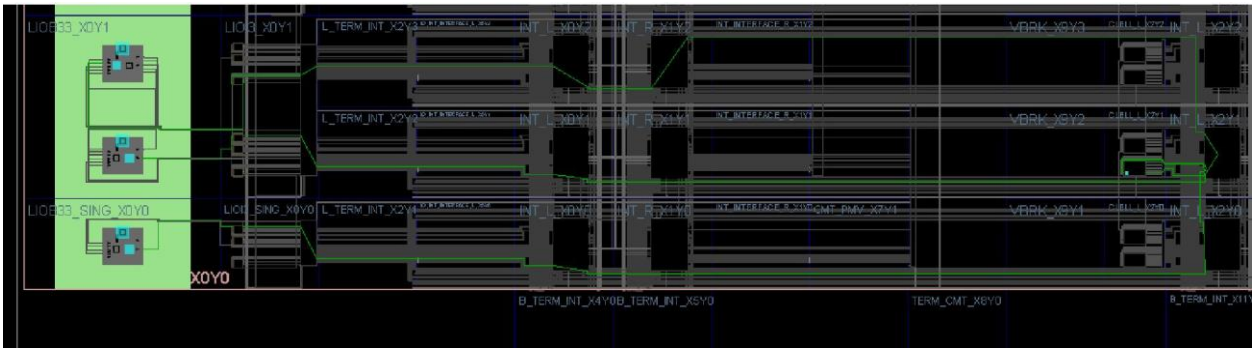
Simulation:-



RTL Analysis:-



Implementation: -



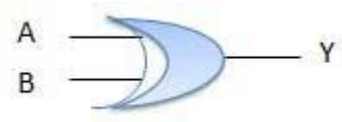
3. XOR GATE: XOR or Ex-OR gate is a special type of gate. It can be used in the half adder, full adder and subtractor. The exclusive-OR gate is abbreviated as EX-OR gate or sometime as X-OR gate. It has n input (n >= 2) and one output.

$$Y = A \text{ XOR } B \text{ XOR } C \ldots\ldots N$$

$$Y = A \oplus B \oplus C \ldots\ldots N$$

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

LOGIC DIAGRAM



TRUTH TABLE

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

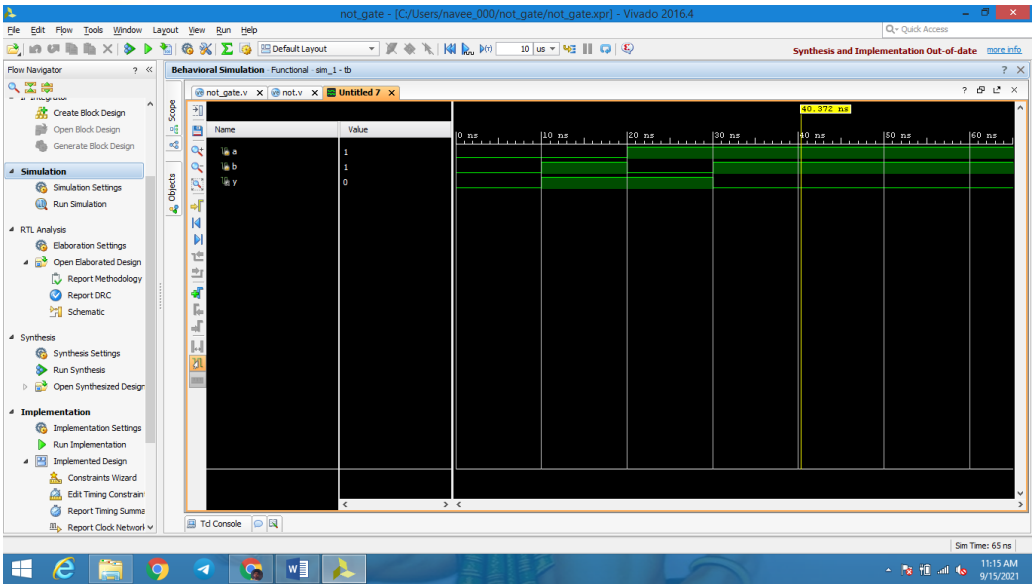
Code:-

```
`
timescale 1ns / 1ps
module xor_gate(a,b,y);
input a,b;
output y;
assign y =(a^b);
endmodule
```

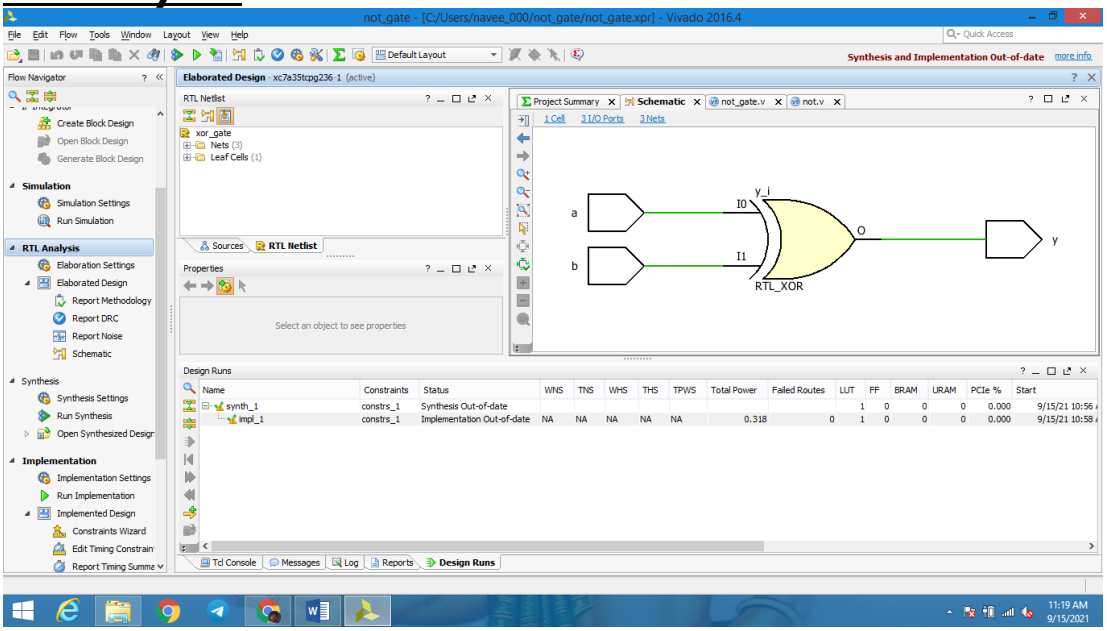
Testbench:-

```
`timescale 1ns / 1ps
module tb;
  reg a,b;
  wire y;

    xor_gate a1(a,b,y);
  initial begin
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    #25 $finish;
  end
endmodule
```

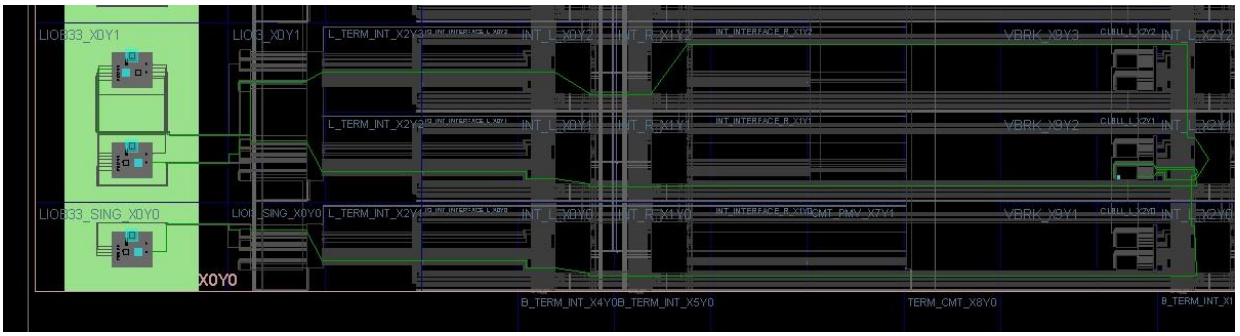
Simulation:-



RTL Analysis:-

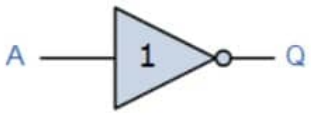


Implementation: -



4. Not Gate:- Also Known to be inverter.

Truth Table:-

Symbol	Truth Table	
 Inverter or NOT Gate	A	Q
	0	1
	1	0
Boolean Expression Q = NOT A , Ã		

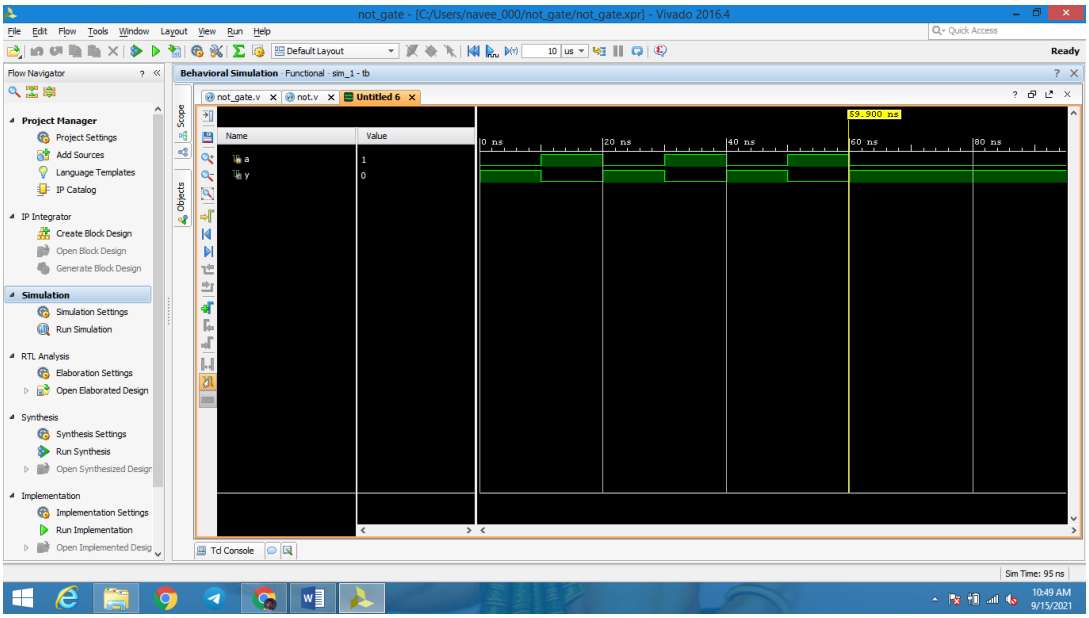
Code:-

```
`timescale 1ns / 1ps
module not_gate(a,y);
input a;
output y;
 assign y=~a;
endmodule
```

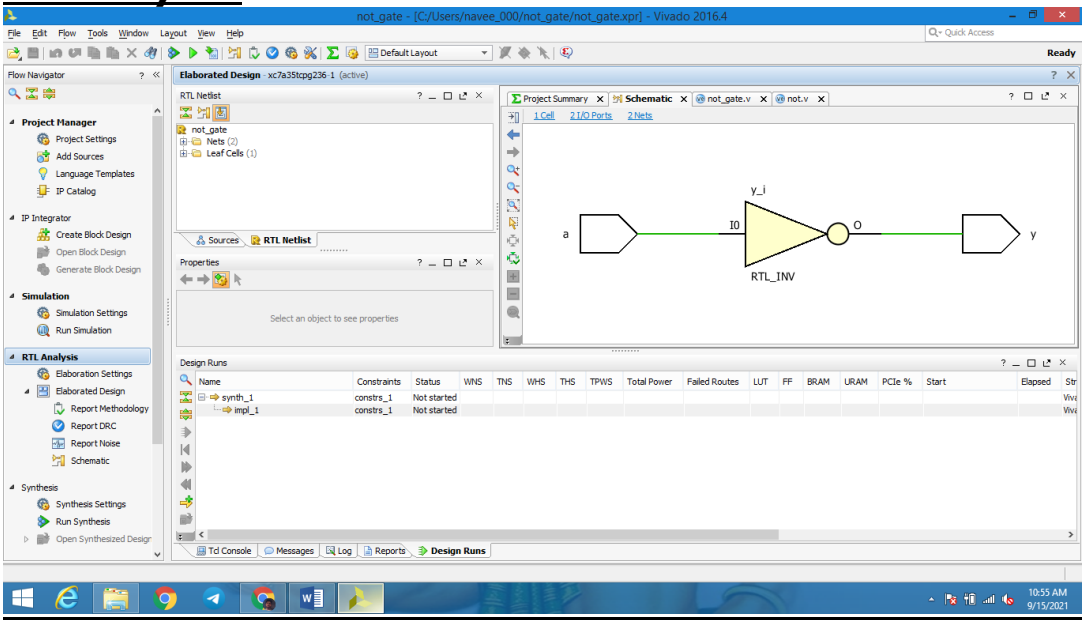
Testbench:-

```
`timescale 1ns / 1ps
module tb;
  reg a;
  wire y;
  not_gate a1(a,y);
  initial begin
    a=0; #10;
    a=1; #10;
    a=0; #10;
    a=1; #10;
    a=0; #10;
    a=1; #10;
    a=0; #10;
    #25 $finish;
  end
endmodule
```

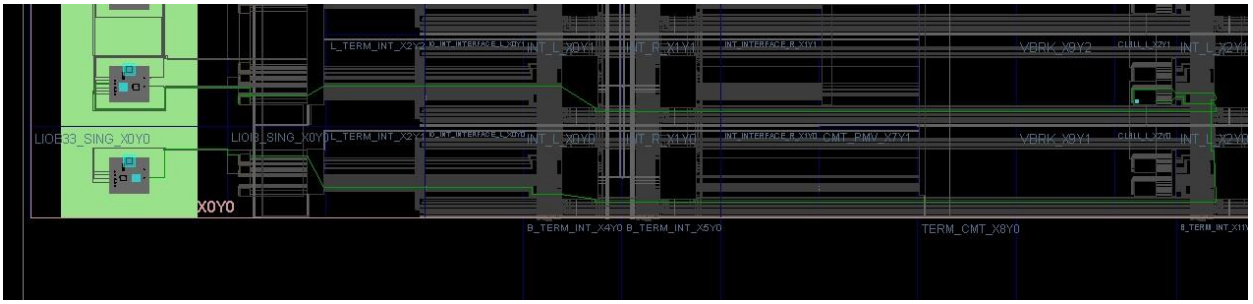
Simulation:-



RTL Analysis:-

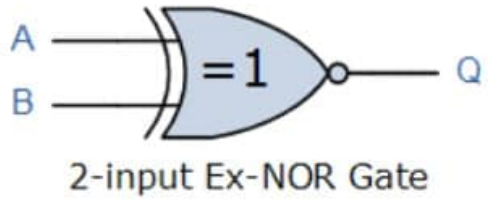


Implementation: -



5.Xnor Gate:-

Truth Table:-

<table>
<tr><th colspan="2">Symbol</th><th colspan="3">Truth Table</th></tr>
<tr><td colspan="2" rowspan="5"> 

2-input Ex-NOR Gate</td><td>A</td><td>B</td><td>Q</td></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="5">Boolean Expression Q = A XNOR B</td></tr>
</table>

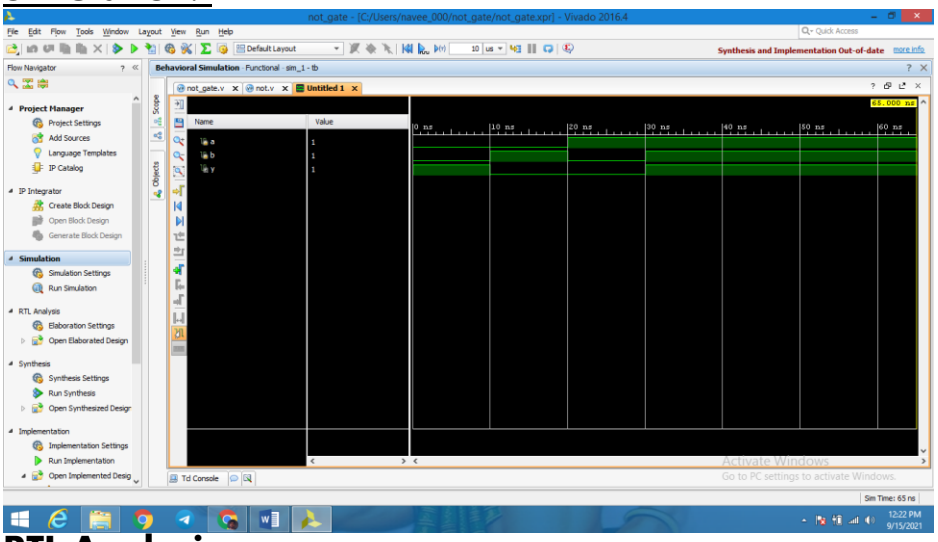
Code:-

```
`timescale 1ns / 1ps
module xnor_gate(a,b,y);
input a,b;
output y;
 assign y = ~(a^b);
 endmodule
```

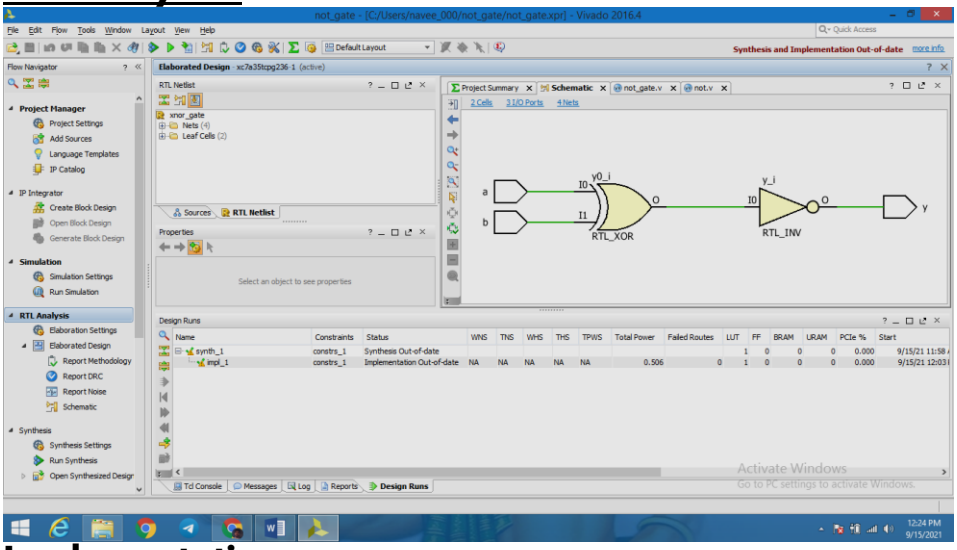
Testbench:-

```
`timescale 1ns / 1ps
module tb;
  reg a,b;
  wire y;
  xnor_gate a1(a,b,y);
  initial begin
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    #25 $finish;
  end
endmodule
```

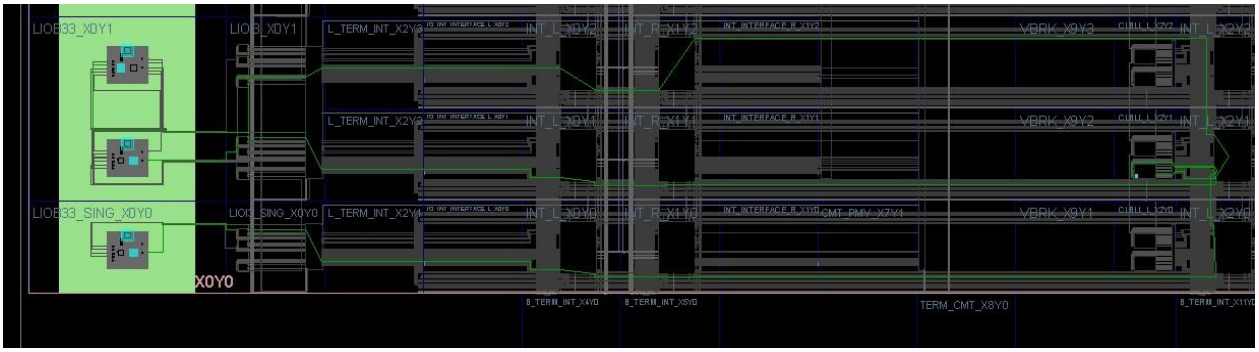
Simulation:-



RTL Analysis:-



Implementation: -



6.NAND Gate:-

Truth Table:-

Q = A NAND B

Truth Table

Input A	Input B		Output Q
0	0		1
0	1		1
1	0		1
1	1		0

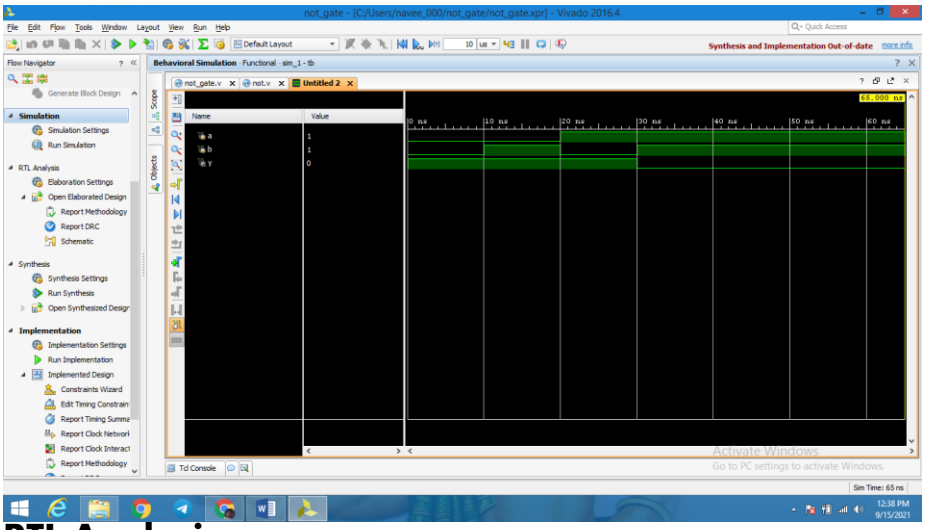
Code:-

```
`timescale 1ns / 1ps
module nand_gate(a,b,y);
 input a,b;
output y;
assign y = ~(a&b);
endmodule
```

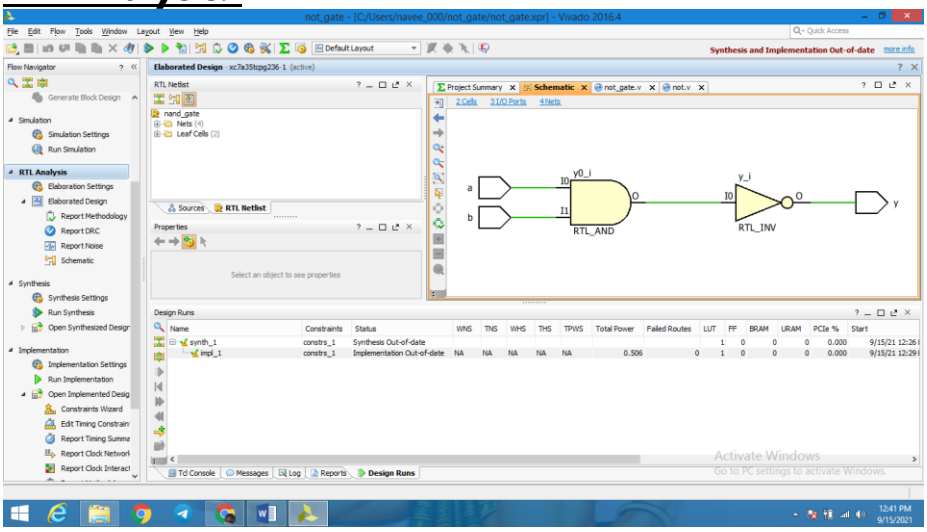
Testbench:-

```
`timescale 1ns / 1ps
module tb;
  reg a,b;
  wire y;
  nand_gate a1(a,b,y);
  initial begin
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    #25 $finish;
  end
endmodule
```

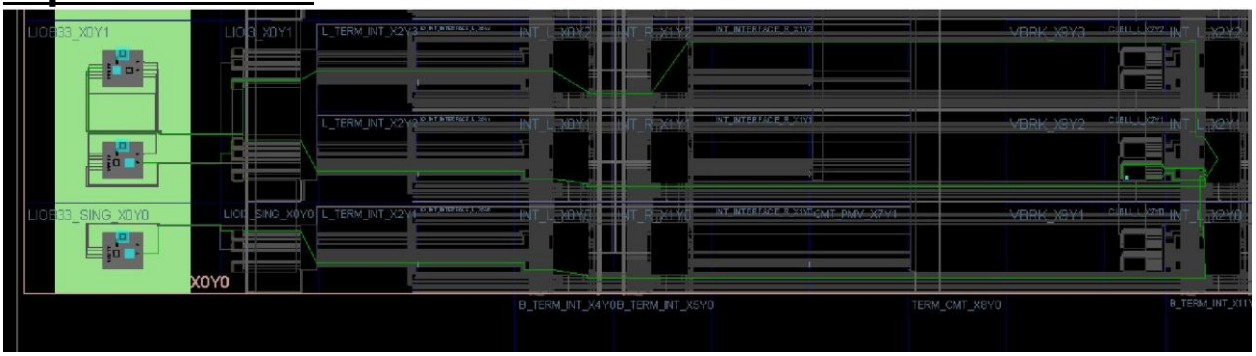
Simulation:-



RTL Analysis:-

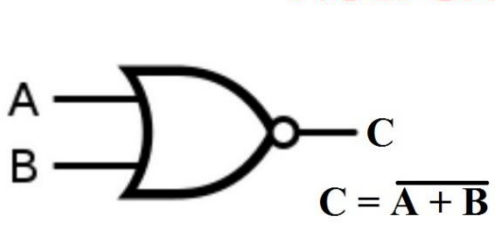


Implementation: -



7.NOR Gate:-

Truth Table:-



TRUTH TABLE

INPUT		OUTPUT
A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

ProjectIoT123.com

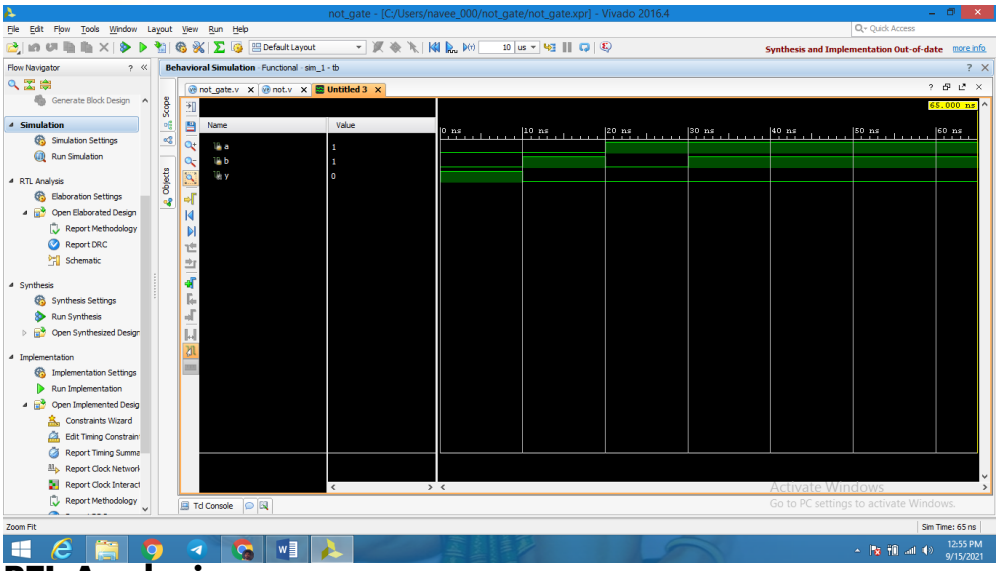
Code:-

```
`timescale 1ns / 1ps
module nor_gate(a,b,y);
input a,b;
output y;
assign y = ~(a|b);
endmodule
```

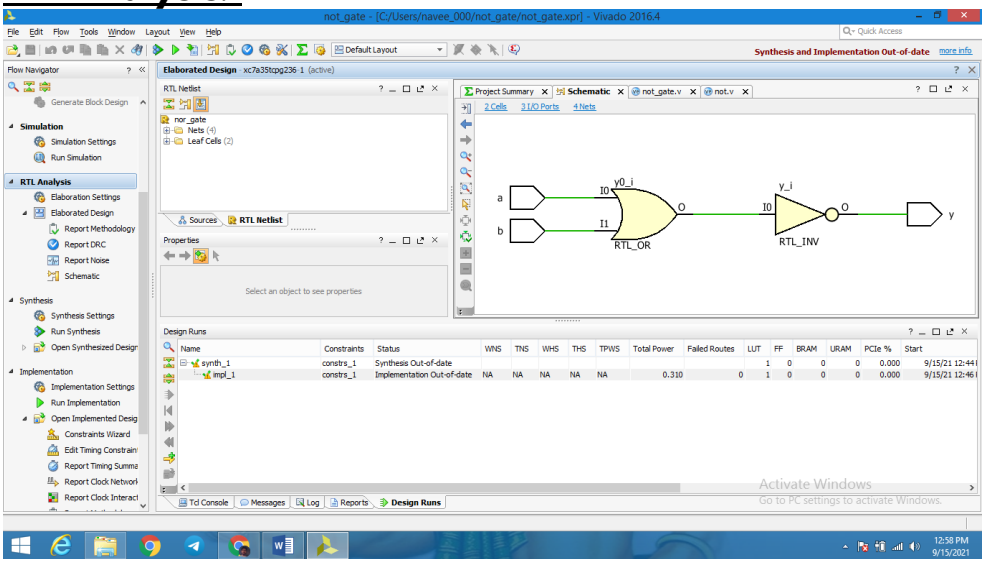
Testbench:-

```
`timescale 1ns / 1ps
module tb;
  reg a,b;
  wire y;
  nor_gate a1(a,b,y);
  initial begin
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    #25 $finish;
  end
endmodule
```

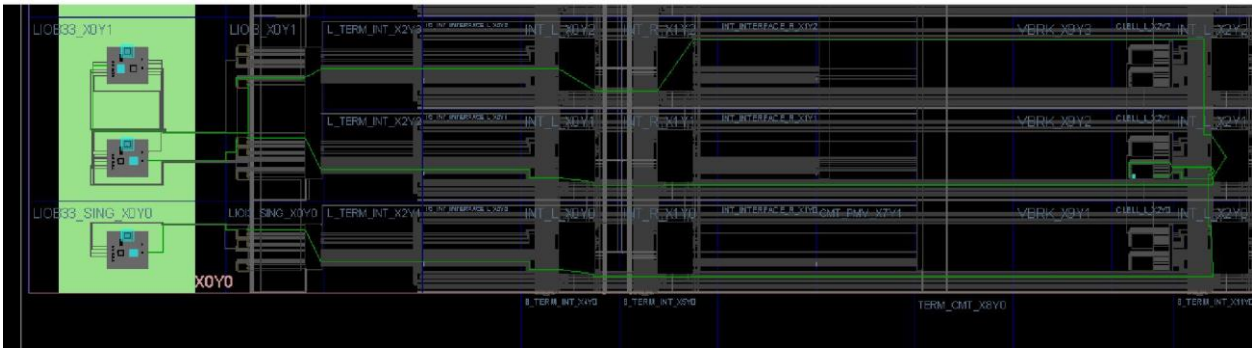
Simulation:-



RTL Analysis:-



Implementation: -



Learning Outcome: In this experiment I have learned how to design different types ofbasic gates.

Conclusion: The study of basic gates has been done successfully using Vivado.

NATIONAL INSTITUTE OF TECHNOLOGY (NIT) DELHI

EXPERIMENT – 2

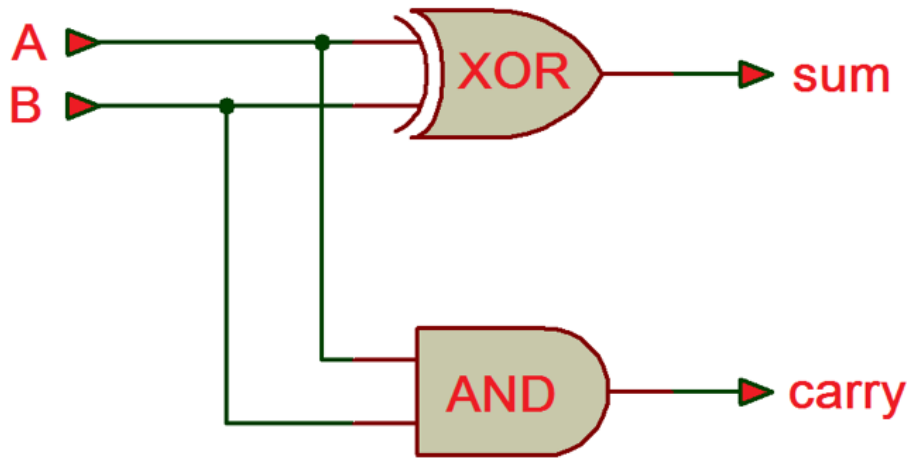
Binary Adder

Objective: Implementation of the HALF ADDER and FULL ADDER using Xilinx's Vivado.

Requirements: Xilinx's Vivado tool

Introduction: A Binary Adder is a digital circuit that performs the arithmetic sum oftwo binary numbers provided with any length.

HALF ADDER:_



Truth Table:-

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

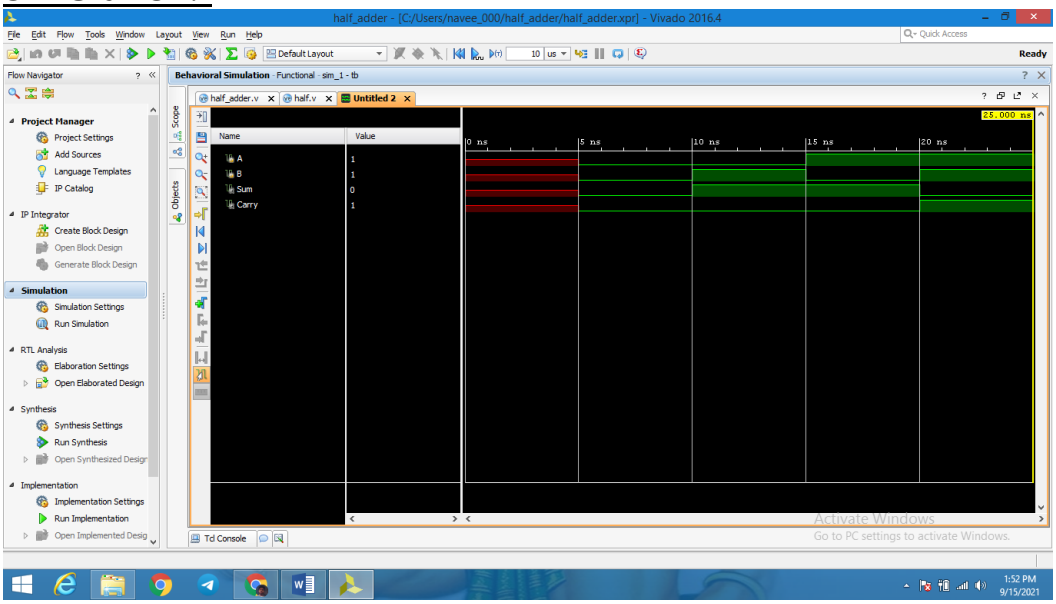
Code:-

```
module half_adder (Sum,Carry,A,B);
 input A,B;
 output Sum,Carry;
  assign Sum = A^B;
 assign Carry = A&B;
 endmodule
```

Testbench:-

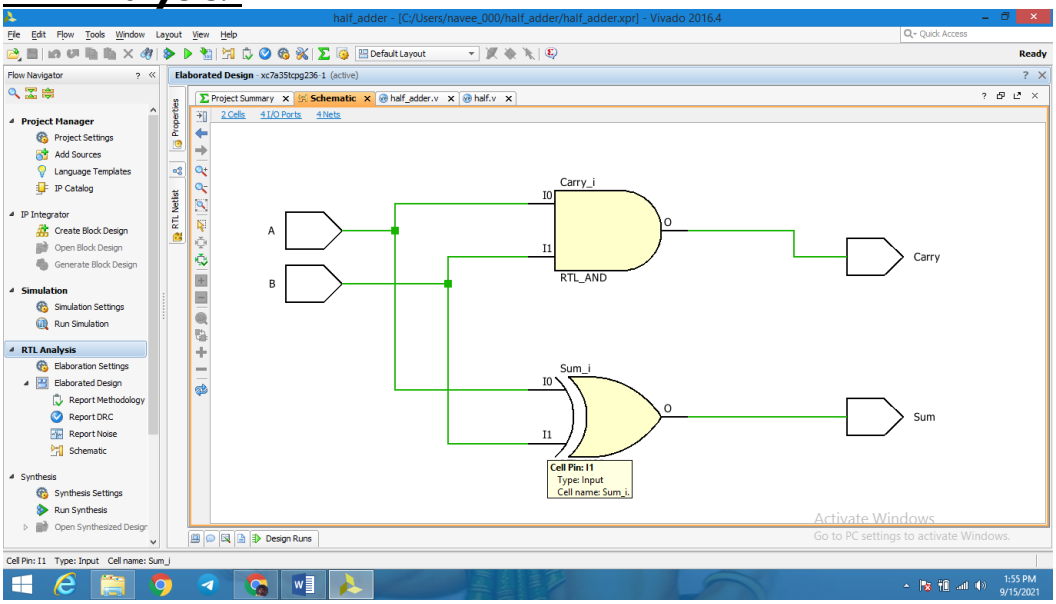
```
`timescale 1ns / 1ps
module tb;
reg A,B;
wire Sum,Carry;
half_adder a1(Sum,Carry,A,B);
 initial
 begin
 #5 A=0; B=0;
 #5 A=0; B=1;
 #5 A=1; B=0;
 #5 A=1; B=1;
 #5 $finish;
 end
endmodule
```

Simulation:-

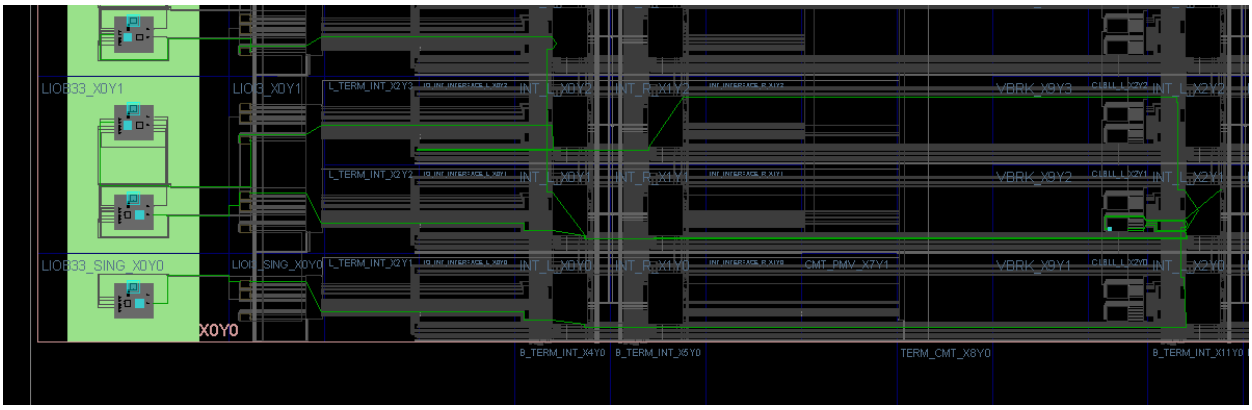


NATIONAL INSTITUTE OF TECHNOLOGY (NIT) DELHI

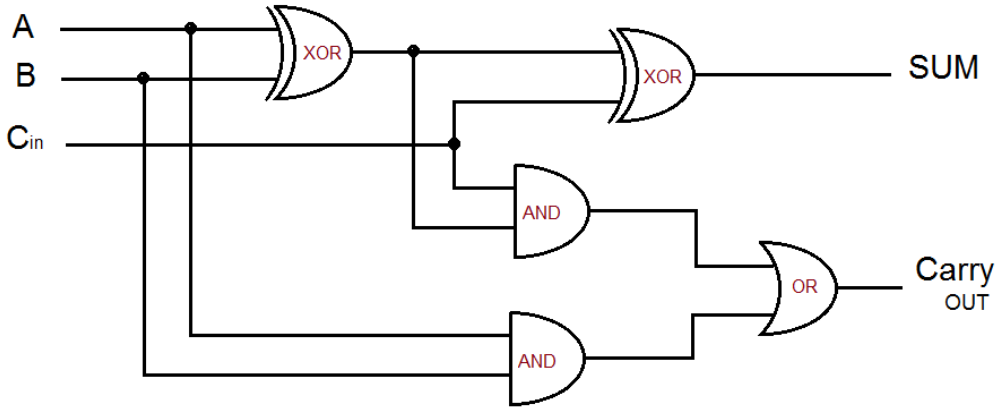
RTL Analysis:-



Implementation: -



FULL ADDER



Truth Table:-

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

Code:-

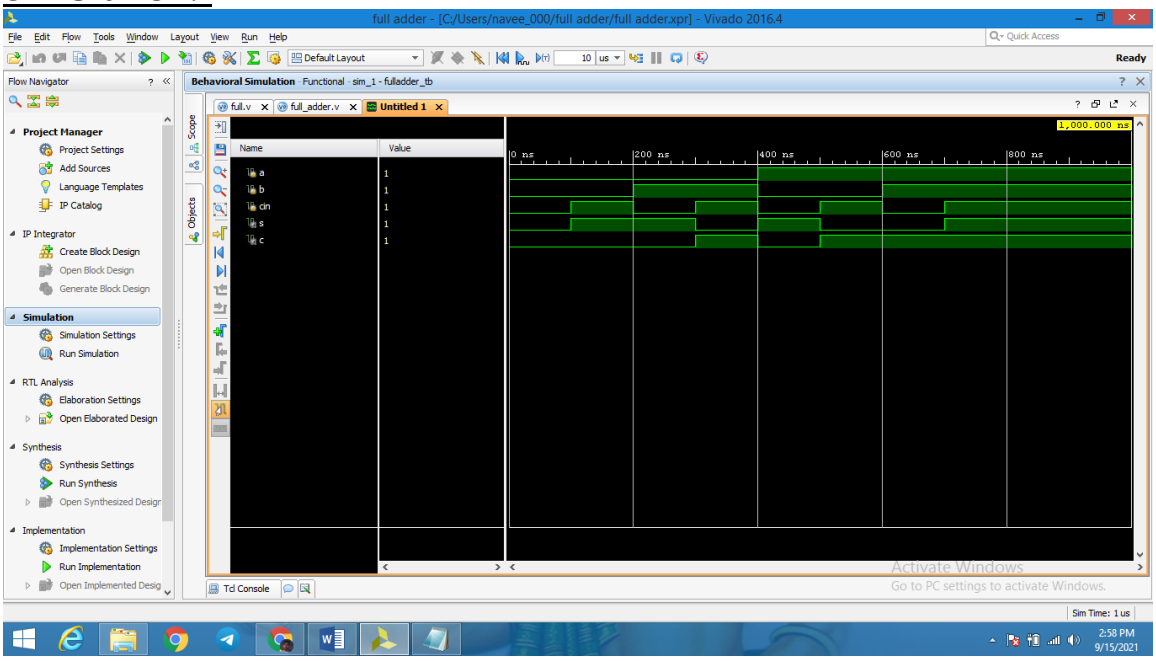
```
`timescale 1ns / 1ps
module fulladder(s,cout,a,b,cin);
input a,b,cin;
output s,cout;
wire i1,i2,i3;
xor(i1,a,b);
and(i2,a,b);
xor(s,cin,i1);
and(i3,i1,cin);
or(cout,i2,i3);
endmodule
```

Testbench:-

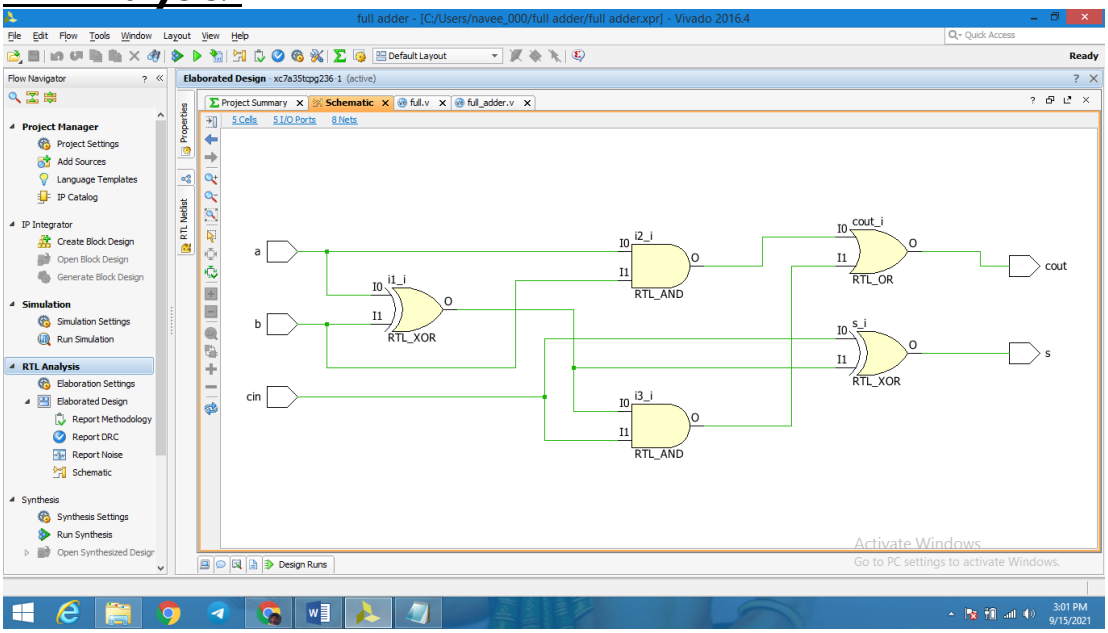
```
`timescale 1ns / 1ps
module fulladder_tb;
reg a,b,cin;
```

```
wire s,c;
 fulladder fulladder_tb(s,c,a,b,cin);
 initial
  begin
   #000 a=0; b=0; cin=0;
   #100 a=0; b=0; cin=1;
   #100 a=0; b=1; cin=0;
   #100 a=0; b=1; cin=1;
   #100 a=1; b=0; cin=0;
   #100 a=1; b=0; cin=1;
   #100 a=1; b=1; cin=0;
   #100 a=1; b=1; cin=1;
   end
    initial
     begin
      $monitor($time,"a=%b,b=%b,cin=%b,s=%b,c=%b",a,b,s,c,cin);
     end
    endmodule
```

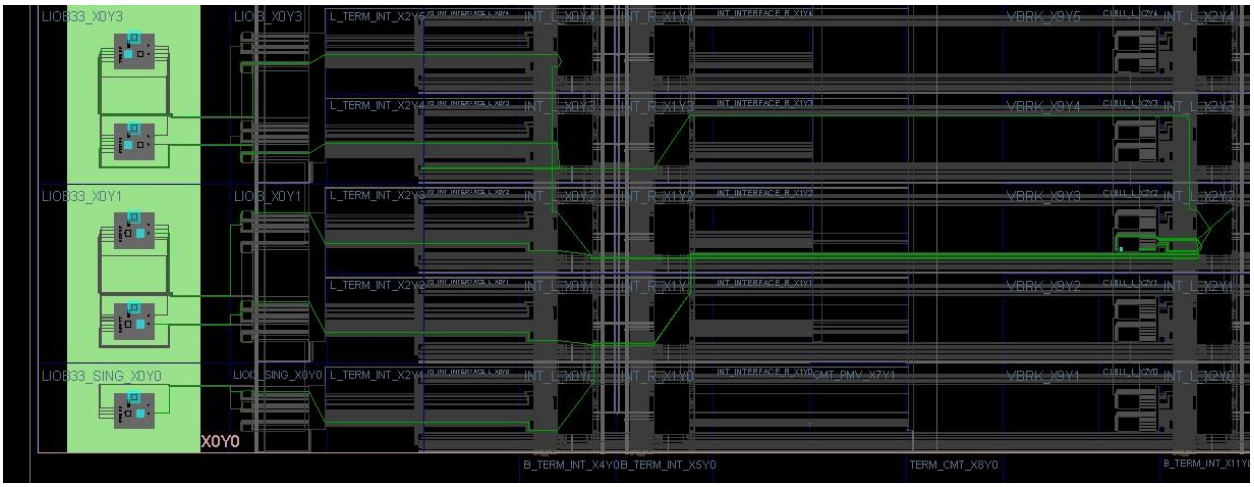
Simulation:-



RTL Analysis:-



Implementation: -



Conclusion:- The study has been done successfully using VIVADO tool.

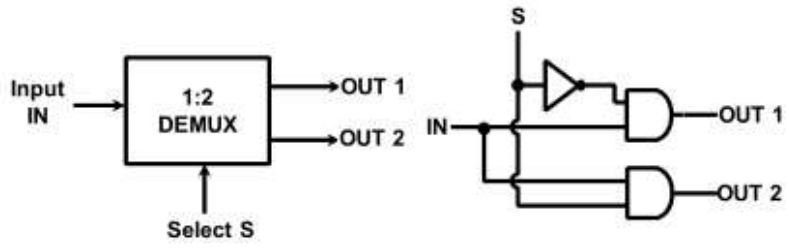
Learning Outcome: In this experiment I have learned how to design different types of Adders

Experiment 3

Multiplexers and Demultiplexers:-

Objective: Implementation of the 1:2 DeMux and 2:1 Mux using Xilinx's Vivado.
Requirements: Xilinx's Vivado tool
Introduction: Multiplexer is a special type of combinational circuit. There are n-data inputs, one output and m select inputs with 2m = n. It is a digital circuit which selects one of the n data inputs and routes it to the output. The selection of one of the n inputs is done by the selected inputs. Depending on the digital code applied at the selected inputs, one out of n data sources is selected and transmitted to the single output Y. E is called the strobe or enable input which is useful for the cascading. It is generally an active low terminal that means it will perform the required operation when it is low.

1:2 DEMUX



Select (S)	Input (IN)	Out 1	Out 2
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

Code:-

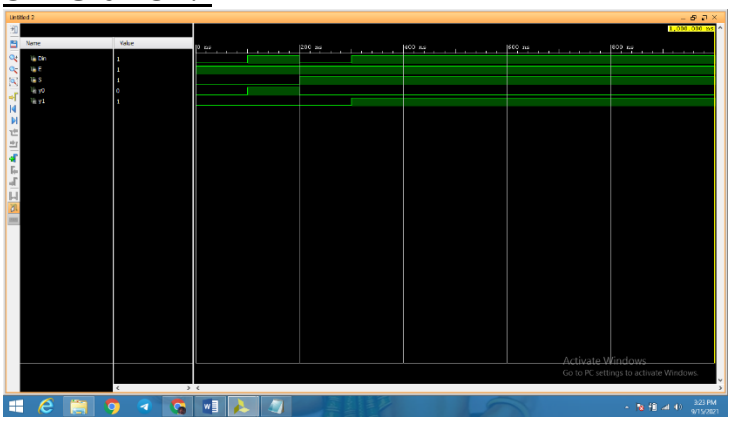
```
`timescale 1ns / 1ps
module demx(Din, E,S,y0,y1);
input Din,E,S;
output y0,y1;
assign y0 = (E & Din &(~S));
assign y1 = (E & Din & S);
endmodule
```

Testbench:-

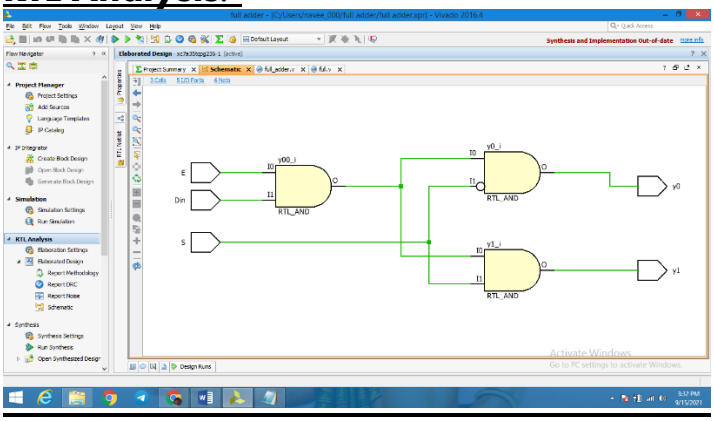
```
module demux1_tb;
reg Din,E,S;
wire y0,y1;
demx demux1_tb(Din, E,S,y0,y1);
initial
```

```
begin
#000 E=1'b1;Din=1'b0;S=1'b0;
#100 E=1'b1;Din=1'b1;S=1'b0;
#100 E=1'b1;Din=1'b0;S=1'b1;
#100 E=1'b1;Din=1'b1;S=1'b1;
end
initial
begin
$monitor($time,"Din=%b,E=%b,S=%b,y0=%b,y1=%b,",Din,E,S,y0,y1);
end

endmodule
```

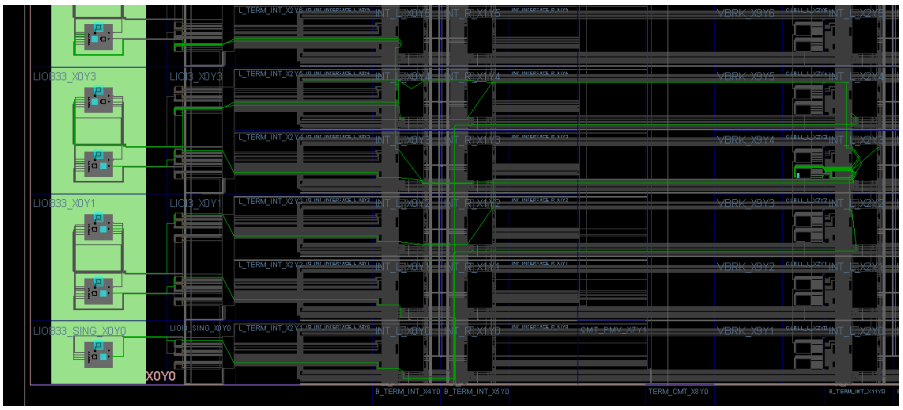
Simulation:-



RTL Analysis:-

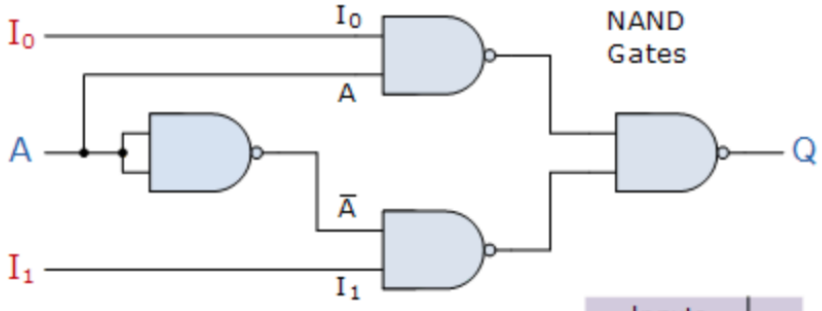


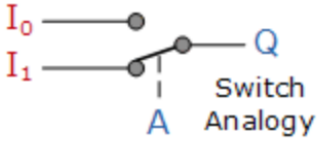
Implementation: -



2:1 MUX

2-input Multiplexer Design





Truth Table

Inputs			
A	I_1	I_0	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

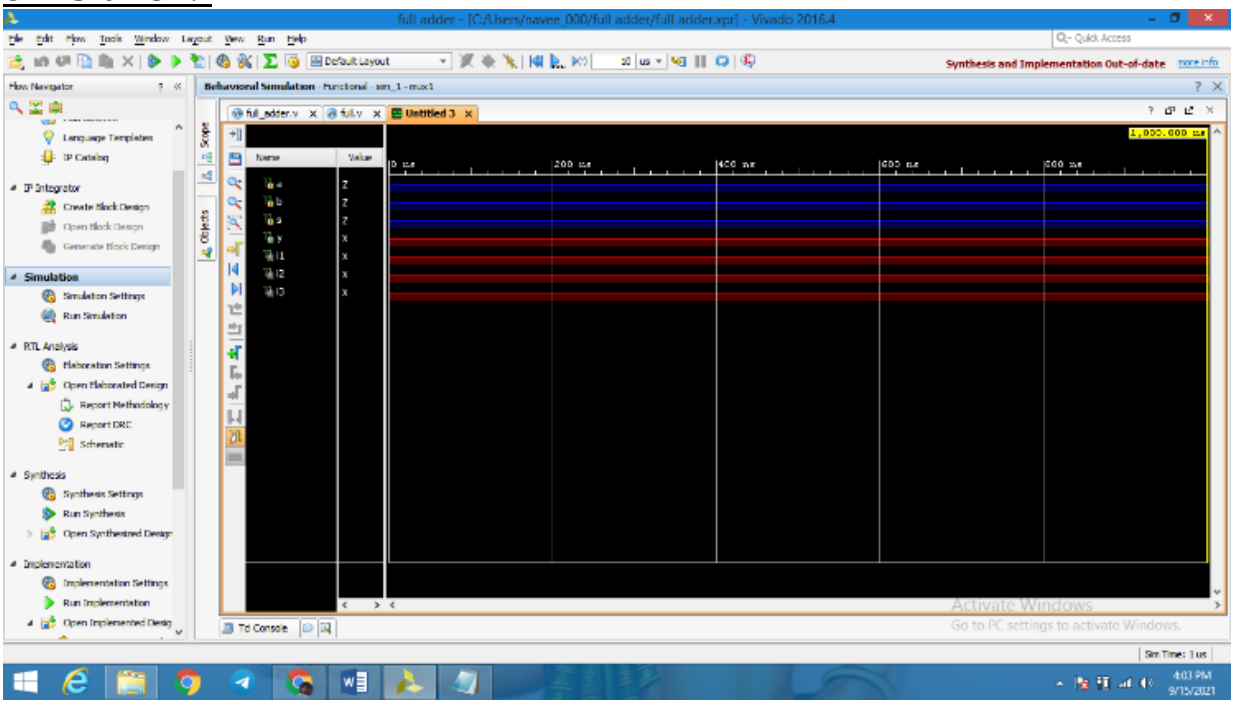
Code:-

```
`timescale 1ns / 1ps
module mux1(y,a,b,s,);
input a,b,s;
output y;
wire i1,i2,i3;
and(i1,a,i2);
not(i2,s);
and(i3,s,b);
or(y,i1,i3);
endmodule
```

Testbench:-

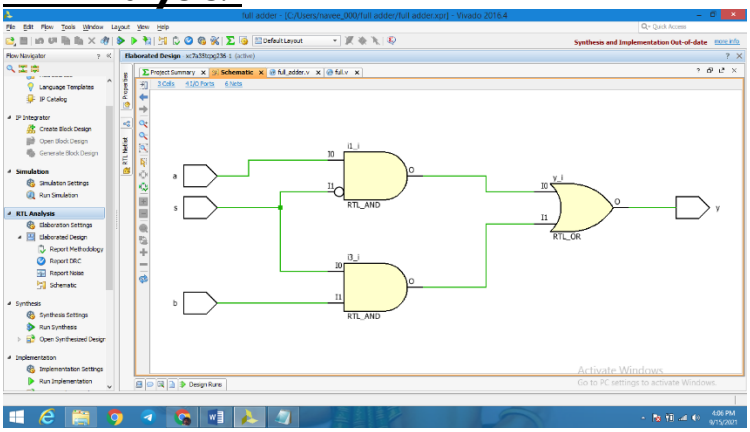
```
`timescale 1ns / 1ps
module mux1_tb;
reg a,b,s;
wire y;
mux1 mux1_tb(y,a,b,s);
initial
 begin
 #000 s=0; a=0; b=0;
 #100 s=0; a=0; b=1;
 #100 s=0; a=1; b=0;
 #100 s=0; a=1; b=1;
 #100 s=1; a=0; b=0;
 #100 s=1; a=0; b=1;
 #100 s=1; a=1; b=0;
 #100 s=1; a=1; b=1;
 end
  initial
   begin
   $monitor($time,"a=%b,b=%b,s=%b,y=%b",a,b,s,y);
   End
```

Simulation:-

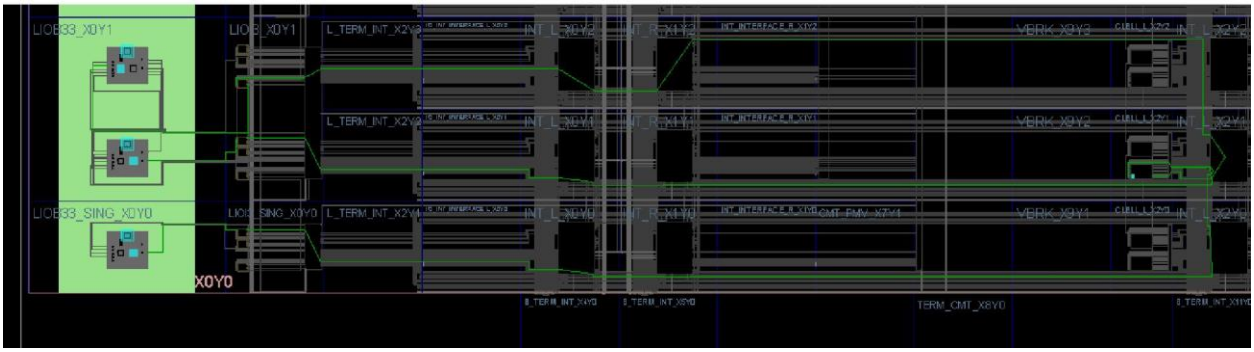


NATIONAL INSTITUTE OF TECHNOLOGY (NIT) DELHI

RTL Analysis:-



Implementation: -



Conclusion :- Verified using Verilog code.

Learning Outcome: In this experiment I have learned how to design different types of MUX/DEMUX.

Experiment- 4

Binary Subractors:-

Objective:- To verify half and full subracters using Verilog code.

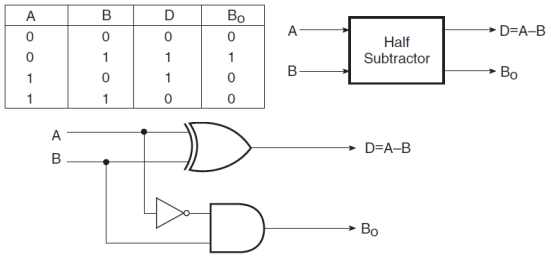
Introduction:- Subracters are basic thins used in digital electronics circuit for various operations.

Requirements:- Vivado software.

Half Subracter:-

Truth Table:-

A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



Code:-

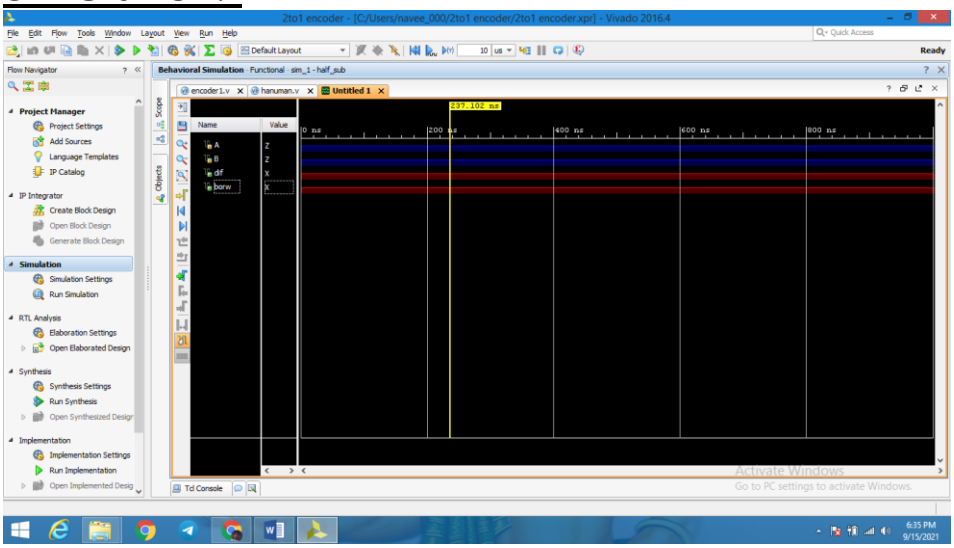
```
module half_sub(dif,borw,A,B);
 input A,B;
 output dif,borw;
 assign dif = (A^B);
 assign borw = (~A&B);
endmodule
```

Test Bench:-

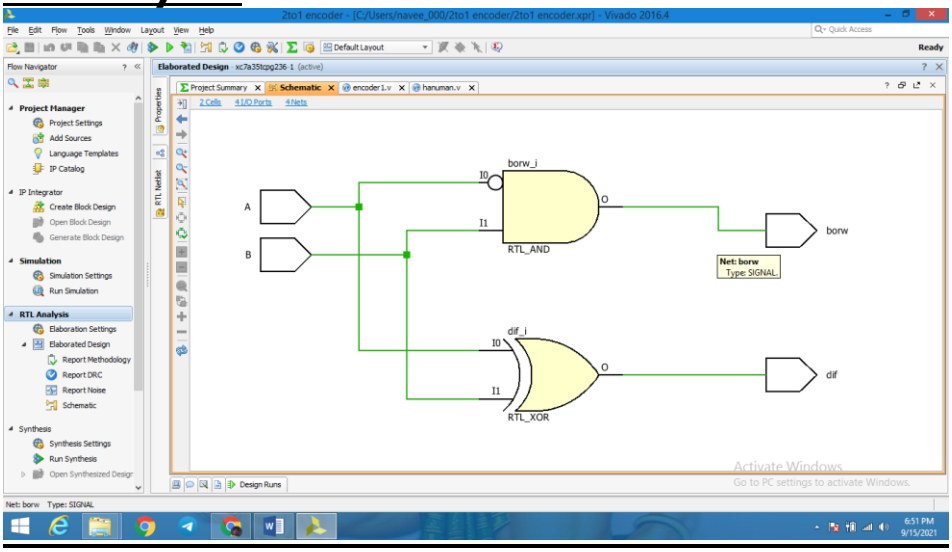
```
`timescale 1ns / 1ps
    module tb;
 reg a,b;
 wire c,d;
 half_sub A1(.A(a), .B(b), .dif(c), .borw(d));
 initial
 begin
 $dumpfile("dump.vcd");
 $dumpvars(0,tb);
```

```
#10 a=1'b0; b=1'b0;
#10 a=1'b0; b=1'b1;
#10 a=1'b1; b=1'b0;
#10 a=1'b1; b=1'b1;
#10 $finish;
end
initial
$monitor($time,"a=%b, b=%b, c=%b, d=%b",a,b,c,d);
Endmodule
```

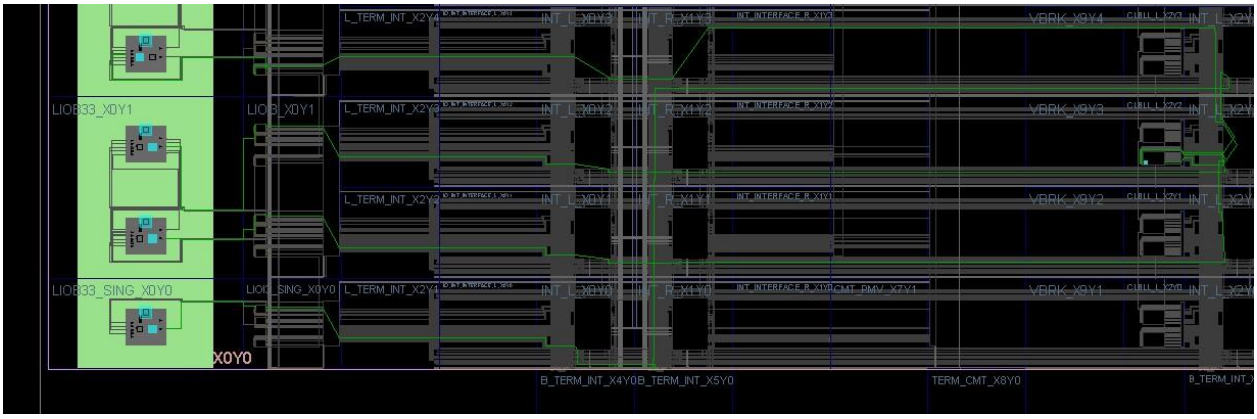
Simulation:-



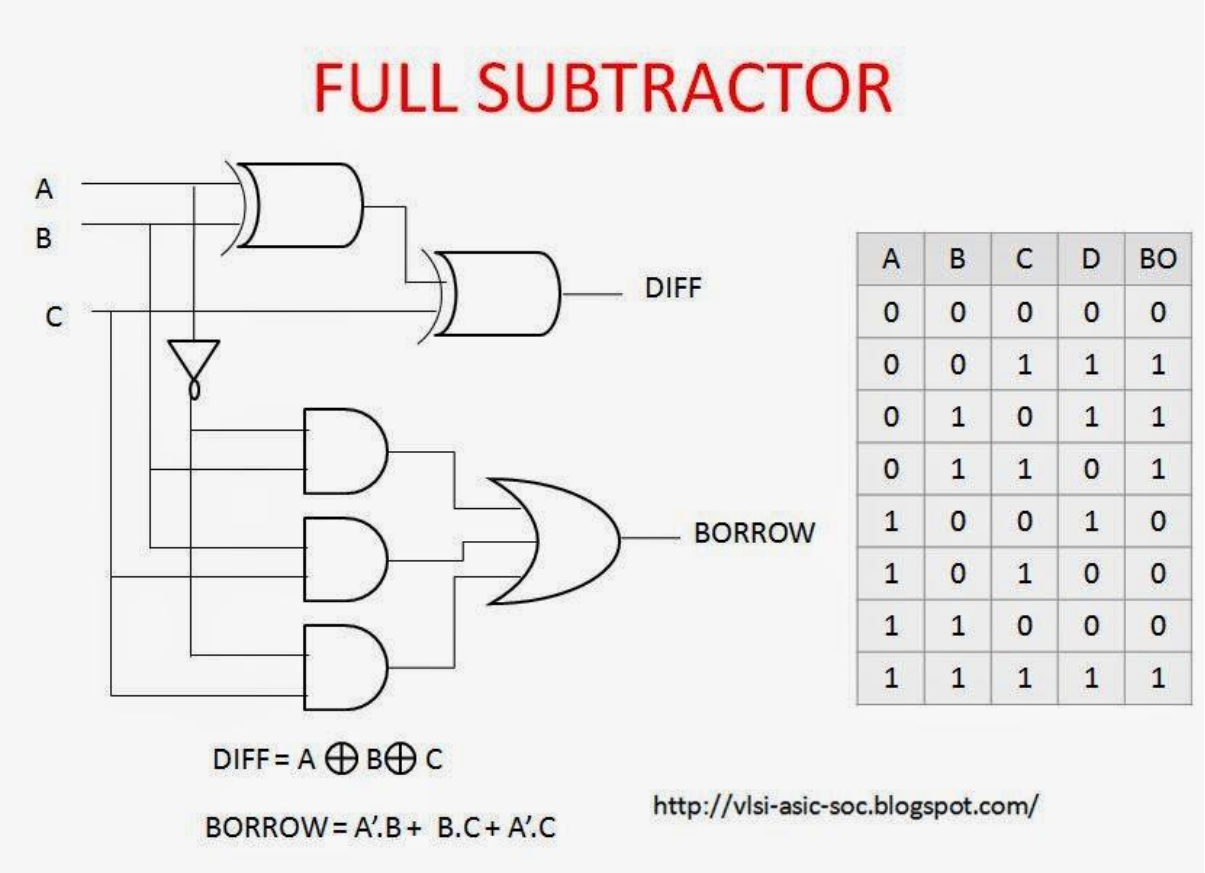
RTL Analysis:-



Implementation: -



FULL SUBTRACTOR



A	B	C	D	BO
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

Code:-

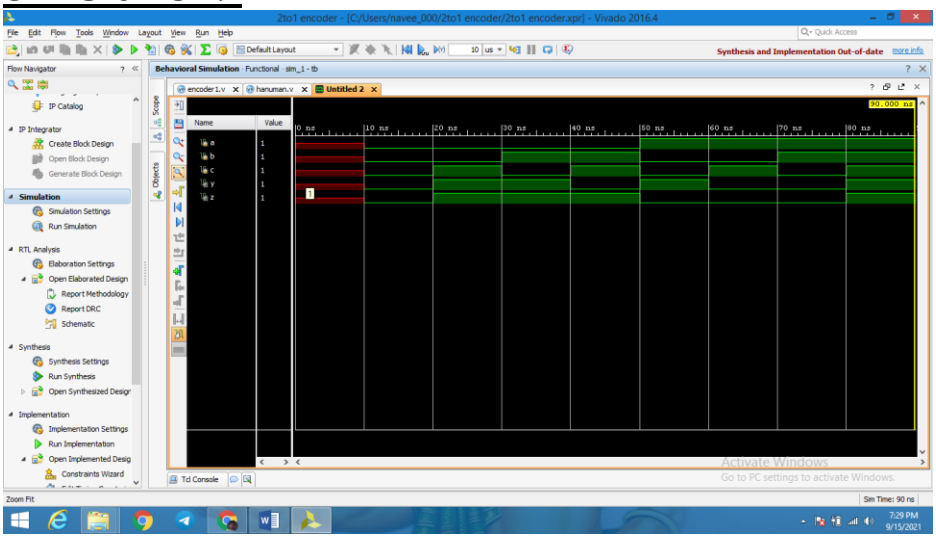
```
`timescale 1ns / 1ps
module full_sub(dif,borw,A,B,C);
 input A,B,C;
 output dif,borw;
 assign dif = (A^B^C);
 assign borw = (~A&(B^C))|(B&C);
endmodule
```

Testbench:-

```
`timescale 1ns / 1ps
```

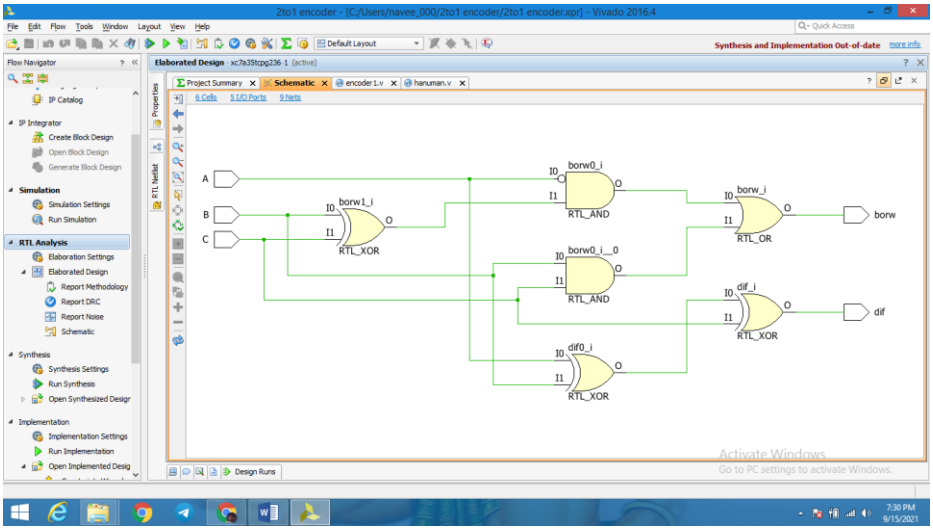
```
module tb;
 reg a,b,c;
 wire y,z;
 full_sub A1(.A(a), .B(b), .C(c), .dif(y), .borw(z));
 initial
 begin
 #10 a=1'b0; b=1'b0; c=1'b0;
 #10 a=1'b0; b=1'b0; c=1'b1;
 #10 a=1'b0; b=1'b1; c=1'b0;
 #10 a=1'b0; b=1'b1; c=1'b1;
 #10 a=1'b1; b=1'b0; c=1'b0;
 #10 a=1'b1; b=1'b0; c=1'b1;
 #10 a=1'b1; b=1'b1; c=1'b0;
 #10 a=1'b1; b=1'b1; c=1'b1;
 #10 $finish;
 end
 initial
 $monitor($time,"a=%b, b=%b, c=%b, y=%b, z=%b",a,b,c,y,z);
 Endmodule
```

Simulation:-

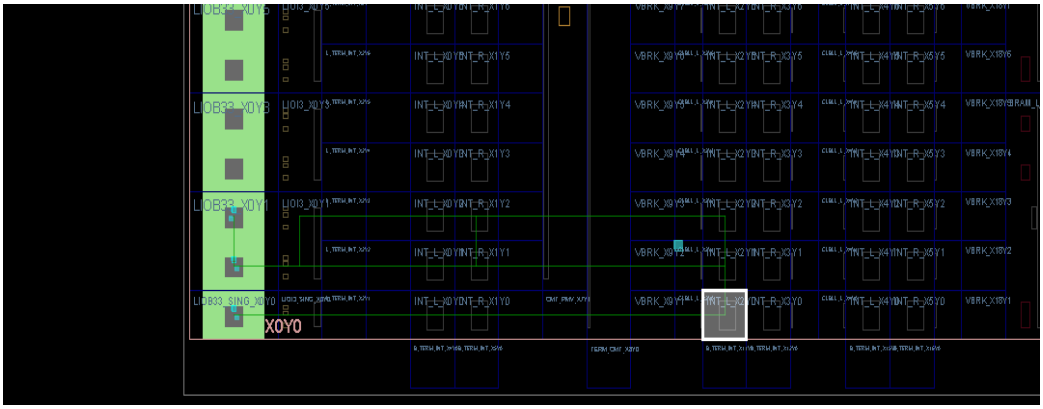


Schematic:-

NATIONAL INSTITUTE OF TECHNOLOGY (NIT) DELHI



Implementation: -



Conclusion:- Verified.

Learning Outcome: In this experiment I have learned how to design different types of Subtractors.

Experiment 5

Flip Flop:-

Objective: Implementation of the D flip flop, T flip flop, SR and JK flip flop using Xilinx's Vivado.

Requirements: Xilinx's Vivado tool

Introduction: Now a days flip flop are used as temporary memory in various electronics circuit.

D-Flip Flop:-

D Flip-flop

Symbol

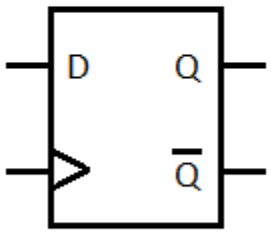


Table of truth:

clk	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

Code:-

```
`timescale 1ns / 1ps
module dff(input clk,rstn,d,output reg q);
always @(posedge clk)
begin
if (!rstn)
q=0;
else
q=d;
```

```
    end
endmodule
```

Testbench:-

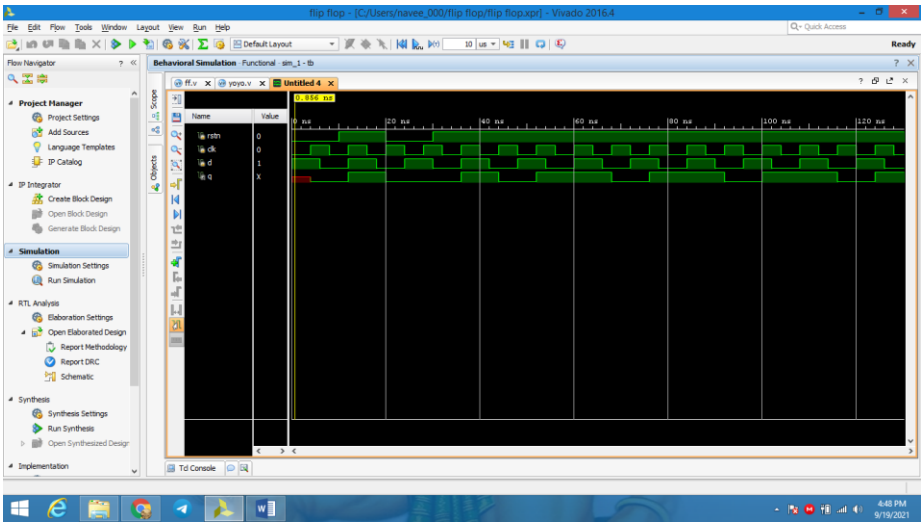
```
    `timescale 1ns / 1ps

    module tb;
    reg rstn,clk,d;
    wire q;
    dff x(clk,rstn,d,q);
    initial begin
    rstn=0; clk=0; d=1; #10;
    rstn=1; #10;
    rstn=0; #10;
    rstn=1; #10;
    #90 $finish;
    end
    always #4 clk=~clk;
    always #6 d=~d;
endmodule
```

Testbench:-

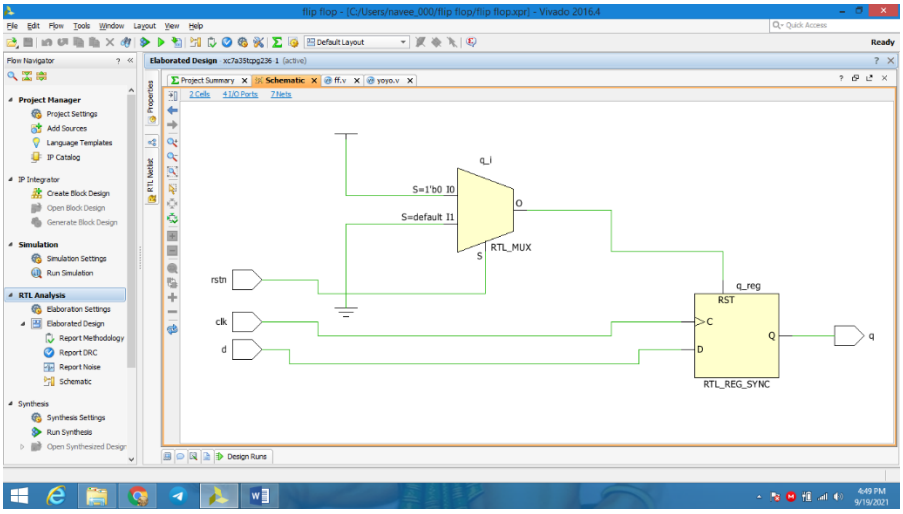
```
    `timescale 1ns / 1ps

    module tb;
    reg rstn,clk,d;
    wire q;
    dff x(clk,rstn,d,q);
    initial begin
    rstn=0; clk=0; d=1; #10;
    rstn=1; #10;
```

```
    rstn=0; #10;
    rstn=1; #10;
    #90 $finish;
    end
    always #4 clk=~clk;
    always #6 d=~d;
endmodule
```

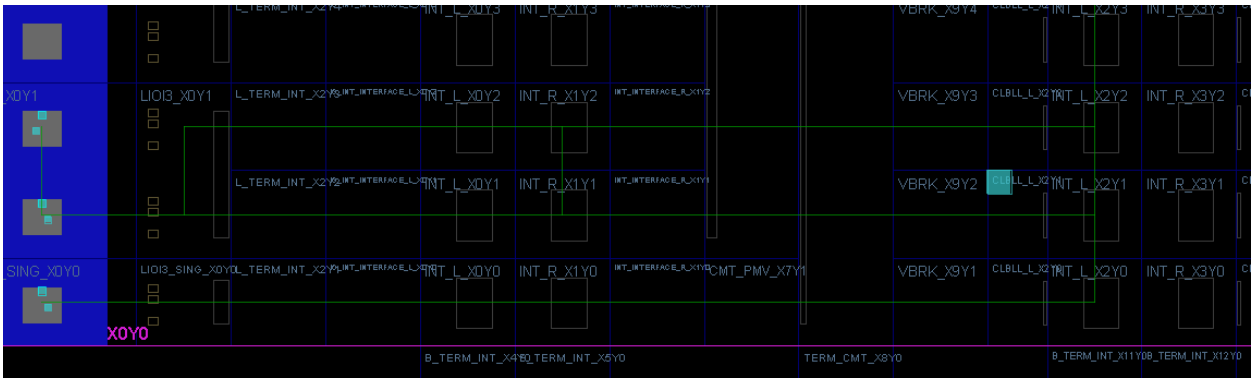
Simulation:-



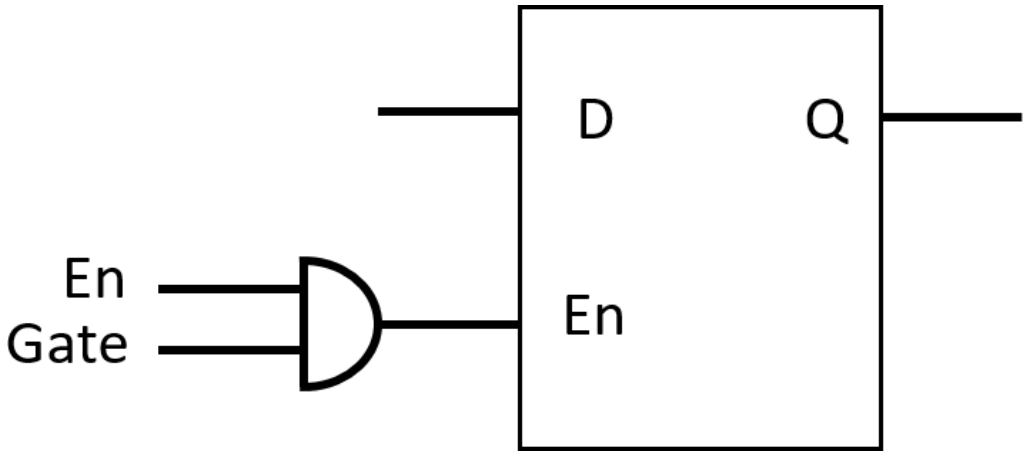
RTL Analysis: -



Implementation: -



D-Flip Flop with Gated Enable: -



Input			Output
En	Gate	D	Q
0	0	0	z
0	0	1	z
0	1	0	z
0	1	1	z
1	0	0	z

Code: -

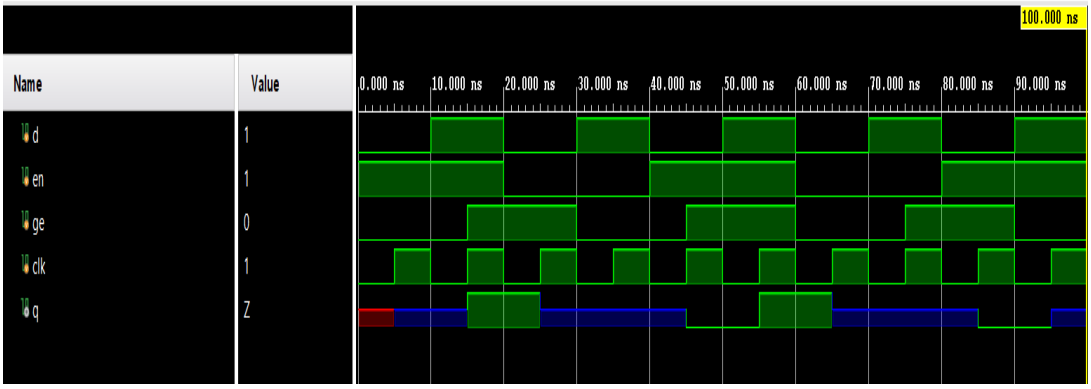
```
module DFF_en(Q,EN,GE,D,clk);
input EN,GE,D,clk;
output reg Q;
always @(posedge clk)
begin
if(EN&GE)
Q<=D;
else
Q<=1'bz;
end
endmodule
```

Testbench: -

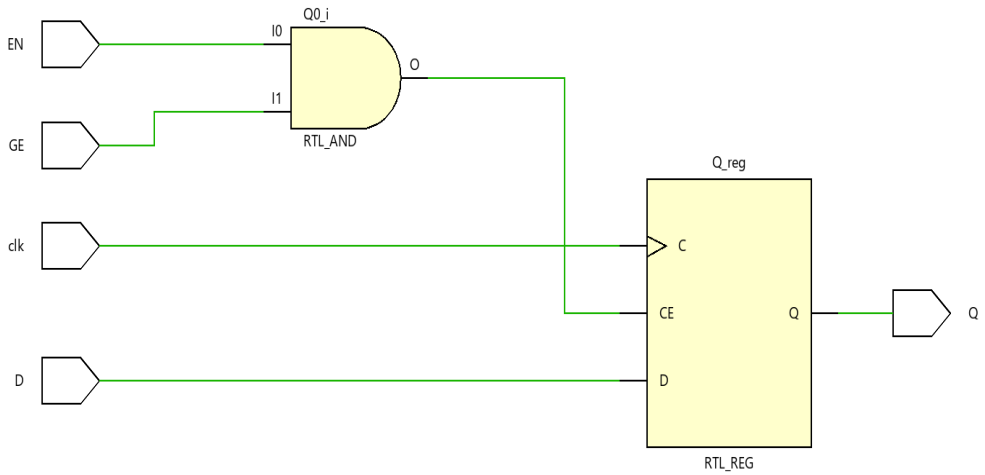
```
module DFF_en_t();
reg d,en,ge,clk;
wire q;
DFF_en DFF(.Q(q),.EN(en),.GE(ge),.D(d),.clk(clk));
initial
begin
d=1'b0;
en=1'b1;
ge=1'b0;
clk=1'b0;
#100 $finish;
end
always #5 clk=~clk;
always #15 ge=~ge;
always #10 d=~d;
always #20 en=~en;
endmodule initial
begin
d=1'b0;
```

```
en=1'b1;
ge=1'b0;
clk=1'b0;
#100 $finish;
end
always #5 clk=~clk;
always #15 ge=~ge;
always #10 d=~d;
always #20 en=~en;
```

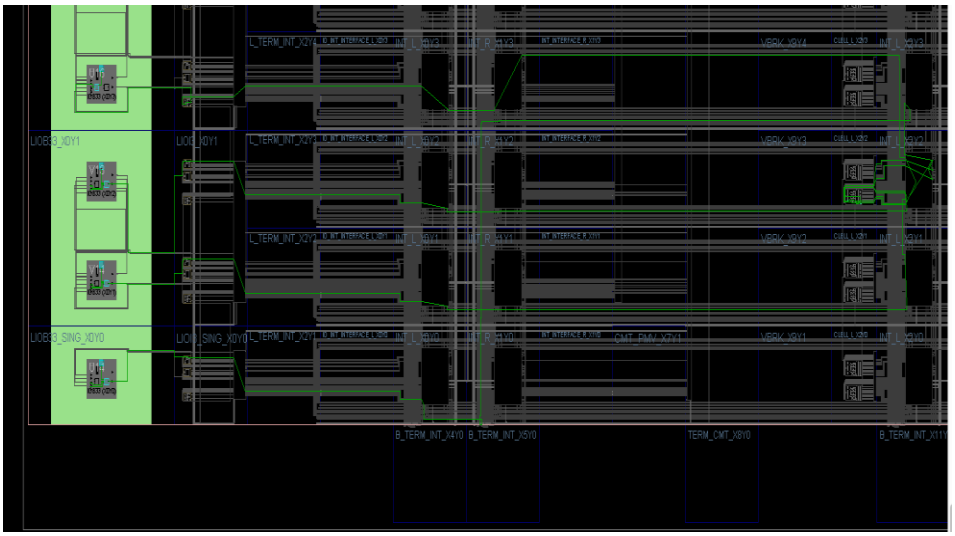
Simulation: -



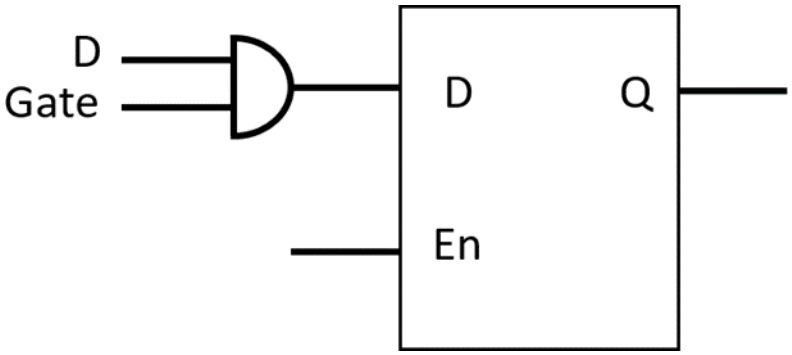
RTL Analysis: -



Implementation: -



D-Flip Flop with Gated Data: -



Input			Output
En	Gate	D	Q
0	0	0	z
0	0	I	z
0	I	0	z
0	I	I	z
I	0	0	0
I	0	I	0
I	I	0	0
I	I	I	I

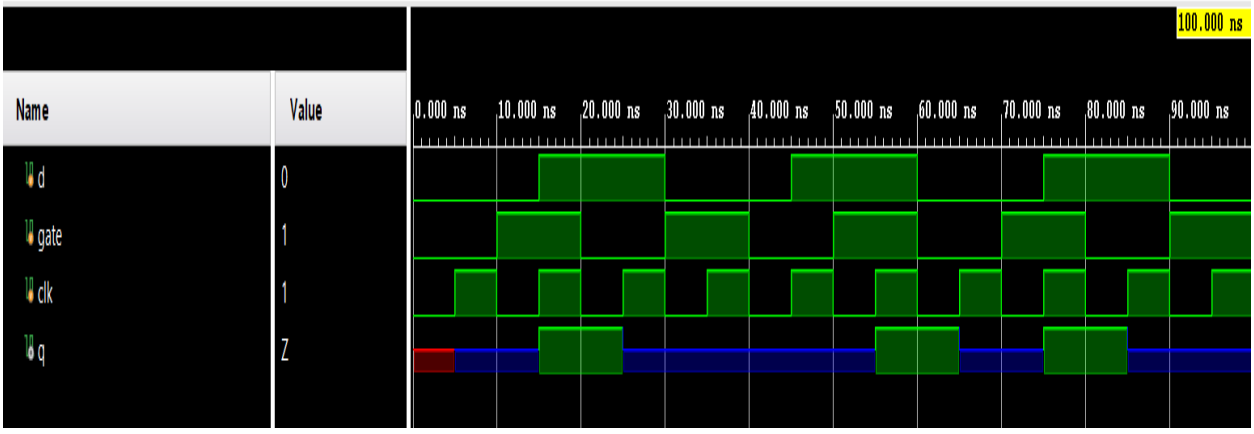
Code: -

```
module dff_gatedinput(D,GATE,CLK,Q);
input D,GATE,CLK;
output reg Q;
always @(posedge CLK)
begin
if(GATE&D)
Q<=D;
else
Q<=1'bz;
end
endmodule
```

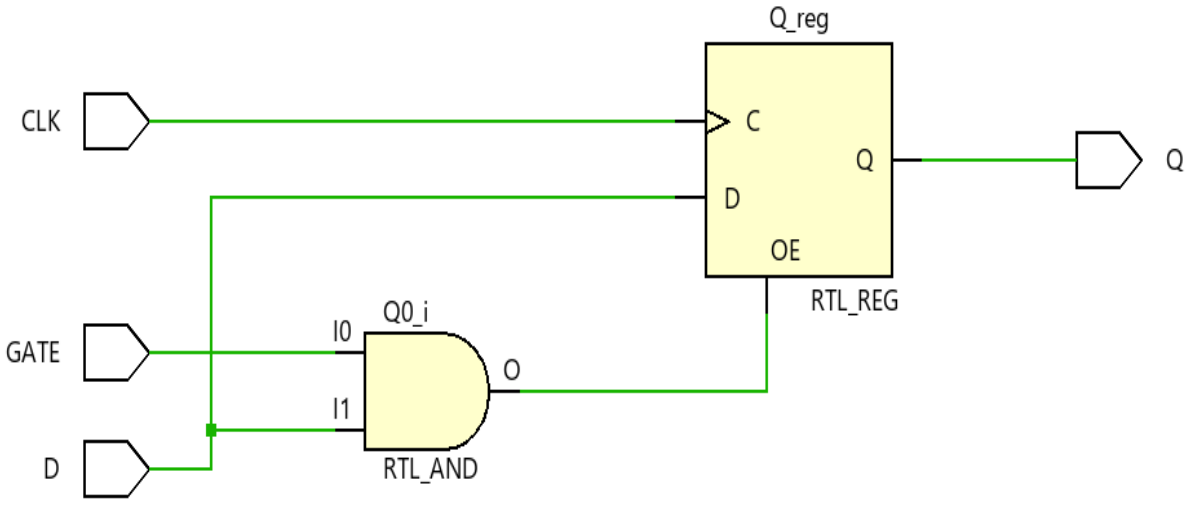
Testbench: -

```
module dff_gatedin_t();
reg d,gate,clk;
wire q;
dff_gatedinput FF(.D(d),.GATE(gate),.CLK(clk),.Q(q));
initial
begin
d=1'b0;
gate=1'b0;
clk=1'b0;
#100 $finish;
end
always #5 clk=~clk;
always #10 gate=~gate;
always #15 d=~d;
endmodule
```

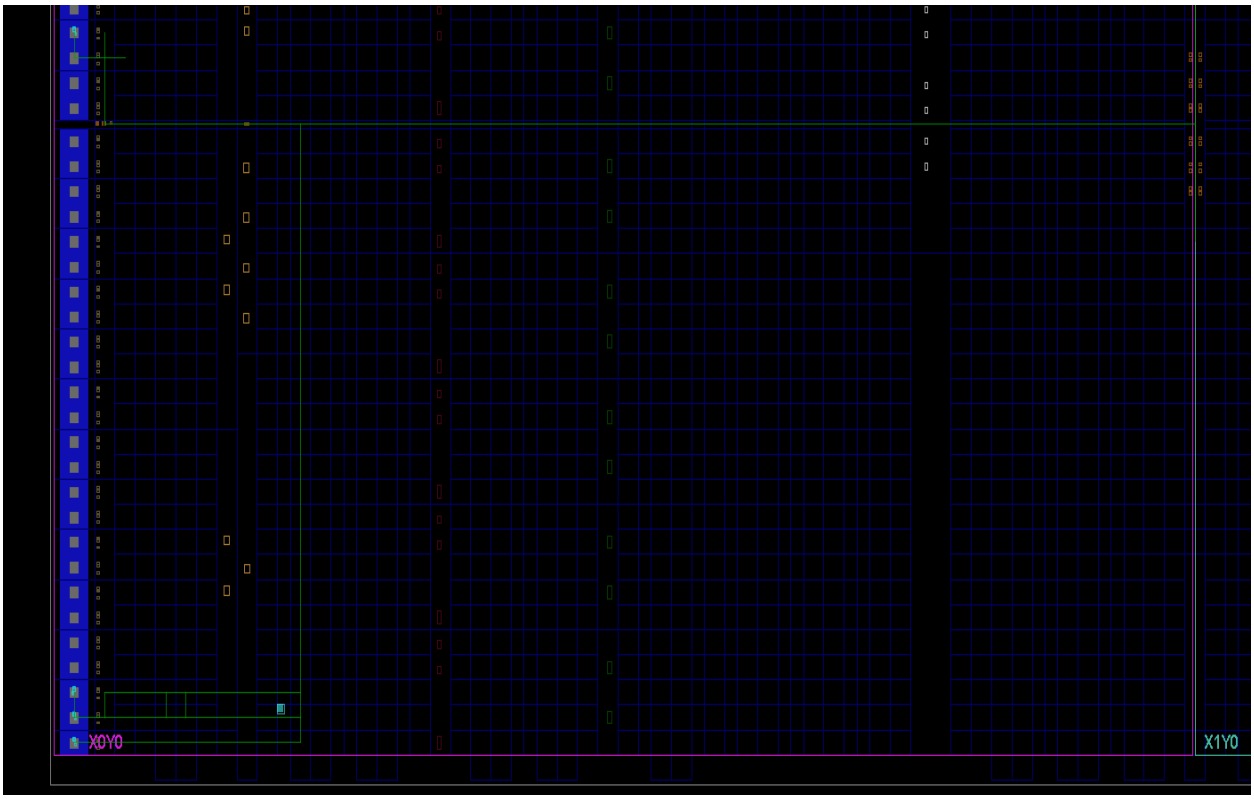
Simulation: -



RTL Analysis: -

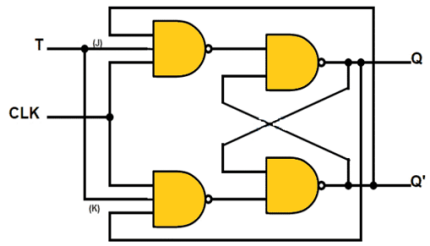


Implementation: -



T-Flip Flop: -

T Flip Flop Circuit 74HC74



T	Q	Q'
0	0	0
1	0	1
0	1	0
1	1	0

CIRCUITS DIY
SIMPLIFYING ELECTRONICS

Code: -

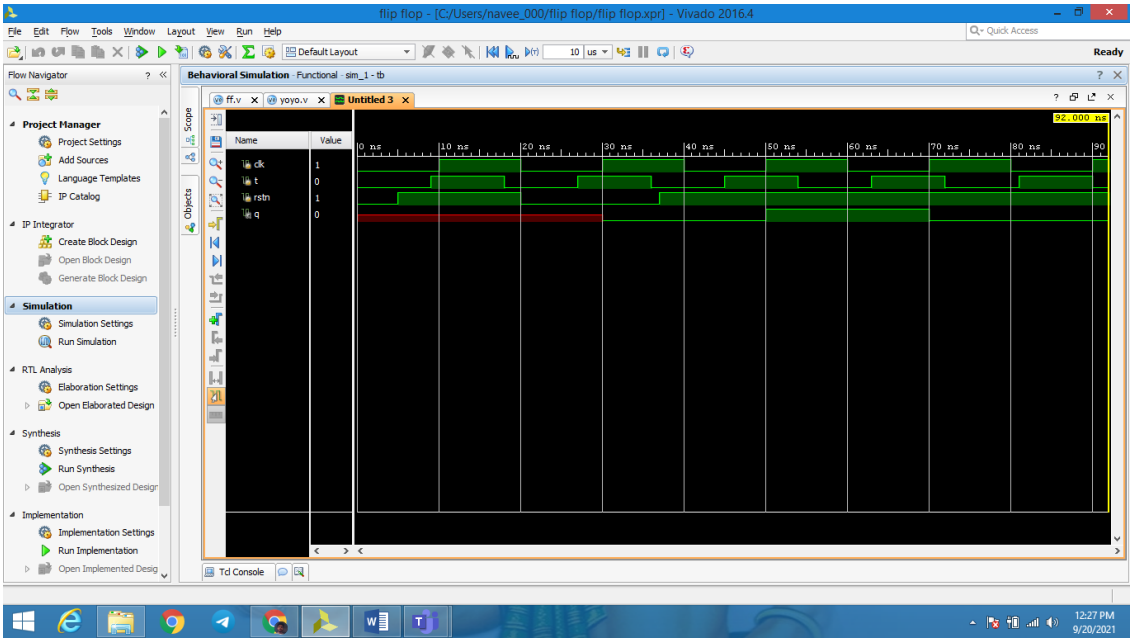
```
`timescale 1ns / 1ps
module tff(input clk,t,rstn,output reg q);
always @(posedge clk)
begin
if(!rstn)
q=0;
else if(t)
q=~q;
else
q=q;
end
endmodule
```

Testbench: -

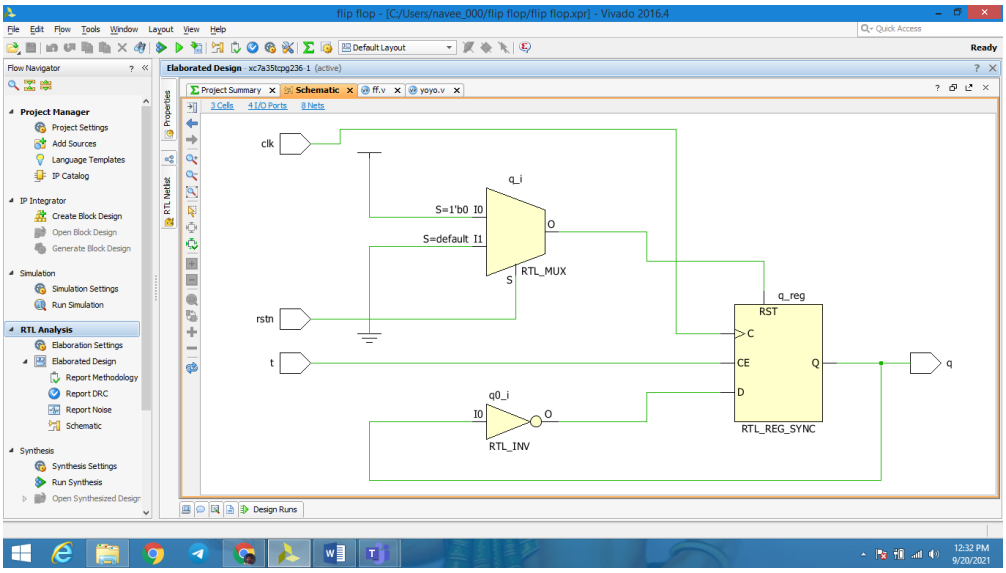
```
`timescale 1ns / 1ps
```

```
module tb;
reg clk,t,rstn;
wire q;
tff x(clk,t,rstn,q);
initial begin
rstn=0; clk=0; t=0; #5;
rstn=1; #15;
rstn=0; #17;
rstn=1; #10;
#45 $finish;
end
always #10 clk=~clk;
always #9 t=~t;
endmodule
```

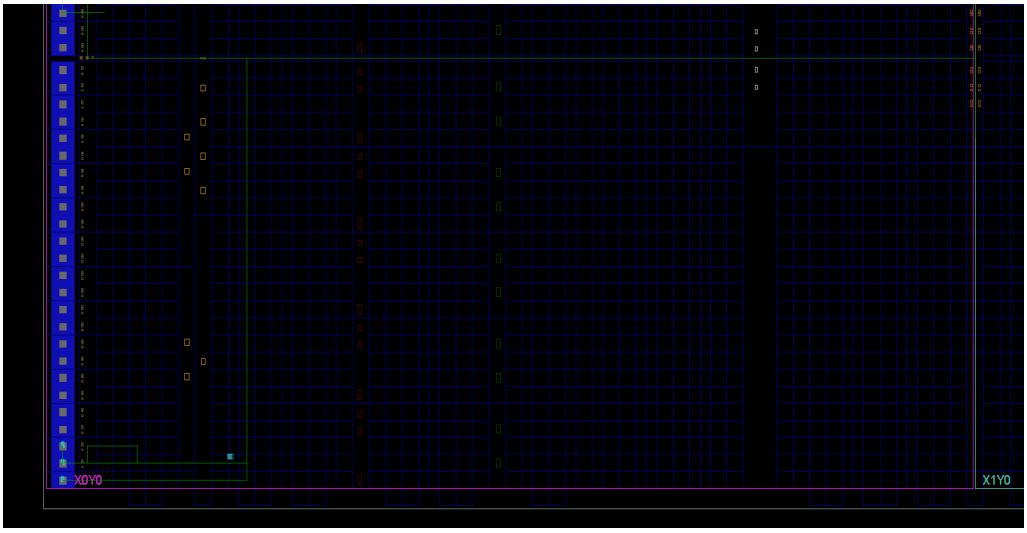
Simulation:-



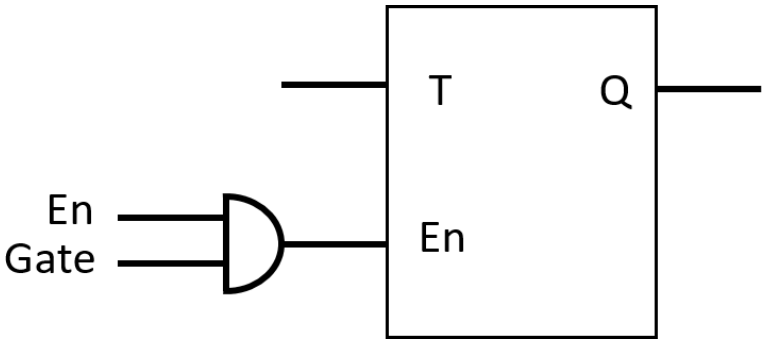
RTL Analysis: -



Implementation: -



T-Flip Flop with Gated Enable: -



Input				Output
T	En	Gate	Q	Q_nxt
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0

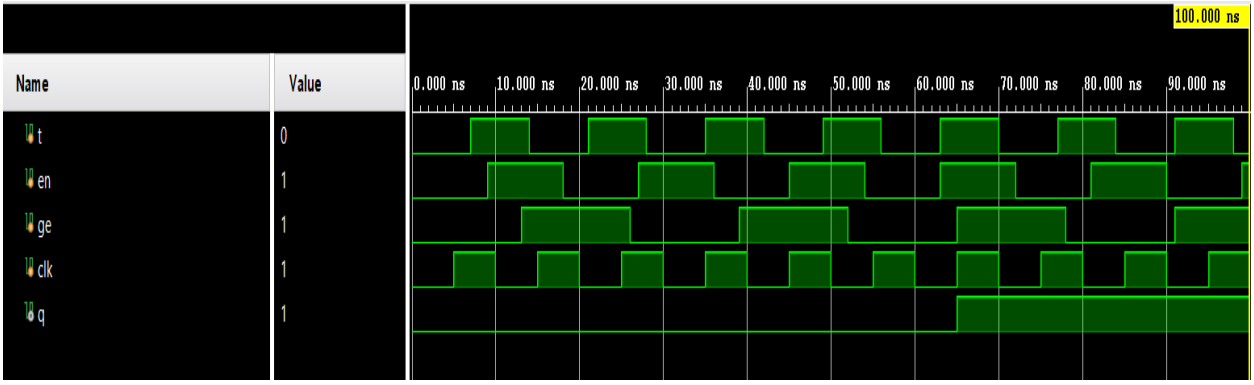
Code: -

```
module tff_gateenable(Q,T,EN,GE,clk);
input T,EN,GE,clk;
output reg Q;
initial Q=1'b0;
always @(posedge clk)
begin
if(T&EN&GE)
Q<=~Q;
else
Q<=Q;
end
endmodule
```

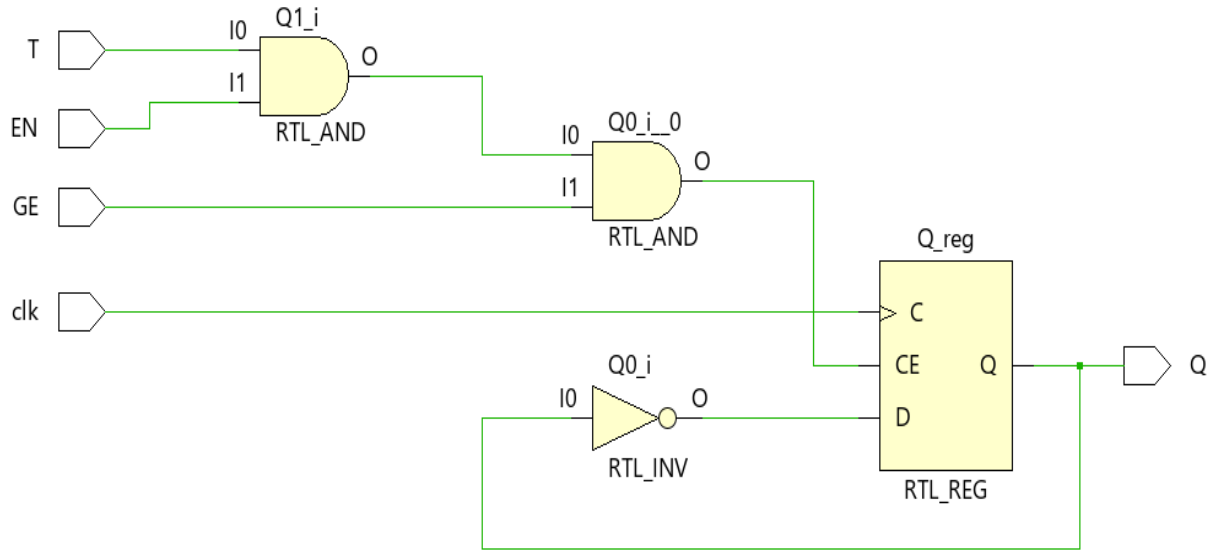
Testbench: -

```
module tff_gateenable_test();
reg t,en,ge,clk;
wire q;
tff_gateenable FF(.Q(q),.T(t),.EN(en),.GE(ge),.clk(clk) );
initial
begin
t=1'b0;
en=1'b0;
ge=1'b0;
clk=1'b0;
#100 $finish;
end
always #7 t=~t;
always #9en=~en;
always #13 ge=~ge;
always #5 clk=~clk;
endmodule
```

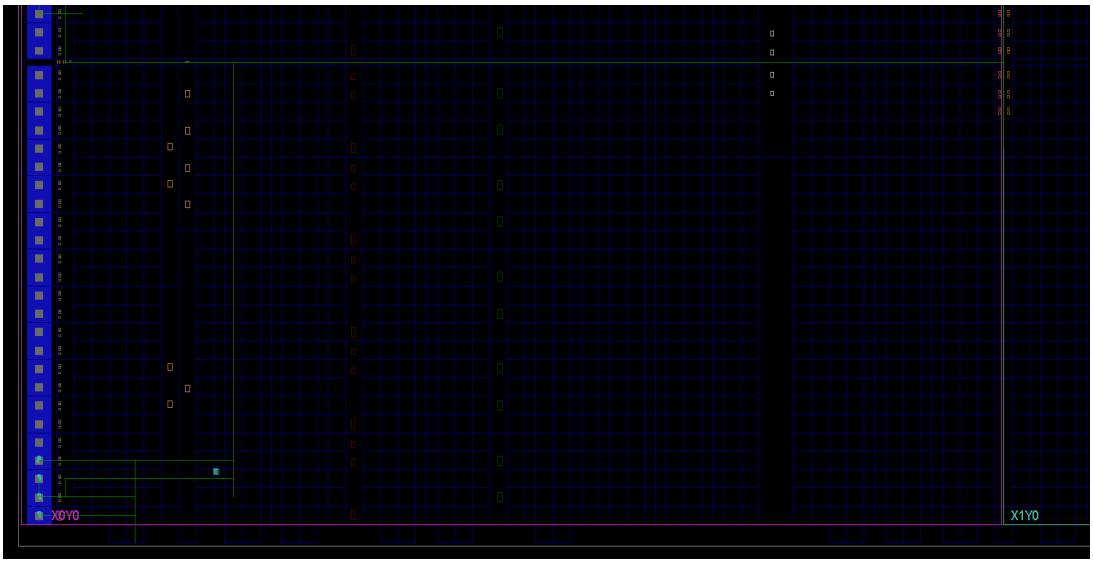
Simulation: -



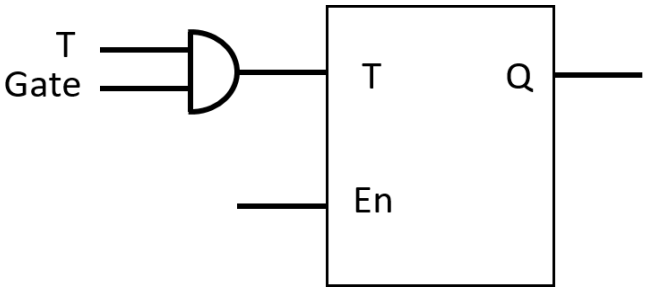
RTL Analysis: -



Implementation: -



T-Flip Flop with Gated Data: -



Input				Output
En	T	Gate	Q	Q_nxt
0	0	0	0	0
0	0	0	I	I
0	0	I	0	0
0	0	I	I	I
0	I	0	0	0
0	I	0	I	I
0	I	I	0	0
0	I	I	I	I
I	0	0	0	0
I	0	0	I	I
I	0	I	0	0
I	0	I	I	I
I	I	0	0	0

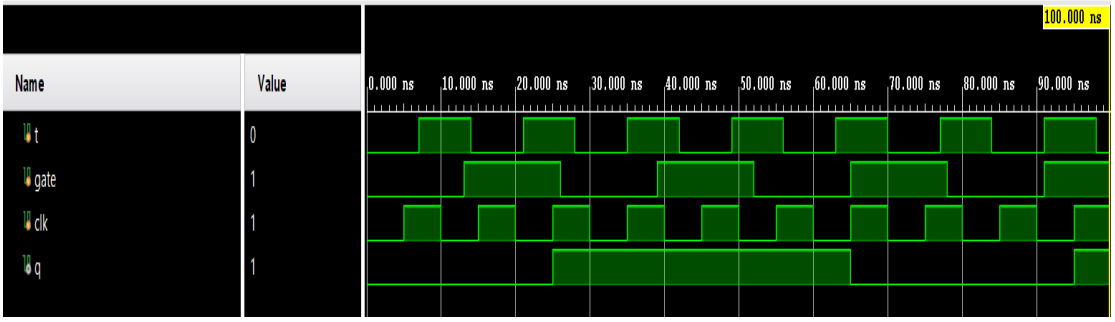
Code: -

```
module tff_gatedin(Q,T,GATE,clk);
input T,GATE,clk;
output reg Q;
initial Q=1'b0;
always @(posedge clk)
begin
if(T&GATE)
Q<=~Q;
else
Q<=Q;
end
endmodule
```

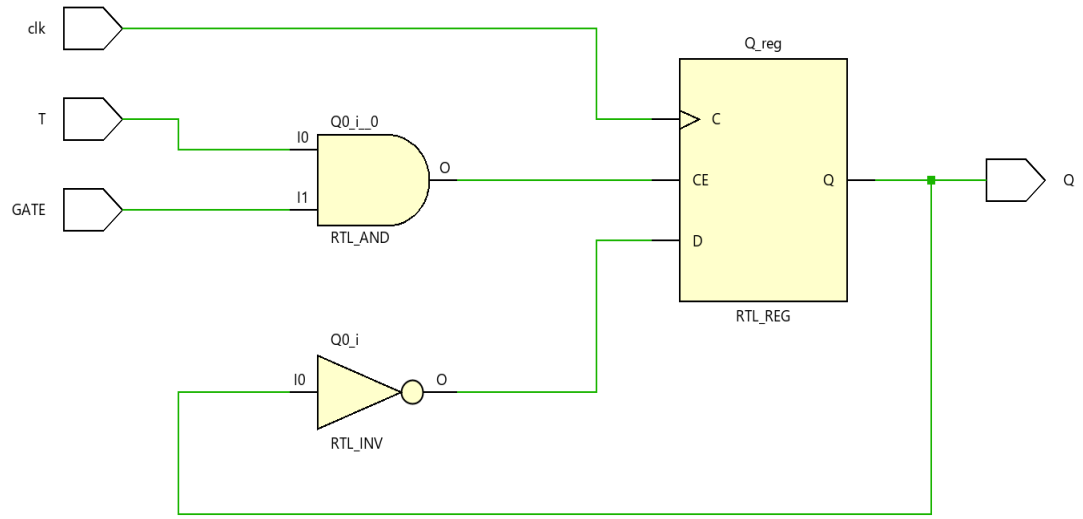
Testbench: -

```
module tff_gatedin_test();
reg t,gate,clk;
wire q;
tff_gatedin FF(.Q(q),.T(t),.GATE(gate),.clk(clk) );
initial
begin
t=1'b0;
gate=1'b0;
clk=1'b0;
#100 $finish;
end
always #7 t=~t;
always #13 gate=~gate;
always #5 clk=~clk;
endmodule
```

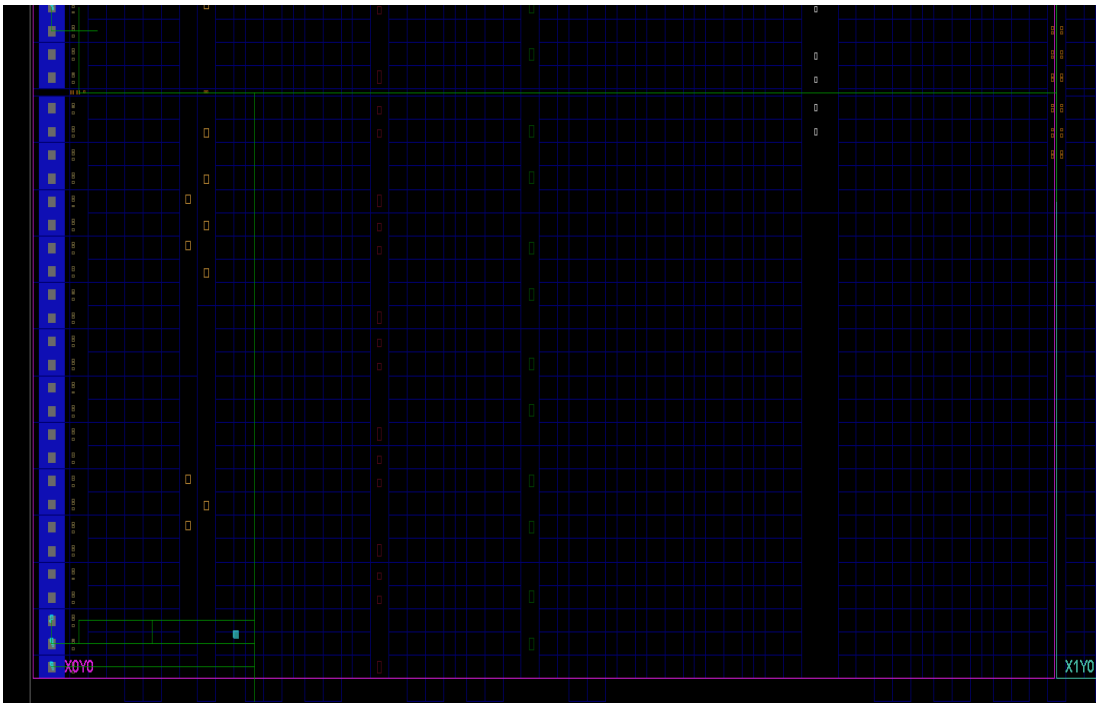
Simulation: -



RTL Analysis: -

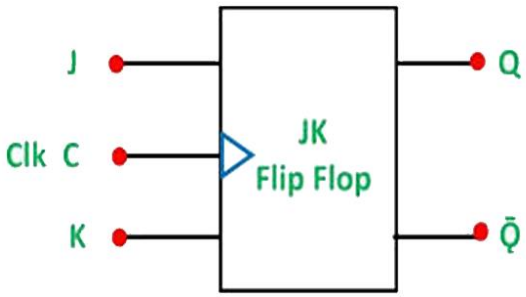


Implementation: -



JK Flip Flop: -

JK Flip Flop Circuit 74LS73



Clk	J	K	Q	Q'	State
1	0	0	Q	Q'	No change in state
1	0	1	0	1	Resets Q to 0
1	1	0	1	0	Sets Q to 1
1	1	1	-	-	Toggles

CIRCUITS DIY
SIMPLIFYING ELECTRONICS

Code:-

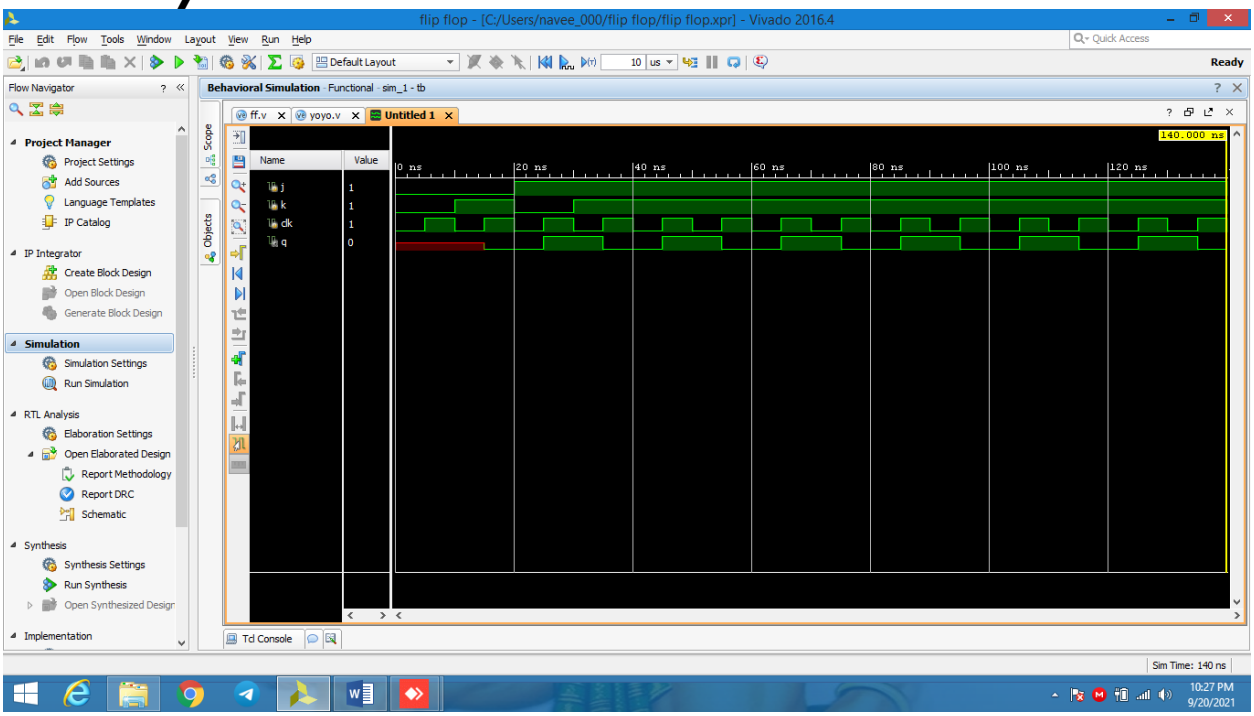
```
`timescale 1ns / 1ps
module doctjk(input j,k,clk, output reg q);
always @(posedge clk)
case ({j,k})
2'b00: q=q;
2'b01: q=0;
2'b10: q=1;
2'b11: q=~q;
endcase
endmodule
```

Testbench:-

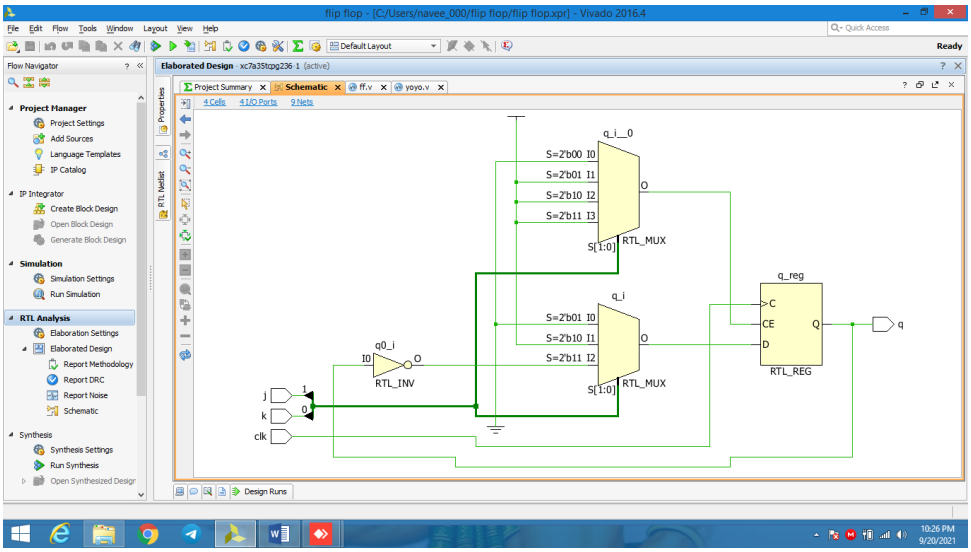
```
`timescale 1ns / 1ps
module tb;
reg j,k,clk;
```

```
    wire q;
    doctjk x(j,k,clk,q);
    initial begin
    j=0; k=0; clk=0; #10;
    j=0; k=1; #10;
    j=1; k=0; #10;
    j=1; k=1; #10;
    #100 $finish;
    end
    always #5 clk=~clk;
endmodule
```

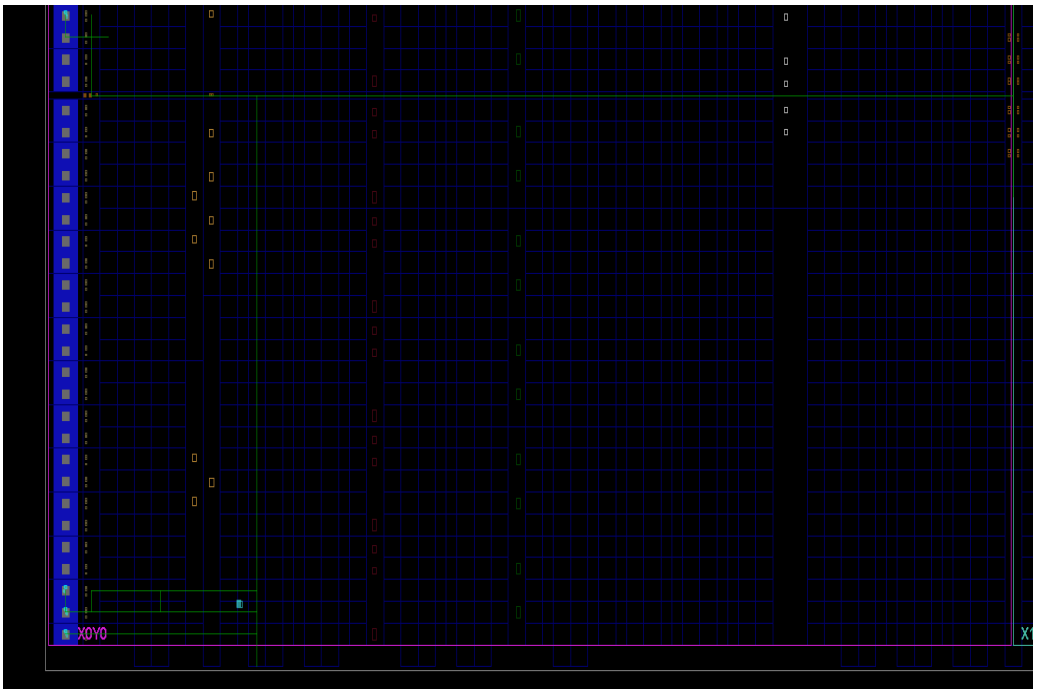
RTL Analysis:-



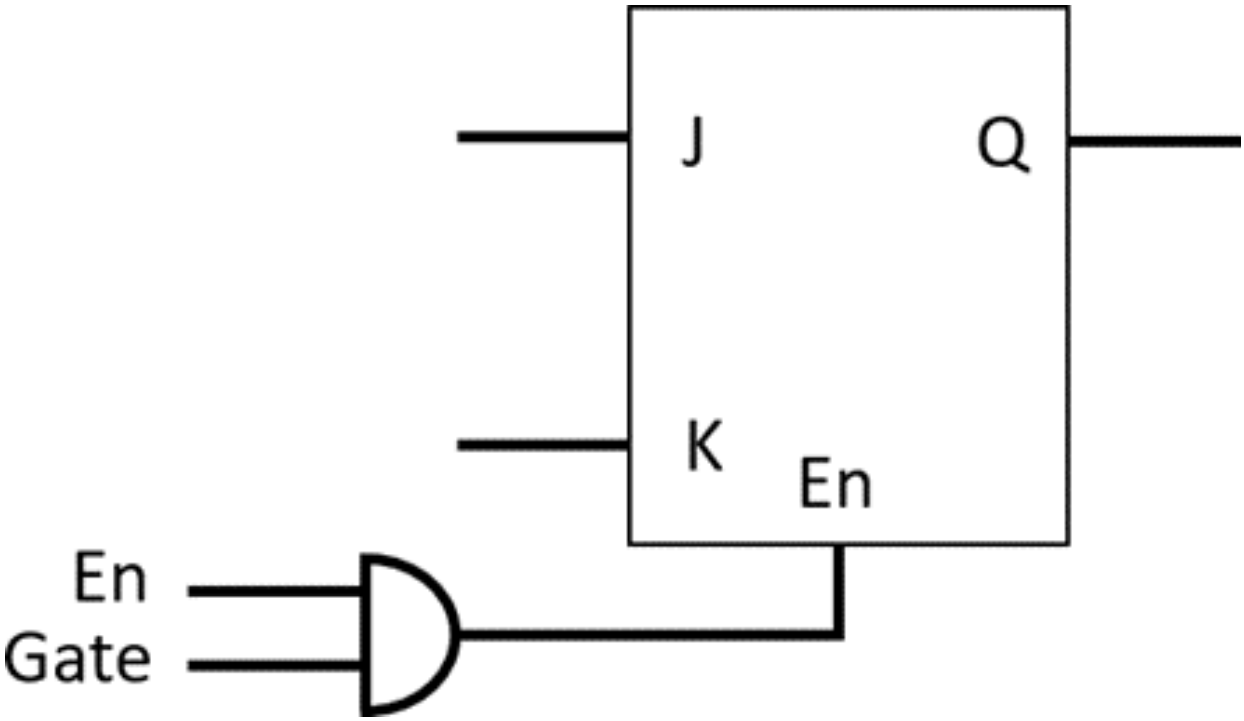
Simulation:-



Implementation: -



JK Flip Flop with gated enable: -



J	K	EN	GATE	Q	QN
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	0	1	1
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	0

Code: -

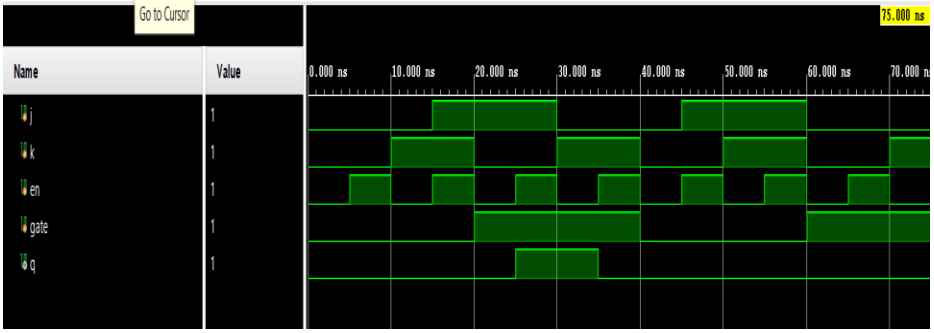
```
module jkgatedenable(Q,J,K,EN,GATE);
input J,K,EN,GATE;
output reg Q;
initial Q=1'b0;
always @(*)
begin
if(EN&GATE)
begin
if(J^K)
Q<=J;
else if(J&K)
Q<=~Q;
else
Q=Q;
end
else
Q<=Q;
end
endmodule
```

Testbench: -

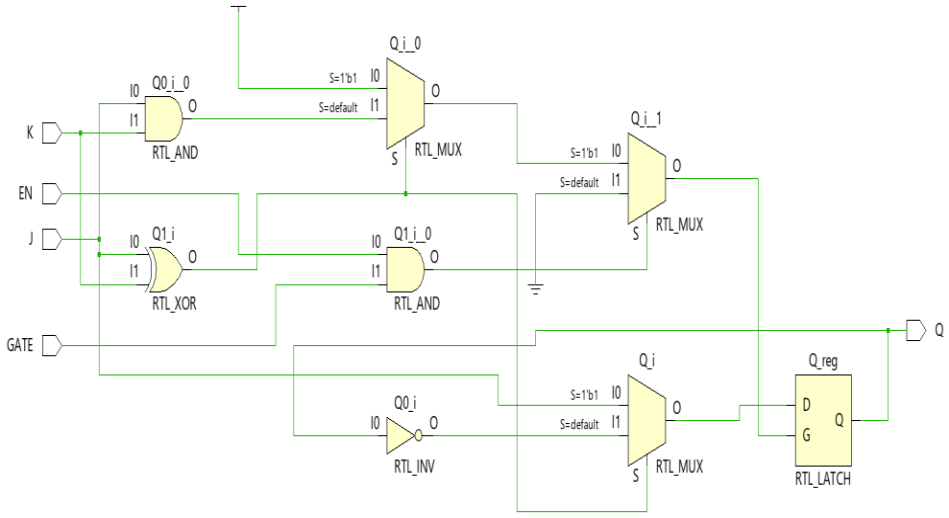
```
module jkgatedenable_test();

reg j,k,en,gate;

wire q;

jkgatedenable ff(.Q(q),.J(j),.K(k),.EN(en),.GATE(gate));

initial

begin

j=1'b0;
```

```
k=1'b0;
en=1'b0;
gate=1'b0;
#80 $finish;
end
always #5 en=~en;
always #15 j=~j;
always #10 k=~k;
always #20 gate=~gate;
endmodule
```

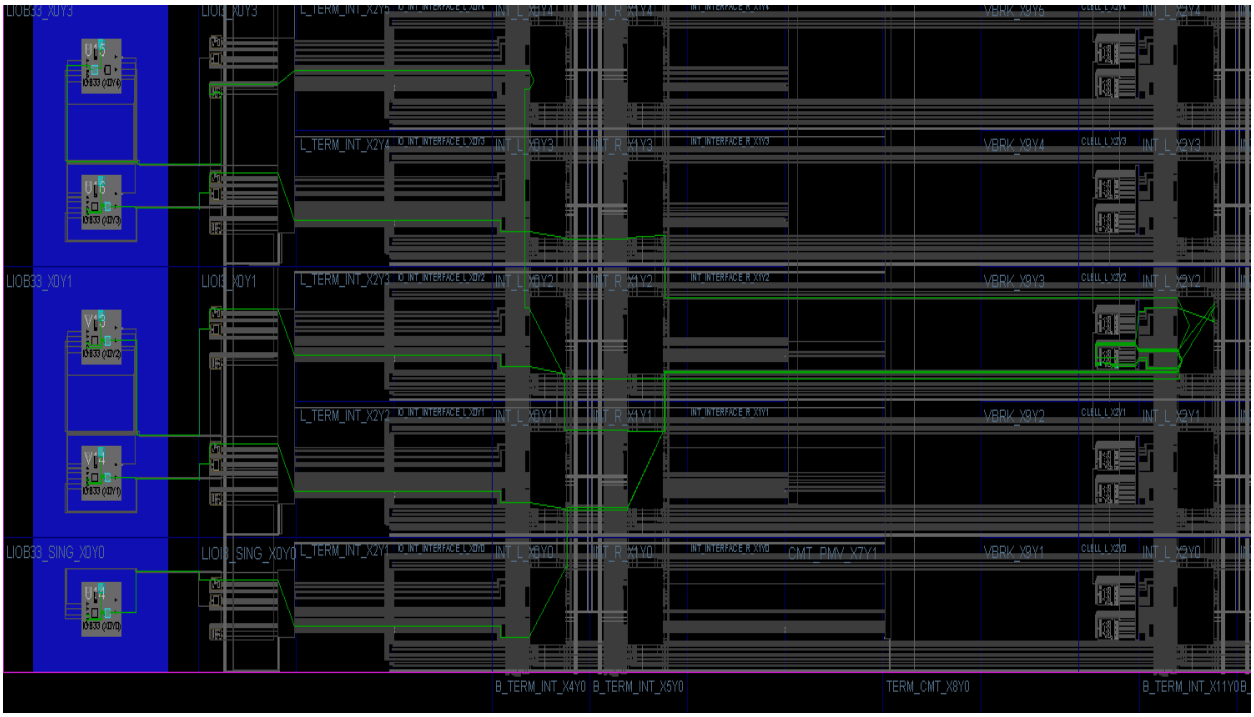
Simulation: -



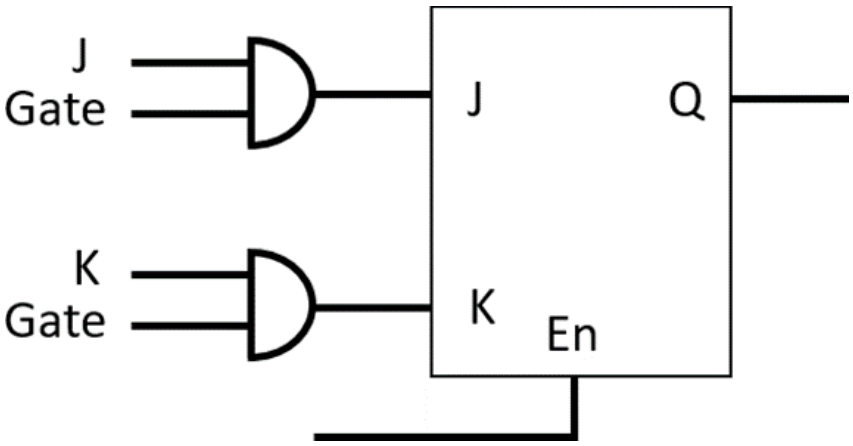
RTL Analysis: -



Implementation: -



JK Flip Flop with gated data: -



Input						Output
En	**J**	Gate J	K	Gate K	Q	Q_nxt
0	0	0	0	0	0	0
0	0	0	0	0	I	I
0	0	0	0	I	0	0
0	0	0	0	I	I	I
0	0	0	I	0	0	0
0	0	0	I	0	I	I
0	0	0	I	I	0	0
0	0	0	I	I	I	I
0	0	I	0	0	0	0
0	0	I	0	0	I	I
0	0	I	0	I	0	0
0	0	I	0	I	I	I

0	0	I	I	0	0	0
0	0	I	I	0	I	I
0	0	I	I	I	0	0
0	0	I	I	I	I	I
0	I	0	0	0	0	0
0	I	0	0	0	I	I
0	I	0	0	I	0	0
0	I	0	0	I	I	I
0	I	0	I	0	0	0
0	I	0	I	0	I	I
0	I	0	I	I	0	0
0	I	0	I	I	I	I
0	I	I	0	0	0	0
0	I	I	0	0	I	I
0	I	I	0	I	0	0
0	I	I	0	I	I	I
0	I	I	I	0	0	0
0	I	I	I	0	I	I
0	I	I	I	I	0	0
0	I	I	I	I	I	I
I	0	0	0	0	0	0
I	0	0	0	0	I	I
I	0	0	0	I	0	0
I	0	0	0	I	I	I
I	0	0	I	0	0	0

I	0	0	I	0	I	I
I	0	0	I	I	0	0
I	0	0	I	I	I	0
I	0	I	0	0	0	0
I	0	I	0	0	I	I
I	0	I	0	I	0	0
I	0	I	0	I	I	I
I	0	I	I	0	0	0
I	0	I	I	0	I	I
I	0	I	I	I	0	0
I	0	I	I	I	I	0
I	I	0	0	0	0	0
I	I	0	0	0	I	I
I	I	0	0	I	0	0
I	I	0	0	I	I	I
I	I	0	I	0	0	0
I	I	0	I	0	I	I
I	I	0	I	I	0	0
I	I	0	I	I	I	0
I	I	I	0	0	0	I
I	I	I	0	0	I	I
I	I	I	0	I	0	I
I	I	I	0	I	I	I
I	I	I	I	0	0	I
I	I	I	I	0	I	I

I	I	I	I	I	0	I
I	I	I	I	I	I	0

Code: -

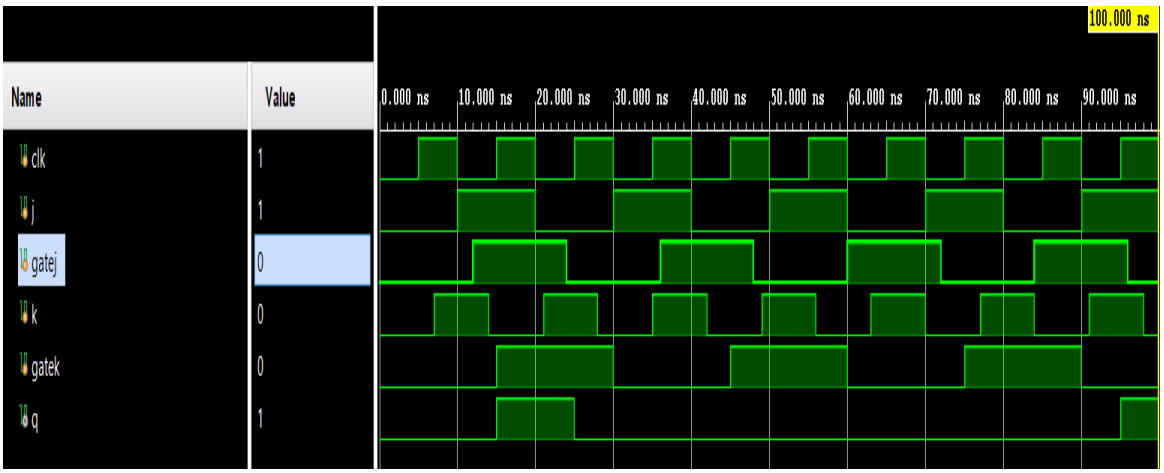
```
module JKgatedin(J,K,GATEJ,GATEK,CLK,Q);
input J,K,GATEJ,GATEK,CLK;
output reg Q;
wire JJ,KK;
initial Q=1'b0;
assign JJ=GATEJ&J;
assign KK=GATEK&K;
always @(posedge CLK)
begin
if(JJ^KK)
Q<=JJ;
else if(JJ&KK)
Q<=~Q;
else
Q<=Q;
end
endmodule
```

Testbench: -

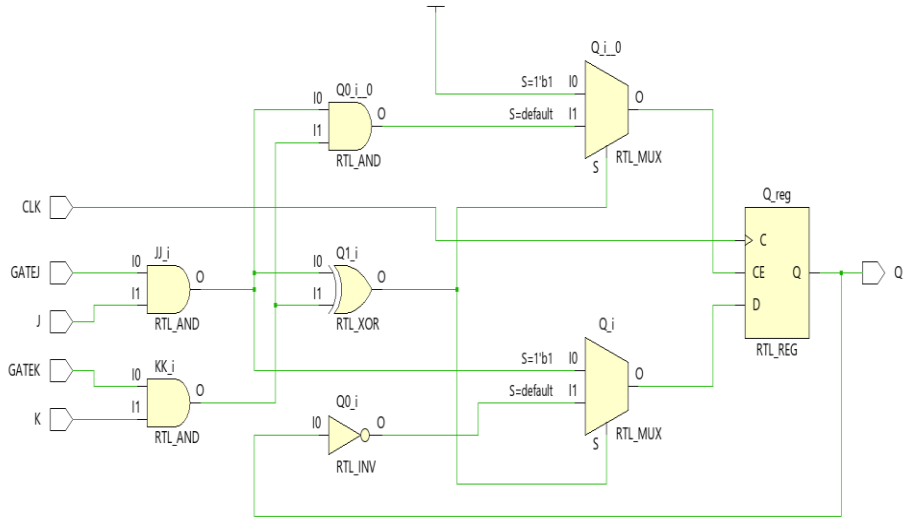
```
module JKgatedin_test();

reg j,k,gatej,gatek,clk;

wire q;

JKgatedin ff(.J(j),.K(k),.GATEJ(gatej),.GATEK(gatek),.CLK(clk),.Q(q));

Initial

Begin

j=1'b0;
```

```
k=1'b0;
clk=1'b0;
gatej=1'b0;
gatek=1'b0;
#100 $finish;
end
always #5 clk=~clk;
always #10 j=~j;
always #7 k=~k;
always #12 gatej=~gatej;
always #15 gatek=~gatek;
endmodule
```

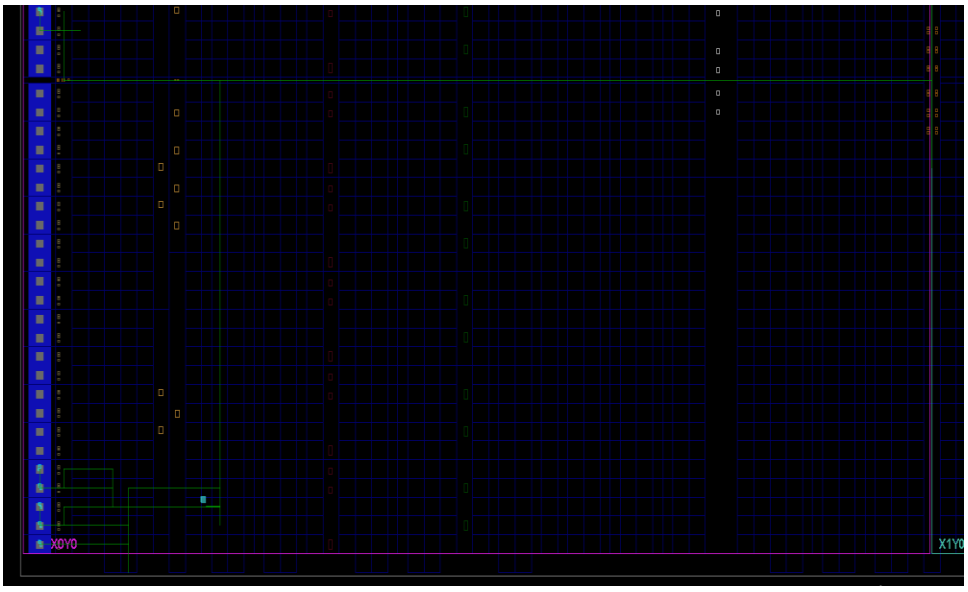
Simulation: -



RTL Analysis: -



Implementation: -



SR Flip Flop:-

Clock	S	R	Q
0	×	×	Q_n
1	0	0	$Q_n \rightarrow$ Hold
1	0	1	$0 \rightarrow$ Hold
1	1	0	$1 \rightarrow$ Hold
1	1	1	Invalid

Code:-

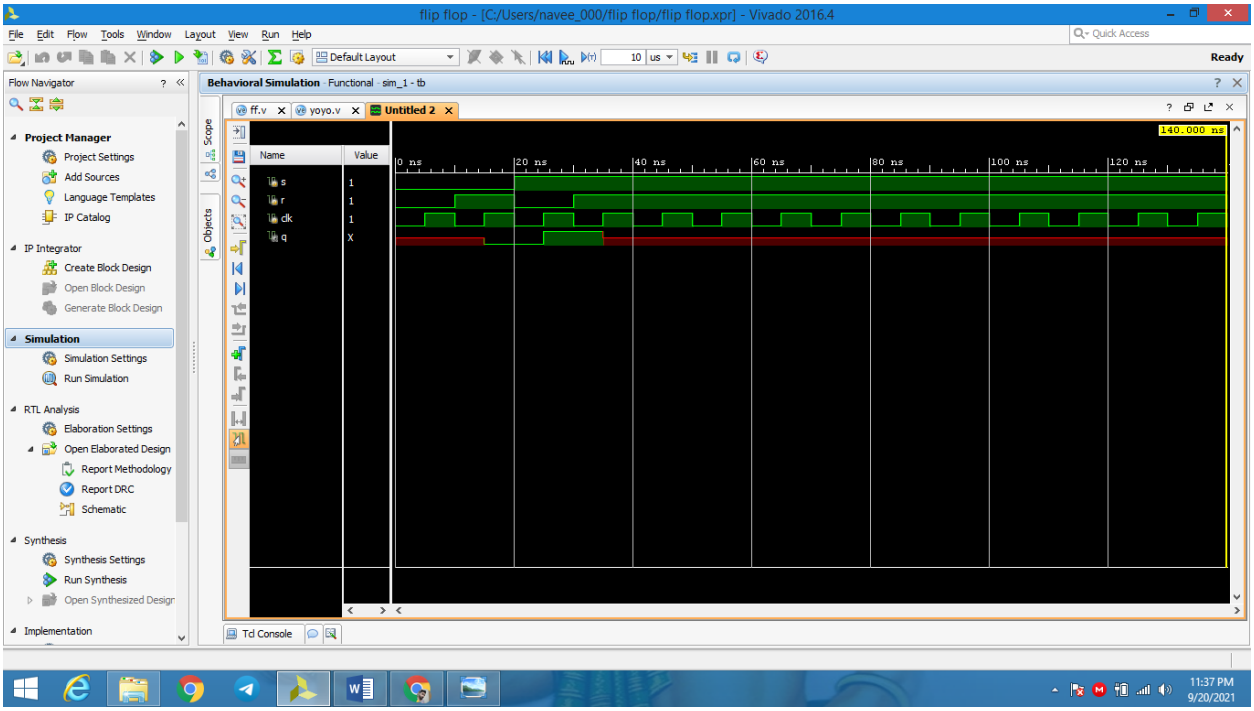
```
`timescale 1ns / 1ps
module docsr(input s,r,clk, output reg q);
always @(posedge clk)
case ({s,r})
2'b00: q=q;
2'b01: q=0;
2'b10: q=1;
2'b11: q=1'bx;
endcase
endmodule
```

Testbench:-

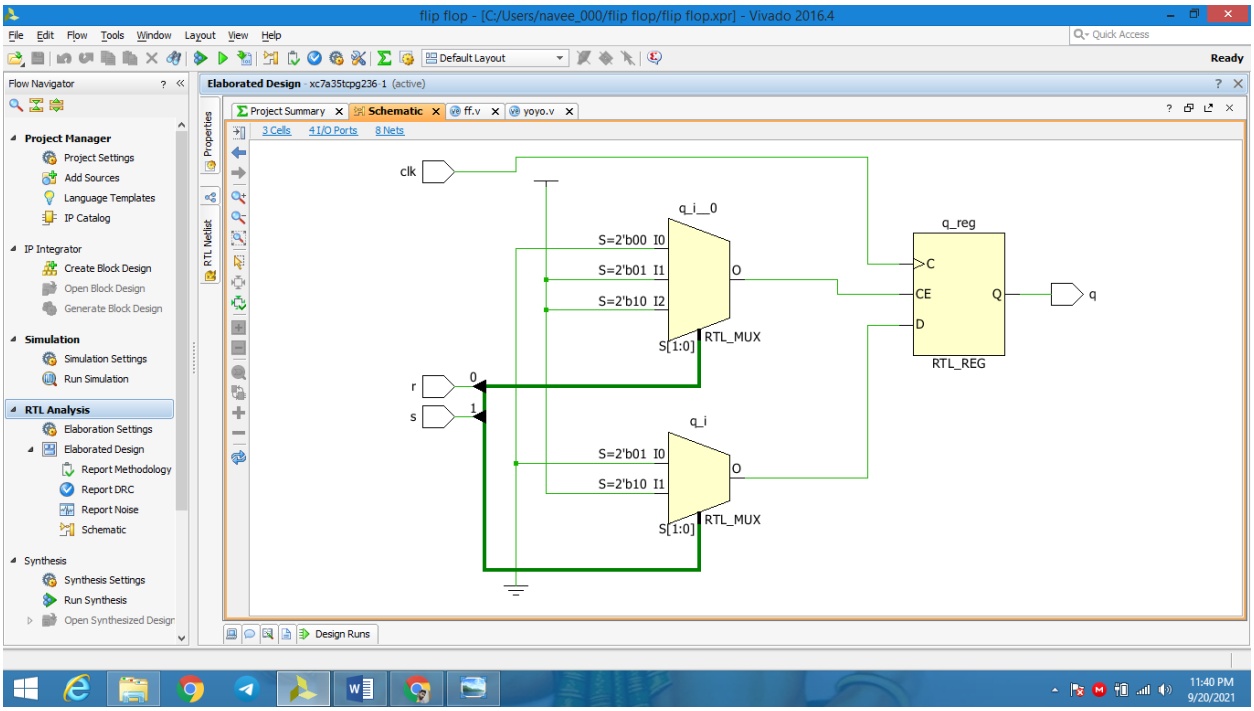
```
`timescale 1ns / 1ps
module tb;
reg s,r,clk;
wire q;
docsr x(s,r,clk,q);
initial begin
s=0; r=0; clk=0; #10;
s=0; r=1; #10;
```

```
    s=1; r=0; #10;
    s=1; r=1; #10;
    #100 $finish;
    end
    always #5 clk=~clk;
endmodule
```

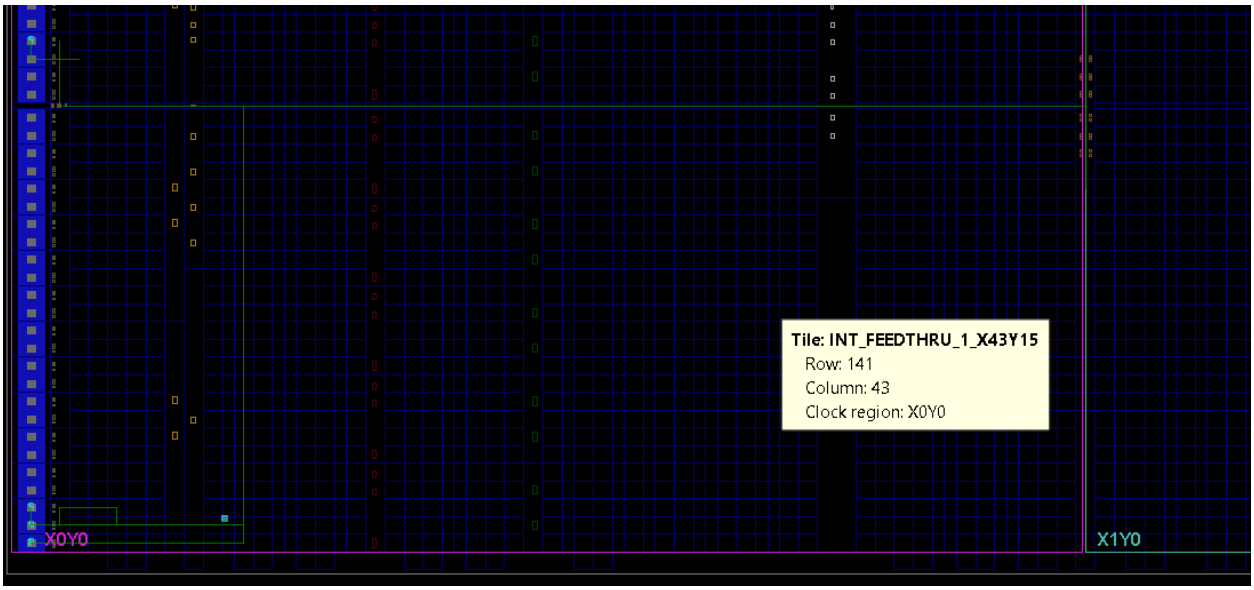
Simulation:-



RTL Analysis:-



Implementation



Conclusion:- Hence we verified JK,SR,D and T flipflop using Xilinx Vivado.

Learning Outcome: **In this experiment I have learned how to design different types of Flip Flops**.

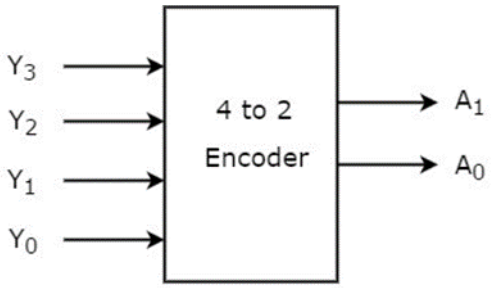
NATIONAL INSTITUTE OF TECHNOLOGY (NIT) DELHI

Experiment - 6

Encoder and Decoder

Encoder: -

An Encoder is a combinational circuit that performs the reverse operation of Decoder. It has a maximum of 2^n input lines and 'n' output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with 'n' bits. It is optional to represent the enable signal in encoders.



Input				Output	
Y3	Y2	Y1	Y0	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

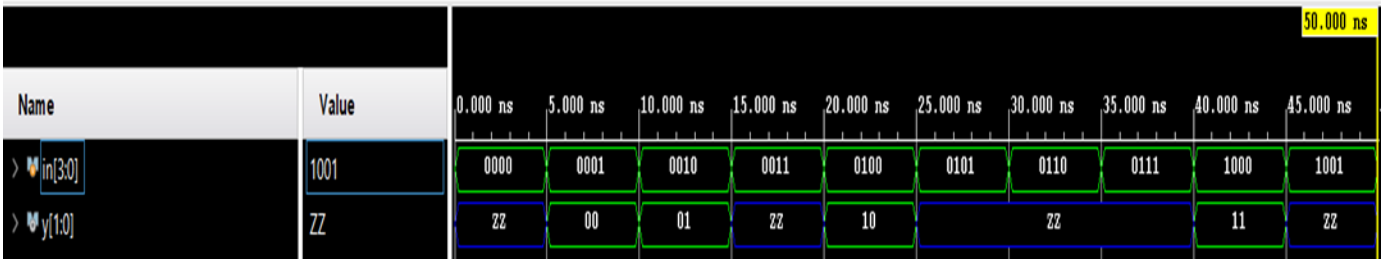
Code: -

```
module encoder(IN,Y);
    input [3:0]IN;
    output reg [1:0]Y;
    always @(*)
    begin
    case(IN)
    4'b0001 : Y= 2'b00;
    4'b0010 : Y= 2'b01;
    4'b0100 : Y= 2'b10;
    4'b1000 : Y= 2'b11;
    default : Y= 2'bzz;
    endcase
    end
    endmodule
```

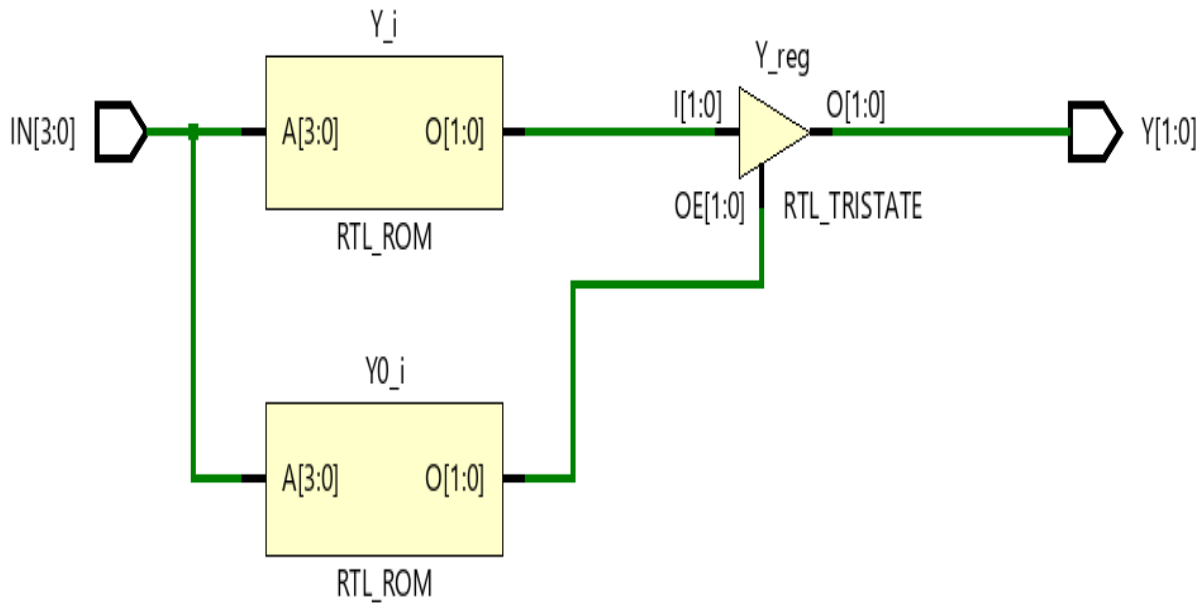
Testbench: -

```
    module encoder_test();
    reg [3:0]in;
    wire [1:0]y;
    encoder E(.IN(in),.Y(y));
    initial
    begin
    in=4'b0000;
    #50 $finish;
    end
    always #5 in=in+4'b0001;
    endmodule
```

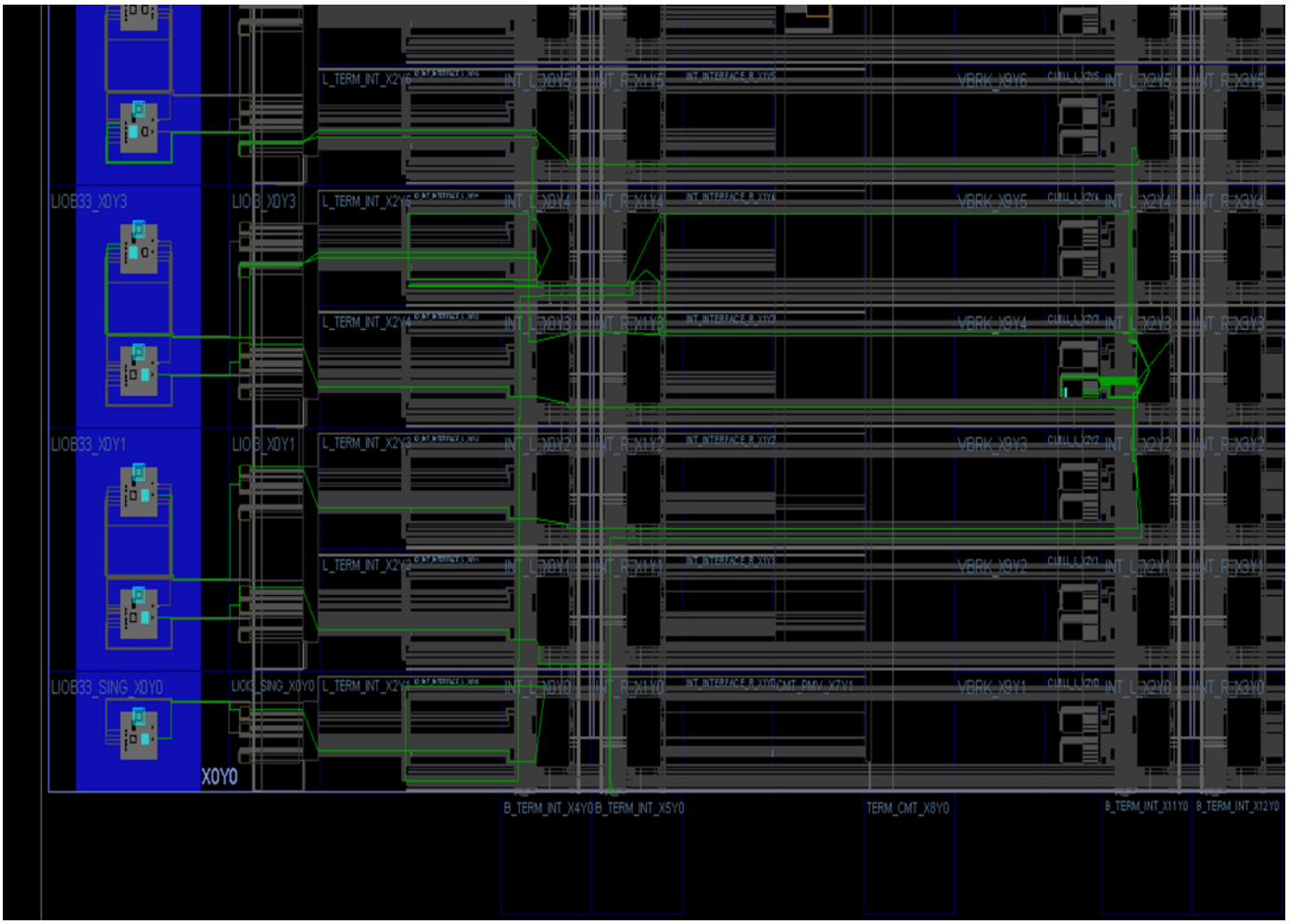
Simulation: -



RTL Analysis: -



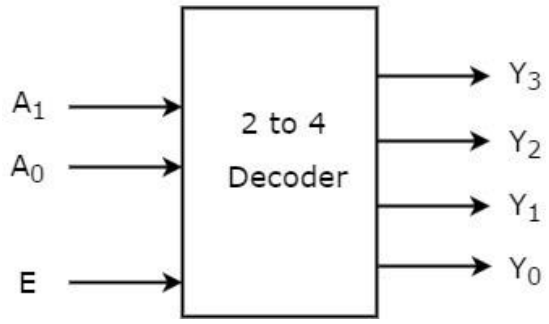
Implementation: -



Decoder: -

Decoder is a combinational circuit that has 'n' input lines and maximum of 2n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means the decoder detects a particular code. The outputs of the decoder are nothing but the minterms of 'n' input variables lines, when it is enabled.

Block Diagram:



Truth Table:

Input		Output			
A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

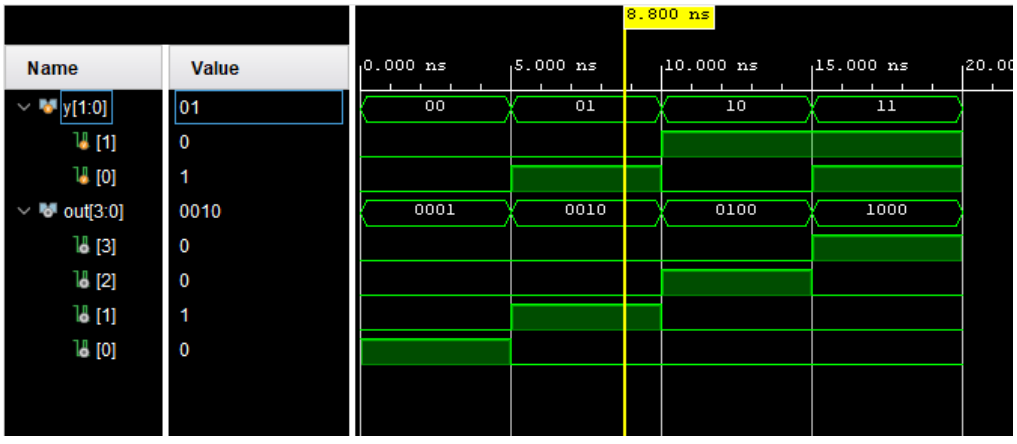
Code: -

```
module decoder(in, out); input [1:0]in; output reg [3:0]out;
always@(in) begin if (in == 2'b00)
out = 4'b0001; else if(in == 2'b01)
out = 4'b0010; else if(in == 2'b10)
out = 4'b0100; else if(in == 2'b11)
out = 4'b1000;
end
endmodule
```

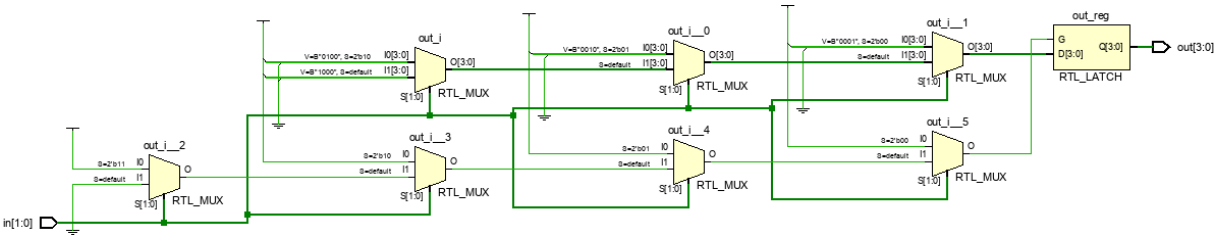
Testbench: -

```
module decoder_testbench;
reg [1:0]y;
wire [3:0]out;
decoder d1(.in(y), .out(out)); initial
begin
y = 2'b00;
#20 $finish; end
always #5 y = y + 1'b1;
endmodule
```

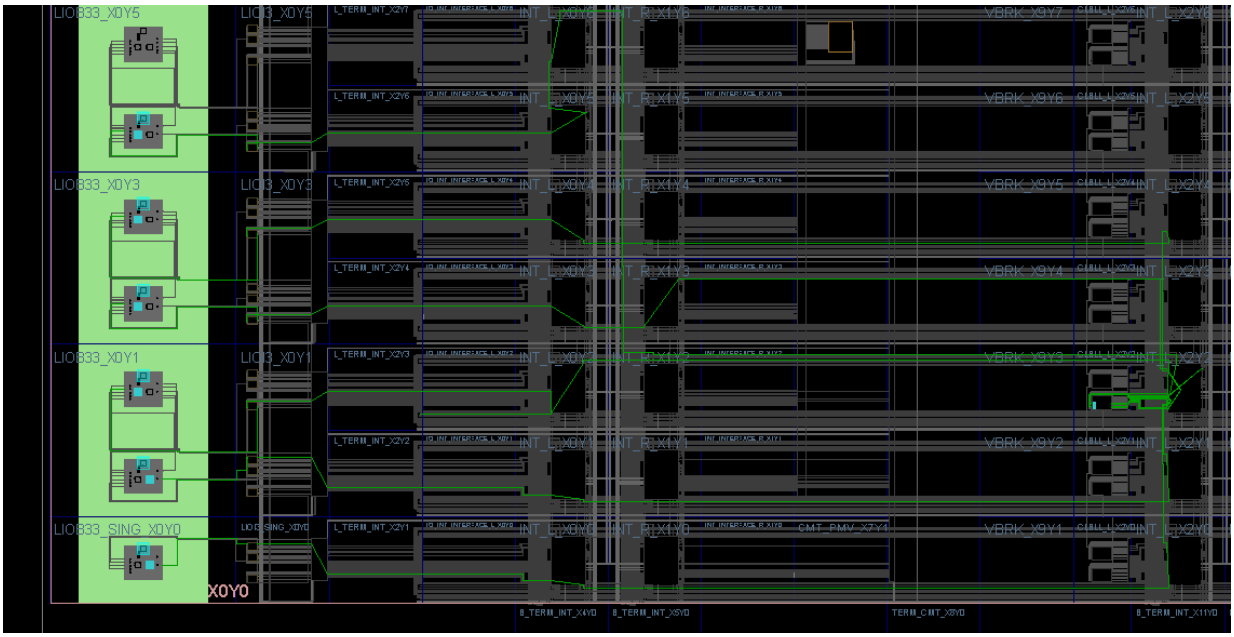
Simulation: -



RTL Analysis: -



Implementation: -



Learning Outcome: In this experiment I have learned how to design different types of Encoder/Decoder.

Experiment - 7

Counter FSM

Introduction: Counter is a sequential circuit. A digital circuit which is used for counting pulses is known as a counter. Counter is the widest application of flip-flops. It is a group of flip-flops with a clock signal applied. Counters are of two types.

- Asynchronous or ripple counters.
- Synchronous counters.

Requirements: Vivado

Objective: Using Up and Down counter or Bi-directional counter, start with value 32 and down count with 2 till we reach 0, then count up with 3 till we reach 27, then count down with 2 till we reach 1

Code:-

```
`timescale 1ns / 1ps
module counter(clk,count,reset);
input reset,clk;
output reg [5:0] count;
 reg [5:0]a= -6'b000010;
reg var = 0;
always @ (posedge clk or posedge reset)
 begin
if (reset)
count <= 6'b100000;
else
```

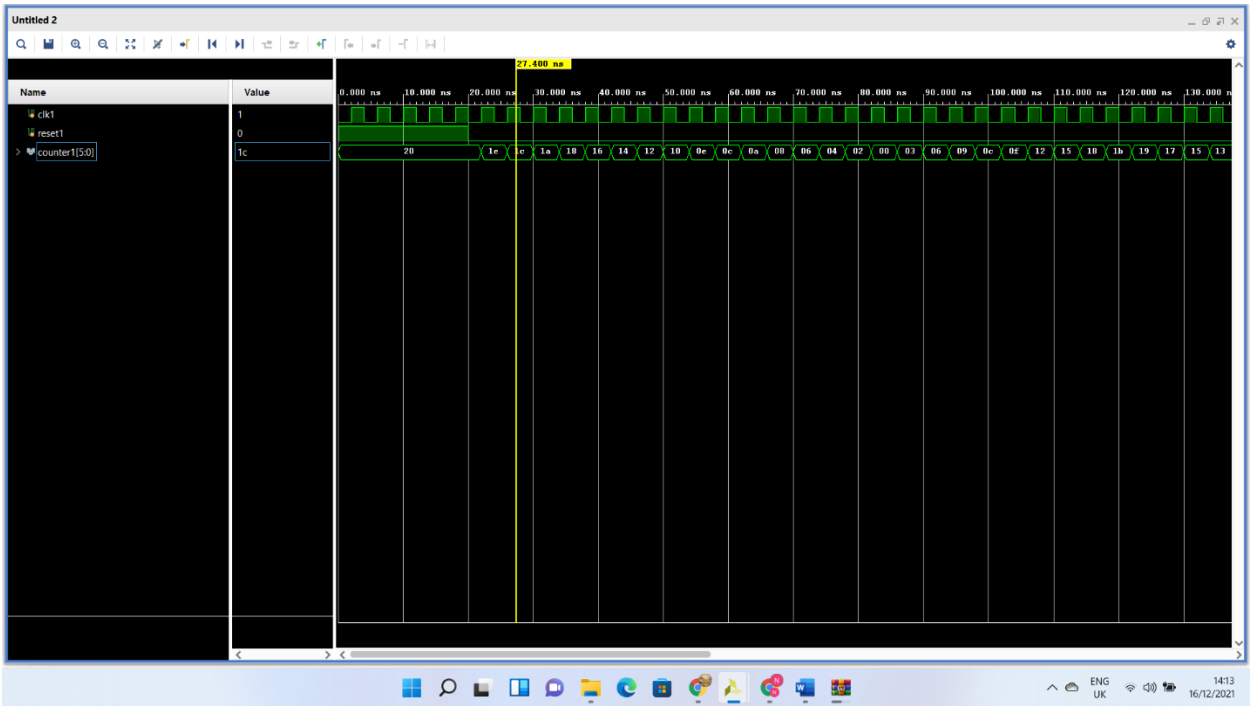
```
    begin
    if(count == 6'b000000)
    a = 6'b000011;

    else if (count == 6'b011011)
     a = -6'b000010;
    else if (count==6'b000111 & var == 1)
    begin

    a = 6'b000000;



    end
    else if (count==6'b000111 & var == 0)
    begin

    var = 1;



    end
    count <= count + a;



    end
    end
endmodule
```

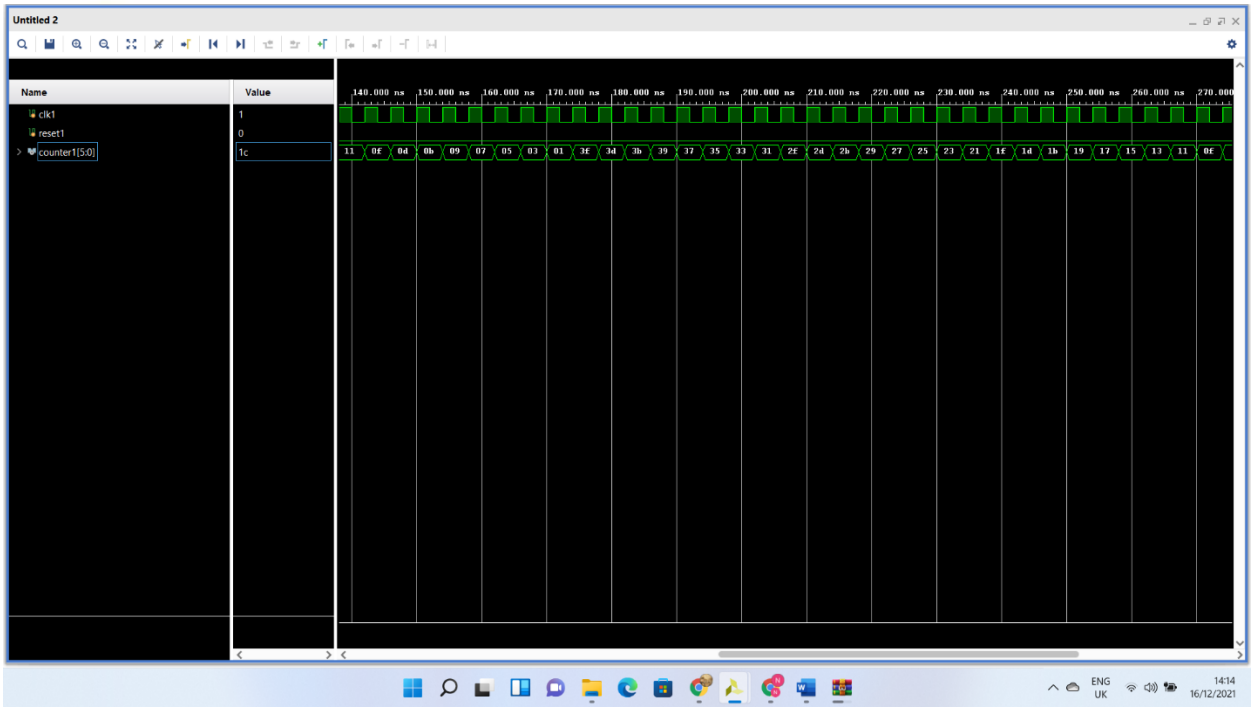
Testbench: -

```
module testbench;
 reg clk1,reset1;
 wire [5:0] counter1;
counter G ( .clk(clk1), .reset(reset1), .count(counter1));
initial
 begin
clk1 = 0;
reset1 = 1;
#20 reset1 = 0;
#300 $finish;
 end
```

```
always clk1 = #2 ~clk1;
 endmodule
```

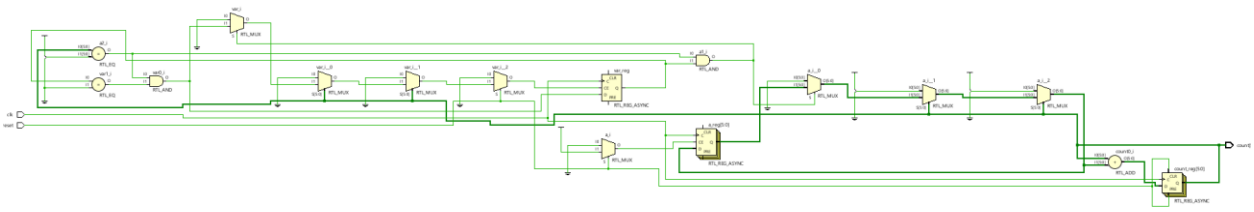
Simulation: -



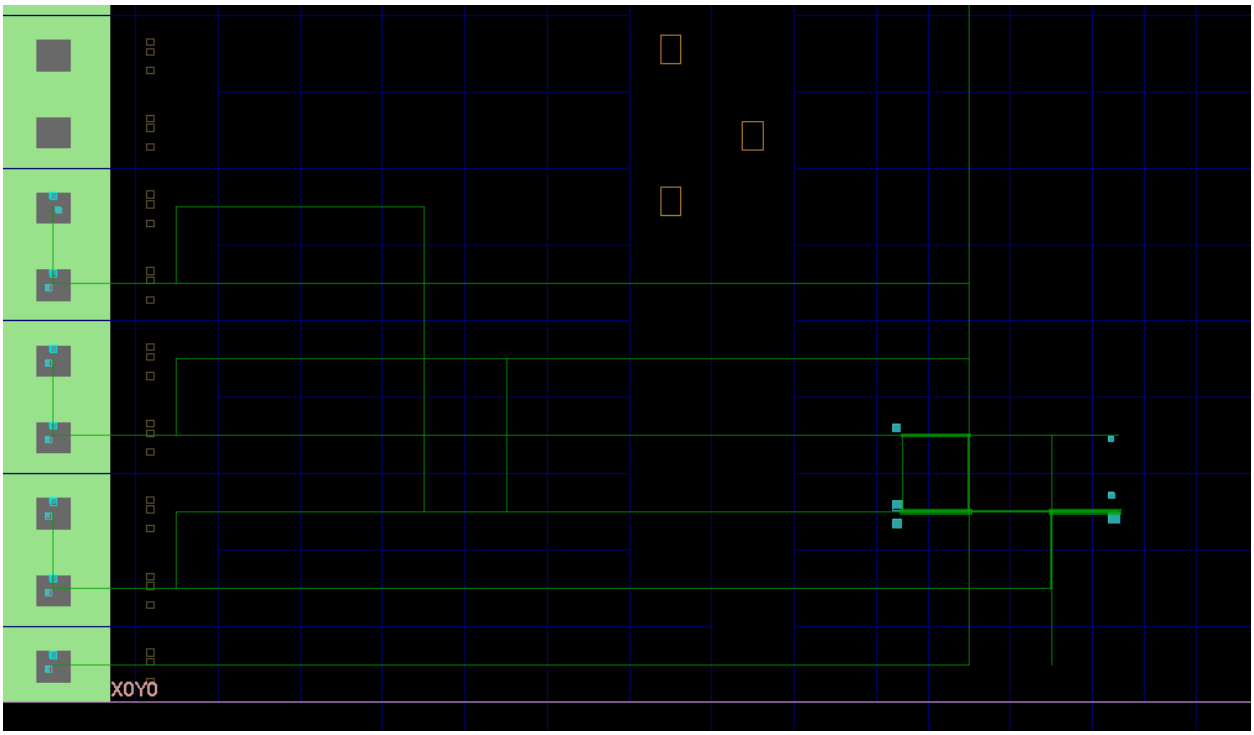

Untitled 2
Name
Value
clk1
1
reset1
0
counter1[5:0]
1c
140.000 ns
150.000 ns
160.000 ns
170.000 ns
180.000 ns
190.000 ns
200.000 ns
210.000 ns
220.000 ns
230.000 ns
240.000 ns
250.000 ns
260.000 ns
11 0E 0d 0b 09 07 05 03 01 3E 3d 3b 39 37 35 33 31 2E 2d 2b 29 27 25 23 21 1E 1d 1b 19 17 15 13 11 0E
ENG
UK
14:14
16/12/2021

*since I don't know how to adjust clock accordingly so I have attached three screenshots.

RTL Analysis: -



Implementation: -



Learning Outcome: In this experiment I have learned how to design FSM.