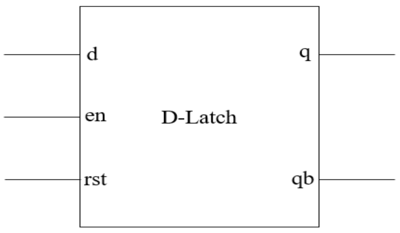
1. Write a Verilog code for D-Latch and verify its functionality using test bench. Synthesize the design, tabulate the area, power and timing report.

Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

D-Latch: *Latch is an electronic device that can be used to store one bit of information. The D latch is used to capture, or 'latch' the logic level which is present on the Data line when the enable input is high. If the data on the D line changes state while the enable is high, then the output, Q, follows the input, D. When the enable input falls to logic 0, the last state of the D input is trapped and held in the latch.*

D-Latch block diagram:



D-Latch truth table:

rst	**en**	**d**	**q**	**qb**
1	X	X	0	1
0	0	X	q	qb
0	1	0	0	1
0	1	1	1	0

Verilog code for D-Latch:

```
module dlatch (q, qb, d, en, rst);
output reg q;
output qb;
input d, en, rst;

always @(d, en, rst)

begin
if (rst)
q = 0;

else
```

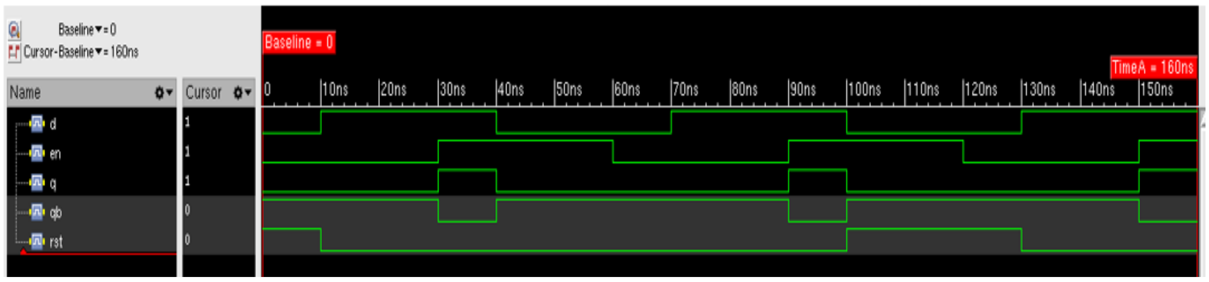
```
if (en)
q = d;
end

assign qb = ~q;

endmodule
```

Testbench for D-Latch module:

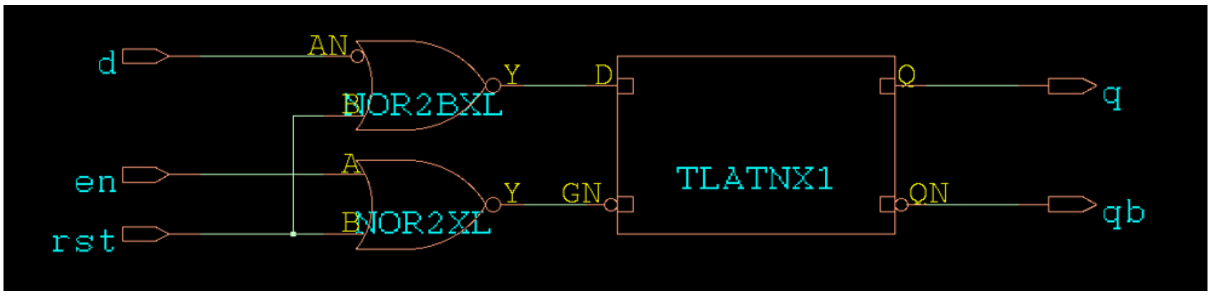
```
module dlatch_test;
reg d, en, rst;
wire q, qb;

dlatch d1 (q, qb, d, en, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “d =”, d, “en =”, en, “rst =”, rst, “q =”, q, “qb =”, qb);
#160 $finish;
end

initial
begin
en = 0;
d = 0;
end

always
begin
#10 d = ~d;
#20 en = ~en;
end

initial
begin
rst = 1;
#10 rst = 0;
#90 rst = 1;
#30 rst = 0;
end

endmodule
```

Result:

Simulation:



Schematic:



Area report:

```
 Instance Module  Cell Count  Cell Area  Net Area   Total Area  Wireload
-------------------------------------------------------------------------
 dl                        3     59.875     0.000       59.875    <none> (D)
```

Power report:

```
                   Leakage   Dynamic    Total
Instance Cells Power(nW) Power(nW) Power(nW)
--------------------------------------------
dl           3     1.930  1947.052  1948.982
```

Timing report:

```
Path 1: UNCONSTRAINED
     Startpoint: (F) q_reg/GN
       Endpoint: (R) qb


      Data Path:-      458

#--------------------------------------------------------------------------------------------
# Timing Point   Flags    Arc    Edge   Cell      Fanout Load Trans Delay Arrival Instance
#                                                        (fF)  (ps)  (ps)   (ps)  Location
#--------------------------------------------------------------------------------------------
  q_reg/GN        -        -      F     (arrival)      1    -     0     -      0    (-,-)
  q_reg/QN        -      GN->QN R     TLATNX1        1  0.0    60   458    458    (-,-)
  qb              -        -      R     (port)         -    -     -     0    458    (-,-)
#--------------------------------------------------------------------------------------------
```

2. **Write a Verilog code for D Flip-Flop with synchronous and asynchronous reset and verify its functionality using test bench. Synthesize the design, tabulate the area, power and timing report.**

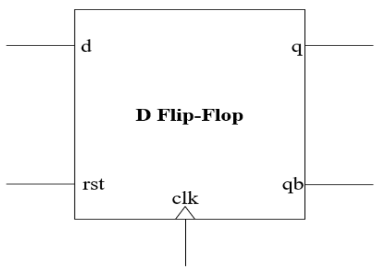
Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

D Flip-Flop: *A D (or Delay) Flip Flop is a digital electronic circuit used to delay the change of state of its output signal (Q) until the next rising edge of a clock timing input signal occurs. The D Flip Flop acts as an electronic memory component since the output remains constant unless deliberately changed by altering the state of the D input followed by a rising clock signal.*

a) D Flip-Flop with synchronous reset:

D Flip-Flop with synchronous reset block diagram:



D Flip-Flop with synchronous reset truth table:

rst	**clk**	**d**	**q**	**qb**
1	↑	X	0	1
0	↑	0	0	1
0	↑	1	1	0
1	↑	X	0	1

Verilog Code for D Flip-Flop with synchronous reset:

```
module d_ff (q, qb, d, clk, rst);
output reg q;
output qb;
input d, clk, rst;

always @(posedge clk)

begin
if (rst)
q = 0;
else
```

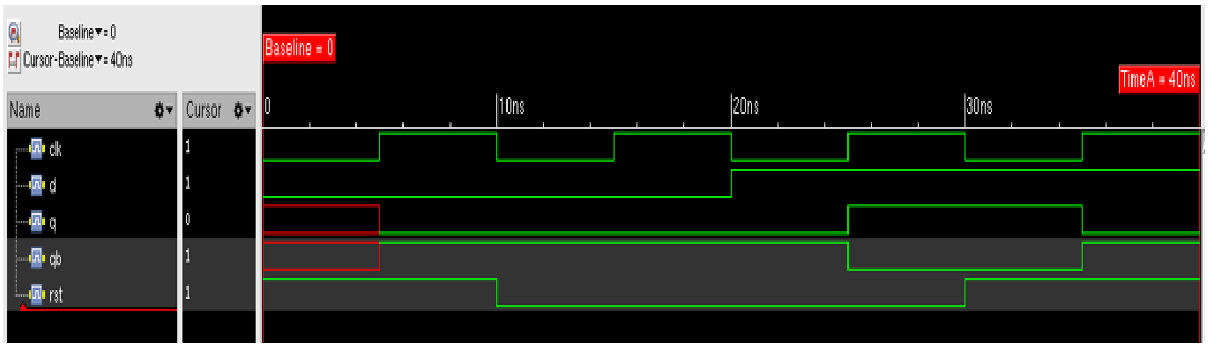
```
q = d;
end

assign qb = ~q;

endmodule
```

Testbench for D Flip-Flop with synchronous reset module:

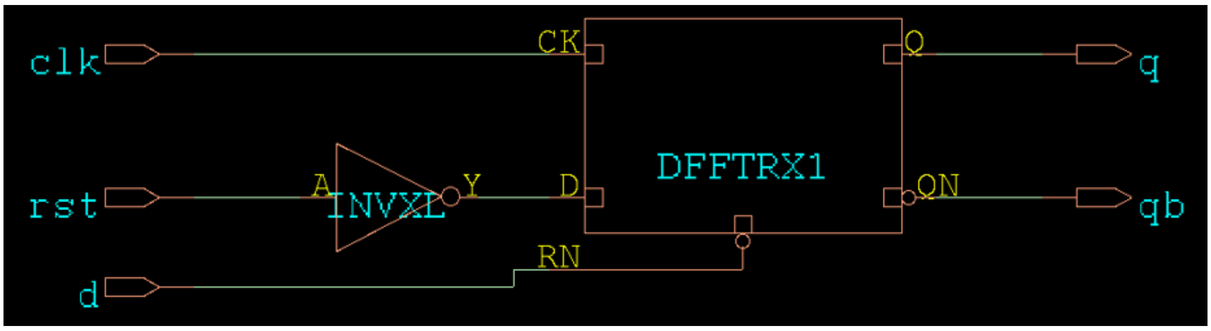
```
module d_ff_test;
reg clk, rst, d;
wire q, qb;
d_ff d1(q, qb, d, clk, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “rst =”, rst, “d =”, d, “q =”, q, “qb =”, qb);
#40 $finish;
end

initial
clk = 0;
always
#5 clk = ~clk;

initial
begin
rst = 1; d = 0;
#10 rst = 0;
#10 d = 1;
#10 rst = 1;
end

endmodule
```

Result:

__Simulation:__

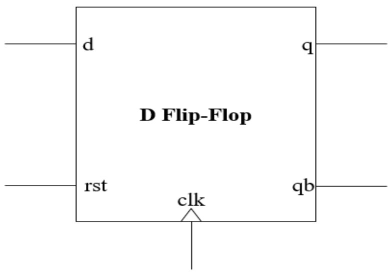


Schematic:



b) D Flip-Flop with asynchronous reset:

D Flip-Flop with asynchronous reset block diagram:



D Flip-Flop with synchronous reset truth table:

rst	clk	d	q	qb
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0
0	X	X	0	1

Verilog Code for D Flip-Flop with asynchronous reset:

```
module dff (q, qb, d, clk, rst);
output reg q;
output qb;
input d, clk, rst;

always @(posedge clk, negedge rst)

begin
if (!rst)
q = 0;
else
q = d;
end
```

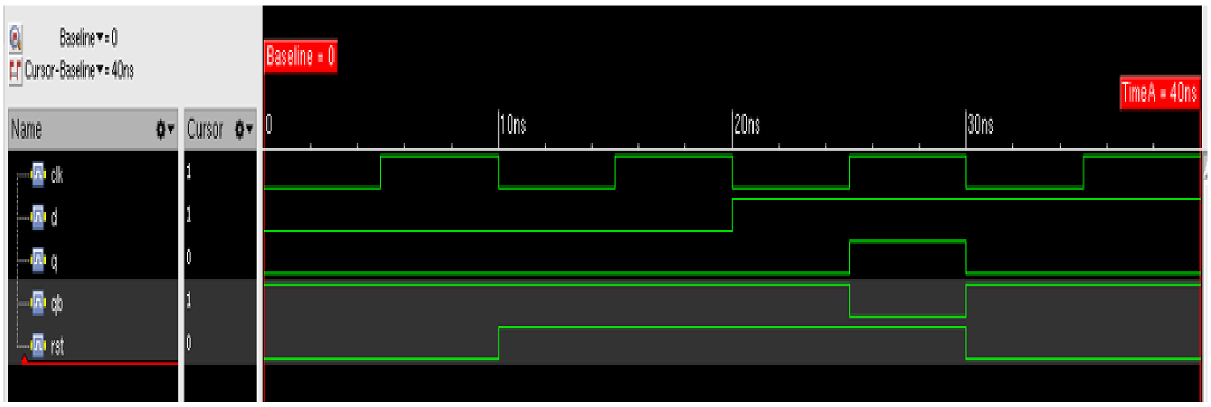
```
assign qb = ~q;

endmodule
```

Testbench for D Flip-Flop with asynchronous reset module:

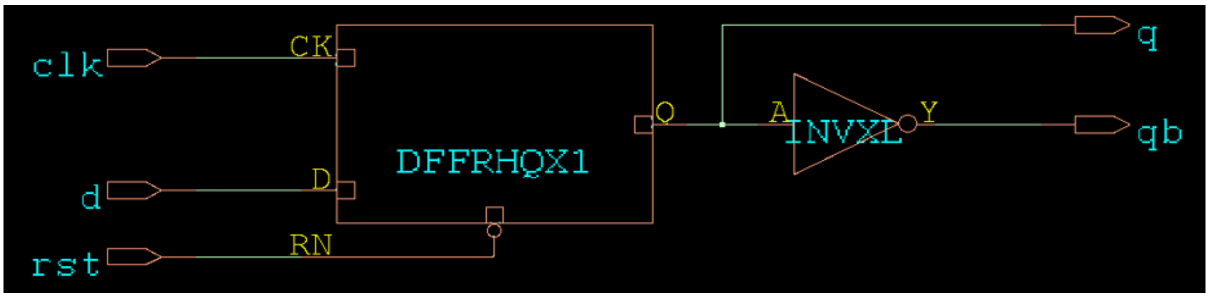
```
module dff_test;
reg d, rst, clk;
wire q, qb;
dff d1 (q, qb, d, clk, rst);
initial
begin
$monitor ("time=%0d", $time, "ns", "rst =", rst, "d =", d, "q =", q, "qb =", qb);
#40 $finish;
end

initial
begin
d = 0;
clk = 0;
end

always
#5 clk = ~clk;

initial
begin
rst = 0;
#10 rst =1;
#10 d =1;
#10 rst = 0;
end

endmodule
```

Result:

Simulation:



Schematic:



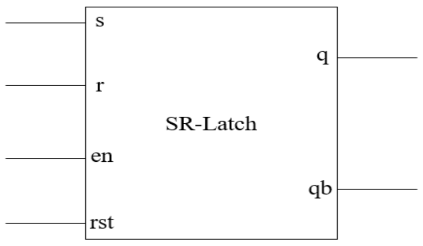
3. Write a Verilog code for SR-Latch and verify its functionality using test bench. Synthesize the design, tabulate the area, power and timing report.

Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

SR-Latch: An Set and Reset Latch has two inputs S and R and two outputs q and qb. The state of this latch is determined by the condition of q. If q is 1 the latch is said to be SET and if q is 0 the latch is said to be RESET.

SR-Latch block diagram:



SR-Latch truth table:

rst	en	S	r	q	qb
1	X	X	X	0	1
0	0	X	X	q	qb
0	1	0	0	q	qb
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	X	X

Verilog Code for SR-Latch:

```
module srlatch (q, qb, s, r, en, rst);
output reg q;
output qb;
input s, r, en, rst;

always @ (s, r, en, rst)

begin
if (rst)
q = 0;
else
if (en)
begin
if (s == 0 && r == 0)
```

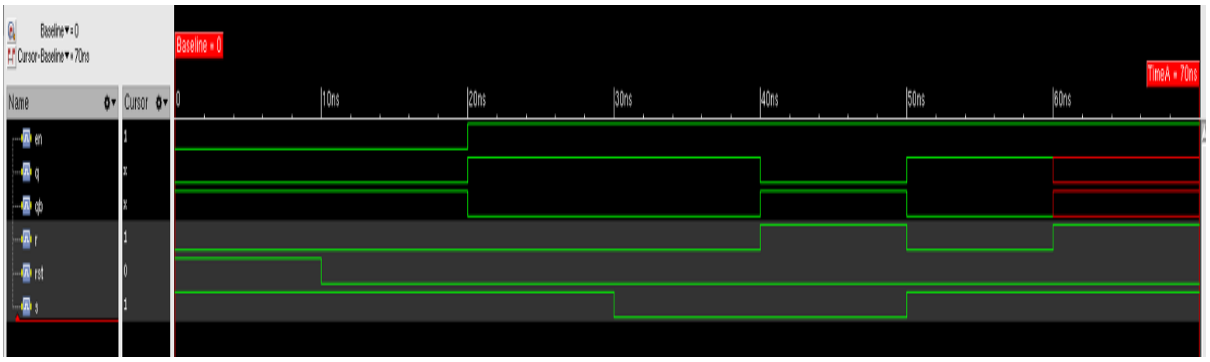
```
q = q;
else
if (s == 0 && r == 1)
q = 0;
else
if (s == 1 && r == 0)
q = 1;
else
if (s == 1 && r == 1)
q = 1’bx;
end
end

assign qb = ~q;

endmodule
```

Testbench for SR-Latch module:

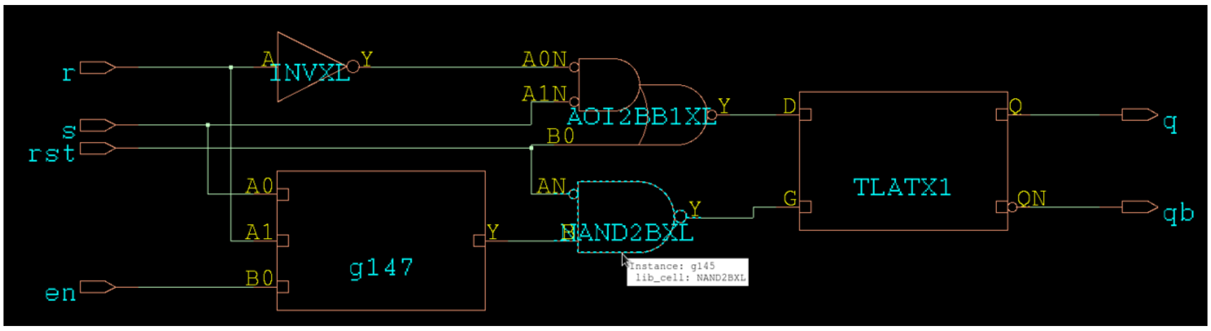
```
module srlatch_test;
reg s, r, en, rst;
wire q, qb;

srlatch s1(q, qb, s, r, en, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “s =”, s, “r =”, r, “en =”, en, “rst =”, rst, “q =”, q, “qb
=”, qb);
#70 $finish;
end

initial
begin
rst = 1; en = 0; s = 1; r = 0;
#10; rst = 0;
#10; en = 1;
#10; s = 0; r = 0;
#10; s = 0; r = 1;
#10; s = 1; r = 0;
#10; s = 1; r = 1;
end

endmodule
```

Result:

Simulation:



Schematic:



4. Write a Verilog code for SR Flip-Flop with synchronous and asynchronous reset and verify its functionality using test bench. Synthesize the design, tabulate the area, power and timing report.

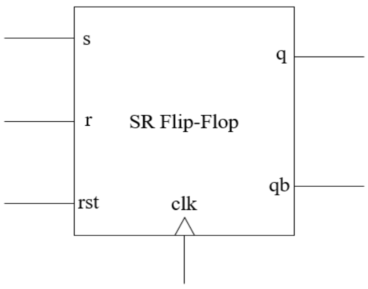
Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

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 labelled 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".*

a) SR Flip-Flop with synchronous reset:

SR Flip-Flop with synchronous reset block diagram:



SR Flip-Flop with synchronous reset truth table:

rst	**clk**	**S**	**r**	**q**	**qb**
1	↑	X	X	0	1
0	↑	0	0	q	qb
0	↑	0	1	0	1
0	↑	1	0	1	0
0	↑	1	1	X	X

Verilog Code for SR Flip-Flop with synchronous reset:

```
module sr_ff (q, qb, s, r, clk, rst);
output reg q;
output qb;
input s, r, clk, rst;

always @(posedge clk)

begin
```

```
if (rst)
q = 0;
else
if (s == 0 && r == 0)
q = q;
else
if (s == 0 && r == 1)
q = 0;
else
if (s == 1 && r == 0)
q = 1;
else
if (s == 1 && r == 1)
q = 1'bx;
end
assign qb = ~q;

endmodule
```

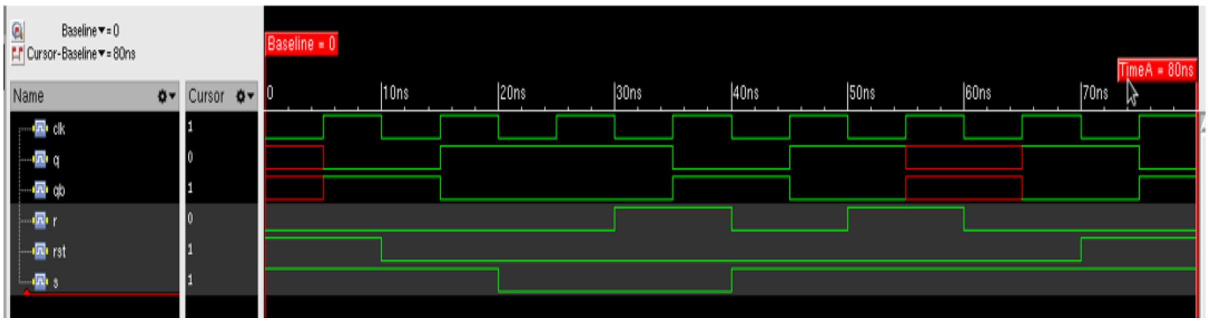
Testbench for SR Flip-Flop with synchronous reset module:

```
module sr_ff_test;
reg s, r, clk, rst;
wire q, qb;

sr_ff s1(q, qb, s, r, clk, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “s =”, s, “r =”, r, “rst =”, rst, “q =”, q, “qb =”, qb);
#80 $finish;
end

initial
clk = 1'b0;

always
#5 clk = ~clk;

initial
begin
rst = 1; s = 1; r = 0;
#10; rst = 0;
#10; s = 0; r = 0;
#10; s = 0; r = 1;
#10; s = 1; r = 0;
#10; s = 1; r = 1;
#10; s = 1; r = 0;
#10; rst = 1;
```

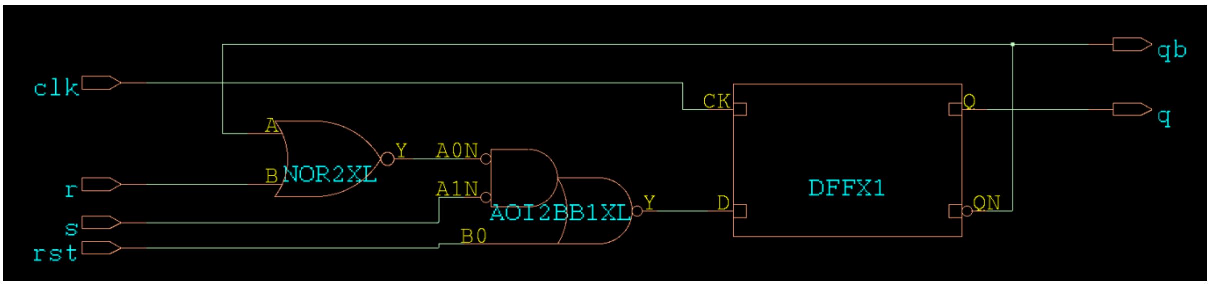
end
endmodule

Result:

Simulation:

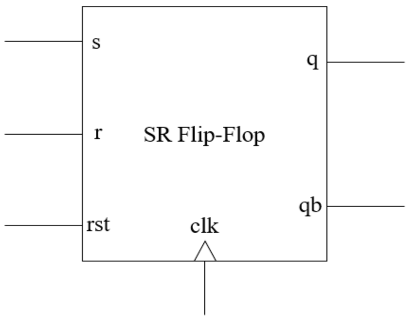


Schematic:



b) SR Flip-Flop with asynchronous reset:

SR Flip-Flop with asynchronous reset block diagram:



SR Flip-Flop with asynchronous reset truth table:

rst	clk	S	r	q	qb
0	X	X	X	0	1
1	↑	0	0	q	qb
1	↑	0	1	0	1
1	↑	1	0	1	0
1	↑	1	1	X	X

Verilog Code for SR Flip-Flop with asynchronous reset:

```
module srff (q, qb, s, r, clk, rst);
output reg q;
output qb;
input s, r, clk, rst;

always @(posedge clk, negedge rst)

begin
if (!rst)
q = 0;
else
if (s == 0 && r == 0)
q = q;
else
if (s == 0 && r == 1)
q = 0;
else
if (s == 1 && r == 0)
q = 1;
else
if (s == 1 && r == 1)
q = 1'bx;
end

assign qb = ~q;

endmodule
```

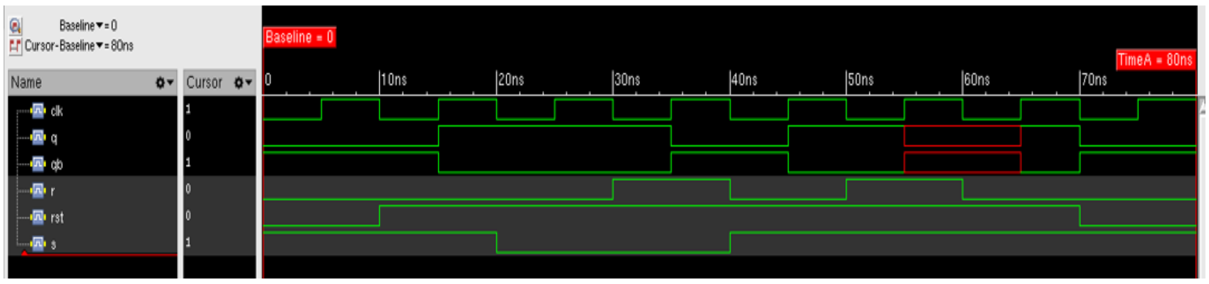
Testbench for SR Flip-Flop with asynchronous reset module:

```
module srff_test;
reg s, r, clk, rst;
wire q, qb;

srff s1(q, qb, s, r, clk, rst);

initial
begin
$monitor ("time = %0d", $time, "ns", "s =", s, "r =", r, "rst =", rst, "q =", q, "qb =", qb);
#80 $finish;
end

initial
clk = 1'b0;

always
#5 clk = ~clk;
```

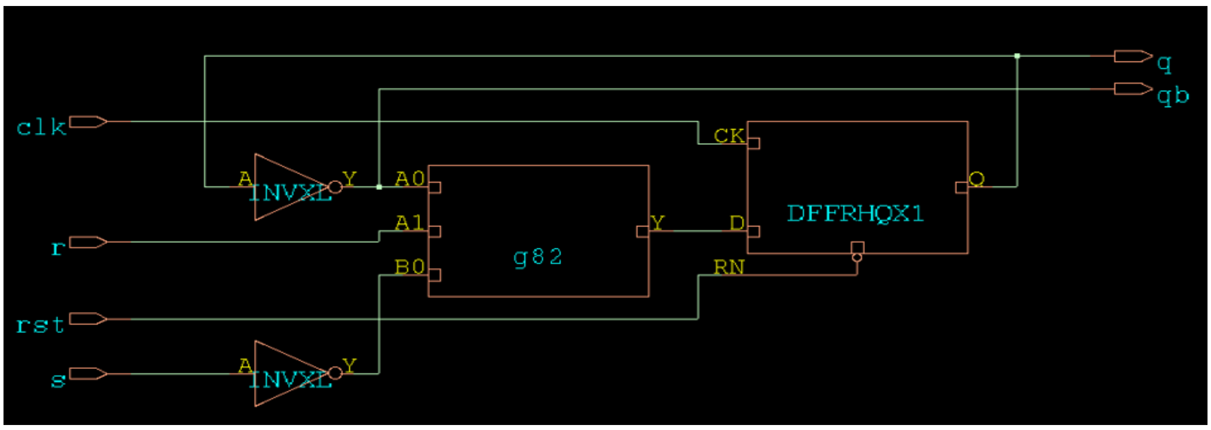
```
initial
begin
rst = 0; s = 1; r = 0;
#10; rst = 1;
#10; s = 0; r = 0;
#10; s = 0; r = 1;
#10; s = 1; r = 0;
#10; s = 1; r = 1;
#10; s = 1; r = 0;
#10; rst = 0;
end

endmodule
```

Result:

Simulation:



Schematic:



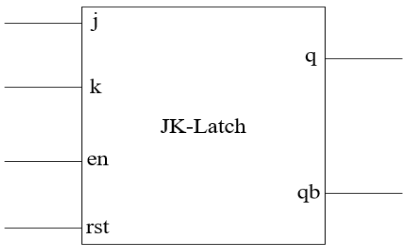
5. Write a Verilog code for JK-Latch and verify its functionality using test bench. Synthesize the design, tabulate the area, power and timing report.

Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

JK-Latch: *JK latch is similar to SR latch. This latch consists of 2 inputs j and k. The ambiguous state has been eliminated here: when the inputs of JK latch are high, then output toggles.*

JK-Latch block diagram:



JK-Latch truth table:

rst	**en**	**J**	**k**	**q**	**qb**
1	X	X	X	0	1
0	0	X	X	q	qb
0	1	0	0	q	qb
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	~q	~qb

Verilog Code for JK-Latch:

```
module jklatch (q, qb, j, k, en, rst);
output reg q;
output qb;
input j, k, en, rst;

always @ (j, k, en, rst)

begin
if (rst)
q = 0;
else
if (en)
begin
if (j == 0 && k == 0)
q = q;
```

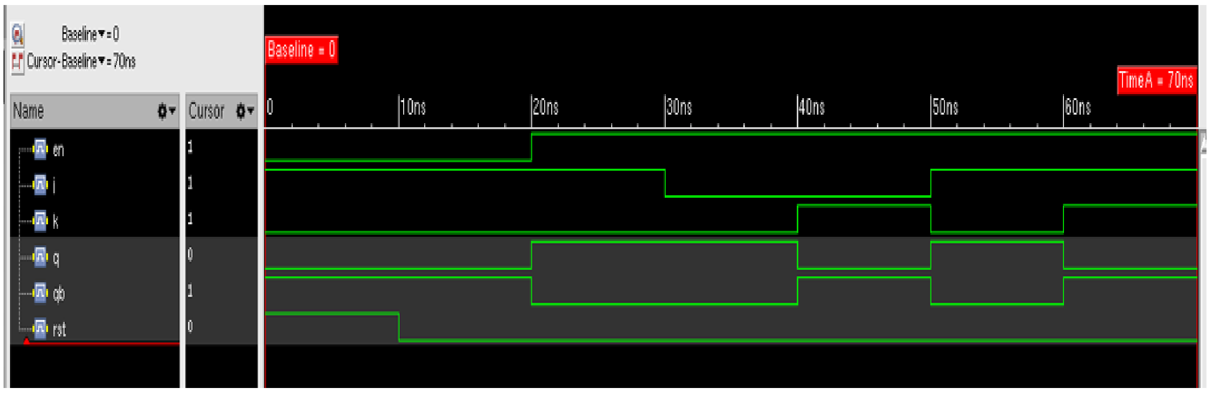
```
else
if (j == 0 && k == 1)
q = 0;
else
if (j == 1 && k == 0)
q = 1;
else
if (j == 1 && k == 1)
q = ~q;
end
end

assign qb = ~q;

endmodule
```

Testbench for JK-Latch module:

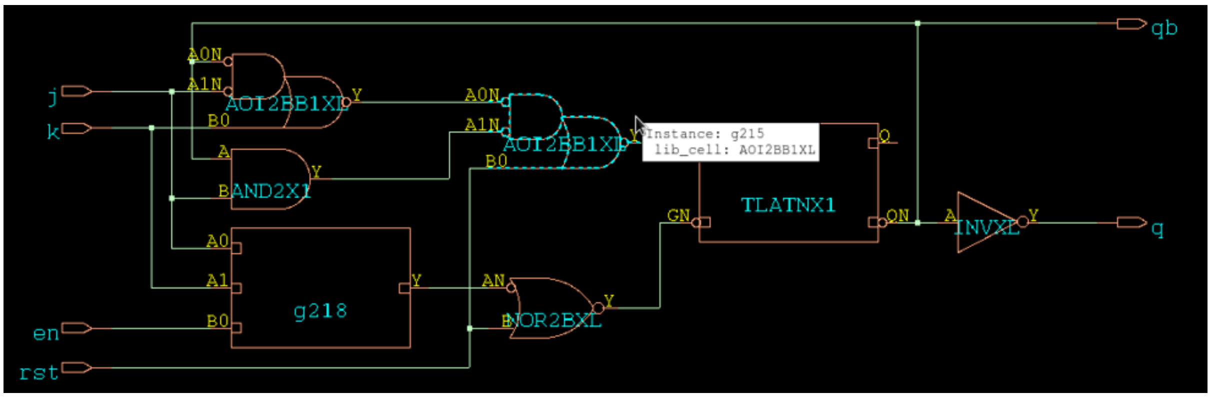
```
module jklatch_test;
reg j, k, en, rst;
wire q, qb;

jklatch j1(q, qb, j, k, en, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “j =”, j, “k =”, k, “en =”, en, “rst =”, rst, “q =”, q, “qb
=”, qb);
#70 $finish;
end

initial
begin
rst = 1; en = 0; j = 1; k = 0;
#10; rst = 0;
#10; en = 1;
#10; j = 0; k = 0;
#10; j = 0; k = 1;
#10; j = 1; k = 0;
#10; j = 1; k = 1;
end

endmodule
```

Result:

Simulation:



Schematic:



6. Write a Verilog code for JK Flip-Flop with synchronous and asynchronous reset and verify its functionality using test bench. Synthesize the design, tabulate the area, power and timing report.

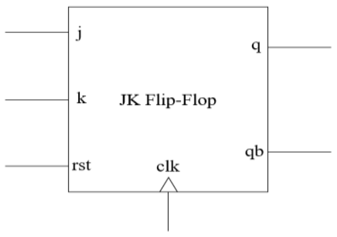
Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

JK Flip-Flop: *The JK flip-flop is the most versatile of the basic flip flops. A JK flip-flop is used in clocked* sequential *logic circuits to store one bit of data. It is almost identical in function to an SR flip flop. The only difference is eliminating the undefined state where both S and R are 1. Due to this additional clocked input, a JK flip-flop has four possible input combinations, such as "logic 1", "logic 0", "no change" and "toggle".*

a) JK Flip-Flop with synchronous reset:

JK Flip-Flop with synchronous reset block diagram:



JK Flip-Flop with synchronous reset truth table:

rst	**clk**	**J**	**k**	**q**	**qb**
1	↑	X	X	0	1
0	↑	0	0	q	qb
0	↑	0	1	0	1
0	↑	1	0	1	0
0	↑	1	1	~q	~qb

Verilog Code for JK Flip-Flop with synchronous reset:

```
module jk_ff (q, qb, j, k, clk, rst);
output reg q;
output qb;
input j, k, clk, rst;

always @(posedge clk)

begin
if (rst)
q = 0;
```

```
else
if (j == 0 && k == 0)
q = q;
else
if (j == 0 && k == 1)
q = 0;
else
if (j == 1 && k == 0)
q = 1;
else
if (j == 1 && k == 1)
q = ~q;
end

assign qb = ~q;

endmodule
```

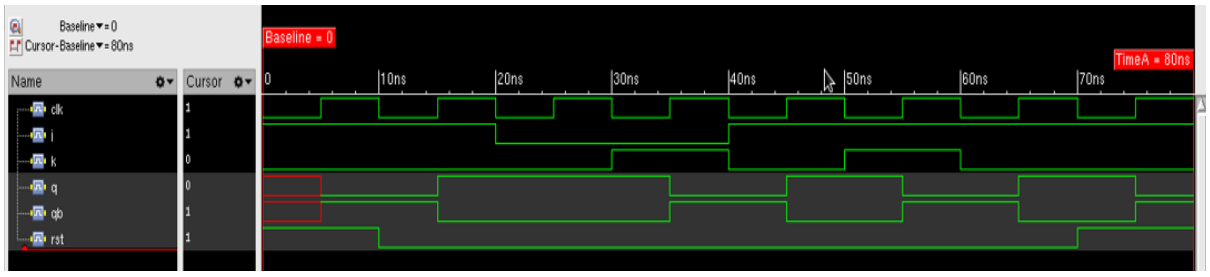
Testbench for JK Flip-Flop with synchronous reset module:

```
module jk_ff_test;
reg j, k, clk, rst;
wire q, qb;

jk_ff j1(q, qb, j, k, clk, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “j =”, j, “k =”, k, “rst =”, rst, “q =”, q, “qb =”, qb);
#80 $finish;
end

initial
clk = 1’b0;

always
#5 clk = ~clk;

initial
begin
rst = 1; j = 1; k = 0;
#10; rst = 0;
#10; j = 0; k = 0;
#10; j = 0; k = 1;
#10; j = 1; k = 0;
#10; j = 1; k = 1;
#10; j = 1; k = 0;
#10; rst = 1;
end
```

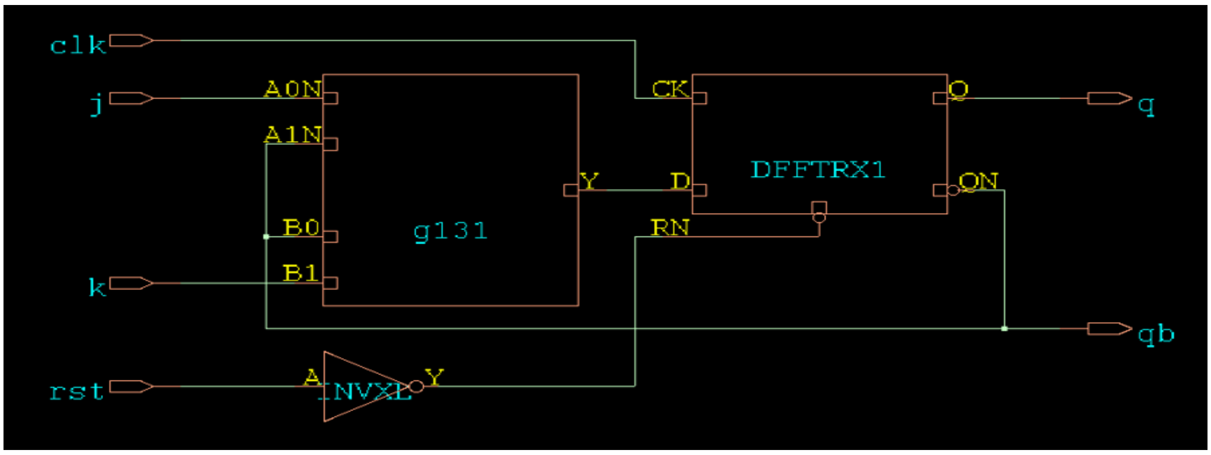
endmodule

Result:

Simulation:

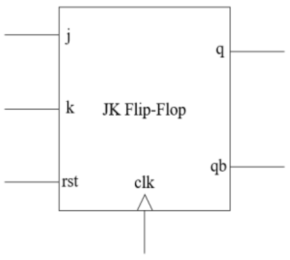


Schematic:



b) JK Flip-Flop with asynchronous reset:

JK Flip-Flop with asynchronous reset block diagram:



JK Flip-Flop with asynchronous reset truth table:

rst	clk	J	k	q	qb
0	X	X	X	0	1
1	↑	0	0	q	qb
1	↑	0	1	0	1
1	↑	1	0	1	0
1	↑	1	1	~q	~qb

Verilog Code for JK Flip-Flop with asynchronous reset:

```
module jkff (q, qb, j, k, clk, rst);
output reg q;
output qb;
input j, k, clk, rst;

always @(posedge clk, negedge rst)

begin
if (!rst)
q = 0;
else
if (j == 0 && k == 0)
q = q;
else
if (j == 0 && k == 1)
q = 0;
else
if (j == 1 && k == 0)
q = 1;
else
if (j == 1 && k == 1)
q = ~q;
end

assign qb = ~q;

endmodule
```

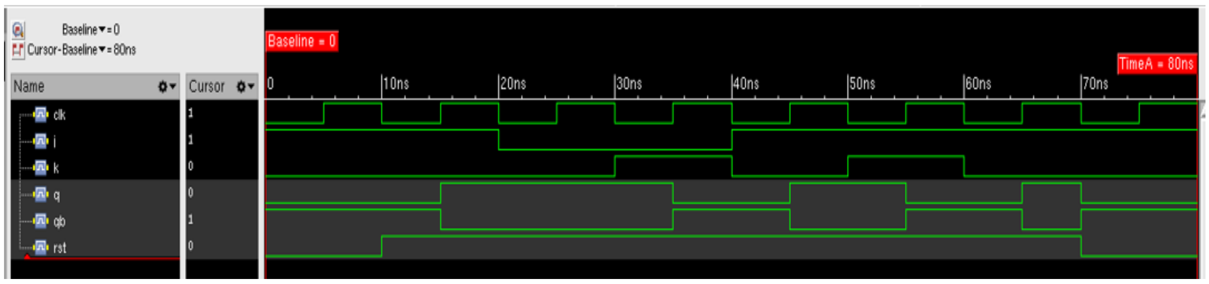
Testbench for JK Flip-Flop with asynchronous reset module:

```
module jkff_test;
reg j, k, clk, rst;
wire q, qb;

jkff j1(q, qb, j, k, clk, rst);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “j =”, j, “k =”, k, “rst =”, rst, “q =”, q, “qb =”, qb);
#80 $finish;
end

initial
clk = 1’b0;

always
#5 clk = ~clk;
```

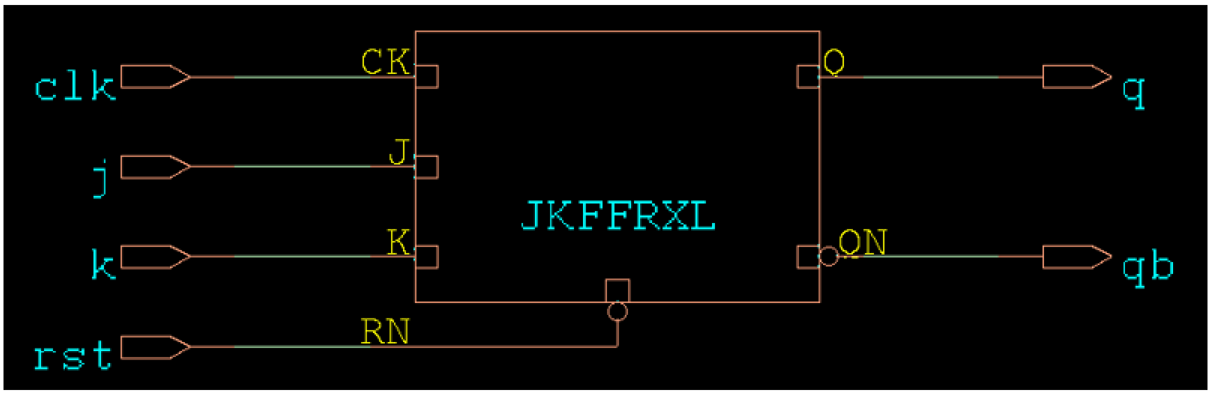
```
initial
begin
rst = 0; j = 1; k = 0;
#10; rst = 1;
#10; j = 0; k = 0;
#10; j = 0; k = 1;
#10; j = 1; k = 0;
#10; j = 1; k = 1;
#10; j = 1; k = 0;
#10; rst = 0;

end

endmodule
```

Result:

Simulation:



Schematic:



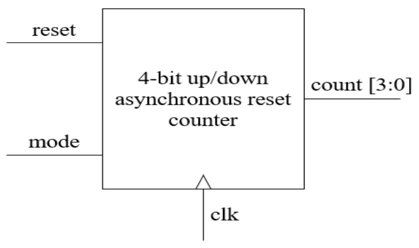
7. **Write verilog code for 4-bit up/down asynchronous reset counter and carry out the following:**
   a. **Verify the functionality using testbench**
   b. **Synthesize the design by setting area and timing constraints. Obtain the gate level netlist, find the critical path and maximum frequency of operation. Record the area requirement in terms of number of cells required and properties of each cell in terms of driving strength, power and area requirements.**
   c. **Perform the above for 32-bit up/down counter and identify the critical path, delay of the critical path, and maximum frequency of operation, total number of cells required and total area.**

Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

4-bit up/down: *An up/down counter is a digital counter which can be set to count either from 0 to maximum value or maximum value to 0. The direction of the count (mode) is selected using a single bit input. The up/down counter has 3 inputs - clk, reset and a up or down mode input. The output is count which is 4 bit in size. When Up mode is selected, counter counts from 0 to 15 and then again from 0 to 15. When Down mode is selected, counter counts from 15 to 0 and then again from 15 to 0. Changing mode doesn't reset the Count value to zero. You have to apply high value to reset, to reset the counter output.*

4-bit up/down asynchronous reset counter block diagram:



4-bit up/down asynchronous reset counter truth table:

clk	reset	mode	Count
X	1	X	0
↑	0	0	0
↑	0	0	1
↑	0	0	2
↑	0	0	3
↑	0	0	4
↑	0	1	3
↑	0	1	2
↑	0	1	1
↑	0	1	0

↑	0	1	F
↑	0	1	E
↑	0	1	D
↑	1	X	0

a) 4-bit up/down asynchronous reset counter

Verilog code for 4-bit up/down asynchronous reset counter:

```
module cnt_updown (count, mode, clk, reset);
output reg [3:0] count;
input mode;
input clk, reset;

always @ (posedge clk, posedge reset)

if (reset)
count = 4'b0;
else
if (mode)
count = count + 1;
else
count = count - 1;

endmodule
```

Testbench for 4-bit up/down asynchronous reset counter:

```
module tb_cnt_updown;
reg mode;
reg clk, reset;
wire [3:0] count;

cnt_updown M1(count, mode, clk, reset);

initial
begin
$monitor ("time = %0d", $time, "ns", "reset = 0x%0h", reset, " mode = 0x%0h", mode, " count
= 0x%0h", count);
#320 $finish;
end

always
#5 clk = ~clk;

initial
```

```
begin
mode = 1; clk=0; reset=1;
#10; reset = 0;
#100; mode = 0;
#50; reset = 1;
#30; reset = 0;
#100; mode = 1;
#10; reset = 1;
end

endmodule
```

Creating an SDC File:

- In terminal type "gedit constraints_top.sdc" to create an SDC file. ***<u>(This file is common for all programs)</u>***
- The SDC file must contain the following commands.

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]

set_input_delay -max 0.8 -clock clk [all_inputs]
set_output_delay -max 0.8 -clock clk [all_outputs]

set_input_transition 0.2 [all_inputs]
set_max_capacitance 30 [get_ports]

set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]

set_clock_uncertainty 0.01 [get_ports "clk"]

set_input_transition 0.12 [all_inputs]
set_load 0.15 [all_outputs]
set_max_fanout 30.00 [current_design]
```

Performing Synthesize:

- The following are commands to perform synthesize (4-bit and 32-bit up/down asynchronous reset counter)

```
genus -gui
read_lib /home/install/cad/slow.lib
read_hdl cnt_updown.v
elaborate
read_sdc constraints_top.sdc
```

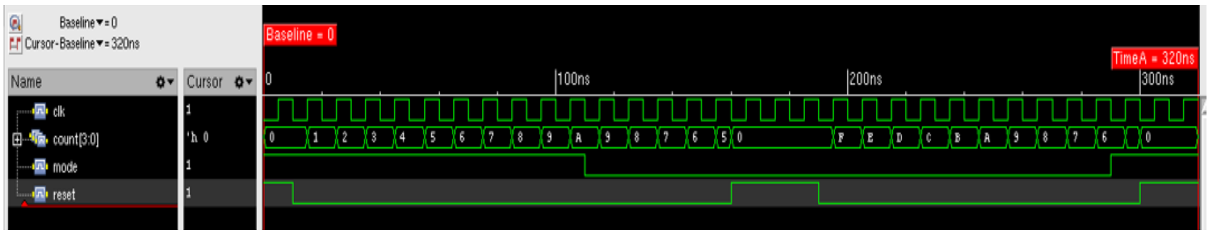
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt

write_hdl > cnt_updown_netlist.v
write_sdc > cnt_updown.sdc

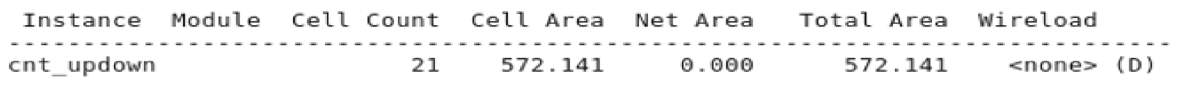
report_power
report_gates
report_timing
report_area
report_qor -levels_of_logic -power -exclude_constant_nets

Result:

Simulation:

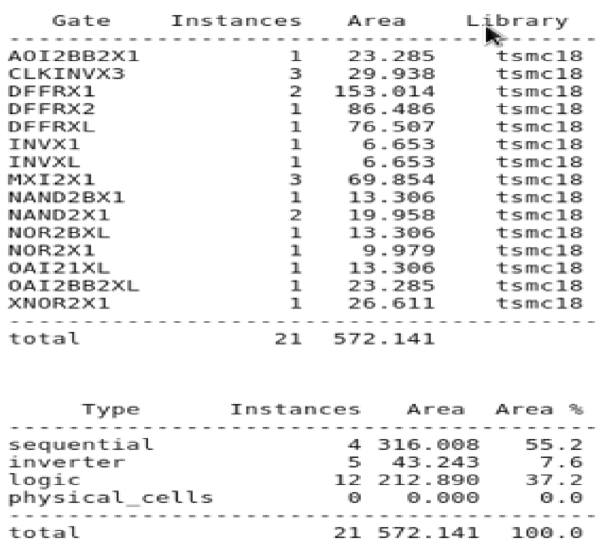


Area Report:



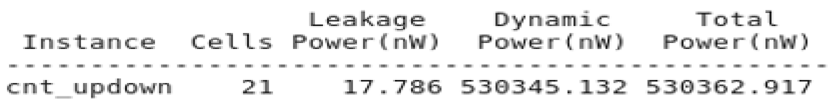
```
  Instance   Module  Cell Count  Cell Area  Net Area   Total Area  Wireload
-------------------------------------------------------------------------------
 cnt_updown                  21    572.141     0.000      572.141    <none> (D)
```

Gates Report:



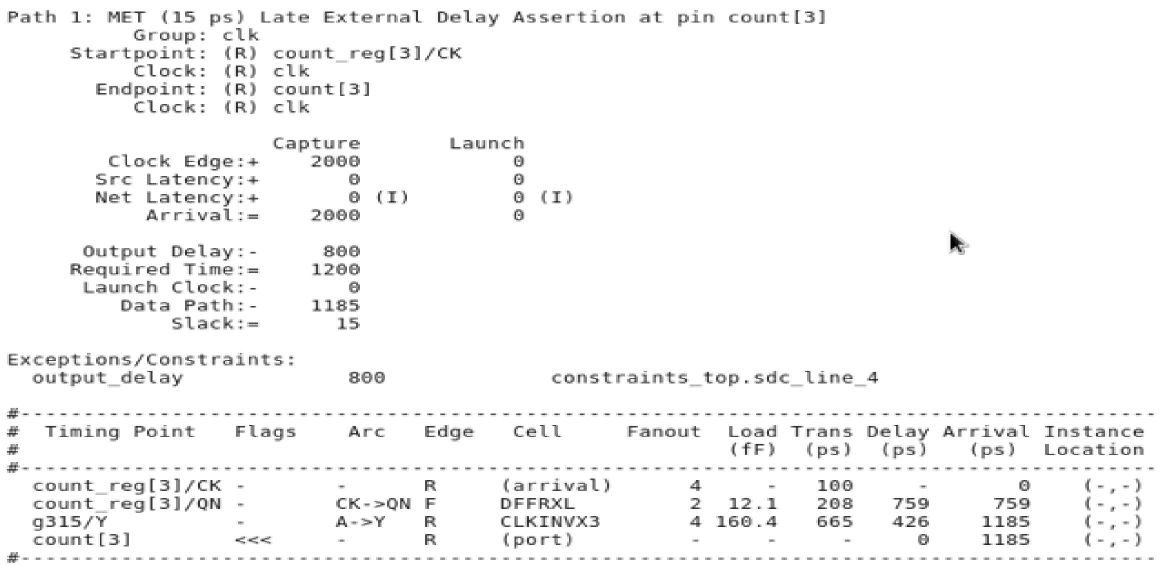
```
    Gate      Instances   Area      Library
-------------------------------------------
AOI2BB2X1            1    23.285     tsmc18
CLKINVX3             3    29.938     tsmc18
DFFRX1               2   153.014     tsmc18
DFFRX2               1    86.486     tsmc18
DFFRXL               1    76.507     tsmc18
INVX1                1     6.653     tsmc18
INVXL                1     6.653     tsmc18
MXI2X1               3    69.854     tsmc18
NAND2BX1             1    13.306     tsmc18
NAND2X1              2    19.958     tsmc18
NOR2BXL              1    13.306     tsmc18
NOR2X1               1     9.979     tsmc18
OAI21XL              1    13.306     tsmc18
OAI2BB2XL            1    23.285     tsmc18
XNOR2X1              1    26.611     tsmc18
-------------------------------------------
total               21   572.141


     Type        Instances    Area    Area %
-------------------------------------------
sequential               4 316.008    55.2
inverter                 5  43.243     7.6
logic                   12 212.890    37.2
physical_cells           0   0.000     0.0
-------------------------------------------
total                   21 572.141   100.0
```

Power Report:



```
                          Leakage     Dynamic      Total
  Instance     Cells  Power(nW)  Power(nW)   Power(nW)
-----------------------------------------------------------
cnt_updown      21     17.786 530345.132 530362.917
```

Timing Report:



```
Path 1: MET (15 ps) Late External Delay Assertion at pin count[3]
          Group: clk
     Startpoint: (R) count_reg[3]/CK
          Clock: (R) clk
       Endpoint: (R) count[3]
          Clock: (R) clk

                      Capture       Launch
     Clock Edge:+        2000            0
    Src Latency:+           0            0
    Net Latency:+           0 (I)        0 (I)
        Arrival:=        2000            0

   Output Delay:-         800
  Required Time:=        1200
   Launch Clock:-           0
      Data Path:-        1185
          Slack:=          15

Exceptions/Constraints:
  output_delay             800            constraints_top.sdc_line_4

#--------------------------------------------------------------------------------------------
# Timing Point      Flags    Arc    Edge    Cell        Fanout  Load Trans Delay Arrival Instance
#                                                               (fF)  (ps)  (ps)   (ps)  Location
#--------------------------------------------------------------------------------------------
  count_reg[3]/CK -          -      R     (arrival)        4      -   100     -      0    (-,-)
  count_reg[3]/QN -          CK->QN F     DFFRXL           2   12.1   208   759    759    (-,-)
  g315/Y          -          A->Y   R     CLKINVX3         4  160.4   665   426   1185    (-,-)
  count[3]        <<<        -      R     (port)           -      -     -     0   1185    (-,-)
#--------------------------------------------------------------------------------------------
```

Required Time = 1200ps

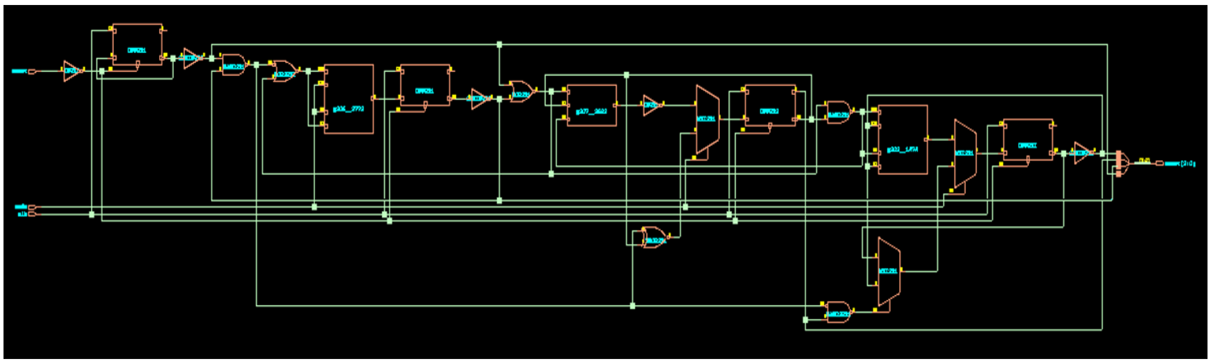
Arrival Time = 1185ps

Slack = Required Time – Arrival Time = 15ps

Critical Path Delay = Clock Period – Slack = 2ns – 0.015ns = 1.985ns

Maximum Frequency of operation = $\frac{1}{\text{Arrival Time}}$ = *843.88MHz*

Schematic:



b) 32-bit up/down asynchronous reset counter

Verilog code for 32-bit up/down asynchronous reset counter:

```
module cnt_updown (count, mode, clk, reset);
output reg [31:0] count;
input mode;
input clk, reset;

always @ (posedge clk, posedge reset)

if (reset)
count = 32'b0;
else
if (mode)
count = count + 1;
else
count = count - 1;

endmodule
```

Testbench for 32-bit up/down asynchronous reset counter:

```
module tb_cnt_updown;
reg mode;
reg clk, reset;
wire [31:0] count;

cnt_updown M1(count, mode, clk, reset);

initial
begin
$monitor ("time = %0d", $time, "ns", "reset = 0x%0h", reset, " mode = 0x%0h", mode, " count
= 0x%0h", count);
#900 $finish;
end

always
#5 clk = ~clk;

initial
begin
mode = 1; clk=0; reset=1;
#10; reset = 0;
#300; mode = 0;
#150; reset = 1;
```

```
#90; reset = 0;
#300; mode = 1;
#10; reset = 1;
end

endmodule
```

8. Write verilog code for 32-bit ALU supporting four logical and four arithmetic operations, use case statement and if statement for ALU behavioral modeling.

a. Perform functional verification using test bench.

b. Synthesize the design targeting suitable library by setting area and timing constraints.

c. For various constraints set, tabulate the area, power and delay for the synthesized netlist.

d. Identify the critical path and set the constraints to obtain optimum gate level netlist with suitable constraints.

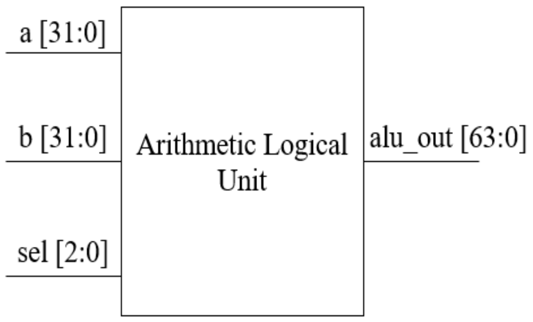
Compare the synthesize results of ALU modeled using if and case statements.

Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

Arithmetic Logical Unit (ALU): *ALU is the fundamental building block of the processor, which is responsible for carrying out the arithmetic and logical functions. ALU comprises of combinational logic that implements arithmetic operations such as addition, subtraction, multiplication, division etc.., and logical operations such as AND, OR, NAND, NOR etc.., The ALU reads two input operands a and b. The operation to perform on these input operands is selected using control input sel. The ALU performs the selected operation on the input operands a and b and produces the output alu_out.*

Arithmetic Logical Unit (ALU) block diagram:



Arithmetic Logical Unit (ALU) truth table:

a	b	sel	alu_out
32'h FEDCBA98	32'h 89ABCDEF	0	64'h 00000001_88888887
		1	64'h 00000000_7530ECA9
		2	64'h 890F2A50_AD05EBE8
		3	64'h 00000000_00000001
		4	64'h 00000000_88888888
		5	64'h 00000000_FFFFFFFF
		6	64'h 00000000_77777777
		7	64'h FFFFFFFF_88888888

a) 32-bit ALU using case statement:

Verilog code for 32-bit ALU using case statement:

```
module alu (a, b, sel, alu_out);
input [31:0] a, b;
input [2:0] sel;
output reg [63:0] alu_out;
always @ (*)
begin
case (sel)
3’b000: alu_out = a + b;
3’b001: alu_out = a - b;
3’b010: alu_out = a * b;
3’b011: alu_out = a / b;
3’b100: alu_out = a & b;
3’b101: alu_out = a | b;
3’b110: alu_out = a ^ b;
3’b111: alu_out = ~(a ^ b);
default:;
endcase
end
endmodule
```

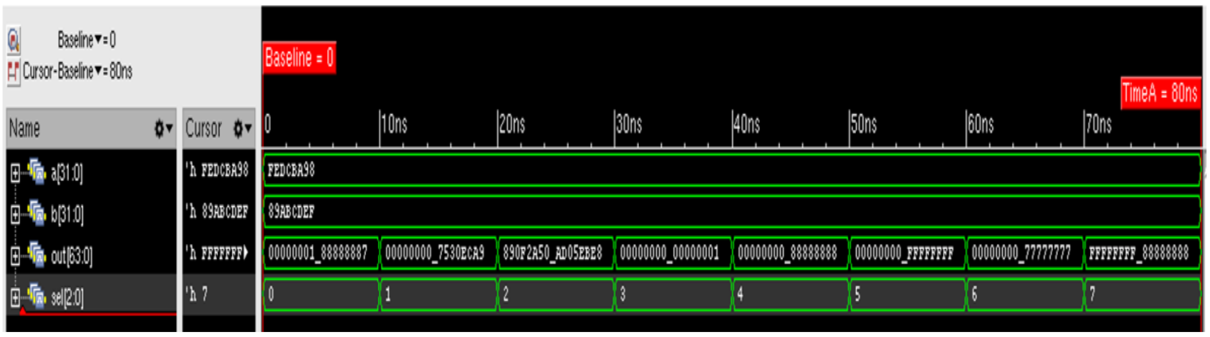
Testbench for 32-bit ALU using case statement:

```
module alu_test;
reg [31:0] a, b;
reg [2:0] sel;
wire [63:0] alu_out;

alu a1 (a, b, sel, alu_out);

initial
begin
a = 32’hFEDCBA98;
b = 32’h89ABCDEF;
sel = 3’b000;
$monitor (“a = 0x%0h  b = 0x%0h  sel = 0x%0h  alu_out = 0x%0h”, a, b, sel, alu_out);
#80; $finish;
end
```

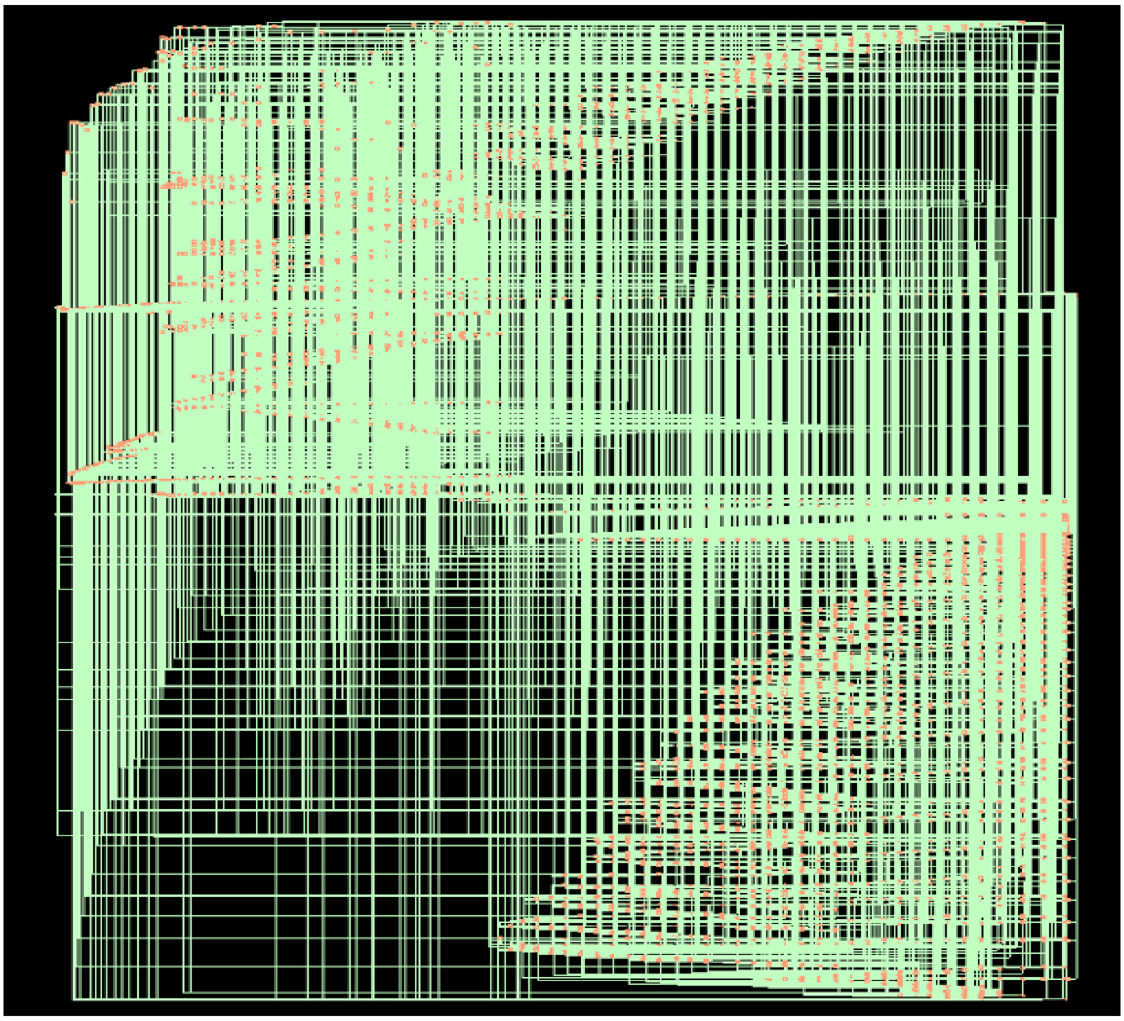
```
always
#10 sel = sel + 3'b001;

endmodule
```

Result:

Simulation:



Schematic:



b) 32-bit ALU using if statement:

Verilog code for 32-bit ALU using if statement:

```
module alu (a, b, sel, alu_out);
input [31:0] a, b;
input [2:0] sel;
output reg [63:0] alu_out;
always @ (*)
begin
if (sel = = 3’b000)
alu_out = a + b;
else if (sel == 3’b001)
alu_out = a – b;
else if (sel == 3’b010)
alu_out = a * b;
else if (sel == 3’b011)
alu_out = a / b;
else if (sel == 3’b100)
alu_out = a & b;
else if (sel == 3’b101)
alu_out = a | b;
else if (sel == 3’b110)
alu_out = a ^ b;
else
alu_out = ~(a ^ b);
end
endmodule
```

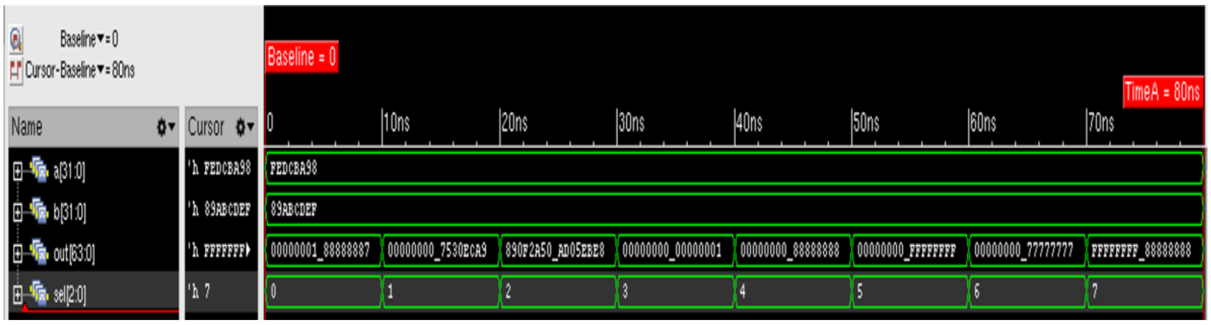
Testbench for 32-bit ALU using if statement:

```
module alu_test;
reg [31:0] a, b;
reg [2:0] sel;
wire [63:0] alu_out;

alu a1 (a, b, sel, alu_out);

initial
begin
a = 32’hFEDCBA98;
```

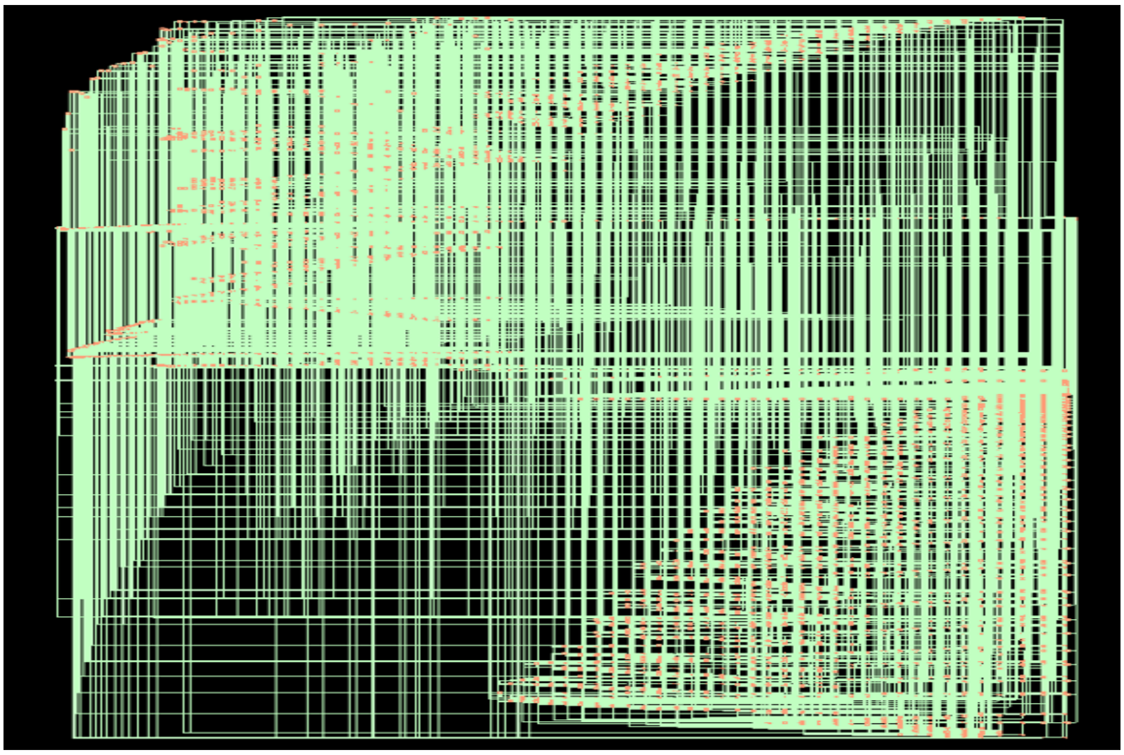
```
b = 32'h89ABCDEF;
sel = 3'b000;
$monitor ("a = 0x%0h  b = 0x%0h  sel = 0x%0h  alu_out = 0x%0h", a, b, sel, alu_out);
#80; $finish;
end

always
#10 sel = sel + 3'b001;

endmodule
```

Result:

__Simulation:__



__Schematic:__



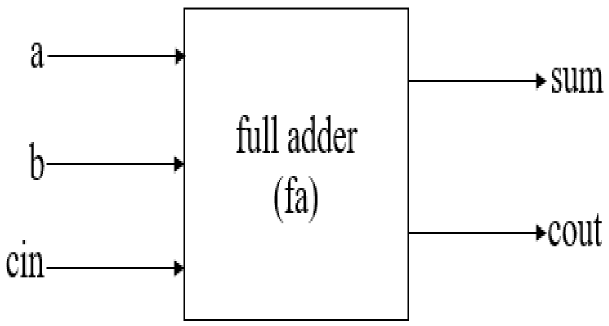
9. Write verilog code for 4-bit adder and verify its functionality using test bench. Synthesize the design by setting proper constraints and obtain netlist. From the report generated identify the critical path, and maximum delay, total number of cells, power requirement and total area required. Change the constraints and obtain optimum synthesis results.

Tools required:

- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

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 cin. The output carry is designated as cout and the normal output is sum.

Full-Adder block diagram:

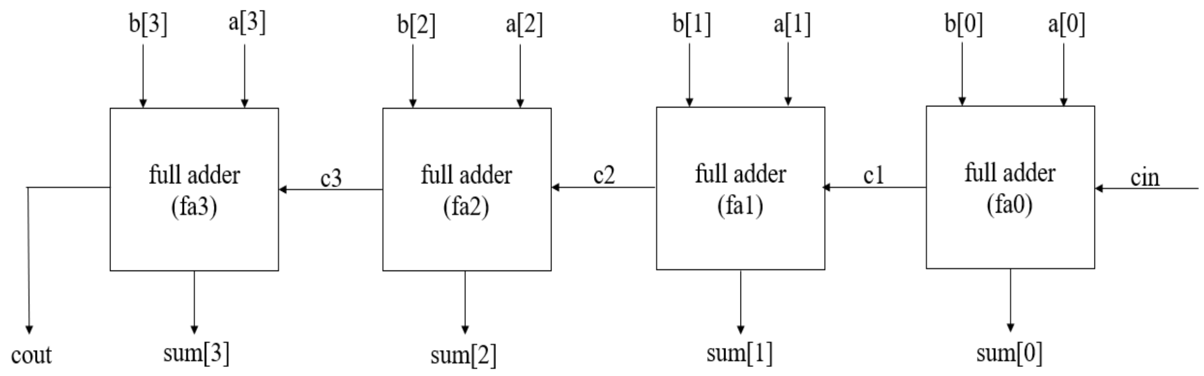


Full-Adder truth table:

Inputs			Outputs	
a	**b**	**cin**	**sum**	**cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

4-bit Full Adder: Binary adders are implemented to add two binary numbers. So in order to add two 4-bit binary numbers, we will need to use 4 full-adders. The 4 full-adders are connected in cascade form. In this implementation, cout of each full-adder is connected to next cin.

4-bit Full-Adder block diagram:



4-bit Full-Adder truth table:

Inputs			Outputs	
a	**b**	**cin**	**sum**	**cout**
4’b0001	4’b1010	0	4’b1011	0
4’b1100	4’b1101		4’b1001	1
4’b0101	4’b1011		4’b0000	1
4’b1111	4’b1111		4’b1110	1
4’b0001	4’b1010	1	4’b1100	0
4’b1100	4’b1101		4’b1010	1
4’b0101	4’b1011		4’b0001	1
4’b1111	4’b1111		4’b1111	1

Verilog code for 1-bit full-adder:

```
module full_adder (a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;

assign sum = a ^ b ^ cin;
assign cout = (a & b) | (cin & (a ^ b));

endmodule
```

Verilog code for 4-bit full-adder:

```
module four_bit_adder (a, b, cin, sum, cout);
input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;

wire c1, c2, c3;
```

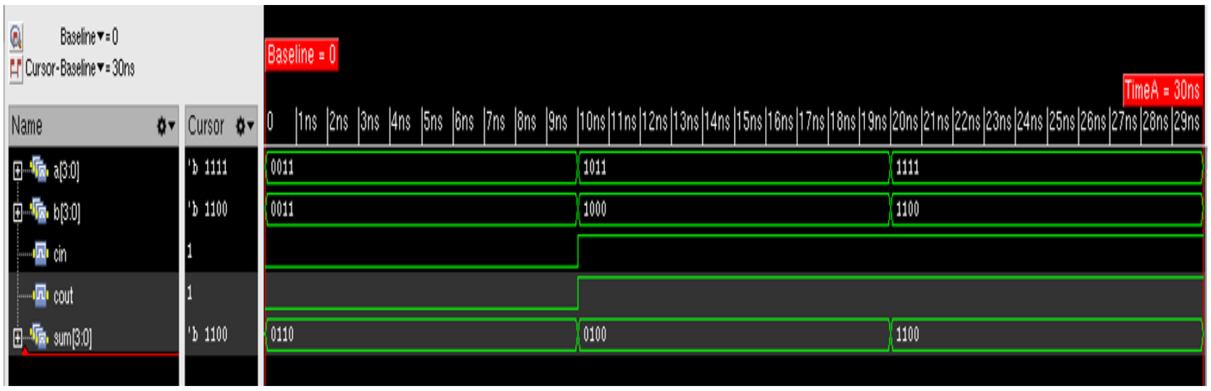
```
full_adder fa0 (a[0], b[0], cin, sum[0], c1);
full_adder fa1 (a[1], b[1], c1, sum[1], c2);
full_adder fa2 (a[2], b[2], c2, sum[2], c3);
full_adder fa3 (a[3], b[3], c3, sum[3], cout);

endmodule
```

Test bench for 4-bit full-adder:

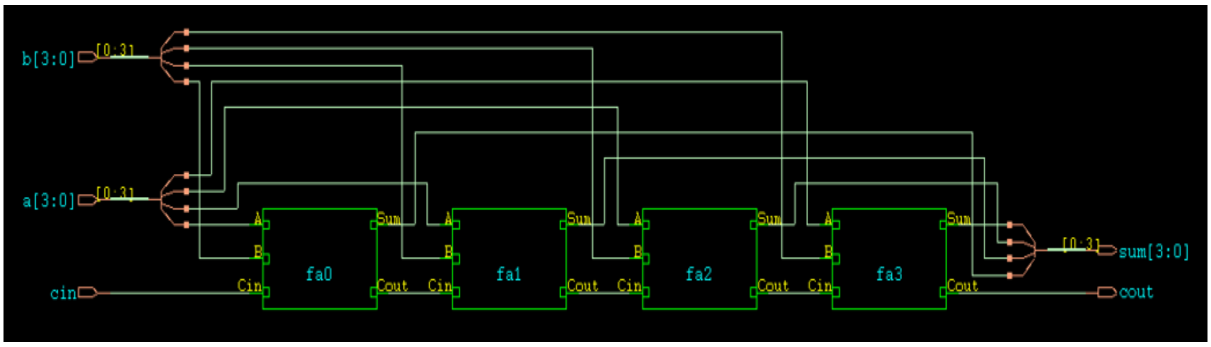
```
module test_adder;
reg [3:0] a, b;
reg cin;
wire [3:0] sum;
wire cout;

four_bit_adder f1 (a, b, cin, sum, cout);

initial
begin
$monitor (“time = %0d”, $time, “ns”, “a = %0b”, a, “b = %0b”, b, “cin = %0b”, cin, “sum = %0b”, sum, “cout = %0b”, cout);
#30 $finish;
end

initial
begin
a = 4’b0011; b = 4’b0011; cin = 1’b0;
#10; a = 4’b1011; b = 4’b1000; cin = 1’b1;
#10; a = 4’b1111; b = 4’b1100; cin = 1’b1;
end

endmodule
```

Result:

Simulation:


Baseline ▾ = 0
Cursor-Baseline ▾ = 30ns
Baseline = 0
TimeA = 30ns
Name
Cursor
0 1ns 2ns 3ns 4ns 5ns 6ns 7ns 8ns 9ns 10ns 11ns 12ns 13ns 14ns 15ns 16ns 17ns 18ns 19ns 20ns 21ns 22ns 23ns 24ns 25ns 26ns 27ns 28ns 29ns
a[3:0] 'b 1111 0011 1011 1111
b[3:0] 'b 1100 0011 1000 1100
cin 1
cout 1
sum[3:0] 'b 1100 0110 0100 1100

Schematic:


b[3:0]
[0:3]
a[3:0]
[0:3]
cin
A
B
Cin
fa0
Sum
Cout
fa1
fa2
fa3
[0:3]
sum[3:0]
cout