1. 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.

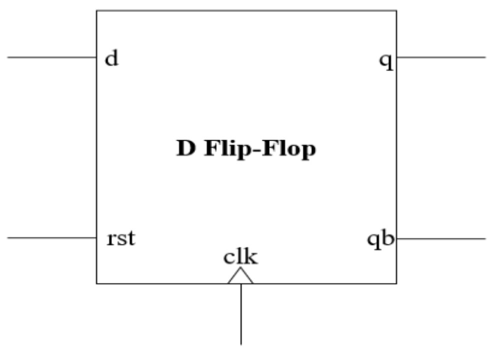
Tools required:

- *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:

Non-GUI Output:

GUI Output:

Area Report

Power Report

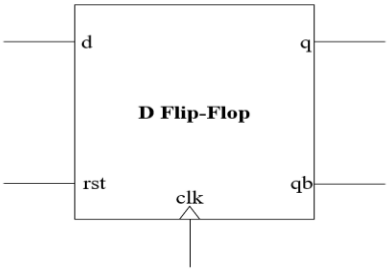
Gates Report:

Timing Report:

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:

Non-GUI Output:

GUI Output:

Area Report

Power Report

Gates Report:

Timing Report:

Schematic:

2. **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.**

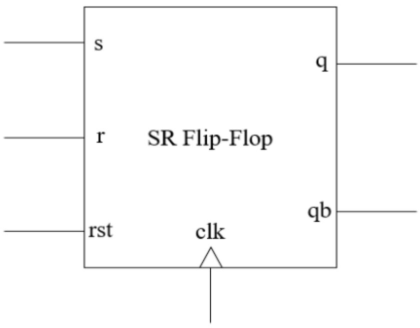
Tools required:

- *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:

Non-GUI Output:

GUI Output:

Area Report

Power Report

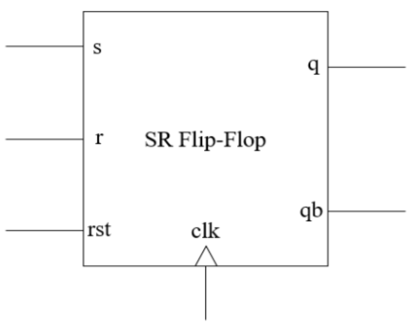
Gates Report:

Timing Report:

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:

Non-GUI Output:

GUI Output:

Area Report

Power Report

Gates Report:

Timing Report:

Schematic:

3. **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.**

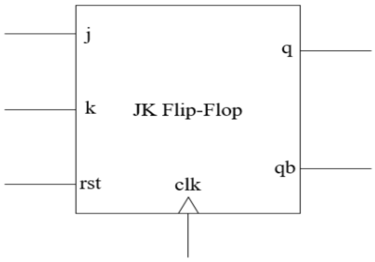
Tools required:

- *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:

Non-GUI Output:

GUI Output:

Area Report

Power Report

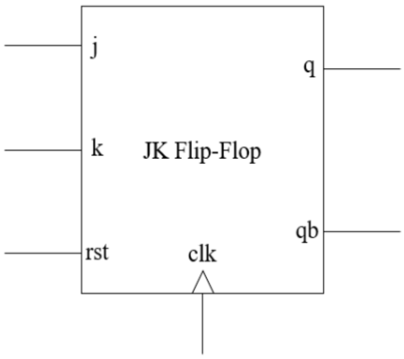
Gates Report:

Timing Report:

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:

Non-GUI Output:

GUI Output:

Area Report

Power Report

Gates Report:

Timing Report:

Schematic:

4. 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.

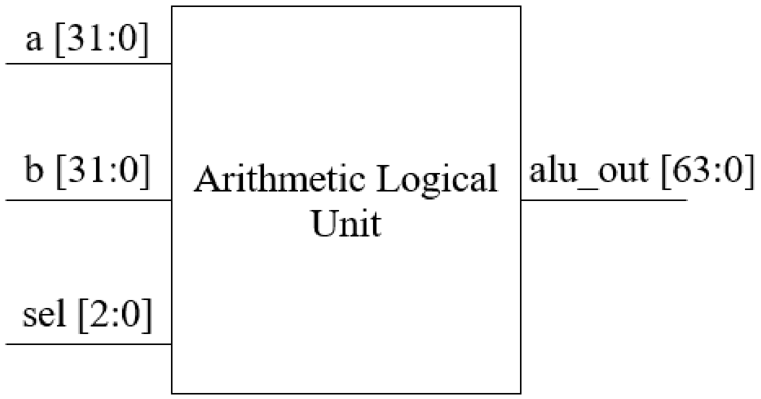
  Compare the synthesize results of ALU modeled using if and case statements.

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	operation	alu_out
32’h FEDCBA98	32’h 89ABCDEF	0	Addition	64’h 00000001_88888887
		1	Subtraction	64’h 00000000_7530ECA9
		2	Multiplication	64’h 890F2A50_AD05EBE8
		3	Division	64’h 00000000_00000001
		4	AND	64’h 00000000_88888888
		5	OR	64’h 00000000_FFFFFFFF
		6	XOR	64’h 00000000_77777777
		7	XNOR	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
```

Creating an SDC File:

- In terminal type "gedit alu_top.sdc" to create an SDC file.

set_input_delay -max 0.2 [get_ports "a"]

set_input_delay -max 0.25 [get_ports "b"] *set_output_delay -max 0.2 [get_ports “alu_out”]*

Performing Synthesize:

The following are commands to perform synthesize

genus -gui

read_lib /home/install/cad/slow.lib

read_hdl alu.v

elaborate

read_sdc alu_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 > alu_netlist.v

write_sdc > alu_top.sdc

report_power

report_gates

report_timing

report_area

report_qor -levels_of_logic -power -exclude_constant_nets

Result:

Non-GUI output:

GUI Output:

Area report:

Power report:

Gates Report:

Timing report:

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:

Non-GUI output:

GUI Output:

Area report:

Power report:

Gates Report:

Timing report:

Schematic:

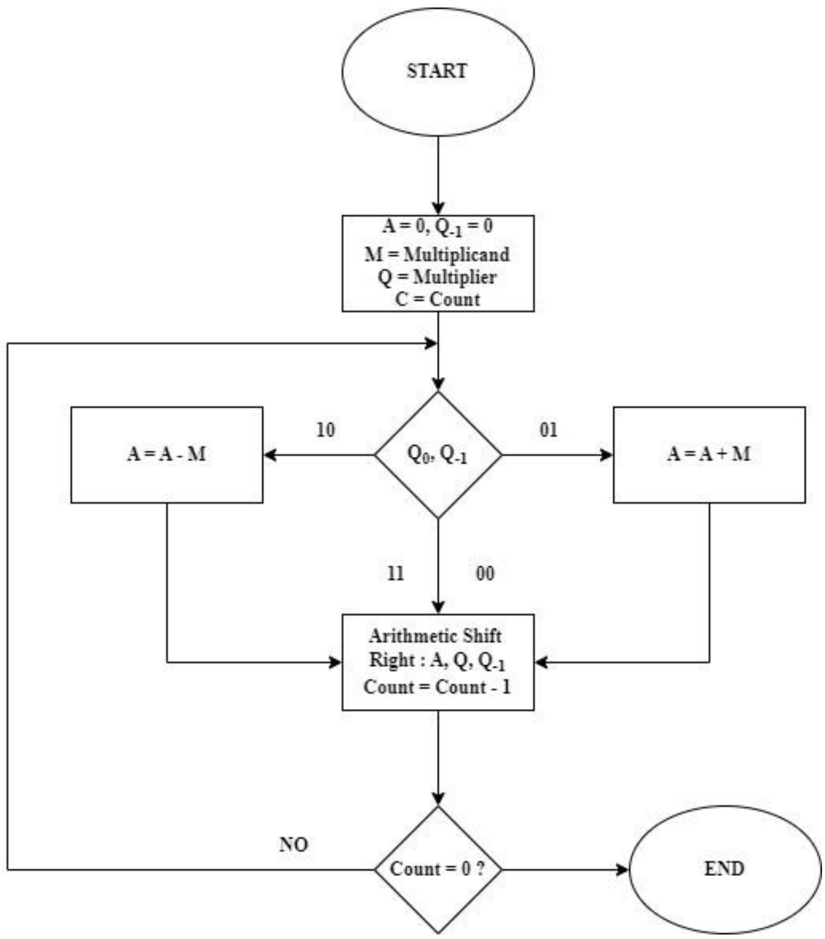
*****Compare the results of 32-bit ALU case statement and 32-bit ALU using if statement*****

5. **Write Verilog code for 4-bit shift and add multiplier and carry out the following:**
   a. **Verify the functionality using test bench**
   b. **Synthesize the design by setting proper constraints and obtain the gate level netlist From the report generated identify Critical path, Maximum delay, Total number of cells, Power requirement and Total area required.**

Tools required:

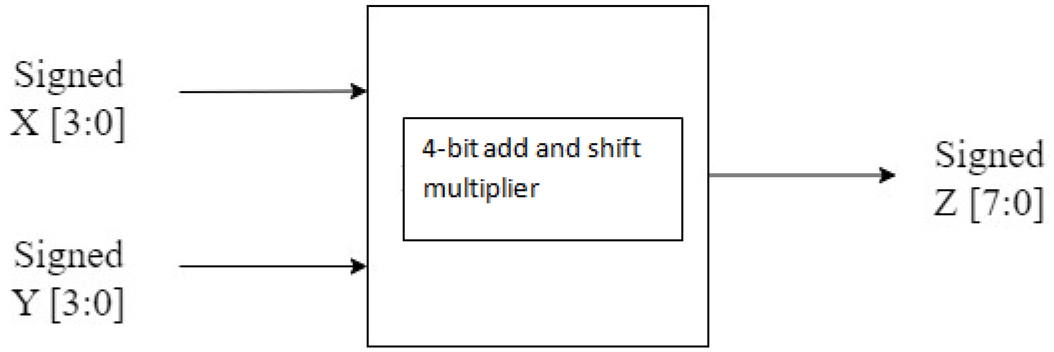
- *Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)*
- *Synthesis: Genus*

4-bit shift and add multiplier *is one which multiplies 2 signed integers in 2's complement. This is depicted in the following figure with a brief description. This approach uses fewer additions and subtractions than more straightforward algorithms.*



The multiplicand and multiplier are placed in the m and Q registers respectively. A 1-bit register is placed logically to the right of the LSB (least significant bit) Q0 of Q register. This is denoted by Q-1. A and Q-1 are initially set to 0. Control logic checks the two bits Q0 and Q-1. If the two bits are same (00 or 11) then all of the bits of A, Q, Q-1 are shifted 1 bit to the right. If they are not the same and if the combination is 10 then the multiplicand is subtracted from A and if the combination is 01 then the multiplicand is added with A. In both the cases results are stored in A, and after the addition or subtraction operation, A, Q, Q-1 are right shifted. The shifting is the arithmetic right shift operation where the left most bit namely, An-1 is not only shifted into An-2 but also remains in An-1. This is to preserve the sign of the number in A and Q. The result of the multiplication will appear in the A and Q.

4-bit shift and add multiplier block diagram:



4-bit shift and add multiplier truth table:

X	Y	Z
0	0	0
-5	7	-35

Verilog code for 4-bit shift and add multiplier:

```
module mul (X, Y, Z);
input signed [3:0] X, Y;
output signed [7:0] Z;
reg signed [7:0] Z;
reg [1:0] temp;
integer i;
reg E1;
reg [3:0] Y1;

always @ (X, Y)
begin
Z = 8'd0;
E1 = 1'd0;
for (i = 0; i < 4; i = i + 1)
begin
temp = {X[i], E1};
Y1 = - Y;

case (temp)
2'd2: Z [7 : 4] = Z [7 : 4] + Y1;
2'd1: Z [7 : 4] = Z [7 : 4] + Y;
default: begin end
endcase

 Z = Z >> 1;
Z[7] = Z[6];
E1 = X[i];
  end
```

```
    if (Y == 4'd8)
        begin
            Z = - Z;
        End
end

endmodule
```

Testbench for 4-bit shift and add multiplier:

```
module mul_tb;
reg signed [3:0] X, Y;
wire signed [7:0] Z;

mul b1 (.X(X), .Y(Y), .Z(Z));

initial
begin

X = 0;
Y = 0;
#10;
X=-5;
Y=7;
#20; $finish;

end

initial
$monitor($time, "X = %d", X," Y = %d",Y, "Z = %d", Z);

endmodule
```

Result:

Non-GUI Output:

GUI Output:

Area Report

Power Report

Gates Report:

Timing Report:

Schematic:

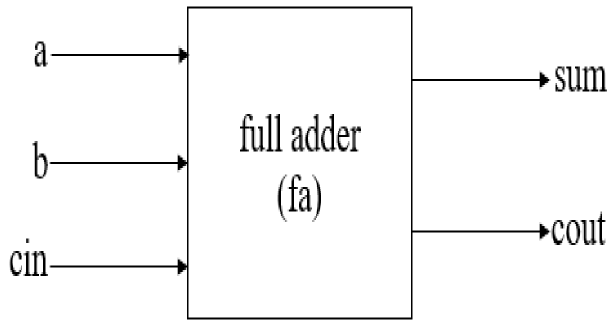
6)Write verilog code for 4-bit adder and verify its functionality using test bench. Synthesize the design by setting proper constraints and obtain netlist. From the report generated identify the critical path, and maximum delay, total number of cells, power requirement and total area required.

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:

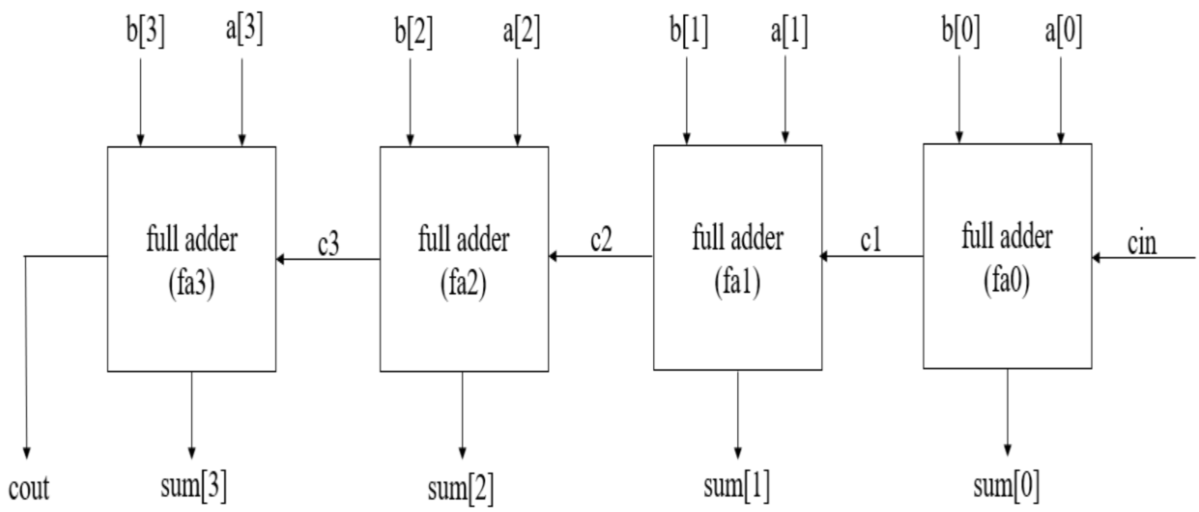


Full-Adder truth table:

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:

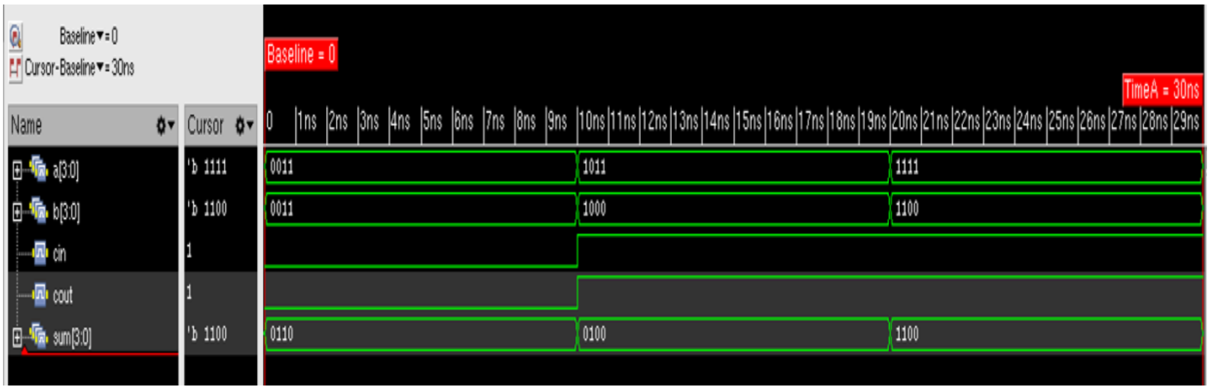
```
module four_bit_adder (a, b, cin, sum, cout);
input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;

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:

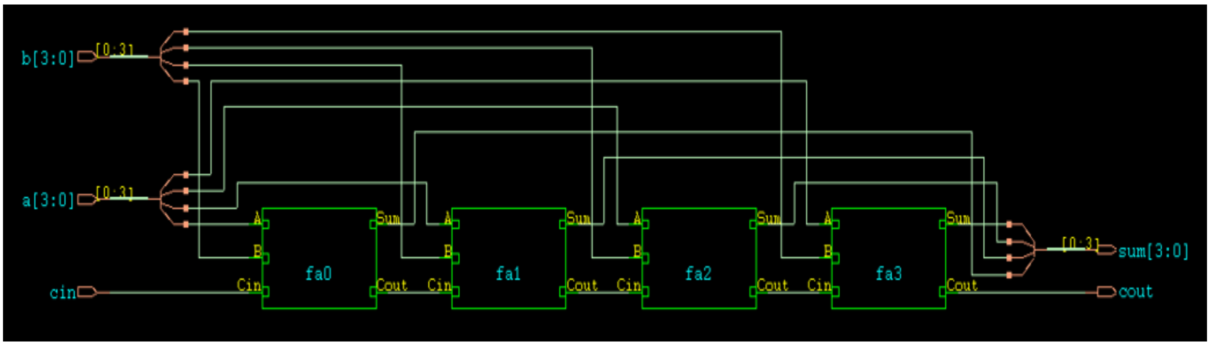
```
module test_adder;
reg [3:0] a, b;
reg cin;
wire [3:0] sum;
wire cout;

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:



Schematic:



Result:

Non-GUI Output:

GUI Output:

Area Report

Power Report

Gates Report:

Timing Report:

Schematic:

7)Four bit Synchronous MOD-N counter with Asynchronous reset
*Write Verilog Code
* Verify functionality using Test-bench

* Synthesize the design targeting suitable library and by setting area and timing constraints
* Tabulate the Area, Power and Delay for the Synthesized netlist Identify Critical path

```
module modn_counter #(
    parameter MODULO = 10,
    parameter WIDTH = $clog2(MODULO)  // Calculate the required bit-width
) (
    input wire clk,
    input wire rst,
    input wire en,
    output reg [WIDTH-1:0] count
);

    always @(posedge clk or posedge rst) begin
        if (rst)
            count <= 0;
        else if (en) begin
            if (count == MODULO - 1)
                count <= 0;
            else
                count <= count + 1;
        end
    end
endmodule
```

```
module tb_modn_counter;
    reg clk, rst, en;
    wire [3:0] count;
    modn_counter #(.MODULO(10)) uut (
```

```
.clk(clk),
.rst(rst),
.en(en),
.count(count)
);
  always #5 clk = ~clk;
   initial begin
      clk = 0; rst = 1; en = 0;
      #10 rst = 0; en = 1;
      #100 $finish;
   end
   initial begin
      $monitor("Time: %t | Count: %d", $time, count);
   end
endmodule
```

Creating an SDC File:

- In terminal type “gedit constraints_top.sdc” to create an SDC file. ***(This file is common for all programs)***
- 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]

Result:

Non-GUI Output:

GUI Output:

Area Report

Power Report

Gates Report:

Timing Report:

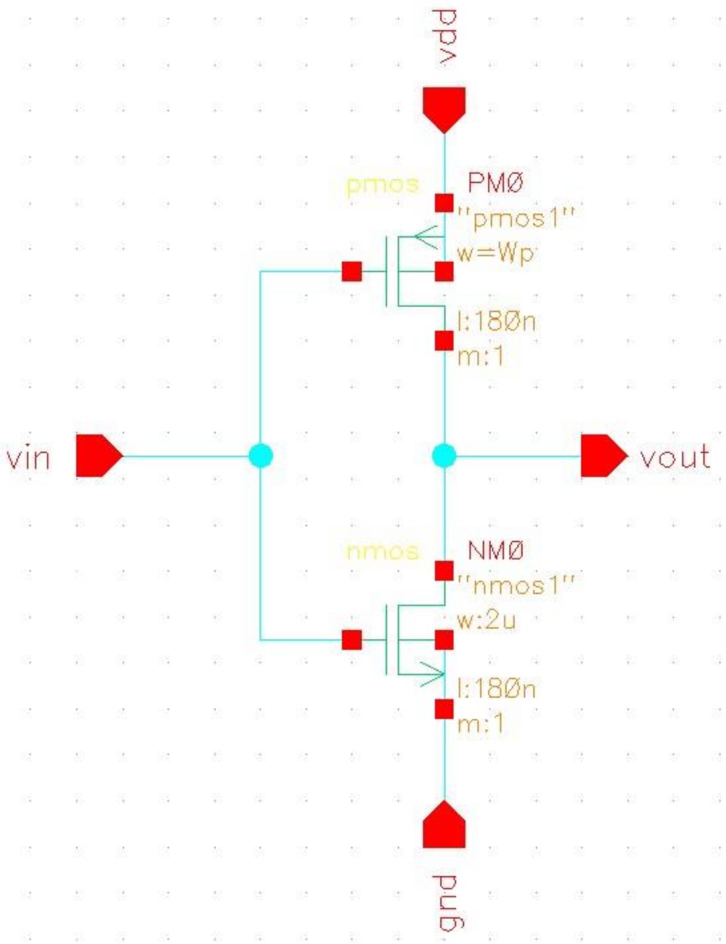
Schematic:

8.a) Capture the schematic of CMOS inverter with load capacitance of 0.1pF and set the widths of inverter with Wn = Wp, Wn = 2Wp, Wn = Wp/2 and length at the selected technology. Carry out the following:

i) **Set the input signal to a pulse with rise time, fall time of 1ns and pulse width of 10ns and time period of 20ns and plot the input voltage and output voltage of designed inverter?**

ii) From the simulation result compute t_{pHL}, t_{pLH} and t_d for all three geometrical settings of width?

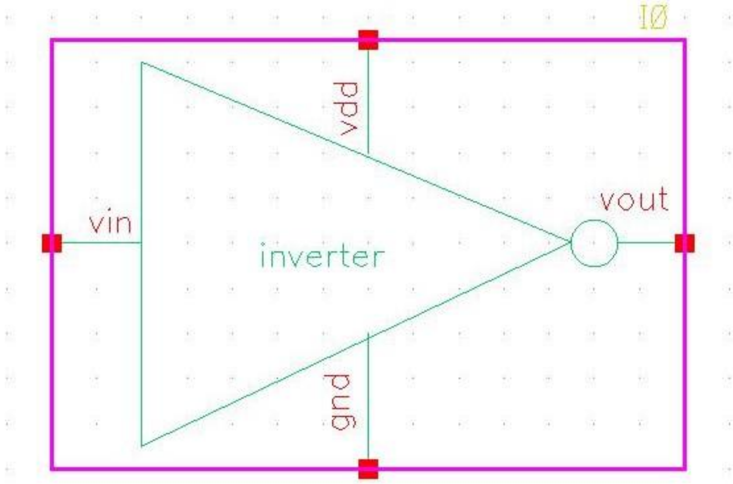
iii) Tabulate the results of delay and find the best geometry for minimum delay for CMOS inverter



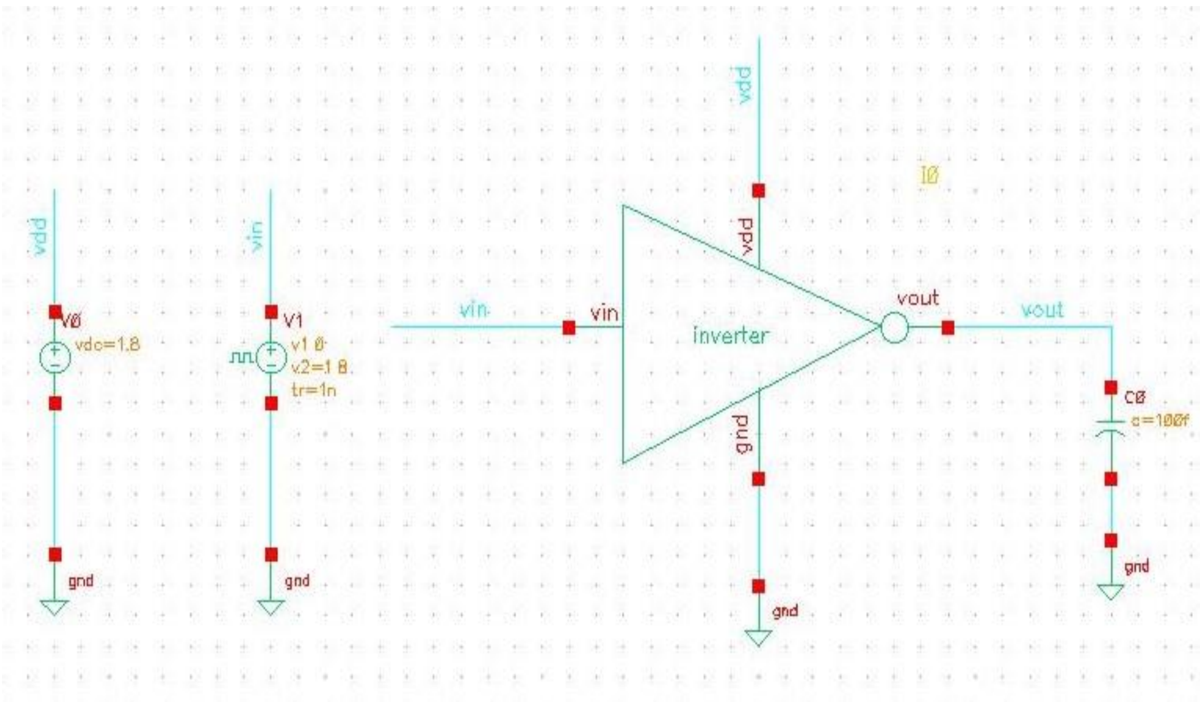
CMOS Inverter schematic

Table of components for building the schematic:

Library Name	Cell Name	Properties
gpdk180	pmos	W = Wp, L = 180n
gpdk180	nmos	W = 2u, L = 180n



CMOS Inverter symbol



CMOS Inverter test schematic

Table of components for building the test schematic:

Library Name	Cell Name	Properties
analogLib	Vpulse	V1 = 0, V2 = 1.8, Period = 20n, Rise time = 1n, Fall time = 1n, Pulse width = 10n
analogLib	Vdc	Vdc = 1.8
analogLib	gnd	

analogLib	cap	0.1pF

Table of values to setup for different analysis:

<table>
<tr><th>Analysis Name</th><th>Settings</th><th>Properties</th></tr>
<tr><td>Transient</td><td>trans</td><td>Stop time = 100n, moderate</td></tr>
<tr><td rowspan="3">DC</td><td><u>DC Analysis</u></td><td>Save DC Operating point</td></tr>
<tr><td><u>Sweep Variable</u>
Component
Parameter</td><td>Component Name = Select input signal component (Vpulse)
Parameter Name = dc</td></tr>
<tr><td><u>Sweep Range</u>
Start – Stop</td><td>Start = 0, Stop 1.8</td></tr>
</table>

Analog Simulation with spectre for inverter:

a) Transient Response

b) DC Response

Tabulated Values of Delays and DC Operating Points :

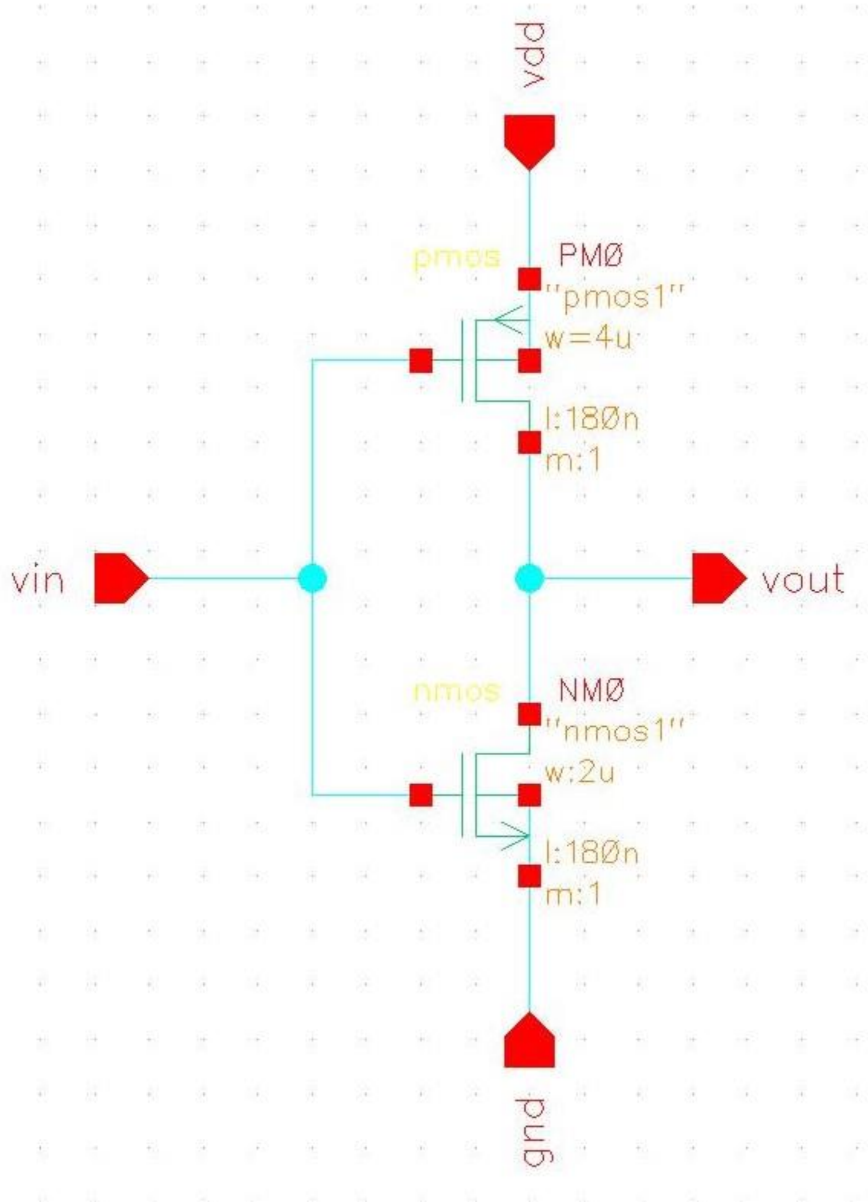
Values of t_{phl}, t_{plh} and t_{pd} for different geometries

Width setting	MOSFET	Width	t_{phl} (ps)	t_{plh} (ps)	t_{pd} (ps)
Wp = Wn	pmos	2u			
	nmos	2u			
Wp = Wn / 2	pmos	1u			
	nmos	2u			
Wp = 2 Wn	pmos	4u			
	nmos	2u			

DC operating point values for different geometries

Width setting	MOSFET	Width	Vin (mV)	Vout (mV)
Wp = Wn	pmos	2u		
	nmos	2u		
Wp = Wn / 2	pmos	1u		
	nmos	2u		
Wp = 2 Wn	pmos	4u		
	nmos	2u		

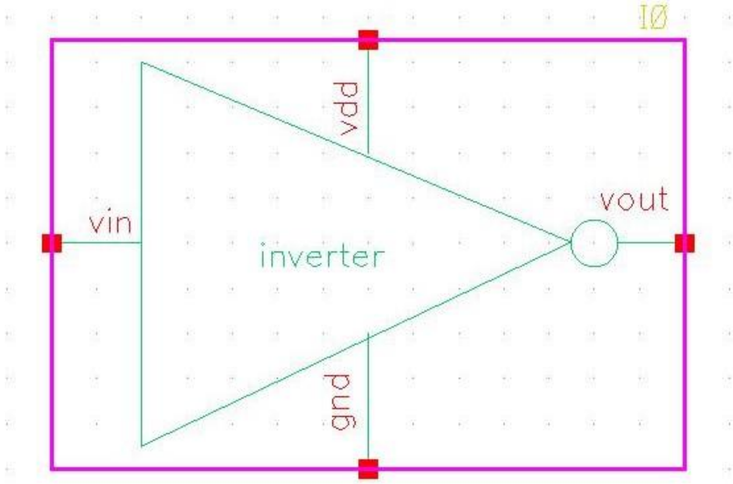
1. **b) Draw layout of inverter with Wp/Wn = 40/20, use optimum layout methods. Verify for DRS and LVS, extract parasitic and perform post layout simulations, compare the results of with pre-layout simulations. Record the observations.**



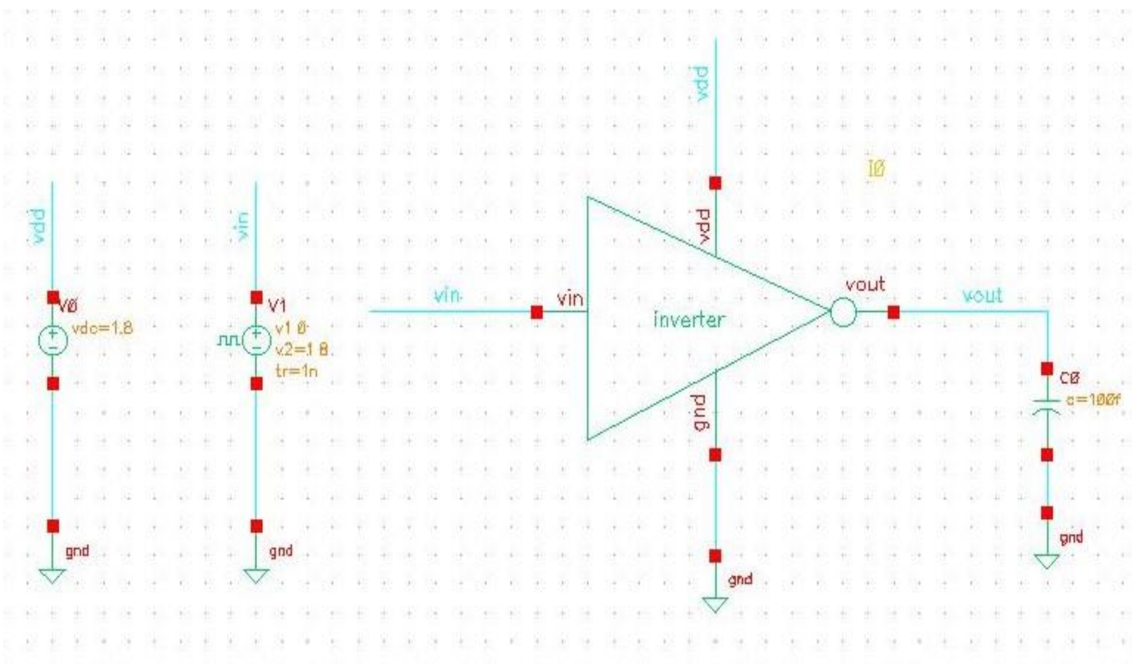
CMOS Inverter schematic

Table of components for building the schematic:

Library Name	Cell Name	Properties
gpdk180	pmos	W = 4u, L = 180n
gpdk180	nmos	W = 2u, L = 180n



CMOS Inverter symbol



CMOS Inverter test schematic

Table of components for building the test schematic:

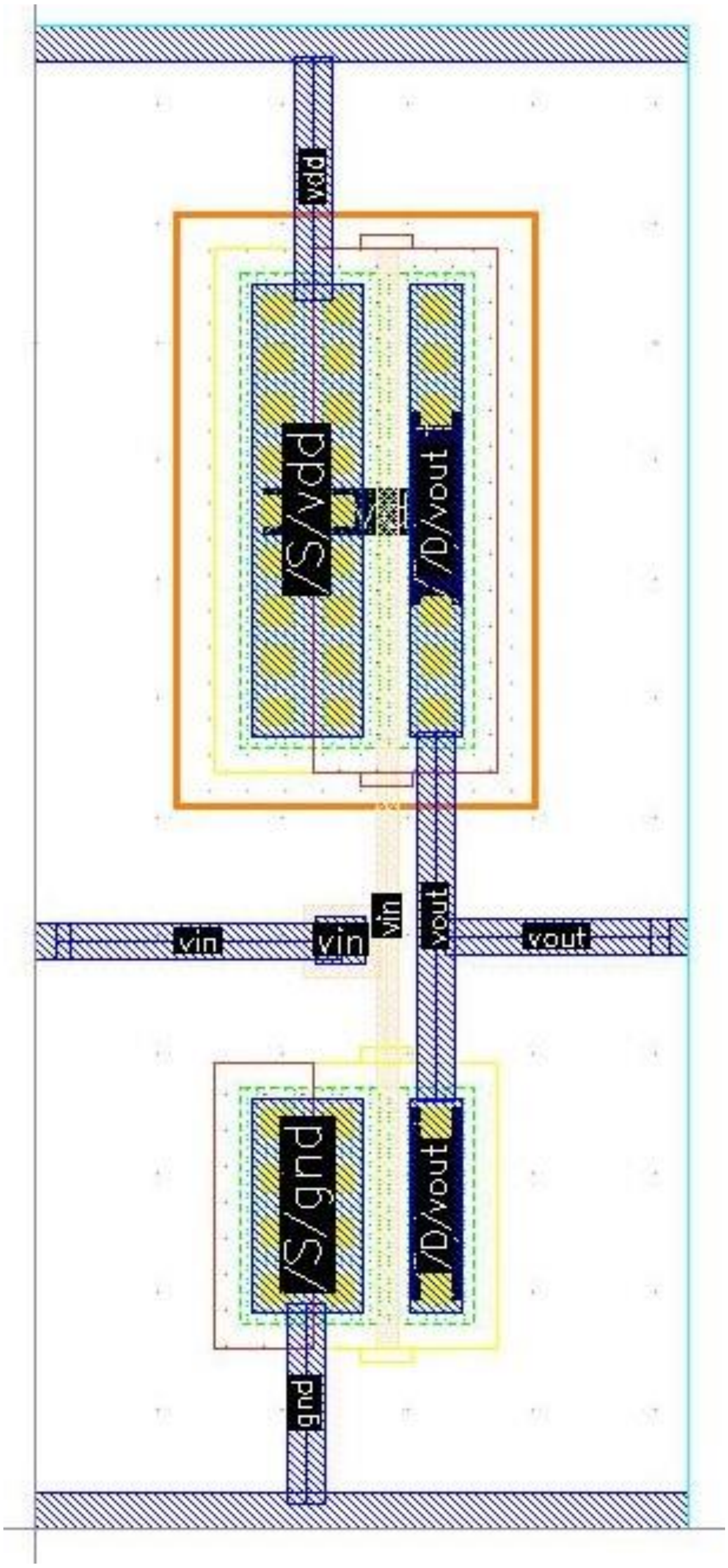
Library Name	Cell Name	Properties
analogLib	Vpulse	V1 = 0, V2 = 1.8, Period = 20n, Rise time = 1n, Fall time = 1n, Pulse width = 10n
analogLib	Vdc	Vdc = 1.8
analogLib	gnd	
analogLib	cap	0.1pF

Analog Simulation with spectre for inverter test schematic:

a) Transient Response

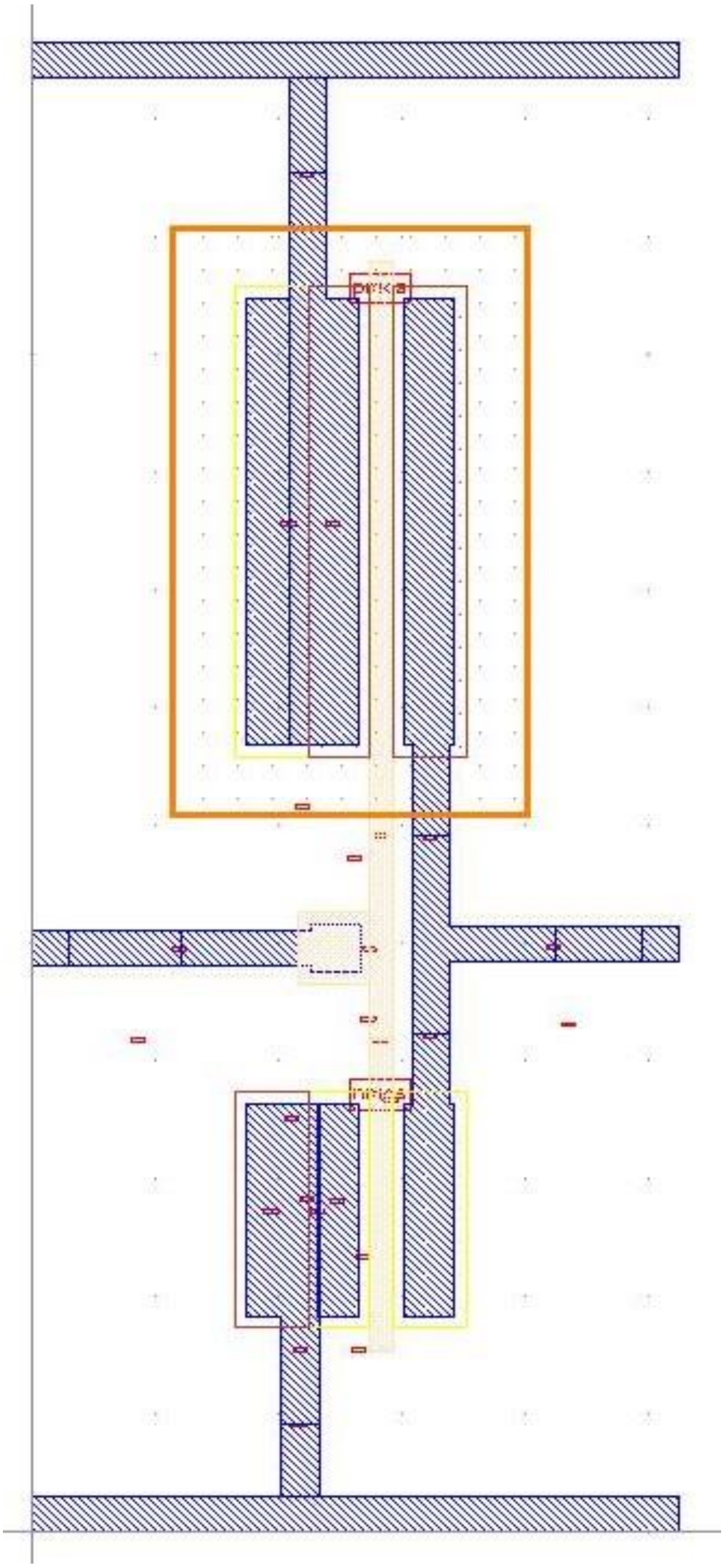
b) DC Response

CMOS Inverter Layout:



CMOS Inverter Layout

CMOS Inverter av_extracted view:



CMOS Inverter av_extracted view

Analog Simulation with spectre for inverter layout:

a) *Transient Response*

b) *DC Response*

Tabulated Values of Delay:

Values of t_{phl}, t_{plh} and t_{pd} for Wp/Wn = 40/20

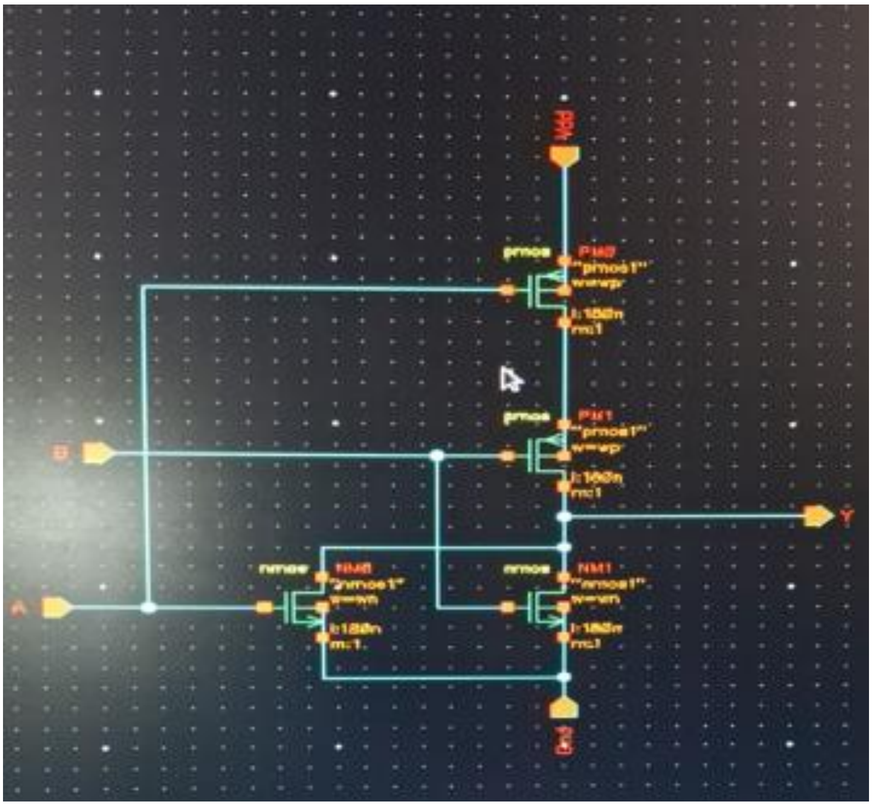
	t_{phl} (ps)	t_{plh} (ps)	t_{pd} (ps)
CMOS Inverter Test Schematic			
CMOS Inverter Layout			

DC operating point values for Wp/Wn = 40/20

	Vin (mV)	Vout (mV)
CMOS Inverter Test Schematic		
CMOS Inverter Layout		

9 a) Capture the schematic of 2-input CMOS NOR gate having similar delay as that of CMOS inverter computed in experiment above. Verify the functionality of

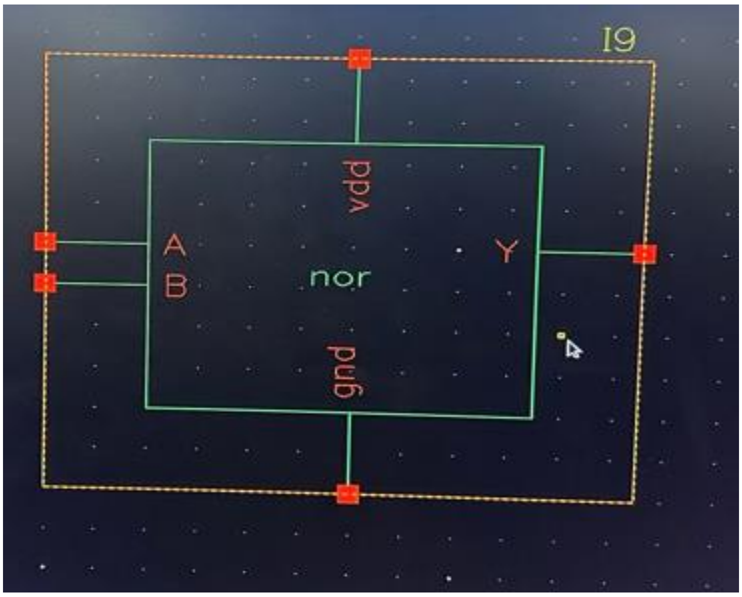
NOR gate and also find out the delay t_d for all four possible combinations of input vectors. Tabulate the result. Increase the drive strength to 2X and 4X and tabulate the result.



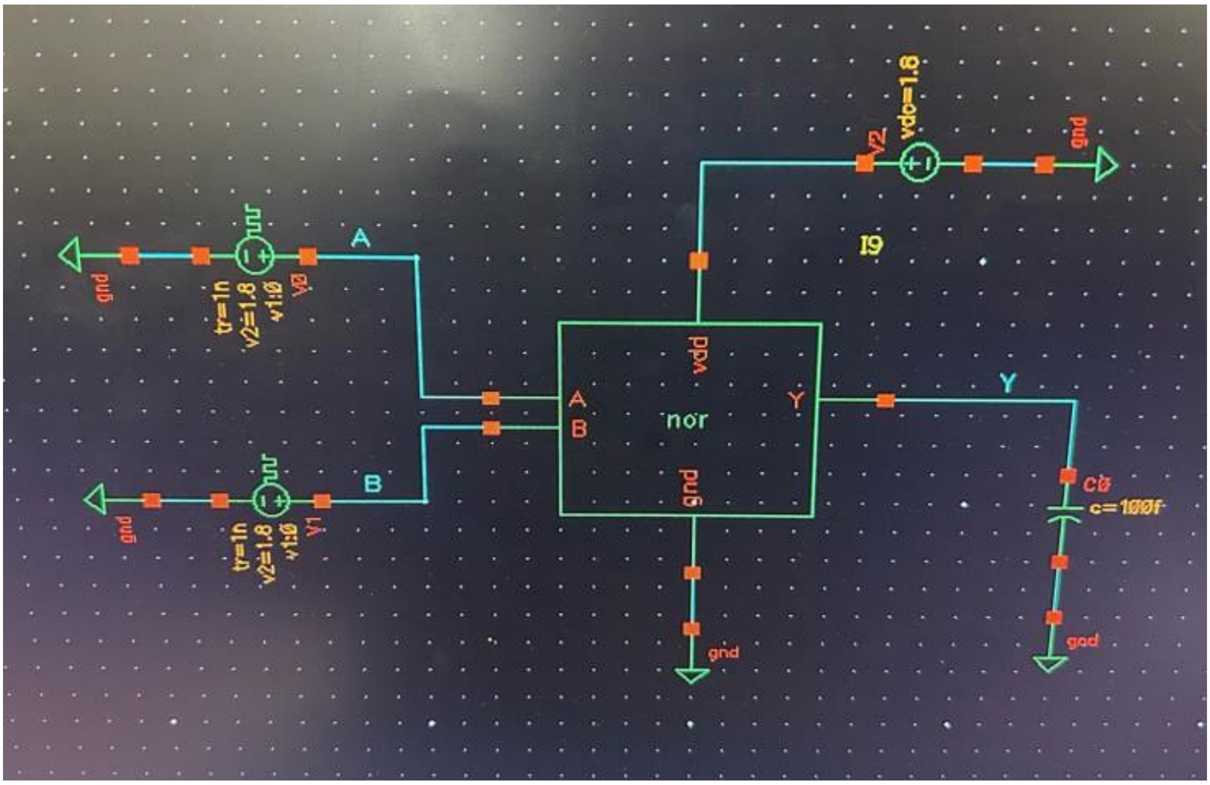
Two Input CMOS NOR Gate schematic

Table of components for building the schematic:

Library Name	Cell Name	Properties
gpdk180	pmos	W = Wp, L = 180n
gpdk180	nmos	W = Wn, L = 180n



Two Input CMOS NOR Gate symbol



Two Input CMOS NOR Gate test schematic

Table of components for building the test schematic:

Library Name	Cell Name	Properties
analogLib	Vpulse	V1 = 0, V2 = 1.8, Period = 30n, Rise time = 1n, Fall time = 1n, Pulse width = 15n
analogLib	Vpulse	V1 = 0, V2 = 1.8, Period = 20n, Rise time = 1n, Fall time = 1n, Pulse width = 10n
analogLib	Vdc	Vdc = 1.8
analogLib	gnd	
analogLib	cap	0.1pF

Table of values to setup for different analysis:

Analysis Name	Settings	Properties

Transient	trans	Stop time = 50n, moderate

Analog Simulation with spectre for Two Input CMOS NOR Gate:

a) Transient Response

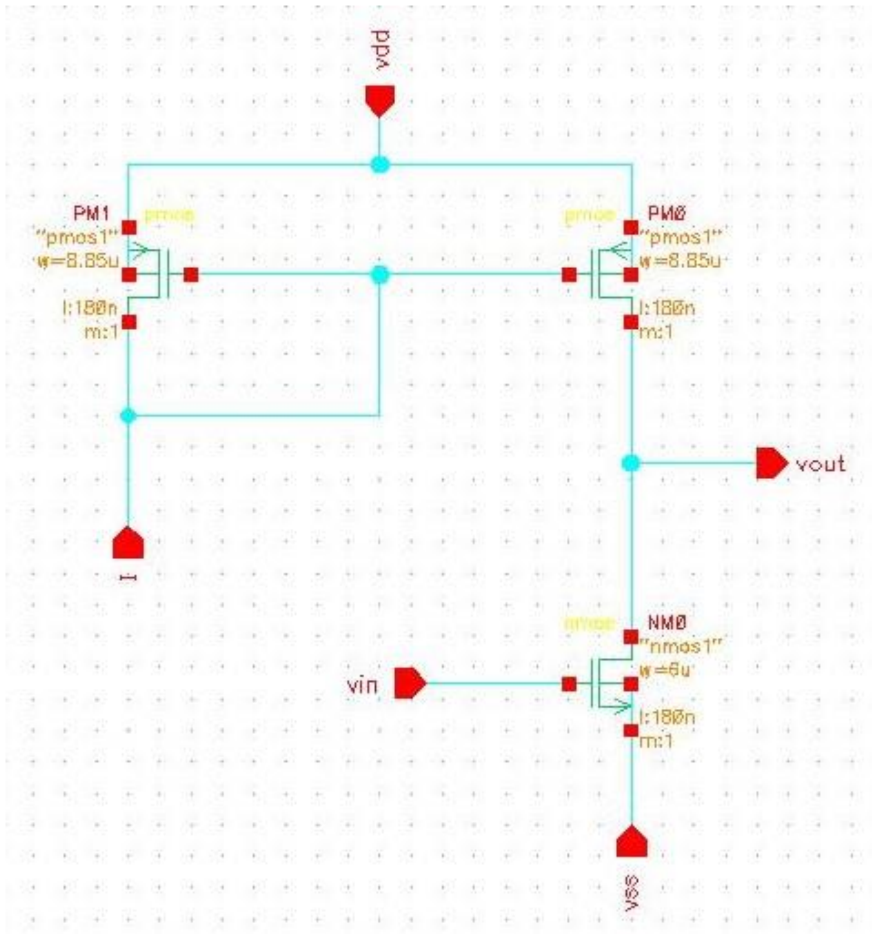
<u>Tabulated Values of Delay:</u>

Values of t_{phl}, t_{plh} and t_{pd} for different geometries

<table>
<tr><th>MOSFET</th><th>Width</th><th>t_{phl} (ps)</th><th>t_{plh} (ps)</th><th>t_{pd} (ps)</th></tr>
<tr><td>pmos</td><td>8u</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>nmos</td><td>1u</td></tr>
<tr><td>pmos</td><td>16u</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>nmos</td><td>2u</td></tr>
<tr><td>pmos</td><td>32u</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>nmos</td><td>4u</td></tr>
</table>

10 a) Capture the schematic of Common Source Amplifier with PMOS Current Mirror load and find its transient response and AC response? Measure the Unity Gain Bandwidth (UGB), amplification factor by varying transistor geometries, study the impact of variation in width to UGB.

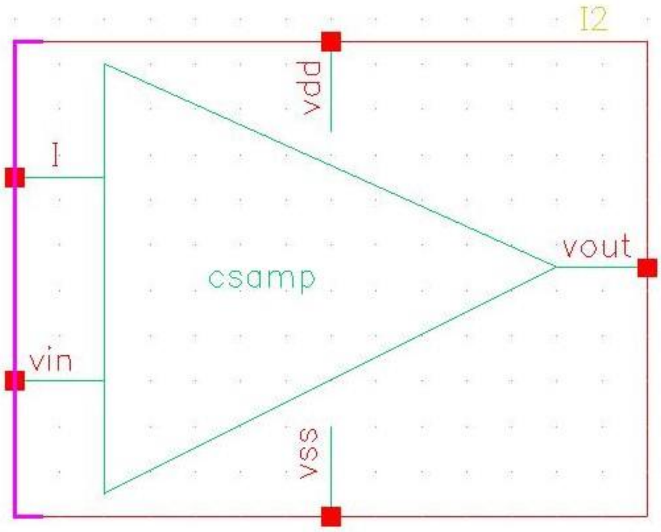
b) Draw layout of common source amplifier, use optimum layout methods. Verify for DRS and LVS, extract parasitic and perform post layout simulations, compare the results of with pre-layout simulations. Record the observations.



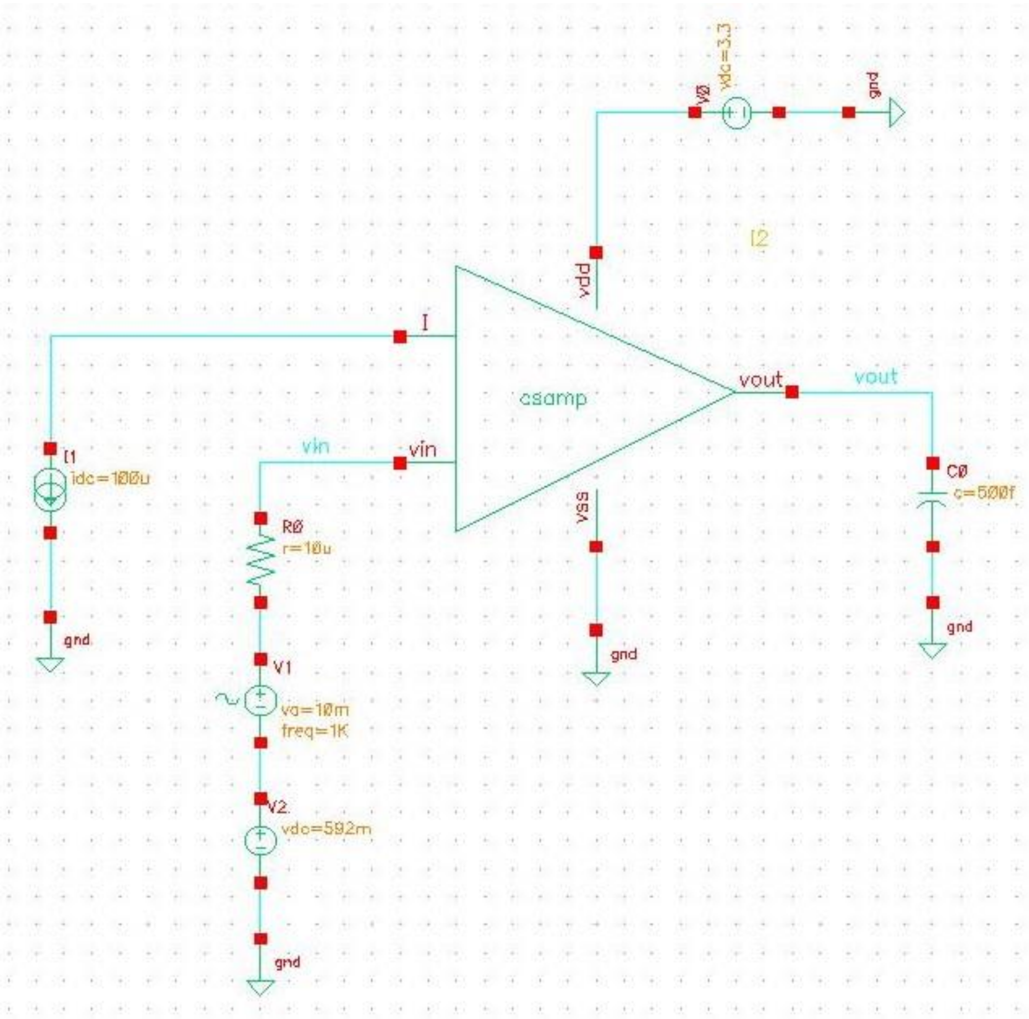
Common Source Amplifier schematic

Table of components for building the schematic:

Library Name	Cell Name	Properties
gpdk180	pmos	W = 8.85u, L = 180n
gpdk180	nmos	W = 6u, L = 180n



Common Source Amplifier symbol



Common Source Amplifier test schematic

Table of components for building the test schematic:

Library Name	Cell Name	Properties
analogLib	Vdc	DC Voltage = 3.3 V (V_{dd})
analogLib	Vsin	AC Magnitude = 1 V, Amplitude = 10u V, Frequency = 1K Hz
analogLib	Vdc	DC Voltage = 592m V
analogLib	res	Resistance = 10u Ohms
analogLib	idc	DC Current = 100u A
analogLib	cap	500f F

Table of values to setup for different analysis:

Analysis Name	Settings	Properties
Transient	trans	Stop time = 5m, moderate
DC	DC Analysis	Save DC Operating point
	Sweep Variable Component Parameter	Component Name = Select input signal component (Vpulse) Parameter Name = dc
	Sweep Range Start – Stop	Sweep Type = Linear Start = -5, Stop = 5, Step size = 10m V
AC	Sweep Range Start – Stop	Sweep Type = Logarithm, Start = 10, Stop = 10G, Points Per Decade = 10

Analog Simulation with spectre for Common Source Amplifier:

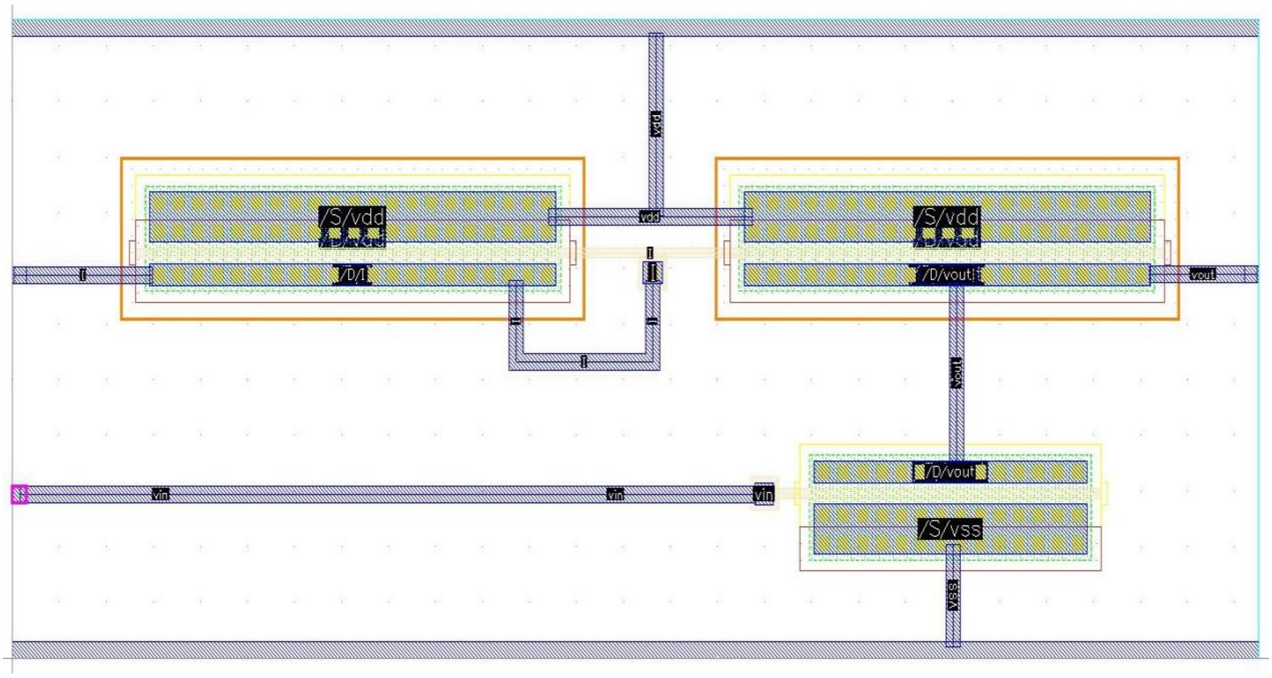
a) *Transient Response*

b) *DC Response*

c) AC Response

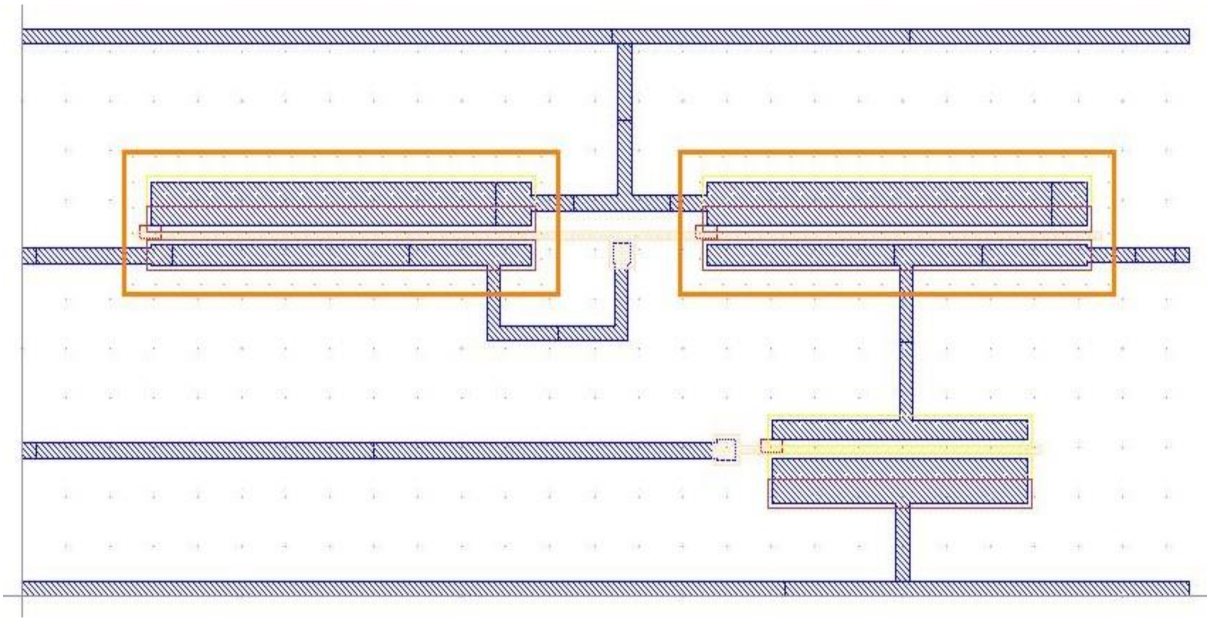
d) AC Magnitude and Phase Response

Common Source Amplifier Layout:



Common Source Amplifier Layout

Common Source Amplifier av_extracted view:



Common Source Amplifier av_extracted view

Analog Simulation with spectre for Common Source Amplifier:

a) *Transient Response*

b) *DC Response*

c) AC Response

d) AC Magnitude and Phase Response

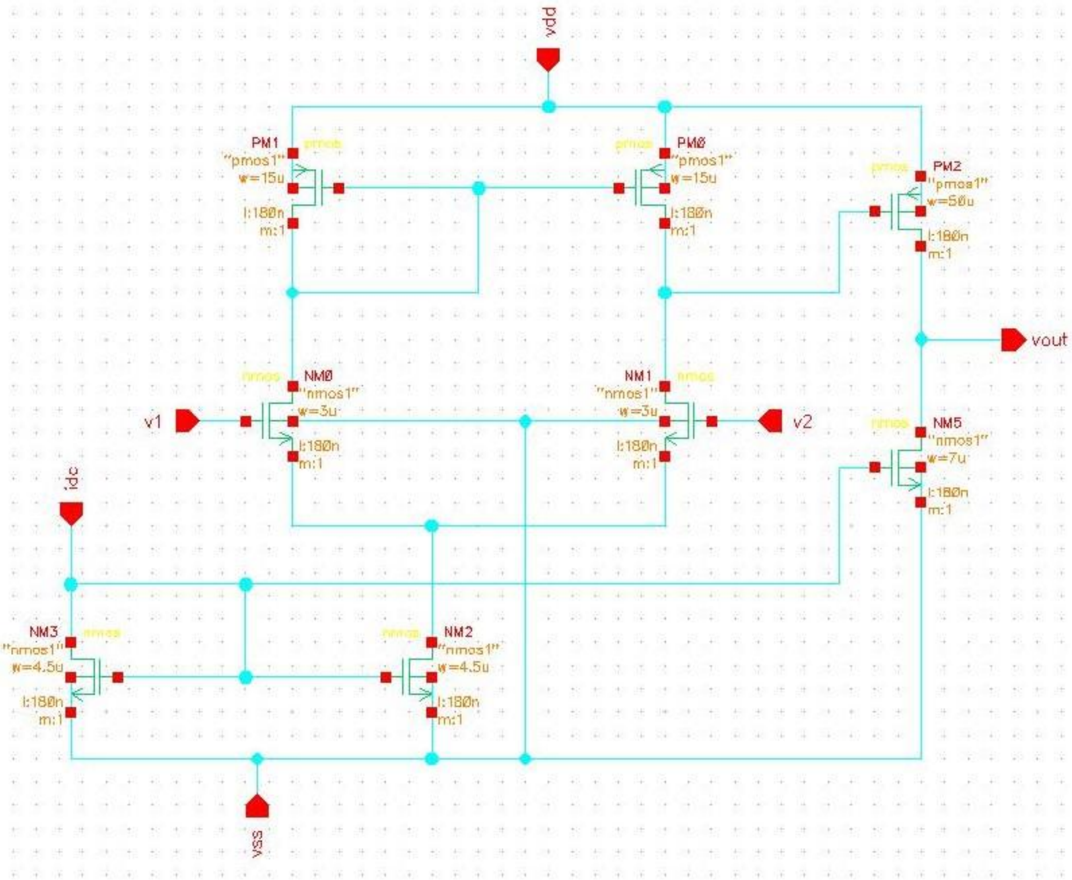
Results:

Common Source Amplifier		
	Gain	**UGB**
Schematic		
Layout		

11 a) Construct the schematic of two-stage operational amplifier and measure the following:

i) **UGB**
ii) **dB bandwidth**
iii) **Gain margin and phase margin with and without coupling capacitance**
iv) **Use the op-amp in the inverting and non-inverting configuration and verify its functionality.**
v) **Study the UGB, 3dB bandwidth, gain and power requirement in op-amp by varying the stage wise transistor geometries and record the observation.**

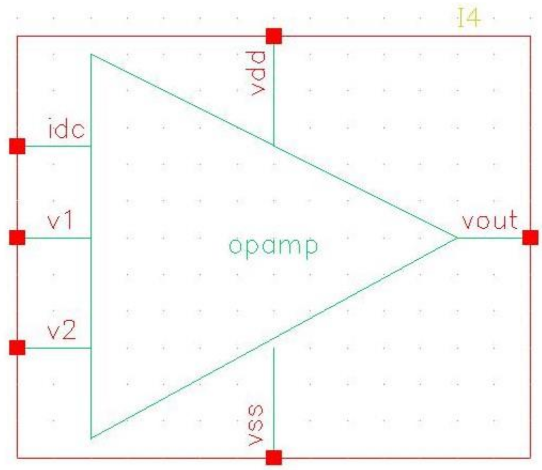
b) Draw layout of two-stage operational amplifier with minimum transistor width set to 300 (in 180/90/45 nm technology), choose appropriate transistor geometries as per the results obtained in 11. a. Use optimum layout methods. Verify for DRS and LVS, extract parasitic and perform post layout simulations, compare the results of with pre-layout simulations. Record the observations.



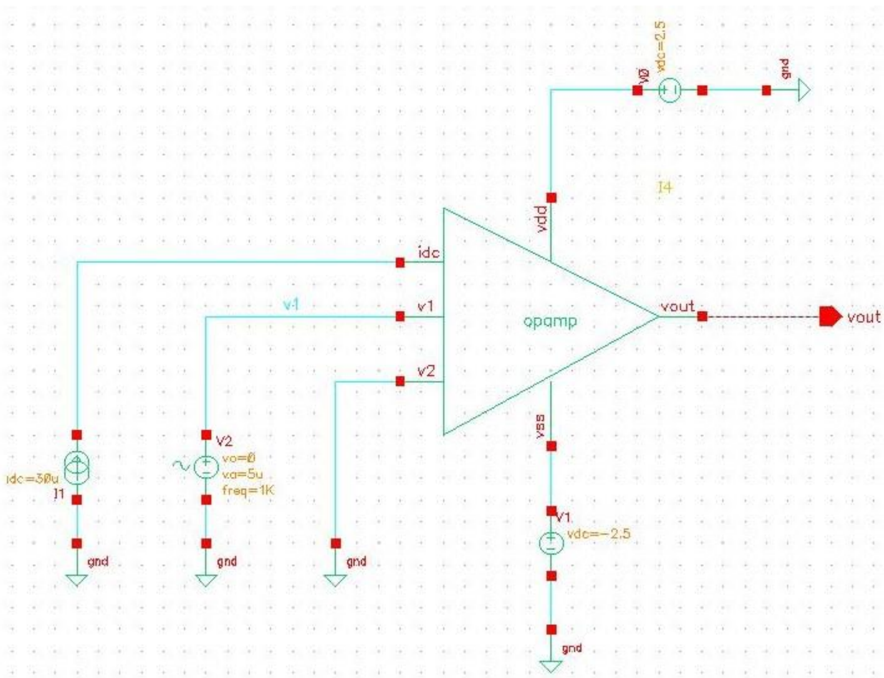
Operational Amplifier schematic

Table of components for building the schematic:

Library Name	Cell Name	Properties
gpdk180	pmos	W = 15u, L = 180n W = 50u, L = 180n
gpdk180	nmos	W = 3u, L = 180n W = 4.5u, L = 180n W = 7u, L = 180n



Operational Amplifier symbol



Operational Amplifier test schematic

Table of components for building the test schematic:

Library Name	Cell Name	Properties
analogLib	Vdc	DC Voltage = 2.5 V (V_{dd}) DC Voltage = -2.5 V (V_{ss})
analogLib	Vsin	AC Magnitude = 1 V, DC Voltage = 0 V, Offset Voltage = 0 V Amplitude = 5u V, Frequency = 1K Hz
analogLib	idc	DC Current = 30u A

Table of values to setup for different analysis:

Analysis Name	Settings	Properties
Transient	trans	Stop time = 5m, moderate
DC	DC Analysis	Save DC Operating point
	Sweep Variable Component Parameter	Component Name = Select input signal component (Vpulse) Parameter Name = dc
	Sweep Range Start – Stop	Start = -5, Stop = 5
AC	Sweep Range Start – Stop	Sweep Type = Automatic, Start = 100, Stop = 10G,

Analog Simulation with spectre for Operational Amplifier:

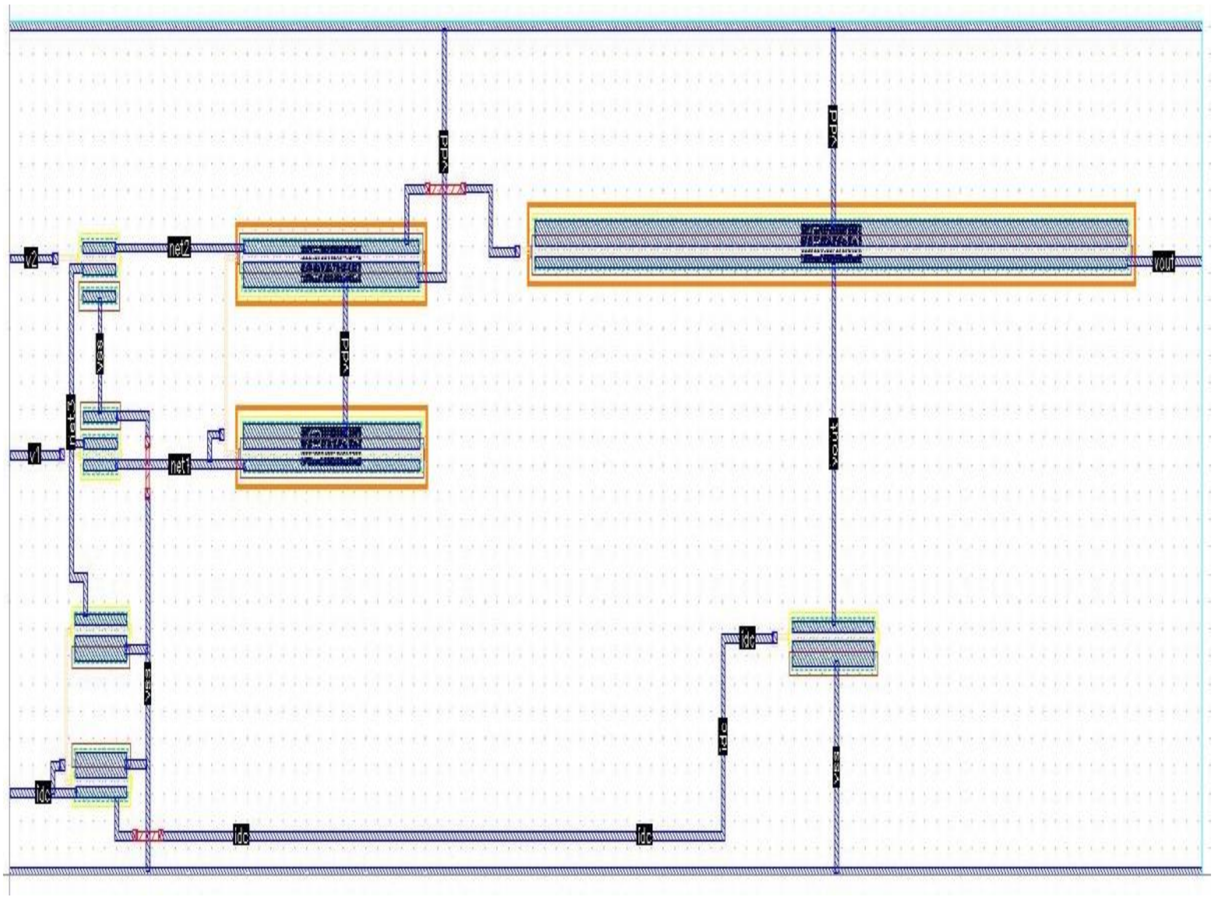
a) Transient Response

b) DC Response

c) *AC Response*

d) *AC Magnitude and Phase Response*

Operational Amplifier Layout:



Operational Amplifier Layout

Analog Simulation with spectre for Operational Amplifier:

a) *Transient Response*

b) *DC Response*

c) *AC Response*

d) *AC Magnitude and Phase Response*

Results:

Two-stage Operational Amplifier

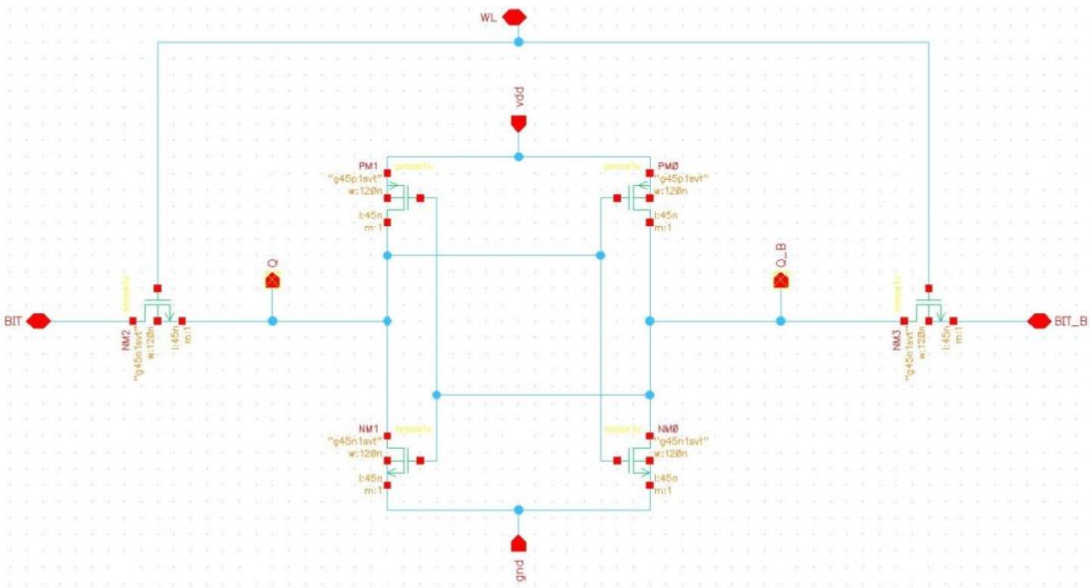
	Gain	UGB
Schematic		
Layout		

DEMONSTRATION EXPERIMENTS (For CIE)

12 Design and characterize 6T binary SRAM cell and measure the following:

i) Read Time, Write Time, SNM, Power

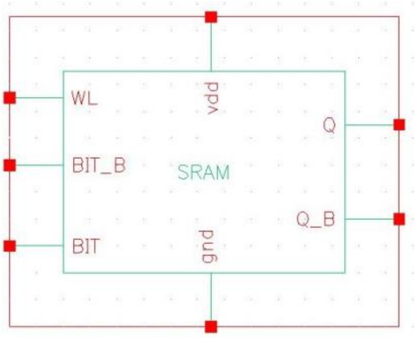
ii) Draw Layout of 6T SRAM, use optimum layout methods. Verify for DRC & LVS, extract parasitic and perform post layout simulations, compare the results with pre-layout simulations. Record the observations.



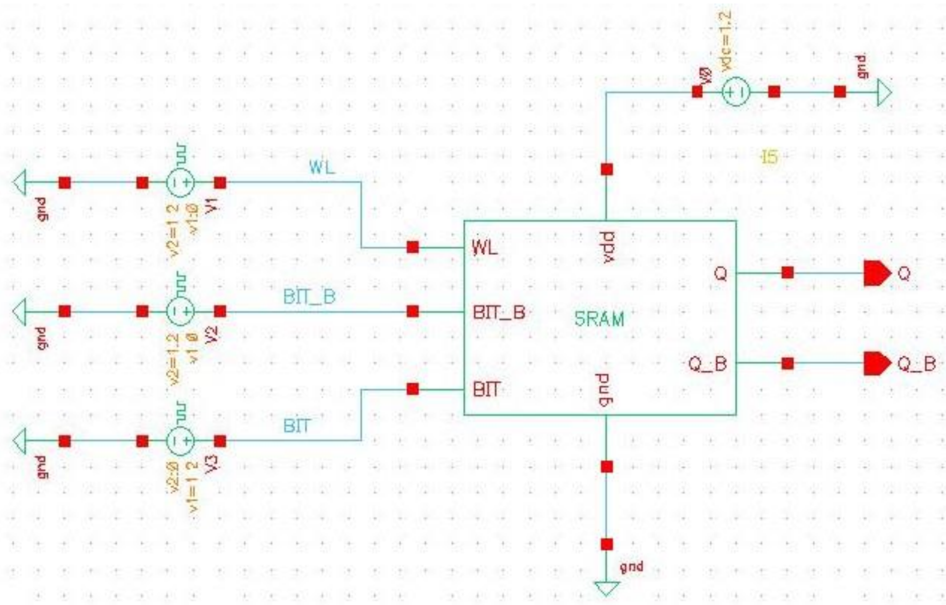
6T binary SRAM cell schematic

Table of components for building the schematic:

Library Name	Cell Name	Properties
gpdk45	pmos	W = 120n, L = 45n
gpdk45	nmos	W = 120n, L = 45n



6T binary SRAM cell symbol



6T binary SRAM cell test schematic

Table of components for building the test schematic:

Library Name	Cell Name	Properties
analogLib	Vdc	DC Voltage = 1.2 V (V_{dd})
analogLib	Vpulse	DC Voltage = 1.2 V, V1 = 0 V, V2 = 1.8, Period = 2n
analogLib	Vpulse	DC Voltage = 1.2 V, V1 = 1.2 V, V2 = 0, Period = 2n
analogLib	Vpulse	DC Voltage = 1.2 V, V1 = 0 V, V2 = 1.2, Period = 4n
analogLib	gnd	

Table of values to setup for different analysis:

Analysis Name	Settings	Properties
Transient	trans	Stop time = 8n, moderate
DC	DC Analysis	Save DC Operating point
	Sweep Variable Component Parameter	Component Name = Select input signal component (Vpulse) Parameter Name = dc
	Sweep Range Start – Stop	Start = 0, Stop = 1.2

Analog Simulation with spectre for 6T binary SRAM cell:

a) Transient Response

b) DC Response

c) SNM

d) Power

Results:

6T binary SRAM cell

Read Time	Write Time	SNM	Power

13.Write verilog code for UART and carry out the following:

a. Verify the functionality using testbench.

b. Synthesize the design by setting area and timing constraints.

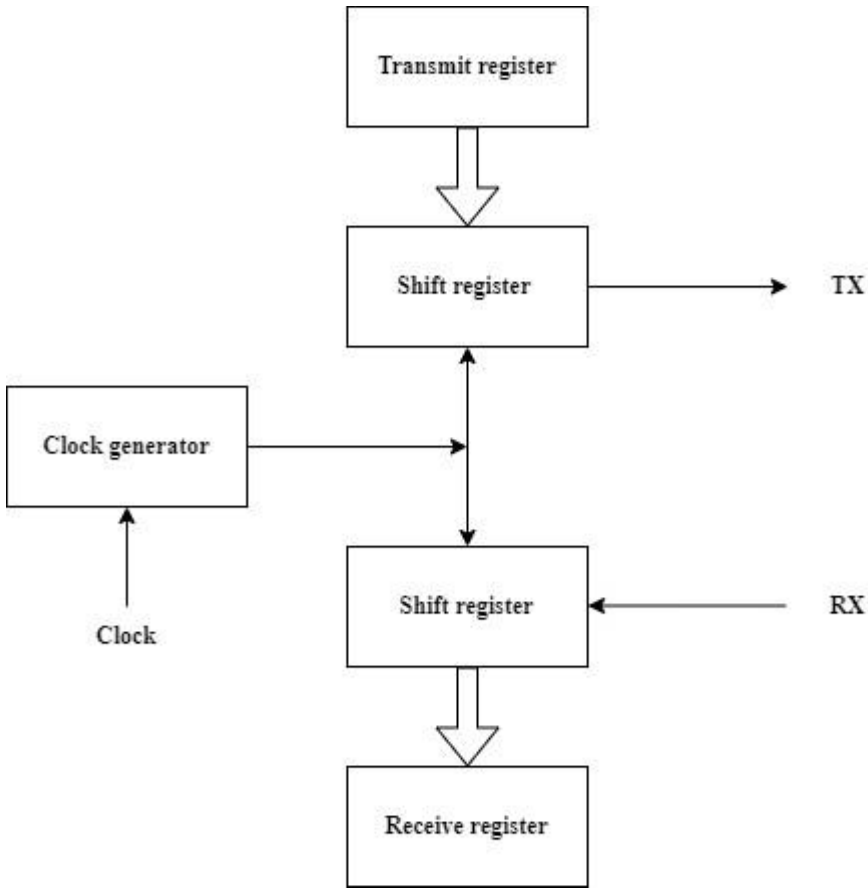
c. **Tabulate the area, power and delay for the synthesized netlist, identified the critical path.**

Tools required:

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

UART: *A universal asynchronous receiver-transmitter (UART) is a peripheral device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel.*

UART Block diagram:



Verilog code for UART:

```
module uart_transmitter (input clk, input reset, input [7:0] data_in, output reg tx_out, output reg tx_busy);

localparam IDLE = 2'b00;
localparam START_BIT = 2'b01;
localparam DATA_BITS = 2'b10;
localparam STOP_BIT = 2'b11;

reg state_t;

reg [2:0] state;
reg [3:0] bit_counter;
reg [7:0] transmit_data_reg;

parameter BAUD_RATE = 9600;
parameter CLOCK_FREQ = 50_000_000;

localparam BIT_CYCLES = CLOCK_FREQ / BAUD_RATE;

always @ (posedge clk or posedge reset) begin
if (reset) begin
state <= IDLE;
tx_busy <= 0;
bit_counter <= 0;
transmit_data_reg <= 8'b0;
end
else
begin
case (state)
IDLE: begin
tx_out <= 1;
if (tx_busy) begin
state <= START_BIT;
```

```
bit_counter <= 0;
end
end

START_BIT: begin
tx_out <= 0;
state <= DATA_BITS;
end

DATA_BITS: begin
tx_out <= transmit_data_reg[bit_counter];
bit_counter <= bit_counter + 1;
if (bit_counter == 7) begin
state <= STOP_BIT;
end
end
STOP_BIT: begin
tx_out <= 1;
state <= IDLE;
tx_busy <= 0;
end
endcase
end
end

always @ (posedge clk) begin
if (reset)
transmit_data_reg <= 8'b0;
else if (state == IDLE && !tx_busy)
transmit_data_reg <= data_in;
end

always @ (posedge clk) begin
```

```
if (!reset && !tx_busy && state == IDLE)
tx_busy <= 1;
end
endmodule

module uart_receiver (input clk, input reset, input rx_in, output reg [7:0] data_out,
output reg rx_ready);

localparam IDLE = 2'b00;
localparam START_BIT = 2'b01;
localparam DATA_BITS = 2'b10;
localparam STOP_BIT = 2'b11;

reg state_t;

reg [2:0] state;
reg [3:0] bit_counter;
reg [7:0] receive_data_reg;

parameter BAUD_RATE = 9600;
parameter CLOCK_FREQ = 50_000_000;
localparam BIT_CYCLES = CLOCK_FREQ / BAUD_RATE;
always @ (posedge clk or posedge reset) begin
if (reset) begin
state <= IDLE;
rx_ready <= 0;
bit_counter <= 0;
receive_data_reg <= 8'b0;
end
else
begin
case (state)
IDLE: begin
```

```
if (!rx_in) begin
state <= START_BIT;
bit_counter <= 0;
end
end

START_BIT: begin
state <= DATA_BITS;
end

DATA_BITS: begin
receive_data_reg[bit_counter] <= rx_in;
bit_counter <= bit_counter + 1;
if (bit_counter == 7) begin
state <= STOP_BIT;
end
end

STOP_BIT: begin
state <= IDLE;
rx_ready <= 1;
end
endcase
end
end

always @ (posedge clk) begin
if (reset)
data_out <= 8'b0;
else if (rx_ready)
data_out <= receive_data_reg;
end
endmodule
```

Test bench for UART:

```
module uart_tb;

parameter BAUD_RATE = 9600;
parameter CLOCK_FREQ = 50_000_000;

reg clk;
reg reset;
reg [7:0] data_in;
reg rx_in;

wire tx_out;
wire tx_busy;
wire [7:0] data_out;
wire rx_ready;

uart_transmitter uart_tx (.clk(clk), .reset(reset), .data_in(data_in), .tx_out(tx_out),
.tx_busy(tx_busy));

uart_receiver uart_rx (.clk(clk), .reset(reset), .rx_in(rx_in), .data_out(data_out),
.rx_ready(rx_ready));

always #5 clk = ~clk;

initial
begin
clk = 0;
reset = 1;
data_in = 8'b11011011;
#10 reset = 0;
#10 BAUD_RATE * 10;
```

```
#10 rx_in = 0;
#10 rx_in = 1;
#10 rx_in = 0;
#10 rx_in = 1;
#100 $finish;
end
endmodule
```

Result:

Non-GUI output:

GUI output:

Area report:

Power report:

Timing report:

Schematic: