VLSI LAB MANUAL

Semester: VII **Subject code: 15ECL77**

LIST OF EXPERIMENTS

Course outcomes:

On the completion of this laboratory course, the students will be able to:

- Write test bench to simulate various digital circuits.
- Interpret concepts of DC Analysis, AC Analysis and Transient Analysis in analog circuits.
- Design and simulate basic CMOS circuits like inverter, common source amplifier and differential amplifiers.
- Use basic amplifiers and further design higher level circuits like operational amplifier and analog/digital converters to meet desired parameters.
- Use transistors to design gates and further using gates realize shift registers and adders to meet desired parameters.

PART A: ASIC – Digital Design

PROCEDURE FOR CREATING DIGITAL SIMULATION USING VERILOG AND CADENCE DIGITAL TOOL

Open Terminal Window and use the following commands

Procedure for Simulation:

1. Change directory to Cadence_Digital_labs/Workarea/Inverter using commands:

```
>csh
>source/cad/cshrc
```

2. Type the programs design code and testbench code using editor in workarea folder and save

Create the design file (Verilog module) using editor gedit
>gedit filename.v
The file name can be changed with respect to the experiments.

Create the testbench file (Verilog module) using editor gedit
>gedit tb_filename.v
The testbench file name can be changed with respect to the design file.

3. Compile the source Descriptions: (i) Compile the Inverter description with the -messages option:
>ncvlog –mess (name of the file) (name of the testbench file)

Example: >ncvlog -mess inv.v inv_tb

Note: Check out for error and warnings. If any, then go back to text editor and edit and the compile

4. Elaborate the top level Design using the command:

```
>ncelab   -mess  -access +rwc     (name of the testbench file without file type)
```

Example: >ncelab -mess -access+rwc inv_tb //Do not include file type for testbench file

5. Simulate the Top-Level Design in Non GUI mode
>ncsim -mess -gui (name of the testbench file)

Now a console and Design Browser windows of Simvision are opened and click on the waveform button in the toolbar to send the selected objects to waveform window.

Waveform Window opens and Press run to run the simulation for a time period specified in the time field.

Procedure for Synthesis using Cadence Tool

The Verilog code, technology library and the constraint files are input to the logical synthesis. The tool will generate the gate level netlist and gives the output as: gatelevel schematic and the report of area, power and timing.

6. Create a file named Contraints_sdc.sdc

The timing constraints are defined in this file. Example of one such file is as shown –

```
create_clock -name clk -period 10 -waveform {0 5} [get_ports "clk"]

set_clock_transition -rise 0.1 [get_clocks "clk"]

set_clock_transition -fall 0.1 [get_clocks "clk"]

set_clock_uncertainty 1.0 [get_ports "clk"]

set_input_delay -max 1.0 [get_ports "A"] -clock [get_clocks "clk"]

set_input_delay -max 1.0 [get_ports "B"] -clock [get_clocks "clk"]

set_output_delay -max 1.0 [get_ports "sum"] -clock [get_clocks "clk"]
```

There are three different parts in the constraint file:

a. Clock definition and clock constraints –

Clock definition

```
create_clock -name clk -period 10 -waveform {0 5} [get_ports "clock"]
```

Clock rise time

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

Clock fall time

set_clock_transition -fall 0.1 [get_clocks " clk "]

Uncertainties of Clock

set_clock_uncertainty 1.0 [get_ports " clk "]

b. Input port timing constraints –

set_input_delay -max 1.0 [get_ports " A"] -clock [get_clocks " clk "]

Input port delay

set_input_delay -max 1.0 [get_ports " B"] -clock [get_clocks " clk "]

Input port delay

c. Output port timing constraints –

Output port delay

set_output_delay -max 1.0 [get_ports " sum "] -clock [get_clocks " clk "]

The port names that are used in the constraint file (bolded) must match with the names that are used in the Verilog program of the main design module. The constraints are defined for all the ports in the design.

7. Synthesis the top level design

The verilog file to be synthesized must be copied into this directory.

Copy rc folder that created for simulation in desktop into user directory.

Open the rc_script file in editor and change the file name to:

```
read_hdl  {filename.v}      // filename.v is the Design code file
          {inv.v}           // for example inv.v as filename.v
```

```
>rc  -f  -gui  rc_script.tcl
```

Script file contains the Verilog RTL code, standard library file for a particular technology,

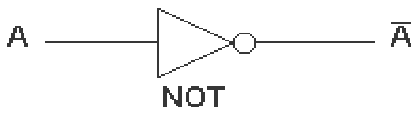
1. Inverter

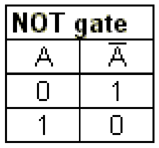
AIM: To write verilog code for an inverter circuit and its testbench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:

The NOT gate or an inverter is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.





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

PROGRAM:

/*Inverter program Verilog code-file name inv.v */

```
module inv(i1,o1);
input i1;
output o1;
assign o1 = ~i1;
endmodule
```

/*Inverter program Verilog code for testbench -file name tb_inv.v*/

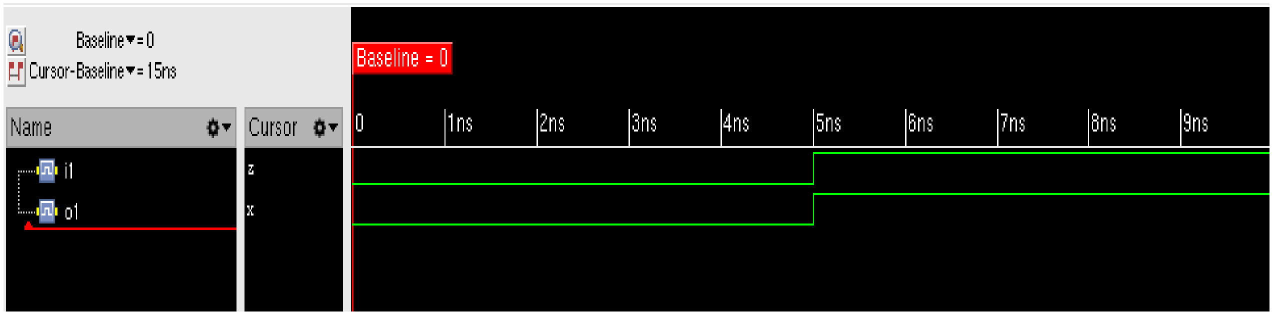
```
module tb_inv;
reg  i1;
wire  o1;
inv  dut_inv(i1,o1);
initial begin
i1 = 1'b0;
#5 i1 = 1'b1;
#5 i1 = 1'bX;
#5 i1 = 1'bZ;
end
endmodule
```

Constraints file for Synthesis:

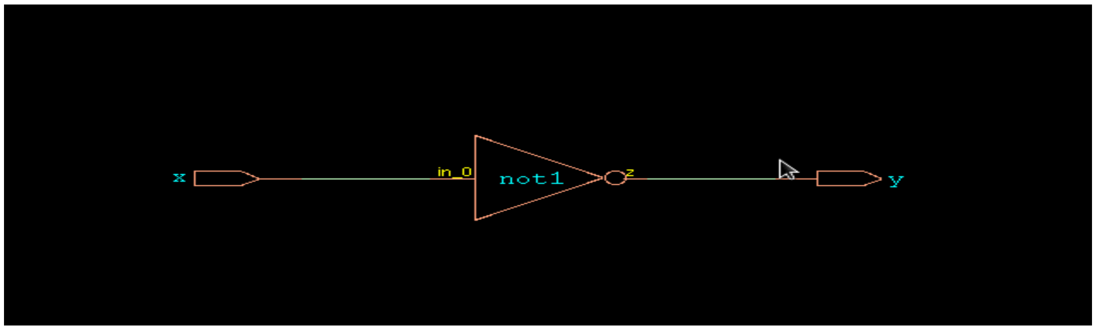
```
set_input_delay -max 1.0 [get_ports "i1"]
set_output_delay -max 1.0 [get_ports "o1"]
```

OUTPUT:

Simulation:



Synthesis



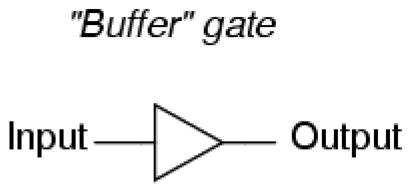
RESULT: Verilog code for the inverter circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

2. A Buffer

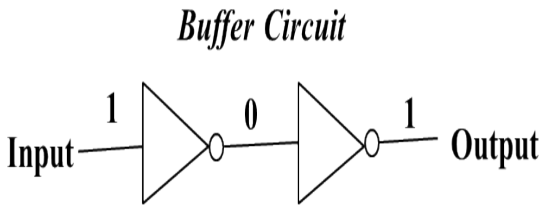
AIM: To write Verilog code for an buffer circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY: A special logic gate called a buffer is manufactured to perform the same function as two inverters. Its symbol is simply a triangle, with no inverting "bubble" on the output terminal. Buffer gates merely serve the purpose of signal amplification: taking a "weak" signal source that isn't capable of sourcing or sinking much current, and boosting the current capacity of the signal so as to be able to drive a load.



Input	Output
0	0
1	1



PROGRAM :

/*Buffer program Verilog code-file name buffer.v*/	**/*Buffer program Verilog code for testbench -file name tb_buffer.v*/**

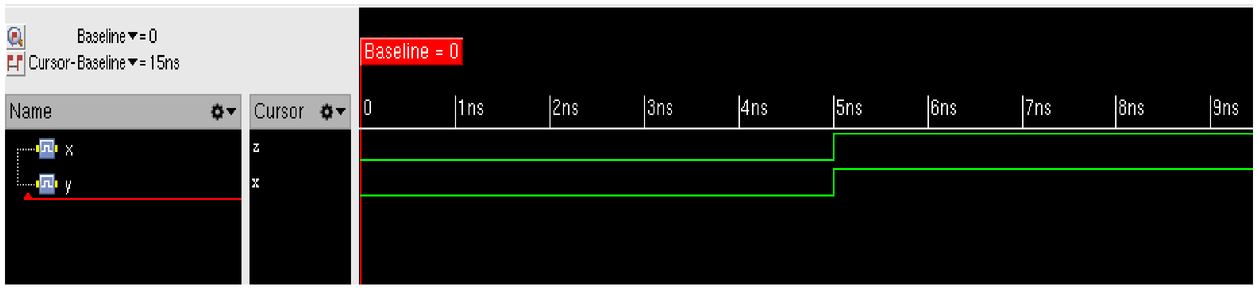
module buffer(x,y);
input x;
output y;
wire t1;
inv ins1(x,t1);
inv ins2(t1,y);
endmodule
	module tb_buffer;
	reg x;
	wire y;
	buffer dut_buffer(x,y);
	initial begin
	x = 1'b0;
	#5 x = 1'b1;
	#5 x = 1'bX;
	#5 x = 1'bZ;
	end
	endmodule

Constraints file for Synthesis:

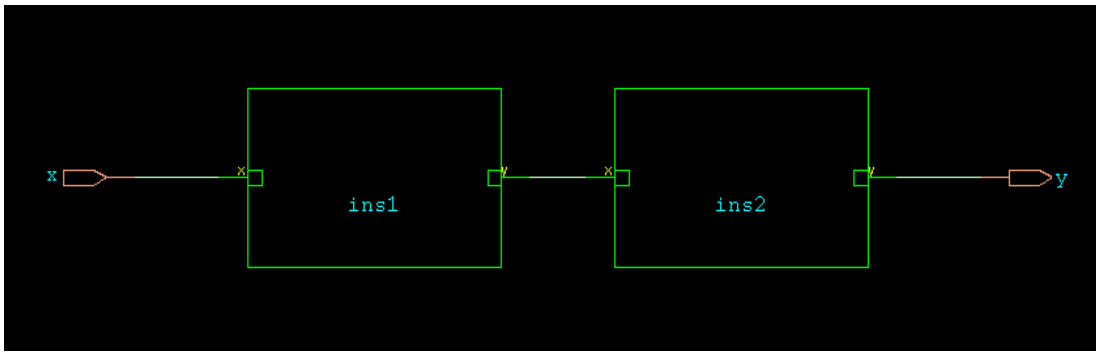
```
set_input_delay -max 1.0 [get_ports "x"]
set_output_delay -max 1.0 [get_ports "y"]
```

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the buffer circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

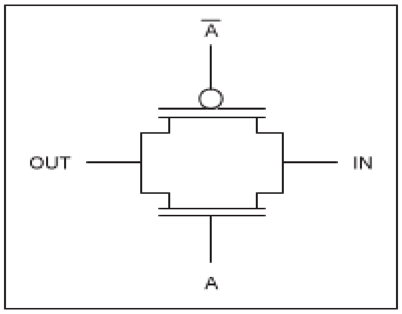
3. Transmission Gate

AIM: To write verilog code for Transmission Gate circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

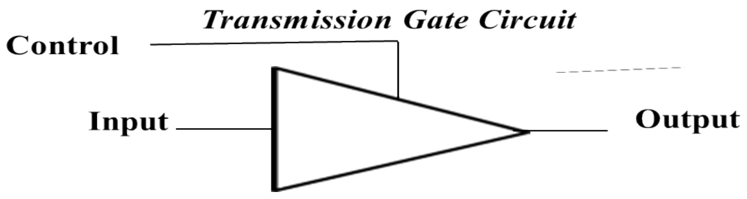
THEORY: A transmission gate, or analog switch, is defined as an electronic element that will selectively block or pass a signal level from the input to the output. Basic Operation This solid-state switch is comprised of a pMOS transistor and nMOS transistor. The control gates are biased in a complementary manner so that both transistors are either on or off. When the voltage on node A is a Logic 1, the complementary Logic 0 is applied to node active-low A, allowing both transistors to conduct and pass the signal at IN to OUT. When the voltage on node active-low A is a Logic 0, the complementary Logic 1 is applied to node A, turning both transistors off and forcing a high-impedance condition on both the IN and OUT nodes. This high-impedance condition represents the third "state" (high, low, or high-Z) that the channel may reflect downstream. The schematic diagram (Figure 1) includes the arbitrary labels for IN and OUT, as the circuit will operate in an identical manner if those labels were reversed. This design provides true bidirectional connectivity without degradation of the input signal.

Symbol of Transmission Gate



Truth table of Transmission gate

Control	IN	OUT
0	X	0
1	0	0
1	1	1



PROGRAM:

/*transmission gate program Verilog code-file name transgate.v*/	**/*transmission gate program Verilog code for testbench -file name tb_transgate.v*/**

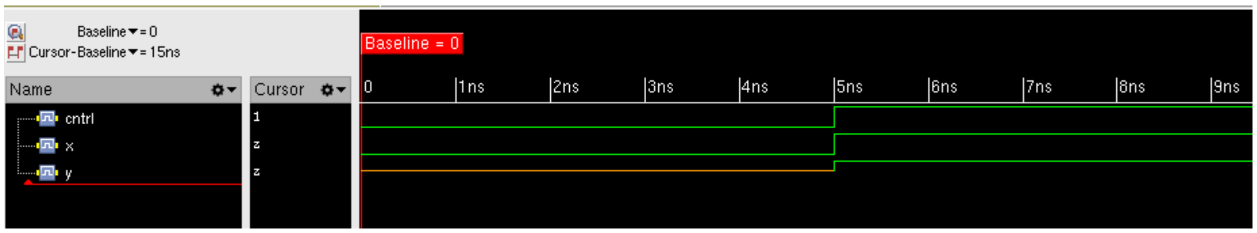
module transgate(x,cntrl,y);
input x,cntrl;
output y;
assign y = cntrl?x:1'bz;
endmodule
	module tb_transgate;
	reg x,cntrl;
	wire y;
	transgate dut_transgate(x,cntrl,y);
	initial begin
	x = 1'b0; cntrl = 1'b0;
	#5 x = 1'b1; cntrl = 1'b1;
	#5 x = 1'bx; cntrl = 1'b0;
	#5 x = 1'bz; cntrl = 1'b1;
	end
	endmodule

Constraints file for Synthesis:

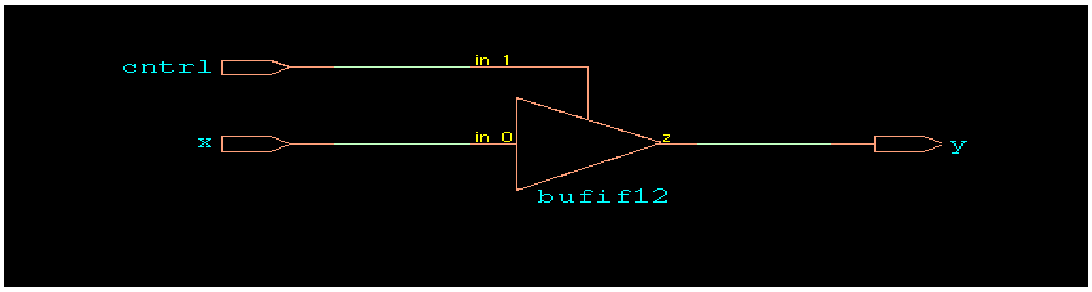
```
set_input_delay -max 1.0 [get_ports "x"]
set_input_delay -max 1.0 [get_ports "cntrl"]
set_output_delay -max 1.0 [get_ports "y"]
```

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the transmission gate circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

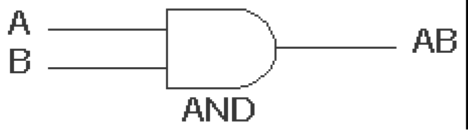
4. Basic Gates

AIM:
To write verilog code for an universal Gate circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:
Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates.

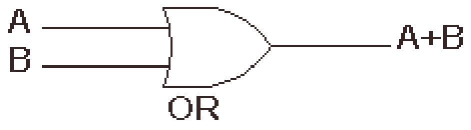
1. AND gate



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

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

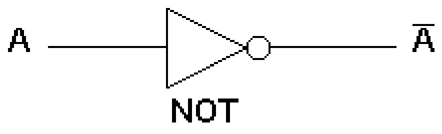
2. OR gate



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

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

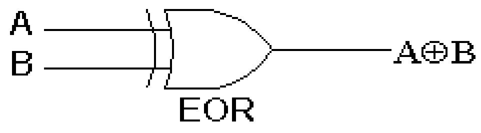
3. NOT gate



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

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.

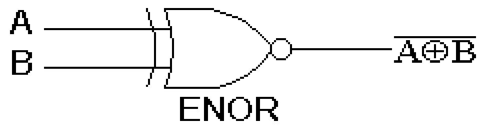
4. EXOR gate



2 Input EXOR gate		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

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

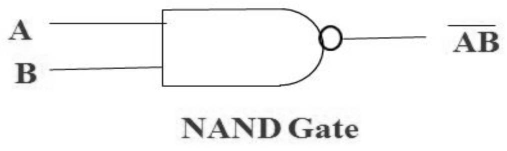
5. ENOR gate



2 Input EXNOR gate		
A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR'** gate circuit does the opposite to the EOR gate. It will give a low output if **either, but not both**, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

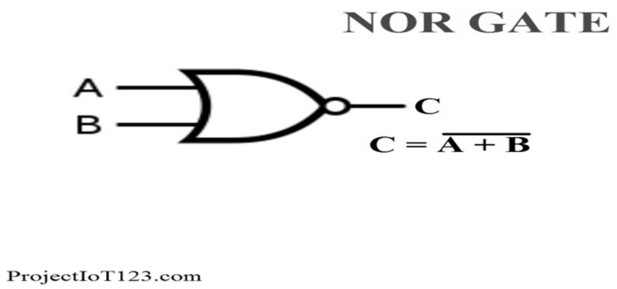
6. NAND gate

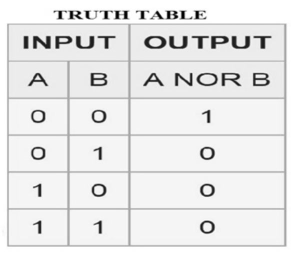


2 Input NAND gate		
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

7. NOR GATE

A NOR gate is logically an inverted OR gate. It has the following truth table:





TRUTH TABLE

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

Truth Table

A	B	NOT	AND	NAND	OR	NOR	EX-OR	EX-NOR
0	0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	0	1	0	0	1

PROGRAM:

```
/*Basic gates program Verilog code-file name basicgates.v*/

module basicgates(i1,i2,sel,y);
input i1,i2;
input [2:0] sel;
output reg y;
always@(sel or i1 or i2)
begin
case (sel)
3'b000: y = ~i1;
3'b001: y = i1 & i2;
3'b010: y = i1 |  i2;
3'b011: y = ~(i1 & i2);
3'b100: y = ~(i1 | i2);
3'b101: y = (i1 ^ i2);
3'b110: y = ~(i1 ^ i2);
3'b111: y = i2;
endcase
end
endmodule
```

```
/*Basic gates program Verilog code for testbench file name tb_basicgates.v*/

module  tb_basicgates;
reg  i1,i2;
reg  [2:0] sel;
wire y;
basicgates  dut_basicgates(i1,i2,sel,y);
initial begin
   i1 = 1'b1  ; i2 = 1'b0;
sel = 3'b000;
#5 sel = 3'b001;
#5 sel = 3'b010;
#5 sel = 3'b011;
#5 sel = 3'b100;
#5 sel = 3'b101;
#5 sel = 3'b110;
#5 sel = 3'b111;
#5;
end
endmodule.
```

Constraints file for Synthesis:

set_input_delay -max 1.0 [get_ports "i1"]

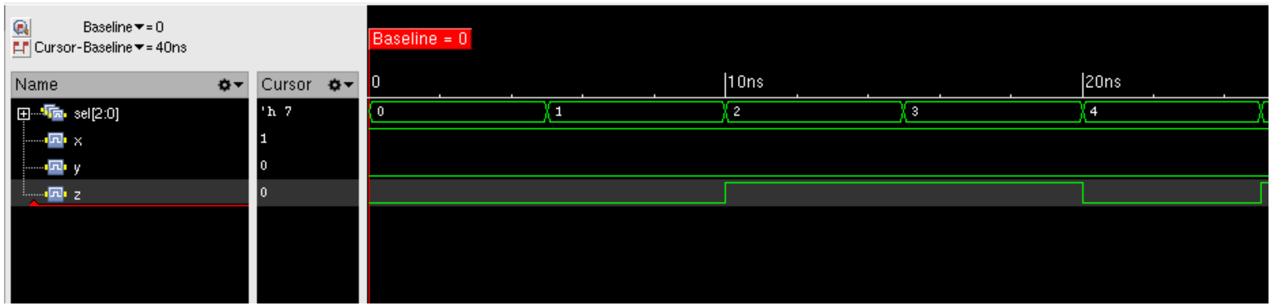
set_input_delay -max 1.0 [get_ports "i2"]

set_input_delay -max 1.0 [get_ports "sel"]

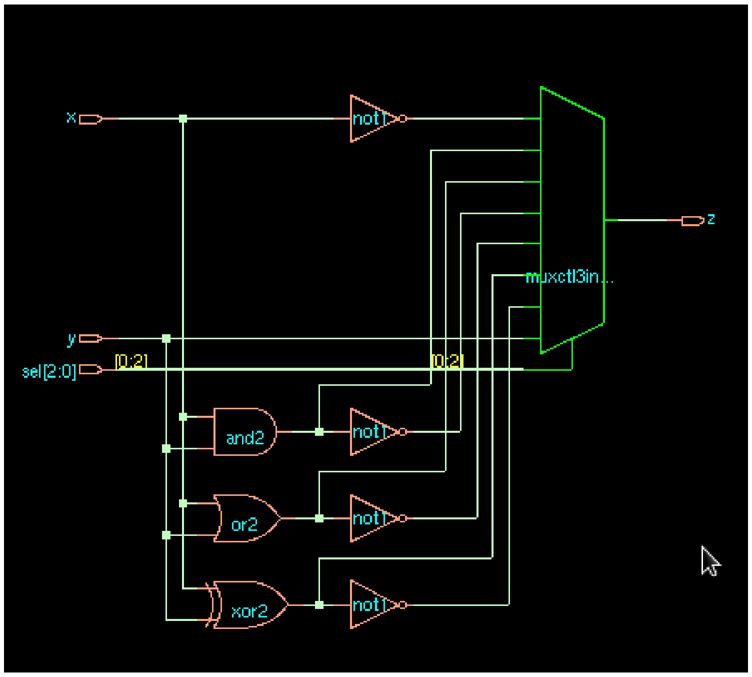
set_output_delay -max 1.0 [get_ports "y"]

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the all basic gates and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

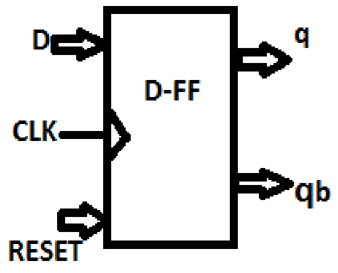
5a. D Flip-Flop

AIM: To write verilog code for an D flip-flop circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY: In electronics, a flip-flop or latch is a circuit that has two stable states and can be used to store state information. A flip-flop is a bistable multivibrator. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Flip-flops and latches are used as data storage elements. A flip-flop stores a single bit (binary digit) of data; one of its two states represents a "one" and the other represents a "zero". Such data storage can be used for storage of state, and such a circuit is described as sequential logic.

D flip-flop The D flip-flop is widely used. It is also known as a "data" flip-flop. The D flip-flop captures the value of the D-input at a definite portion of the clock cycle (such as the rising edge of the clock). That captured value becomes the Q output. At other times, the output Q does not change. The D flip-flop can be viewed as a memory cell, a zero-order hold, or a delay line.



D-FF SYMBOL

Truth Table

reset	clock	d	q	qb
0	↑	1	1	0
0	↑	0	0	1
0	0	X	X	Hold
1	↑	X	0	1

('**X**' denotes a **Don't care** condition, meaning the signal is irrelevant & **'Hold'** means output follows the previous state).

PROGRAM :

```
/*D Flip-flop program Verilog code-file name dff.v*/

module dff(clk,rst,din,q,qb);
input clk,rst,din;
output  q,qb;
reg q;
assign qb = ~q;
always@(posedgeclk)
begin
if(rst)
q<= 1'b0;
else
q<=din;
end
endmodule
```

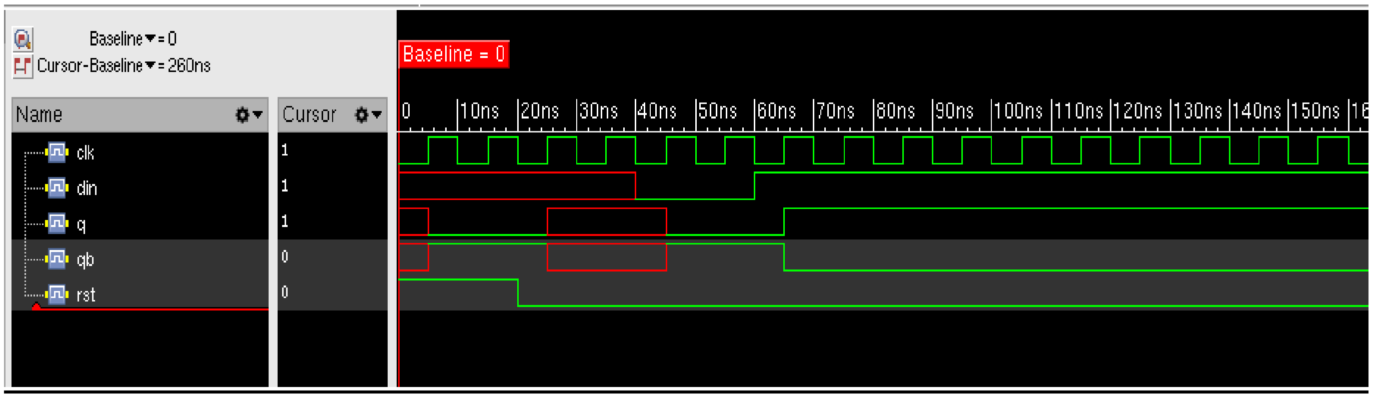
```
/*D flip-flop program Verilog code for testbench -file name tb_dff.v*/

module  tb_dff;
reg  clk,rst;
reg  ndin;
wire q,qb;
dff  dut_dff(clk,rst,din,q,qb);
initial begin  /* generating the clock stimulus */
clk = 1'b0;
forever #5 clk = ~clk;
end
initial begin /* generating the flop input stimulus */
rst = 1'b1;
#20  rst = 1'b0;
#20  din = 1'b0;
#20  din = 1'b1;
#200 $finish;
end
endmodule
```

Constraints file for Synthesis:

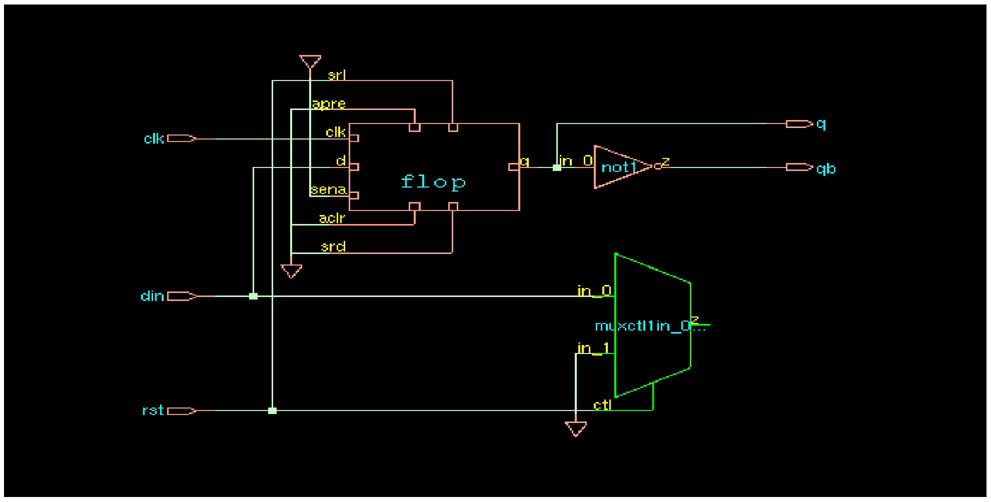
```
create_clock -name clk -period 10 -waveform {0 5} [get_ports "clk"]

set_clock_transition -rise 0.1 [get_clocks "clk"]

set_clock_transition -fall 0.1 [get_clocks "clk"]

set_clock_uncertainty 1.0 [get_ports "clk"]

set_input_delay -max 1.0 [get_ports "rst"] -clock [get_clocks "clk"]

set_input_delay -max 1.0 [get_ports "din"] -clock [get_clocks "clk"]

set_output_delay -max 1.0 [get_ports "q"] -clock [get_clocks "clk"]

set_output_delay -max 1.0 [get_ports "qb"] -clock [get_clocks "clk"]
```

OUTPUT

Simulation:


Baseline ▾ = 0
Cursor-Baseline ▾ = 260ns
Baseline = 0
Name
Cursor
clk 1
din 1
q 1
qb 0
rst 0
0 10ns 20ns 30ns 40ns 50ns 60ns 70ns 80ns 90ns 100ns 110ns 120ns 130ns 140ns 150ns

Synthesis:



RESULT: Verilog code for the D flip-flop circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

5b. JK Flip-Flop

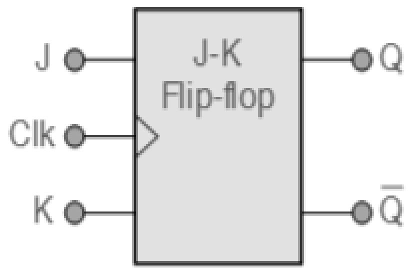
AIM: To write verilog code for an JK flip-flop circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

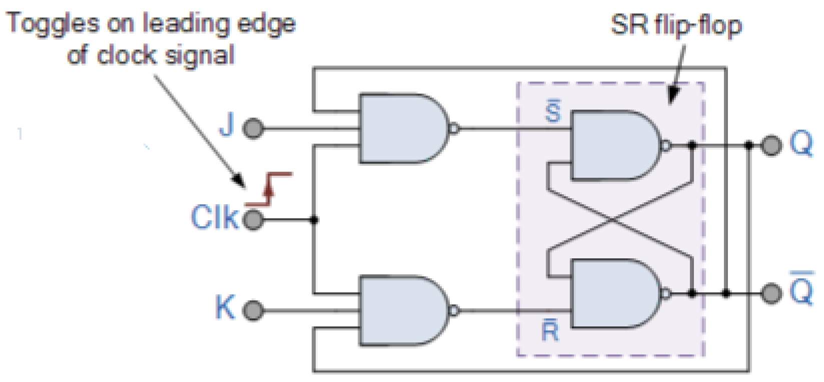
THEORY:

Then the JK flip-flop is basically an SR flip-flop with feedback which enables only one of its two input terminals, either SET or RESET to be active at any one time thereby eliminating the invalid condition seen previously in the SR flip-flop circuit. Also when both the J and the K inputs are at logic level “1” at the same time, and the clock input is pulsed either “HIGH”, the circuit will “toggle” from its SET state to a RESET state, or visa-versa. This result in the JK flip-flop acting more like a T-type toggle flip-flop when both terminals are “HIGH”

Although this circuit is an improvement on the clocked SR flip-flop it still suffers from timing problems called “race” if the output Q changes state before the timing pulse of the clock input has time to go “OFF”. To avoid this the timing pulse period (T) must be kept as short as possible (high frequency). As this is sometimes not possible with modern TTL IC’s the much improved MasterSlave JK Flip-flop was developed.



logic symbol



Circuit Diagram

Truth Table:

	Clk	J	K	Q	Qb	Comments
	↑	0	0	Hold		No change
Same as for SR latch	↑	0	1	0	1	Reset Q>>0
	↑	1	0	1	0	Set Q>>1
Toggle action	↑	1	1	0	1	Toggle
				1	0	

PROGRAM:

/*JK Flip-flop program Verilog code-file name jkff.v*/

```
module jkff(j,k,rst,clk,q,qb);
input  j,k,clk,rst;
output q,qb;
reg q;
assign qb = ~q;
always@(posedgeclk)
begin
if(rst)
q <= 1'b0;
else
begin
case ({j,k})
 2'b00: q <= q;
 2'b01: q <= 1'b0;
 2'b10: q <= 1'b1;
 2'b11: q <= ~q; /*according to the definition
of JK flip-flop*/
endcase
end
end
endmodule
```

/*JK Flip-flop program Verilog code for testbench -file name tb_jkff.v*/

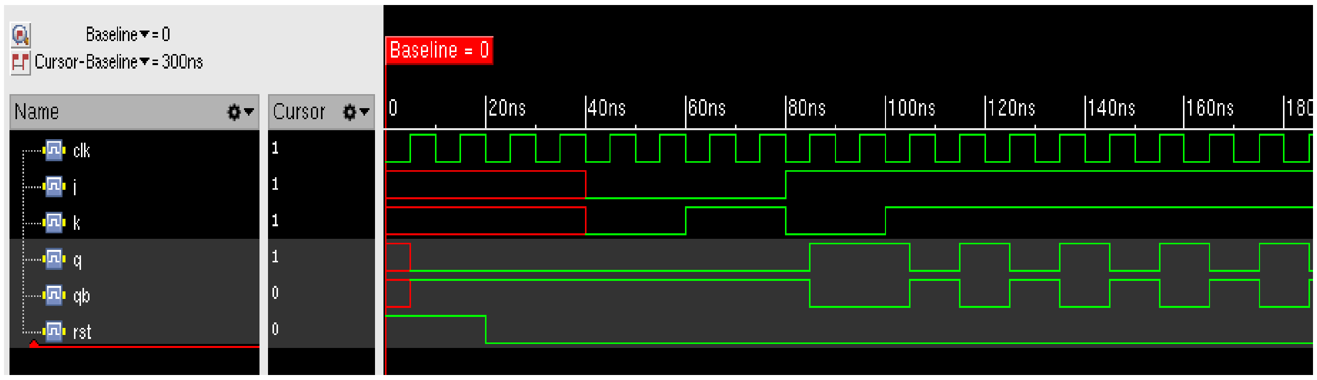
```
module tb_jkff;
reg  clk,rst;
reg  j,k;
wire q,qb;
jkff  dut_jkff(j,k,rst,clk,q,qb);
initial begin  /* generating the clock stimulus*/
clk = 1'b0;
forever #5 clk = ~clk;
end

initial begin /* generating the flop input
stimulus*/
rst = 1'b1;
#20 rst = 1'b0;
#20 j = 1'b0 ; k = 1'b0;
#20 j = 1'b0 ; k = 1'b1;
#20 j = 1'b1 ; k = 1'b0;
#20 j = 1'b1 ; k = 1'b1;
#200 $finish;
end
endmodule
```

Constraints file for Synthesis:

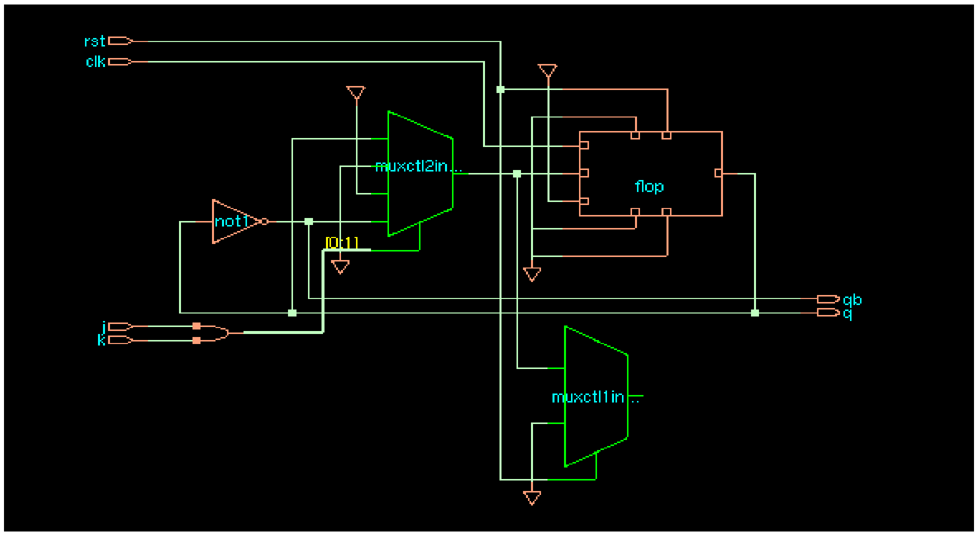
```
create_clock -name clk -period 10 -waveform {0 5} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 1.0 [get_ports "clk"]
set_input_delay -max 1.0 [get_ports "rst"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "j"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "k"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "q"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "qb"] -clock [get_clocks "clk"]
```

OUTPUT:

Simulation:



Synthesis:



RESULT:

Verilog code for the JK flip-flop circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

5c. MASTER SLAVE JK Flip-Flop

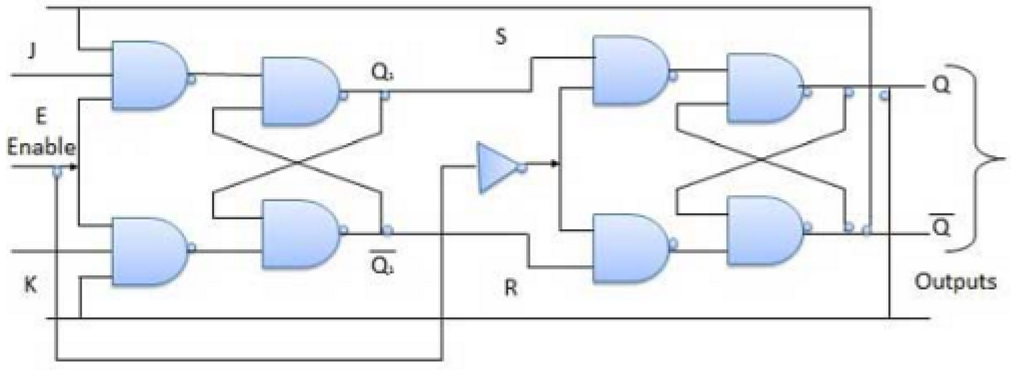
AIM: To write verilog code for an MASTER SLAVE JK flip-flop circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

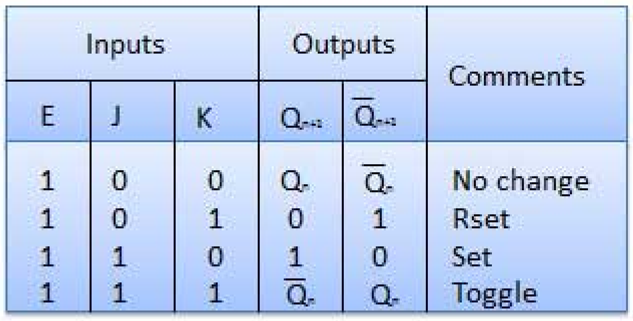
THEORY:

Master slave JK FF is a cascade of two S-R FF with feedback from the output of second to input of first. Master is a positive level triggered. But due to the presence of the inverter in the clock line, the slave will respond to the negative level. Hence when the clock = 1 (positive level) the master is active and the slave is inactive. Whereas when clock = 0 (low level) the slave is active and master is inactive.

Circuit Diagram



Truth Table



Inputs			Outputs		Comments
E	J	K	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	Q_n	$\overline{Q}_n$	No change
1	0	1	0	1	Rset
1	1	0	1	0	Set
1	1	1	$\overline{Q}_n$	Q_n	Toggle

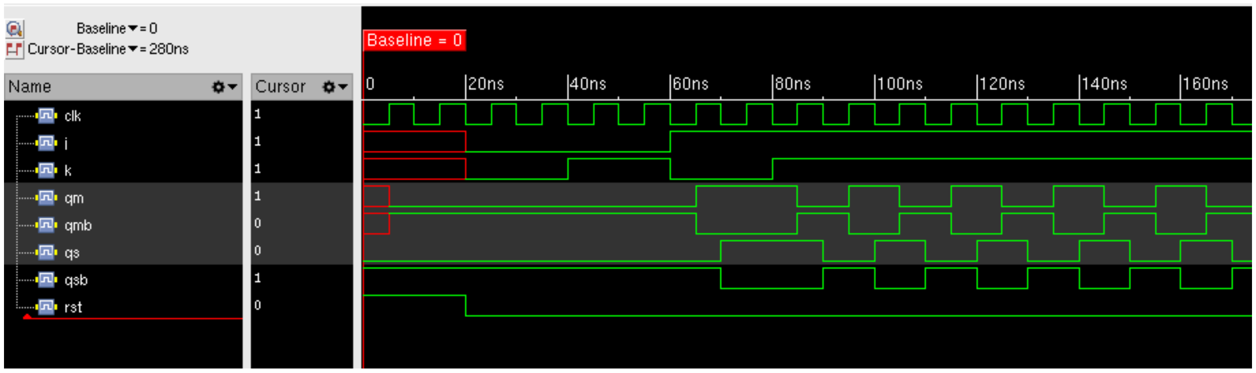
PROGRAM:

/*Master – slave JK Flip-flop program Verilog code-file name ms_jkff.v*/	**/*Master – Slave JK Flip-flop program Verilog code for testbench -file name tb_msjkff.v*/**

module ms_jkff(j,k,rst,clk,qs,qsb,qm,qmb);
input j,k,clk,rst;
output qs,qsb,qm,qmb;
wire clkbar;
assign clkbar = ~clk;
jkff jk1(qm,qmb,j,k,clk,rst);
jkff jk1(qs,qsb,qm,qmb,clkbar,rst);
endmodule
	module tb_msjkff;
	reg j,k,clk,rst;
	wire qs,qsb,qm,qmb;
	ms_jkff ms1(j,k,rst,clk,qs,qsb,qm,qmb);
	initial
	clk=1’b0;
	always
	#10 clk=~clk;
	initial
	begin
	rst=1’b1; j=1’b0; k=1’b0;
	#15 rst =1’b0;
	#25 k=1’b1; j=1’b0;
	#25 j=1’b1; k=1’b0;
	#25 k=1’b1; j=1’b1;
	#200 $finish;
	End
	endmodule

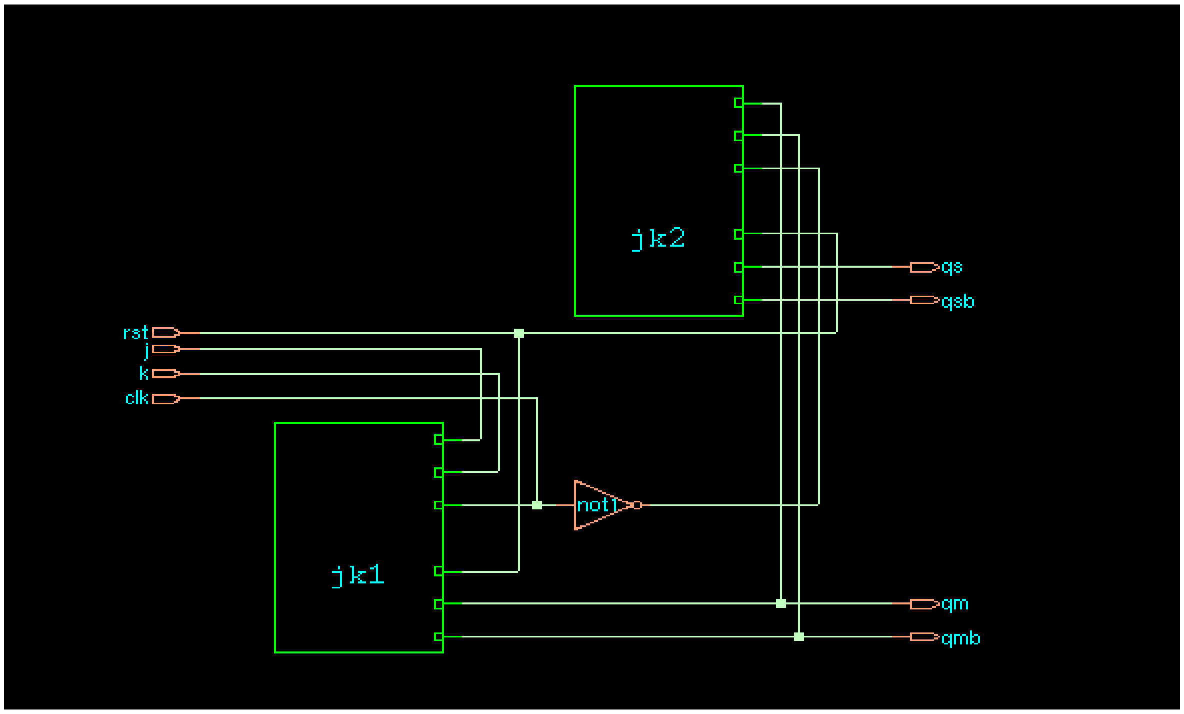
OUTPUT:

<u>Simulation:</u>


Baseline = 0
Cursor-Baseline = 280ns

Name	Cursor
clk	1
j	1
k	1
qm	1
qmb	0
qs	0
qsb	1
rst	0

Synthesis:



RESULT: Verilog code for the master slave JK flip-flop circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

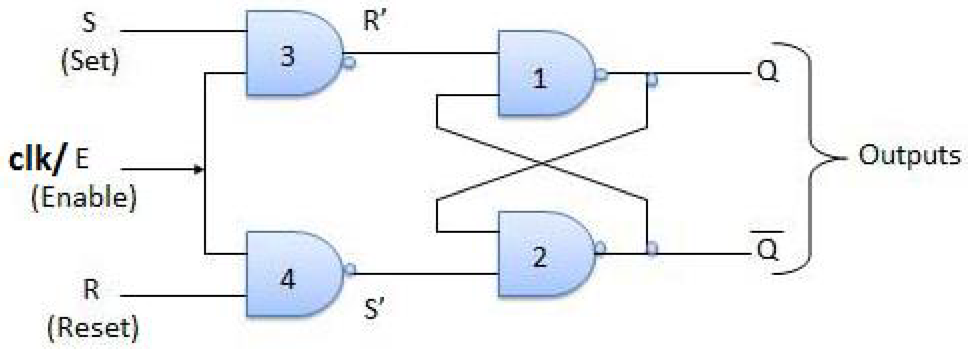
5d. SR Flip-Flop

AIM: To write verilog code for an SR flip-flop circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:

It can be seen that when both inputs S = "1" and R = "1" the outputs Q and Q can be at either logic level "1" or "0", depending upon the state of the inputs S or R BEFORE this input condition existed. Therefore the condition of S = R = "1" does not change the state of the outputs Q and Q. However, the input state of S = "0" and R = "0" is an undesirable or invalid condition and must be avoided. The condition of S = R = "0" causes both outputs Q and Q to be HIGH together at logic level "1" when we would normally want Q to be the inverse of Q. The result is that the flip-flop loses control of Q and Q, and if the two inputs are now switched "HIGH" again after this condition to logic "1", the flip-flop becomes unstable and switches to an unknown data state based upon the unbalance.



Truth Table

Clk	R	S	Q	Qb	comments
↑	0	0	Hold		No change
↑	0	1	1	0	set
↑	1	0	0	1	reset
↑	1	1	z	z	indeterminate
0	x	x	Hold		No change

PROGRAM:

/*SR Flip-flop program Verilog code-file name srff.v*/	**/*SR Flip-flop program Verilog code for testbench-file name tb_srff.v*/**

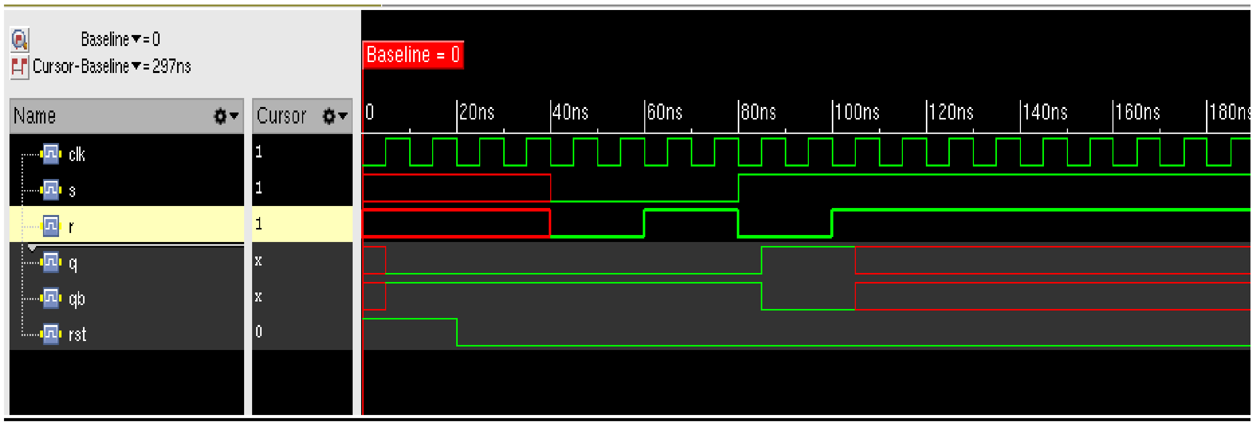
module srff(s,r,clk,rst,q,qb);
input clk,rst;
input s,r;
output q,qb;
reg q; /* always declare "q" as register before you use it inside always block */
assign qb = ~q;
always@(posedgeclk)
begin
if(rst)
q <= 1'b0;
else
begin
case ({s,r})
2'b00: q <= q;
2'b01: q <= 1'b0;
2'b10: q <= 1'b1;
2'b11: q <= 1'bx;
endcase
end
end
endmodule
	module tb_srff;
	reg clk,rst;
	reg s,r;
	wire q,qb;
	srff dut_srff(s,r,clk,rst,q,qb);
	initial begin /*generating the clock stimulus*/
	clk = 1'b0;
	forever #5 clk = ~clk;
	end
	initial begin /*generating the flop input stimulus*/
	rst = 1'b1;
	#20 rst = 1'b0;
	#20 s = 1'b0 ; r = 1'b0;
	#20 s = 1'b0 ; r = 1'b1;
	#20 s = 1'b1 ; r = 1'b0;
	#20 s = 1'b1 ; r = 1'b1;
	#200 $finish;
	end
	endmodule

Constraints file for Synthesis:

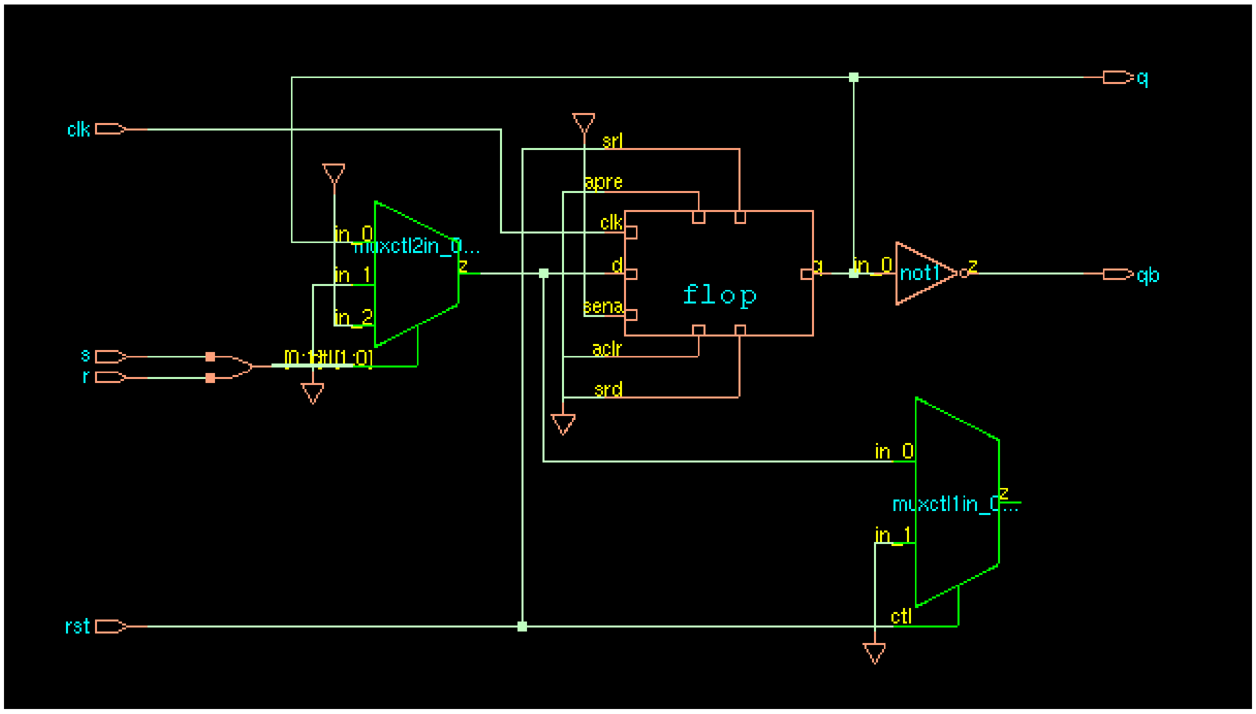
```
create_clock -name clk -period 10 -waveform {0 5} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 1.0 [get_ports "clk"]
set_input_delay -max 1.0 [get_ports "rst"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "s"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "r"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "q"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "qb"] -clock [get_clocks "clk"]
```

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the SR flip-flop circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

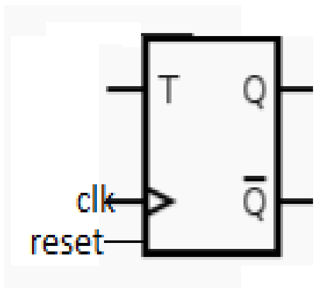
5e. T Flip-Flop

AIM: To write verilog code for an T flip-flop circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:
A circuit symbol for a T-type flip-flop If the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value. This behavior is described by the truth table.

When T is held high, the toggle flip-flop divides the clock frequency by two; that is, if clock frequency is 4 MHz, the output frequency obtained from the flip-flop will be 2 MHz. This "divide by" feature has application in various types of digital counters. A T flip-flop can also be built using a JK flip-flop (J & K pins are connected together and act as T) or a D flip-flop (T input XOR Previous drives the D input).



Logic symbol

Truth Table

Reset	Clock	T	Q_{n-1}	Q	Q_b
0	↑	0	0	0	1
0	↑	0	1	1	0
0	↑	1	0	1	0
0	↑	1	1	0	1
1	↑	x	x	0	1

PROGRAM:

/*T Flip-flop program Verilog code-file name tff.v*/	**/*T Flip-flop program Verilog code for testbench-file name tb_tff.v*/**

module tff(clk,rst,T,q,qb);
input clk,rst,T;
output q,qb;
reg q;
assign qb = ~q;
always@(posedgeclk)
begin
if(rst)
q<= 1'b0;
else if(T)
q<=~q;
else
q<=q;
end
endmodule
	module tb_tff;
	reg clk,rst;
	reg T;
	wire nq,qb;
	tff dut_tff(clk,rst,T,q,qb);
	initial begin /* generating the clock stimulus */
	clk = 1'b0;
	forever #5 clk = ~clk;
	end
	initial begin /* generating the flop input stimulus */
	rst = 1'b1;
	#20 rst = 1'b0;
	#20 T = 1'b0;
	#20 T = 1'b1;

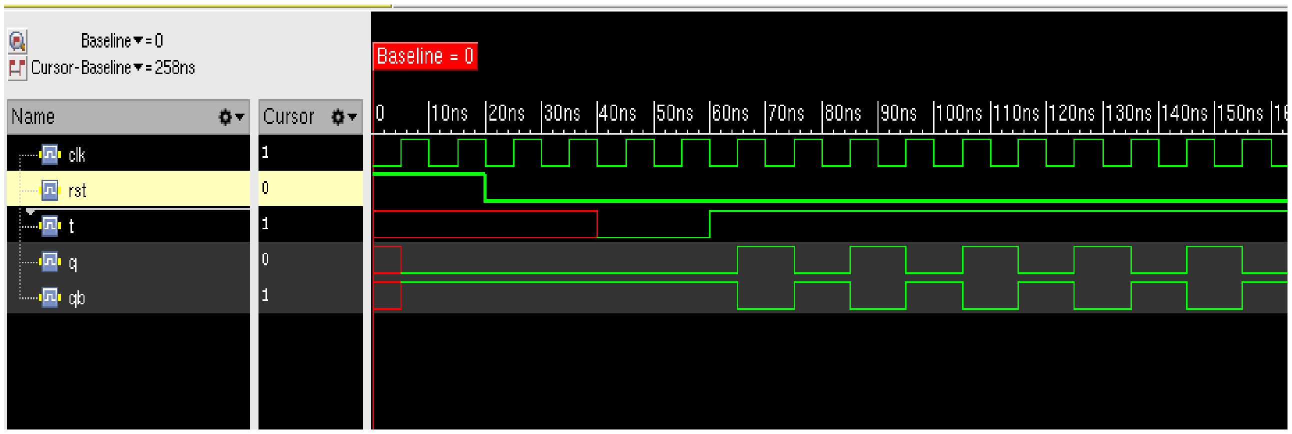
	#200 $finish;
	end
	endmodule

Constraints file for Synthesis:

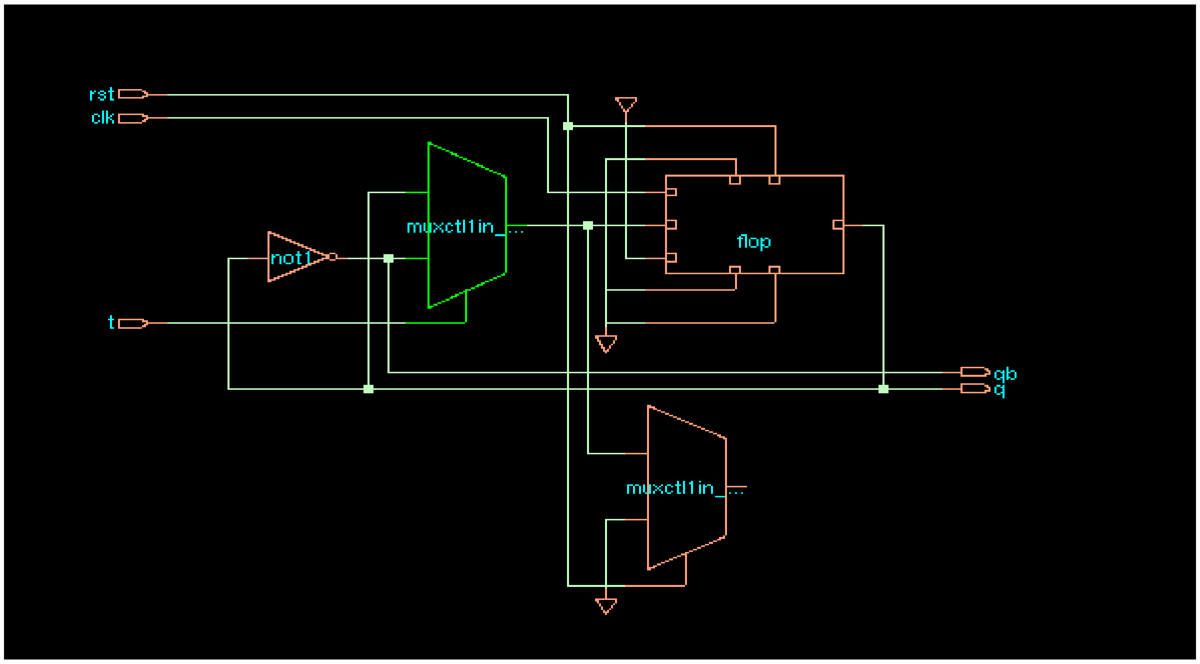
```
create_clock -name clk -period 10 -waveform {0 5} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 1.0 [get_ports "clk"]
set_input_delay -max 1.0 [get_ports "rst"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "T"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "q"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "qb"] -clock [get_clocks "clk"]
```

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the T flip-flop circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

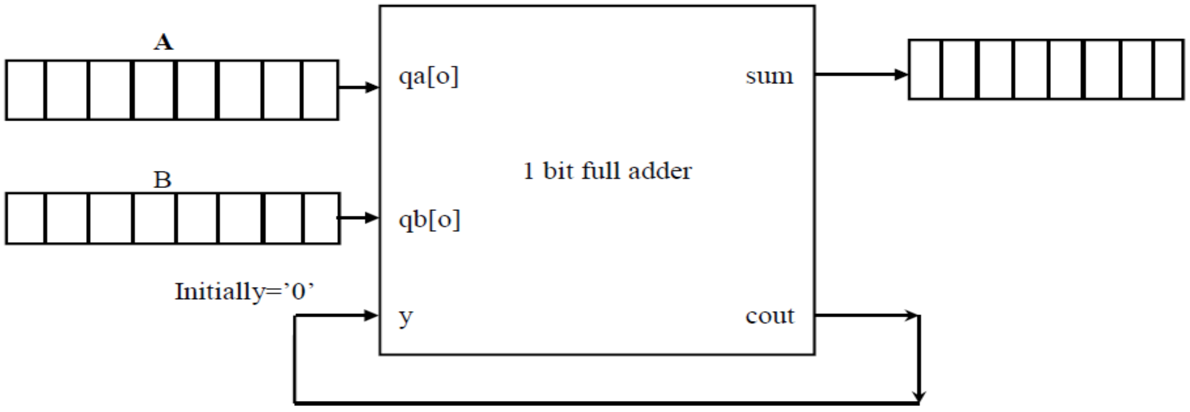
6a. Serial Adder

AIM: To write verilog code for Serial adder circuit and its testbench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:

Serial adder is implemented using a 1 bit full adder. In this adder, there is a serial computation of inputs i.e. only single least significant bit (LSB) is taken as input to the 1 bit full adder along with the carry bit of previous computation and sum and carry out (Cout) is obtained for that bit position. Similarly the same process is carried out by shifting the input bits towards right and then all the 8-bits of both the input signals are computed along with all the carry outs generated and the sum is stored.



Truth Table

A	B	C	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

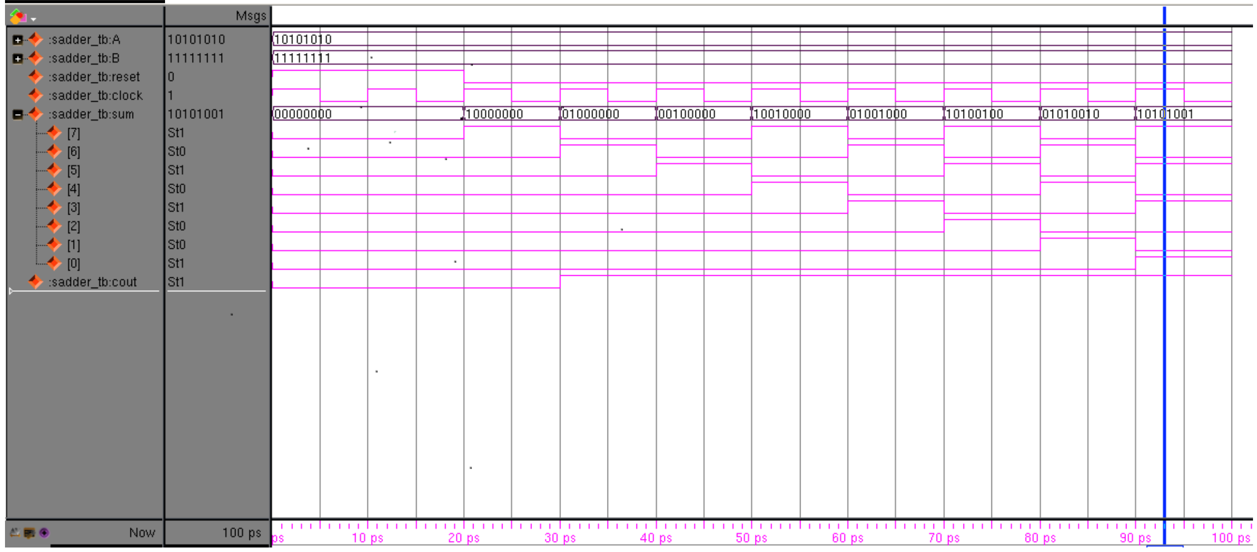
PROGRAM:

/*Serial Adder program Verilog code-file serialadder.v*/	**/*Full Adder program Verilog code for Test bench-file name tb_serial_adder.v*/**

```
//Verilog code
module serial_adder(clk,rst,A,B,FSUM);
input [7:0] A,B;
input clk, rst;
output  [8:0] FSUM;
reg [3:0] i;
reg [8:0] c;
reg [7:0] sum;
assign  FSUM = {c[8],sum[7:0]};

always@(posedge clk)
begin
if(rst)
begin
c[0] <= 1’b0;
i <= 4’d0;
end
else
begin
sum[i] <= A[i]^B[i]^C[i];
C[i+1] <= A[i]&B[i] | B[i]&C[i] & C[i]&A[i];
i <= i+1;
end
end
endmodule
```
	```
	module tb_serial_adder;
	reg  [7:0] A,B;
	reg  clk,rst;
	wire [8:0] FSUM;
	serial_adder dut_serial_adder(clk,
	rst,A,B,FSUM)’;
	initial begin
	clk=1'b0;
	forever #5 clk=~clk;
	end
	initial begin
	rst=1'b1;
	#20  rst = 1’b0;
	A = 8’d124;
	B = 8’d242;
	#100 rst = 1’b1;
	#20   rst = 1’b0;
	A = 8’d200;
	B = 8’d196;
	#140  $finish;
	end
	endmodule
	```

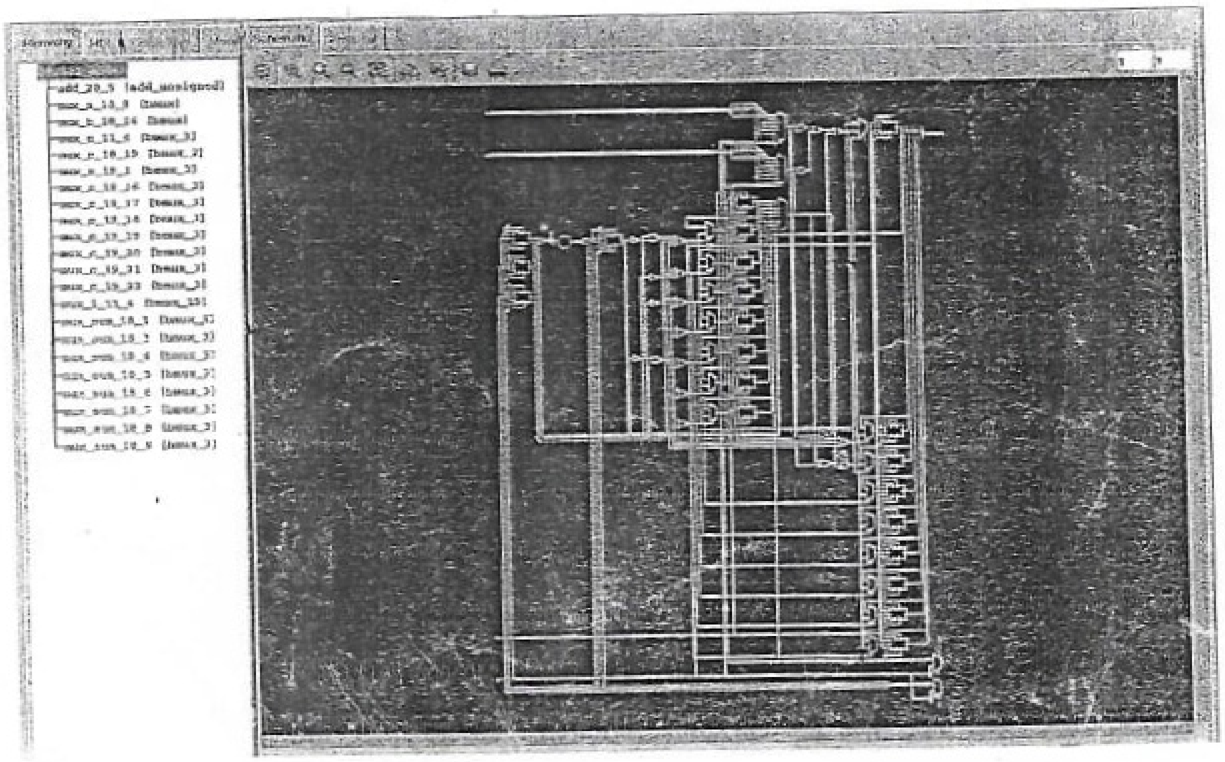
OUTPUT:

Simulation:



```
                     Msgs
:sadder_tb:A         10101010
:sadder_tb:B         11111111
:sadder_tb:reset     0
:sadder_tb:clock     1
:sadder_tb:sum       10101001
  [7]                St1
  [6]                St0
  [5]                St1
  [4]                St0
  [3]                St1
  [2]                St0
  [1]                St0
  [0]                St1
:sadder_tb:cout      St1
Now                  100 ps
```

Synthesis:



RESULT: Verilog code for the serial adder circuit and its test bench for verification written, the waveform is observed and the code is synthesized with the technological library and is verified.

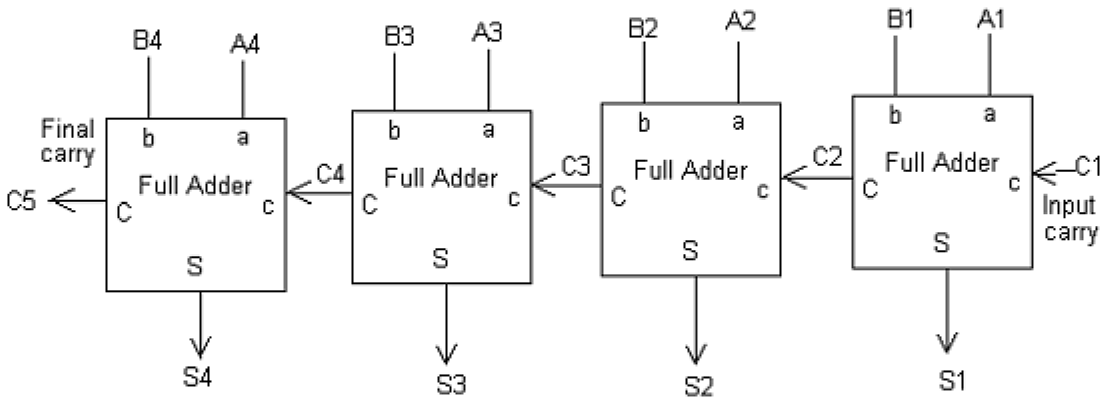
6b. Parallel Adder

AIM: To write verilog code for an parallel adder circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:

Addition is a fundamental operation for any digital system, digital signal processing or control system. A fast and accurate operation of a digital system is greatly influenced by the performance of the resident adders. Adders are also very important component in digital systems because of their extensive use in other basic digital operations such as subtraction, multiplication and division Parallel adder is a combinatorial circuit (not clocked, does not have any memory and feedback) adding every bit position of the operands in the same time. Thus it is requiring number of bit-adders(full adders + 1 half adder) equal to the number of bits to be added. The Parallel adder is constructed by cascading full adders (FA) blocks in series. One full adder is responsible for the addition of two binary digits at any stage of the ripple carry. The carryout of one stage is fed directly to the carry-in of the next stage.



PROGRAM:

/*Parallel Adder program Verilog code-file name parallel_adder.v*/	**/*Parallel Adder program Verilog code for test bench-file name tb_parallel_adder.v*/**

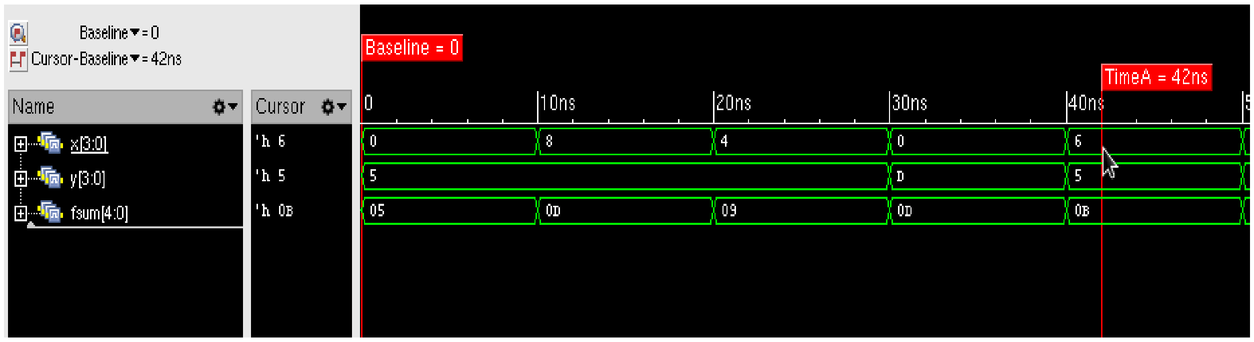
module parallel_adder(in1,in2,fsum);
input [3:0] in1,in2;
output [4:0] fsum; /* this is 5 bit because we are considering the final carryout(1 bit) also along with the sum (4bit)*/
wire [2:0] tc; /* this is for the intermediate carry that gets generated for the parallel adder*/
wire cin ; /* initial carry input*/
	module tb_parallel_adder;
	reg [3:0] in1,in2;
	wire [4:0] fsum;
	parallel_adder
	dut_parallel_adder(in1,in2,fsum);
	initial begin
	in1 = 4'b0000; in2 = 4'b0101;
	#10 in1 = 4'b1000; in2 = 4'b0101;
	#10 in1 = 4'b0100; in2 = 4'b0101;

```
assign cin = 1'b0; /* for a 4 bit parallel adder cin is
always 0 */
fulladder  FA1(in1[0],in2[0],cin,fsum[0],tc[0]);
fulladder  FA2(in1[1],in2[1],tc[0],fsum[1],tc[1]);
fulladder  FA3(in1[2],in2[2],tc[1],fsum[2],tc[2]);
fulladder  FA4(in1[3],in2[3],tc[2],fsum[3],fsum[4]);
endmodule
```

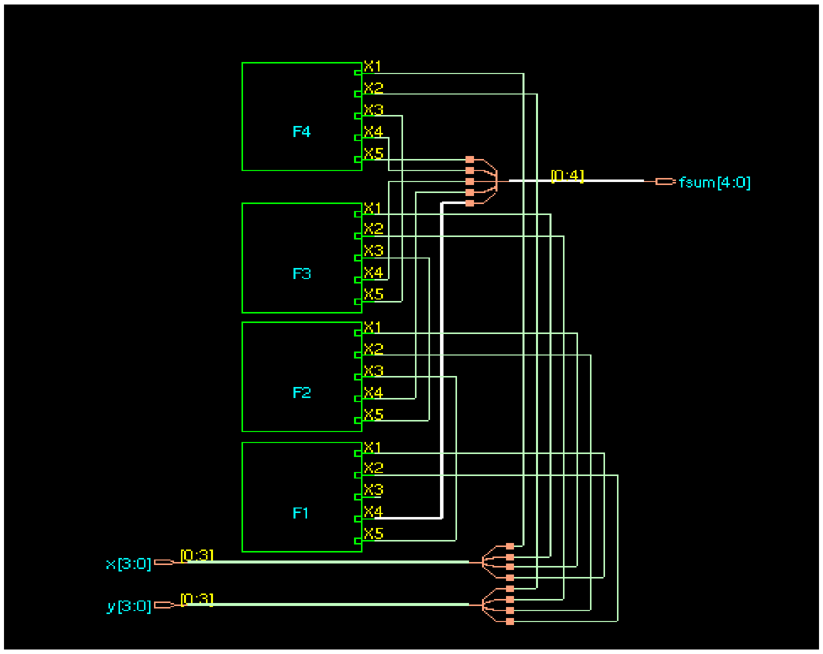
```
#10 in1 = 4'b0000; in2 = 4'b1101;
#10 in1 = 4'b0110; in2 = 4'b0101;
#10 in1 = 4'b1000; in2 = 4'b0111;
#30;
end
endmodule
```

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the parallel adder circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

7. Asynchronous counter

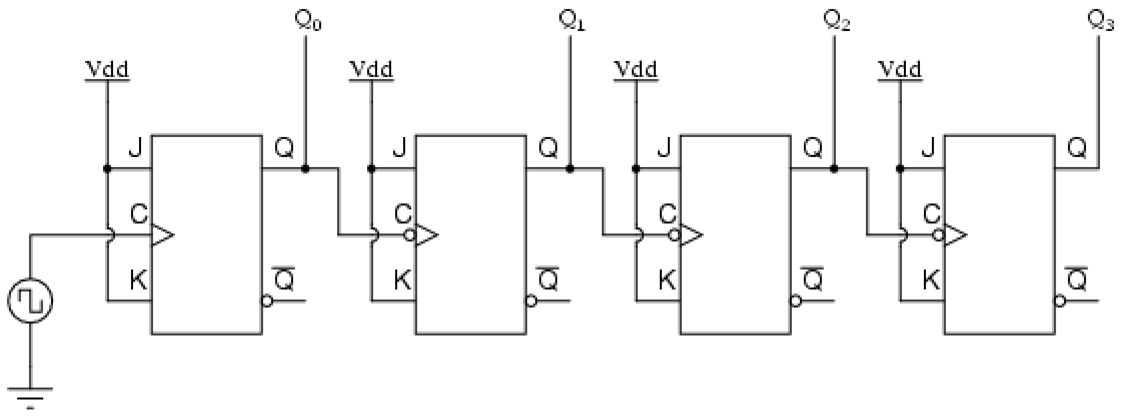
AIM: To write Verilog code for asynchronous counter circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

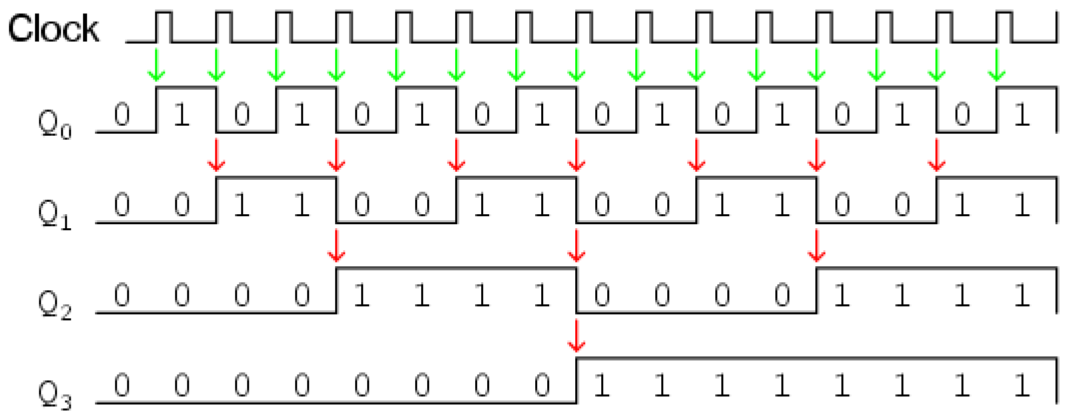
TOOL REQUIRED: Cadence Tool

THEORY:
A ripple counter is an asynchronous counter where only the first flip-flop is clocked by an external clock. All subsequent flip-flops are clocked by the output of the preceding flip-flop. Asynchronous counters are also called ripple-counters because of the way the clock pulse ripples it way through the flip-flops.

The MOD of the ripple counter or asynchronous counter is 2^n if n flip-flops are used. For a 4-bit counter, the range of the count is 0000 to 1111 (24 -1). A counter may count up or count down or count up and down depending on the input control. The count sequence usually repeats itself. When counting up, the count sequence goes from 0000, 0001, 0010, ... 1110 , 1111 , 0000, 0001, ... etc. When counting down the count sequence goes in the opposite manner: 1111, 1110, ... 0010, 0001, 0000, 1111, 1110, ... etc. Four J-K flip-flops connected in such a way to always be in the "toggle" mode, we need to determine how to connect the clock inputs in such a way so that each succeeding bit toggles when the bit before it transitions from 1 to 0. The Q outputs of each flip-flop will serve as the respective binary bits of the final, four-bit count. If we used flip-flops with negative-edge triggering (bubble symbols on the clock inputs), we could simply connect the clock input of each flip-flop to the Q output of the flip-flop before it, so that when the bit before it changes from a 1 to a 0, the "falling edge" of that signal would "clock" the next flip-flop to toggle the next bit.

A four-bit "up" counter




Clock
Q_0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Q_1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
Q_2 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
Q_3 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

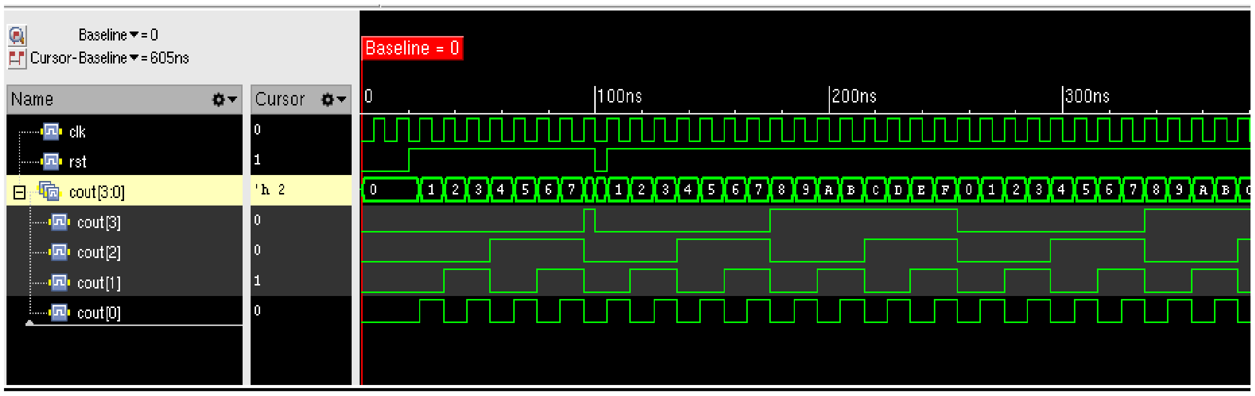
PROGRAM:

```
/*Asynchronous counter program Verilog code-file nameasynchcounter.v*/


module asynccounter(clk,rst,cout);
 input  clk,rst;
output [3:0] cout;
reg [3:0] cout;
always@(posedgeclk or negedgerst)
begin
if(!rst)  /* we are considering active_low
rst*/
cout<=4'd0;
else
cout<= cout+1;
end
endmodule
```

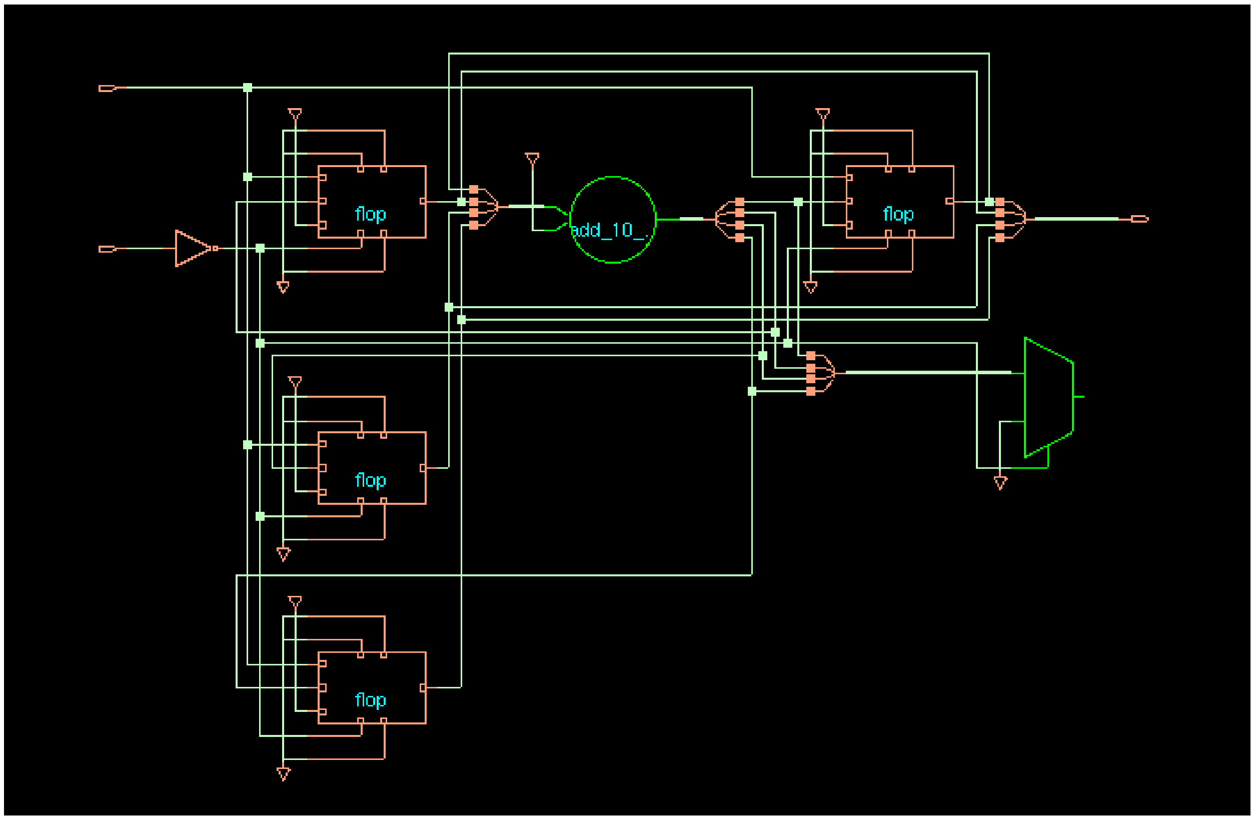
```
/*Asynchronous counter program Verilog
code for testbench-file
nametb_asyncounter.v*/

module  tb_asynccounter;
reg  clk,rst;
wire [3:0] cout;
asynccounter  dut_asynccounter(clk,rst,cout);
initial begin  /* generating the clock stimulus
*/
clk = 1'b0;
forever #5 clk = ~clk;
end
initial begin
rst = 1'b0;
#20 rst = 1'b1;
#80 rst = 1'b0;  /* this stimulus is to show that
this is a  asynchronous reset and so we can
reset it in between the clock */
#5 rst = 1'b1 ;/* we will see in the simulation
that this reset will  be recognised by the
design irrespective to what the  clock status is
, thus making it a asynchronous design*/
#500 $finish;
end
endmodule
```

OUTPUT:

Simulation:



Synthesis:



RESULT: Verilog code for the asynchronous counter circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

8. Synchronous Counter

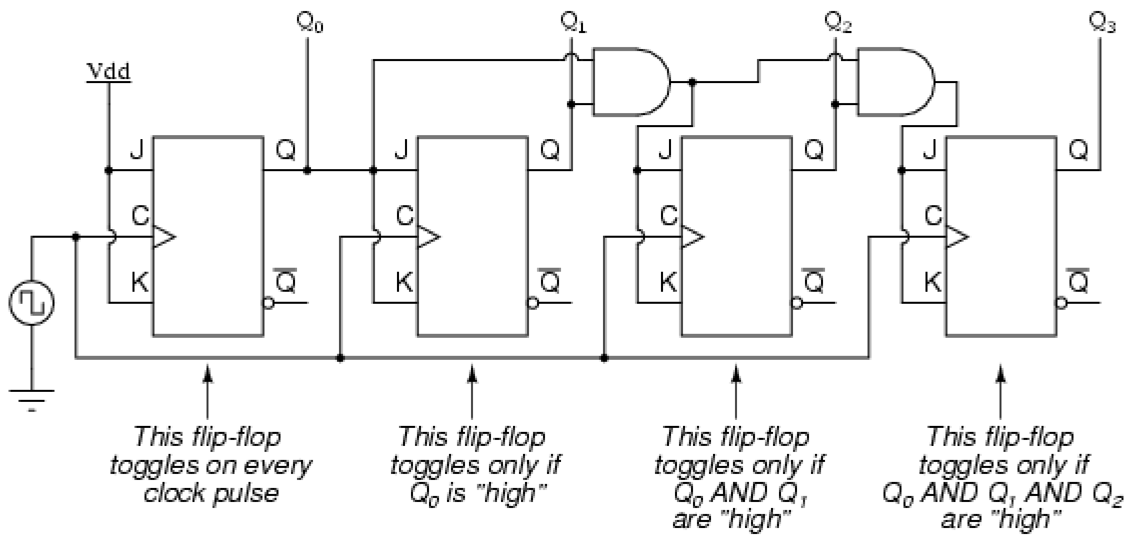
AIM: To write verilog code for synchronous counter circuit and its test bench for verification, observe the waveform and synthesize the code with technological library with given Constraints.

TOOL REQUIRED: Cadence Tool

THEORY:

A synchronous counter, in contrast to an asynchronous counter, is one whose output bits change state simultaneously, with no ripple. The only way we can build such a counter circuit from J-K flip-flops is to connect all the clock inputs together, so that each and every flip-flop receives the exact same clock pulse at the exact same time. The figure shows a four-bit synchronous “up” counter. Each of the higher-order flip-flops are made ready to toggle (both J and K inputs “high”) if the Q outputs of all previous flip-flops are “high.” Otherwise, the J and K inputs for that flip-flop will both be “low,” placing it into the “latch” mode where it will maintain its present output state at the next clock pulse. Since the first (LSB) flip-flop needs to toggle at every clock pulse, its J and K inputs are connected to Vcc or Vdd, where they will be “high” all the time. The next flip-flop need only “recognize” that the first flip-flop’s Q output is high to be made ready to toggle, so no AND gate is needed. However, the remaining flip-flops should be made ready to toggle only when all lower-order output bits are “high,” thus the need for AND gates.

A four-bit synchronous "up" counter


Q_0
Q_1
Q_2
Q_3
Vdd
J
C
K
Q
$\overline{Q}$
This flip-flop toggles on every clock pulse
This flip-flop toggles only if Q_0 is "high"
This flip-flop toggles only if Q_0 AND Q_1 are "high"
This flip-flop toggles only if Q_0 AND Q_1 AND Q_2 are "high"

PROGRAM:

/*Synchronous counter program Verilog code-file name Synchcounter.v*/

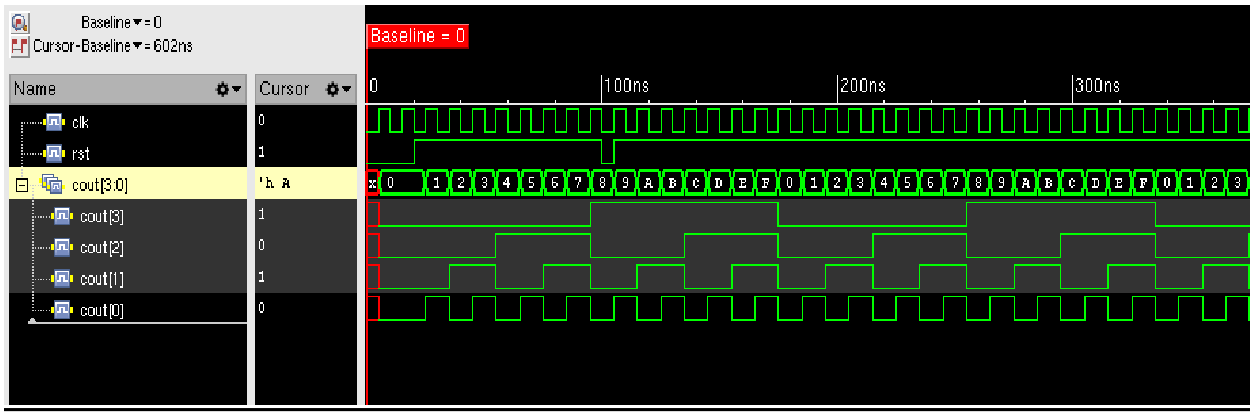
```
module synccounter(clk,rst,cout);
input  clk,rst;
output  [3:0] cout;
reg [3:0] cout;
always@(posedgeclk)
begin
if(!rst)  /* we are considering
active_lowrst*/
cout<=4'd0;
else
cout<= cout+1;
end
endmodule
```

/*Synchronous counter program Verilog code for test bench-file name tb_synchcounter.v*/

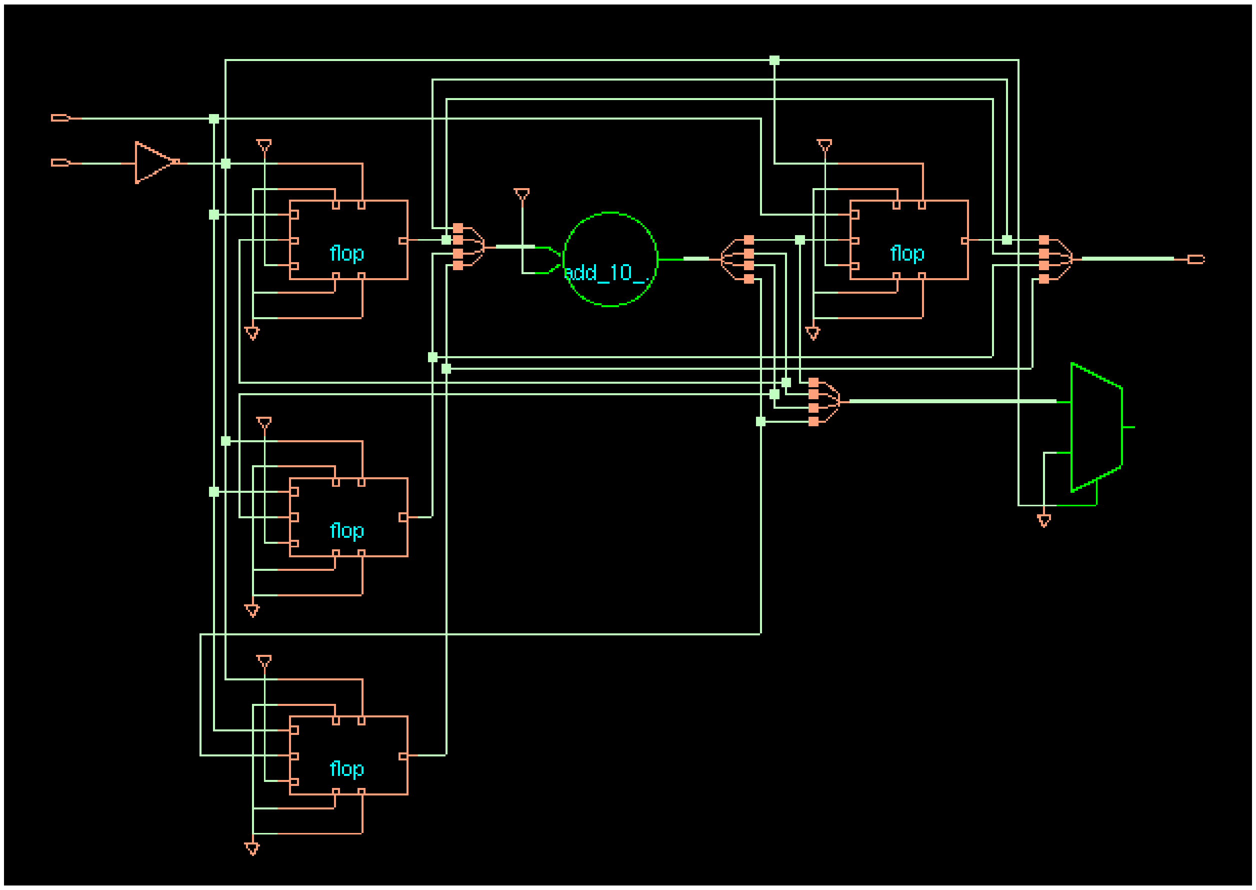
```
module tb_synccounter;
reg   clk,rst;
wire [3:0] cout;
synccounter  dut_synccounter(clk,rst,cout);
initial begin  /*generating the clock stimulus */
clk = 1'b0;
forever #5 clk = ~clk;
end
initial begin
rst = 1'b0;
#20 rst = 1'b1;
#80 rst = 1'b0;  /* this stimulus is to show that
this is synchronous reset and so we cannot reset
it in between the clock */
#5 rst = 1'b1 ;/* we will see in the simulation
that this reset will not be recognized by the
design thus making it a synchronous design*/
#500 $finish;
end
endmodule
```

OUTPUT:

<u>Simulation:</u>



Synthesis:



RESULT: Verilog code for the synchronous counter circuit and its test bench for verification is written, the waveform is observed and the code is synthesized with the technological library and is verified.

9 Successive Approximation Register

Aim: To compile and to simulate the Verilog code for the successive approximation Register.

Program

```
/*SAR program Verilog code-file name
SAR.v*/

module sar
(digitalout,done,comp,start,reset,clk);
output [3:0] digitalout;
output done;
input clk, start, reset, comp;
reg [3:0]ring_count;
reg [3:0]digital;
wire D4,set0,set1,set2,set3;
assign D4 = ring_count[0];
assign done = !D4;
always @(posedge clk or negedge reset)
begin
if (~reset)
ring_count <= 4'b1000;
else
begin
if (start)
ring_count <= 4'b1000;
else
ring_count <= (ring_count>>1);
end
end
assign set3 = ring_count[3];
assign set2 = ring_count[2];
assign set1 = ring_count[1];
assign set0 = ring_count[0];
always @(posedge clk or negedge reset)
begin
if(~reset)
digital[3] <= 1'b1;
else
if(start)
digital[3] <= 1'b1;
else if(set3)
digital[3] <= comp;
end
always @(posedge clk or negedge reset)
```

```
/*SAR program Verilog code for test bench-
file name tb_SAR.v*/

module sar_tb;
reg clk,reset,start;
reg [63:0] vref_d,vin_d;
wire done, comp;
wire [3:0] digitalout;
real vref_real = 7.5;
sar s1 (digitalout,done,comp,start,reset,clk);
dac d1 (comp,digitalout,vref_d,vin_d,clk,start);
initial
begin
clk = 1'b1;
start = 1'b1;
#4000 $finish;
end
always #10 clk = ~clk;
initial
begin
#1;reset = 1'b1;
#10; reset = 1'b0;
#1; reset = 1'b1;
end
initial
begin
#10 ;
stimulus (0.0,0.5,vref_real,8'd5);
end
task stimulus (input analog, input step, input
reference, input [7:0]delay);
real analog,step;
real reference;
begin
while(analog <= reference)
begin
repeat(delay)
@(posedge clk);
start <= 1'b0;
vref_d = $realtobits (reference);
```

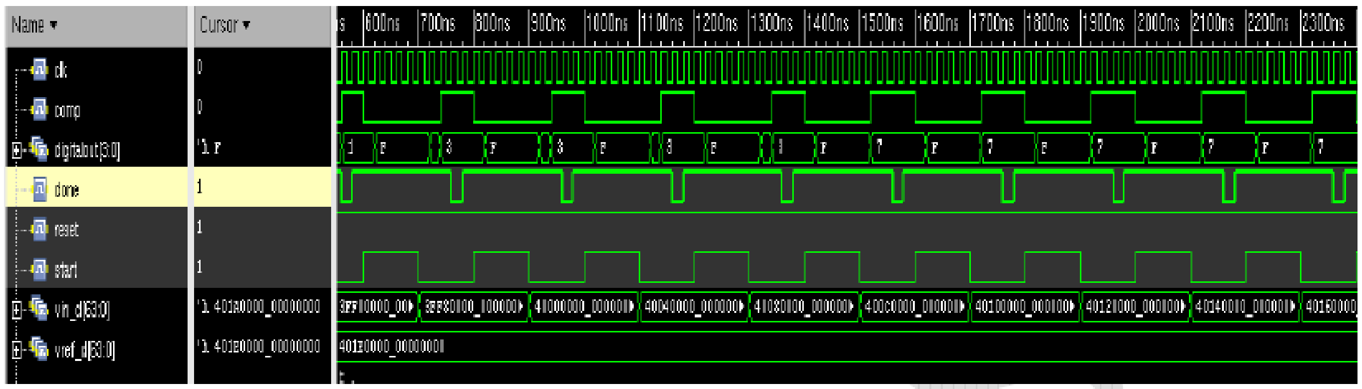
```
begin
if(~reset)
digital[2] <= 1'b1;
else
if(start)
digital[2] <= 1'b1;
else if(set2)
digital[2] <= comp;
end
always @(posedge clk or negedge reset)
begin
if(~reset)
digital[1] <= 1'b1;
else
if(start)
digital[1] <= 1'b1;
else if(set1)
digital[1] <= comp;
end
always @(posedge clk or negedge reset)
begin
if(~reset)
digital[0] <= 1'b1;
else
if(start)
digital[0] <= 1'b1;
else if(set0)
digital[0] <= comp;
end
assign digitalout = (digital) | (ring_count);
endmodule
module dac
(comp,sar_out,vref_d,vin_d,clk,start);
output comp;
input clk,start;
input [3:0]sar_out;
input [63:0]vref_d;
input [63:0]vin_d;
reg comp;
real v_dac,vref,vin;
always @ (vin_d or start)
begin
vref = $bitstoreal(vref_d);
vin = $bitstoreal(vin_d);
end
always @*
begin
```

```
vin_d = $realtobits (analog);
@(posedge done)
analog = analog + step;
@(posedge clk);
start <= 1'b1;
end
end
endtask
endmodule
```

```
if(start)
comp = 1'b0;
else
begin
v_dac = (vref/15)*(sar_out);
if (vin<v_dac)
comp = 1'b0;
else
comp = 1'b1;
end
end
endmodule
```

Output:

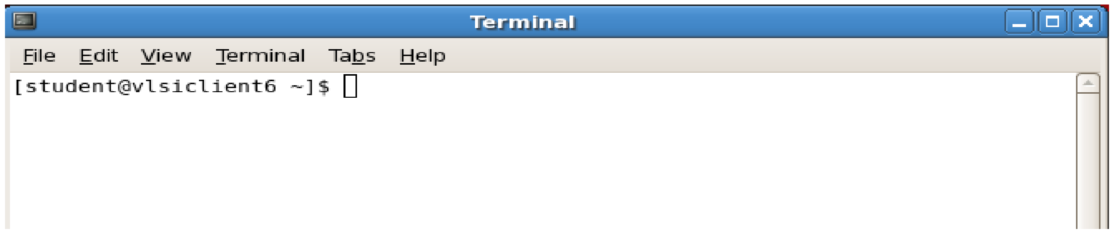
Simulation:



PART B - Analog Design Flow

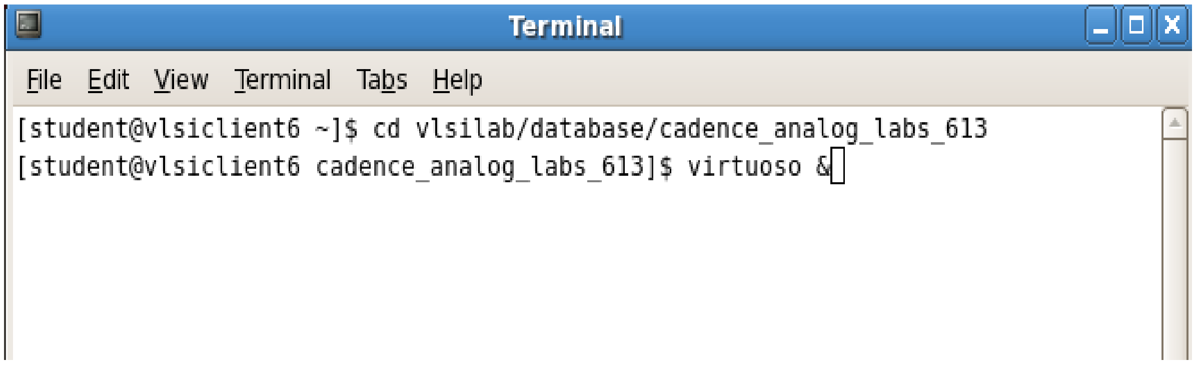
Procedure

1. Open the terminal.



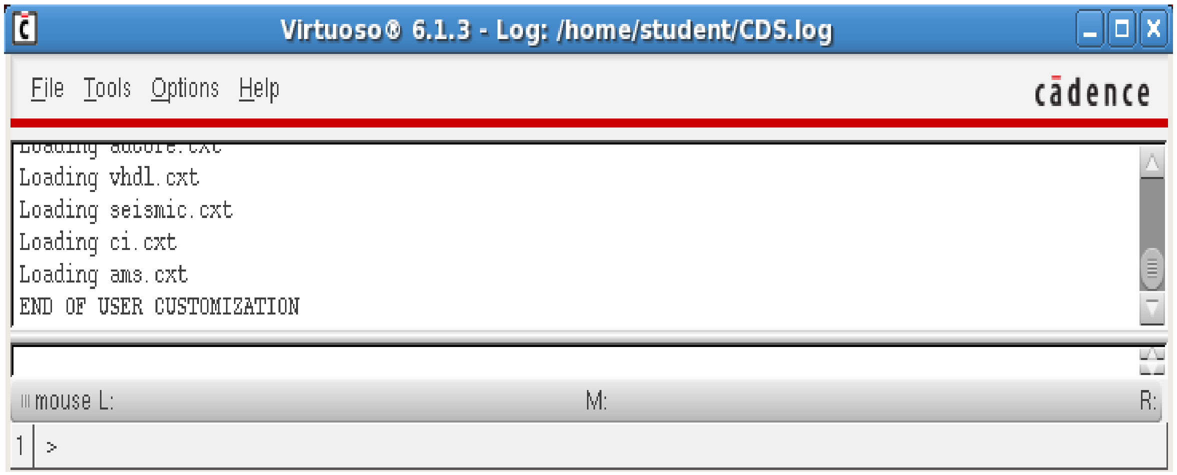
```
[student@vlsiclient6 ~]$
```

2. To invoke the tool, go to the specified directory where the tool is installed to. In the same terminal window, enter:
   - **virtuoso&**



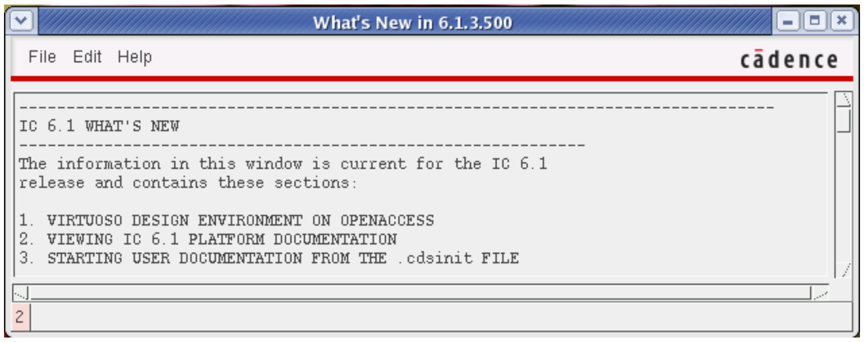
```
[student@vlsiclient6 ~]$ cd vlsilab/database/cadence_analog_labs_613
[student@vlsiclient6 cadence_analog_labs_613]$ virtuoso &
```

The Virtuoso or Command Interpreter window(CIW) appears at the bottom of the screen



```
Loading vhdl.cxt
Loading seismic.cxt
Loading ci.cxt
Loading ams.cxt
END OF USER CUSTOMIZATION
```

3. If the “What’s New ...” window appears, close it with the **File— Close** command.



```
-----------------------------------------------------------------------------------
IC 6.1 WHAT'S NEW
-------------------------------------------------------------
The information in this window is current for the IC 6.1
release and contains these sections:

1. VIRTUOSO DESIGN ENVIRONMENT ON OPENACCESS
2. VIEWING IC 6.1 PLATFORM DOCUMENTATION
3. STARTING USER DOCUMENTATION FROM THE .cdsinit FILE
```

4. Keep opened CIW window for the labs.
5. The constraint file discussed in Part A can be considered for Analog Design flow.

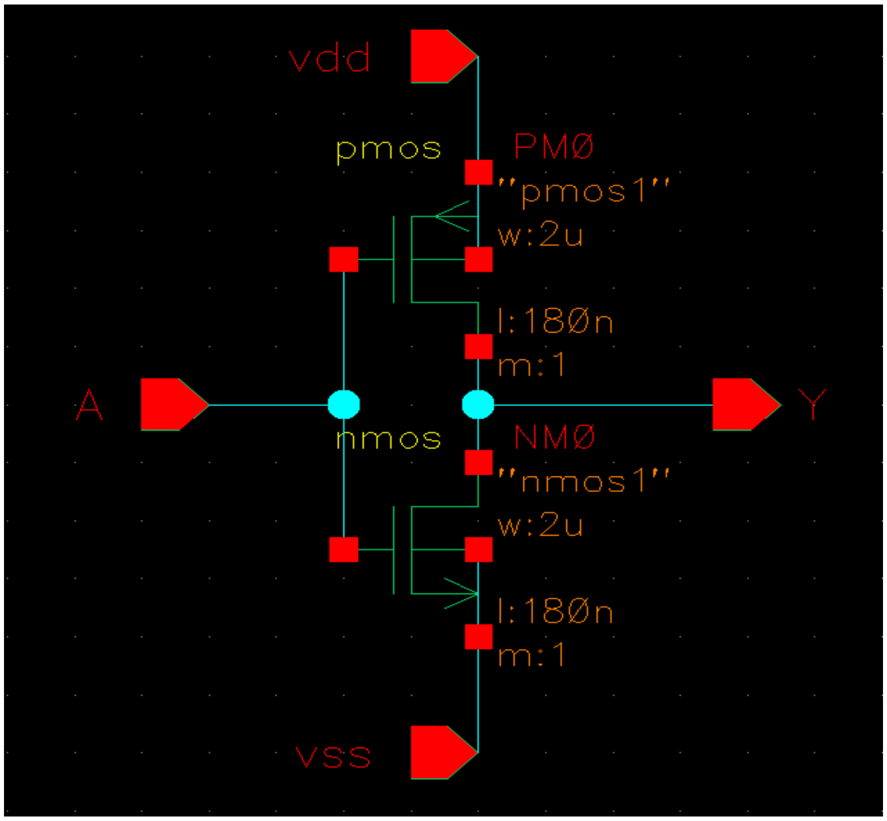
Lab 1: AN INVERTER

AIM : To simulate the schematic of the CMOS inverter, and then to perform the physical verification for the layout of the same.

TOOL REQUIRED : Cadence Tool

THEORY: The inverter is universally accepted as the most basic logic gate doing a Boolean operation on a single input variable. Fig.1 depicts the symbol, truth table and a general structure of a CMOS inverter. As shown, the simple structure consists of a combination of an pMOS transistor at the top and a nMOS transistor at the bottom.

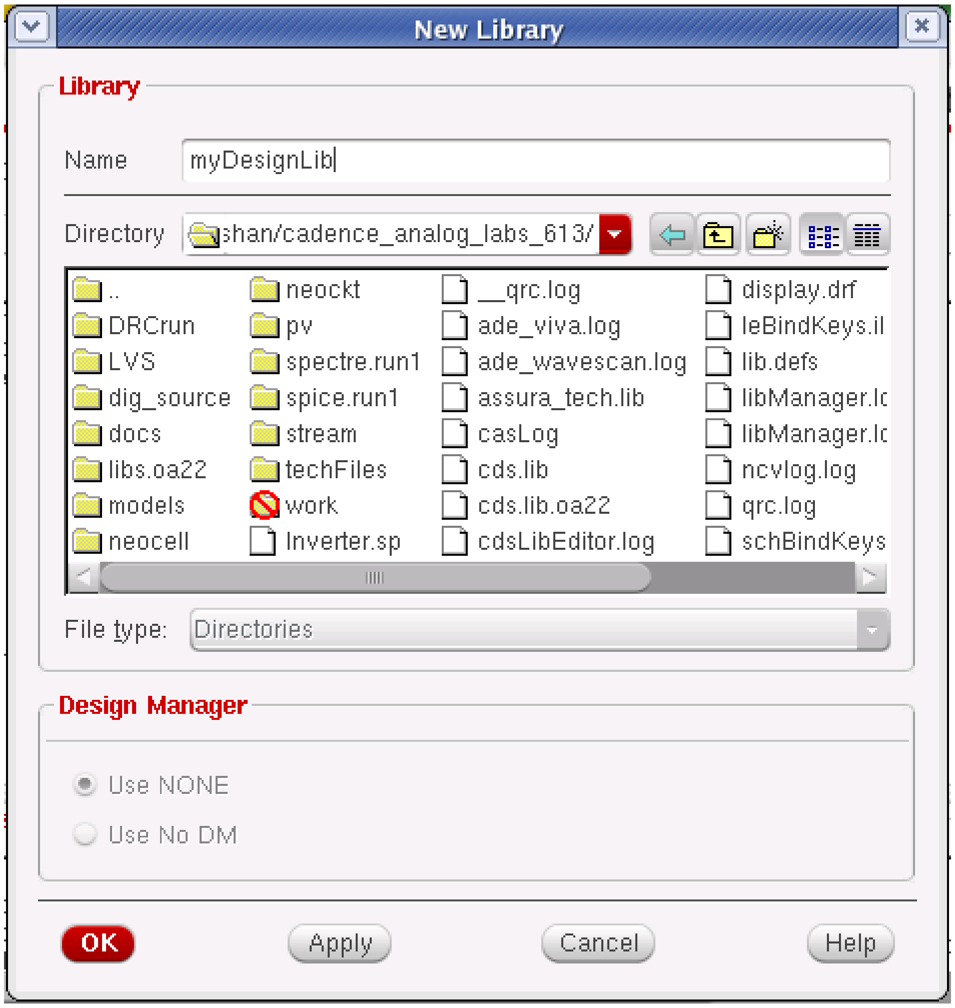
Schematic Capture



Schematic Entry

Creating a New library

1. In the Library Manager, execute **File - New – Library**. The new library form appears.
2. In the “New Library” form, type “**myDesignLib**” in the Name section.



Name: myDesignLib

Directory: shan/cadence_analog_labs_613/

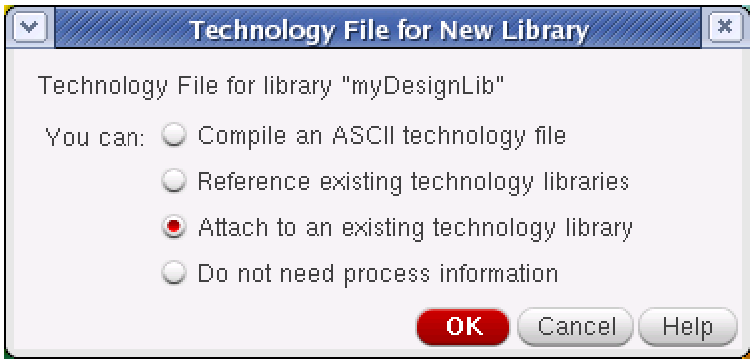
..	neockt	__qrc.log	display.drf
DRCrun	pv	ade_viva.log	leBindKeys.il
LVS	spectre.run1	ade_wavescan.log	lib.defs
dig_source	spice.run1	assura_tech.lib	libManager.lc
docs	stream	casLog	libManager.lc
libs.oa22	techFiles	cds.lib	ncvlog.log
models	work	cds.lib.oa22	qrc.log
neocell	Inverter.sp	cdsLibEditor.log	schBindKeys

File type: Directories

[✓ Use NONE] Use No DM

3. In the field of Directory section, verify that the path to the library is set to~/**Database/cadence_analog_labs_613** and click **OK**.

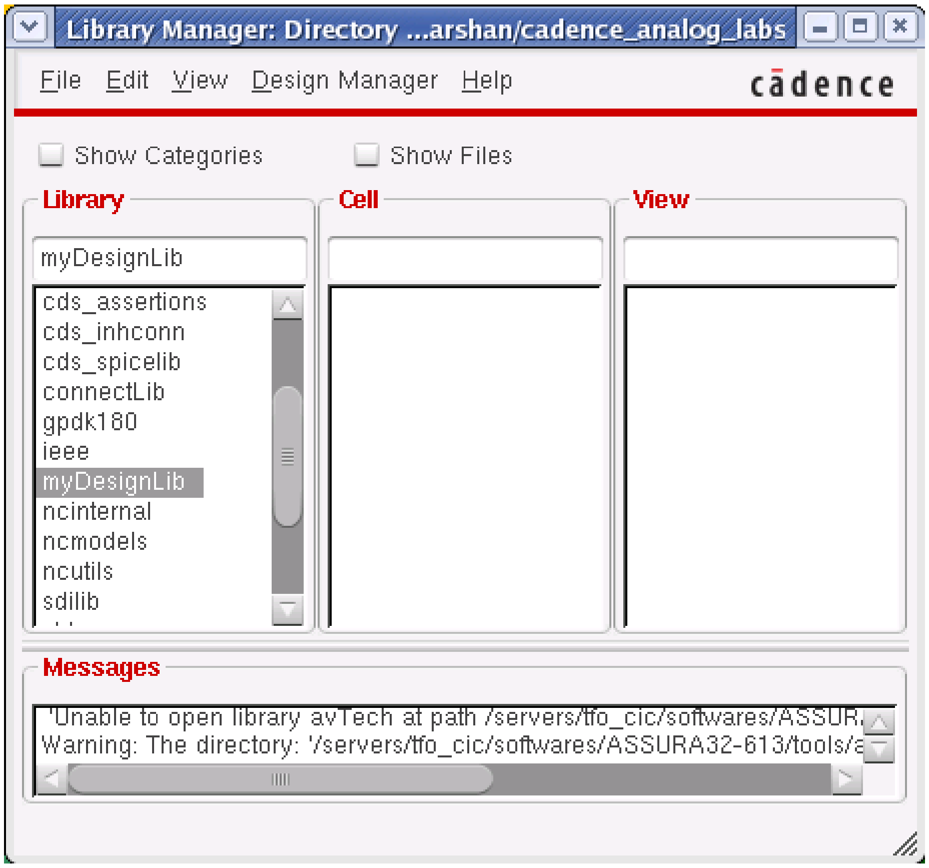
4. In the next "**Technology File for New library**" form, select option **Attach to an existing techfile** and click **OK**.



Technology File for library "myDesignLib"

You can:
- Compile an ASCII technology file
- Reference existing technology libraries
- [✓ Attach to an existing technology library]
- Do not need process information

5. In the "**Attach Design Library to Technology File**" form, select **gpdk180** from the cyclic field and click **OK**.
6. After creating a new library you can verify it from the library manager.
7. If you right click on the "**myDesignLib**" and select properties, you will find that **gpdk180** library is attached as techlib to "**myDesignLib**".

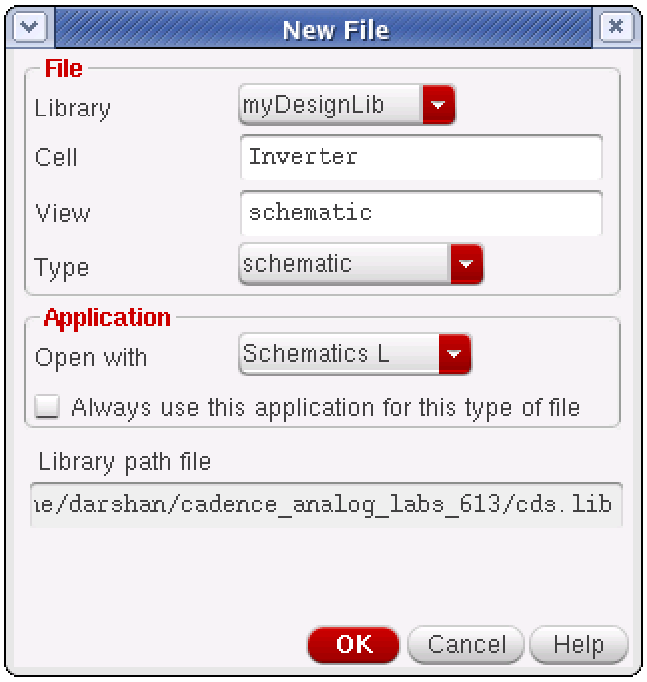


```
'Unable to open library avTech at path /servers/tfo_cic/softwares/ASSUR
Warning: The directory: '/servers/tfo_cic/softwares/ASSURA32-613/tools/a
```

Creating a Schematic Cellview

In this section we will learn how to open new schematic window in the new "**myDesignLib**" library and build the inverter schematic as shown in the figure at the start of this lab.

1. In the CIW or Library manager, execute **File – New – Cellview**.
2. Set up the New file form as follows:



3. Click **OK** when done the above settings. A blank schematic window for the **Inverter** design appears.

Adding Components to schematic

1. In the Inverter schematic window, click the **Instance** fixed menu icon to display the Add Instance form.
Tip: You can also execute **Create — Instance** or press **i**.

2. Click on the **Browse** button. This opens up a Library browser from which you can select components and the **symbol** view .

3. After completing the Add Instance form, move cursor to the schematic window and click **left** to place a component.

This is a table of components for building the Inverter schematic.

Library name	Cell Name	Properties/Comments
gpdk180	pmos	For M0: Model name = pmos1, **W= wp**, L=180n
gpdk180	nmos	For M1: Model name = nmos1, W= 2u, L=180n

4. After entering components, click **Cancel** in the Add Instance form or press **Esc** with your cursor in the schematic window.

Adding pins to Schematic

1. Click the **Pin** fixed menu icon in the schematic window.

You can also execute **Create — Pin** or press ***p***.
The Add pin form appears.

2. Type the following in the Add pin form in the exact order leaving space between the pin names.

Pin Names	Direction
vin	Input
vout	Output

3.Select**Cancel** from the Add – pin form after placing the pins.

Adding Wires to a Schematic

Add wires to connect components and pins in the design.

1. Click the **Wire (narrow)** icon in the schematic window.

You can also press the ***w*** key, or execute **Create — Wire (narrow).**

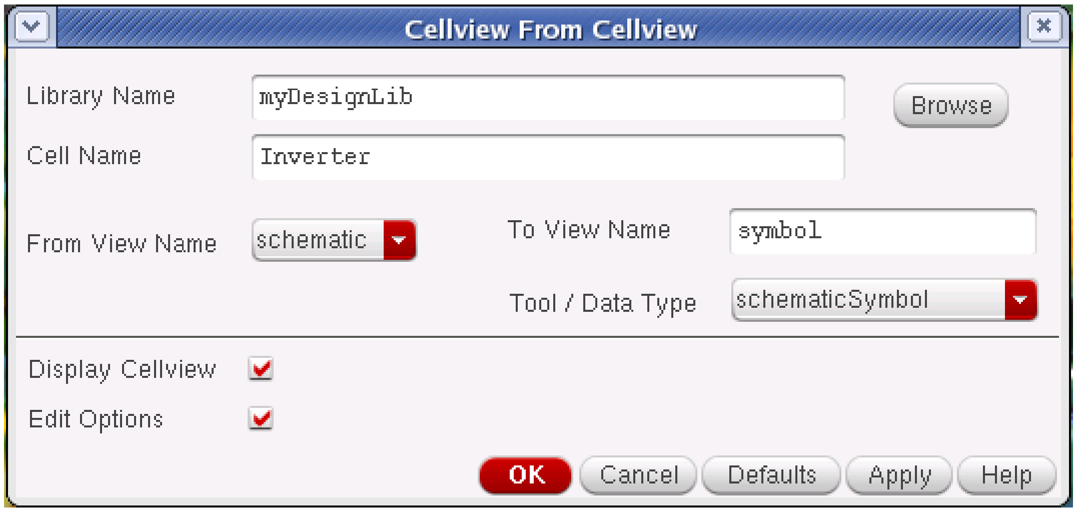
2. In the schematic window, click on a pin of one of your components as the first point for your wiring. A diamond shape appears over the starting point of this wire.
3. Follow the prompts at the bottom of the design window and click **left** on the destination point for your wire. A wire is routed between the source and destination points.
4. Complete the wiring as shown in figure and when done wiring press **ESC** key in the schematic window to cancel wiring

Saving the Design

1. Click the ***Check and Save*** icon in the schematic editor window.
2. Observe the CIW output area for any errors.

Symbol Creation

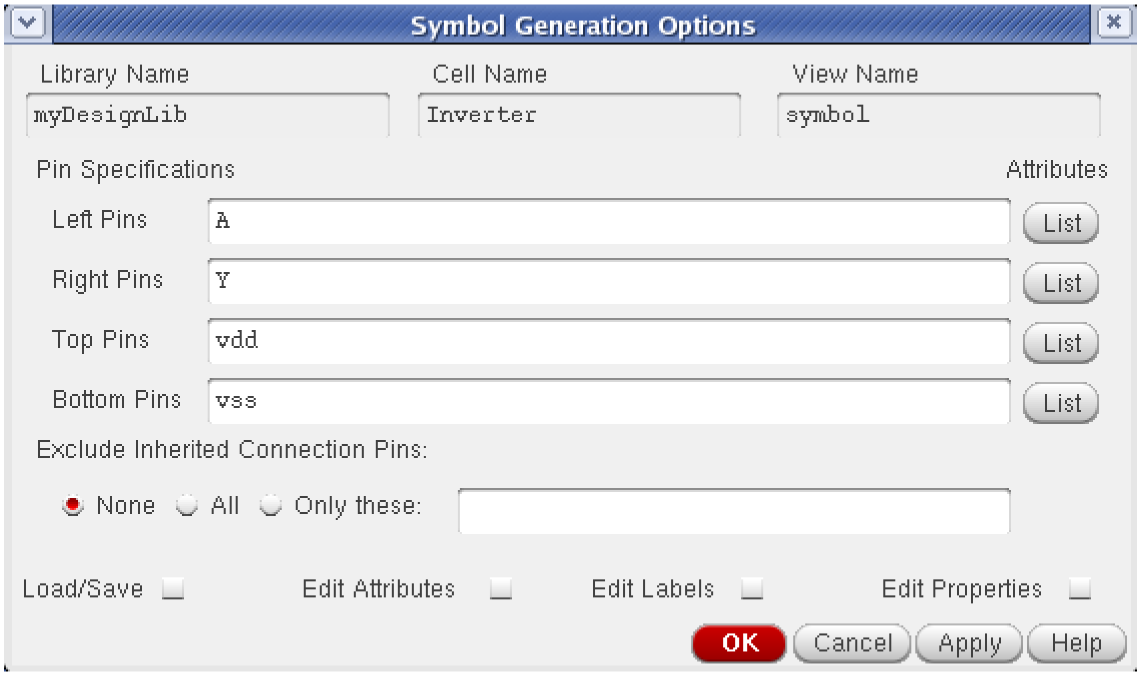
1. In the Inverter schematic window, execute **Create — Cellview— From Cellview**. The **CellviewFromCellview**form appears. With the Edit Options function active, you can control the appearance of the symbol to generate.
2. Verify that the **From View Name** field is set to **schematic**, and the **To View Name** field is set to **symbol**, with the **Tool/Data Type** set as **SchematicSymbol**.



Library Name	myDesignLib
Cell Name	Inverter
From View Name	schematic
To View Name	symbol
Tool / Data Type	schematicSymbol
Display Cellview	✓
Edit Options	✓

3. Click **OK** in the **CellviewFromCellview**form.
The Symbol Generation Form appears.

4.Modify the **Pin Specifications** as follows:



Library Name	Cell Name	View Name
myDesignLib	Inverter	symbol

Pin Specifications

Left Pins	A
Right Pins	Y
Top Pins	vdd
Bottom Pins	vss

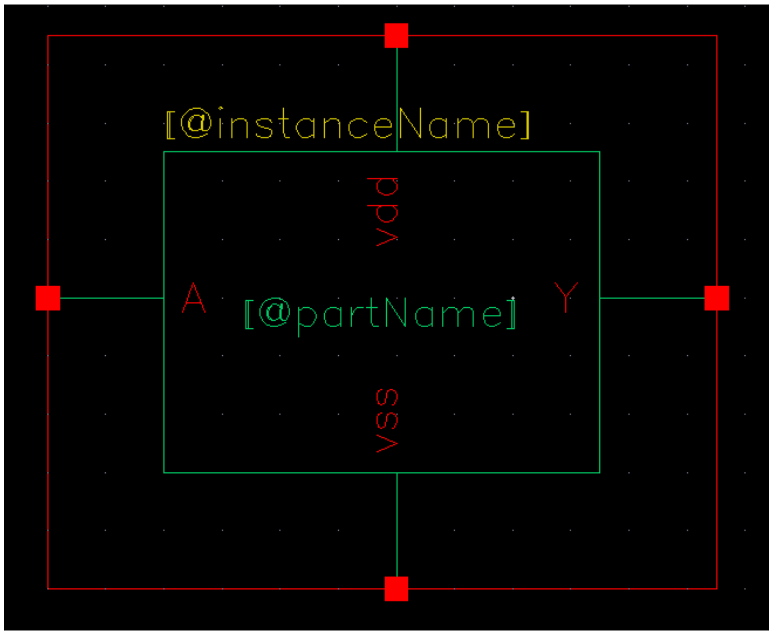
Exclude Inherited Connection Pins:

[✓ None] All Only these:

Load/Save ☐ Edit Attributes ☐ Edit Labels ☐ Edit Properties ☐

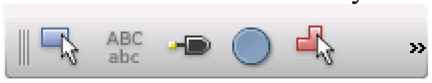
5. Click **OK** in the Symbol Generation Options form.

6. A new window displays an automatically created Inverter symbol as shown here.



Editing a Symbol

In this section we will modify the inverter symbol to look like a Inverter gate symbol.



1. Move the cursor over the automatically generated symbol, until the green rectangle is highlighted, click **left** to select it.

2. Click **Delete** icon in the symbol window, similarly select the red rectangle and delete that.

3. Execute **Create – Shape – polygon**, and draw a shape similar to triangle.

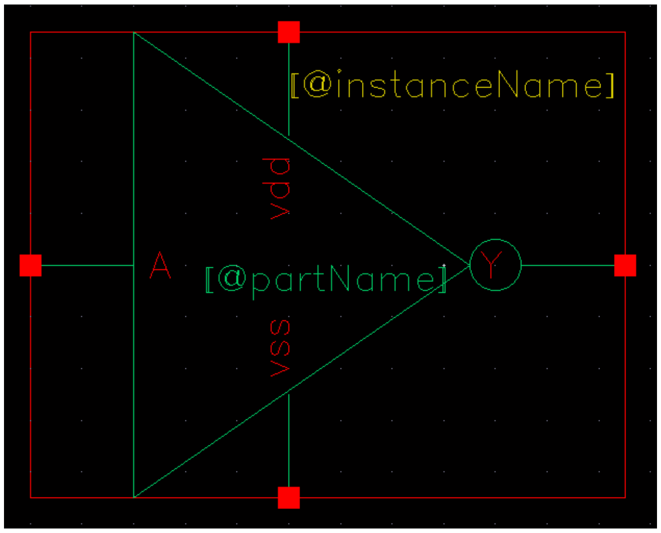
4. After creating the triangle press ***ESC*** key.

5. Execute **Create – Shape – Circle** to make a circle at the end of triangle.

6. You can move the pin names according to the location.

7. Execute **Create — Selection Box**. In the Add Selection Box form, click **Automatic**. A new red selection box is automatically added.

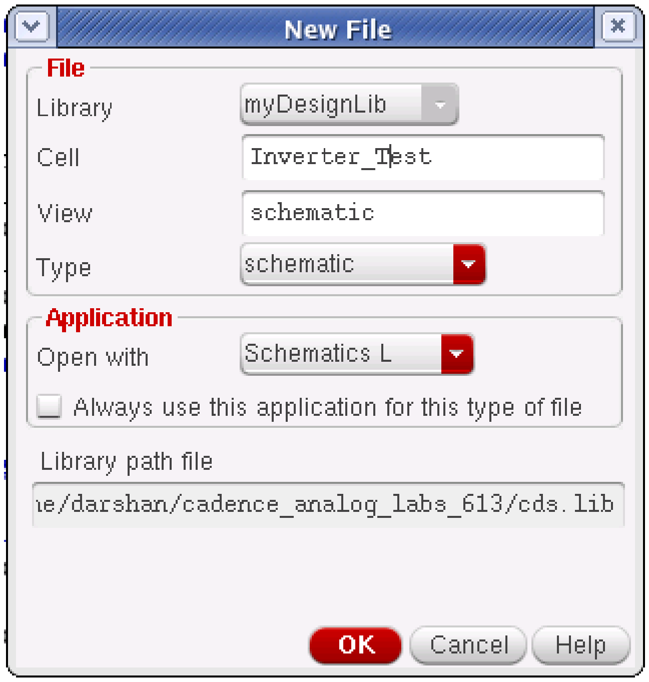
8. After creating symbol, click on the ***save*** icon in the symbol editor window to save the symbol. In the symbol editor, execute **File — Close** to close the symbol view window.



Building the Inverter_Test Design

Creating the Inverter_TestCellview

1. In the CIW or Library Manager, execute **File— New— Cellview**.
2. Set up the New File form as follows:



File
Library myDesignLib
Cell Inverter_Test
View schematic
Type schematic

Application
Open with Schematics L
Always use this application for this type of file

Library path file
ıe/darshan/cadence_analog_labs_613/cds.lib

3. Click **OK** when done. A blank schematic window for the **Inverter_Test**design appears.

Building the Inverter_TestCircuit
1. Using the component list and Properties/Comments in this table,
build the **Inverter_Test**schematic.

Library name	Cellview name	Properties/Comments
myDesignLib	Inverter	Symbol
analogLib	vpulse	v1=0, v2=1.8,td=0 tr=tf=1ns, ton=10n, T=20n
analogLib	vdc, gnd	vdc=1.8

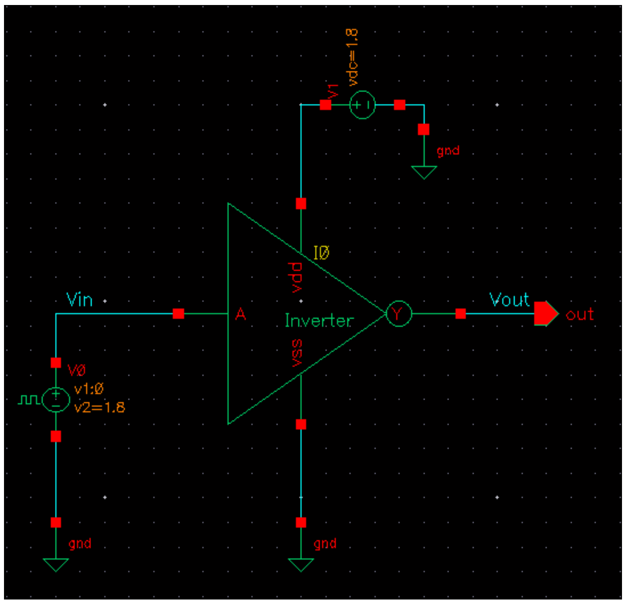
2. Add the above components using **Create — Instance** or by pressing **I**.

3. Click the **Wire (narrow)** icon and wire your schematic

4.Click**Create — Wire Name** or press **L** to name the input **(Vin)** and output **(Vout)** wires as in the below schematic.

5.Click on the ***Check and Save*** icon to save the design

6.The schematic should look like this.



7. Leave your **Inverter_Test**schematic window open for the next section.

Analog Simulation with Spectre

Starting the Simulation Environment

Start the Simulation Environment to run a simulation.
1. In the **Inverter_Test** schematic window, execute
Launch – ADE L

The **Virtuoso Analog Design Environment (ADE)** simulation window appears.

Choosing a Simulator

Set the environment to use the **Spectre® tool**, a high speed, highly accurate analog simulator. Use this simulator with the **Inverter*_Test*** design, which is made-up of analog components.

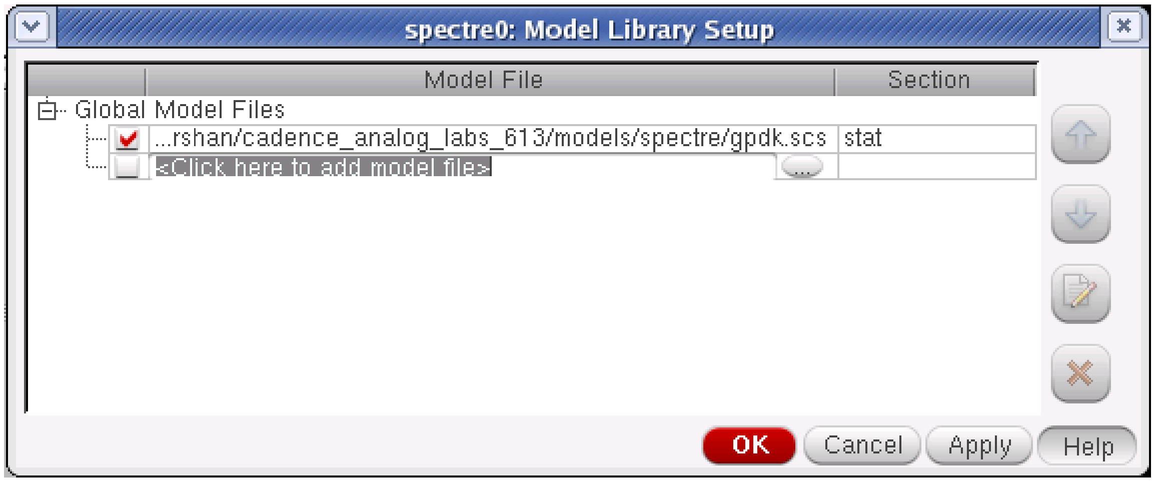
1. In the simulation window (ADE), execute
Setup— Simulator/Directory/Host.

2. In the Choosing Simulator form, set the Simulator field to **spectre**
(Not spectreS) and click **OK**.

Setting the Model Libraries

The Model Library file contains the model files that describe the nmos and pmos devices during simulation.

1. In the simulation window (ADE),
Execute **Setup - Model Libraries.**
The Model Library Setup form appears. Click the **browse** button to add **gpdk.scs** if not added by default as shown in the **Model Library Setup** form.
Remember to select the section type as **stat** in front of the gpdk.scsfile. Your Model Library Setup window should now looks like the below figure.



spectre0: Model Library Setup

Model File	Section
Global Model Files	
✓ ...rshan/cadence_analog_labs_613/models/spectre/gpdk.scs	stat
<Click here to add model file>	

To view the model file, highlight the expression in the Model Library File field and

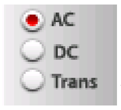
Click **Edit File**.

2. To complete the Model Library Setup, move the cursor and click **OK**.

The Model Library Setup allows you to include multiple model files. It also allows you to use the Edit button to view the model file.

Choosing Analyses

This section demonstrates how to view and select the different types of analyses to complete the circuit when running the simulation.

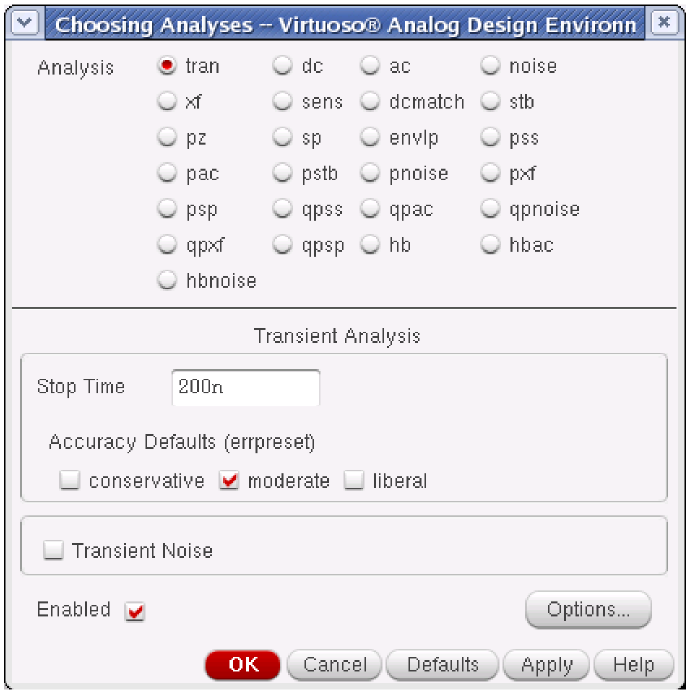
1. In the Simulation window (ADE), click the **Choose - Analyses** icon. 

You can also execute **Analyses - Choose.**

The Choosing Analysis form appears. This is a dynamic form, the bottom of the form changes based on the selection above.

2. To setup for transient analysis

a. In the Analysis section select **tran**
b. Set the stop time as **200n**
c. Click at the **moderate** or **Enabled** button at the bottom, and then click **Apply**.



Analysis: [✓ tran] xf pz pac psp qpxf hbnoise dc sens sp pstb qpss qpsp ac dcmatch envlp pnoise qpac hb noise stb pss pxf qpnoise hbac

Transient Analysis

Stop Time 200n

Accuracy Defaults (errpreset)

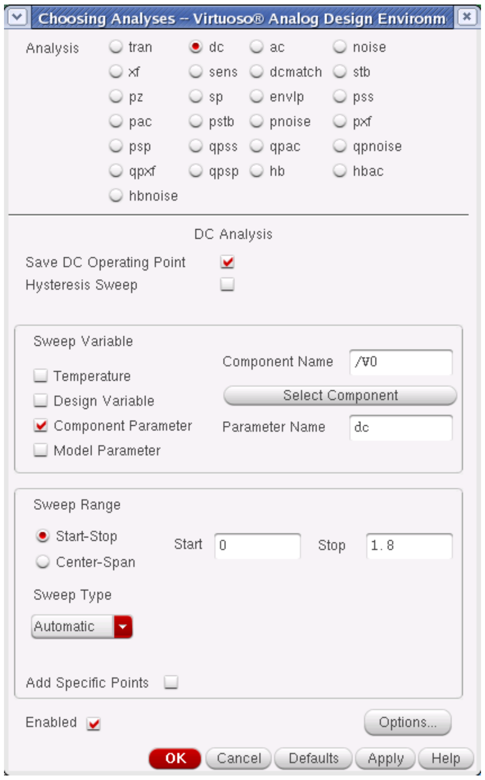
conservative [✓ moderate] liberal

Transient Noise

Enabled [✓]

3. To set up for DC Analyses:
   a. In the Analyses section, select **dc**.
   b. In the DC Analyses section, turn on **Save DC Operating Point.**
   c. Turn on the **Component Parameter**.
   d. Double click the **Select Component**, Which takes you to the schematic window.
   e. Select input signal **vpulse source** in the test schematic window.

f. Select **“DC Voltage”** in the **Select Component Parameter** form and click OK.
f. In the analysis form type **start** and **stop** voltages as **0** to **1.8** respectively.
g. Check the enable button and then click **Apply.**


Choosing Analyses -- Virtuoso® Analog Design Environm

Analysis: tran [● dc] ac noise xf sens dcmatch stb pz sp envlp pss pac pstb pnoise pxf psp qpss qpac qpnoise qpxf qpsp hb hbac hbnoise

DC Analysis

Save DC Operating Point ✓
Hysteresis Sweep ☐

Sweep Variable
- Temperature ☐
- Design Variable ☐
- Component Parameter ✓
- Model Parameter ☐

Component Name: /V0
Select Component
Parameter Name: dc

Sweep Range
[● Start-Stop] Center-Span
Start: 0 Stop: 1.8

Sweep Type: Automatic

Add Specific Points ☐

Enabled ✓

4.Click***OK*** in the Choosing Analyses Form.

Setting Design Variables

Set the values of any design variables in the circuit before simulating.
Otherwise, the simulation will not run.

1. In the Simulation window, click the **Edit Variables** icon.
The Editing Design Variables form appears.
2. Click **Copy From** at the bottom of the form.
The design is scanned and all variables found in the design are listed.
In a few moments, the **wp**variable appears in the Table of Design variables section.

3. Set the value of the **wp**variable:
With the **wp**variable highlighted in the Table of Design Variables,

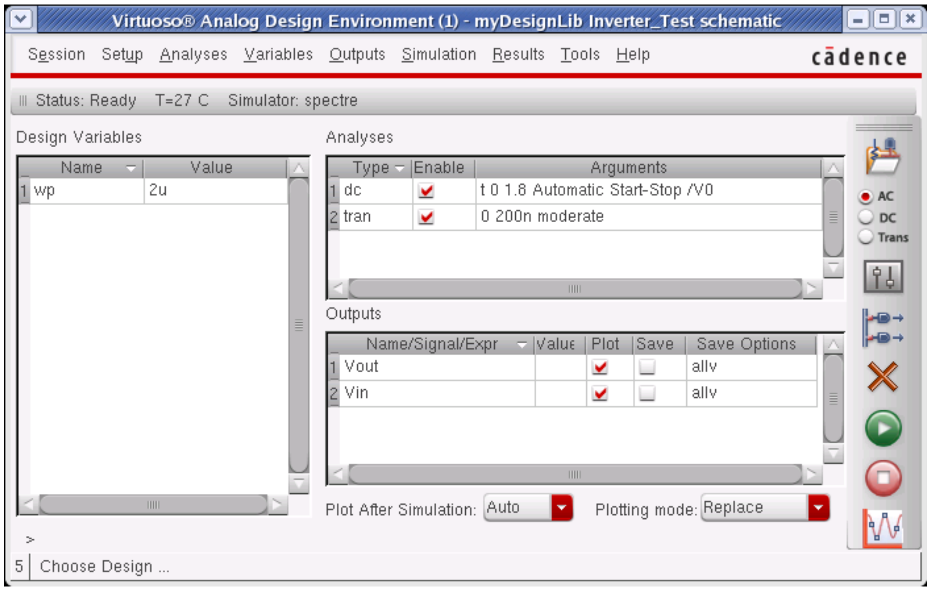
Value(Expr)	2u

Click **Change** and notice the update in the Table of Design Variables.

4. Click **OK** or **Cancel** in the Editing Design Variables window.

Selecting Outputs for Plotting

1. Execute **Outputs – To be plotted – Select on Schematic** in the simulation window.
2. Follow the prompt at the bottom of the schematic window, Click on output net **Vout**, input net **Vin** of the Inverter. Press **ESC** with the cursor in the schematic after selecting it.


Virtuoso® Analog Design Environment (1) - myDesignLib Inverter_Test schematic
Session Setup Analyses Variables Outputs Simulation Results Tools Help
cādence
AC
DC
Trans
Choose Design ...

Status: Ready T=27 C Simulator: spectre

Design Variables

	Name	Value
1	wp	2u

Analyses

	Type	Enable	Arguments
1	dc	✓	t 0 1.8 Automatic Start-Stop /V0
2	tran	✓	0 200n moderate

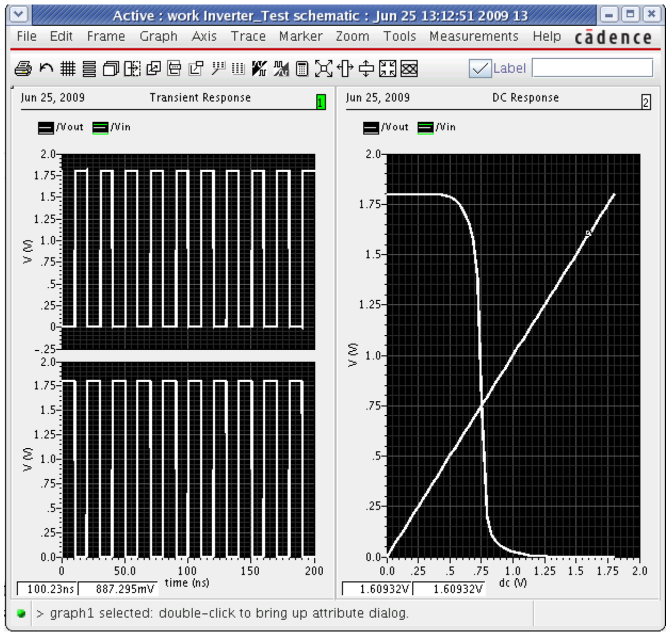
Outputs

	Name/Signal/Expr	Value	Plot	Save	Save Options
1	Vout		✓		allv
2	Vin		✓		allv

Plot After Simulation: Auto Plotting mode: Replace

Running the Simulation

1. Execute **Simulation – Netlist and Run** in the simulation window to start the Simulation or the icon, this will create the netlist as well as run the simulation.
2. When simulation finishes, the Transient, DC plots automatically will be popped up along with log file.


Active : work Inverter_Test schematic : Jun 25 13:12:51 2009 13
File Edit Frame Graph Axis Trace Marker Zoom Tools Measurements Help
cādence
Label
Jun 25, 2009
Transient Response
/Vout
/Vin
V (V)
time (ns)
100.23ns
887.295mV
Jun 25, 2009
DC Response
dc (V)
1.60932V
1.60932V
> graph1 selected: double-click to bring up attribute dialog.

Saving the Simulator State

We can save the simulator state, which stores information such as model library file, outputs, analysis, variable etc. This information restores the simulation environment without having to type in all of setting again.

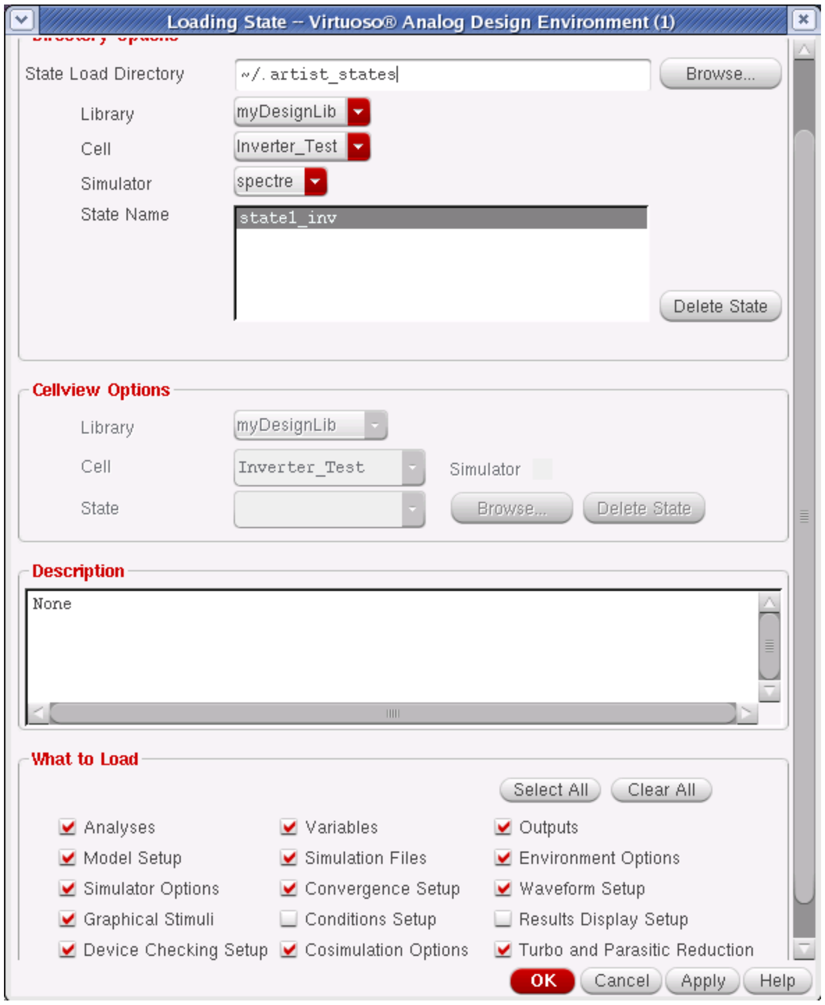
1. In the Simulation window, execute **Session – Save State**.
The Saving State form appears.

2. Set the **Save as** field to **state1_inv** and make sure all options are selected under what to save field.

3. Click **OK** in the saving state form. The Simulator state is saved.

Loading the Simulator State

1. From the ADE window execute **Session – Load State.** In the Loading State window, set the State name to **state1_inv** as shown



Loading State -- Virtuoso® Analog Design Environment (1)

State Load Directory: ~/.artist_states
Library: myDesignLib
Cell: Inverter_Test
Simulator: spectre
State Name: state1_inv

Cellview Options
Library: myDesignLib
Cell: Inverter_Test
State:

Description
None

What to Load
- [x] Analyses
- [x] Variables
- [x] Outputs
- [x] Model Setup
- [x] Simulation Files
- [x] Environment Options
- [x] Simulator Options
- [x] Convergence Setup
- [x] Waveform Setup
- [x] Graphical Stimuli
- [ ] Conditions Setup
- [ ] Results Display Setup
- [x] Device Checking Setup
- [x] Cosimulation Options
- [x] Turbo and Parasitic Reduction

2. Click **OK** in the Loading State window.

Parametric Analysis

Parametric Analysis yields information similar to that provided by the Spectre® sweep feature, except the data is for a full range of sweeps for each parametric step. The Spectre sweep feature provides sweep data at only one specified condition.

You will run a parametric DC analysis on the **wp**variable, of the PMOS device of the Inverter design by sweeping the value of **wp**.

Run a simulation before starting the parametric tool. You will start by loading the state from the previous simulation run.

Run the simulation and check for errors. When the simulation ends, a single waveform in the waveform window displays the DC Response at the **Vout** node.

Starting the Parametric Analysis Tool

1. In the Simulation window, execute **Tools—Parametric Analysis**.
The Parametric Analysis form appears.

2. In the Parametric Analysis form, execute
Setup—Pick Name For Variable—Sweep 1.

A selection window appears with a list of all variables in the design that you can sweep. This list includes the variables that appear in the Design Variables section of the Simulation window.

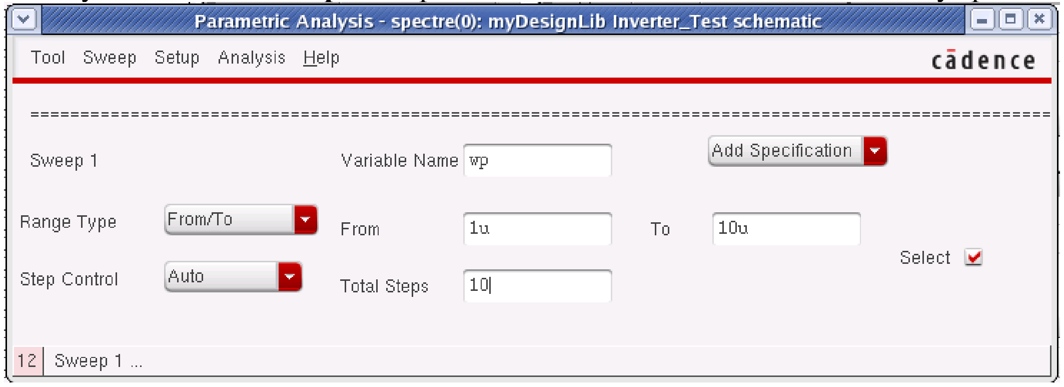
3. In the selection window, double click left on **wp**. The Variable Name field for Sweep 1 in the Parametric Analysis form is set to **wp**.

4. Change the Range Type and Step Control fields in the Parametric Analysis form as shown below:

Range Type **From/To** From **1u** To**10u**

Step Control **Auto**Total Steps **10**

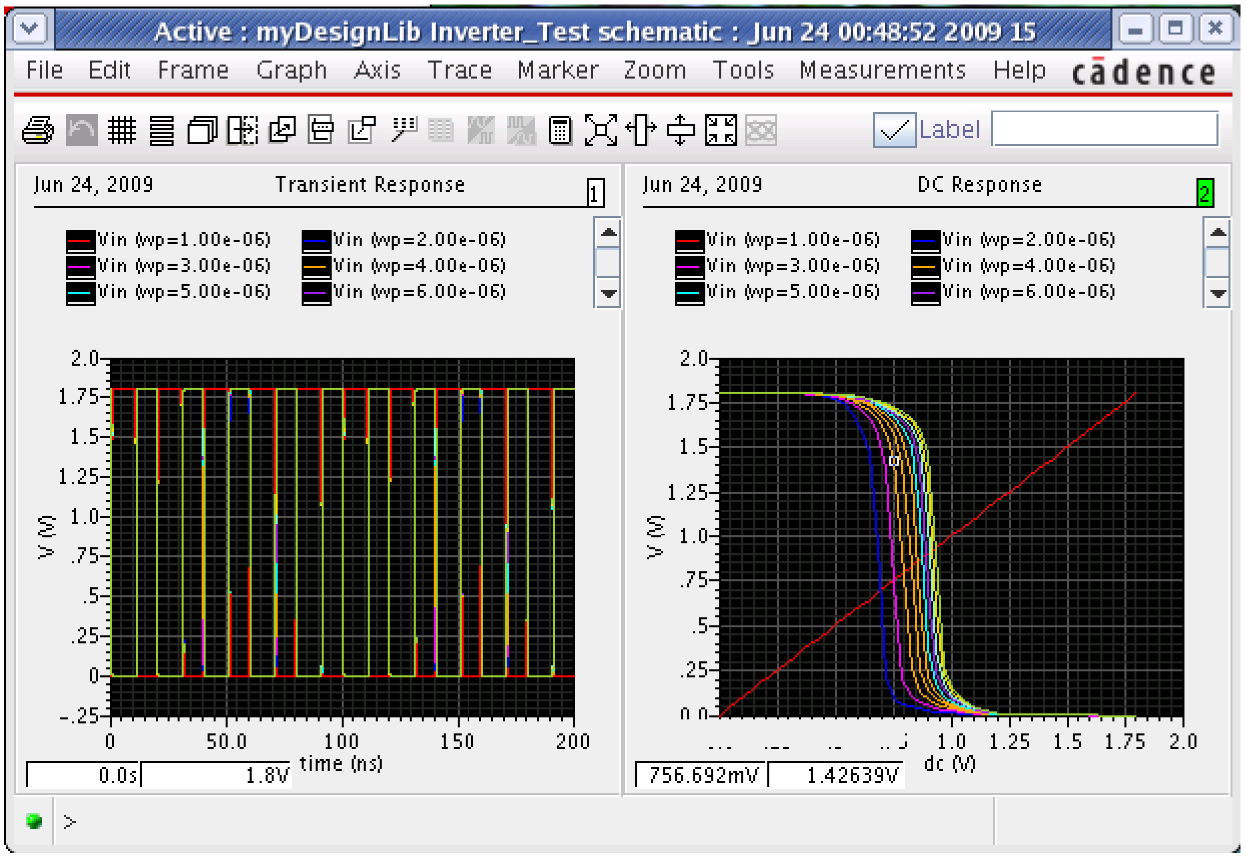
These numbers vary the value of the **wp**of the pmos between 1um and 10um at ten evenly spaced intervals.


Parametric Analysis - spectre(0): myDesignLib Inverter_Test schematic
Tool Sweep Setup Analysis Help
cādence
Add Specification
12 Sweep 1 ...

Sweep 1
Variable Name wp
Range Type From/To
From 1u
To 10u
Step Control Auto
Total Steps 10
Select ✓

5.Execute **Analysis—Start**.

The Parametric Analysis window displays the number of runs remaining in the analysis and the current value of the swept variable(s). Look in the upper right corner of the window. Once the runs are completed the waves can window comes up with the plots for different runs.



Creating Layout View of Inverter

1. From the **Inverter** schematic window menu execute
Launch – Layout XL. A **Startup Option** form appears.
2. Select ***Create New*** option. This gives a New Cell View Form
3. Check the Cellname**(Inverter)**, Viewname**(layout).**
4. Click **OK** from the New Cellview form.
LSW and a blank layout window appear along with schematic window

Adding Components to Layout

1. Execute **Connectivity – Generate – All from Source** or click the icon in the layout editor window, **Generate Layout** form appears. Click **OK** which imports the schematic components in to the Layout window automatically.
2. Re arrange the components with in PR-Boundary as shown in the next page.
3. To rotate a component, Select the component and execute **Edit –Properties**. Now select the degree of rotation from the property edit form.

4. To Move a component, Select the component and execute **Edit -Move** command.

Making interconnection

1. Execute **Connectivity –Nets – Show/Hide selected Incomplete Nets** or click the icon in the Layout Menu.

2. Move the mouse pointer over the device and click **LMB** to get the connectivity information, which shows the guide lines (or flight lines) for the inter connections of the components.

3. From the layout window execute **Create – Shape – Path/ Create wire** or **Create – Shape – Rectangle** (for vdd and gnd bar**)** and select the appropriate Layers from the **LSW** window and Vias for making the inter connections

Creating Contacts/Vias

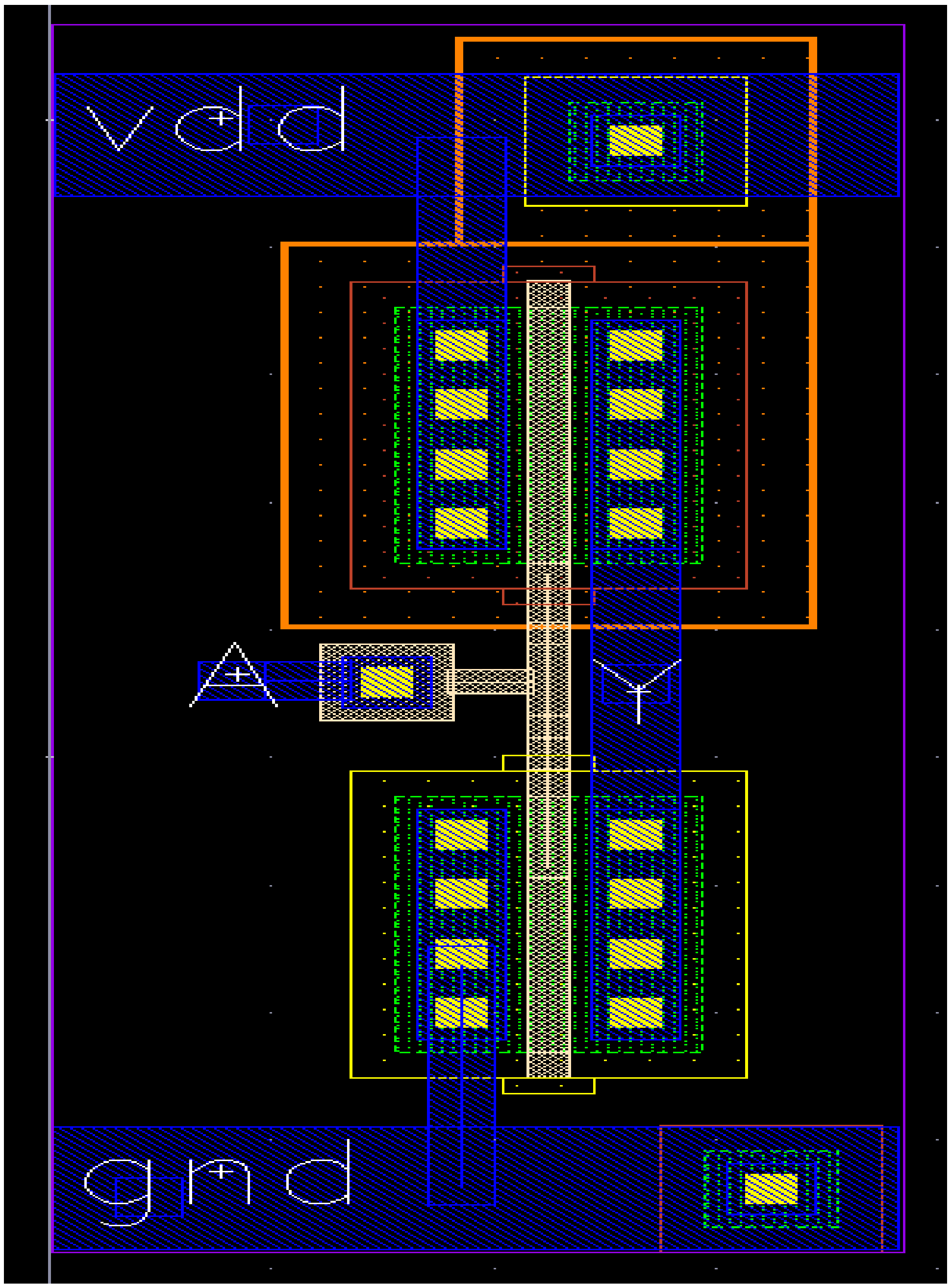
You will use the contacts or vias to make connections between two different layers.

1. Execute **Create-Via** or select command to place different Contacts, as given in below table

Connection	Contact Type
For Metal1- Poly Connection	Metal1-Poly
For Metal1- Psubstrate Connection	Metal1-Psub
For Metal1- Nwell Connection	Metal1-Nwell

Saving the design

1. Save your design by selecting **File — Save**or click 💾 to save the layout, and layout should appear as below.

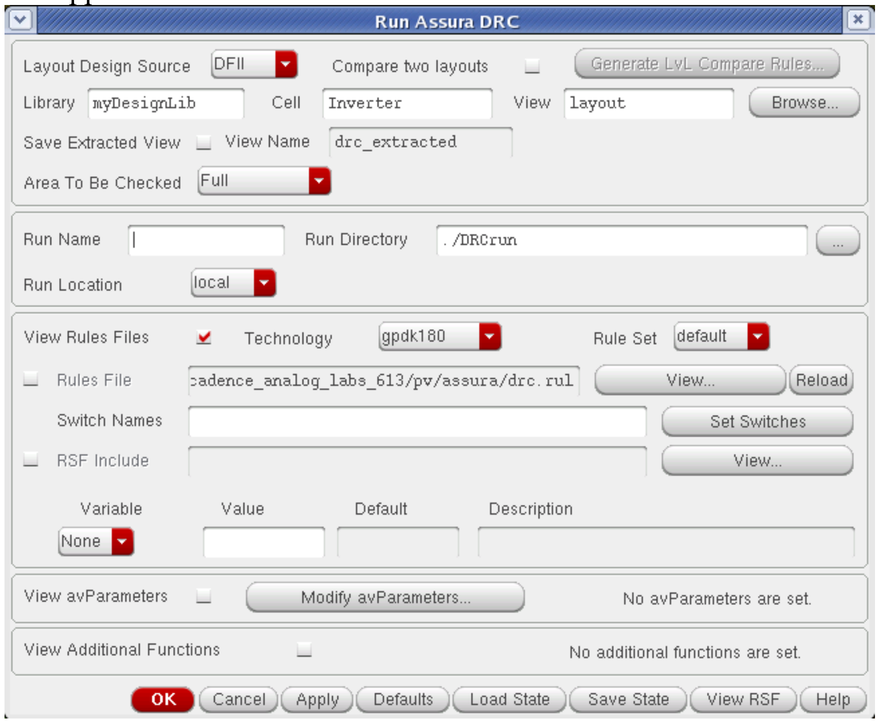


Physical Verification

Assura DRC

Running a DRC

1. Open the Inverter layout form the CIW or library manger if you have closed that.
Press **shift – f** in the layout window to display all the levels.
2. Select **Assura - Run DRC** from layout window.
The DRC form appears. The Library and Cellname are taken from the current design window, but rule file may be missing. Select the Technology as **gpdk180.** This automatically loads the rule file.
Your DRC form should appear like this



Run Assura DRC

Layout Design Source: DFII — Compare two layouts: ☐ — Generate LvL Compare Rules...
Library: myDesignLib — Cell: Inverter — View: layout — Browse...
Save Extracted View: ☐ — View Name: drc_extracted
Area To Be Checked: Full

Run Name: — Run Directory: ./DRCrun
Run Location: local

View Rules Files: ✓ — Technology: gpdk180 — Rule Set: default
☐ Rules File: cadence_analog_labs_613/pv/assura/drc.rul — View... — Reload
Switch Names: — Set Switches
☐ RSF Include: — View...

Variable	Value	Default	Description
None			

View avParameters: ☐ — Modify avParameters... — No avParameters are set.
View Additional Functions: ☐ — No additional functions are set.

Setting up the Technology libraries(if not present)
Choose **Assura** from layout window-**Technology,** set path to **gpdk180.**

3. Click **OK** to start DRC.

4. A Progress form will appears. You can click on the watch log file to see the log file.

5. When DRC finishes, a dialog box appears asking you if you want to view your DRC results, and then click **Yes** to view the results of this run.

6. If there any DRC error exists in the design **View Layer Window** (VLW) and **Error Layer Window** (ELW) appears. Also the errors highlight in the design itself.

7. Click **View – Summary** in the ELW to find the details of errors.

8. You can refer to rule file also for more information, correct all the DRC errors and **Re – run** the DRC.

9. If there are no errors in the layout then a dialog box appears with **No DRC errors found** written in it, click on **close** to terminate the DRC run.

ASSURA LVS

In this section we will perform the LVS check that will compare the schematic netlist and the layout netlist.

Running LVS

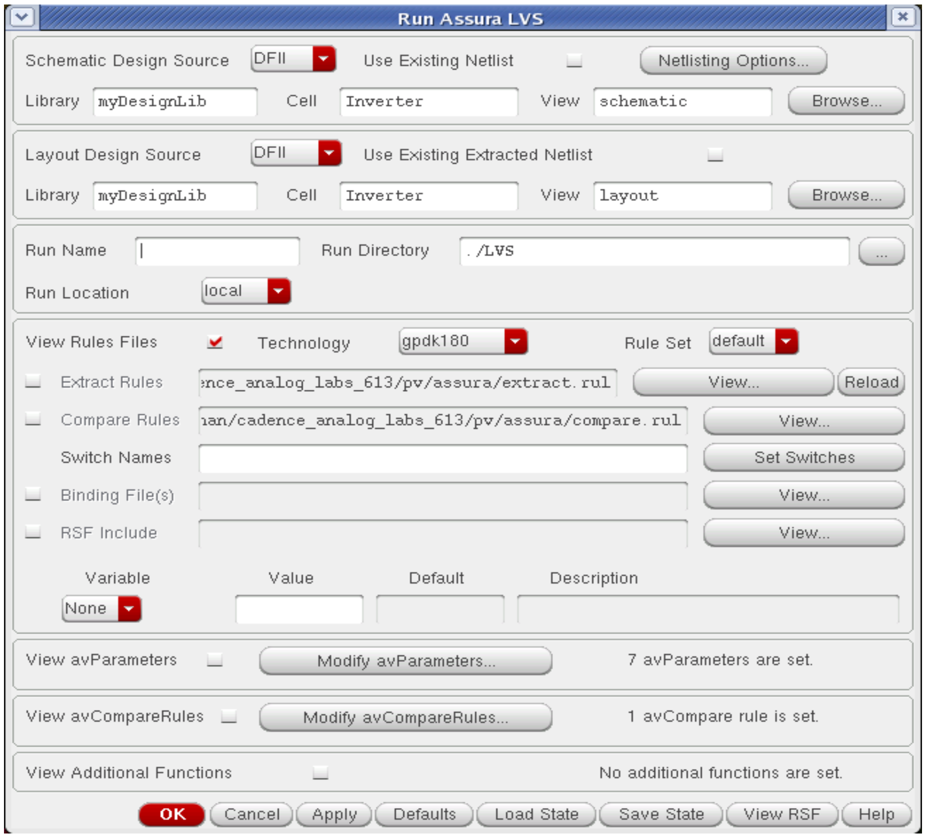
1. Select **Assura – Run LVS** from the layout window.

The Assura Run LVS form appears. It will automatically load both the schematic and layout view of the cell.

2. Change the following in the form and click **OK**
3. The LVS begins and a Progress form appears.
4. If the schematic and layout matches completely, you will get the form displaying **Schematic and Layout Match**.
5. If the schematic and layout do not matches, a form informs that the LVS completed successfully and asks if you want to see the results of this run.
6. Click **Yes** in the form.

LVS debug form appears, and you are directed into LVS debug environment.

7. In the **LVS debug form** you can find the details of mismatches and you need to correct all those mismatches and **Re – run** the LVS till you will be able to match the schematic with layout.



Schematic Design Source: DFII
Use Existing Netlist: ☐
Library: myDesignLib Cell: Inverter View: schematic

Layout Design Source: DFII
Use Existing Extracted Netlist: ☐
Library: myDesignLib Cell: Inverter View: layout

Run Name:
Run Directory: ./LVS
Run Location: local

View Rules Files: ✓ Technology: gpdk180 Rule Set: default
☐ Extract Rules: ence_analog_labs_613/pv/assura/extract.rul
☐ Compare Rules: ıan/cadence_analog_labs_613/pv/assura/compare.rul
Switch Names:
☐ Binding File(s):
☐ RSF Include:

Variable	Value	Default	Description
None			

View avParameters ☐ 7 avParameters are set.

View avCompareRules ☐ 1 avCompare rule is set.

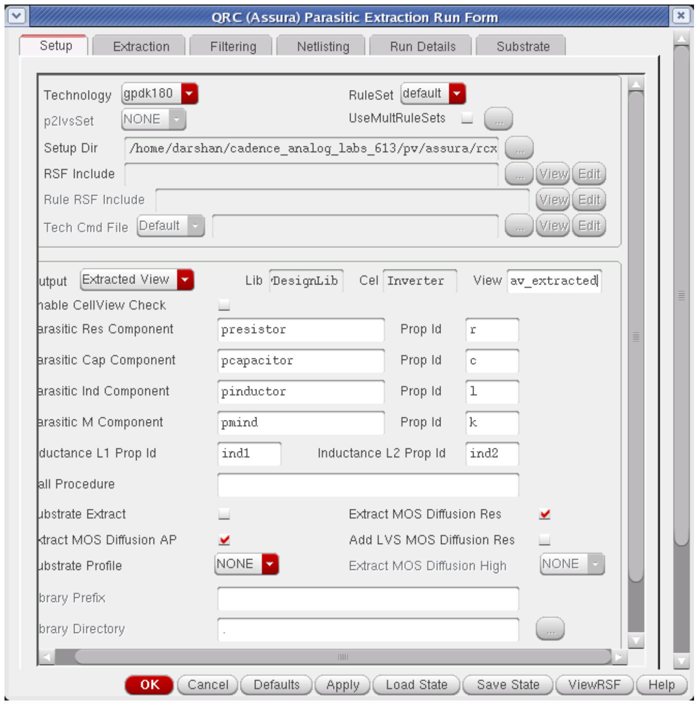
View Additional Functions ☐ No additional functions are set.

Assura RCX

In this section we will extract the RC values from the layout and perform analog circuit simulation on the designs extracted with RCX.
Before using RCX to extract parasitic devices for simulation, the layout should match with schematic completely to ensure that all parasites will be backannoted to the correct schematic nets.

Running RCX

1. From the layout window execute **Assura – Run RCX**.
2. Change the following in the Assura parasitic extraction form. Select **output** type under **Setup** tab of the form.



QRC (Assura) Parasitic Extraction Run Form

Setup | Extraction | Filtering | Netlisting | Run Details | Substrate

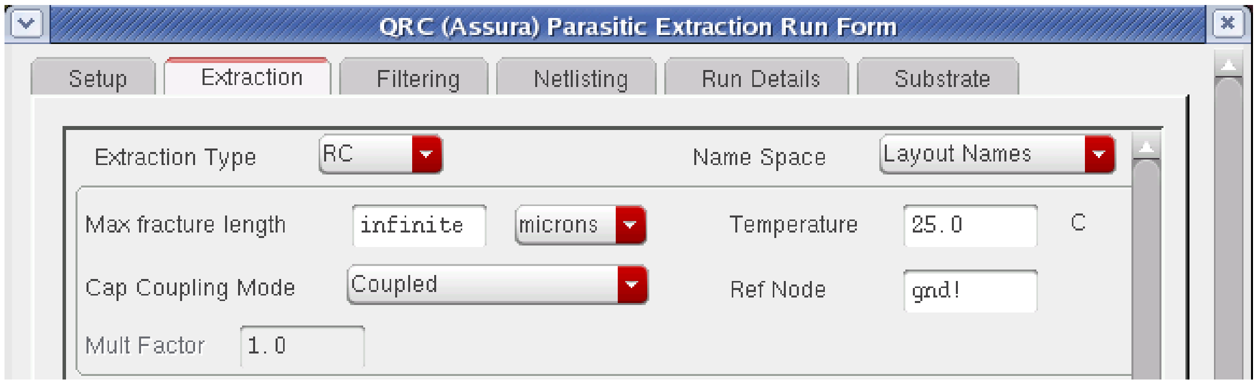
Technology	gpdk180	RuleSet	default
p2lvsSet	NONE	UseMultRuleSets	
Setup Dir	/home/darshan/cadence_analog_labs_613/pv/assura/rcx		
RSF Include			
Rule RSF Include			
Tech Cmd File	Default		

utput	Extracted View	Lib	DesignLib	Cel	Inverter	View	av_extracted

nable CellView Check	☐		
arasitic Res Component	presistor	Prop Id	r
arasitic Cap Component	pcapacitor	Prop Id	c
arasitic Ind Component	pinductor	Prop Id	l
arasitic M Component	pmind	Prop Id	k
ductance L1 Prop Id	ind1	Inductance L2 Prop Id	ind2
all Procedure			
ubstrate Extract	☐	Extract MOS Diffusion Res	✓
tract MOS Diffusion AP	✓	Add LVS MOS Diffusion Res	☐
ubstrate Profile	NONE	Extract MOS Diffusion High	NONE
brary Prefix			
brary Directory	.		

OK | Cancel | Defaults | Apply | Load State | Save State | ViewRSF | Help

3. In the **Extraction** tab of the form, choose Extraction type, Cap Coupling Mode and specify the Reference node for extraction



QRC (Assura) Parasitic Extraction Run Form

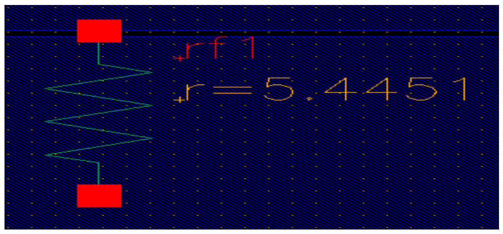
Setup | Extraction | Filtering | Netlisting | Run Details | Substrate

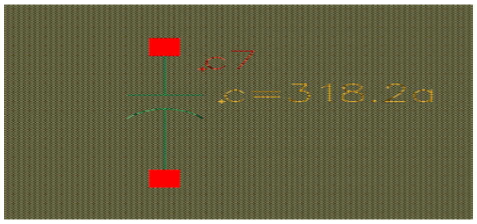
Extraction Type	RC	Name Space	Layout Names
Max fracture length	infinite microns	Temperature	25.0 C
Cap Coupling Mode	Coupled	Ref Node	gnd!
Mult Factor	1.0		

4. In the **Filtering** tab of the form, **Enter Power Nets** as **vdd!,vss!** and**Enter Ground Nets** as **gnd!**
   Click **OK** in the Assura parasitic extraction form when done. The RCX progress form appears, in the progress form click **Watch log file** to see the output log file.

5. When RCX completes, a dialog box appears, informs you that **Assura RCX run Completed successfully.**

6. You can open the **av_extracted** view from the library manager and view the parasitic.

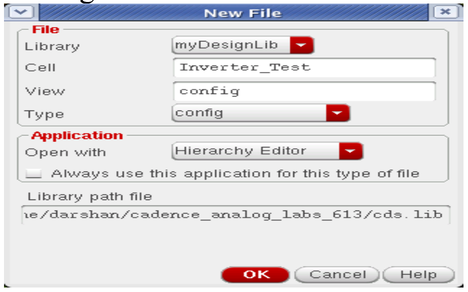




Creating the Configuration View

In this section we will create a config view and with this config view we will run thesimulation with and without parasitic.

1. In the CIW or Library Manager, execute **File – New – Cellview**
2. In the Create New file form, set the following:



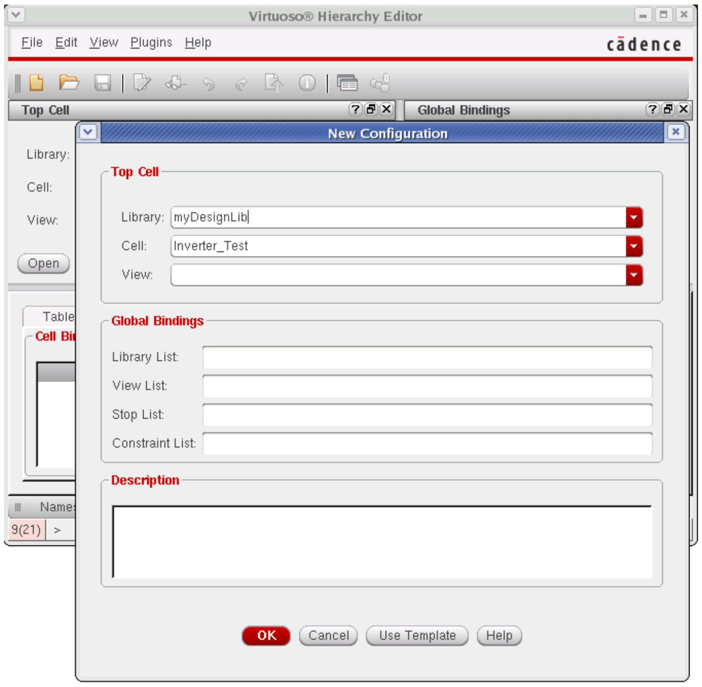
New File

File
Library: myDesignLib
Cell: Inverter_Test
View: config
Type: config

Application
Open with: Hierarchy Editor
Always use this application for this type of file

Library path file
ne/darshan/cadence_analog_labs_613/cds.lib

3. Click **OK** in create **New File** form.The **Hierarchy Editor** form opens and a **New Configuration** form opens in front of it.



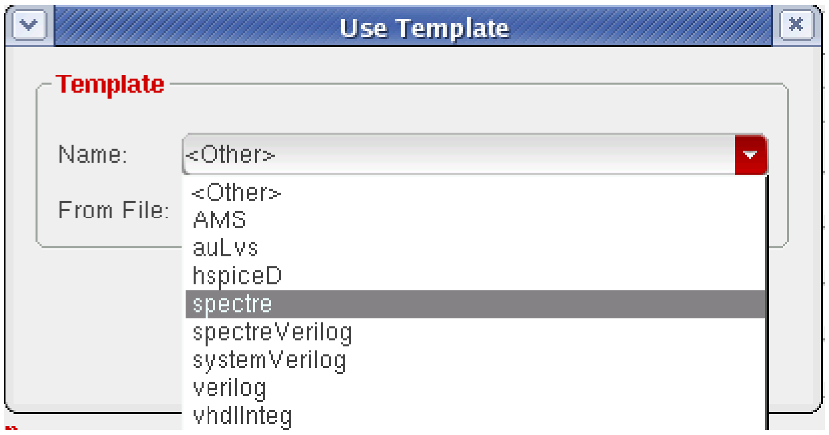
New Configuration

Top Cell
Library: myDesignLib
Cell: Inverter_Test
View:

Global Bindings
Library List:
View List:
Stop List:
Constraint List:

Description

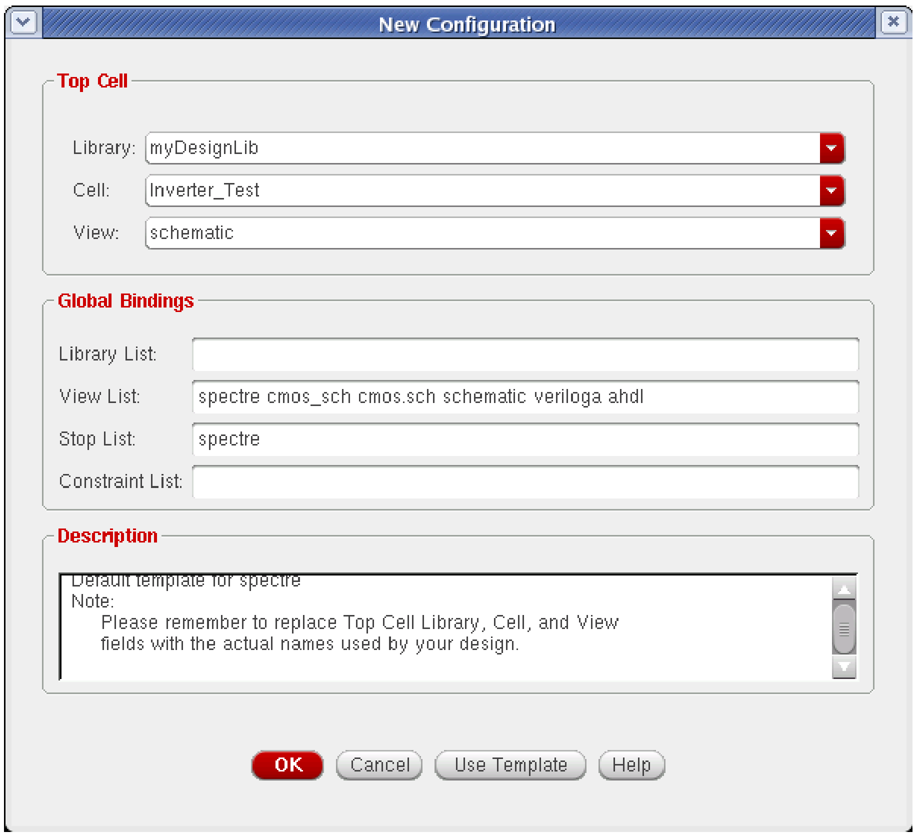
4. Click **Use template** at the bottom of the **New Configuration** form and select **Spectre**in the cyclic field and click **OK**. The Global Bindings lists are loaded from the template.



Name: <Other>

<Other>
AMS
auLvs
hspiceD
spectre
spectreVerilog
systemVerilog
verilog
vhdlInteg

5. Change the **Top Cell** View to **schematic** and remove the default entry from the **Library List** field.
6. Click **OK** in the New Configuration form.

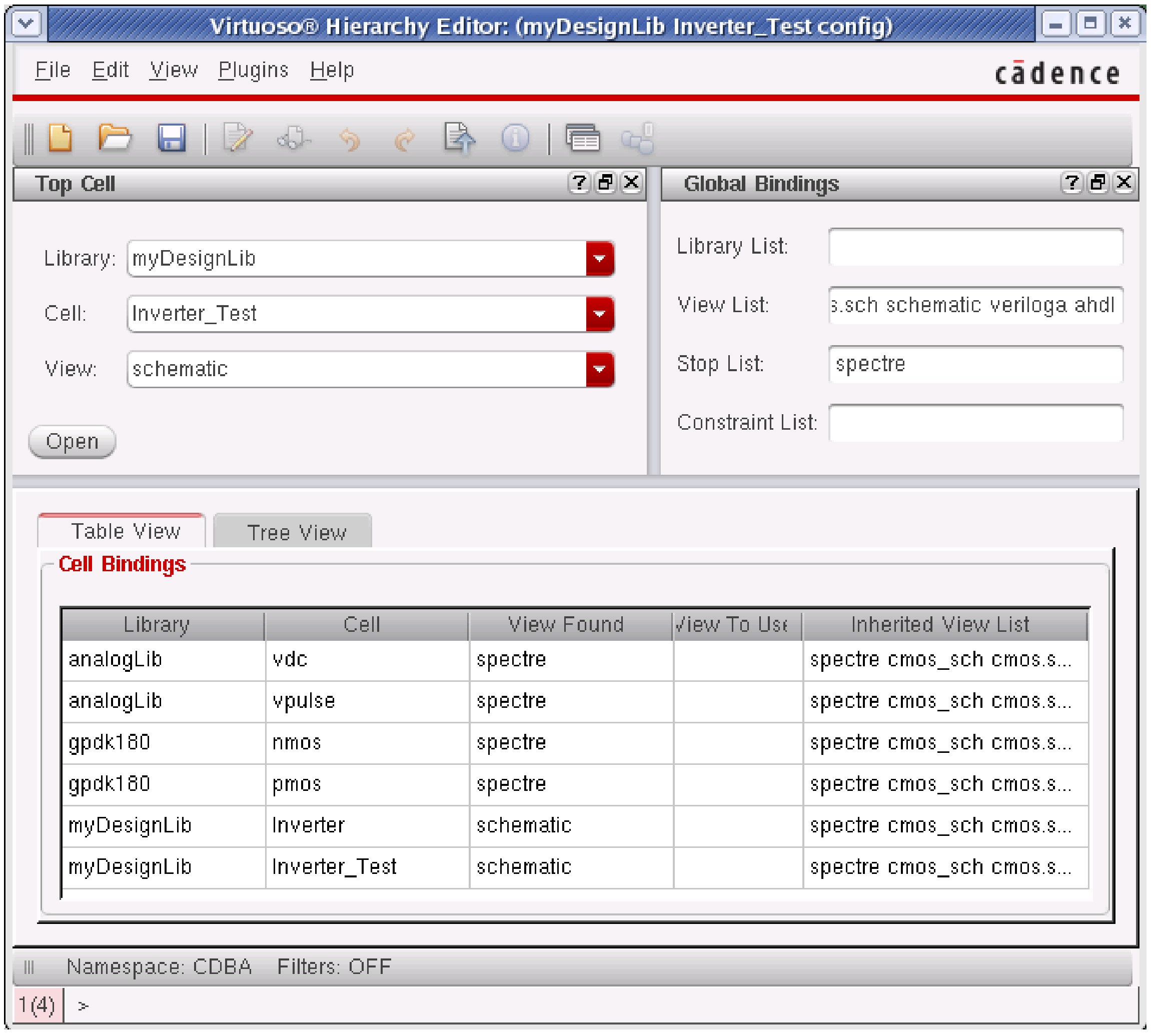


Library: myDesignLib
Cell: Inverter_Test
View: schematic

Library List:
View List: spectre cmos_sch cmos.sch schematic veriloga ahdl
Stop List: spectre
Constraint List:

Default template for spectre
Note:
Please remember to replace Top Cell Library, Cell, and View fields with the actual names used by your design.

7. The hierarchy editor displays the hierarchy for this design using table format.

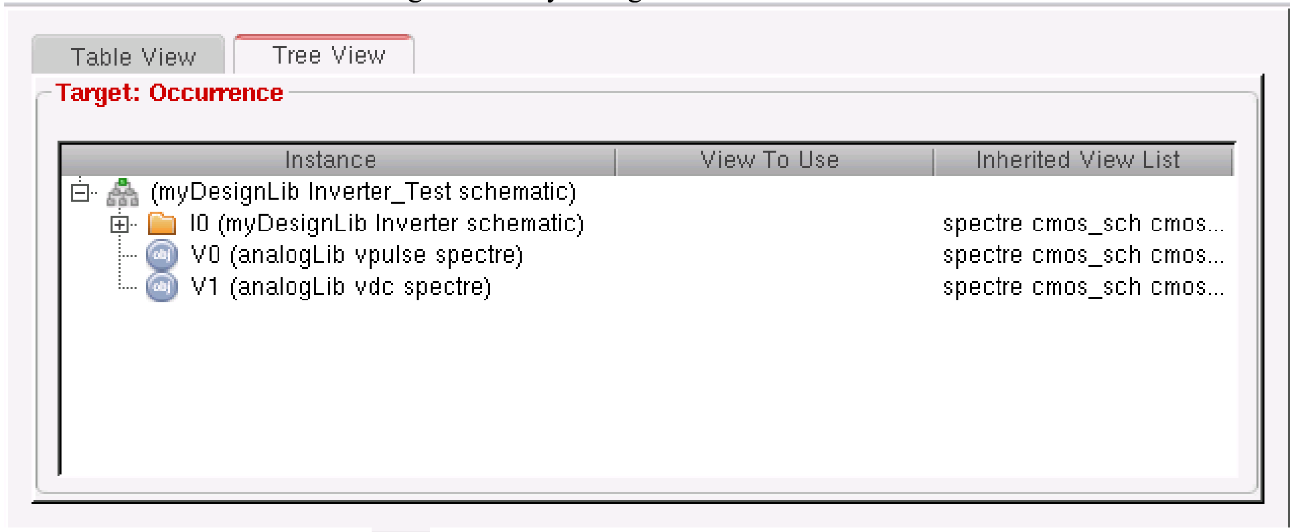

Virtuoso® Hierarchy Editor: (myDesignLib Inverter_Test config)
File Edit View Plugins Help
cādence
Top Cell
Global Bindings
Open
Table View
Tree View
Cell Bindings
Namespace: CDBA Filters: OFF
1(4) >

Library:	myDesignLib
Cell:	Inverter_Test
View:	schematic

Library List:	
View List:	s.sch schematic veriloga ahdl
Stop List:	spectre
Constraint List:	

Library	Cell	View Found	View To Use	Inherited View List
analogLib	vdc	spectre		spectre cmos_sch cmos.s...
analogLib	vpulse	spectre		spectre cmos_sch cmos.s...
gpdk180	nmos	spectre		spectre cmos_sch cmos.s...
gpdk180	pmos	spectre		spectre cmos_sch cmos.s...
myDesignLib	Inverter	schematic		spectre cmos_sch cmos.s...
myDesignLib	Inverter_Test	schematic		spectre cmos_sch cmos.s...

8. Click the **Tree View** tab. The design hierarchy changes to tree format. The form should look like this:



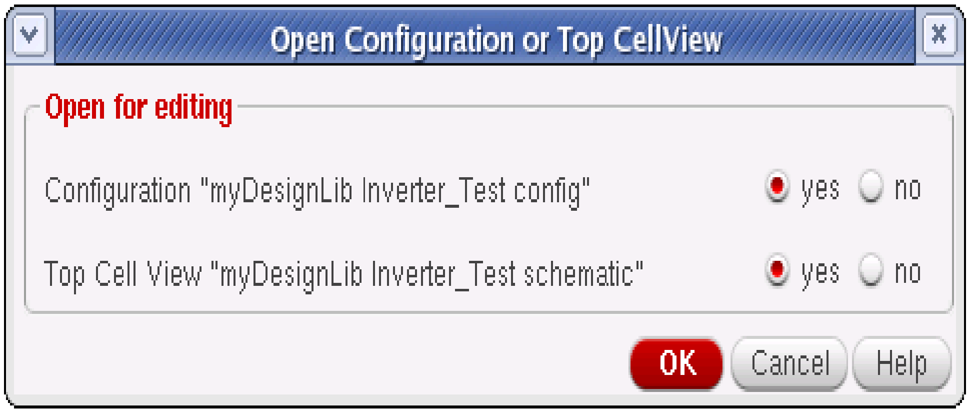
Instance	View To Use	Inherited View List
(myDesignLib Inverter_Test schematic)		
I0 (myDesignLib Inverter schematic)		spectre cmos_sch cmos...
V0 (analogLib vpulse spectre)		spectre cmos_sch cmos...
V1 (analogLib vdc spectre)		spectre cmos_sch cmos...

Save the current configuration

9. Close the Hierarchy Editor window. Execute **File – Close Window**

To run the Circuit without Parasites

1. From the Library Manager open **Inverter_Test**Config view.
Open Configuration or Top cellview form appear


Open Configuration or Top CellView

Open for editing

Configuration "myDesignLib Inverter_Test config" [✓ yes] no

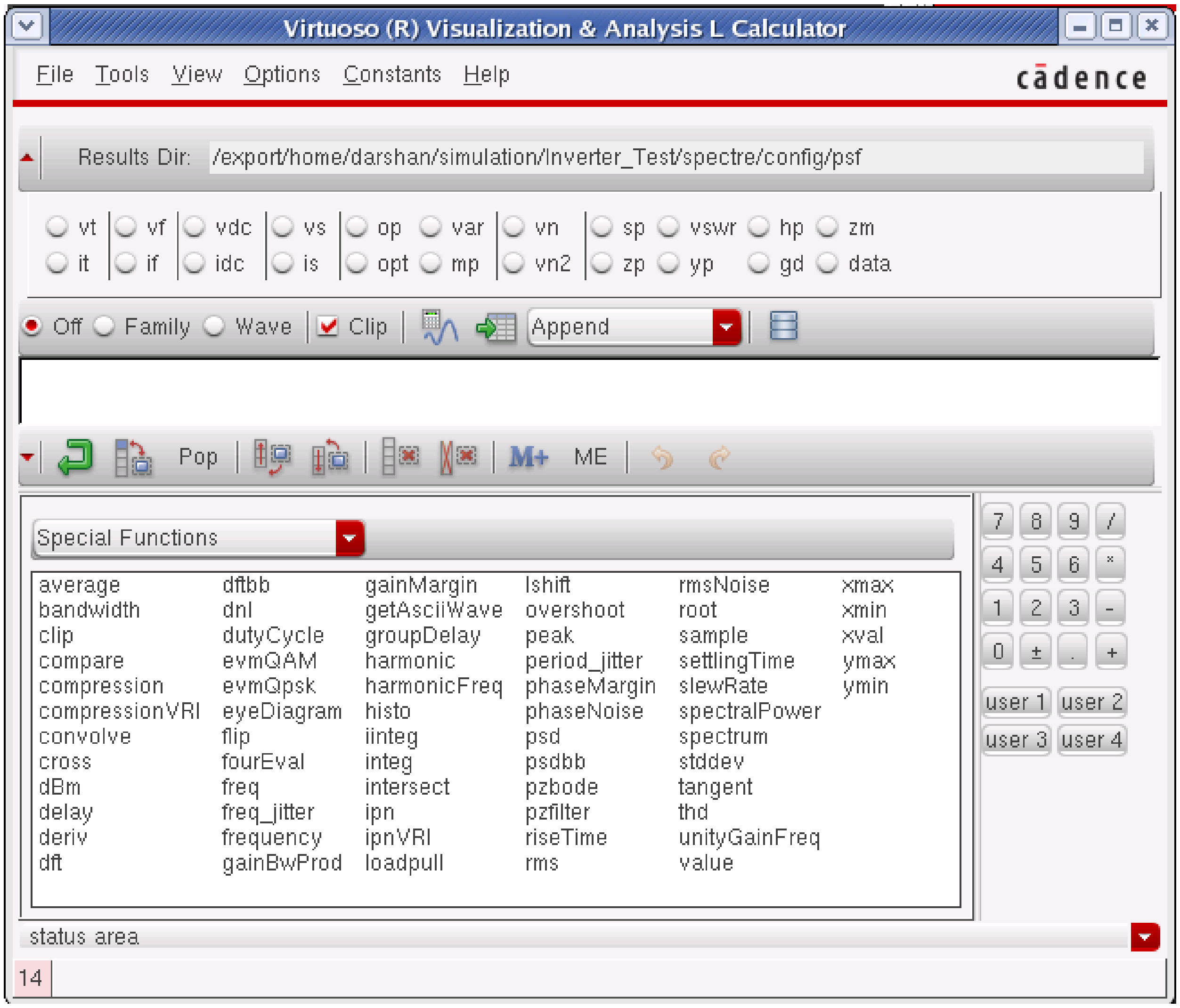
Top Cell View "myDesignLib Inverter_Test schematic" [✓ yes] no

2. In the form, turn on the both cyclic buttons to **Yes** and click **OK.**
The Inverter_Test schematic and Inverter_Testconfig window appears. Notice the window banner of schematic also states **Config: myDesignLibInverter_Testconfig.**
3. Execute **Launch – ADE L** from the schematic window.
4. Now you need to follow the same procedure for running the simulation. Executing **Session– Load state,** the Analog Design Environment window loads the previous state.

5. Click **Netlist and Run** icon to start the simulation. The simulation takes a few seconds and then waveform window appears.
6. In the CIW, note the netlisting statistics in the **Circuit inventory** section. This list includes all nets, designed devices, source and loads. There are no parasitic components. Also note down the circuit inventory section.

Measuring the Propagation Delay

1. In the waveform window execute **Tools – Calculator**
2. The calculator window appears


Virtuoso (R) Visualization & Analysis L Calculator
File Tools View Options Constants Help
cādence
vt vf vdc vs op var vn sp vswr hp zm
it if idc is opt mp vn2 zp yp gd data
Off Family Wave Clip Append
Pop M+ ME
Special Functions
average bandwidth clip compare compression compressionVRI convolve cross dBm delay deriv dft
dftbb dnl dutyCycle evmQAM evmQpsk eyeDiagram flip fourEval freq freq_jitter frequency gainBwProd
gainMargin getAsciiWave groupDelay harmonic harmonicFreq histo iinteg integ intersect ipn ipnVRI loadpull
lshift overshoot peak period_jitter phaseMargin phaseNoise psd psdbb pzbode pzfilter riseTime rms
rmsNoise root sample settlingTime slewRate spectralPower spectrum stddev tangent thd unityGainFreq value
xmax xmin xval ymax ymin
7 8 9 / 4 5 6 * 1 2 3 - 0 ± . +
user 1 user 2 user 3 user 4
status area
14

Results Dir: /export/home/darshan/simulation/Inverter_Test/spectre/config/psf

From the functions select **delay,** this will open the delay data panel.

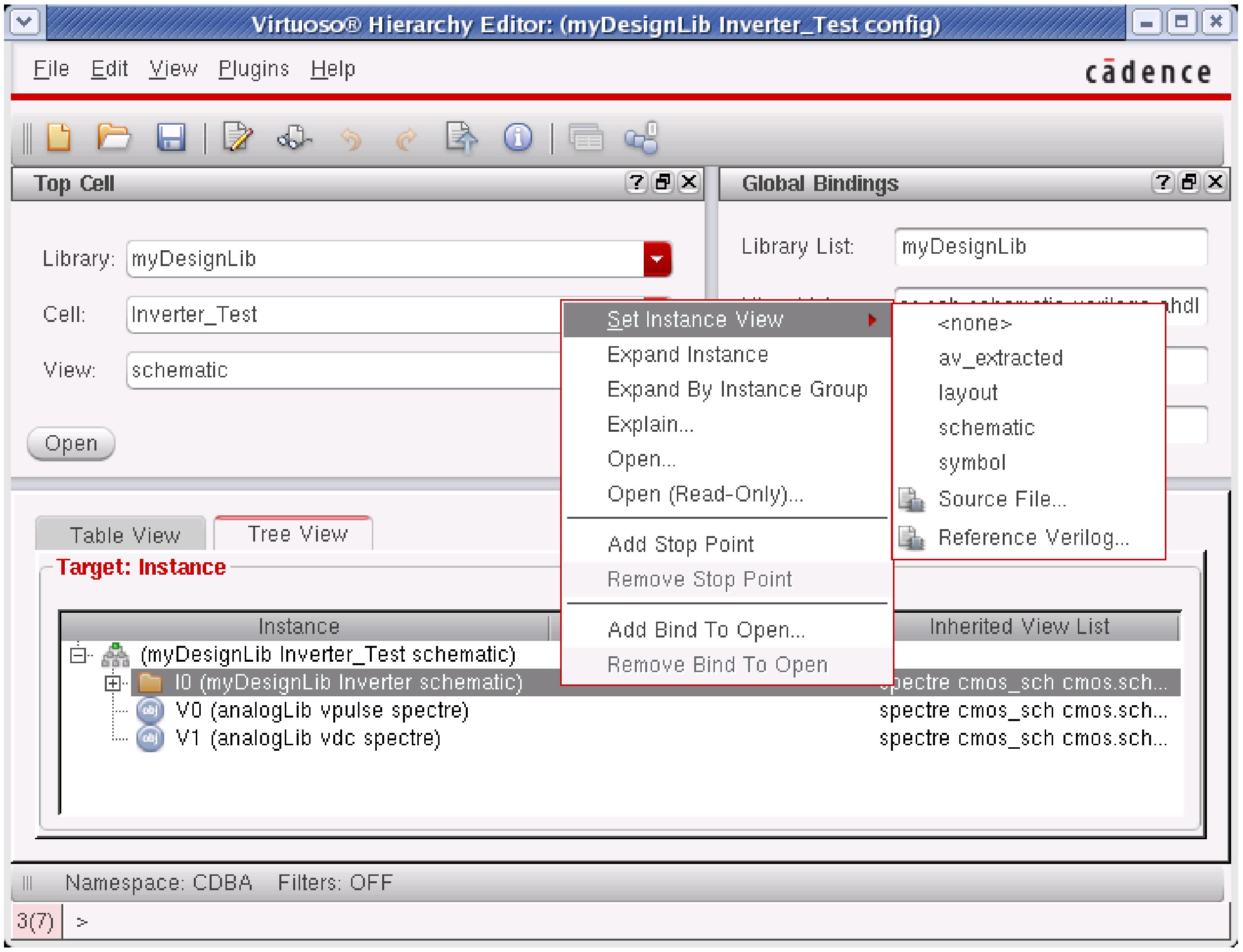
3. Place the cursor in the text box for Signal1, select the **wave** button and select the input waveform from the waveform window.
4. Repeat the same for Signal2, and select the output waveform.
5. Set the **Threshold value 1** and **Threshold value 2** to 0.9, this directs the calculator to calculate delay at 50% i.e. at 0.9 volts.
6. Execute **OK** and observe the expression created in the calculator buffer.
7. Click on **Evaluate the buffer icon** to perform the calculation, note down the value returned after execution.
8. Close the calculator window

To run the Circuit with Parasites

In this exercise, we will change the configuration to direct simulation of the **av_extracted**view which contains the parasites.

1. Open the same Hierarchy Editor form, which is already set for Inverter_Testconfig.
2. Select the **Tree View** icon: this will show the design hierarchy in the tree format.
3. Click **right** mouse on the Inverter schematic.

A pull down menu appears. Select **av_extracted** view from the **Set Instance view** menu, the View to use column now shows av_extracted view



4. Click on the **Recompute the hierarchy** icon, the configuration is now updated from schematic to av_extracted view.

5. From the **Analog Design Environment** window click **Netlist and Run** to start the simulation again.

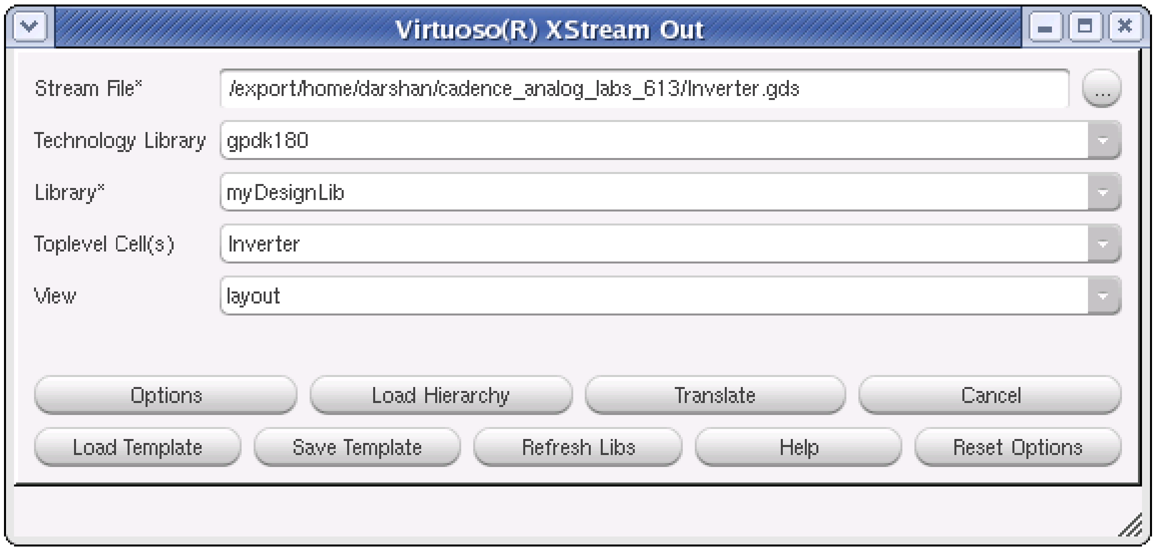
6. When simulation completes, note the **Circuit inventory conditions**, this time the list shows all nets, designed devices, sources and parasitic devices as well.

7. Calculate the delay again and match with the previous one. Now you can conclude how much delay is introduced by these parasites, now our main aim should to minimize the delay due to these parasites so number of iteration takes place for making an optimize layout.

Generate Streaming Data

Streaming Out the Design

1. Select **File – Export – Stream** from the CIW menu and **Virtuoso Xstream out** form appears change the following in the form.

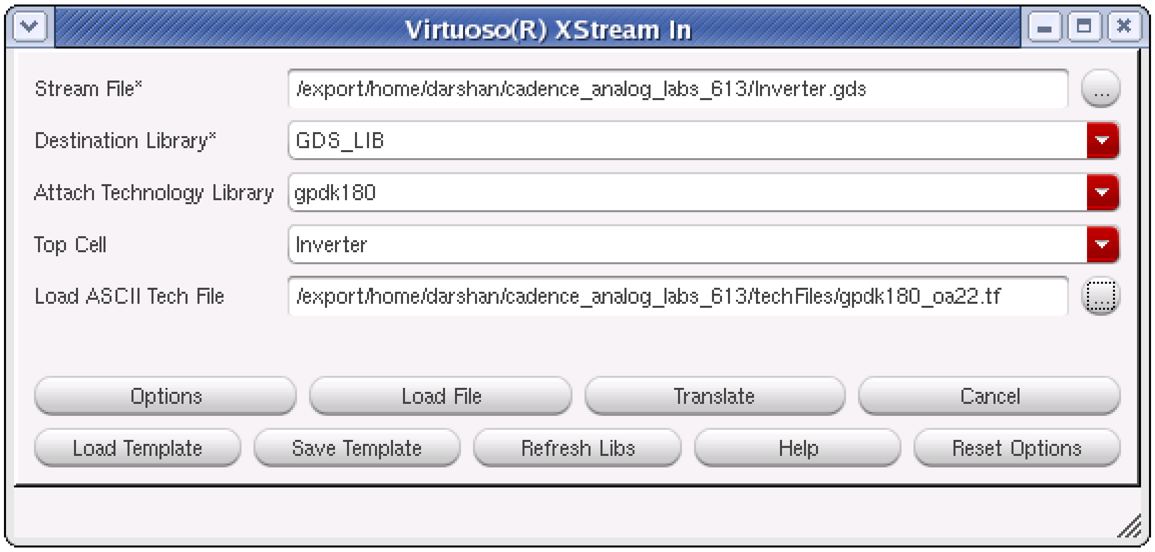


Stream File*	/export/home/darshan/cadence_analog_labs_613/Inverter.gds
Technology Library	gpdk180
Library*	myDesignLib
Toplevel Cell(s)	Inverter
View	layout

2. Click on the **Options** button.
3. In the **StreamOut-Options** form select under **Layers** tab and click **OK.**
4. In the **Virtuoso XStream Out** form, click **Translate** button to start the stream translator.
5. The stream file **Inverter.gds** is stored in the specified location.

Streaming In the Design

1. Select **File – Import – Stream** from the CIW menu and change the following in the form.



Stream File*	/export/home/darshan/cadence_analog_labs_613/Inverter.gds
Destination Library*	GDS_LIB
Attach Technology Library	gpdk180
Top Cell	Inverter
Load ASCII Tech File	/export/home/darshan/cadence_analog_labs_613/techFiles/gpdk180_oa22.tf

You need to specify the **gpdk180_oa22.tf** file. This is the entire technology file that has been dumped from the design library.
2. Click on the **Options** button.
3. In the **StreamOut-Options** form select under **Layers** tab and click **OK.**
4. In the **Virtuoso XStream Out** form, click **Translate** button to start the stream translator.
5. From the Library Manager open the **Inverter** cellview from the **GDS_LIB** library and notice the design.

RESULT:
a. The schematic for the inverter is drawn and verified the following: DC Analysis, Transient Analysis
b. The Layout for the inverter is drawn and verified the DRC, LVS, RC Extraction

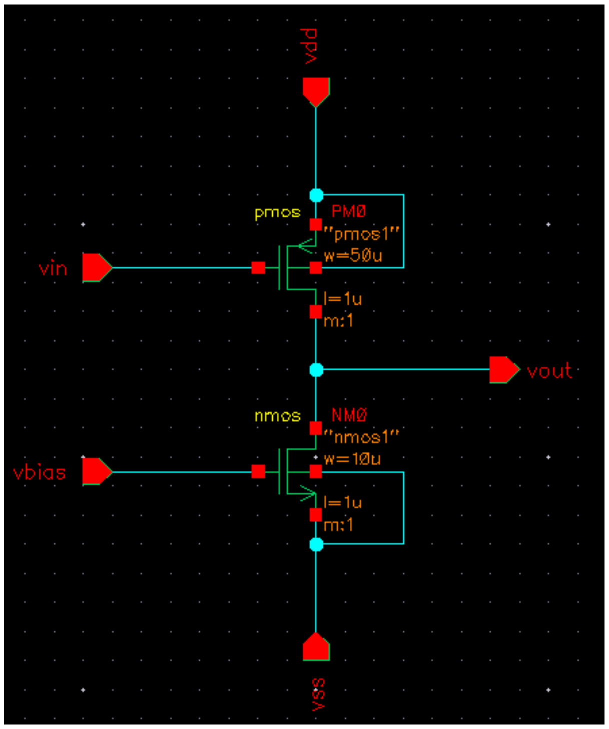
Lab 2a: COMMON SOURCE AMPLIFIER

AIM: To simulate the schematic of the Common Source Amplifier, and then to perform the physical verification for the layout of the same.

TOOL REQUIRED: Cadence Tool

THEORY: In electronics, a common-source amplifier is one of three basic single-stage field-effect transistor (FET) amplifier topologies, typically used as a voltage or transconductance amplifier. The easiest way to tell if a FET is common source, common drain, or common gate is to examine where the signal enters and leaves. The remaining terminal is what is known as "common". In this example, the signal enters the gate, and exits the drain. The only terminal remaining is the source. This is a common-source FET circuit. The analogous bipolar junction transistor circuit is the common-emitter amplifier. The common-source (CS) amplifier may be viewed as a transconductance amplifier or as a voltage amplifier. (See classification of amplifiers). As a transconductance amplifier, the input voltage is seen as modulating the current going to the load. As a voltage amplifier, input voltage modulates the amount of current flowing through the FET, changing the voltage across the output resistance according to Ohm's law. However, the FET device's output resistance typically is not high enough for a reasonable transconductance amplifier (ideally infinite), nor low enough for a decent voltage amplifier (ideally zero). Another major drawback is the amplifier's limited high-frequency response.

Schematic Capture



Schematic Entry

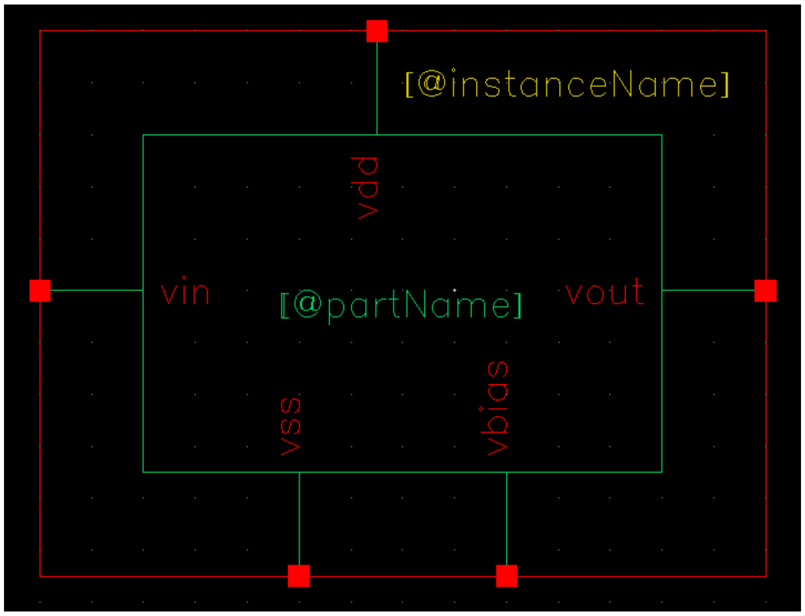
This is a table of components for building the Common Source Amplifier schematic

Library name	Cell Name	Properties/Comments
gpdk180	pmos	Model Name = pmos1; W= 50u ; L= 1u
gpdk180	nmos	Model Name =nmos1; W= 10u ; L= 1u

Type the following in the ADD pin form in the exact order leaving space between the pin names

Pin Names	Direction
vin vbias	Input
vout	Output
vddvss	Input

Symbol Creation

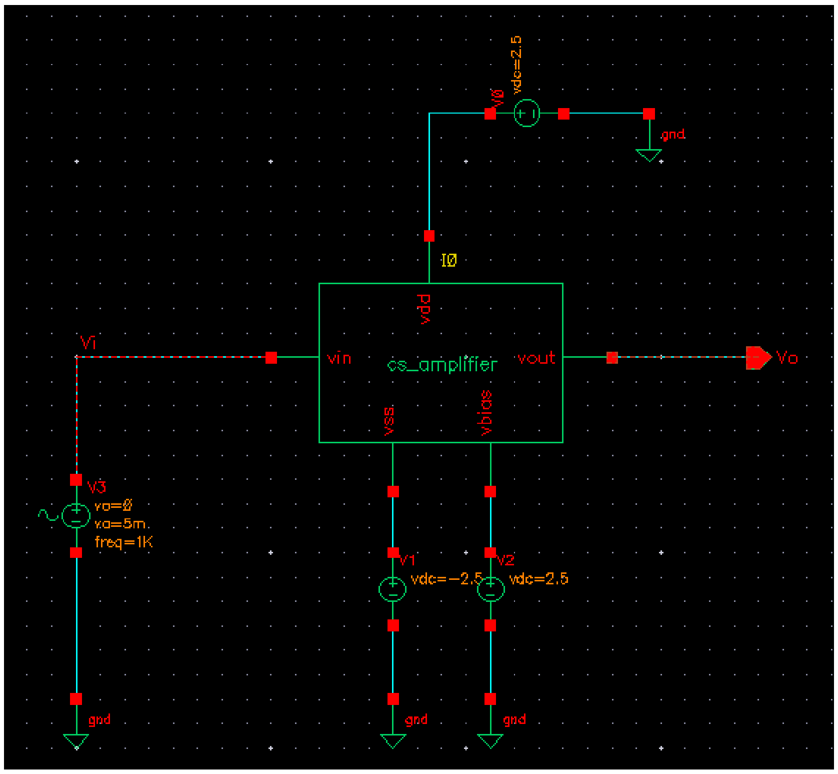


Building the Common Source Amplifier Test Design

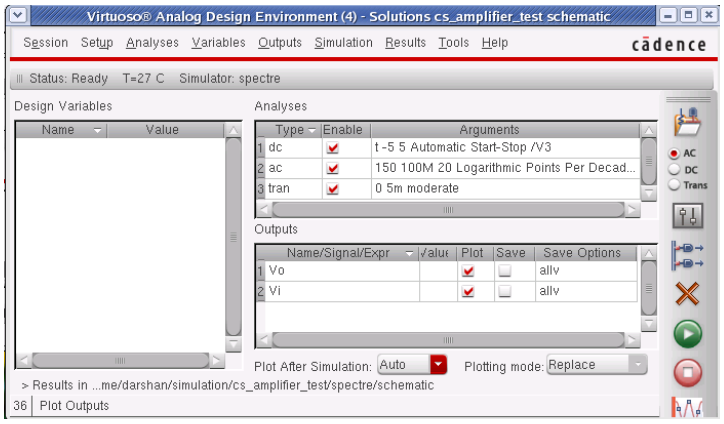
Using the component list and Properties/Comments in the table, build the cs-amplifier_test schematic as shown below.

Library name	Cell Name	Properties/Comments
myDesignLib	cs_amplifier	Symbol
analogLib	vsin	Define pulse specification as

		AC Magnitude= 1; DC Voltage= 0; Offset Voltage= 0; Amplitude= 5m Frequency= 1K
analogLib	Vdd,vss,gnd	vdd=2.5 ; vss= -2.5



Analog Simulation with Spectre


Virtuoso® Analog Design Environment (4) - Solutions cs_amplifier_test schematic
Session Setup Analyses Variables Outputs Simulation Results Tools Help
cādence
AC
DC
Trans

Status: Ready T=27 C Simulator: spectre

Design Variables

Name	Value

Analyses

	Type	Enable	Arguments
1	dc	✓	t -5 5 Automatic Start-Stop /V3
2	ac	✓	150 100M 20 Logarithmic Points Per Decad...
3	tran	✓	0 5m moderate

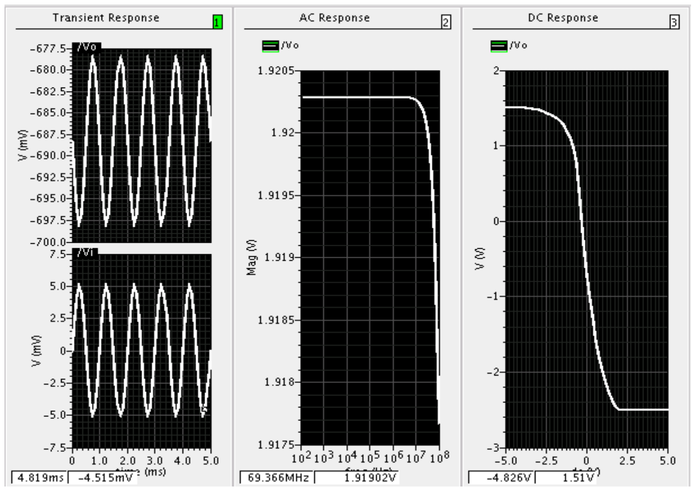
Outputs

	Name/Signal/Expr	Value	Plot	Save	Save Options
1	Vo		✓	☐	allv
2	Vi		✓	☐	allv

Plot After Simulation: Auto Plotting mode: Replace

> Results in ...me/darshan/simulation/cs_amplifier_test/spectre/schematic

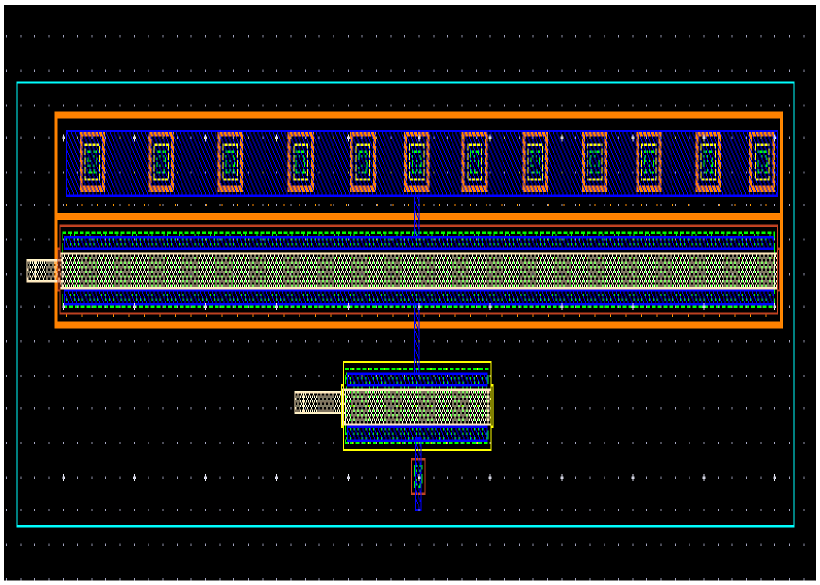
36 Plot Outputs



Creating a layout view of Common Source Amplifier

Complete the DRC, LVS check using the Assura tool.

Extract RC parasites for back annotation and Re-simulation



RESULT:

a. The schematic for the common source amplifier is drawn and verified the following: DC Analysis, DC Analysis, and Transient Analysis.

b. The Layout for the common source amplifier is drawn and verified the DRC, LVS, and RC Extraction.

Lab 2b: Common Drain Amplifier

AIM: To simulate the schematic of the Common Drain Amplifier, and then to perform the physical verification for the layout of the same.

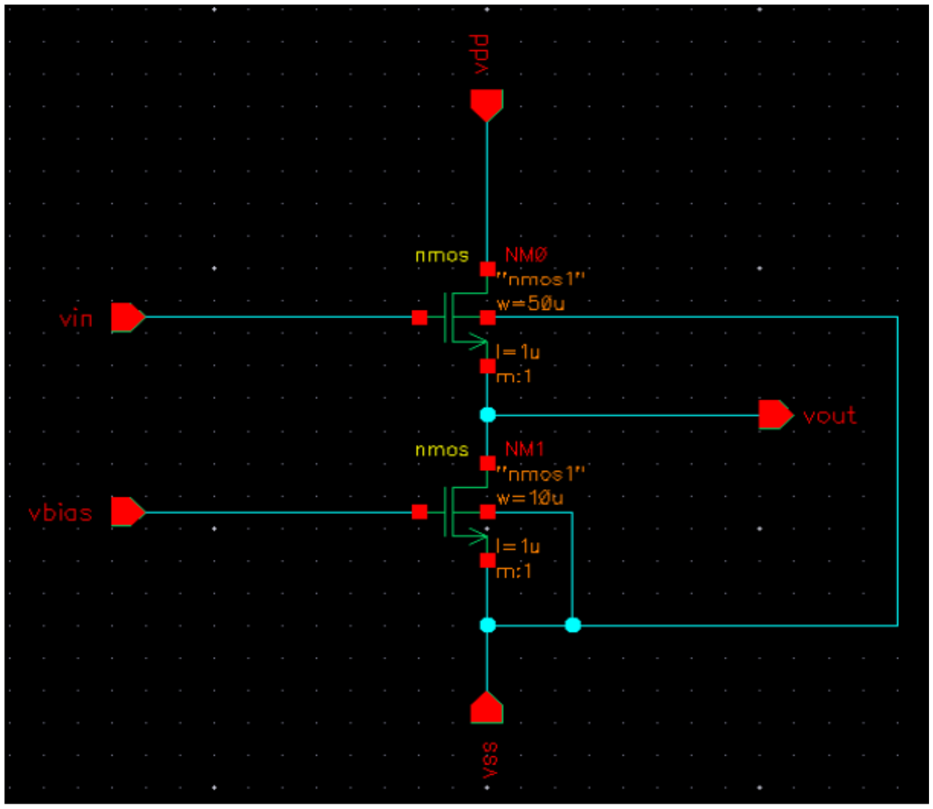
TOOL REQUIRED: Cadence Tool

THEORY: Common drain amplifier is a source follower or buffer amplifier circuit using a MOSFET. The output is simply equal to the input minus about 2.2V. The advantage of this circuit is that the MOSFET can provide current and power gain; the MOSFET draws no current from the input. It provides low output impedance to any circuit using the output of the follower, meaning that the output will not drop under load.

Its output impedance is not as low as that of an emitter follower using a bipolar transistor (as you can verify by connecting a resistor from the output to -15V), but it has the advantage that the input impedance is infinite. The MOSFET is in saturation, so the current across it is determined by the gatesource voltage. Since a current source keeps the current constant, the gate-source voltage is also constant.

Schematic Capture

Circuit diagram in schematic entry



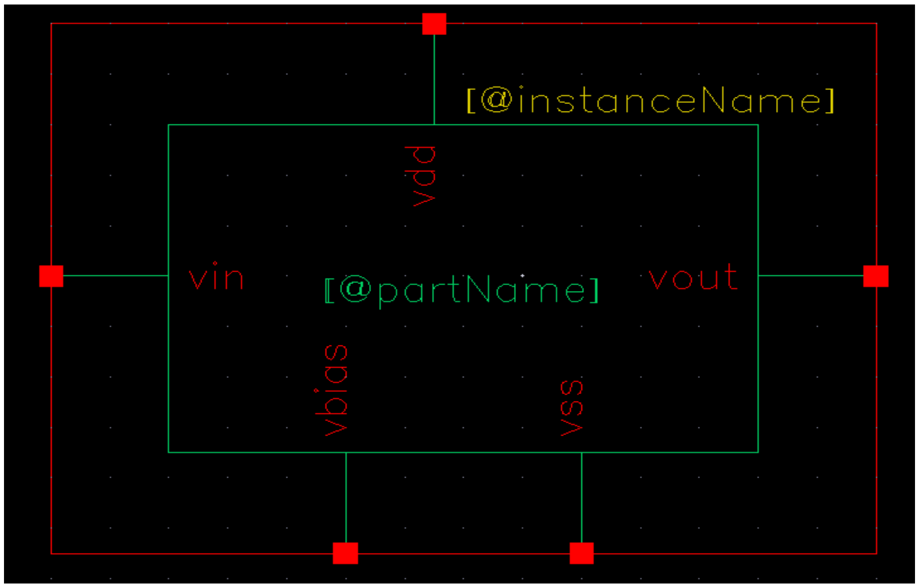
This is a table of components for building the Common Drain Amplifier schematic

Library name	Cell Name	Properties/Comments
gpdk180	nmos	Model Name = nmos1; W= 50u ; L= 1u
gpdk180	nmos	Model Name = nmos1; W= 10u ; L= 1u

Type the following in the ADD pin form in the exact order leaving space between the pin names.

Pin Names	Direction
vin, vbias	Input
vout	Output
vdd vss	Input

Symbol Creation

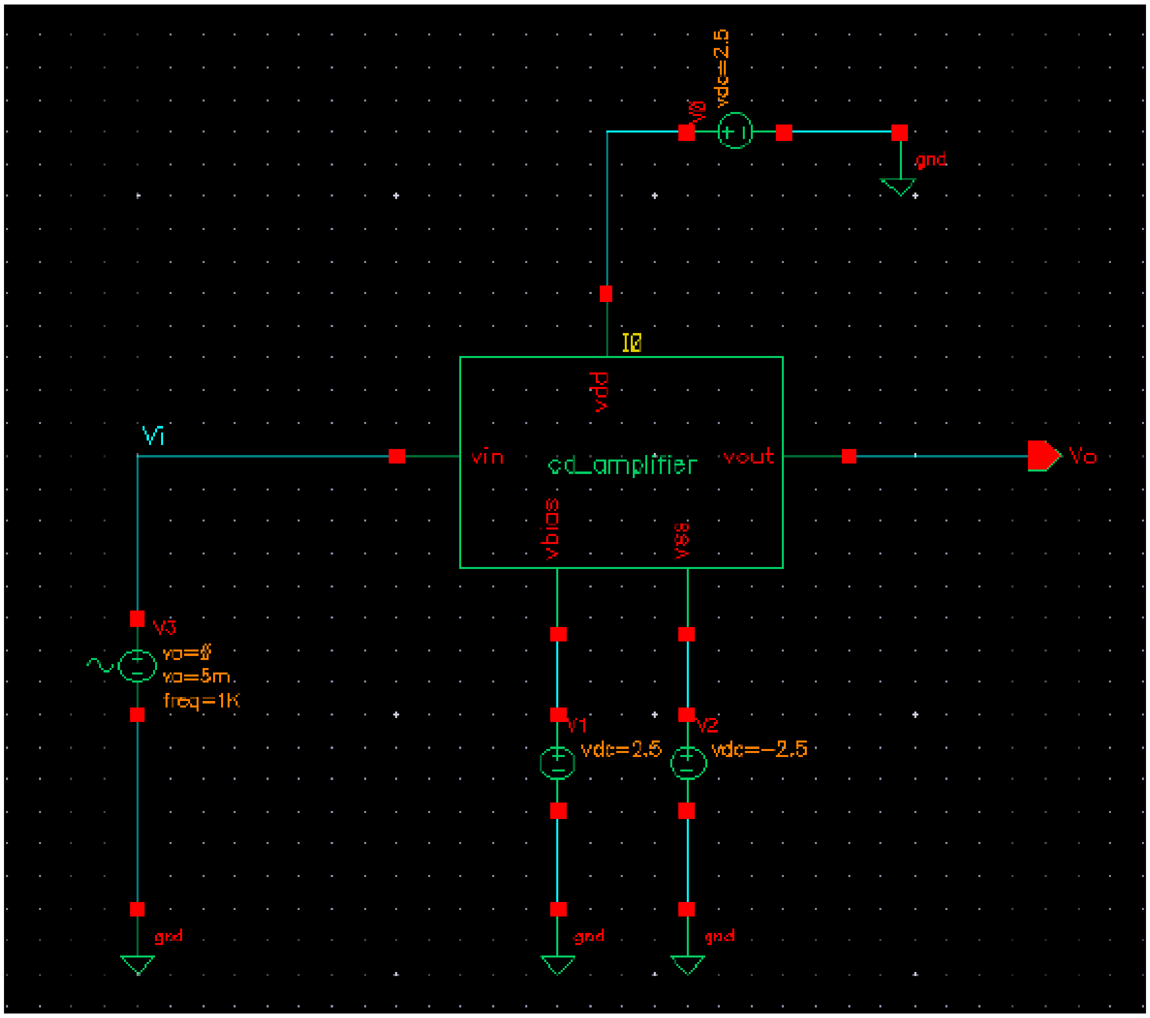


Building Common Drain Amplifier test Design

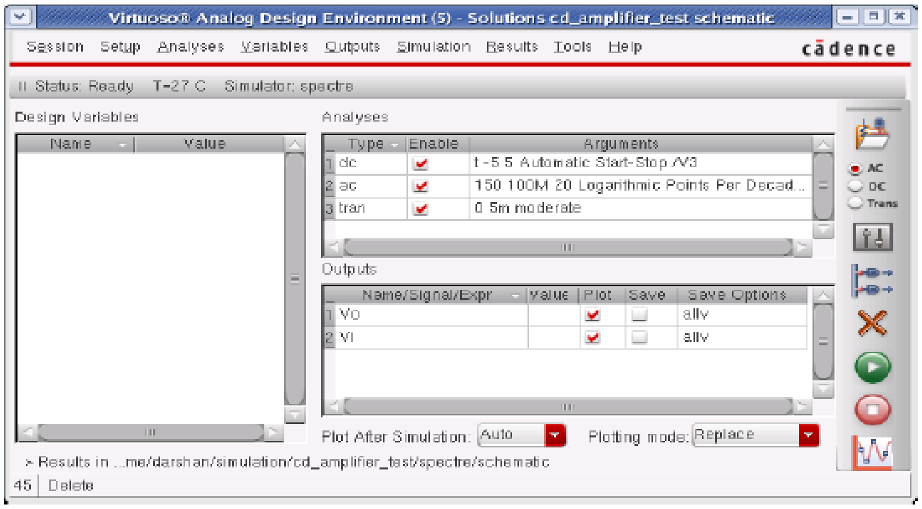
Using the component list and Properties/Comments in the table, build the cd-amplifier_test schematic as shown below.

Library name	Cellview name	Properties/Comments
myDesignLib	cd_amplifier	Symbol
analogLib	vsin	Define pulse specification as AC Magnitude= 1; DC Voltage= 0; Offset Voltage= 0; Amplitude= 5m; Frequency= 1K
analogLib	vdd,vss,gnd	vdd=2.5 ; vss= -2.5

Test Circuit



Analog simulation with spectre



Virtuoso® Analog Design Environment (5) - Solutions cd_amplifier_test schematic

Session Setup Analyses Variables Outputs Simulation Results Tools Help

Status: Ready T=27 C Simulator: spectre

Design Variables

Name	Value

Analyses

	Type	Enable	Arguments
1	dc	✓	t -5 5 Automatic Start-Stop /V3
2	ac	✓	150 100M 20 Logarithmic Points Per Decad...
3	tran	✓	0 5m moderate

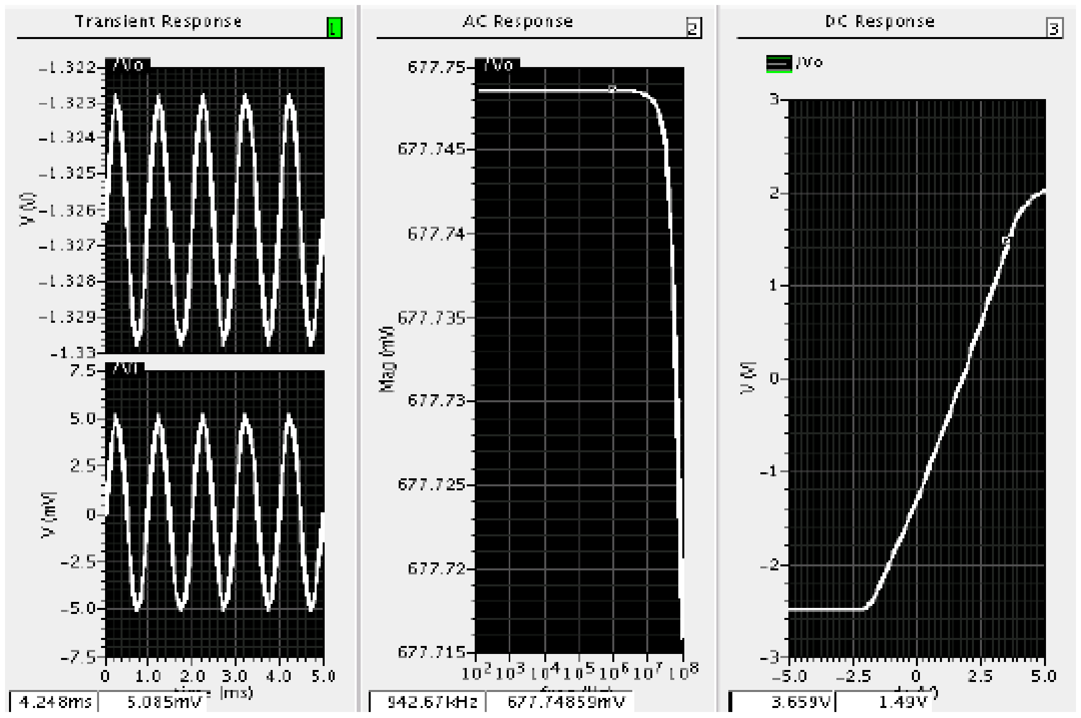
Outputs

	Name/Signal/Expr	Value	Plot	Save	Save Options
1	Vo		✓		allv
2	Vi		✓		allv

Plot After Simulation: Auto Plotting mode: Replace

> Results in .../me/darshan/simulation/cd_amplifier_test/spectre/schematic

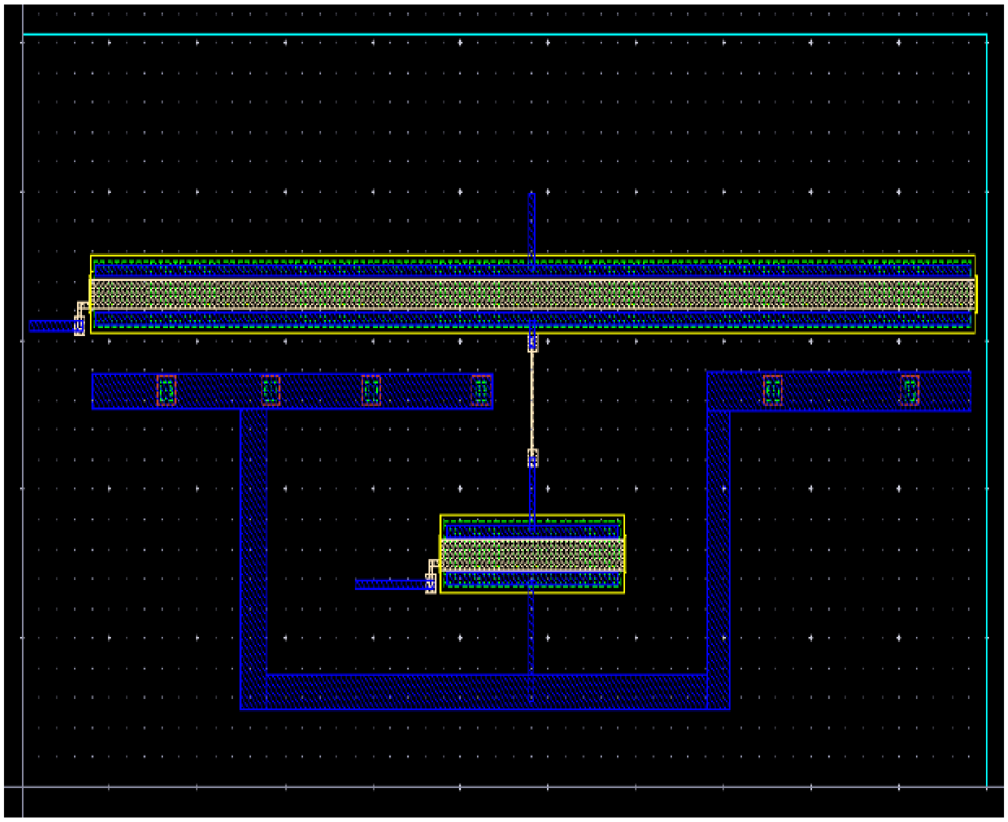
45 Delete



Creating a layout view of Common Drain Amplifier

Complete the DRC, LVS check using the Assura tool.

Extract RC parasites for back annotation and Re-simulation.



RESULT:

a. The schematic for the common drain amplifier is drawn and verified the following: DC Analysis, DC Analysis, and Transient Analysis.

b. The Layout for the common drain amplifier is drawn and verified the DRC, LVS, and RC Extraction.

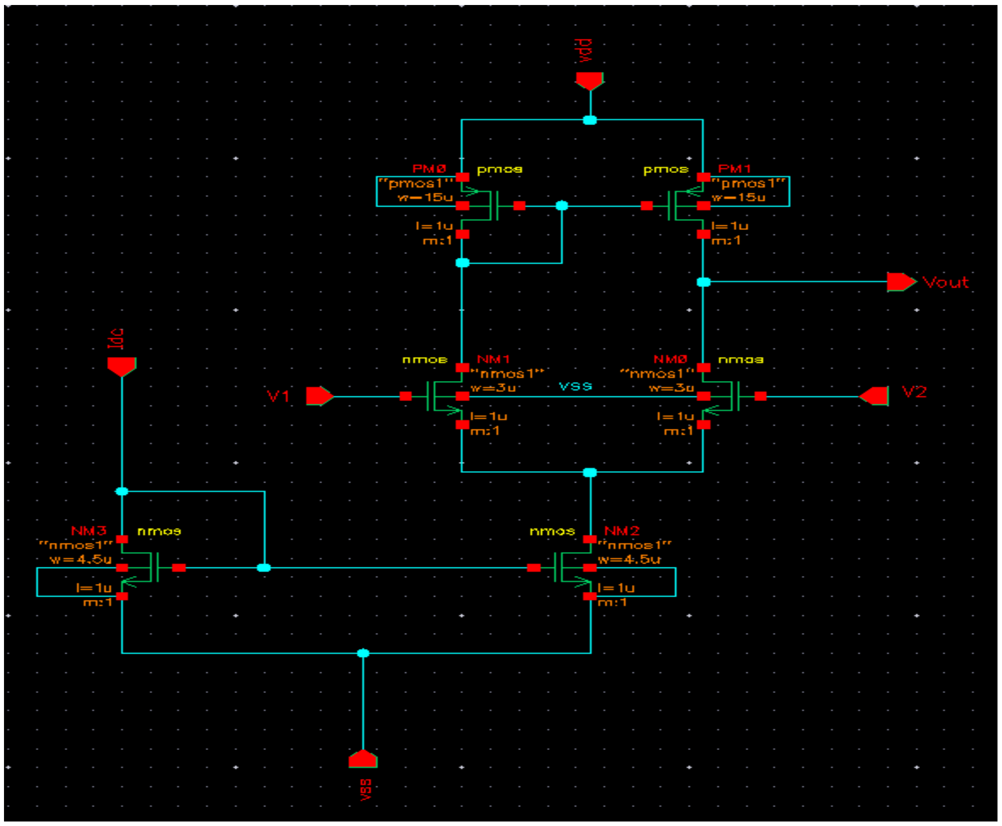
Lab 2c: DIFFERENTIAL AMPLIFIER

AIM : To simulate the schematic of the differential amplifier, and then to perform the physical verification for the layout of the same.

TOOL REQUIRED : Cadence Tool

THEORY : The differential amplifier is probably the most widely used circuit building block in analog integrated circuits, principally op amps. We had a brief glimpse at one back in Chapter 3 section 3.4.3 when we were discussing input bias current. The differential amplifier can be implemented with BJTs or MOSFETs. A differential amplifier multiplies the voltage difference between two inputs (Vin+ - Vin-) by some constant factor Ad, the differential gain. It may have either one output or a pair of outputs where the signal of interest is the voltage difference between the two outputs. A differential amplifier also tends to reject the part of the input signals that are common to both inputs (Vin+ + Vin-)/2 . This is referred to as the common mode signal.

Schematic Capture

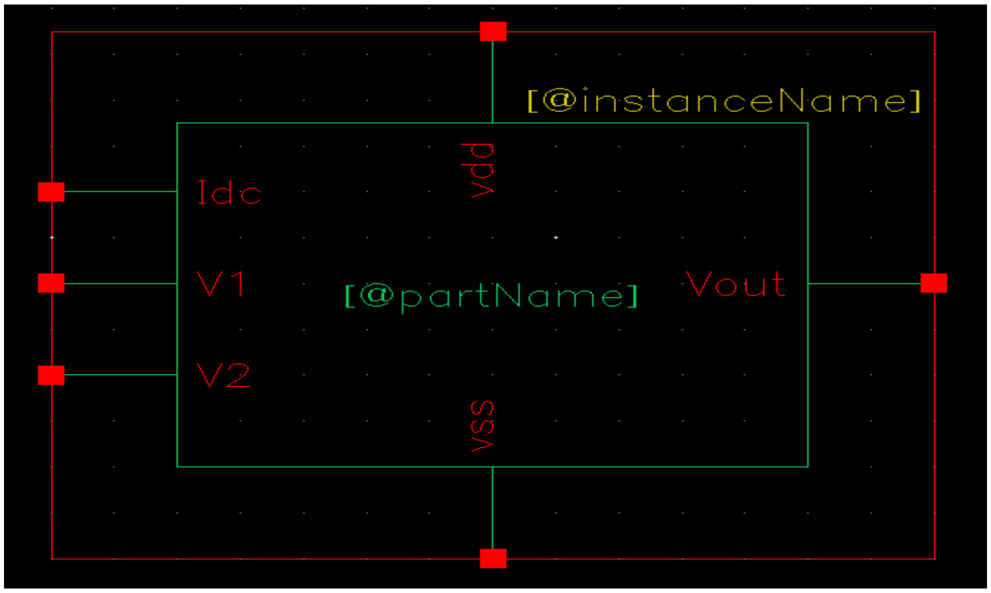


Schematic Entry

Parameters

Library name	Cell Name	Properties/Comments
gpdk180	Nmos	Model Name = nmos1 (NM0, NM1) ; W= 3u ; L =1u
gpdk180	Nmos	Model Name =nmos1 (NM2, NM3) ; W=4.5u;L=1u
gpdk180	Pmos	Model Name =pmos1 (PM0, PM1); W= 15u ; L= 1u

Symbol Creation



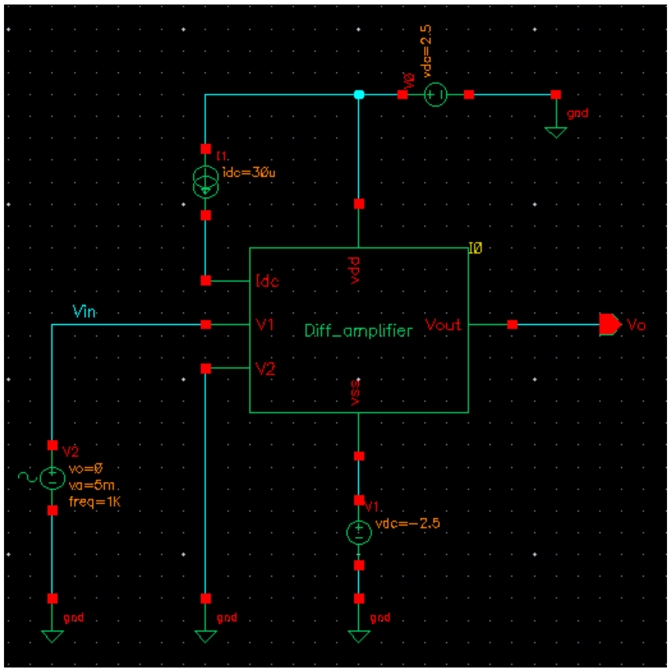
Building the Diff_amplifier_testCircuit

Using the component list and Properties/Comments in this table, build the Diff_amplifier_test schematic

Library name	Cellview name	Properties/Comments
myDesignLib	Diff_amplifier	Symbol
analogLib	vsin	Define specification as AC Magnitude= 1; Amplitude= 5m; Frequency= 1K

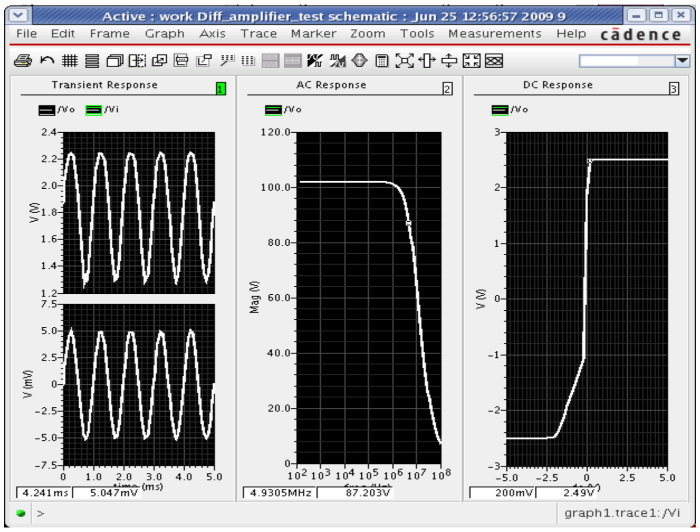
analogLib	vdd, vss, gnd	Vdd=2.5 ; Vss= -2.5
analogLib	Idc	Dc current = 30u

Schematic Diagram of the test circuit:

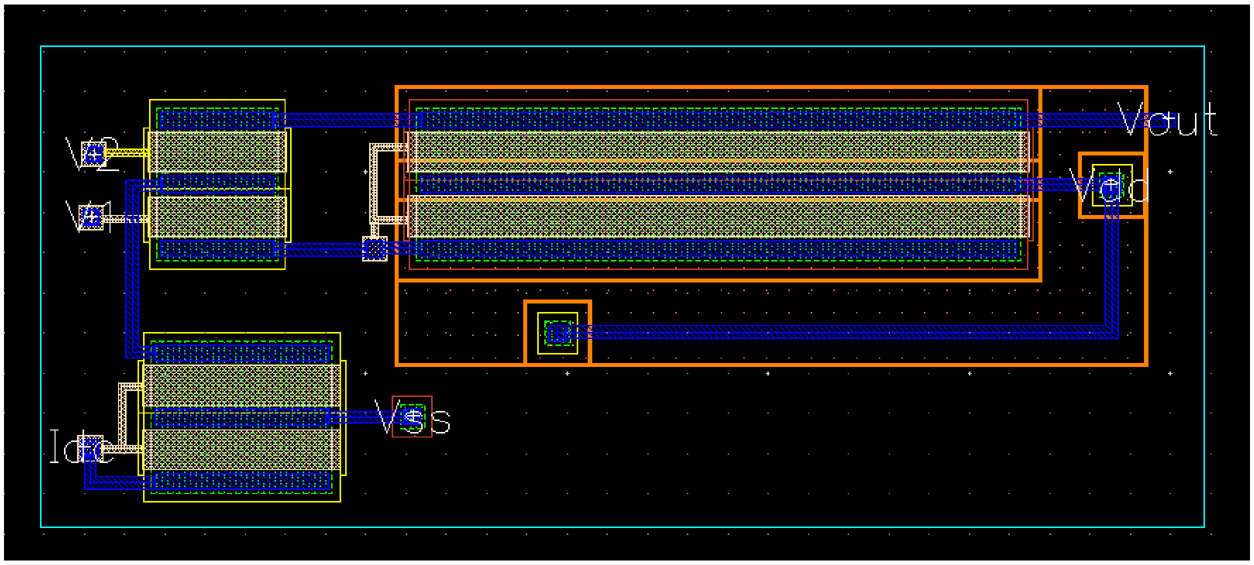


Running the Simulation

1. Execute **Simulation – Netlist and Run** in the simulation window to start the simulation, this will create the netlist as well as run the simulation.
When simulation finishes, the Transient, DC and AC plots automatically will be popped up along with netlist.



Layout



RESULT:

a. The schematic for the differential Amplifier is drawn and verified the following: DC Analysis, DC Analysis, and Transient Analysis.

b. The Layout for the differential amplifier is drawn and verified the DRC, LVS, and RC Extraction.

Lab 3: OPERATIONAL AMPLIFIER

AIM:
To simulate the schematic of the Operational Amplifier, and then to perform the physical verification for the layout of the same

TOOL REQUIRED: Cadence Tool

THEORY:

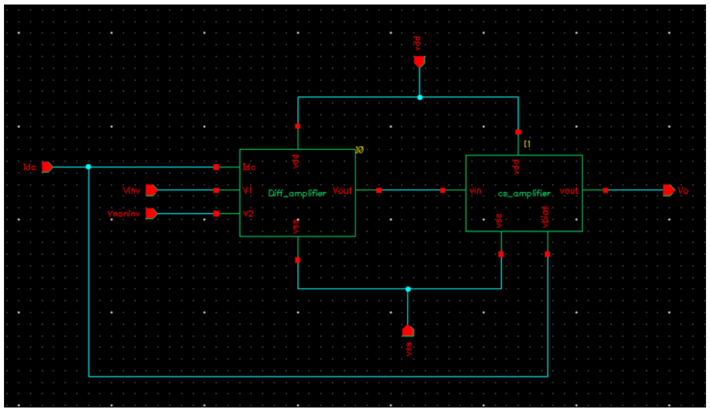
An operational amplifier (often op-amp or opamp) is a DC-coupled high-gain electronic voltage amplifier with a differential input and, usually, a single-ended output. In this configuration, an op-amp produces an output potential (relative to circuit ground) that is typically hundreds of thousands of times larger than the potential difference between its input terminals.

The popularity of the op-amp as a building block in analog circuits is due to its versatility. Due to negative feedback, the characteristics of an op-amp circuit, its gain, input and output impedance, bandwidth etc. are determined by external components and have little dependence on temperature coefficients or manufacturing variations in the op-amp itself.

Op-amps are among the most widely used electronic devices today, being used in a vast array of consumer, industrial, and scientific devices. Op-amps may be packaged as components, or used as elements of more complex integrated circuits.

The op-amp is one type of differential amplifier. The amplifier's differential inputs consist of a non-inverting input (+) with voltage V+ and an inverting input (–) with voltage V−; ideally the op-amp amplifies only the difference in voltage between the two, which is called the differential input voltage. The output voltage of the opamp Vout is given by the equation: Vout = AOL (V+ - V-) where AOL is the open-loop gain of the amplifier (the term "open-loop" refers to the absence of a feedback loop from the output to the input).

Schematic Capture



Schematic Entry

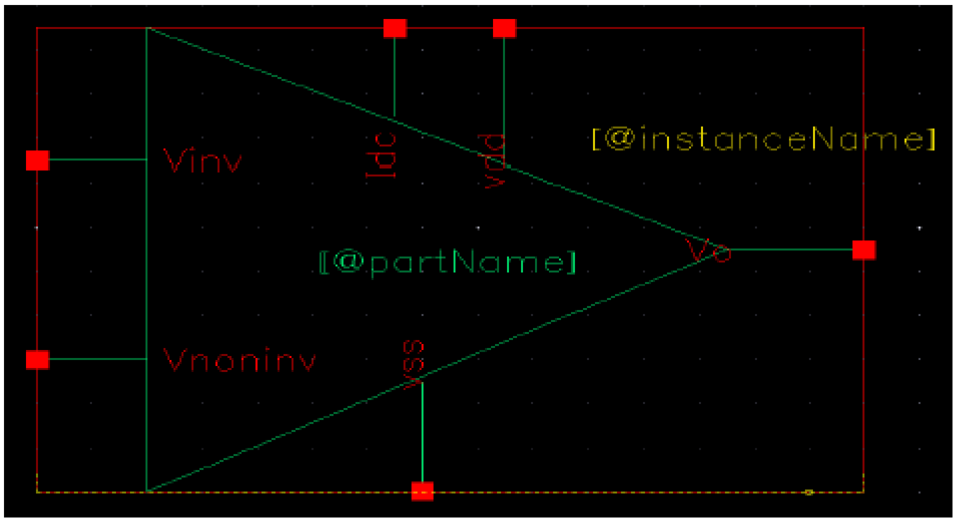
This is a table of components for building the Operational Amplifier schematic.

Library name	Cell Name	Properties/Comments
myDesignLib	Diff_amplifier	Symbol
myDesignLib	cs_amplifier	Symbol

Type the following in the ADD pin form in the exact order leaving space between the pin names.

Pin Names	Direction
Idc,Vinv,Vnoninv	Input
Vo	Output
vdd, vss	Input

Symbol Creation

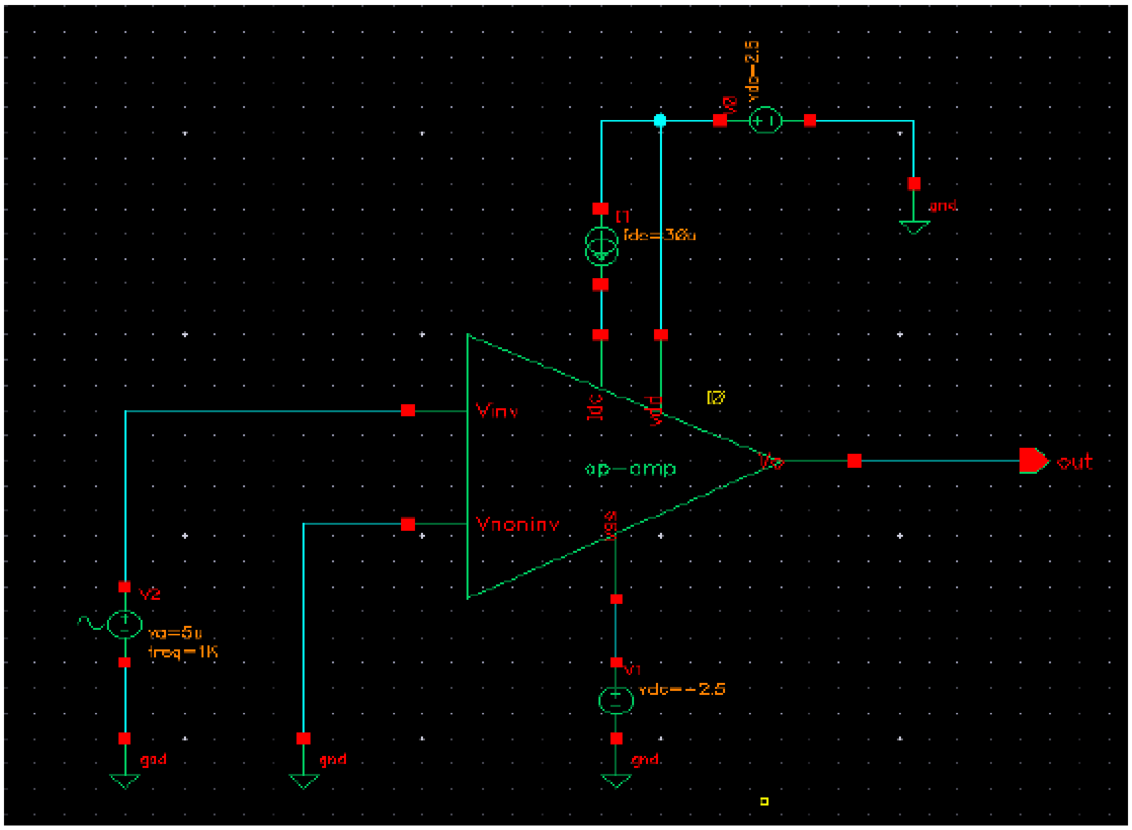


Library name	Cellview name	Properties/Comments
myDesignLib	op-amp	Symbol
analogLib	vsin	Define pulse specification as AC Magnitude= 1; DC Voltage= 0; Offset Voltage= 0; Amplitude= 5m; Frequency= 1K
analogLib	vdc, gnd	vdd=2.5 ; vss= -2.5
analogLib	Idc	Dc current = 30u

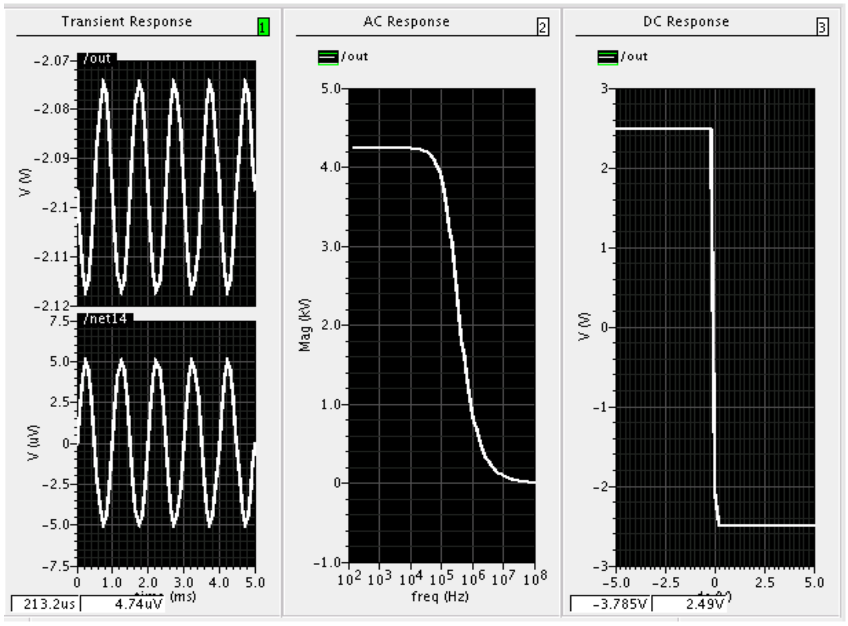
Building the Operational Amplifier Test Design

Using the component list and Properties/Comments in the table, build the op-amp_test schematic as shown below.

Library name	**Cellview name**	**Properties/Comments**
myDesignLib	op-amp	Symbol
analogLib	vsin	Define pulse specification as AC Magnitude= 1; DC Voltage= 0; Offset Voltage= 0; Amplitude= 5m; Frequency= 1K
analogLib	vdc, gnd	vdd=2.5 ; vss= -2.5
analogLib	Idc	Dc current = 30u



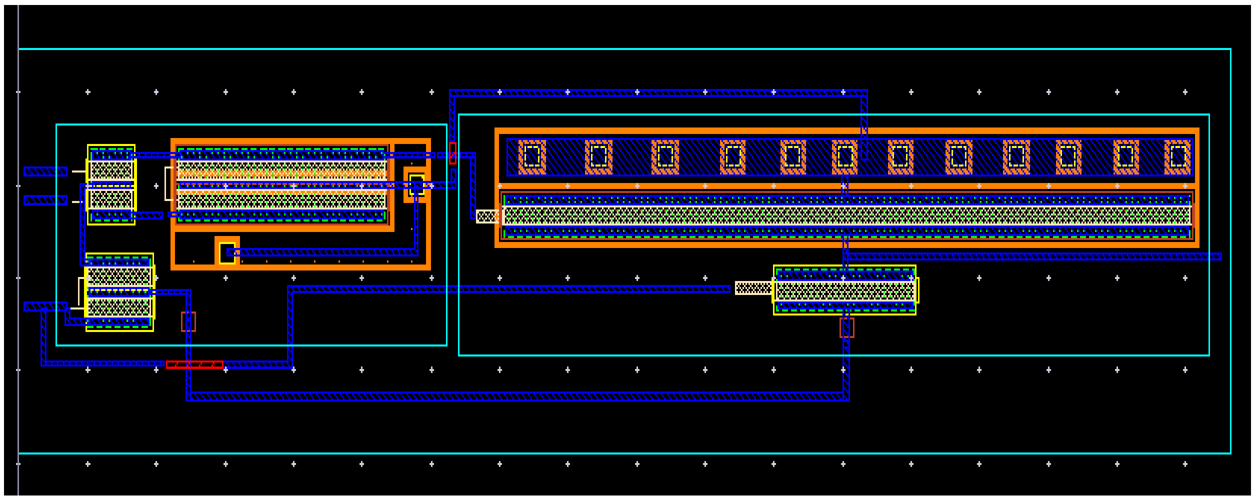
Analog Simulation with Spectre



Creating a layout view of Operational Amplifier

Complete the DRC, LVS check using the assura tool.

Extract RC parasites for back annotation and Re-simulation.



RESULT:

a. The schematic for the Operational Amplifier is drawn and verified the following: DC Analysis, DC Analysis, and Transient Analysis.

b. The Layout for the operational amplifier is drawn and verified the DRC, LVS, and RC Extraction.

Lab 3: R-2R DAC

AIM:

To simulate the schematic of the R-2R DAC, and then to perform the physical verification for the layout of the same.

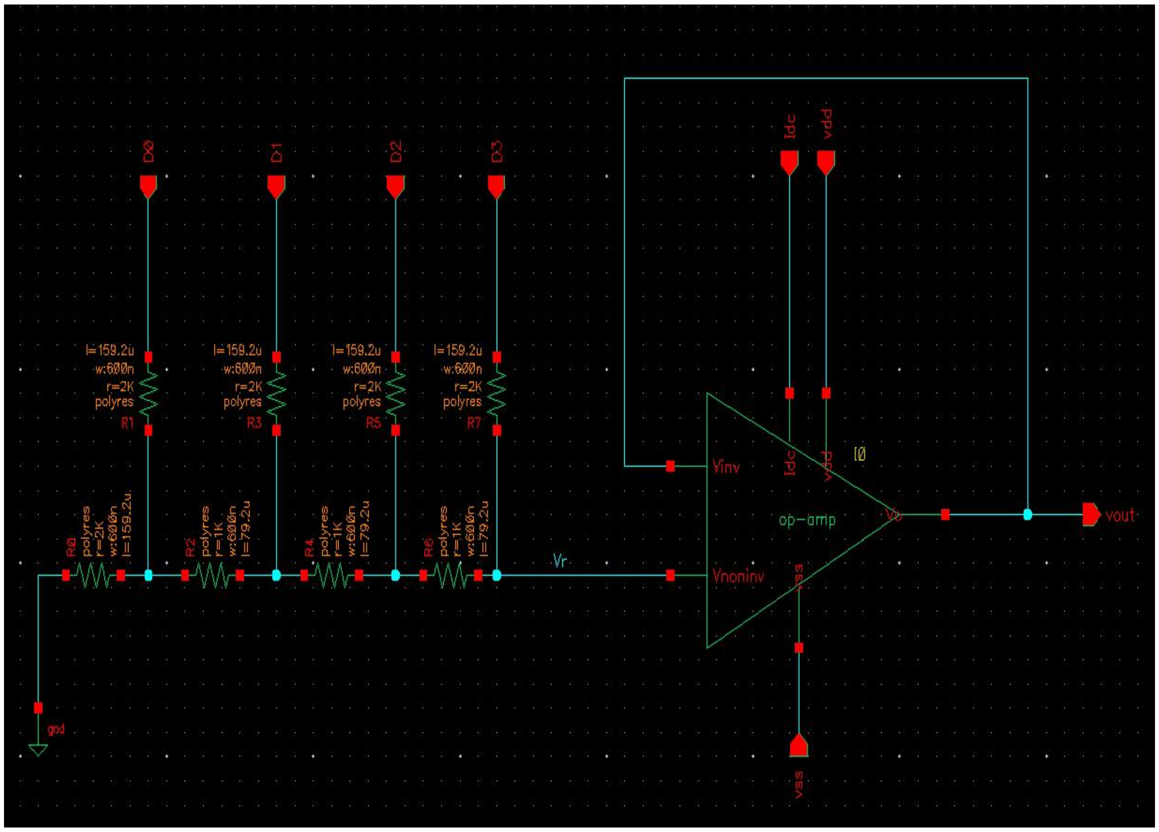
TOOL REQUIRED: Cadence Tool

THEORY:

The R-2R resistor ladder network directly converts a parallel digital symbol/word into an analog voltage. Each digital input (b0, b1, etc.) adds its own weighted contribution to the analog output. This network has some unique and interesting properties.

- Easily scalable to any desired number of bits
- Uses only two values of resistors which make for easy and accurate fabrication and integration
- Output impedance is equal to R, regardless of the number of bits, simplifying filtering and further analog signal processing circuit design

Schematic Capture



Schematic Entry

Use the techniques learned in the Lab1 and Lab2 to complete the schematic of R-2R DAC.

This is a table of components for building the R-2R DAC schematic.

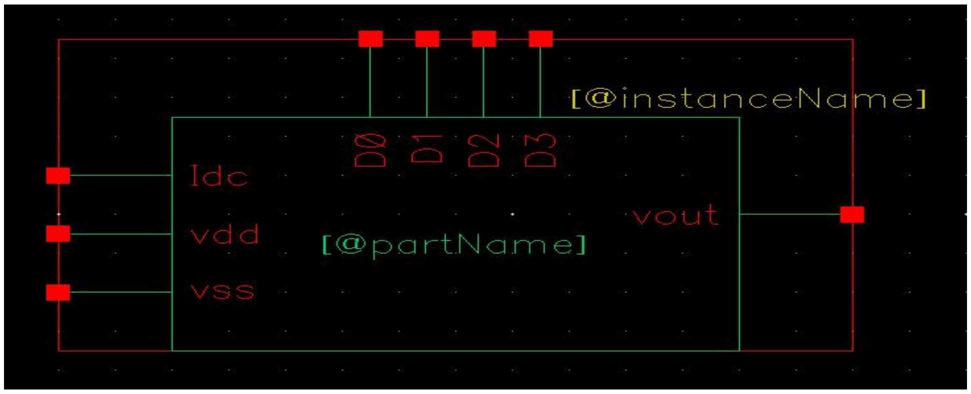
Library name	Cell Name	Properties/Comments
gpdk180	polyres	R = 2k
gpdk180	polyres	R = 1k
MyDesignLib	op-amp	Symbol
analogLib	Idc, gnd	idc = 30u

Type the following in the ADD pin form in the exact order leaving space between the pin names.

Pin Names	Direction
Do D1 D2 D3	Input
Vout	Output
vdd, vss	Input

Symbol Creation

Use the techniques learned in the Lab1 and Lab2 to complete the symbol of R-2R DAC.

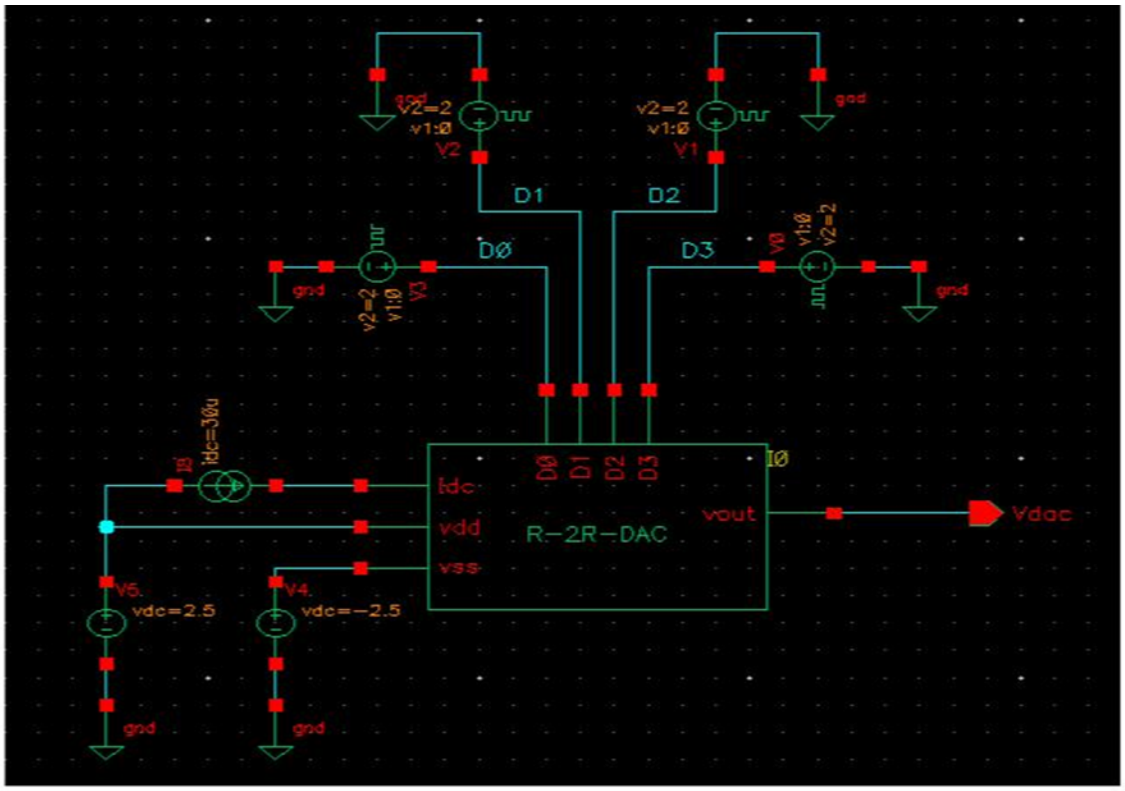


Building the R-2R DAC Test Design

Using the component list and Properties/Comments in the table, build he R-2R-DAC_test schematic as shown below.

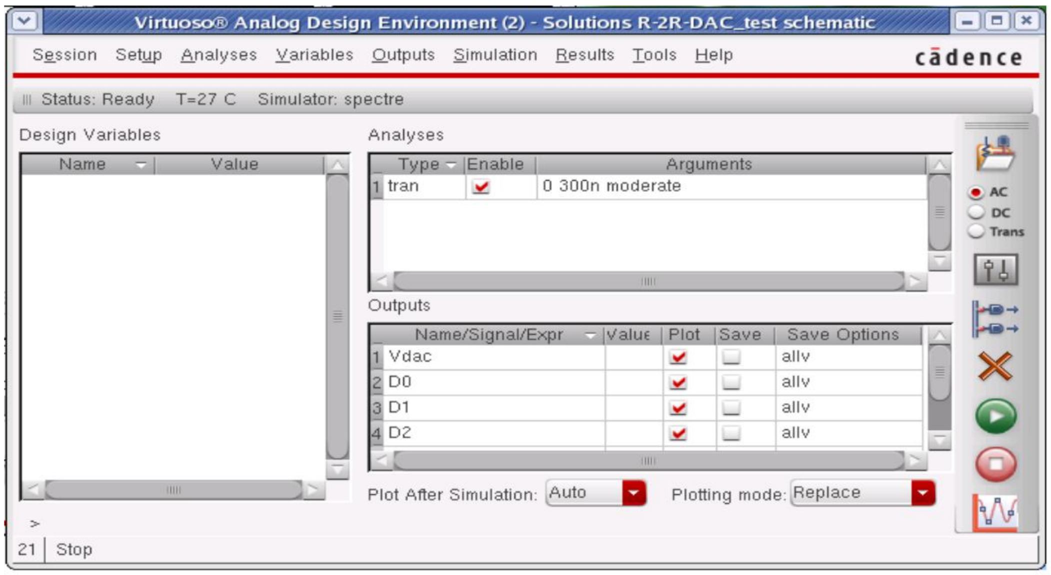
Library name	**Cellview name**	**Properties/Comments**
myDesignLib	R-2R-DAC	Symbol
analogLib	vpulse	For V0: v1= 0 v2 = 2 Ton = 5n T = 10n For V1: v1= 0 v2 = 2 Ton = 10n T = 20n For V2: v1= 0 v2 = 2 Ton = 20n T = 40n For V3: v1= 0 v2 = 2 Ton = 40n T = 80n
analogLib	vdc, gnd	vdd = 2 vss = -2

Note: Remember to set the values for **vdd** and **vss**. Otherwise your circuit will have nopower.



Analog Simulation with Spectre

Use the techniques learned in the Lab1 and Lab2 to complete the simulation of R-2R-DAC and ADE window and waveform should look like below.



Virtuoso® Analog Design Environment (2) - Solutions R-2R-DAC_test schematic

Status: Ready T=27 C Simulator: spectre

Analyses

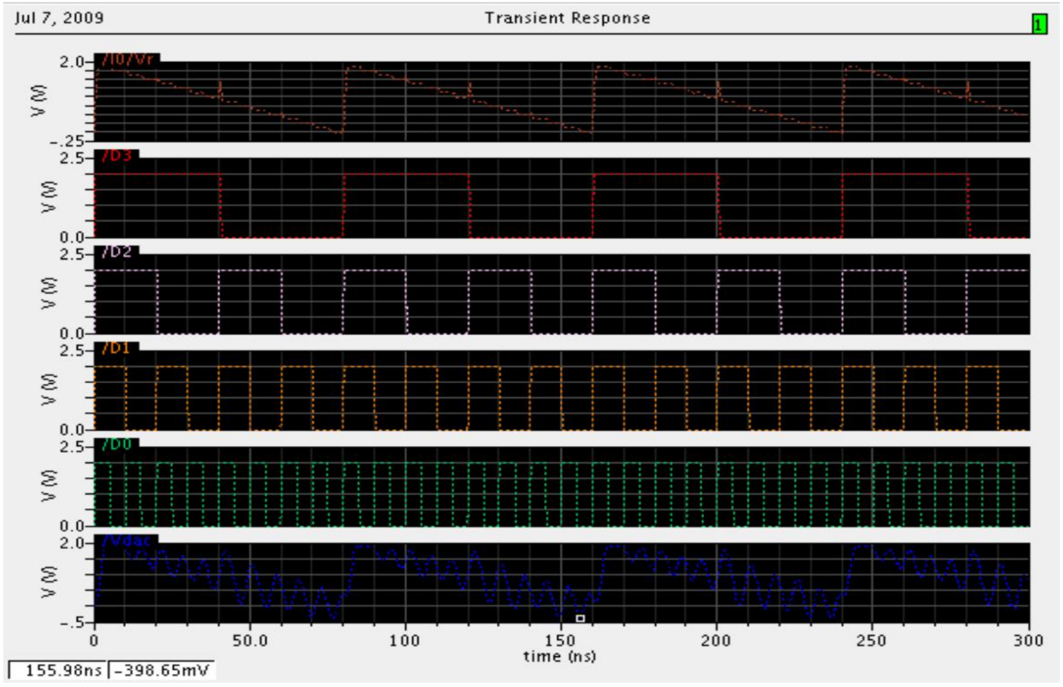
	Type	Enable	Arguments
1	tran	✓	0 300n moderate

Outputs

	Name/Signal/Expr	Value	Plot	Save	Save Options
1	Vdac		✓	☐	allv
2	D0		✓	☐	allv
3	D1		✓	☐	allv
4	D2		✓	☐	allv

Plot After Simulation: Auto Plotting mode: Replace

21 Stop

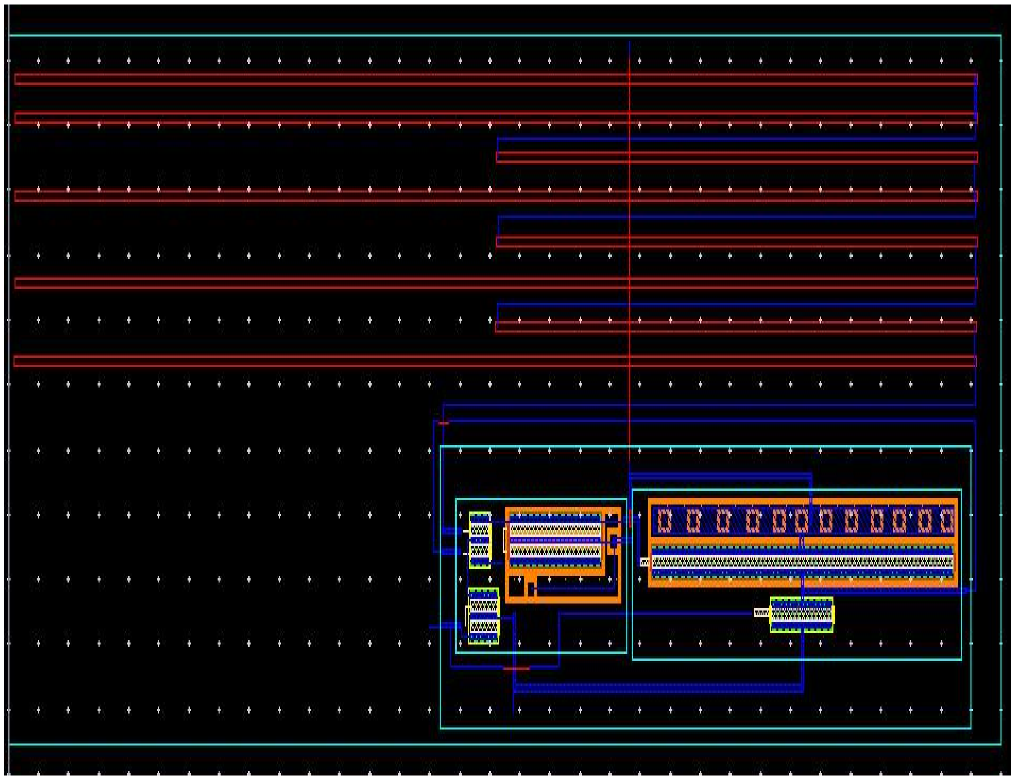


Creating a layout view of R-2R DAC

Use the techniques learned in the Lab1 and Lab2 to complete the layout of R-2R-DAC.

Complete the DRC, LVS check using the assura tool.

Extract RC parasites for back annotation and Re-simulation.



RESULT:

1.The schematic for the operational amplifier is drawn and verified the following:
DC Analysis, DC Analysis, Transient Analysis.
2.The Layout for the operational amplifier is drawn and verified the DRC, LVS, RC Extraction.

VIVA QUESTIONS

1. Why don't we use just one NMOS or PMOS transistor as a transmission gate?
2. What are set up time & hold time constraints? What do they signify?
3. Explain Clock Skew?
4. Why is NAND gate preferred over NOR gate for fabrication?
5. What is Body Effect?
6. Why is the substrate in NMOS connected to Ground and in PMOS to VDD?
7. What is the fundamental difference between a MOSFET and BJT ?
8. Why PMOS and NMOS are sized equally in a Transmission Gates?
9. What happens when the PMOS and NMOS are interchanged with one another in an inverter?
10. Why are pMOS transistor networks generally used to produce high signals, while nMOS networks are used to product low signals?
11. What is Latch Up? Explain Latch Up with cross section of a CMOS Inverter. How do you avoid Latch Up?
12. Difference between Synchronous and Asynchronous reset.
13. What is DRC ?
14. What is LVS ?
15. What is RCX ?
16. What are the differences between SIMULATION and SYNTHESIS?
17. What is a counter?
18. What are the differences between flip-flop and latch?
19. How can you convert JK flip-flop into Jk?
20. What are different types of adders?
21. Give the excitation table for JK flip-flop?
22. Give the excitation table for SR flip-flop?
23. Give the excitation table for D flip-flop?
24. Give the excitation table for T flip-flop?
25. What is the race around condition?
26. What is an amplifier?
27. What is an op-amp?
28. What is differential amplifier?
29. What is elaboration?
30. What is transient analysis?
31. What is DC analysis?
32. What is AC analysis?