PART –B
Digital Design

Getting Started

Cadence can be run only on Unix terminals or PCs loaded with Linux (or UnixTerminal emulators) and X Windows servers like Exceed, X-Win32, or Xfree (Linux). BeforeStarting the Cadence, there are a few configuration files that are needed in the home Working directory. These files determine the environment in which Cadence runs, what libraries are to be included in the current session, etc. These files are in "cshrc" file in the user directory of "cadence".

A work directory has been created for each user when cadence is to be used, so that all the files generated by Cadence user will be creating their directory locally. Ex.: **(you're U.S.N. No.)"**. This will store the Cadence work Environment and files for the present user only. To For sourcing the script file, c-shell is used**.**

1. **Create new folder in desktop. Open the new terminal from the same folder.**

2. For sourcing the script file

⇨ **csh**

⇨ **source /home/installs/cshrc**

3. For creating library file

⇨ **gedit cds.lib** (**Define design_lib ./design.lib**)

⇨ **gedit hdl.var**
(**Define WORK design_lib**
Define NCELABOPTS –messages)
mkdir design.lib

4. For creating Verilog file/testbench file and simulation

⇨ **gedit file_name.v** (To create Verilog file in graphical editor)

⇨ **gedit file_testbench** (To create Verilog Testbench file in vi editor)

⇨ **nclaunch &**

//Compile program and test program

Select filename.v – launch Verilog compiler
Select filename_tb.v – launch Verilog compiler
Select design_lib→ Testbench filename--- Launch Elaborator
Select Snapshots→ design_lib.testbenchfilename.module -→ Launch Simulator
In simulator window → select testbench filename → send to waveform window → run

Note: Verify and close the simulation window

5. For Synthesis

A. Combinational circuits

genus

//Reading the library files
read_lib /home/installs/FOUNDRY/digital/45nm/dig/lib/slow.lib

//PATH OF LIBRARY FILE
set_db lef_library {/home/installs/FOUNDRY/digital/45nm/dig/lef/gsclib045_macro.lef /home/installs/FOUNDRY/digital/45nm/dig/lef/gsclib045_tech.lef}
//(2-lef files need to be copied)

read_hdl filename.v

elaborate

//sdc file (Need to write for all input and output ports which are there in the program)
set_input_delay -max 0.2 [get_ports "a"]
set_input_delay -max 0.2 [get_ports "b"]
set_output_delay -max 0.2 [get_ports "y"]

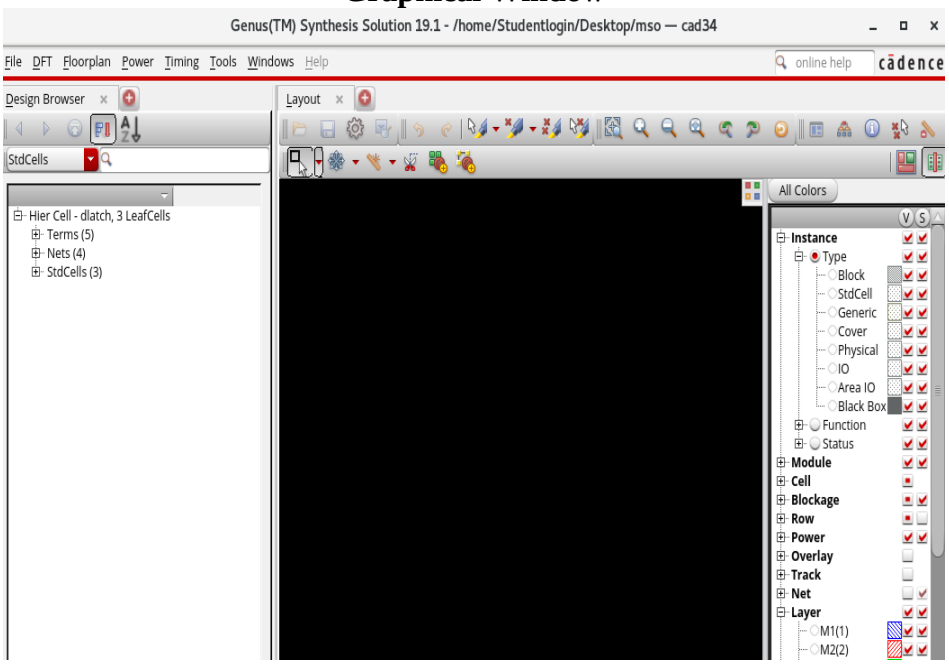
// if sdc file created type
//read_sdc cons.sdc

set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic // Translation: converts program to basic gates
syn_map // Mapping: converts logic to basic gates
syn_opt //optimization
report_power >filename_report.rpt
report_area >filename_area.rpt
report_gates >filename.rpt
report_timing -unconstrained >filename_timing.rpt
gui_show

After this command Graphical Window will open as shown below.

- ⇨ To view schematic: Select Hier Cell and click on schematic view to view schematic
- ⇨ Generate power report
- ⇨ To obtain Area, power and timing report

Graphical Window



B. For Sequential Circuit

genus
//Reading the library files
read_lib /home/installs/FOUNDRY/digital/45nm/dig/lib/slow.lib ///pATH OF LIBRARY FILE

//2-lef files need to be copied

set_db lef_library {/home/installs/FOUNDRY/digital/45nm/dig/lef/gsclib045_macro.lef /home/installs/FOUNDRY/digital/45nm/dig/lef/gsclib045_tech.lef}
read_hdl filename.v
elaborate

//sdc file
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 0.01 [get_ports "clk"]
set_input_delay -max 0.2 [get_ports "j"] -clock [get_clocks "clk"]
set_input_delay -max 0.2 [get_ports "k"] -clock [get_clocks "clk"]

```
set_input_delay -max 0.2 [get_ports "j"]
set_input_delay -max 0.2 [get_ports "k"]
set_output_delay -max 0.2 [get_ports "q"]
set_output_delay -max 0.2 [get_ports "qb"]


// if sdc file created type
//read_sdc cons.sdc

set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic        // Translation: converts program to basic gates
syn_map            // Mapping: converts logic to basic gates
syn_opt            //optimization
report_power >jkff_report.rpt
report_area >jkff_area.rpt
report_gates >jkff_gates.rpt
report_timing -unconstrained >jkff_timing.rpt
gui_show
```

EXPERIMENT 1: ADDERS

Aim: To write a verilog code for 4bit adder and verify the functionality using Test bench.

- Synthesize, Analyse Reports and Netlist, Critical Path and Max Operating Frequency.
- From the report generated find the total number of cells, power requirement and total area requirement.

Tool Required:

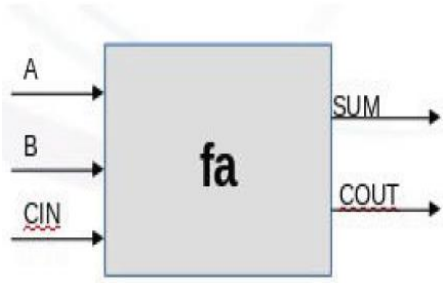
- Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)
- Synthesis: Genus

Design Information and Bock Diagram:

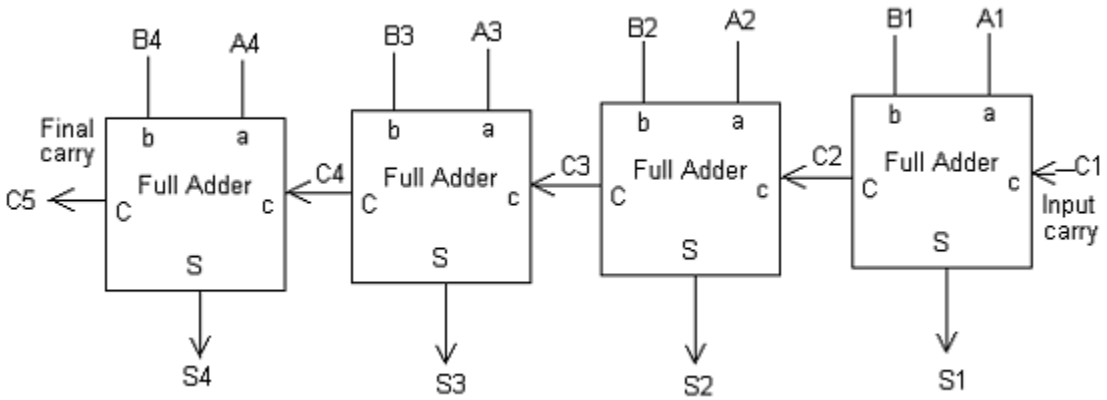
A full adder is a combinational circuit that performs the arithmetic sum of three input bits Ai, addend Bi and carry in C in from the previous adder. Its results contain the sum Si and the carry out, C out to the next stage. So to design a 4-bit adder circuit we start by designing the 1 –bit full adder then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram below. For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY.

1. PARALLEL ADDER

Circuit Diagram:



A	B	C	SUM OUT	CARRY OUT
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



//Verilog code

// Full Adder

```
module fa(a,b,cin,sum,cout);
input a,b,cin;
output sum;
output cout;
assign sum=(a^b)^cin;
assign cout=((a&b)|(b&cin)|(cin&a));
endmodule
```

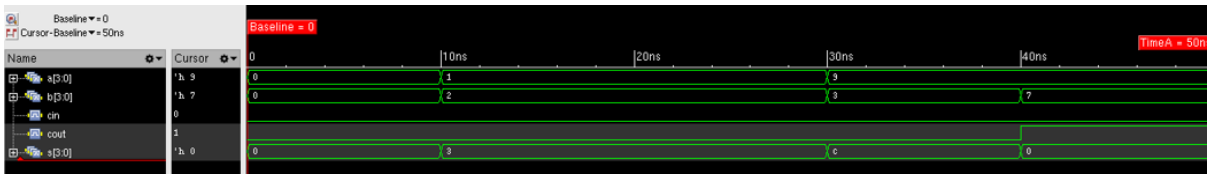
// **Parallel Adder**

```
module pa(a,b,cin,s,cout);
input [3:0]a,b;
input cin;
output [3:0]s;
output cout;
wire [2:0]c;
fa f1(a[0],b[0],cin,s[0],c[0]);
fa f2(a[1],b[1],c[0],s[1],c[1]);
fa f3(a[2],b[2],c[1],s[2],c[2]);
fa f4(a[3],b[3],c[2],s[3],cout);
endmodule
```

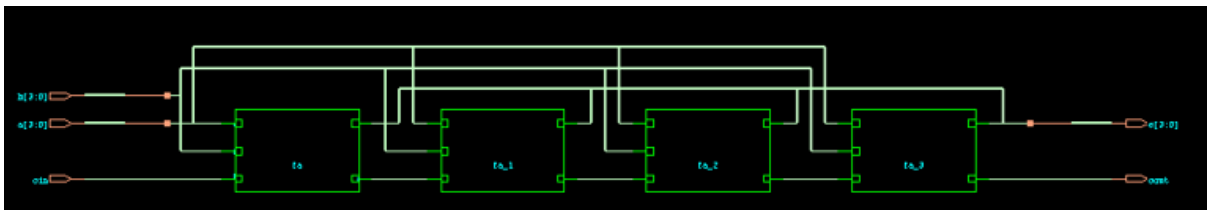
//Test bench

```
module patest;
reg [3:0]a,b;
reg cin=1'b0;
wire cout;
wire [3:0]s;
pa p1(a,b,cin,s,cout);
initial
begin
a=4'b0;
b=4'b0;
#10 a=4'd1;
b=4'd2;
#20 a=4'd9;
b=4'd3;
#10 a=4'd9;
b=4'd7;
#10;
end
endmodule
```

Simulation results



RTL Schematic



Synthesis Result

Power Report

Leakage	Internal	Switching	Total
$6.7E^{-10}$	$1.8E^{-5}$	$2.54E^{-6}$	$1.44E^{-5}$

Timing Report

Input Delay	Data path delay	Arrival time
200	294	494

Area Report

Cell count	Total Area
200	294

EXPERIMENT 2: 32-bit ALU

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

- To Verify the Functionality using Test Bench
- Synthesize and compare the results using if and case statements
- Identify Critical Path and constraints

Tool Required:

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

Design Information and Block Diagram:

The ALU will take in two 32-bit values, and control line. An Arithmetic unit does the following task like addition subtraction, multi-fiction and logical operations. As the input is given in 32 bit we get 32 bit output. The arithmetic will show only one output at a time so a selector is necessary to select one of the operator.

Source Code – Using Case Statement :

```
module alu_32bit_case(y,a,b,f);
input [31:0]a;
input [31:0]b;
input [2:0]f;
output reg [31:0]y;
always@(*)
begin
case(f)
3'b000:y=a&b; //AND Operation
3'b001:y=a|b; //OR Operation
3'b010:y=~(a&b); //NAND Operation
3'b011:y=~(a|b); //NOR Operation
3'b010:y=a+b; //Addition
3'b011:y=a-b; //Subtraction
3'b100:y=a*b; //Multiply
default:y=32'bx;
endcase
end
endmodule
```

Test Bench :

```
module alu_32bit_tb_case;
reg [31:0]a;
reg [31:0]b;
reg [2:0]f;
wire [31:0]y;
alu_32bit_case test2(.y(y),.a(a),.b(b),.f(f));
initial
begin
a=32'h00000000;
b=32'hFFFFFFFF;
#10 f=3'b000;
#10 f=3'b001;
#10 f=3'b010;
#10 f=3'b100;
end
initial
#50 $finish;
endmodule
```

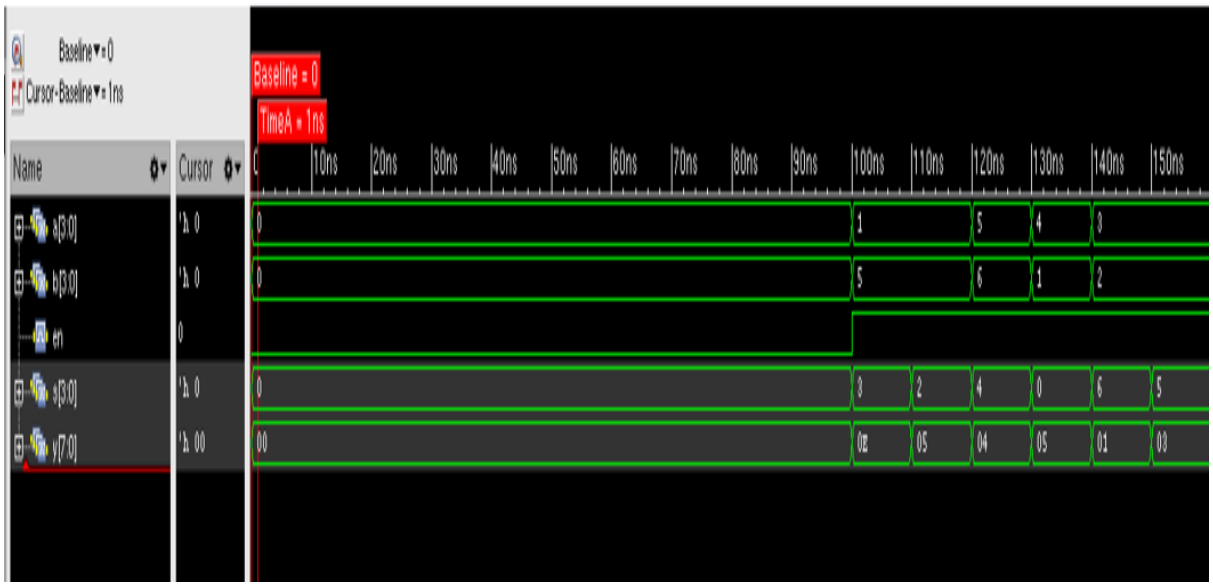
Source Code - Using If Statement :

```
module alu_32bit_if(y,a,b,f);
input [31:0]a;
input [31:0]b;
input [2:0]f;
output reg [31:0]y;
always@(*)
begin
if(f==3'b000)
y=a&b; //AND Operation
else if (f==3'b001)
y=a|b; //OR Operation
else if (f==3'b010)
y=a+b; //Addition
else if (f==3'b011)
y=a-b; //Subtraction
else if (f==3'b100)
y=a*b; //Multiply
else
y=32'bx;
end
endmodule
```

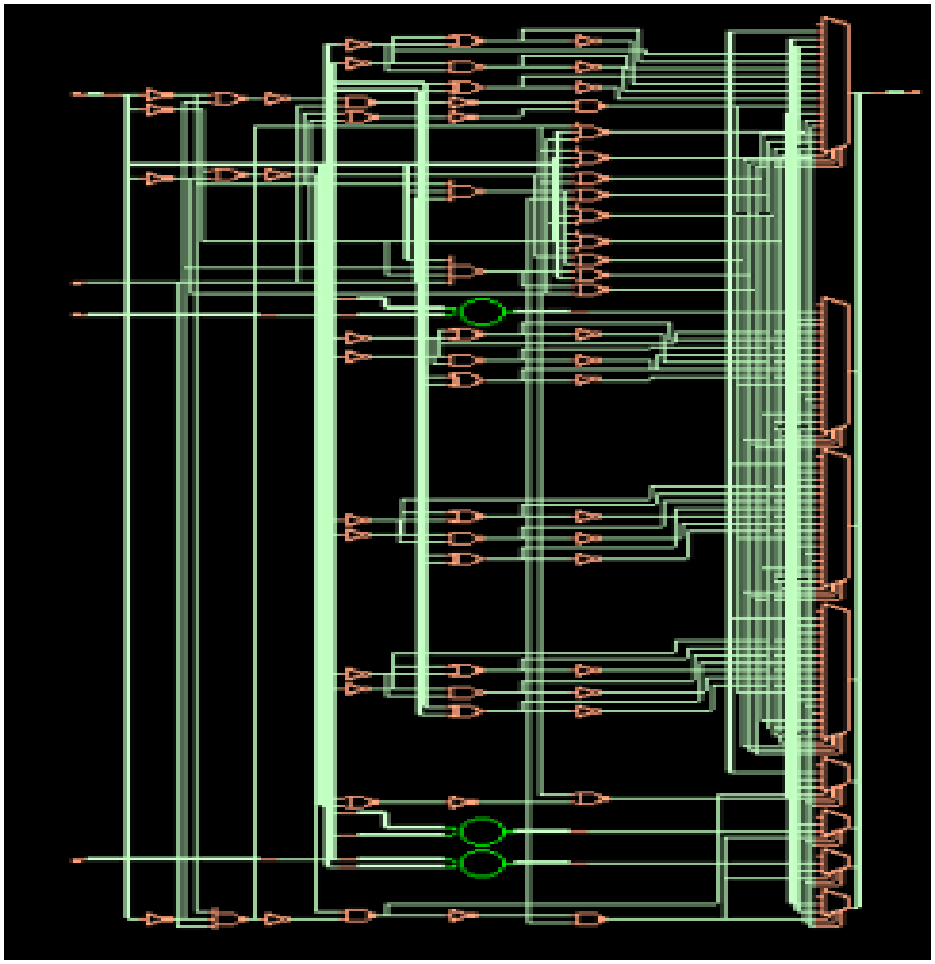
Test bench :

```
module alu_32bit_tb_if;
reg [31:0]a;
reg [31:0]b;
reg [2:0]f;
wire [31:0]y;
alu_32bit_if test(.y(y),.a(a),.b(b),.f(f));
initial
begin
a=32'h00000000;
b=32'hFFFFFFFF;
#10 f=3'b000;
#10 f=3'b001;
#10 f=3'b010;
#10 f=3'b100;
end
initial
#50 $finish;
endmodule
```

Simulation Results



Synthesis Results
RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.215E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

Input Delay	Data path delay	Arrival time
200	9344	9544

Area Report

Cell count	Total Area
1072	1657.3

EXPERIMENT 3: Latch and Flip-Flops

Aim : Write a verilog code for Latch and Flip-flops (D, SR, JK), Synthesize the design and compare the synthesis report.

Tool Required:

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

Design Information and Bock Diagram:

Latches and flip-flops are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted.

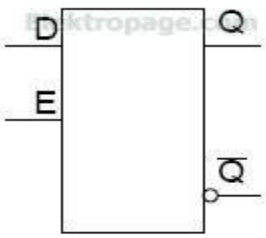
In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.

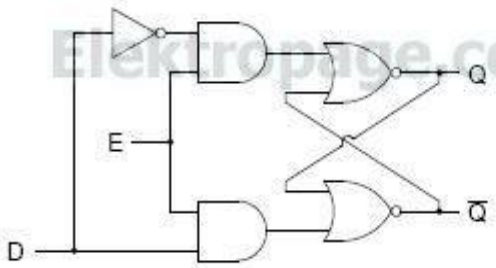
There are basically four main types of latches and flip-flops: SR, D, and JK. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations.

1. D-Latch

Circuit Diagram:

Black Box **Circuit Diagram** **Truth Table**



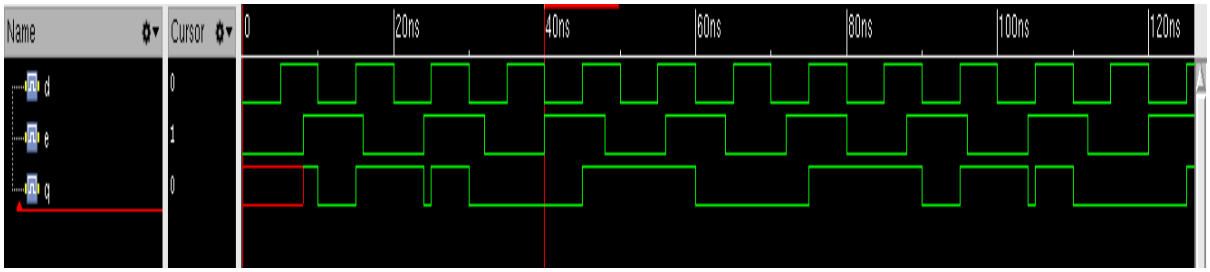


E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

//Verilog code	//Test bench
module DLatch(Q,Qbar,D,en,Reset); output reg Q; output Qbar; input D,en,Reset; assign Reset = ~D; always @(en) begin if (Reset == 1'b1) //If at reset Q <= 1'b0; else Q <= D; end assign Qbar = ~Q; endmodule	module dlatch_tb; reg e,reset; reg d; wire q,qb; dlatch u1(q,qb,d,e,reset); initial begin d = 0; reset=1; e = 0; end always #8 e=~e; always #5 d=~d; initial #20 reset =0; endmodule

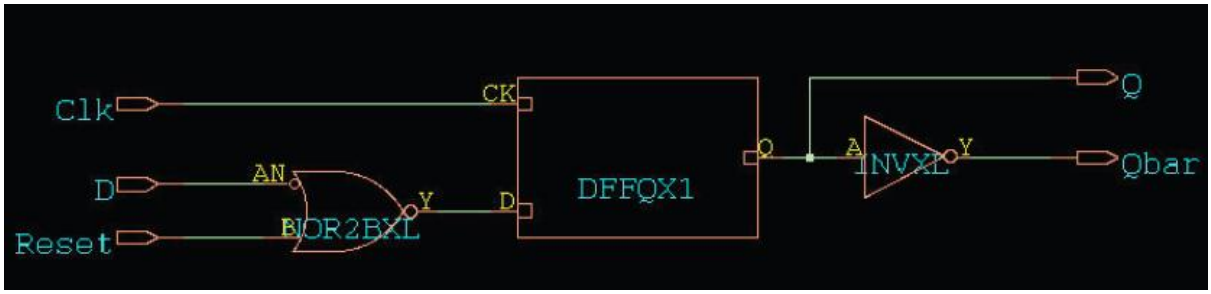
Simulation results

Output Waveform:



Synthesis Results

RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.210E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

Input Delay	Data path delay	Arrival time
200	180	380

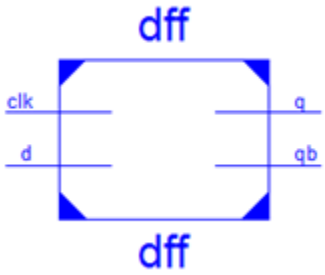
Area Report

Cell count	Total Area
1072	1657.3

2. D-Flip flop

Circuit Diagram:

Black Box



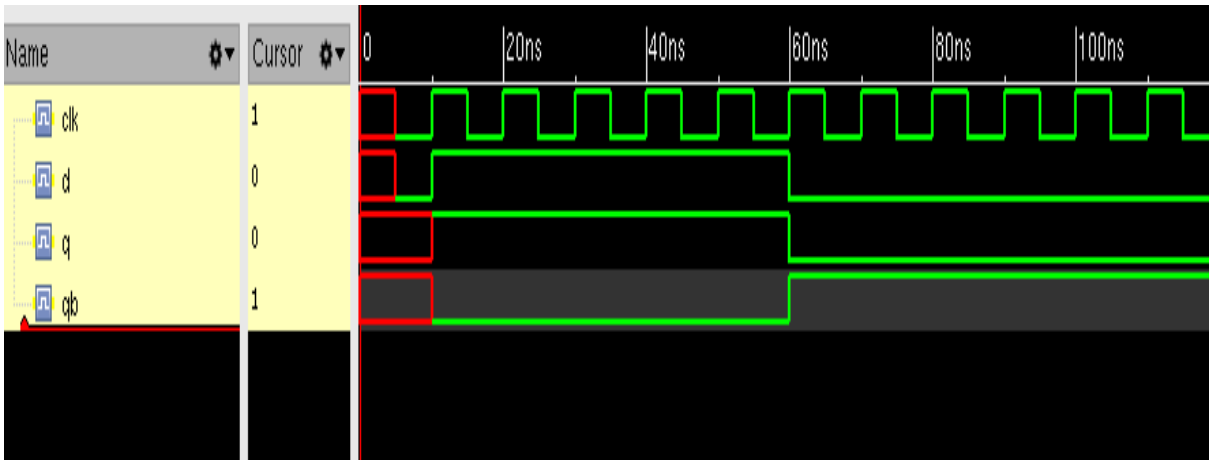
Truth Table

INPUT		OUTPUT	
clk	d	q	qb
X	1	1	0
1	1	1	0
1	0	0	1

//Verilog code module dff(d,clk,q,qb); input d,clk; output q,qb; reg q,qb; always@(posedge clk) begin q=d; qb=~d; end endmodule	**//Test bench** module dfftest; reg clk,d; wire q,qb; dff d1(d,clk,q,qb); initial begin clk = 1'b0; d=1'b0; end always #5 clk=~clk; initial #10 d=1'b0; endmodule

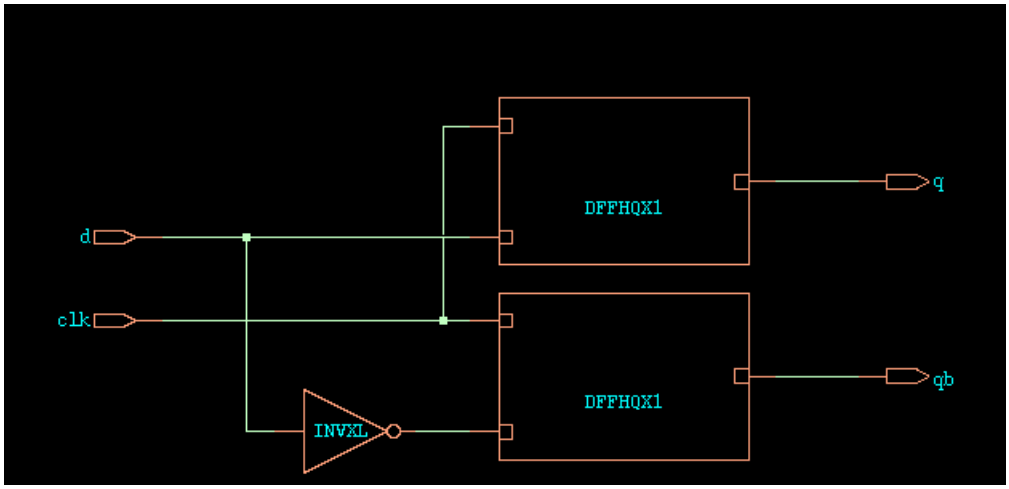
Simulation results

Output Waveform:



Synthesis result

RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.215E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

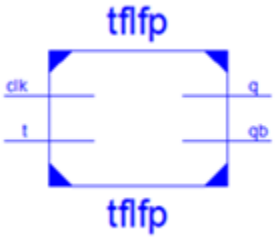
Input Delay	Data path delay	Arrival time
200	9344	9544

Area Report

Cell count	Total Area
1072	1657.3

3. T FLIP-FLOP

Black Box



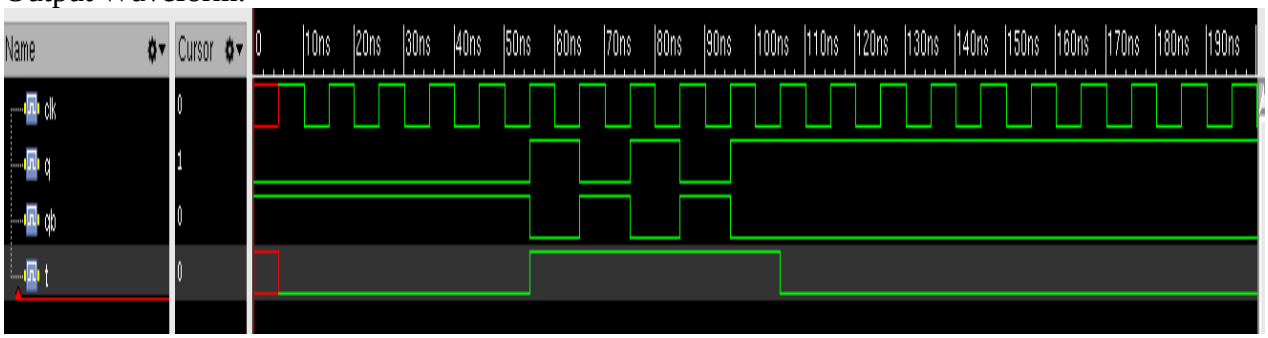
Truth Table

INPUT		OUTPUT	
T	Clk	q	qb
0	1	q	qb
1	1	qb	qb
X	-ve edge	q	qb

//Verilog code	//Test bench
module tff(clk,t,q,qb); input clk,t; output q,qb; reg q=0,qb=1; always @(posedge clk) begin if (t==0) q=q; else q=~q; qb=~q; end endmodule	module tfftest; reg clk,t; wire q,qb; tff t1(clk,t,q,qb); initial clk=1'b0; always #5 clk=~clk; initial begin t=1'b0; #10 t=1'b1; #20 t=1'b1; #20 t=1'b0; #20; end endmodule

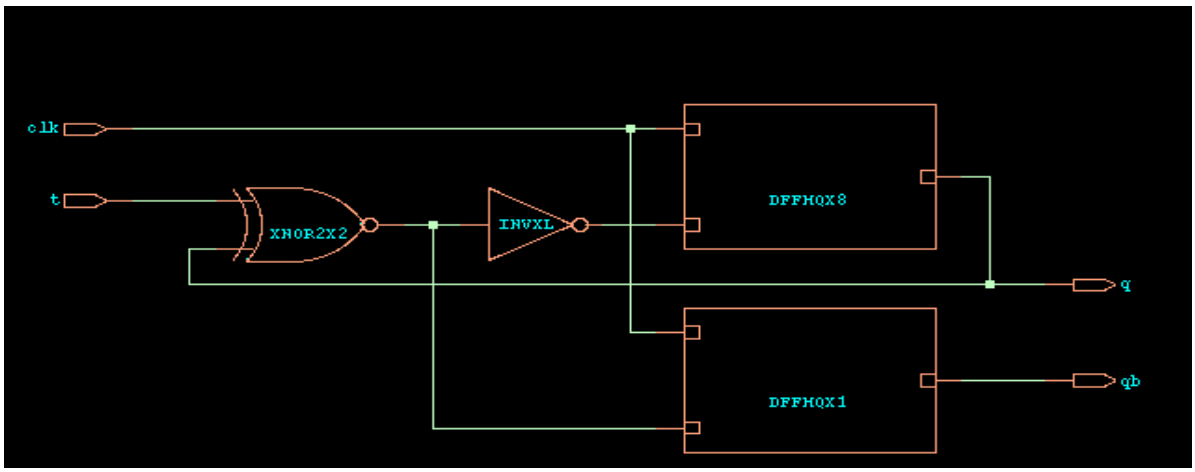
Simulation results

Output Waveform:



Synthesis result

RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.215E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

Input Delay	Data path delay	Arrival time
200	9344	9544

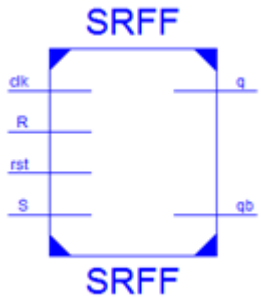
Area Report

Cell count	Total Area
1072	1657.3

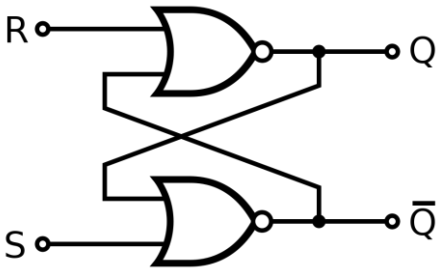
4. SR FLIP-FLOP

Circuit Diagram:

Black Box **Truth Table** **Circuit Diagram**



INPUT				OUTPUT	
rst	Clk	S	R	q	qb
1	X	X	X	0	1
0	X	X	X	1	0
0	1	0	0	Qb	Qbprevious
0	1	0	1	0	1
0	1	1	0	1	0



//Verilog code

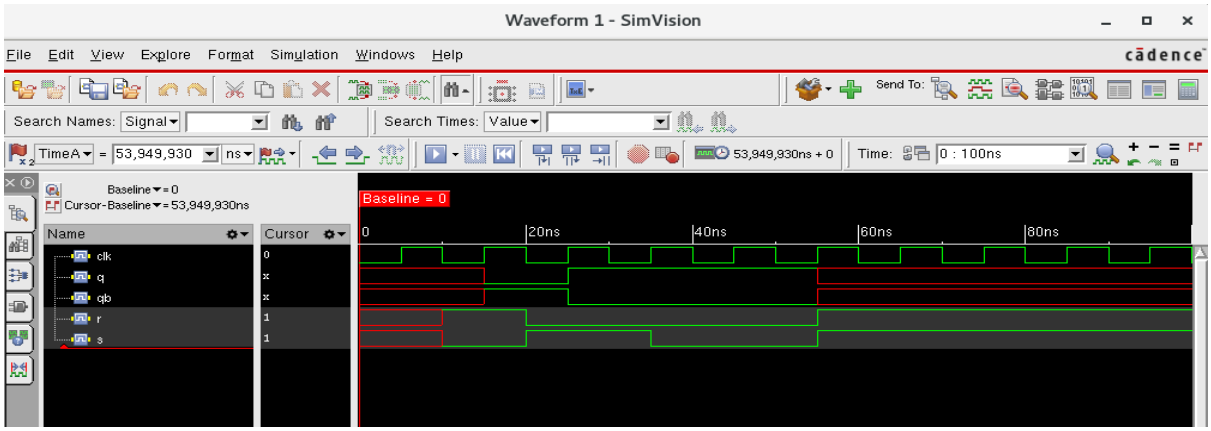
```
module srff(s,r,clk,q,qb);
input s,r,clk; output q,qb;
reg [1:0] sr; reg q,qb;
always @(posedge clk)
begin
sr={s,r};
case(sr)
2'b00:q=q;
2'b01:q=1'b0;
2'b10:q=1'b1;
2'b11:q=1'bx;
endcase
qb=~q;
end
endmodule
```

//Test bench

```
module srfftest;
reg clk,s,r;
wire q,qb;
srff s1(s,r,clk,q,qb);
initial clk=1'b0;
always #5 clk=~clk;
initial
begin
#10 s=1'b0; r=1'b1;
#10 s=1'b1; r=1'b0;
#15 s=1'b0; r=1'b0;
#20 s=1'b1; r=1'b1;
#20;
end
endmodule
```

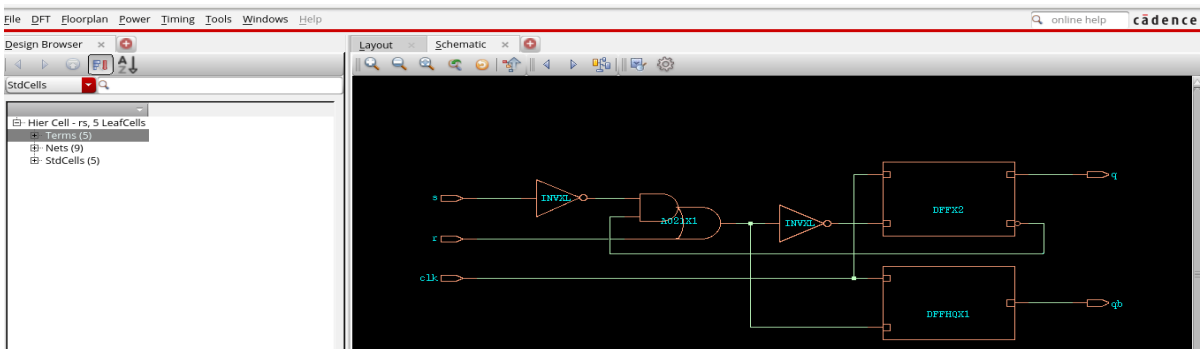
Simulation results

Output Waveform:



Synthesis result

RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.210E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

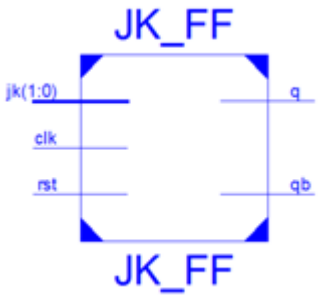
Input Delay	Data path delay	Arrival time
200	180	380

Area Report

Cell count	Total Area
1072	1657.3

5. JK FLIP-FLOP

Black Box



Truth Table

INPUT				OUTPUT	
Rst	Clk	J	K	Q	Qb
0	1	0	0	Previous state	Previous state
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	Qb	Q
0	0	-	-	Previous state	Previous state
1	-	-	-	Previous state	Previous state

//Verilog code

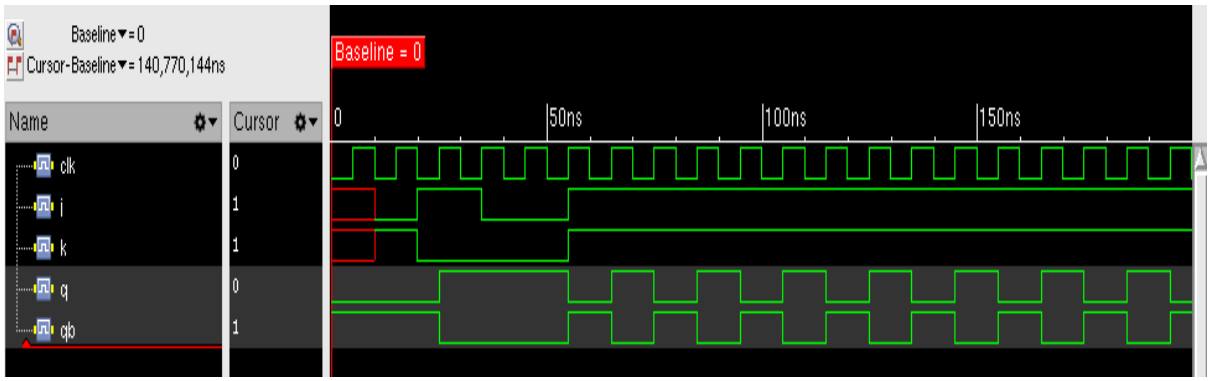
```
module jkff(j,k,clk,q,qb);
input j,k,clk;
output q,qb;
reg[1:0] jk;
reg q=0,qb=1;
always@(posedge clk)
begin
jk={j,k};
case (jk)
2'b00:q=q;
2'b01:q=1'b0;
2'b10:q=1'b1;
2'b11:q=~q;
endcase
qb=~q;
end
endmodule
```

//Test bench

```
module jkfftest;
reg j,k,clk;
wire q,qb;
jkff jk1(j,k,clk,q,qb);
initial clk=1'b0;
always #5 clk=~clk;
initial
begin
j=1'b0; k=1'b0;
#5 j=1'b0; #10 k=1'b1;
#5 j=1'b1;
#5 k=1'b0;
#5 j=1'b1;
#5 k=1'b1;
#20;
end
endmodule
```

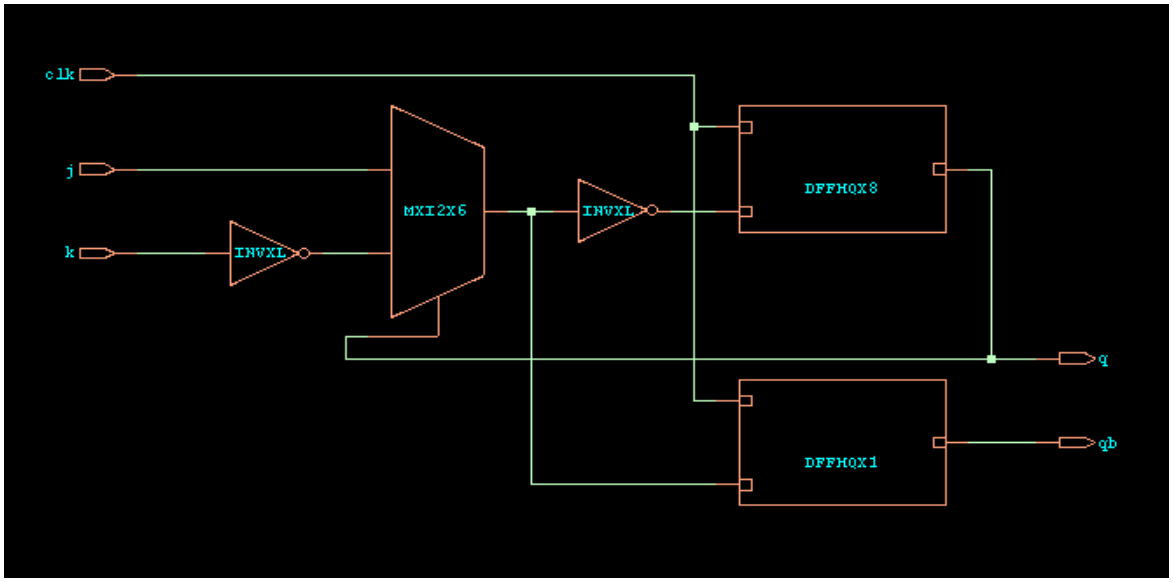
Simulation Results

Output Waveform:



Synthesis results

RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.210E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

Input Delay	Data path delay	Arrival time
200	180	380

Area Report

Cell count	Total Area
1072	1657.3

EXPERIMENT 4: COUNTERS

Aim: To write a verilog code for 4bit up/down asynchronous rest counter and its test-bench for verification.

- Synthesizing the design by setting area and timing constraint and analyse reports.
- Finding the critical path and maximum frequency of operations
- Recording the power, area requirement and properties of each cell in terms of driving strength.

Tool Required:

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

Design Information and Bock Diagram:

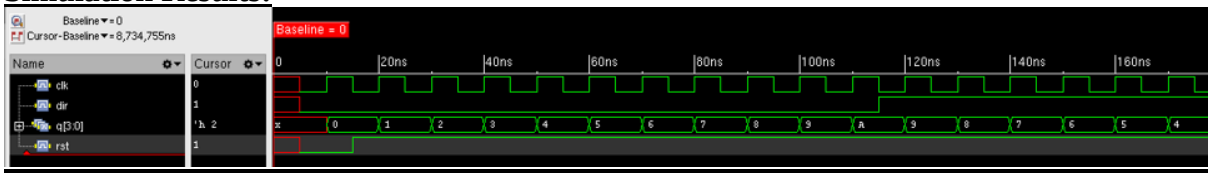
- An up/down counter is a digital counter which can be set to count either from 0 to MAX_VALUE or MAX_VALUE to 0.
- The direction of the count (mode) is selected using a single bit input. The module has 3 inputs - clk, reset which is active high and a UpOrDown mode input. The output is counter which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

ASYNCHRONOUS COUNTER

```
module counter(clk,rst,m,count);
input clk,rst,m;
output reg [3:0]count;
always@(posedge clk or negedge rst)
begin
if(!rst)
count=0;
if(m)
count=count+1;
else
count=count-1;
end
endmodule
```

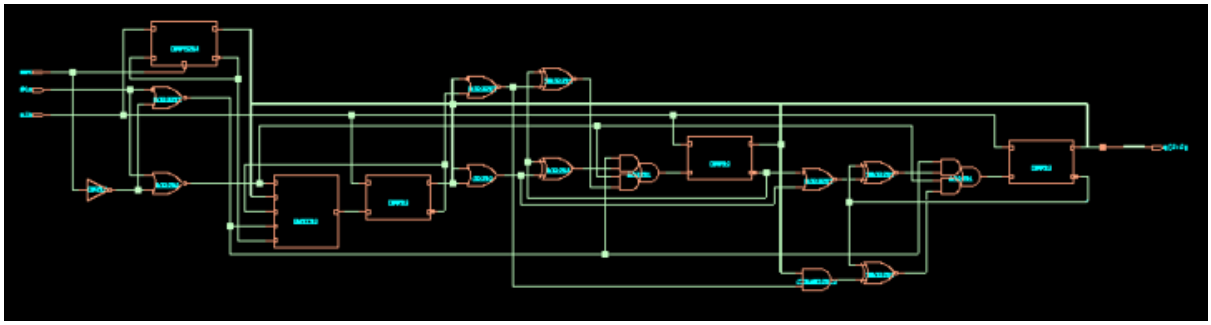
```
module counter_test;
reg clk, rst,m;
wire [3:0]q;
counter C1 (clk,rst,m,count);
initial
begin
clk=1'b0; rst=1'b0; m=1'b0;
end
always
#5 clk=~clk;
initial
begin
#10 rst=1'b1; m=1'b1
#120 m=1'b0;
#100;
end
endmodule
```

Simulation Results:



Synthesis Result

RTL Schematic



Power Report

Leakage	Internal	Switching	Total
$1.210E^{-7}$	$1.24E^{-4}$	$1.64E^{-4}$	$2.89E^{-6}$

Timing Report

Input Delay	Data path delay	Arrival time
200	180	380

Area Report

Cell count	Total Area
1072	1657.3

EXPERIMENT 5: UART

Aim : Write a verilog code for UART and carry out the following:

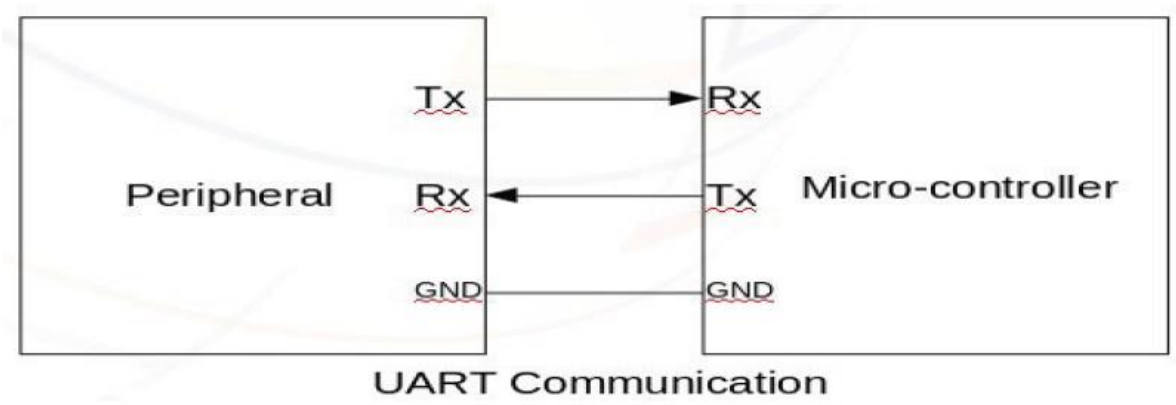
- To Verify the Functionality using test Bench
- Synthesize Design using constraints
- Tabulate Reports using various Constraints
- Identify Critical Path and calculate Max Operating Frequency

Tool Required:

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

Design Information and Bock Diagram:

The **UART** is "Universal Asynchronous Receiver/Transmitter", and it is an inbuilt IC within a micro-controller but not like a communication protocol (I2C & SPI). The main function of UART is to serial data communication. In UART, the communication between two devices can be done in two ways namely serial data communication and parallel data communication. The transmitter section includes three blocks namely transmit hold register, shift register and also control logic. Likewise, the receiver section includes a receive hold register, shift register, and control logic. These two sections are commonly provided by a baud-rate-generator. This generator is used for generating the speed when the transmitter section & receiver section has to transmit or receive the data.



Block diagram of UART

Source Code – Transmitter :

```
// This code contains the UART Transmitter. This transmitter is able
// to transmit 8 bits of serial data, one start bit, one stop bit,
// and no parity bit. When transmit is complete o_Tx_done will be
// driven high for one clock cycle.
// Set Parameter CLKS_PER_BIT as follows:
// CLKS_PER_BIT = (Frequency of i_Clock)/(Frequency of UART)
// Example: 25 MHz Clock, 115200 baud UART
// (25000000)/(115200) = 217
module UART_TX
#(parameter CLKS_PER_BIT = 217)
(
input i_Clock,
```

```
input i_TX_DV,
input [7:0] i_TX_Byte,
output o_TX_Active,
output reg o_TX_Serial,
output o_TX_Done );
parameter IDLE = 3'b000;
parameter TX_START_BIT = 3'b001;
parameter TX_DATA_BITS = 3'b010;
parameter TX_STOP_BIT = 3'b011;
parameter CLEANUP = 3'b100;
reg [2:0] r_SM_Main = 0;
reg [7:0] r_Clock_Count = 0;
reg [2:0] r_Bit_Index = 0;
reg [7:0] r_TX_Data = 0;
reg r_TX_Done = 0;
reg r_TX_Active = 0;
always @(posedge i_Clock)
begin
case (r_SM_Main)
IDLE :
begin
o_TX_Serial <= 1'b1; // Drive Line High for Idle
r_TX_Done <= 1'b0;
r_Clock_Count <= 0;
r_Bit_Index <= 0;
if (i_TX_DV == 1'b1)
begin
r_TX_Active <= 1'b1;
r_TX_Data <= i_TX_Byte;
r_SM_Main <= TX_START_BIT;
end
else
r_SM_Main <= IDLE;
end // case: IDLE
// Send out Start Bit. Start bit = 0
TX_START_BIT :
begin
o_TX_Serial <= 1'b0;
// Wait CLKS_PER_BIT-1 clock cycles for start bit to finish
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main <= TX_START_BIT;
end
else
begin
r_Clock_Count <= 0;
r_SM_Main <= TX_DATA_BITS;
end
end // case: TX_START_BIT
```

```
// Wait CLKS_PER_BIT-1 clock cycles for data bits to finish
TX_DATA_BITS :
begin
o_TX_Serial <= r_TX_Data[r_Bit_Index];
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main <= TX_DATA_BITS;
end
else
begin
r_Clock_Count <= 0;
// Check if we have sent out all bits
if (r_Bit_Index < 7)
begin
r_Bit_Index <= r_Bit_Index + 1;
r_SM_Main <= TX_DATA_BITS;
end
else
begin
r_Bit_Index <= 0;
r_SM_Main <= TX_STOP_BIT;
end
end
end // case: TX_DATA_BITS
// Send out Stop bit. Stop bit = 1
TX_STOP_BIT :
begin
o_TX_Serial <= 1'b1;
// Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main <= TX_STOP_BIT;
end
else
begin
r_TX_Done <= 1'b1;
r_Clock_Count <= 0;
r_SM_Main <= CLEANUP;
r_TX_Active <= 1'b0;
end
end // case: TX_STOP_BIT
// Stay here 1 clock
CLEANUP :
begin
r_TX_Done <= 1'b1;
r_SM_Main <= IDLE;
end
default :
```

```
r_SM_Main <= IDLE;
endcase
end
assign o_TX_Active = r_TX_Active;
assign o_TX_Done = r_TX_Done;
endmodule
```

Source Code – Receiver :

```
// This file contains the UART Receiver. This receiver is able to
// receive 8 bits of serial data, one start bit, one stop bit,
// and no parity bit. When receive is complete o_rx_dv will be
// driven high for one clock cycle.
//
// Set Parameter CLKS_PER_BIT as follows:
// CLKS_PER_BIT = (Frequency of i_Clock)/(Frequency of UART)
// Example: 25 MHz Clock, 115200 baud UART
// (25000000)/(115200) = 217
module UART_RX
#(parameter CLKS_PER_BIT = 217)
(
input i_Clock,
input i_RX_Serial,
output o_RX_DV,
output [7:0] o_RX_Byte
);
parameter IDLE = 3'b000;
parameter RX_START_BIT = 3'b001;
parameter RX_DATA_BITS = 3'b010;
parameter RX_STOP_BIT = 3'b011;
parameter CLEANUP = 3'b100;
reg [7:0] r_Clock_Count = 0;
reg [2:0] r_Bit_Index = 0; //8 bits total
reg [7:0] r_RX_Byte = 0;
reg r_RX_DV = 0;
reg [2:0] r_SM_Main = 0;
// Purpose: Control RX state machine
always @(posedge i_Clock)
begin
case (r_SM_Main)
IDLE :
begin
r_RX_DV <= 1'b0;
r_Clock_Count <= 0;
r_Bit_Index <= 0;
if (i_RX_Serial == 1'b0) // Start bit detected
r_SM_Main <= RX_START_BIT;
else
r_SM_Main <= IDLE;
end
// Check middle of start bit to make sure it's still low
```

```
RX_START_BIT :
begin
if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
begin
if (i_RX_Serial == 1'b0)
begin
r_Clock_Count <= 0; // reset counter, found the middle
r_SM_Main <= RX_DATA_BITS;
end
else
r_SM_Main <= IDLE;
end
else
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main <= RX_START_BIT;
end
end // case: RX_START_BIT
// Wait CLKS_PER_BIT-1 clock cycles to sample serial data
RX_DATA_BITS :
begin
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main <= RX_DATA_BITS;
end
else
begin
r_Clock_Count <= 0;
r_RX_Byte[r_Bit_Index] <= i_RX_Serial;
// Check if we have received all bits
if (r_Bit_Index < 7)
begin
r_Bit_Index <= r_Bit_Index + 1;
r_SM_Main <= RX_DATA_BITS;
end
else
begin
r_Bit_Index <= 0;
r_SM_Main <= RX_STOP_BIT;
end
end
end // case: RX_DATA_BITS
// Receive Stop bit. Stop bit = 1
RX_STOP_BIT :
begin
// Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
r_Clock_Count <= r_Clock_Count + 1;
```

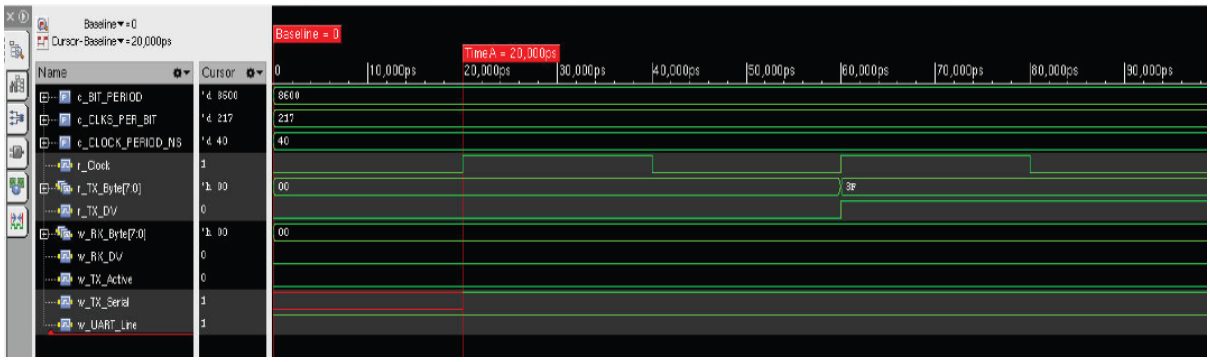
```
r_SM_Main <= RX_STOP_BIT;
end
else
begin
r_RX_DV <= 1'b1;
r_Clock_Count <= 0;
r_SM_Main <= CLEANUP;
end
end // case: RX_STOP_BIT
// Stay here 1 clock
CLEANUP :
begin
r_SM_Main <= IDLE;
r_RX_DV <= 1'b0;
end
default :
r_SM_Main <= IDLE;
endcase
end
assign o_RX_DV = r_RX_DV;
assign o_RX_Byte = r_RX_Byte;
endmodule // UART_RX
```

Test bench :

```
// This testbench will exercise the UART RX.
// It sends out byte 0x37, and ensures the RX receives it correctly.
`timescale 1ns/10ps
`include "uart_tx.v"
`include "uart_rx.v"
module UART_TB ();
// Testbench uses a 25 MHz clock
// Want to interface to 115200 baud UART
// 25000000 / 115200 = 217 Clocks Per Bit.
parameter c_CLOCK_PERIOD_NS = 40;
parameter c_CLKS_PER_BIT = 217;
parameter c_BIT_PERIOD = 8600;
reg r_Clock = 0;
reg r_TX_DV = 0;
wire w_TX_Active, w_UART_Line;
wire w_TX_Serial;
reg [7:0] r_TX_Byte = 0;
wire [7:0] w_RX_Byte;
UART_RX #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_RX_Inst
(.i_Clock(r_Clock), .i_RX_Serial(w_UART_Line), .o_RX_DV(w_RX_DV),
.o_RX_Byte(w_RX_Byte) );
UART_TX #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_TX_Inst
(.i_Clock(r_Clock), .i_TX_DV(r_TX_DV), .i_TX_Byte(r_TX_Byte),
.o_TX_Active(w_TX_Active), .o_TX_Serial(w_TX_Serial), .o_TX_Done() );

// Keeps the UART Receive input high (default) when
```

```
// UART transmitter is not active
assign w_UART_Line = w_TX_Active ? w_TX_Serial : 1'b1;
always
#(c_CLOCK_PERIOD_NS/2) r_Clock <= !r_Clock;

// Main Testing:
initial
begin

// Tell UART to send a command (exercise TX)
@(posedge r_Clock);
@(posedge r_Clock);
r_TX_DV <= 1'b1;
r_TX_Byte <= 8'h3F;
@(posedge r_Clock);
r_TX_DV <= 1'b0;
end
endmodule
```

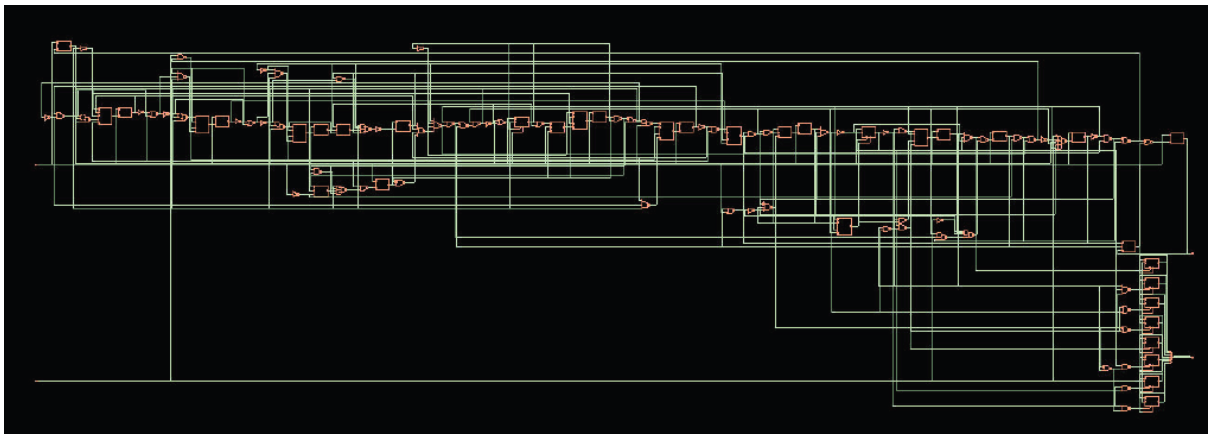
Simulation Results



Synthesize Results

1. read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
2. read_hdl {uart_tx.v / uart_rx.v} //Choose any one
3. elaborate
4. read_sdc constraints_top.sdc //Reading Top Level SDC
5. set_db syn_generic_effort medium //Setting effort medium
6. set_db syn_map_effort medium
7. set_db syn_opt_effort medium
8. syn_generic
9. syn_map
10. syn_opt //Performing Synthesis Mapping and Optimisation
11. report_timing > uart_timing.rep
//Generates Timing report for worst datapath and dumps into file
12. report_area > uart_area.rep
//Generates Synthesis Area report and dumps into a file
13. report_power > uart_power.rep

//Generates Power Report [Pre-Layout]
14. report_qor > uart_qor.rep
15. write_hdl > uart_netlist.v
//Creates readable Netlist File
16. write_sdc > uart_sdc.sdc
//Creates Block Level SDC



Note :-
1. You can tabulate Area, Power and Timing Constraints using any of the SDC Constraints as instructed.
2. Make sure, during synthesis the Report File Names are changed so that the latest reports do not overwrite the earlier ones.

Experiment 6: Physical Design

Aim: For the synthesized netlist carry out the following any two above experiments:

- Floor planning, identify the placement of pads, placement and Routing

Tool Required:

- Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)
- Synthesis: Genus
- Physical Design: Innovus

Mandatory Inputs for PD:

1. Gate Level Netlist [Output of Synthesis]
2. Block Level SDC [Output of Synthesis]
3. Liberty Files (.lib)
4. LEF Files (Layer Exchange Format)

Expected Outputs from PD:

1. GDS II File (Graphical Data Stream for Information Interchange – Feed In for Fabrication Unit).
2. SPEF, SDF

- Make sure the Synthesis for the target design is done and open a terminal from the corresponding workspace.
- Initiate the Cadence tools and **cmd :**innovus (Press Enter)
- For Innovus tool, a GUI opens and also the terminal enters into innovus command prompt where in the tool commands can be entered.

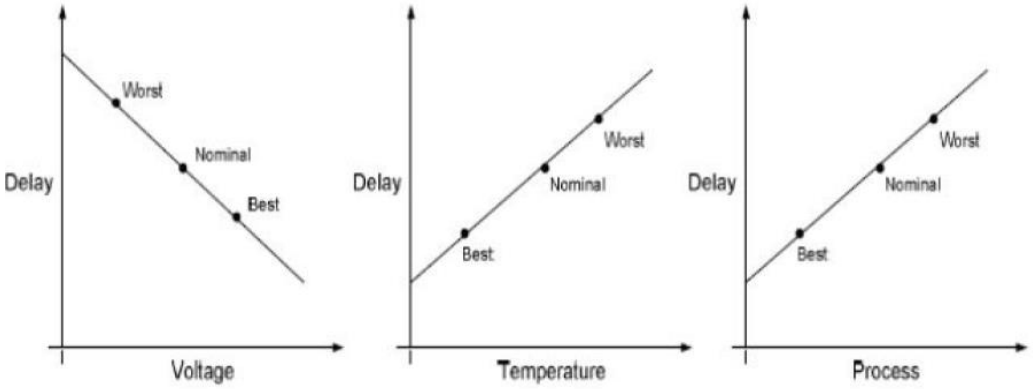
Physical Design involves 5 stages as following :
After Importing Design,
→ Floor Planning
→ Power Planning
→ Placement
→ CTS (Clock Tree Synthesis)
→ RoutingModule

Importing Design

To Import Design, all the Mandatory Inputs are to be loaded and this can be done either using script files named with .globals and .view/.tcl or through GUI as shown below.

The target design considered here is 4bit up down counter design
The procedure shall remain the same for any other design from the above discussed experiments.

Note :
1. For Synthesis, slow.lib was read as input. Each liberty file contains a pre-defined Process, Voltage and Temperature (PVT) values which impact the ease of charge movement.
2. Process, Voltage and Temperature individually affect the ease of currents as depicted below.



3. Hence, slow.lib contains PVT combination (corner) with **slow** charge movement => **Maximum** Delay => **Worst** Performance
4. Similarly, fast.lib contains PVT Combination applicable across its designs to give **Fast** charge movement => **Minimum** Delay => **Best** Performance.
5. When these corners are collaborated with the sdc, they can be used to analyse timing for setup in the worst case and hold in the best case.
6. All these analysis views are to be manually created either in the form of script or using the GUI.

Script file of Default.globals file

```
#################################################################
#  Generated by:      Cadence Encounter 13.23-s047_1
#  OS:                Linux x86_64(Host ID cadence)
#  Generated on:      Tue May 24 02:16:38 2016
#  Design:
#  Command:           save_global Default.globals
#################################################################
#
# Version 1.1
#

set ::TimeLib::tsgMarkCellLatchConstructFlag 1
set conf_qxconf_file {NULL}
set conf_qxlib_file {NULL}
set defHierChar {/}
set init_design_settop 0
set init_gnd_net {VSS}
set init_lef_file {lef/gsclib090_translated.lef lef/gsclib090_translated_ref.lef}
set init_mmmc_file {Default.view}
set init_pwr_net {VDD}
set init_verilog {counter_netlist.v}
set lsgOCPGainMult 1.000000
set pegDefaultResScaleFactor 1.000000
set pegDetailResScaleFactor 1.000000
```

Script file of Default.view (or) Default.tcl file

```
# Version:1.0 MMMC View Definition File
# Do Not Remove Above Line
create_library_set -name MAX_timing -timing {/root/Desktop/counter/lib/90/slow.lib}
create_library_set -name MIn_timing -timing {/root/Desktop/counter/lib/90/fast.lib}
create_constraint_mode -name Constraints -sdc_files {counter_sdc.sdc}
create_delay_corner -name Max_delay -library_set {MAX_timing}
create_delay_corner -name Min_delay -library_set {MIn_timing}
create_analysis_view -name Worst -constraint_mode {Constraints} -delay_corner {Max_delay}
create_analysis_view -name best -constraint_mode {Constraints} -delay_corner {Min_delay}
set_analysis_view -setup {Worst} -hold {best}
```