DIGITAL IC DESIGN LAB (EC1062)

LAB ASSIGNMENT - 2

Submitted By:

ANKIT KUMAR

24EC4224

Department: Microelectronics and VLSI

Submitted To:

Dr. Hemanta Kumar Mondal

Assistant Professor, Department of ECE, NIT Durgapur

Submitted on-

Sign-____________

Design 1: Edge-triggered Flip-Flops (D, J-K, SR, T, Master-Slave)

D Flip-Flop

Objective:
Design and simulate the edge-triggered D Flip-Flop using Verilog.

Specification:

- **Function:** The D flip-flop stores the input value (D) on the rising edge of the clock. The output (Q) follows the D input when the clock edge occurs.
- **Characteristic Equation:**
  $Q(t+1) = D$
  This equation indicates that the next state of Q (at $t+1$) is equal to the D input at time t.
- **Truth Table:**

Clock	D	Q (Next State)
↑	0	0
↑	1	1

J-K Flip-Flop

Objective:
Design and simulate the edge-triggered J-K Flip-Flop using Verilog.

Specification:

- **Function:** The J-K flip-flop toggles the output (Q) when both J and K are high. If J = 1 and K = 0, Q is set. If J = 0 and K = 1, Q is reset.
- **Characteristic Equation:**
  $Q(t+1) = J\overline{Q}(t) + \overline{K}Q(t)$
  This equation describes how the next state $Q(t+1)$ is a function of J, K, and the current state $Q(t)$.
- **Truth Table:**

Clock	J	K	Q (Next State)
↑	0	0	Q (No change)
↑	0	1	0
↑	1	0	1
↑	1	1	$\overline{Q}$ (Toggle)

SR Flip-Flop

Objective:
Design and simulate the edge-triggered SR Flip-Flop using Verilog.

Specification:

- **Function:** The SR flip-flop is a basic set-reset flip-flop. When S = 1 and R = 0, the output is set (Q = 1). When S = 0 and R = 1, the output is reset (Q = 0). Both S and R should not be high simultaneously.
- **Characteristic Equation:**
  $Q(t+1) = S + \overline{R}Q(t)$
  This equation represents the set and reset actions, where the next state depends on the inputs S and R.
- **Truth Table:**

Clock	S	R	Q (Next State)
↑	0	0	Q (No change)
↑	0	1	0
↑	1	0	1
↑	1	1	Invalid (X)

T Flip-Flop

Objective:
Design and simulate the edge-triggered T Flip-Flop using Verilog.

Specification:

- **Function:** The T flip-flop toggles the output (Q) whenever the T input is high at the rising edge of the clock. If T = 0, there is no change.
- **Characteristic Equation:**
  $Q(t+1) = T\overline{Q}(t) + \overline{T}Q(t)$
  The next state $Q(t+1)$ is a toggle function when T = 1, and remains unchanged when T = 0.
- **Truth Table:**

Clock	T	Q (Next State)
↑	0	Q (No change)
↑	1	$\overline{Q}$ (Toggle)

Master-Slave J-K Flip-Flop

Objective:
Design and simulate the Master-Slave J-K Flip-Flop using Verilog.

Specification:

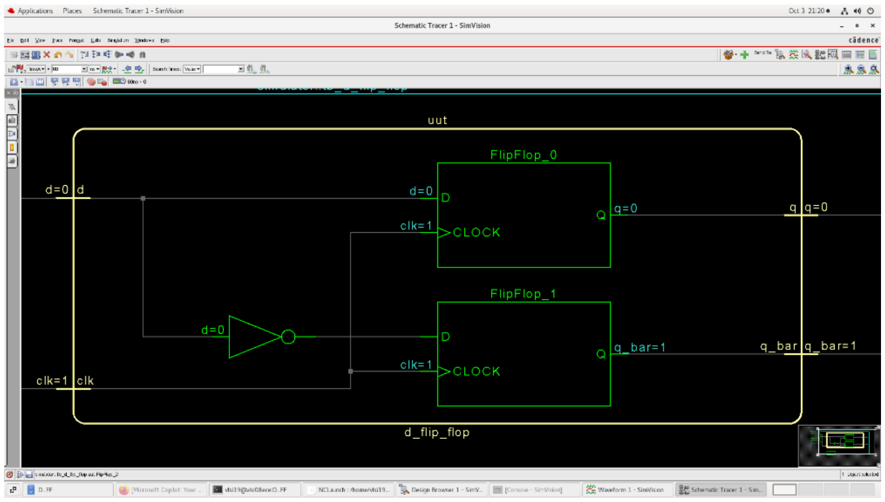
- **Function:** The Master-Slave J-K flip-flop consists of two J-K flip-flops connected in series. The master flip-flop is triggered on the rising edge of the clock, while the slave is triggered on the falling edge. This ensures that any changes in the output happen only after the clock cycle is complete, eliminating the possibility of race conditions.
- **Characteristic Equation:**
  $Q(t+1) = J\overline{Q}(t) + \overline{K}Q(t)$
  This characteristic equation is the same as the standard J-K flip-flop, but the key difference is that the master-slave configuration ensures output changes only after a full clock cycle.
- **Truth Table:**

Clock	J	K	Master (Q*)	Slave (Q) (Next State)
↑	0	0	Q* (No change)	Q (No change)
↑	0	1	0	0
↑	1	0	1	1
↑	1	1	$\overline{Q\backslash *}$ (Toggle)	$\overline{Q}$ (Toggle)

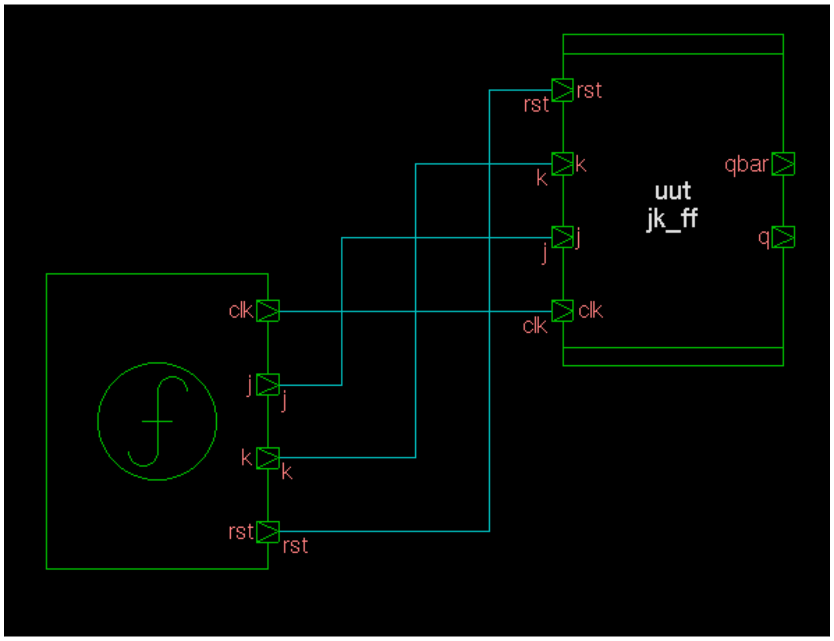
Sign-____________

Block _level Implementation

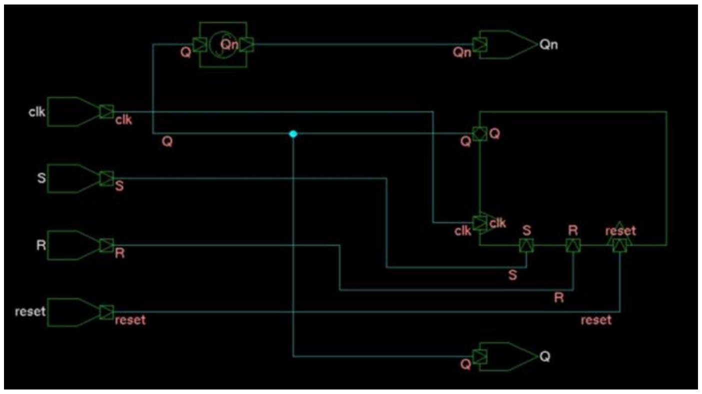
Block-Level Implementation for D Flip-Flop



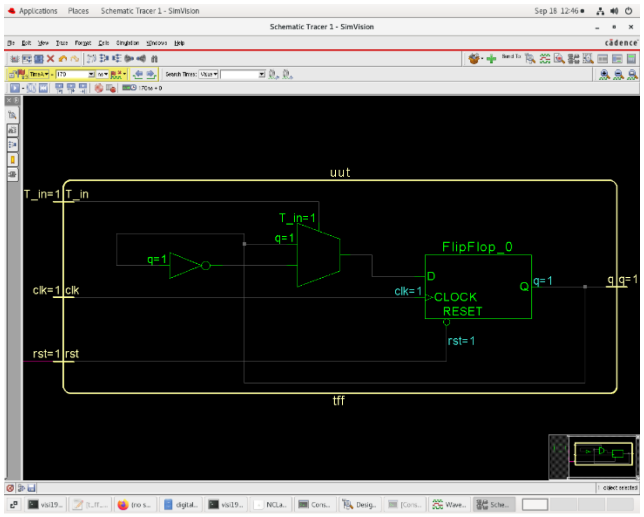
Block-Level Implementation for J-K Flip-Flop



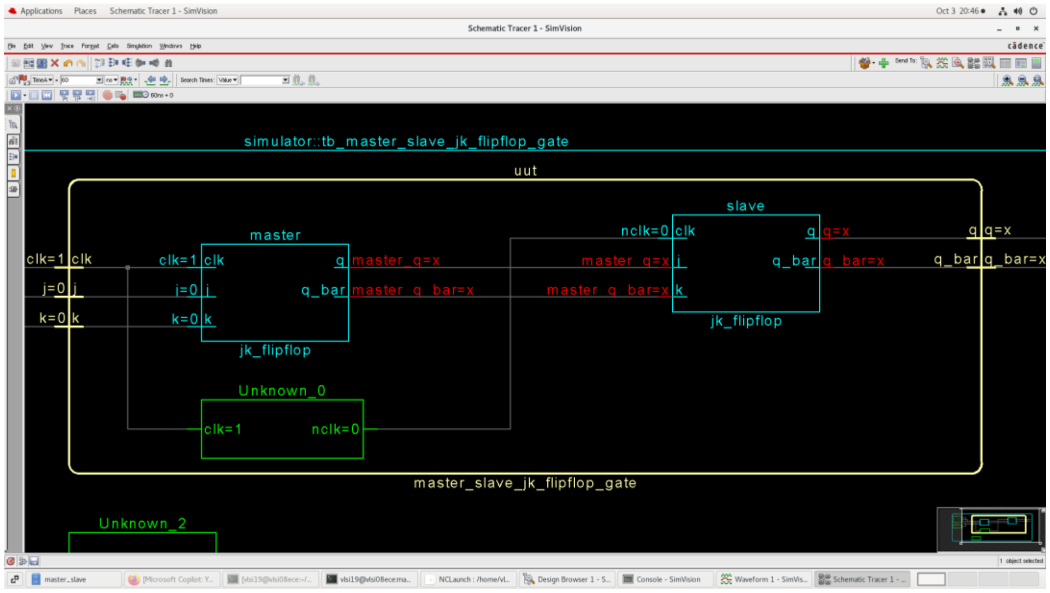
Block-Level Implementation for SR Flip-Flop



Block-Level Implementation for T Flip-Flop



Block-Level Implementation for Master-Slave J-K Flip-Flop

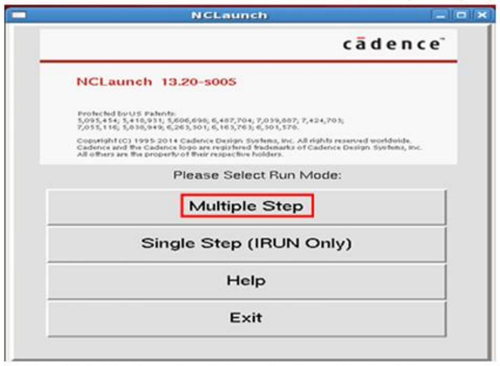


Sign-____________

Simulation Setup (for all flip-flops using NC Launch in Cadence)

Verification
IES (Incisive Enterprise Simulator) is the tool used for verification. Navigate to the Simulation directory where you have kept your RTL and test bench (simulation directory).

➢ Invoke the tool by typing 'nclaunch -new' in the terminal. NCLaunch window will appear like in the below screen shot and in the NCLaunch window, select 'Multiple Step' option.



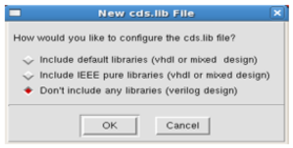
Protected by US Patents:
5,095,454; 5,418,931; 5,606,698; 6,487,704; 7,039,887; 7,424,703;
7,055,116; 5,838,949; 6,263,301; 6,163,763; 6,301,578.

Copyright (C) 1995-2014 Cadence Design Systems, Inc. All rights reserved worldwide.
Cadence and the Cadence logo are registered trademarks of Cadence Design Systems, Inc.
All others are the property of their respective holders.

➢ On clicking the 'Multiple Step' option, 'nclaunch: Open Design Directory window will appear as shown below

➢ Click on 'Create cds.lib File' option and a 'Create a cds.lib file' window will open. Click the 'Save' option.

➢ A 'New cds.lib File' window will appear. Click any of the three options available depending on your RTL and click 'OK'. As the counter design is in verilog, the third option is selected.



How would you like to configure the cds.lib file?

- Include default libraries (vhdl or mixed design)
- Include IEEE pure libraries (vhdl or mixed design)
- [✓ Don't include any libraries (verilog design)]

➢ Click 'OK' in the 'nclaunch: Open Design Directory' window.

➢ In the NCLaunch window, we will be able to see the design as well as the testbench that we kept inside the simulation directory.

Compilation

➢ The next step is to compile (Checks syntax and semantics) the code. For this, select both the design and testbench and choose the appropriate compilers.

Elaboration:

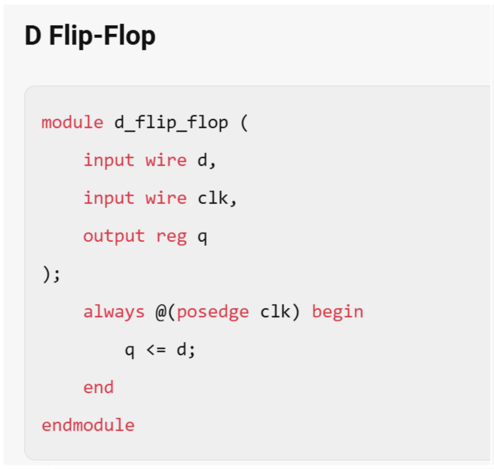
1. Fix errors, then perform **elaboration** to build the design hierarchy and connect signals.
2. After successful compilation, open **worklib** to view design objects.
3. Elaborate the **testbench** (top module) by selecting it and clicking **'launch elaborator'** (**ncelab**).

Simulation:

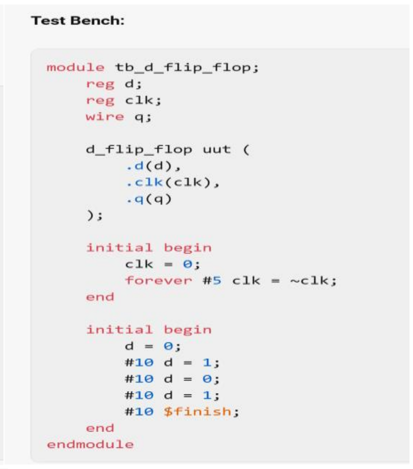
1. Send the elaboration **snapshot** to the simulator.
2. Open **snapshots**, select the snapshot, and click **'launch simulator'**.
3. In the **Design Browser**, select the testbench and choose **'waveform'**.
4. Use **SimVision** to run the simulation and analyze the waveform.
5. Click **'Run'**, and use **pause**, **zoom**, and other tools for debugging and analysis.

Sign-____________

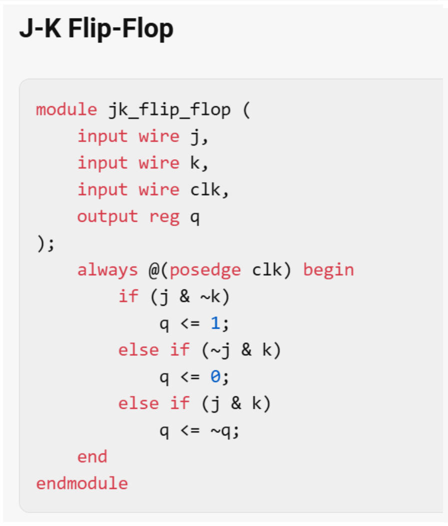
RTL CODING


D Flip-Flop

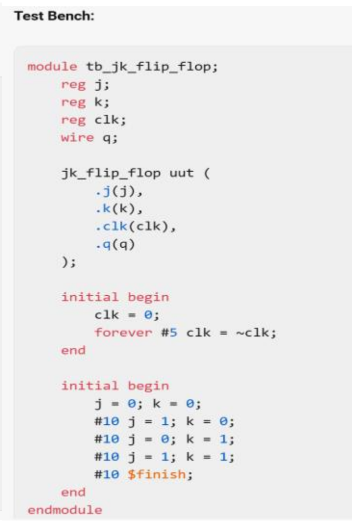
```
module d_flip_flop (
    input wire d,
    input wire clk,
    output reg q
);
    always @(posedge clk) begin
        q <= d;
    end
endmodule
```


Test Bench:

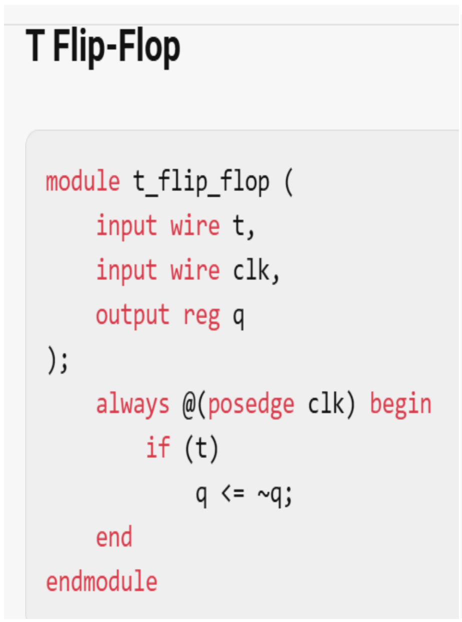
```
module tb_d_flip_flop;
    reg d;
    reg clk;
    wire q;

    d_flip_flop uut (
        .d(d),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        d = 0;
        #10 d = 1;
        #10 d = 0;
        #10 d = 1;
        #10 $finish;
    end
endmodule
```


J-K Flip-Flop

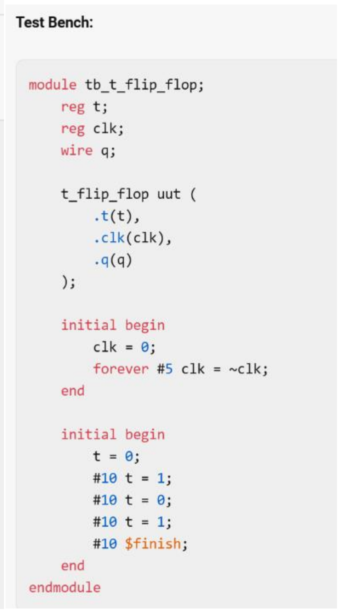
```
module jk_flip_flop (
    input wire j,
    input wire k,
    input wire clk,
    output reg q
);
    always @(posedge clk) begin
        if (j & ~k)
            q <= 1;
        else if (~j & k)
            q <= 0;
        else if (j & k)
            q <= ~q;
    end
endmodule
```


Test Bench:

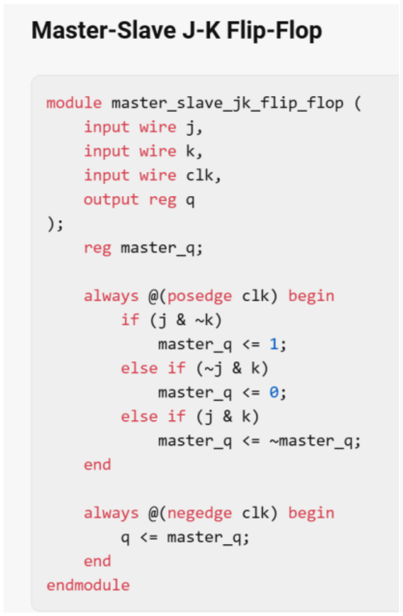
```
module tb_jk_flip_flop;
    reg j;
    reg k;
    reg clk;
    wire q;

    jk_flip_flop uut (
        .j(j),
        .k(k),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        j = 0; k = 0;
        #10 j = 1; k = 0;
        #10 j = 0; k = 1;
        #10 j = 1; k = 1;
        #10 $finish;
    end
endmodule
```


T Flip-Flop

```
module t_flip_flop (
    input wire t,
    input wire clk,
    output reg q
);
    always @(posedge clk) begin
        if (t)
            q <= ~q;
    end
endmodule
```

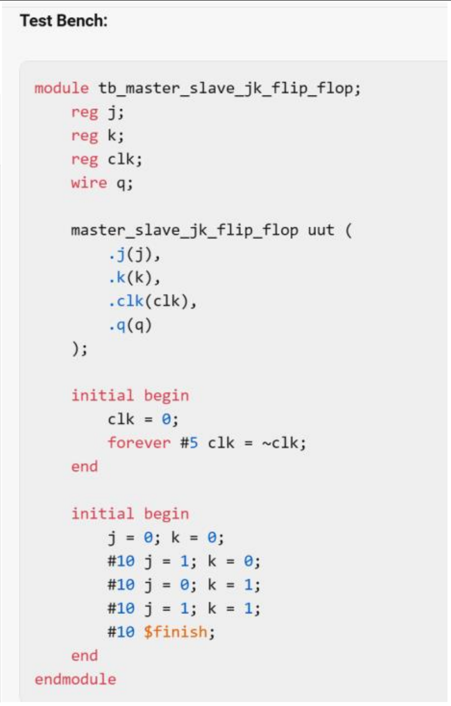

Test Bench:

```
module tb_t_flip_flop;
    reg t;
    reg clk;
    wire q;

    t_flip_flop uut (
        .t(t),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        t = 0;
        #10 t = 1;
        #10 t = 0;
        #10 t = 1;
        #10 $finish;
    end
endmodule
```



Master-Slave J-K Flip-Flop

```
module master_slave_jk_flip_flop (
    input wire j,
    input wire k,
    input wire clk,
    output reg q
);
    reg master_q;

    always @(posedge clk) begin
        if (j & ~k)
            master_q <= 1;
        else if (~j & k)
            master_q <= 0;
        else if (j & k)
            master_q <= ~master_q;
    end

    always @(negedge clk) begin
        q <= master_q;
    end
endmodule
```



Test Bench:

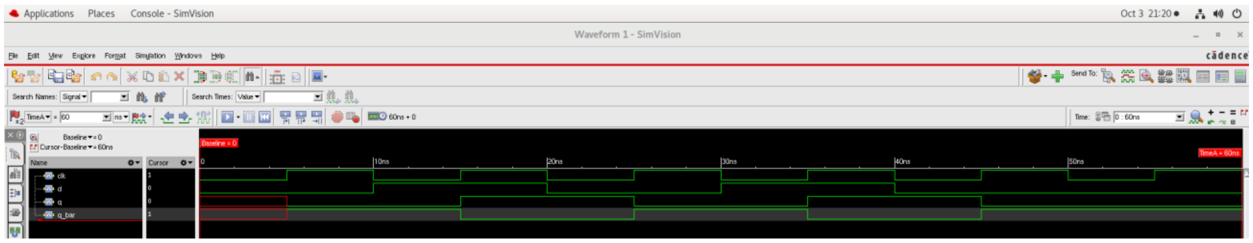
```
module tb_master_slave_jk_flip_flop;
    reg j;
    reg k;
    reg clk;
    wire q;

    master_slave_jk_flip_flop uut (
        .j(j),
        .k(k),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        j = 0; k = 0;
        #10 j = 1; k = 0;
        #10 j = 0; k = 1;
        #10 j = 1; k = 1;
        #10 $finish;
    end
endmodule
```

Experimental Results (Example for all flip-flops)

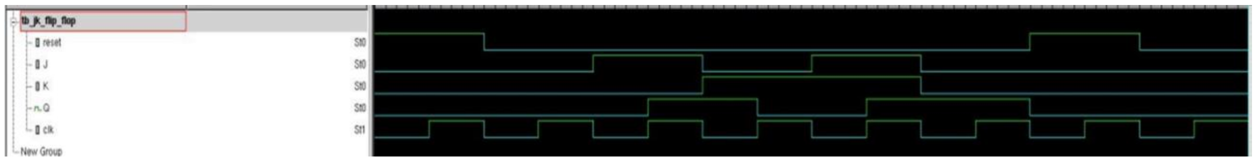
- **D Flip-Flop:** Capture waveform showing Q following D at the clock edge.
- **J-K Flip-Flop:** Capture waveform showing toggle behavior when J = K = 1.
- **SR Flip-Flop:** Show the valid set/reset operations, avoiding the invalid (S=R=1) condition.
- **T Flip-Flop:** Capture the toggling of Q for each clock cycle when T = 1.
- **Master-Slave Flip-Flop:** Demonstrate the proper two-stage operation where the master is triggered by the clock rising edge, and the slave by the falling edge.

(FUNCTIONAL VERIFICATION)- WAVEFORMS

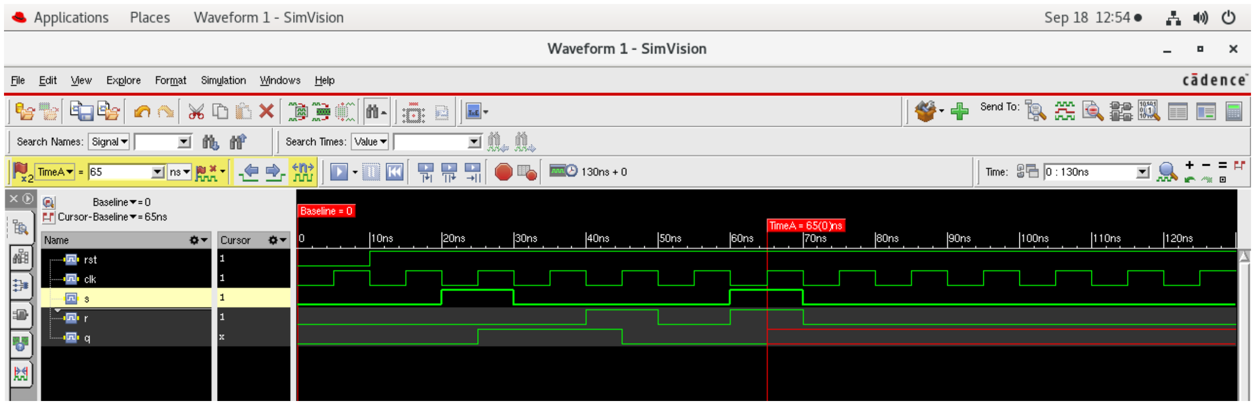
D FLIP- FLOP



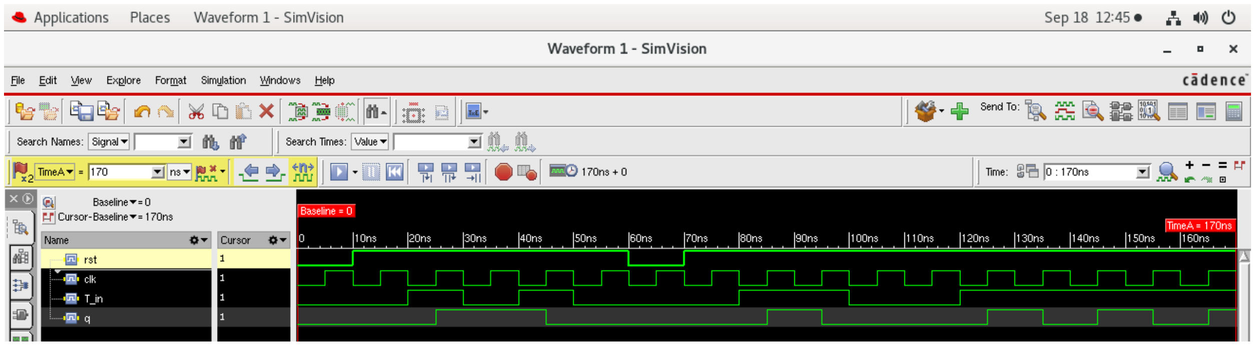
JK FLIP –FLOP



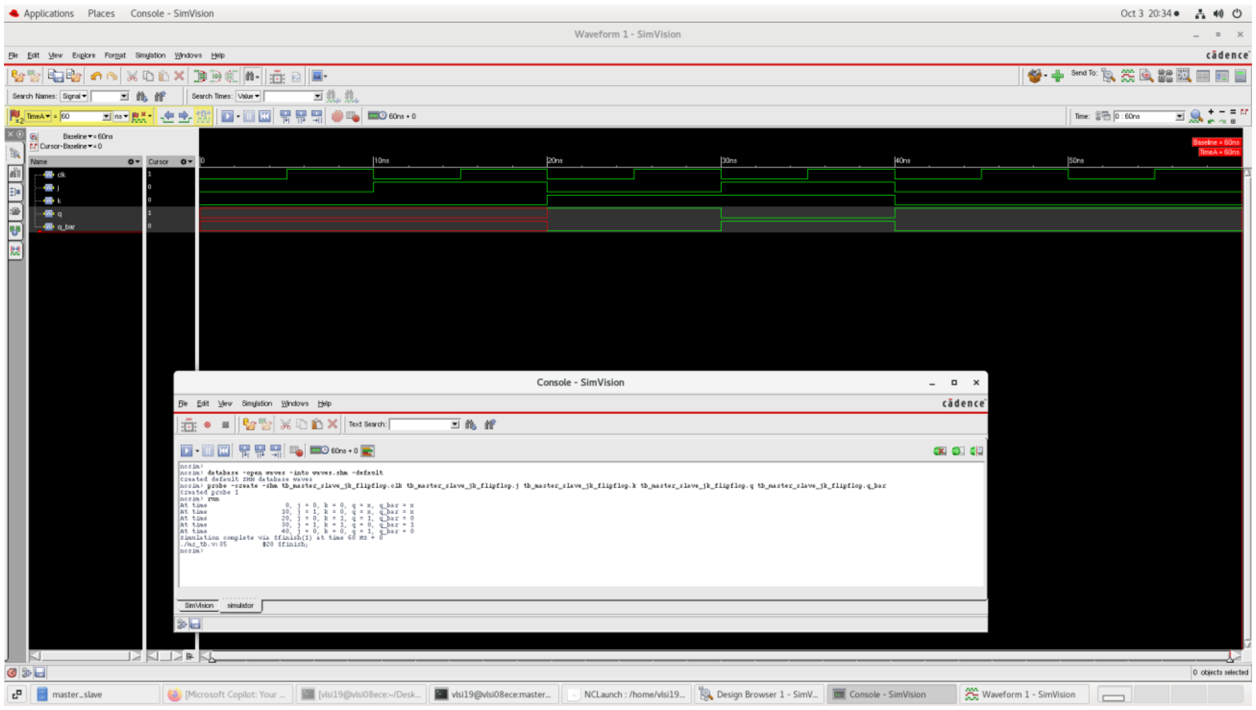
S-R FLIP-FLOP



T FF



MASTER SLAVE J K

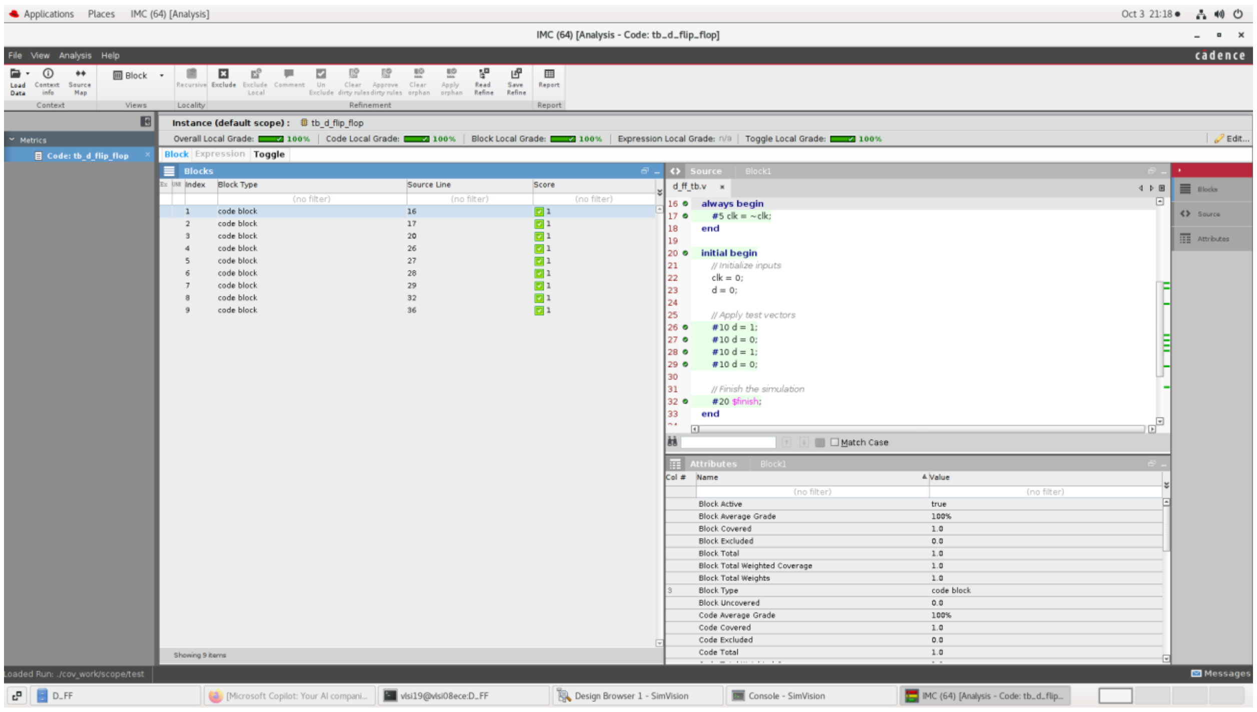


```
ncsim>
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
ncsim> probe -create -shm tb_master_slave_jk_flipflop.clk tb_master_slave_jk_flipflop.j tb_master_slave_jk_flipflop.k tb_master_slave_jk_flipflop.q tb_master_slave_jk_flipflop.q_bar
Created probe 1
ncsim> run
At time                    0, j = 0, k = 0, q = x, q_bar = x
At time                   10, j = 1, k = 0, q = x, q_bar = x
At time                   20, j = 0, k = 1, q = 1, q_bar = 0
At time                   30, j = 1, k = 1, q = 0, q_bar = 1
At time                   40, j = 0, k = 0, q = 1, q_bar = 0
Simulation complete via $finish(1) at time 60 NS + 0
./ms_tb.v:35         #20 $finish;
ncsim>
```

Sign-____________

CODE –COVERAGE

D FF



Instance (default scope) : tb_d_flip_flop

Overall Local Grade: 100% | Code Local Grade: 100% | Block Local Grade: 100% | Expression Local Grade: n/a | Toggle Local Grade: 100%

Index	Block Type	Source Line	Score
1	code block	16	1
2	code block	17	1
3	code block	20	1
4	code block	26	1
5	code block	27	1
6	code block	28	1
7	code block	29	1
8	code block	32	1
9	code block	36	1

Showing 9 items

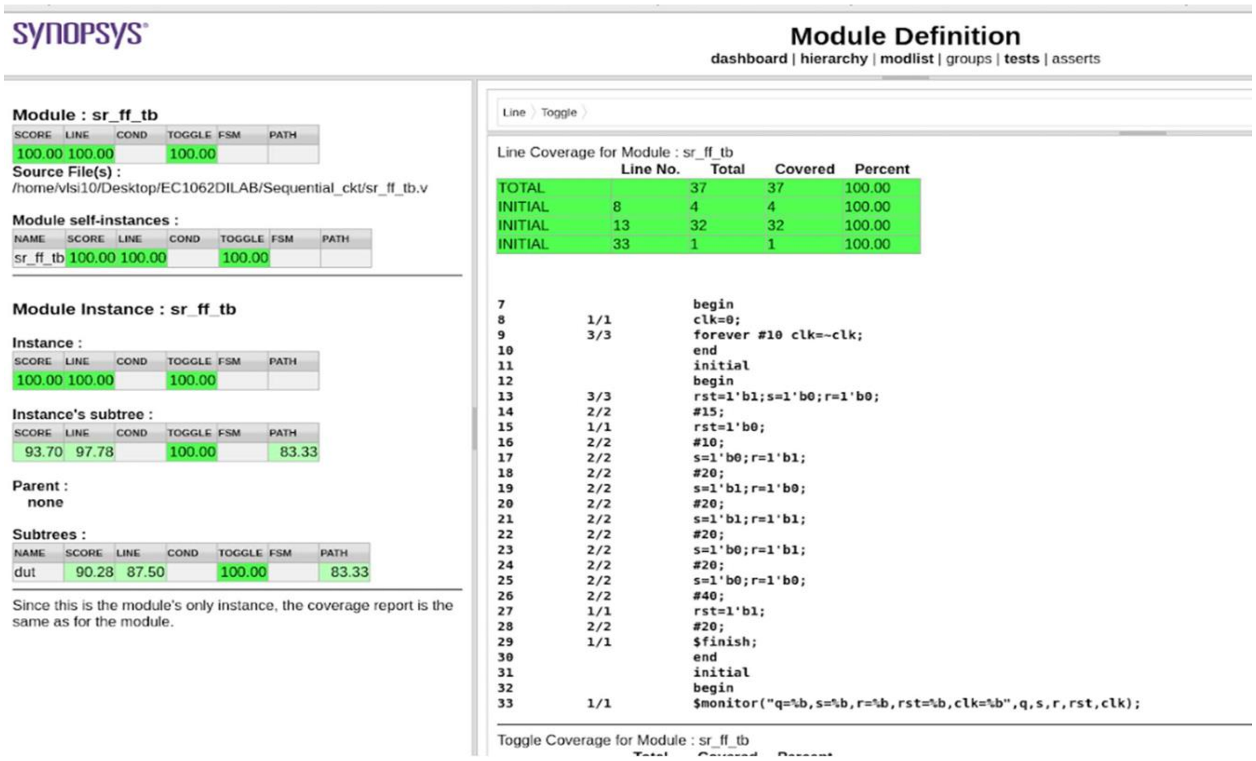
d_ff_tb.v

```
16    always begin
17        #5 clk = ~clk;
18    end
19
20    initial begin
21        // Initialize inputs
22        clk = 0;
23        d = 0;
24
25        // Apply test vectors
26        #10 d = 1;
27        #10 d = 0;
28        #10 d = 1;
29        #10 d = 0;
30
31        // Finish the simulation
32        #20 $finish;
33    end
```

Col #	Name	Value
	Block Active	true
	Block Average Grade	100%
	Block Covered	1.0
	Block Excluded	0.0
	Block Total	1.0
	Block Total Weighted Coverage	1.0
	Block Total Weights	1.0
3	Block Type	code block
	Block Uncovered	0.0
	Code Average Grade	100%
	Code Covered	1.0
	Code Excluded	0.0
	Code Total	1.0

Loaded Run: ./cov_work/scope/test

SR FF



Module Definition

dashboard | hierarchy | modlist | groups | tests | asserts

Module : sr_ff_tb

SCORE	LINE	COND	TOGGLE	FSM	PATH
100.00	100.00		100.00		

Source File(s) :
/home/vlsi10/Desktop/EC1062DILAB/Sequential_ckt/sr_ff_tb.v

Module self-instances :

NAME	SCORE	LINE	COND	TOGGLE	FSM	PATH
sr_ff_tb	100.00	100.00		100.00		

Module Instance : sr_ff_tb

Instance :

SCORE	LINE	COND	TOGGLE	FSM	PATH
100.00	100.00		100.00		

Instance's subtree :

SCORE	LINE	COND	TOGGLE	FSM	PATH
93.70	97.78		100.00		83.33

Parent :
none

Subtrees :

NAME	SCORE	LINE	COND	TOGGLE	FSM	PATH
dut	90.28	87.50		100.00		83.33

Since this is the module's only instance, the coverage report is the same as for the module.

Line 〉 Toggle 〉

Line Coverage for Module : sr_ff_tb

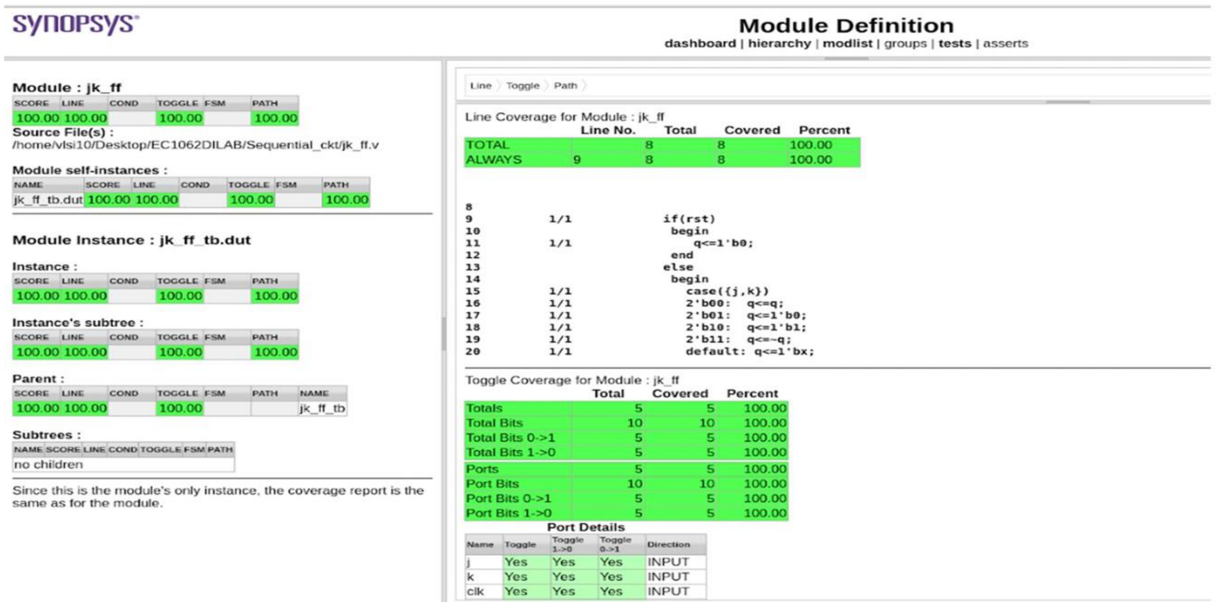
	Line No.	Total	Covered	Percent
TOTAL		37	37	100.00
INITIAL	8	4	4	100.00
INITIAL	13	32	32	100.00
INITIAL	33	1	1	100.00

```
7                    begin
8          1/1       clk=0;
9          3/3       forever #10 clk=~clk;
10                   end
11                   initial
12                   begin
13         3/3       rst=1'b1;s=1'b0;r=1'b0;
14         2/2       #15;
15         1/1       rst=1'b0;
16         2/2       #10;
17         2/2       s=1'b0;r=1'b1;
18         2/2       #20;
19         2/2       s=1'b1;r=1'b0;
20         2/2       #20;
21         2/2       s=1'b1;r=1'b1;
22         2/2       #20;
23         2/2       s=1'b0;r=1'b1;
24         2/2       #20;
25         2/2       s=1'b0;r=1'b0;
26         2/2       #40;
27         1/1       rst=1'b1;
28         2/2       #20;
29         1/1       $finish;
30                   end
31                   initial
32                   begin
33         1/1       $monitor("q=%b,s=%b,r=%b,rst=%b,clk=%b",q,s,r,rst,clk);
```

Toggle Coverage for Module : sr_ff_tb

Sign-____________

JK FF



Module Definition
dashboard | hierarchy | modlist | groups | tests | asserts

Module : jk_ff

SCORE	LINE	COND	TOGGLE	FSM	PATH
100.00	100.00		100.00		100.00

Source File(s) :
/home/vlsi10/Desktop/EC1062DILAB/Sequential_ckt/jk_ff.v

Module self-instances :

NAME	SCORE	LINE	COND	TOGGLE	FSM	PATH
jk_ff_tb.dut	100.00	100.00		100.00		100.00

Module Instance : jk_ff_tb.dut

Instance :

SCORE	LINE	COND	TOGGLE	FSM	PATH
100.00	100.00		100.00		100.00

Instance's subtree :

SCORE	LINE	COND	TOGGLE	FSM	PATH
100.00	100.00		100.00		100.00

Parent :

SCORE	LINE	COND	TOGGLE	FSM	PATH	NAME
100.00	100.00		100.00			jk_ff_tb

Subtrees :

NAME	SCORE	LINE	COND	TOGGLE	FSM	PATH
no children						

Since this is the module's only instance, the coverage report is the same as for the module.

Line Coverage for Module : jk_ff

	Line No.	Total	Covered	Percent
TOTAL		8	8	100.00
ALWAYS	9	8	8	100.00

```
8
9           1/1             if(rst)
10                           begin
11          1/1                 q<=1'b0;
12                           end
13                          else
14                           begin
15          1/1                case({j,k})
16          1/1                 2'b00:  q<=q;
17          1/1                 2'b01:  q<=1'b0;
18          1/1                 2'b10:  q<=1'b1;
19          1/1                 2'b11:  q<=~q;
20          1/1                 default: q<=1'bx;
```

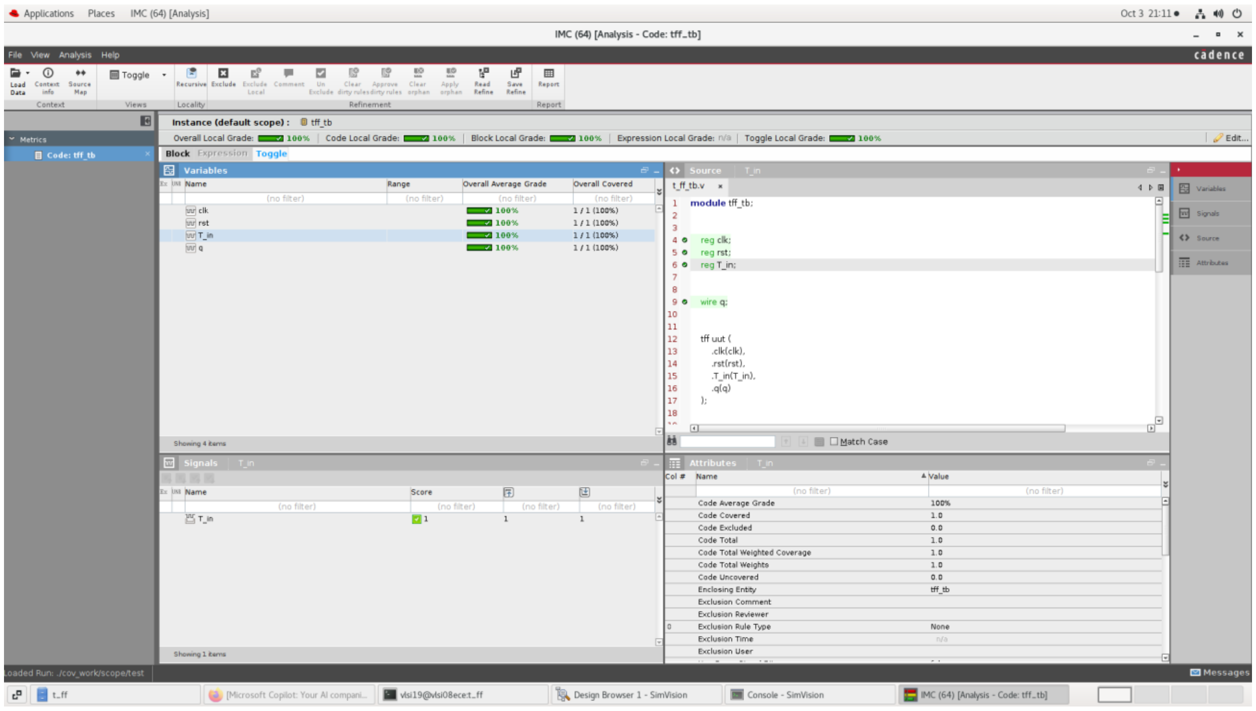
Toggle Coverage for Module : jk_ff

	Total	Covered	Percent
Totals	5	5	100.00
Total Bits	10	10	100.00
Total Bits 0->1	5	5	100.00
Total Bits 1->0	5	5	100.00
Ports	5	5	100.00
Port Bits	10	10	100.00
Port Bits 0->1	5	5	100.00
Port Bits 1->0	5	5	100.00

Port Details

Name	Toggle	Toggle 1->0	Toggle 0->1	Direction
j	Yes	Yes	Yes	INPUT
k	Yes	Yes	Yes	INPUT
clk	Yes	Yes	Yes	INPUT

T FF



Instance (default scope) : tff_tb

Overall Local Grade: 100% | Code Local Grade: 100% | Block Local Grade: 100% | Expression Local Grade: n/a | Toggle Local Grade: 100%

Ex	UNR	Name	Range	Overall Average Grade	Overall Covered
		(no filter)	(no filter)	(no filter)	(no filter)
		clk		100%	1 / 1 (100%)
		rst		100%	1 / 1 (100%)
		T_in		100%	1 / 1 (100%)
		q		100%	1 / 1 (100%)

t_ff_tb.v

```
module tff_tb;


 reg clk;
 reg rst;
 reg T_in;


 wire q;



 tff uut (
   .clk(clk),
   .rst(rst),
   .T_in(T_in),
   .q(q)
 );

```

Ex	UNR	Name	Score			
		(no filter)	(no filter)	(no filter)	(no filter)	
		T_in	1	1	1	

Col #	Name	Value
	(no filter)	(no filter)
	Code Average Grade	100%
	Code Covered	1.0
	Code Excluded	0.0
	Code Total	1.0
	Code Total Weighted Coverage	1.0
	Code Total Weights	1.0
	Code Uncovered	0.0
	Enclosing Entity	tff_tb
	Exclusion Comment	
	Exclusion Reviewer	
0	Exclusion Rule Type	None
	Exclusion Time	n/a
	Exclusion User	

Sign-____________

Observation:

1.a) Edge Triggered D Flip Flop

Simulation Result:

The simulation waveform accurately reflects the expected behavior of the D Flip-Flop for all scenarios covered in the test bench.

Test Coverage Report:

Test Coverage Report generated Successfully and showing 100% Result.

1.b) Edge Triggered JK Flip Flop

Simulation Result:

The simulation waveform accurately reflects the expected behavior of the JK Flip-Flop for all scenarios covered in the test bench.

Test Coverage Report:

Test Coverage Report generated Successfully and showing 100% Result.

1.c) Edge Triggered SR Flip Flop

Simulation Result:

The simulation waveform accurately reflects the expected behavior of the SR Flip-Flop for all scenarios covered in the test bench.

Test Coverage Report:

Test Coverage Report generated Successfully and showing 93.70% Result.

Not able to cover the Unknown State of SR Flip Flop when both input 1 and 1.

1.d) Edge Triggered T Flip Flop

Simulation Result:

The simulation waveform accurately reflects the expected behavior of the T Flip-Flop

for all scenarios covered in the test bench.

Test Coverage Report:

Test Coverage Report generated Successfully and showing 100% Result.

1.e) Master Slave Flip Flop

Simulation Result:

The simulation waveform accurately reflects the expected behavior of the Master Slave FlipFlop for all scenarios covered in the test bench.

Test Coverage Report:

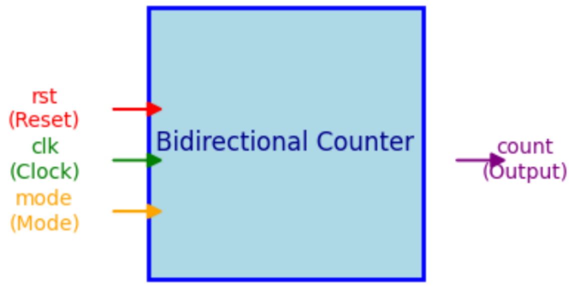
Test Coverage Report generated Successfully and showing 100% Result.

Design No.:2 Bidirectional Counter

Objective:

The objective of this design is to implement and verify a 4-bit bidirectional counter that counts both up and down based on the mode input signal. The design will be coded in Verilog and simulated using the Cadence NC Launch tool.

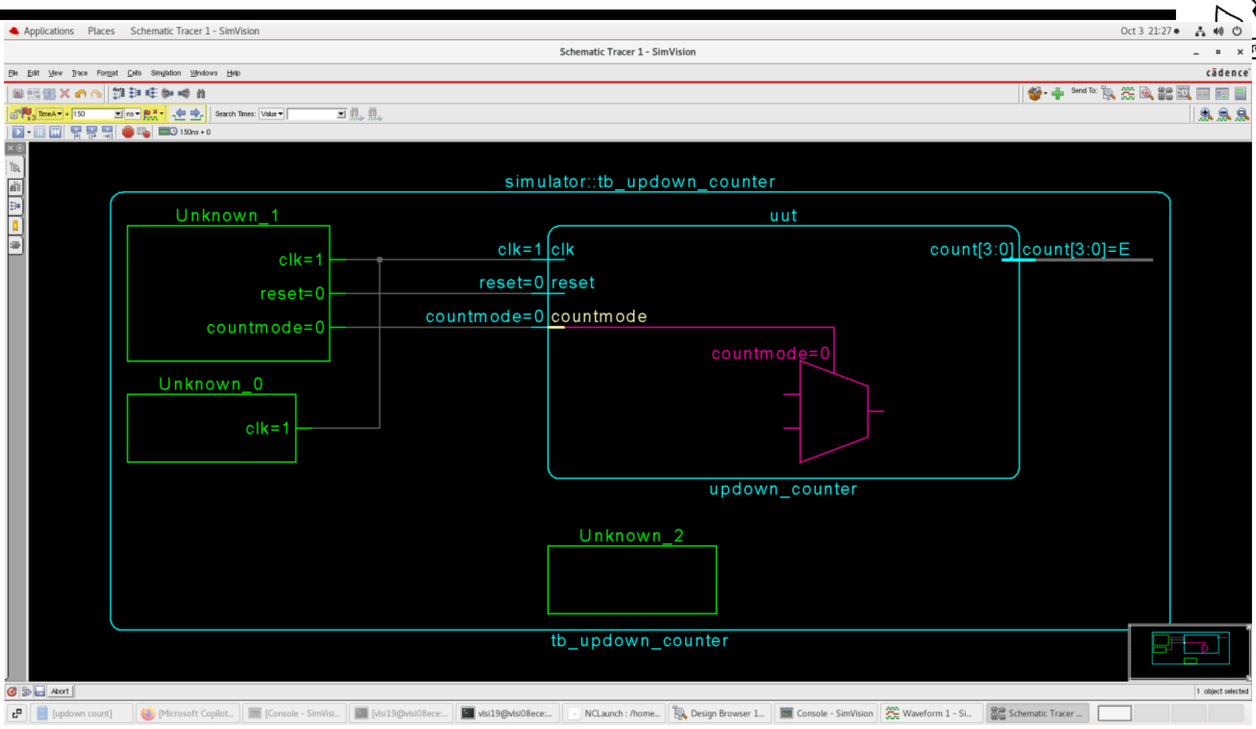
Specification:



- Counter Width: 4-bit
- Mode Input: A single-bit control signal (mode) that determines the direction of the count. When mode = 1, the counter counts up; when mode = 0, the counter counts down.
- Clock Input: Positive edge-triggered clock signal (clk).
- Reset Input: An asynchronous active-high reset (reset) that sets the counter to zero.
- Output: A 4-bit counter output (count) that shows the current count value.

Logic/ Block-level implementation:

The bidirectional counter logic includes:
- Control Unit: Determines the counting direction based on the mode signal.
- 4-bit Register: Stores the current count value.
- Adder/Subtractor Logic: Adds or subtracts based on the mode signal.
- Reset Logic: Sets the counter to zero when the reset signal is high.



RTL Coding:

```
module updown_counter (
    input clk,
    input reset,
    input countmode, // 0 for down, 1 for up
    output reg [3:0] count
);

always @(posedge clk or posedge reset) begin
    if (reset) begin
        count <= 4'b0000;
    end else begin
        if (countmode) begin
            count <= count + 1; // Up counter
        end else begin
            count <= count - 1; // Down counter
        end
    end
end

endmodule
```

Simulation Setup:

1. Tool Used: Cadence NC Launch
2. Simulation Environment: The testbench will apply the necessary clock, reset, and mode signals to observe the counter's behavior.

Sign-____________

Testbench Code:

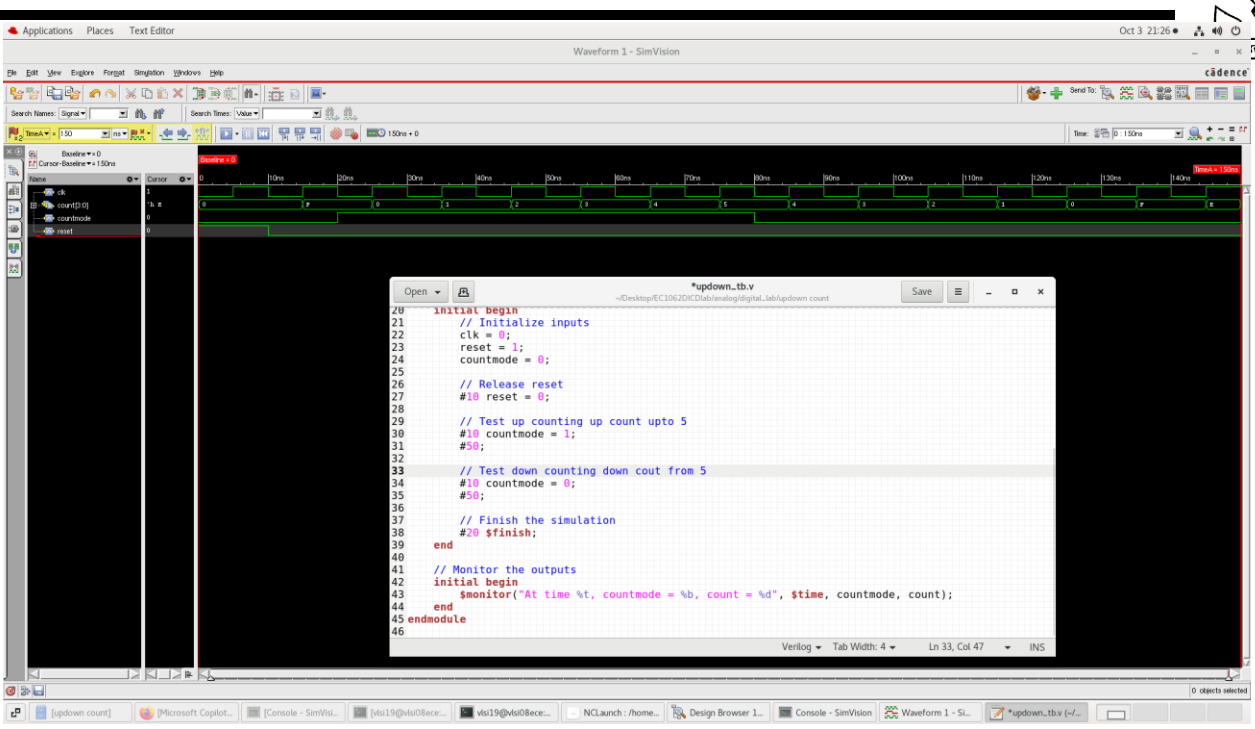
Test bench will be used to check functional verification of bidirectional counter.and code coverage will be checked by invoking IMC tool.

```verilog
module tb_updown_counter;
    reg clk;
    reg reset;
    reg countmode;
    wire [3:0] count;

    // Instantiate the counter
    updown_counter uut (
        .clk(clk),
        .reset(reset),
        .countmode(countmode),
        .count(count)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;
    end

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;
        countmode = 0;

        // Release reset
        #10 reset = 0;

        // Test up counting up count upto 5
        #10 countmode = 1;
        #50;

        // Test down counting down cout from 5
        #10 countmode = 0;
        #50;

        // Finish the simulation
        #20 $finish;
    end

    // Monitor the outputs
    initial begin
        $monitor("At time %t, countmode = %b, count = %d", $time, countmode, count);
    end
endmodule
```

Experimental Results:

The simulation waveforms show the correct operation of the bidirectional counter:
- The counter starts at 0 upon reset.
- It counts down when mode = 0.
- It counts up when mode = 1.
- After the reset signal is applied again, the counter resets to 0 and resumes counting based on the mode signal.
- The simulation confirms that the counter works as expected for both counting directions.

Sign-____________



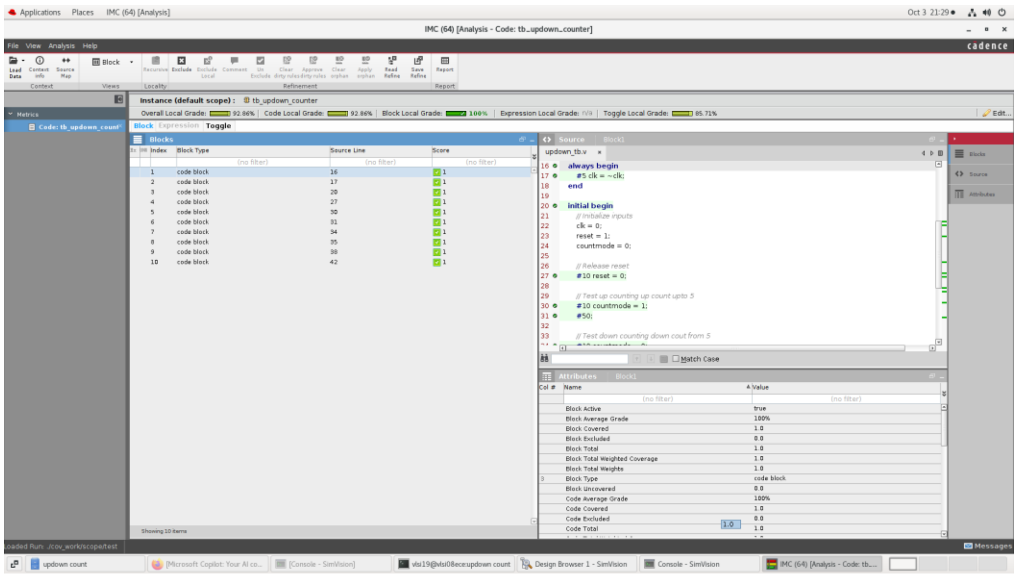
```verilog
    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;
        countmode = 0;

        // Release reset
        #10 reset = 0;

        // Test up counting up count upto 5
        #10 countmode = 1;
        #50;

        // Test down counting down cout from 5
        #10 countmode = 0;
        #50;

        // Finish the simulation
        #20 $finish;
    end

    // Monitor the outputs
    initial begin
        $monitor("At time %t, countmode = %b, count = %d", $time, countmode, count);
    end
endmodule
```

Observations:

- The bidirectional counter successfully switches between counting up and counting down based on the mode signal.
- The asynchronous reset properly initializes the counter to 0 when asserted.
- The design is functionally verified through simulation, and the expected behavior is observed across different test cases.

CODE COVERAGE-

CODE coverage has been checked using IMC TOOL IN CADENCE



Instance (default scope) : tb_updown_counter

Overall Local Grade: 92.86% | Code Local Grade: 92.86% | Block Local Grade: 100% | Expression Local Grade: n/a | Toggle Local Grade: 85.71%

Index	Block Type	Source Line	Score
1	code block	16	✓ 1
2	code block	17	✓ 1
3	code block	20	✓ 1
4	code block	27	✓ 1
5	code block	30	✓ 1
6	code block	31	✓ 1
7	code block	34	✓ 1
8	code block	35	✓ 1
9	code block	38	✓ 1
10	code block	42	✓ 1

updown_tb.v

```verilog
16    always begin
17        #5 clk = ~clk;
18    end
19
20    initial begin
21        // Initialize inputs
22        clk = 0;
23        reset = 1;
24        countmode = 0;
25
26        // Release reset
27        #10 reset = 0;
28
29        // Test up counting up count upto 5
30        #10 countmode = 1;
31        #50;
32
33        // Test down counting down cout from 5
```

Col #	Name	Value
	Block Active	true
	Block Average Grade	100%
	Block Covered	1.0
	Block Excluded	0.0
	Block Total	1.0
	Block Total Weighted Coverage	1.0
	Block Total Weights	1.0
3	Block Type	code block
	Block Uncovered	0.0
	Code Average Grade	100%
	Code Covered	1.0
	Code Excluded	0.0
	Code Total	1.0

Sign-____________

ANKIT KUMAR 24EC4224

Design No.: 3

Objective:

To design and implement a shift register using sequential circuits. A shift register is a sequential logic circuit that can shift data either to the right or left based on control inputs. It is used for data storage, transfer, and conversion of data from serial to parallel or vice versa.

Specification:

1. Type: 4-bit Shift Register (Serial-In Serial-Out)
2. Control: Includes options for shifting left, shifting right, or holding the current state.
3. Inputs:
   - Clock (CLK)
   - Serial Data Input
   - Reset (RST)
   - Shift Direction Control (LEFT/RIGHT)- shift mode
4. Outputs:
   - Serial Data Output
5. Flip-Flop: Edge-triggered D Flip-Flops are used for each bit of the register.

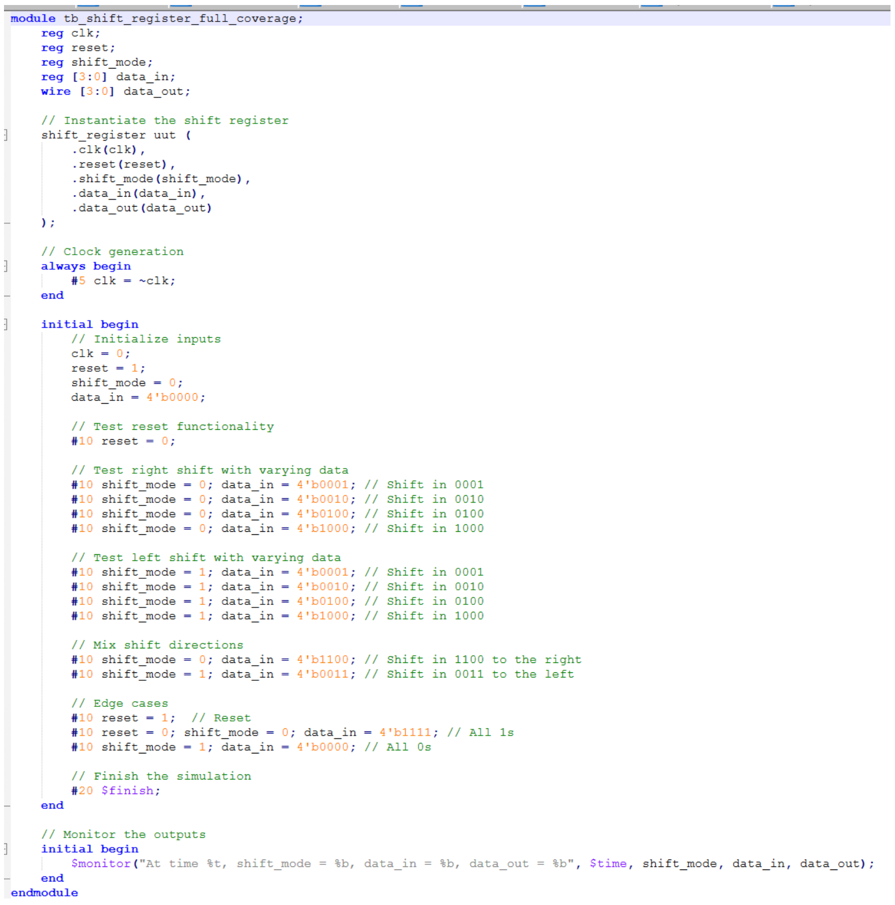
Logic/Block-level implementation:

RTL Coding:

The Verilog code for a 4-bit shift register is provided below. It includes a control signal to choose between shifting left or right and a reset signal to clear the register.

```
module shift_register (
    input clk,
    input reset,
    input shift_mode, // 0 for right shift, 1 for left shift
    input [3:0] data_in,
    output reg [3:0] data_out
);

always @(posedge clk or posedge reset) begin
    if (reset) begin
        data_out <= 4'b0000;
    end else begin
        if (shift_mode) begin
            data_out <= {data_out[2:0], data_in[0]}; // Left shift
        end else begin
            data_out <= {data_in[3], data_out[3:1]}; // Right shift
        end
    end
end

endmodule
```

TESTBENCH



```verilog
module tb_shift_register_full_coverage;
    reg clk;
    reg reset;
    reg shift_mode;
    reg [3:0] data_in;
    wire [3:0] data_out;

    // Instantiate the shift register
    shift_register uut (
        .clk(clk),
        .reset(reset),
        .shift_mode(shift_mode),
        .data_in(data_in),
        .data_out(data_out)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;
    end

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;
        shift_mode = 0;
        data_in = 4'b0000;

        // Test reset functionality
        #10 reset = 0;

        // Test right shift with varying data
        #10 shift_mode = 0; data_in = 4'b0001; // Shift in 0001
        #10 shift_mode = 0; data_in = 4'b0010; // Shift in 0010
        #10 shift_mode = 0; data_in = 4'b0100; // Shift in 0100
        #10 shift_mode = 0; data_in = 4'b1000; // Shift in 1000

        // Test left shift with varying data
        #10 shift_mode = 1; data_in = 4'b0001; // Shift in 0001
        #10 shift_mode = 1; data_in = 4'b0010; // Shift in 0010
        #10 shift_mode = 1; data_in = 4'b0100; // Shift in 0100
        #10 shift_mode = 1; data_in = 4'b1000; // Shift in 1000

        // Mix shift directions
        #10 shift_mode = 0; data_in = 4'b1100; // Shift in 1100 to the right
        #10 shift_mode = 1; data_in = 4'b0011; // Shift in 0011 to the left

        // Edge cases
        #10 reset = 1;  // Reset
        #10 reset = 0; shift_mode = 0; data_in = 4'b1111; // All 1s
        #10 shift_mode = 1; data_in = 4'b0000; // All 0s

        // Finish the simulation
        #20 $finish;
    end

    // Monitor the outputs
    initial begin
        $monitor("At time %t, shift_mode = %b, data_in = %b, data_out = %b", $time, shift_mode, data_in, data_out);
    end
endmodule
```

Simulation Setup:

1. Tool:
   - Cadence NCLAUNCH window was used for the functional verification of the shift register.
   - IMC tool was used for code coverage analysis to ensure thorough testing of the design.

2. Testbench: A testbench was developed to provide clock, reset, and input data to the shift register, including serial data inputs, as well as the control signal for direction.

3. Simulation Process:
   - Functional verification was performed to check the behavior of the shift register for all input cases.
   - The IMC tool provided a code coverage report to confirm that all possible scenarios and branches of the RTL code were tested.

4. Waveform:
   - The waveform was generated and observed in the Cadence simulation environment to ensure

that data shifted correctly through the register based on the control signals and that the reset functionality worked as expected.

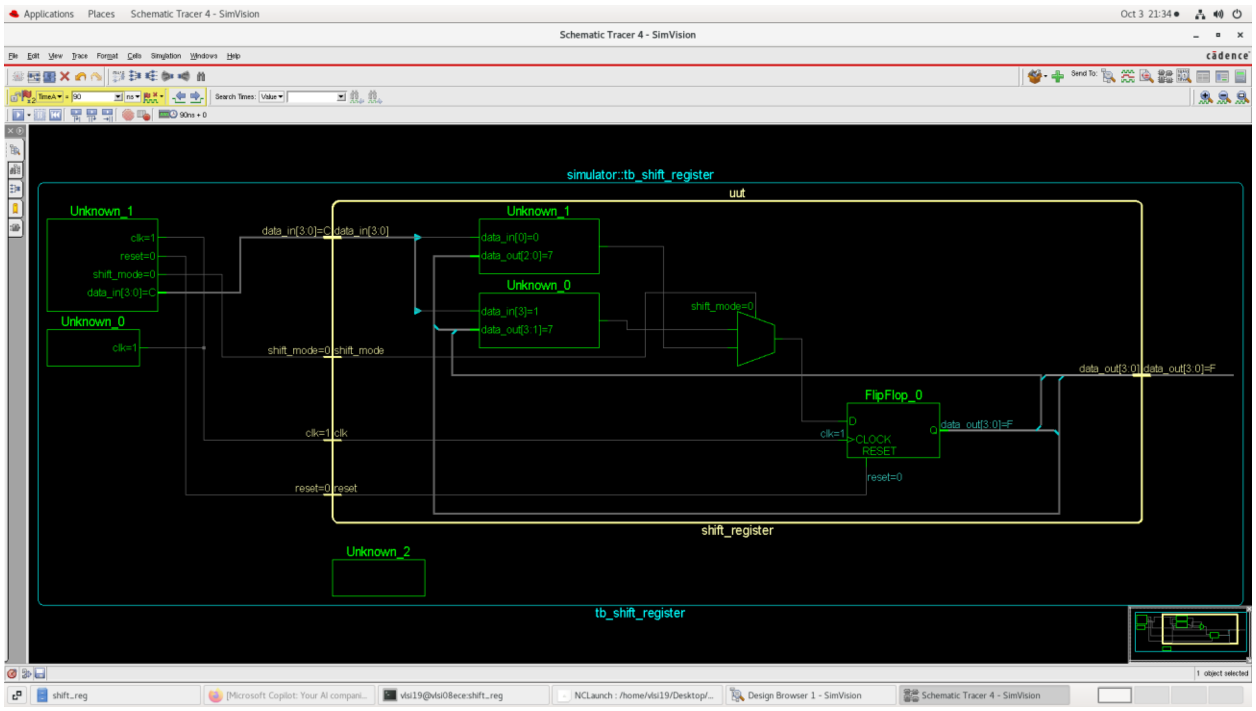
Experimental Results:

The simulation of the shift register produced the following results:

1. Reset: When reset is asserted, the output of the register becomes 0.
2. Shifting Left: With the direction control set to shift left (shift_left = 1), the serial input is shifted into the register from the right side, and data moves left on each clock cycle.
3. Shifting Right: With the direction control set to shift right (shift_left = 0), the serial input is shifted into the register from the left side, and data moves right on each clock cycle.
4. Hold Condition: When no shift control is asserted, the register maintains its current state.
5. Code Coverage: The IMC tool reported 100% coverage, confirming that all states and transitions in the design were verified.

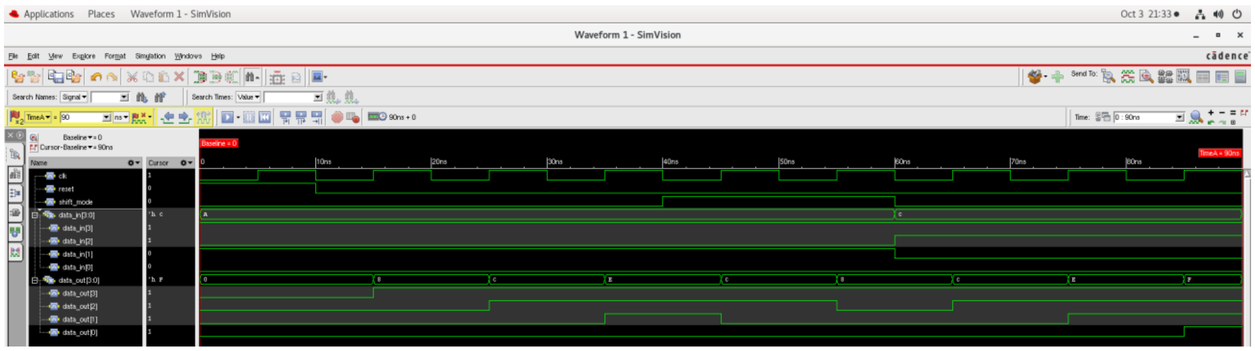
Functional verification-

Logic-diagram

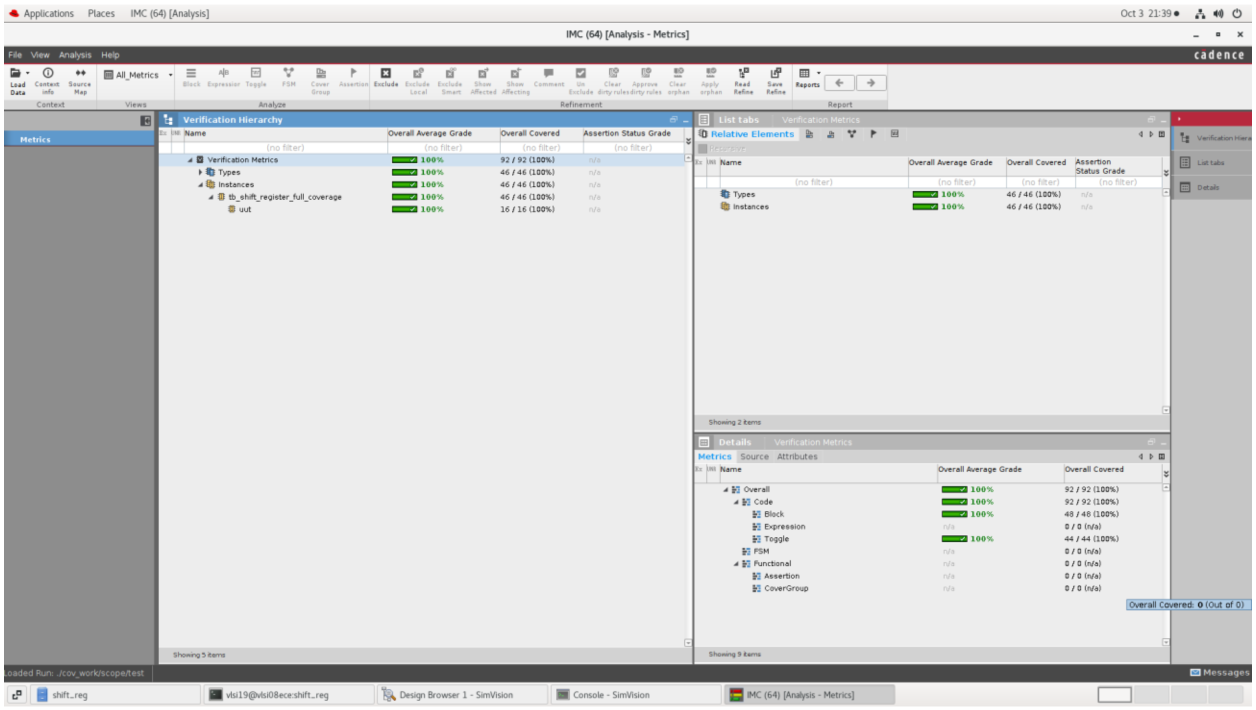


Sign-____________
ANKIT KUMAR 24EC4224

Waveform-



Code coverage-



IMC (64) [Analysis - Metrics]

Verification Hierarchy

Name	Overall Average Grade	Overall Covered	Assertion Status Grade
Verification Metrics	100%	92 / 92 (100%)	n/a
Types	100%	46 / 46 (100%)	n/a
Instances	100%	46 / 46 (100%)	n/a
tb_shift_register_full_coverage	100%	46 / 46 (100%)	n/a
uut	100%	16 / 16 (100%)	n/a

List tabs

Name	Overall Average Grade	Overall Covered	Assertion Status Grade
Types	100%	46 / 46 (100%)	n/a
Instances	100%	46 / 46 (100%)	n/a

Details — Metrics

Name	Overall Average Grade	Overall Covered
Overall	100%	92 / 92 (100%)
Code	100%	92 / 92 (100%)
Block	100%	48 / 48 (100%)
Expression	n/a	0 / 0 (n/a)
Toggle	100%	44 / 44 (100%)
FSM	n/a	0 / 0 (n/a)
Functional	n/a	0 / 0 (n/a)
Assertion	n/a	0 / 0 (n/a)
CoverGroup	n/a	0 / 0 (n/a)

Overall Covered: 0 (Out of 0)

Loaded Run: ./cov_work/scope/test

Observations:

1. The shift register correctly shifts data left or right based on the control signals.
2. The reset functionality works as expected, clearing the register on assertion of the reset signal.
3. Functional verification in Cadence NCLAUNCH ensured the shift register's correct operation under all input cases.
4. The IMC tool's coverage report showed that all lines of code and branches were tested, ensuring the design's robustness.

Sign-_____________

Design No.: 4
Objective:
To design and implement a Moore machine that outputs based solely on the current state, using Cadence EDA tools for functional verification and code coverage analysis.

Specification:

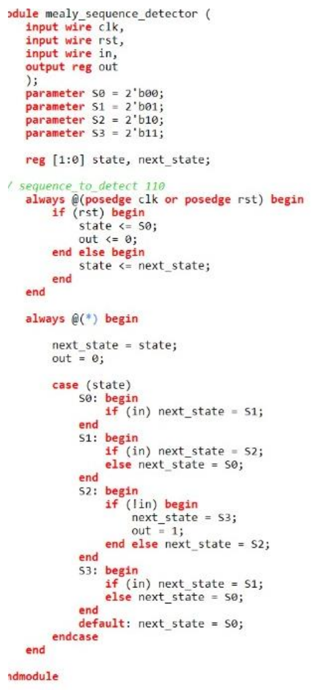
- The Moore machine operates with a set of defined states.
- The output depends only on the current state.
- The state transitions occur on the rising edge of the clock.
- The machine includes an initial state (reset) and transitions based on the input signals.

Logic/ Block-level implementation:

- The Moore machine consists of:
  - A state register to hold the current state.
  - A combinational logic block to determine the next state based on the input.
  - A combinational logic block to determine the output based on the current state.

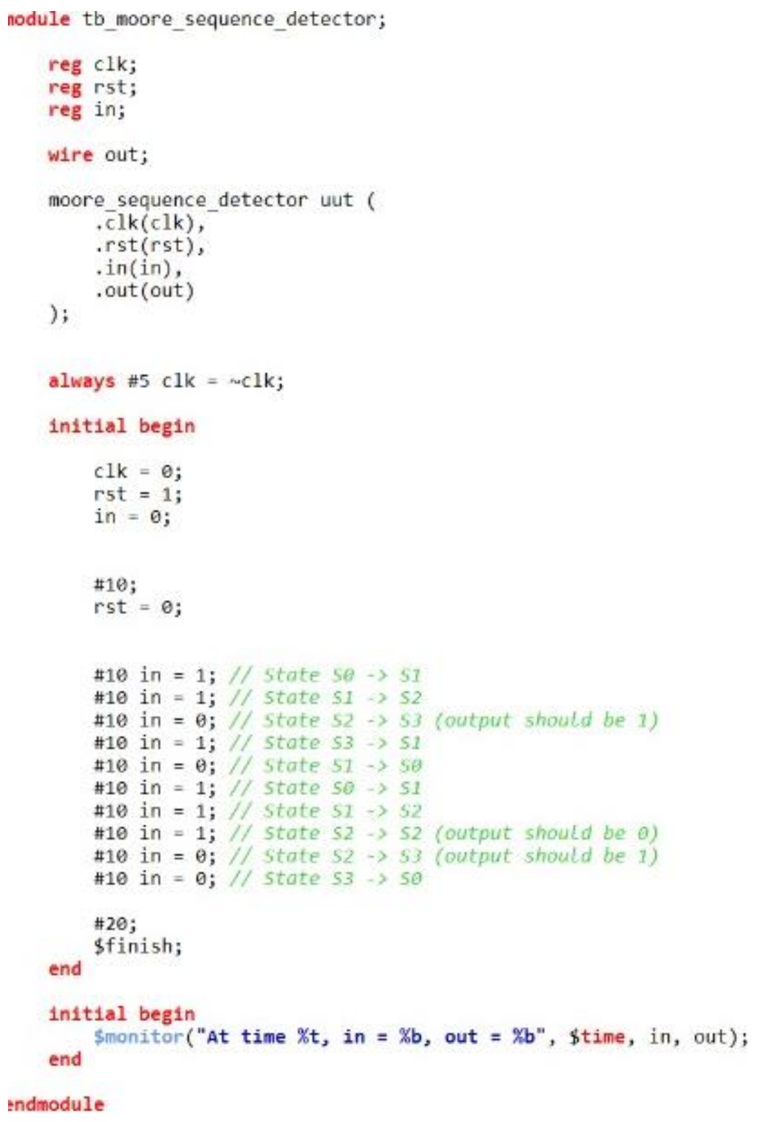
RTL Coding:

Below is an example of an RTL code for a Moore Sequence Detector that detects a “110” sequence in the input stream.\



```
odule mealy_sequence_detector (
   input wire clk,
   input wire rst,
   input wire in,
   output reg out
   );
   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
   parameter S3 = 2'b11;

   reg [1:0] state, next_state;

/ sequence_to_detect 110
   always @(posedge clk or posedge rst) begin
       if (rst) begin
           state <= S0;
           out <= 0;
       end else begin
           state <= next_state;
       end
   end

   always @(*) begin

       next_state = state;
       out = 0;

       case (state)
           S0: begin
               if (in) next_state = S1;
           end
           S1: begin
               if (in) next_state = S2;
               else next_state = S0;
           end
           S2: begin
               if (!in) begin
                   next_state = S3;
                   out = 1;
               end else next_state = S2;
           end
           S3: begin
               if (in) next_state = S1;
               else next_state = S0;
           end
           default: next_state = S0;
       endcase
   end

ndmodule
```

Test bench-



```
module tb_moore_sequence_detector;

   reg clk;
   reg rst;
   reg in;

   wire out;

   moore_sequence_detector uut (
       .clk(clk),
       .rst(rst),
       .in(in),
       .out(out)
   );


   always #5 clk = ~clk;

   initial begin

       clk = 0;
       rst = 1;
       in = 0;


       #10;
       rst = 0;


       #10 in = 1; // State S0 -> S1
       #10 in = 1; // State S1 -> S2
       #10 in = 0; // State S2 -> S3 (output should be 1)
       #10 in = 1; // State S3 -> S1
       #10 in = 0; // State S1 -> S0
       #10 in = 1; // State S0 -> S1
       #10 in = 1; // State S1 -> S2
       #10 in = 1; // State S2 -> S2 (output should be 0)
       #10 in = 0; // State S2 -> S3 (output should be 1)
       #10 in = 0; // State S3 -> S0

       #20;
       $finish;
   end

   initial begin
       $monitor("At time %t, in = %b, out = %b", $time, in, out);
   end

endmodule
```

Simulation Setup:

- Cadence NCLAUNCH was used for functional verification.
- IMC (Integrated Metric Center) was used to perform code coverage analysis, ensuring all possible states and transitions are tested during simulation.

Simulation Parameters:

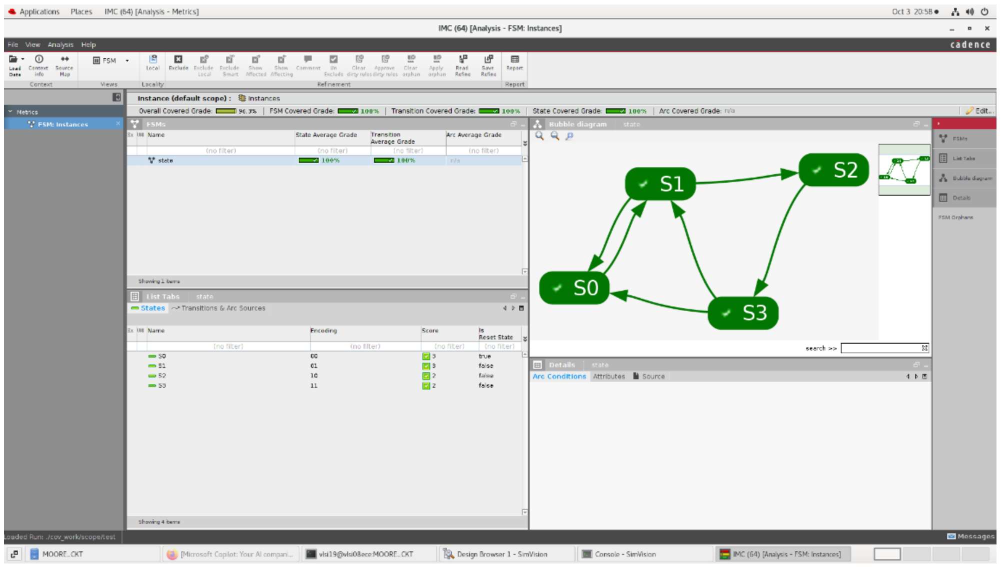
- Inputs: Clock, reset, input signal (`in`).
- Outputs: Output signal (`out`) based on the current state.
- The testbench initializes with reset and provides various input signal sequences to verify the correct state transitions and outputs.

Sign-____________

Experimental Results:

- The Moore machine was verified to transition through the correct states based on the input.
- The output matched the expected results, as the output was determined solely by the current state.

Functional verification-


IMC (64) [Analysis - FSM: Instances]

Instance (default scope) : Instances

Overall Covered Grade: 96.3% | FSM Covered Grade: 100% | Transition Covered Grade: 100% | State Covered Grade: 100% | Arc Covered Grade: n/a

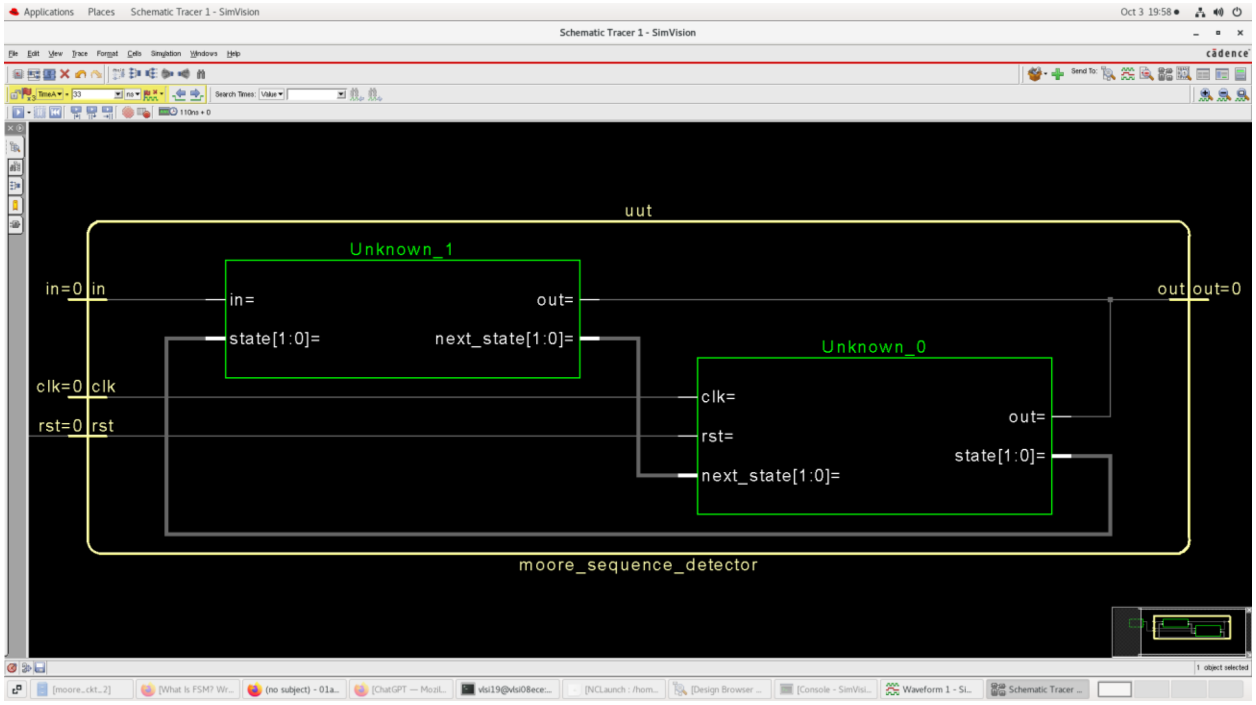
FSMs

Name	State Average Grade	Transition Average Grade	Arc Average Grade
state	100%	100%	n/a

List Tabs — state — States

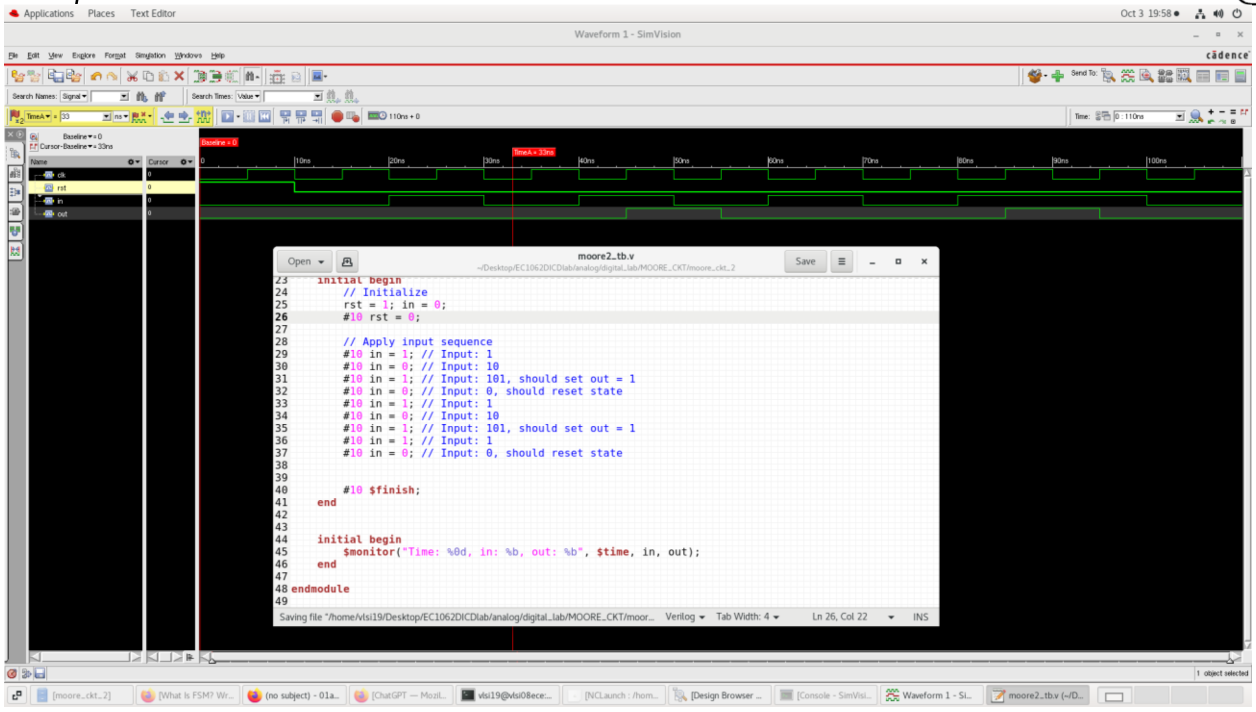
Name	Encoding	Score	Is Reset State
S0	00	✓ 3	true
S1	01	✓ 3	false
S2	10	✓ 2	false
S3	11	✓ 2	false

Logic Diagram



Sign-____________
ANKIT KUMAR 24EC4224

Wavefrom-


moore2_tb.v

```
    initial begin
        // Initialize
        rst = 1; in = 0;
        #10 rst = 0;

        // Apply input sequence
        #10 in = 1; // Input: 1
        #10 in = 0; // Input: 10
        #10 in = 1; // Input: 101, should set out = 1
        #10 in = 0; // Input: 0, should reset state
        #10 in = 1; // Input: 1
        #10 in = 0; // Input: 10
        #10 in = 1; // Input: 101, should set out = 1
        #10 in = 1; // Input: 1
        #10 in = 0; // Input: 0, should reset state


        #10 $finish;
    end


    initial begin
        $monitor("Time: %0d, in: %b, out: %b", $time, in, out);
    end

endmodule
```

Saving file "/home/vlsi19/Desktop/EC1062DICDlab/analog/digital_lab/MOORE_CKT/moor... Verilog Tab Width: 4 Ln 26, Col 22 INS

Observations:

- The Moore machine responded accurately to the given inputs, transitioning between states as expected.
- Using NCLAUNCH, the functional verification was successful, and no design errors were observed during the simulation.
- The IMC tool showed full code coverage, indicating that all possible state transitions were exercised in the testbench.

Design No.: 5
Objective:
To design and implement a Mealy machine that detects the sequence "110" in the input stream, using Cadence EDA tools for functional verification and code coverage analysis.

Specification:

- The Mealy machine is a type of finite state machine where the output depends on both the current state and the input.
- The machine detects the binary sequence "110". When this sequence is detected, the output (`out`) is set high (`1`).
- The state transitions occur on the rising edge of the clock (`clk`).
- The machine has four states:
  - **S0** (initial state): No input detected.
  - **S1**: The first "1" in the sequence is detected.
  - **S2**: The second "1" in the sequence is detected.
  - **S3**: The sequence "110" has been detected, and the output is set high.
- Unlike the Moore machine, in a Mealy machine, the output can change based on the current input and the current state.
- The machine resets to the initial state (S0) when the reset signal (`rst`) is high.
- Each state is encoded using 2 bits.
- The Mealy machine is faster than a Moore machine because the output can change immediately upon receiving an input, without waiting for the state transition.

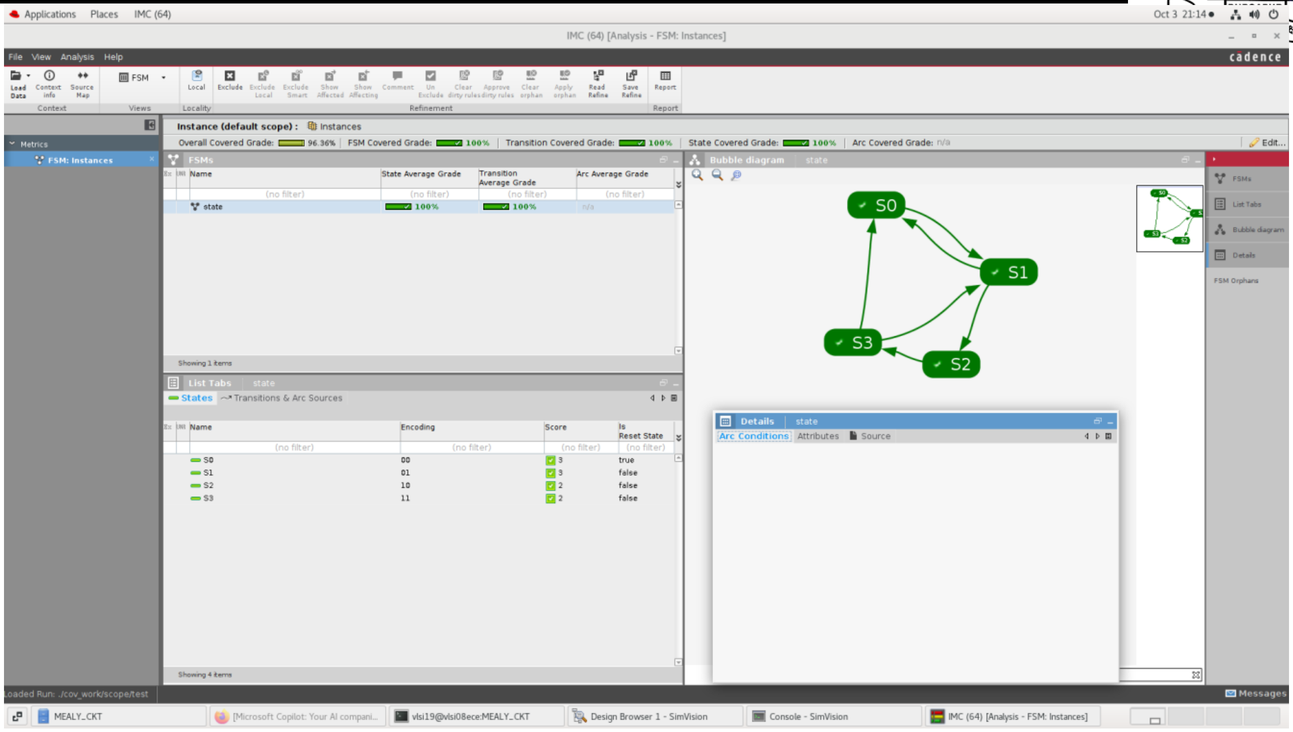
Logic/ Block-level implementation:

- The Mealy machine consists of:
  - **State Register**: Holds the current state of the machine.
  - **Next State Logic**: Determines the next state based on the current state and the input signal.
  - **Output Logic**: Generates the output signal based on both the current state and the input signal.

State Transition Diagram:
The following state transitions occur:

- **S0 → S1**: When `in = 1`.
- **S1 → S2**: When `in = 1`.
- **S2 → S0**: When `in = 0` (the sequence "110" is detected, and the output is set high immediately due to Mealy machine logic).
- **S2 → S2**: When `in = 1` (in case of repeated "1"s).
- **S0 → S0**: When `in = 0` (no valid sequence detected).

Sign-____________



Instance (default scope) : Instances

Overall Covered Grade: 96.36% | FSM Covered Grade: 100% | Transition Covered Grade: 100% | State Covered Grade: 100% | Arc Covered Grade: n/a

FSMs

Ex	UNR	Name	State Average Grade	Transition Average Grade	Arc Average Grade
		(no filter)	(no filter)	(no filter)	(no filter)
		state	100%	100%	n/a

Showing 1 items

List Tabs state

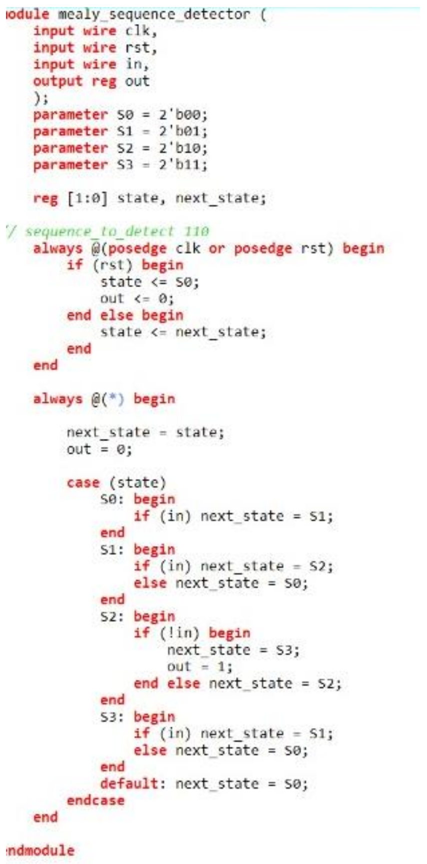
States | Transitions & Arc Sources

Ex	UNR	Name	Encoding	Score	Is Reset State
		(no filter)	(no filter)	(no filter)	(no filter)
		S0	00	✓ 3	true
		S1	01	✓ 3	false
		S2	10	✓ 2	false
		S3	11	✓ 2	false

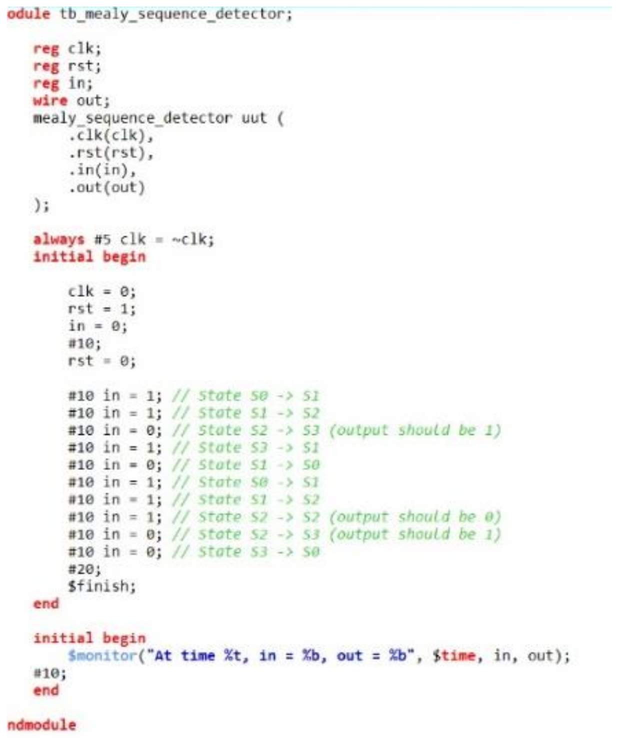
Showing 4 items

Loaded Run: ./cov_work/scope/test

RTL Coding:



```
iodule mealy_sequence_detector (
    input wire clk,
    input wire rst,
    input wire in,
    output reg out
    );
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;

    reg [1:0] state, next_state;

'/ sequence_to_detect 110
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= S0;
            out <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin

        next_state = state;
        out = 0;

        case (state)
            S0: begin
                if (in) next_state = S1;
            end
            S1: begin
                if (in) next_state = S2;
                else next_state = S0;
            end
            S2: begin
                if (!in) begin
                    next_state = S3;
                    out = 1;
                end else next_state = S2;
            end
            S3: begin
                if (in) next_state = S1;
                else next_state = S0;
            end
            default: next_state = S0;
        endcase
    end

ndmodule
```



```
odule tb_mealy_sequence_detector;

   reg clk;
   reg rst;
   reg in;
   wire out;
   mealy_sequence_detector uut (
       .clk(clk),
       .rst(rst),
       .in(in),
       .out(out)
   );

   always #5 clk = ~clk;
   initial begin

       clk = 0;
       rst = 1;
       in = 0;
       #10;
       rst = 0;

       #10 in = 1; // State S0 -> S1
       #10 in = 1; // State S1 -> S2
       #10 in = 0; // State S2 -> S3 (output should be 1)
       #10 in = 1; // State S3 -> S1
       #10 in = 0; // State S1 -> S0
       #10 in = 1; // State S0 -> S1
       #10 in = 1; // State S1 -> S2
       #10 in = 1; // State S2 -> S2 (output should be 0)
       #10 in = 0; // State S2 -> S3 (output should be 1)
       #10 in = 0; // State S3 -> S0
       #20;
       $finish;
   end

   initial begin
       $monitor("At time %t, in = %b, out = %b", $time, in, out);
   #10;
   end

ndmodule
```

Sign-_____________

ANKIT KUMAR 24EC4224

Simulation Setup:

- Cadence NCLAUNCH was used for functional verification.
- IMC (Integrated Metric Center) was used to perform code coverage analysis, ensuring all possible states and transitions are tested during simulation.

Simulation Parameters:

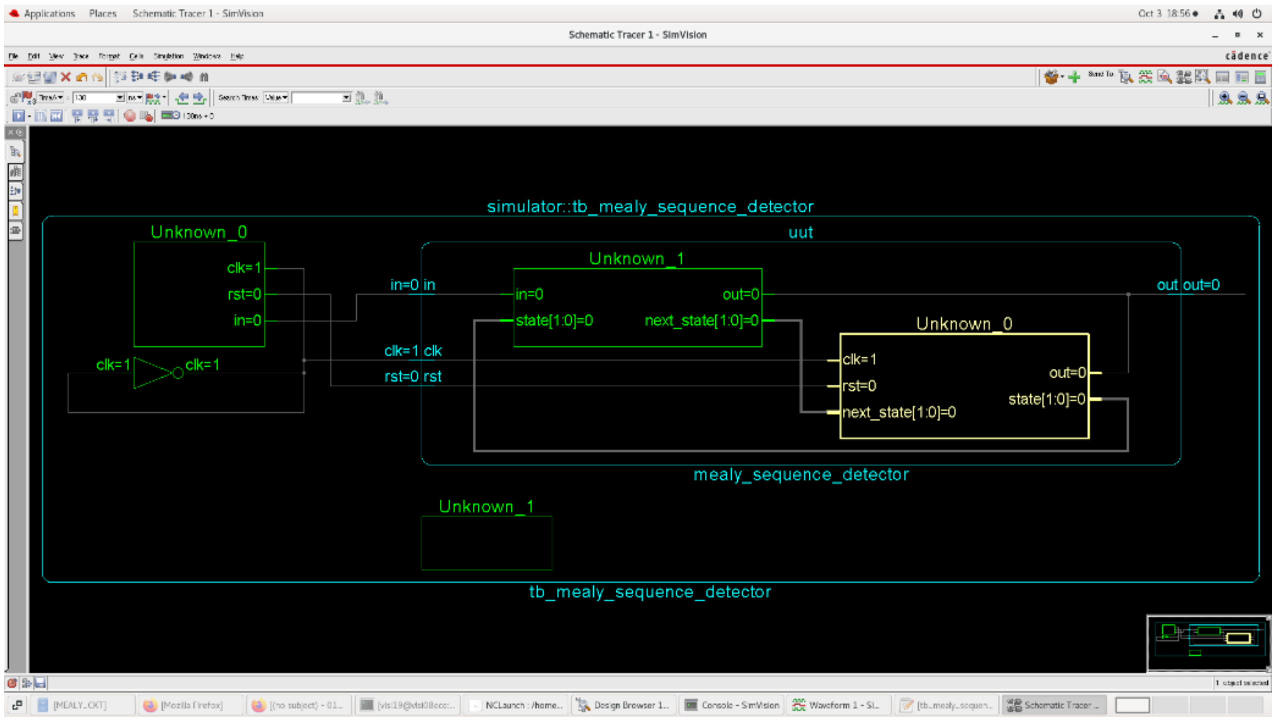
- Inputs: Clock, reset, input signal (`in`).
- Outputs: Output signal (`out`) based on both the current state and input signal.
- The testbench initializes with reset and provides various input signal sequences to verify the correct state transitions and outputs.

Experimental Results:

The Mealy machine was verified to transition through the correct states based on the input sequence "110".The output went high (`out = 1`) as soon as the sequence "110" was detected.

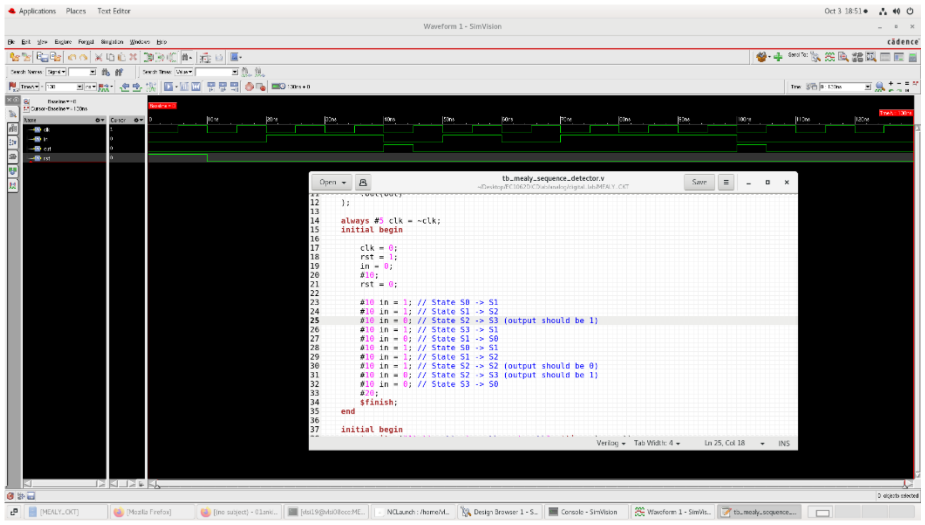
Functional verification

- *Logic diagram-*



Sign-____________

Waveform-



```
    );

    always #5 clk = ~clk;
    initial begin

        clk = 0;
        rst = 1;
        in = 0;
        #10;
        rst = 0;

        #10 in = 1; // State S0 -> S1
        #10 in = 1; // State S1 -> S2
        #10 in = 0; // State S2 -> S3 (output should be 1)
        #10 in = 1; // State S3 -> S1
        #10 in = 0; // State S1 -> S0
        #10 in = 1; // State S0 -> S1
        #10 in = 1; // State S1 -> S2
        #10 in = 1; // State S2 -> S2 (output should be 0)
        #10 in = 0; // State S2 -> S3 (output should be 1)
        #10 in = 0; // State S3 -> S0
        #20;
        $finish;
    end

    initial begin
```

Observations:

- The Mealy machine responded quickly to the input sequence "110", transitioning between states as expected and outputting the correct result.
- Using NCLAUNCH, the functional verification was successful, and no design errors were observed during the simulation.
- The IMC tool showed full code coverage, indicating that all possible state transitions were exercised in the testbench.

Sign-____________