Mini Project Report

On

"Traffic Light Controller Design using Verilog"

Submitted for the partial fulfilment
of

B. Tech.

in

ELECTRONICS AND COMMUNICATION ENGINEERING

SUBMITTED BY:

Student Name :- Ankit Kumar Yadav
(Roll No.) :- 2300270310039

Vth Sem - IIIrd Year
Section- ECE-1

SUBMITTED TO:

Dr. Tukur Gupta
(ASSO. PROF.)
ECE DEPT.

Ajay Kumar Garg Engineering College, Ghaziabad
27th Km Milestone, Delhi-Meerut Expressway, P.O. Adhyatmik Nagar, Ghaziabad-201015

Acknowledgement

I want to express my sincere gratitude and thanks to **Prof.** (**Dr.) Neelesh Kumar Gupta (HoD, ECE Department), Ajay Kumar Garg Engineering College, Ghaziabad** for granting me permission for my industrial training in the field of **" Traffic Light Controller Design using Verilog".**

I express my sincere thanks to **Asso. Prof.** (**Dr.) Tukur Gupta** for his cooperative attitude and consistence guidance, due to which I was able to complete my training successfully.

Finally, I pay my thankful regard and gratitude to the team members for their valuable help, support and guidance.

Student Name :- Ankit Kumar Yadav
Roll No. :- 2300270310039
Vth Sem - III Year
Section- ECE-1

Content

PROJECT

VLSI — Verilog HDL Project Collection

Chapter 1: Introduction to VLSI Design

1.1 Overview

VLSI (Very Large Scale Integration) is the process of creating integrated circuits by embedding millions of transistors on a single silicon chip. It enables compact, power-efficient, and high-speed electronic systems used in modern computing, communication, automotive, and IoT devices.

1.2 Classification of VLSI Design

VLSI design is classified into two primary domains — **Front-End Design** and **Back-End Design**.

🔹 Front-End Design

Front-end design focuses on the logical and behavioral aspects of the circuit. It defines how a chip behaves before it is physically implemented.

Key Stages of Front-End Design: - **Specification:** Defining the required functionality and performance metrics. - **RTL Design:** Writing Register Transfer Level (RTL) code using HDLs such as Verilog or VHDL. - **Functional Verification:** Ensuring that the RTL meets design specifications using testbenches and simulation tools like Icarus Verilog, ModelSim, or Synopsys VCS. - **Synthesis:** Converting RTL into gate-level netlists using tools like Synopsys Design Compiler or Cadence Genus. **Tools and Platforms Used:** - **HDLs:**

Verilog, SystemVerilog, VHDL - **Simulation:** Icarus Verilog, ModelSim,
GTKWave, Synopsys VCS - **Synthesis:** Cadence Genus, Synopsys Design Compiler - **Static Timing Analysis:** PrimeTime, Tempus

🔹 Back-End Design

Back-end design translates the logical netlist into a physical layout that can be fabricated.

Key Stages of Back-End Design: - **Floorplanning:** Arranging major functional blocks within the chip area. - **Placement:** Placing standard cells within the floorplan. - **Clock Tree Synthesis (CTS):** Distributing clock signals uniformly. - **Routing:** Connecting all logic elements physically with metal interconnects. - **Physical Verification:** Ensuring layout meets DRC/LVS standards. - **Tape-Out:** Finalizing design data for fabrication.

Tools and Platforms Used: - **Cadence Innovus** – for PnR (Place and Route) - **Synopsys IC Compiler II** – for physical implementation - **Mentor Calibre** – for DRC/LVS checks

🔹 Applications of VLSI Design

VLSI technology powers a wide range of modern applications: - **Microprocessors and Microcontrollers** - **Digital Signal Processors (DSPs)** - **ASICs (Application-Specific Integrated Circuits)** - **FPGAs (Field Programmable Gate Arrays)** - **Memory Chips (RAM, ROM, Flash)** - **AI/ML Accelerators** - **Communication Systems and IoT Devices**

🔹 Platforms Used for VLSI Design

- **Simulation and Debug:** Icarus Verilog, GTKWave
- **FPGA Implementation:** Xilinx Vivado, Intel Quartus
- **ASIC Design Flow:** Synopsys Design Compiler, Cadence Virtuoso, Mentor Calibre
- **Verification:** UVM (Universal Verification Methodology), SystemVerilog, QuestaSim

🔹 Importance of VLSI in Modern Technology

VLSI design enables integration, miniaturization, and high-performance processing, forming the backbone of modern digital systems like smartphones, embedded systems, and AI chips.

Chapter 2: Introduction to Verilog HDL for Digital VLSI Design

2.1 Overview

Verilog HDL (Hardware Description Language) is a textual language used to describe, simulate, and synthesize digital circuits. It supports modeling at various abstraction levels — behavioral, dataflow, and structural.

2.2 Types of Design Modeling in Verilog

🔹 Behavioral Modeling

- Describes **what** a circuit does rather than **how** it is built.
- Uses high-level constructs such as `always`, `if-else`, and `case` statements.
- Example: Describing a counter or FSM logic.

🔹 Dataflow Modeling

- Describes the circuit in terms of data flow between registers.
- Uses continuous assignment statements (`assign`).
- Example: Combinational circuits like adders or multiplexers.

🔹 Structural Modeling

- Describes the **interconnection** of submodules and gates.
- Example: Buildingafulladder from two half adders.

🔹 Gate-Level Modeling

- Uses built-in gate primitives (`and`, `or`, `xor`, etc.) to describe hardware.
- Useful for post-synthesis verification.

2.3 Verilog Design Flow

1. **Specification:** Define system requirements.
2. **RTL Coding:** Write Verilog code for modules.
3. **Testbench Development:** Write simulation code for verifying functionality.
4. **Simulation:** Run the design in Icarus Verilog or ModelSim.
5. **Synthesis:** Convert RTL to gate-level using Vivado or Synopsys.
6. **Implementation:** Load onto FPGA or proceed to back-end design.

2.4 Platforms and Tools for Verilog Design

Stage	Purpose	Common Tools
Simulation	Verify functionality	Icarus Verilog, ModelSim, VCS
Waveform Analysis	Observe timing and logic	GTKWave
Synthesis	Convert HDL to gate-level netlist	Vivado, Design Compiler
FPGA Programming	Hardware verification	Xilinx Vivado, Quartus Prime
Timing Analysis	Ensure setup/hold timing	PrimeTime, Tempus

2.5 Applications of Verilog in VLSI

- **ASIC Design:** Verilog is used to model, simulate, and verify custom logic.
- **FPGA Prototyping:** Real-time testing of digital systems.
- **Processor Design:** Modeling ALUs, control units, and data paths.
- **Embedded Systems:** Designing custom digital IP cores.

2.6 Advantages of Verilog HDL

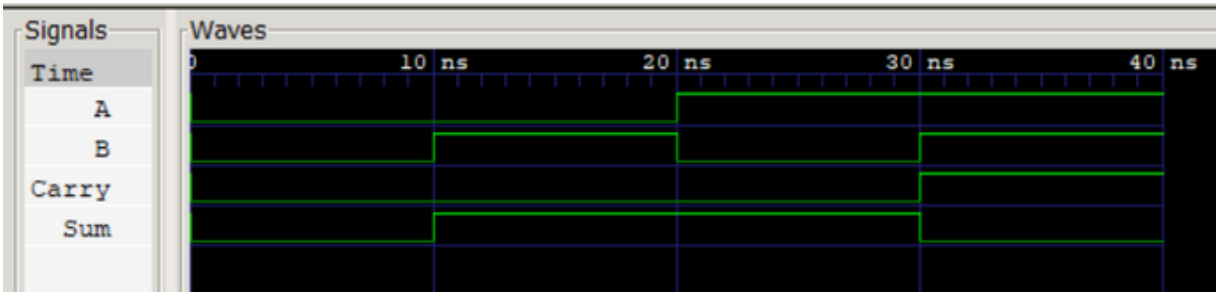
- Supports multiple abstraction levels.
- Industry-standard for ASIC and FPGA design.
- Compatible with advanced verification methodologies (UVM).
- Enables efficient simulation and debugging of digital systems.

Chapter 3: Combinational Circuits

1) Half Adder

```
//half adder
// test
module half_adder(
    input A,     // First input bit
    input B,     // Second input bit
    output Sum,  // Output sum (A XOR B)
    output Carry // Output carry (A AND B)
);

    // Logic implementation
    assign Sum = A ^ B;   // XOR gate for sum
    assign Carry = A & B; // AND gate for carry

endmodule
```

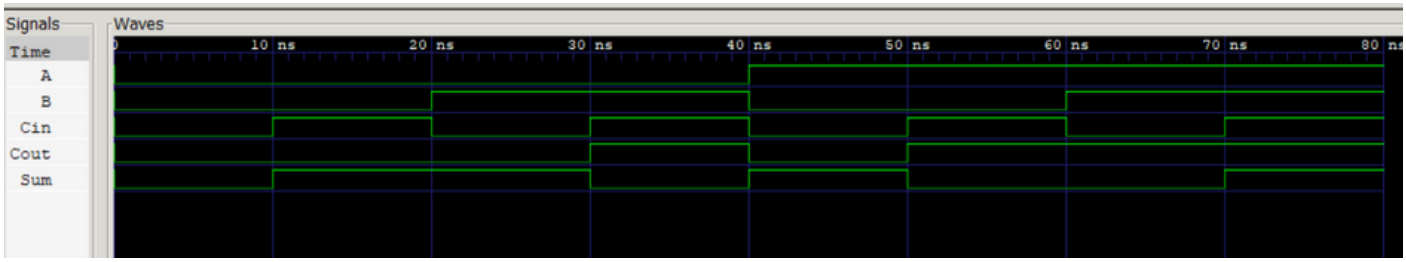
```
//half_adder_tb.v
// Testbench for Half Adder with GTKWave output

`timescale 1ns/1ps  // Define simulation time unit and precision

module testbench;
    reg A, B;          // Inputs are declared as 'reg' because we drive them
    wire Sum, Carry;   // Outputs are declared as 'wire'

    // Instantiate the half adder module
    half_adder uut (
        .A(A),
        .B(B),
        .Sum(Sum),
        .Carry(Carry)
    );

    // Initial block for stimulus
    initial begin
        // Create waveform dump file for GTKWave
        $dumpfile("half_adder.vcd");    // VCD = Value Change Dump
        $dumpvars(0, testbench);        // Dump all variables in testbench

        // Display results in terminal
        $monitor("Time=%0t | A=%b, B=%b | Sum=%b, Carry=%b", $time, A, B, Sum, Carry);

        // Apply all input combinations
        A = 0; B = 0; #10;
        A = 0; B = 1; #10;
        A = 1; B = 0; #10;
        A = 1; B = 1; #10;

        $finish; // End simulation
    end
endmodule
```



2) Full Adder

```v
// File: full_adder.v
// Simple 1-bit Full Adder

module full_adder (
    input wire A,
    input wire B,
    input wire Cin,
    output wire Sum,
    output wire Cout
);
    // Full Adder Logic
    assign Sum  = A ^ B ^ Cin;                  // XOR for sum
    assign Cout = (A & B) | (B & Cin) | (A & Cin); // Carry-out logic
endmodule
```

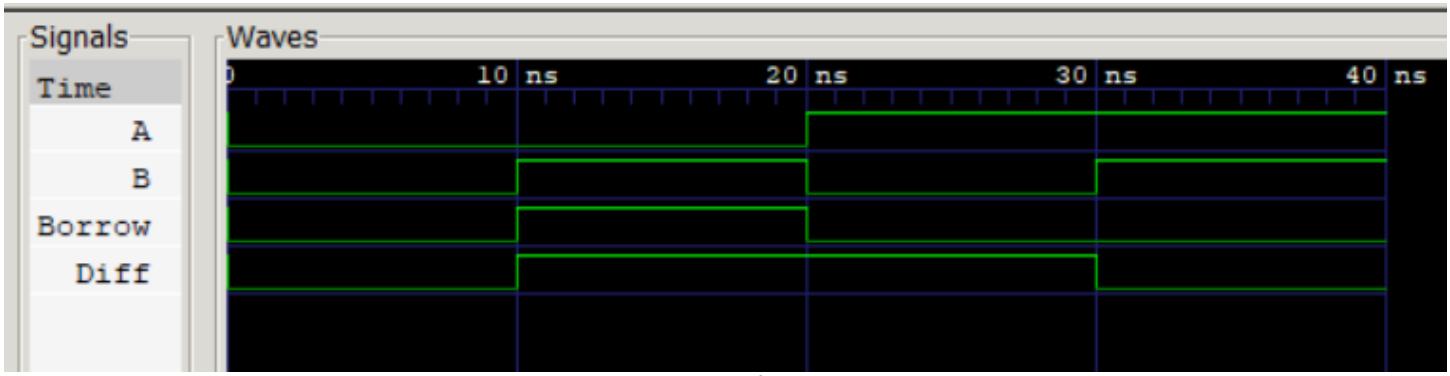
```v
// File: full_adder_tb.v
`timescale 1ns/1ps
//`include "full_adder.v"

module full_adder_tb;
    reg A, B, Cin;
    wire Sum, Cout;

    // Instantiate Device Under Test (DUT)
    full_adder uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    initial begin
        // For GTKWave output
        $dumpfile("full_adder.vcd");
        $dumpvars(0, full_adder_tb);

        // Display results in terminal
        $monitor("A=%b, B=%b, Cin=%b | Sum=%b, Cout=%b", A, B, Cin, Sum, Cout);

        // Apply all 8 input combinations
        A=0; B=0; Cin=0; #10;
        A=0; B=0; Cin=1; #10;
        A=0; B=1; Cin=0; #10;
        A=0; B=1; Cin=1; #10;
        A=1; B=0; Cin=0; #10;
        A=1; B=0; Cin=1; #10;
        A=1; B=1; Cin=0; #10;
        A=1; B=1; Cin=1; #10;

        $finish;
    end
endmodule
```



3) Half Subtractor

```
// Simple Half Subtractor using gate-level modeling
//half_sub.v
//half_sub


module half_sub(
    input A,       // Minuend (number to subtract from)
    input B,       // Subtrahend (number to subtract)
    output Diff,   // Difference output (A XOR B)
    output Borrow  // Borrow output (~A & B)
);

    // Logic implementation
    assign Diff = A ^ B;     // XOR gate gives difference
    assign Borrow = (~A) & B; // Borrow occurs when A=0 and B=1

endmodule
```

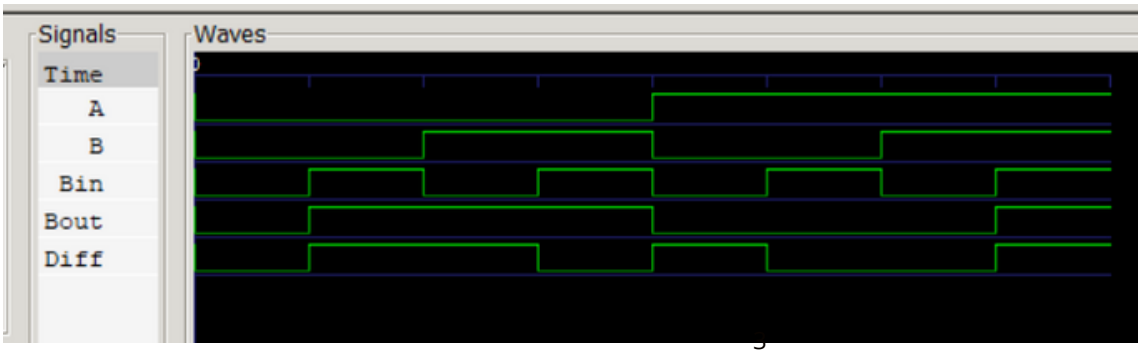
```
//half_sub_tb.v
// testbench
// Testbench for Half Subtractor with GTKWave support

`timescale 1ns/1ps  // Define simulation time and precision

module testbench;
    reg A, B;            // Inputs are reg (driven by initial block)
    wire Diff, Borrow;   // Outputs are wire (driven by DUT)

    // Instantiate the Half Subtractor module (Device Under Test)
    half_sub uut (
        .A(A),
        .B(B),
        .Diff(Diff),
        .Borrow(Borrow)
    );

    // Initial block for test cases
    initial begin
        // Create waveform dump for GTKWave
        $dumpfile("half_sub.vcd");
        $dumpvars(0, testbench);

        // Display results in console
        $monitor("Time=%0t | A=%b, B=%b | Diff=%b, Borrow=%b", $time, A, B, Diff, Borrow);

        // Apply all input combinations (00, 01, 10, 11)
        A = 0; B = 0; #10;
        A = 0; B = 1; #10;
        A = 1; B = 0; #10;
        A = 1; B = 1; #10;

        $finish; // End simulation
    end
endmodule
```



4) Full Subtracter

```
module full_sub(
    input A,        // Minuend bit
    input B,        // Subtrahend bit
    input Bin,      // Borrow input bit
    output Diff,    // Difference output (A - B - Bin)
    output Bout     // Borrow output
);

    // Logic for Difference and Borrow
    assign Diff = A ^ B ^ Bin;                  // XOR for Difference
    assign Bout = (~A & B) | ((~(A ^ B)) & Bin); // Borrow logic

endmodule
```

```
`timescale 1ns/1ps

module testbench;
    reg A, B, Bin;          // Inputs
    wire Diff, Bout;        // Outputs

    // Instantiate the full_sub module
    full_sub uut (
        .A(A),
        .B(B),
        .Bin(Bin),
        .Diff(Diff),
        .Bout(Bout)
    );

    initial begin
        // Create waveform dump for GTKWave
        $dumpfile("full_sub.vcd");
        $dumpvars(0, testbench);

        // Display values in terminal
        $monitor("Time=%0t | A=%b, B=%b, Bin=%b | Diff=%b, Bout=%b",
                 $time, A, B, Bin, Diff, Bout);

        // Apply all 8 input combinations (3-bit)
        A=0; B=0; Bin=0; #10;
        A=0; B=0; Bin=1; #10;
        A=0; B=1; Bin=0; #10;
        A=0; B=1; Bin=1; #10;
        A=1; B=0; Bin=0; #10;
        A=1; B=0; Bin=1; #10;
        A=1; B=1; Bin=0; #10;
        A=1; B=1; Bin=1; #10;

        $finish;
    end
endmodule
```

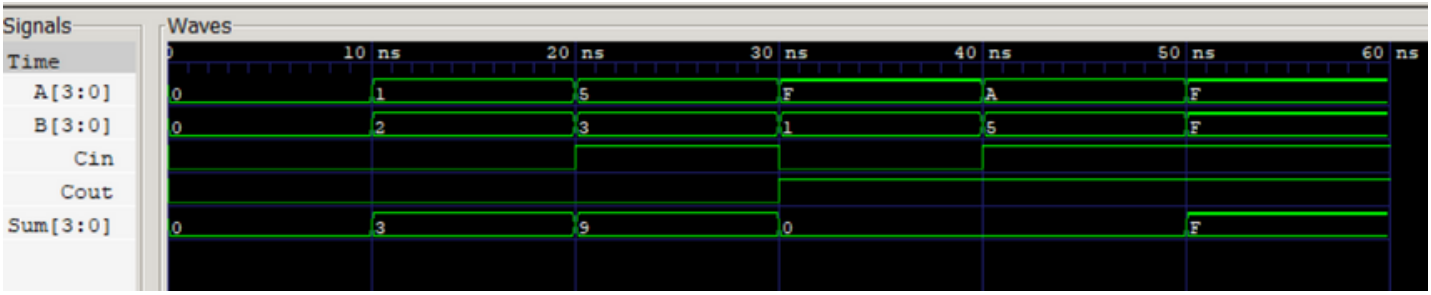


5) 4-bit Adder

```
 // 4-bit Ripple Carry Adder using Full Adders

module adder4bit (
    input [3:0] A,      // 4-bit input A
    input [3:0] B,      // 4-bit input B
    input Cin,          // Carry input
    output [3:0] Sum,   // 4-bit Sum output
    output Cout         // Carry output
);

    wire C1, C2, C3; // Internal carry connections

    // Bit 0
    assign {C1, Sum[0]} = A[0] + B[0] + Cin;

    // Bit 1
    assign {C2, Sum[1]} = A[1] + B[1] + C1;

    // Bit 2
    assign {C3, Sum[2]} = A[2] + B[2] + C2;

    // Bit 3
    assign {Cout, Sum[3]} = A[3] + B[3] + C3;

endmodule
```

```
`timescale 1ns/1ps

module testbench;
    reg [3:0] A, B;   // 4-bit inputs
    reg Cin;          // Carry input
    wire [3:0] Sum;   // 4-bit sum output
    wire Cout;        // Carry output

    // Instantiate the DUT (Device Under Test)
    adder4bit uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    initial begin
        // Create waveform file for GTKWave
        $dumpfile("4bit_adder.vcd");
        $dumpvars(0, testbench);

        // Display results on terminal
        $monitor("Time=%0t | A=%b, B=%b, Cin=%b | Sum=%b, Cout=%b",
                 $time, A, B, Cin, Sum, Cout);

        // Apply test vectors
        A = 4'b0000; B = 4'b0000; Cin = 0; #10;
        A = 4'b0001; B = 4'b0010; Cin = 0; #10;
        A = 4'b0101; B = 4'b0011; Cin = 1; #10;
        A = 4'b1111; B = 4'b0001; Cin = 0; #10;
        A = 4'b1010; B = 4'b0101; Cin = 1; #10;
        A = 4'b1111; B = 4'b1111; Cin = 1; #10;

        $finish; // End simulation
    end
endmodule
```

4bit adder

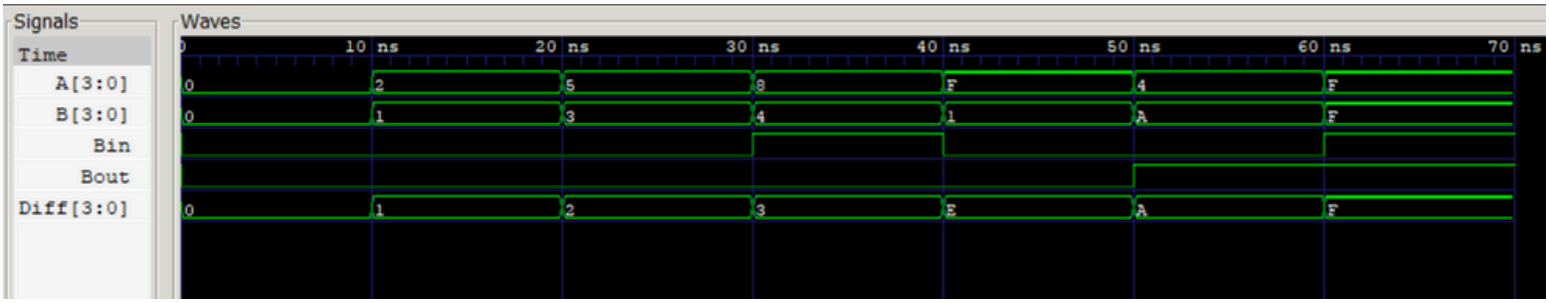


5) 4 Bit Subtractor

```
module sub4bit (
    input [3:0] A,      // 4-bit Minuend
    input [3:0] B,      // 4-bit Subtrahend
    input Bin,          // Initial Borrow input
    output [3:0] Diff, // 4-bit Difference output
    output Bout         // Final Borrow output
);

    // Internal borrow connections between bits
    wire B1, B2, B3;

    // Bit 0 (Least Significant Bit)
    assign Diff[0] = A[0] ^ B[0] ^ Bin;
    assign B1 = (~A[0] & B[0]) | (~(A[0] ^ B[0]) & Bin);

    // Bit 1
    assign Diff[1] = A[1] ^ B[1] ^ B1;
    assign B2 = (~A[1] & B[1]) | (~(A[1] ^ B[1]) & B1);

    // Bit 2
    assign Diff[2] = A[2] ^ B[2] ^ B2;
    assign B3 = (~A[2] & B[2]) | (~(A[2] ^ B[2]) & B2);

    // Bit 3 (Most Significant Bit)
    assign Diff[3] = A[3] ^ B[3] ^ B3;
    assign Bout = (~A[3] & B[3]) | (~(A[3] ^ B[3]) & B3);

endmodule
```

```
`timescale 1ns/1ps

module testbench;
    reg [3:0] A, B;   // 4-bit inputs
    reg Bin;          // Borrow input
    wire [3:0] Diff;  // 4-bit Difference output
    wire Bout;        // Borrow output

    // Instantiate the DUT (Device Under Test)
    sub4bit uut (
        .A(A),
        .B(B),
        .Bin(Bin),
        .Diff(Diff),
        .Bout(Bout)
    );
```

```
    initial begin
        // Create waveform for GTKWave
        $dumpfile("4bit_sub.vcd");
        $dumpvars(0, testbench);

        // Monitor output on terminal
        $monitor("Time=%0t | A=%b, B=%b, Bin=%b | Diff=%b, Bout=%b",
                 $time, A, B, Bin, Diff, Bout);

        // Apply test cases
        A = 4'b0000; B = 4'b0000; Bin = 0; #10;
        A = 4'b0010; B = 4'b0001; Bin = 0; #10;
        A = 4'b0101; B = 4'b0011; Bin = 0; #10;
        A = 4'b1000; B = 4'b0100; Bin = 1; #10;
        A = 4'b1111; B = 4'b0001; Bin = 0; #10;
        A = 4'b0100; B = 4'b1010; Bin = 0; #10;
        A = 4'b1111; B = 4'b1111; Bin = 1; #10;

        $finish;
    end
endmodule
```



5) Multiplexers (2x1, 4x1, 8x1)

A Multiplexer (MUX) is a combinational logic circuit that selects one input from several inputs and forwards it to a single output line, based on the value of selection (control) inputs.

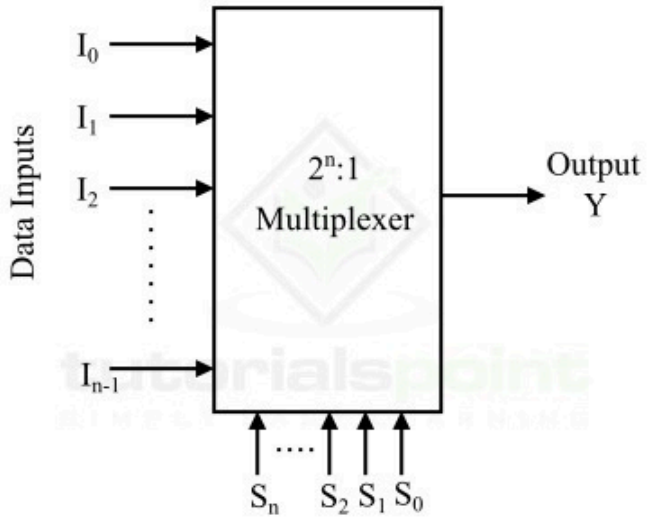
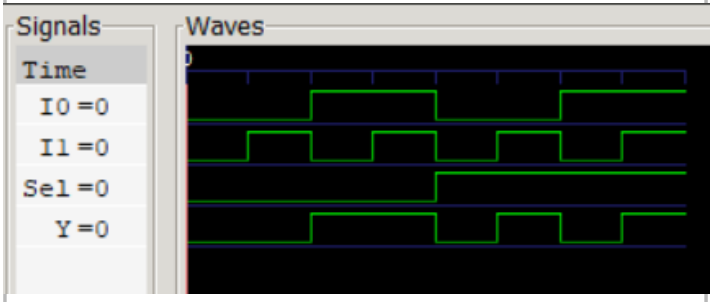


Figure 1 - Digital Multiplexer

5) Multiplexers (2x1)

```
module mux2_1 (
    input I0,        // Input line 0
    input I1,        // Input line 1
    input Sel,       // Select line
    output Y         // Output
);

    // Multiplexer logic:
    // If Sel=0 → Y = I0
    // If Sel=1 → Y = I1
    assign Y = (I0 & ~Sel) | (I1 & Sel);

endmodule
```

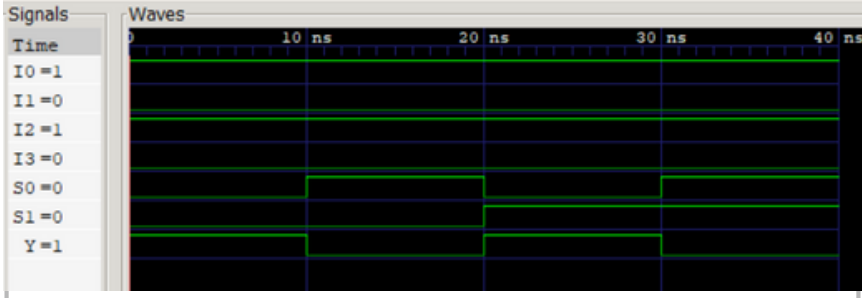
```
`timescale 1ns/1ps

module testbench;
    reg I0, I1, Sel;   // Declare inputs as reg for stimulus
    wire Y;            // Output wire to observe MUX output

    // Instantiate the Device Under Test (DUT)
    mux2_1 uut (
        .I0(I0),
        .I1(I1),
        .Sel(Sel),
        .Y(Y)
    );

    initial begin
        // Dump waveform for GTKWave
        $dumpfile("mux2_1.vcd");
        $dumpvars(0, testbench);

        // Display header in console
        $monitor("Time=%0t | I0=%b, I1=%b, Sel=%b | Y=%b",
                 $time, I0, I1, Sel, Y);

        // Apply test patterns
        I0 = 0; I1 = 0; Sel = 0; #10;
        I0 = 0; I1 = 1; Sel = 0; #10;
        I0 = 1; I1 = 0; Sel = 0; #10;
        I0 = 1; I1 = 1; Sel = 0; #10;

        I0 = 0; I1 = 0; Sel = 1; #10;
        I0 = 0; I1 = 1; Sel = 1; #10;
        I0 = 1; I1 = 0; Sel = 1; #10;
        I0 = 1; I1 = 1; Sel = 1; #10;

        $finish; // End simulation
    end
endmodule
```



5) Multiplexers (4x1)

```
module mux4_1 (
    input I0,        // Input line 0
    input I1,        // Input line 1
    input I2,        // Input line 2
    input I3,        // Input line 3
    input S1,        // Select line 1 (MSB)
    input S0,        // Select line 0 (LSB)
    output Y         // Output
);

    // Multiplexer logic
    assign Y = (~S1 & ~S0 & I0) |
               (~S1 &  S0 & I1) |
               ( S1 & ~S0 & I2) |
               ( S1 &  S0 & I3);

endmodule
```

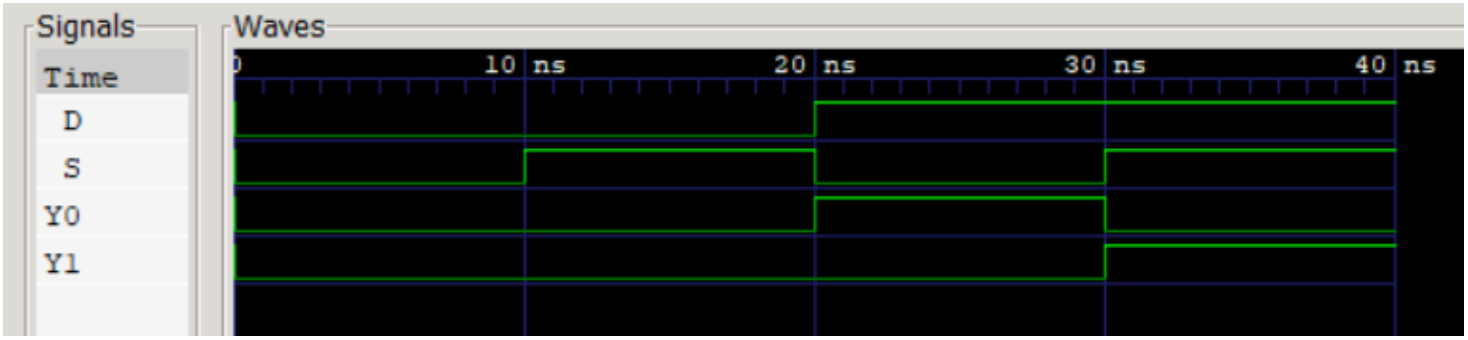
```
`timescale 1ns/1ps

module testbench;
    reg I0, I1, I2, I3;   // Inputs to MUX
    reg S1, S0;           // Select lines
    wire Y;               // Output of MUX

    // Instantiate the Device Under Test (DUT)
    mux4_1 uut (
        .I0(I0),
        .I1(I1),
        .I2(I2),
        .I3(I3),
        .S1(S1),
        .S0(S0),
        .Y(Y)
    );

    initial begin
        // Dump waveform for GTKWave
        $dumpfile("mux4_1.vcd");
        $dumpvars(0, testbench);

        // Display values in terminal
        $monitor("Time=%0t | I0=%b I1=%b I2=%b I3=%b | S1=%b S0=%b | Y=%b",
                 $time, I0, I1, I2, I3, S1, S0, Y);

        // Initialize inputs
        I0 = 1; I1 = 0; I2 = 1; I3 = 0; // Sample pattern

        // Apply all select combinations
        S1 = 0; S0 = 0; #10;   // Expect Y = I0 = 1
        S1 = 0; S0 = 1; #10;   // Expect Y = I1 = 0
        S1 = 1; S0 = 0; #10;   // Expect Y = I2 = 1
        S1 = 1; S0 = 1; #10;   // Expect Y = I3 = 0

        $finish; // End simulation
    end
endmodule
```



6) De-Multiplexer (1x2, 1x4)

```
module demux2_1 (
    input D,          // Input data (to be distributed)
    input S,          // Select line (decides which output gets D)
    output Y0,        // Output 0
    output Y1         // Output 1
);

    // Logic for 1-to-2 Demultiplexer
    assign Y0 = (~S) & D;  // If S=0 → Y0 = D
    assign Y1 = S & D;     // If S=1 → Y1 = D

endmodule
```

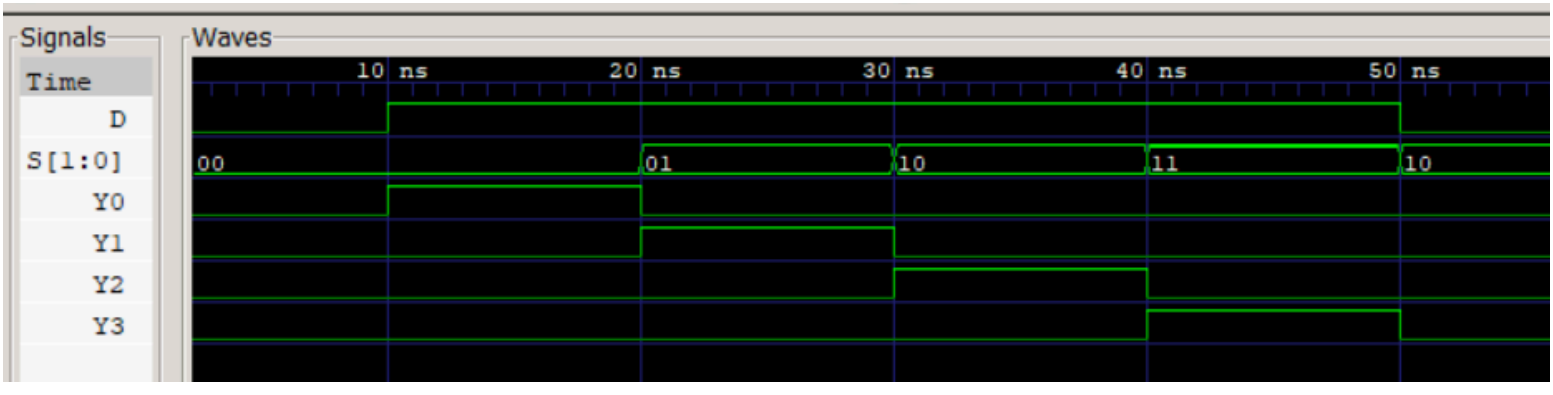
```
`timescale 1ns/1ps

module testbench;
    reg D, S;          // Inputs
    wire Y0, Y1;       // Outputs

    // Instantiate the DEMUX module
    demux2_1 uut (
        .D(D),
        .S(S),
        .Y0(Y0),
        .Y1(Y1)
    );

    // Initial block for test cases
    initial begin
        // Generate waveform dump for GTKWave
        $dumpfile("demux2_1.vcd");
        $dumpvars(0, testbench);

        // Display header
        $display("Time | D S | Y0 Y1");
        $monitor("%4t | %b %b | %b  %b", $time, D, S, Y0, Y1);

        // Test all possible input combinations
        D = 0; S = 0; #10;
        D = 0; S = 1; #10;
        D = 1; S = 0; #10;
        D = 1; S = 1; #10;

        $finish;
    end
endmodule
```



6) ii) De-Multiplexer (1x4)

```
module demux4_1 (
    input D,            // Input data to be distributed
    input [1:0] S,      // 2-bit select lines (S1, S0)
    output Y0,          // Output 0
    output Y1,          // Output 1
    output Y2,          // Output 2
    output Y3           // Output 3
);

    // Logic for 1-to-4 Demultiplexer
    assign Y0 = (~S[1] & ~S[0]) & D;  // S=00 → Y0 = D
    assign Y1 = (~S[1] &  S[0]) & D;  // S=01 → Y1 = D
    assign Y2 = ( S[1] & ~S[0]) & D;  // S=10 → Y2 = D
    assign Y3 = ( S[1] &  S[0]) & D;  // S=11 → Y3 = D

endmodule
```

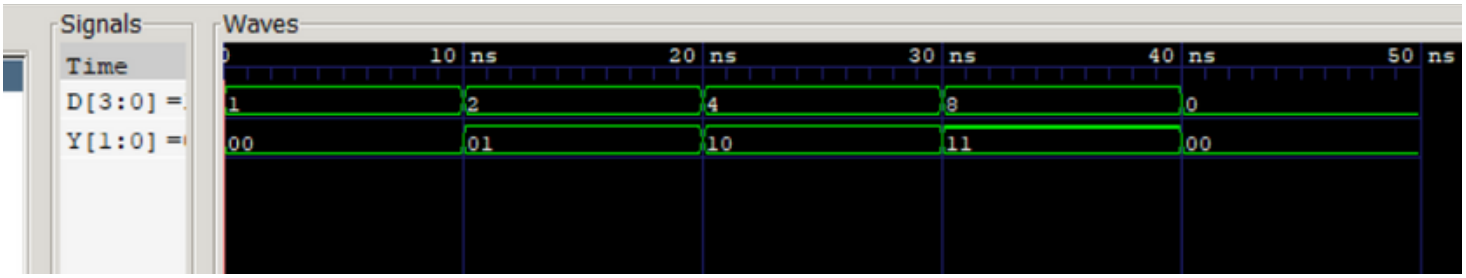
```
`timescale 1ns/1ps

module testbench;
    reg D;              // Input signal
    reg [1:0] S;        // 2-bit select input
    wire Y0, Y1, Y2, Y3;  // Outputs

    // Instantiate the DEMUX module
    demux4_1 uut (
        .D(D),
        .S(S),
        .Y0(Y0),
        .Y1(Y1),
        .Y2(Y2),
        .Y3(Y3)
    );

    // Initial block for test cases
    initial begin
        // Create waveform file for GTKWave
        $dumpfile("demux4_1.vcd");
        $dumpvars(0, testbench);

        // Display header
        $display("Time | D | S1 S0 | Y0 Y1 Y2 Y3");
        $monitor("%4t | %b | %b%b | %b  %b  %b  %b", $time, D, S[1], S[0], Y0, Y1, Y2, Y3);

        // Test all combinations
        D = 0; S = 2'b00; #10;   // No output active since D=0
        D = 1; S = 2'b00; #10;   // Output Y0 active
        D = 1; S = 2'b01; #10;   // Output Y1 active
        D = 1; S = 2'b10; #10;   // Output Y2 active
        D = 1; S = 2'b11; #10;   // Output Y3 active
        D = 0; S = 2'b10; #10;   // Input low again

        $finish; // End simulation
    end
endmodule
```



7) Encoder

```
module encoder_4to2 (
    input  [3:0] D,   // 4 input lines (D3, D2, D1, D0)
    output [1:0] Y    // 2-bit binary output
);

    // Basic 4-to-2 encoding logic
    // The highest-order bit that is '1' determines the output.
    assign Y[1] = D[3] | D[2];          // MSB output
    assign Y[0] = D[3] | D[1];          // LSB output

endmodule
```

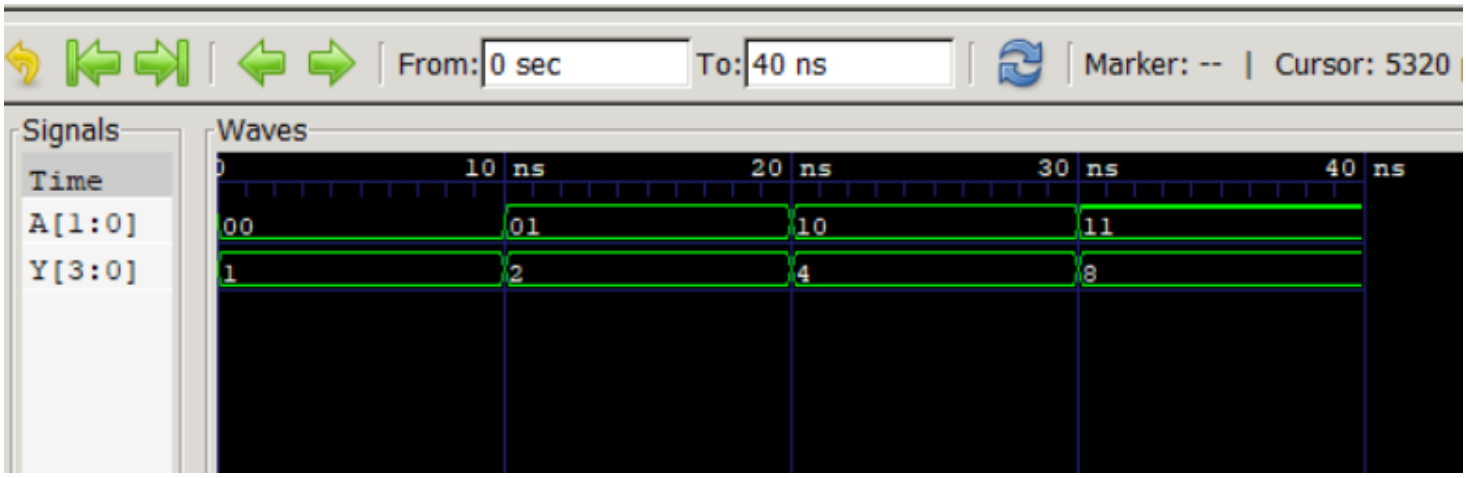
```
`timescale 1ns/1ps

module testbench;
    reg  [3:0] D;       // 4-bit input
    wire [1:0] Y;       // 2-bit encoded output

    // Instantiate the encoder
    encoder_4to2 uut (
        .D(D),
        .Y(Y)
    );

    // Simulation block
    initial begin
        // Generate waveform dump
        $dumpfile("encoder_4to2.vcd");
        $dumpvars(0, testbench);

        // Display format
        $display("Time | D3 D2 D1 D0 | Y1 Y0");
        $monitor("%4t |  %b  %b  %b  %b |  %b  %b",
                 $time, D[3], D[2], D[1], D[0], Y[1], Y[0]);

        // Test cases for all possible inputs
        D = 4'b0001; #10;  // Input = 0001 → Output = 00
        D = 4'b0010; #10;  // Input = 0010 → Output = 01
        D = 4'b0100; #10;  // Input = 0100 → Output = 10
        D = 4'b1000; #10;  // Input = 1000 → Output = 11
        D = 4'b0000; #10;  // No input active (undefined)

        $finish;
    end
endmodule
```



7) Decoder

```
module decoder_2to4 (
    input [1:0] A,    // 2-bit input (A1, A0)
    output [3:0] Y    // 4 output lines
);

    // Logic for 2-to-4 decoder
    // Each output corresponds to one combination of input
    assign Y[0] = (~A[1]) & (~A[0]);  // A=00 → Y0=1
    assign Y[1] = (~A[1]) & ( A[0]);  // A=01 → Y1=1
    assign Y[2] = ( A[1]) & (~A[0]);  // A=10 → Y2=1
    assign Y[3] = ( A[1]) & ( A[0]);  // A=11 → Y3=1

endmodule
```

```
`timescale 1ns/1ps

module testbench;
    reg [1:0] A;       // Input for decoder
    wire [3:0] Y;      // Output of decoder

    // Instantiate the decoder
    decoder_2to4 uut (
        .A(A),
        .Y(Y)
    );

    // Initial block for test cases
    initial begin
        // Generate waveform dump for GTKWave
        $dumpfile("decoder_2to4.vcd");
        $dumpvars(0, testbench);

        // Display header
        $display("Time | A1 A0 | Y3 Y2 Y1 Y0");
        $monitor("%4t |  %b  %b |  %b  %b  %b  %b",
                 $time, A[1], A[0], Y[3], Y[2], Y[1], Y[0]);

        // Test all input combinations
        A = 2'b00; #10;  // Expect Y0 = 1
        A = 2'b01; #10;  // Expect Y1 = 1
        A = 2'b10; #10;  // Expect Y2 = 1
        A = 2'b11; #10;  // Expect Y3 = 1

        $finish;  // End simulation
    end
endmodule
```



8) Comparator (4-bit)

```
//   This is a simple 2-bit digital comparator.
//   It compares two 2-bit numbers (A and B) and produces
//   three outputs: A_greater, A_equal, A_smaller.

module comparator_(
    input [1:0] A,    // 2-bit input A
    input [1:0] B,    // 2-bit input B
    output reg A_greater, // High if A > B
    output reg A_equal,   // High if A == B
    output reg A_smaller  // High if A < B
);

    // Always block to compare A and B
    always @(*) begin
        if (A > B) begin
            A_greater = 1;
            A_equal   = 0;
            A_smaller = 0;
        end
        else if (A == B) begin
            A_greater = 0;
            A_equal   = 1;
            A_smaller = 0;
        end
        else begin
            A_greater = 0;
            A_equal   = 0;
            A_smaller = 1;
        end
    end

endmodule
```

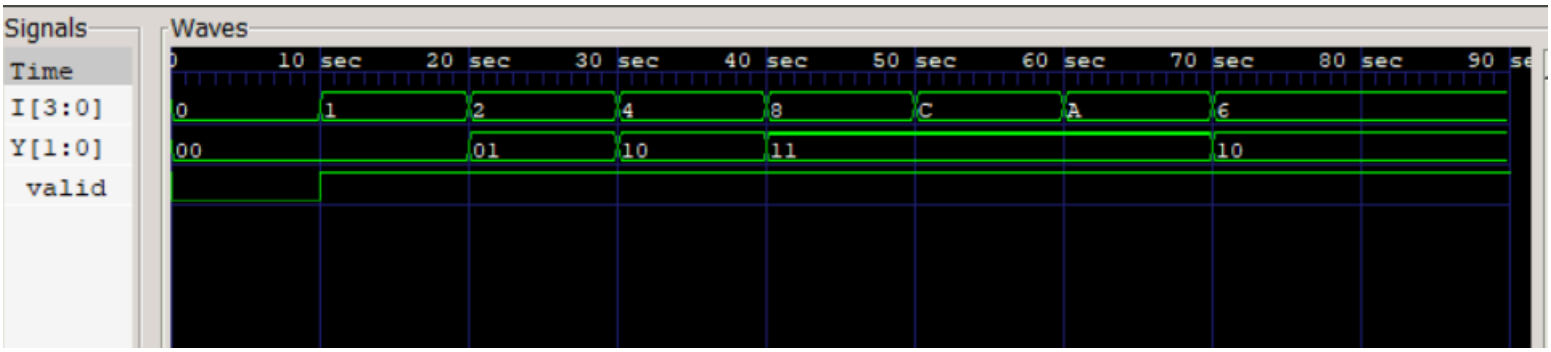
```
module comparator_tb;

    // Declare inputs and outputs
    reg [1:0] A;
    reg [1:0] B;
    wire A_greater, A_equal, A_smaller;

    // Instantiate the Design Under Test (DUT)
    comparator_ dut (
        .A(A),
        .B(B),
        .A_greater(A_greater),
        .A_equal(A_equal),
        .A_smaller(A_smaller)
    );

    // Waveform dump setup
    initial begin
        $dumpfile("comparator.vcd");       // File to save waveform data
        $dumpvars(0, comparator_tb);       // Dump all variables in this module
    end

    // Apply test inputs
    initial begin
        // Display header in console
        $display("A  B  |  Greater Equal Smaller");
        $display("-----------------------------");

        // Test combinations
        A = 2'b00; B = 2'b00; #10; $display("%b  %b  |     %b      %b      %b", A, B, A_greater, A_equal, A_smaller);
        A = 2'b01; B = 2'b00; #10; $display("%b  %b  |     %b      %b      %b", A, B, A_greater, A_equal, A_smaller);
        A = 2'b10; B = 2'b11; #10; $display("%b  %b  |     %b      %b      %b", A, B, A_greater, A_equal, A_smaller);
        A = 2'b11; B = 2'b01; #10; $display("%b  %b  |     %b      %b      %b", A, B, A_greater, A_equal, A_smaller);
        A = 2'b01; B = 2'b01; #10; $display("%b  %b  |     %b      %b      %b", A, B, A_greater, A_equal, A_smaller);

        #10;
        $finish;
    end
```

comparator

9) Priority Encoder (4-bit)

```
//Input  : 4-bit input I (I[3] has highest priority)
// Output : 2-bit encoded output Y
//          'valid' indicates if any input is active

module pri_enc (
    input  [3:0] I,       // 4-bit input lines
    output reg [1:0] Y,   // 2-bit encoded output
    output reg valid       // High when any input is active
);

    always @(*) begin
        // Default outputs
        Y = 2'b00;
        valid = 1'b0;

        // Priority logic (highest bit has highest priority)
        casez (I)
            4'b1???: begin
                Y = 2'b11;  // I3 active
                valid = 1'b1;
            end
            4'b01??: begin
                Y = 2'b10;  // I2 active
                valid = 1'b1;
            end
            4'b001?: begin
                Y = 2'b01;  // I1 active
                valid = 1'b1;
            end
            4'b0001: begin
                Y = 2'b00;  // I0 active
                valid = 1'b1;
            end
            default: begin
                Y = 2'b00;
                valid = 1'b0; // No input active
            end
        endcase
    end

endmodule
```

```
module pri_enc_tb;

    reg  [3:0] I;         // Input to encoder
    wire [1:0] Y;         // Encoded output
    wire valid;           // Valid output signal

    // Instantiate the DUT (Design Under Test)
    pri_enc dut (
        .I(I),
        .Y(Y),
        .valid(valid)
    );

    // Waveform dump for GTKWave
    initial begin
        $dumpfile("pri_enc.vcd");      // File to store waveform data
        $dumpvars(0, pri_enc_tb);      // Dump all variables from testbench
    end

    // Apply test inputs
    initial begin
        $display("Time\tInput\tEncoded\tValid");
        $display("------------------------------");

        I = 4'b0000; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b0001; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b0010; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b0100; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b1000; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b1100; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b1010; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);
        I = 4'b0110; #10; $display("%0t\t%b\t%b\t%b", $time, I, Y, valid);

        #10;
        $finish;
    end

endmodule
```

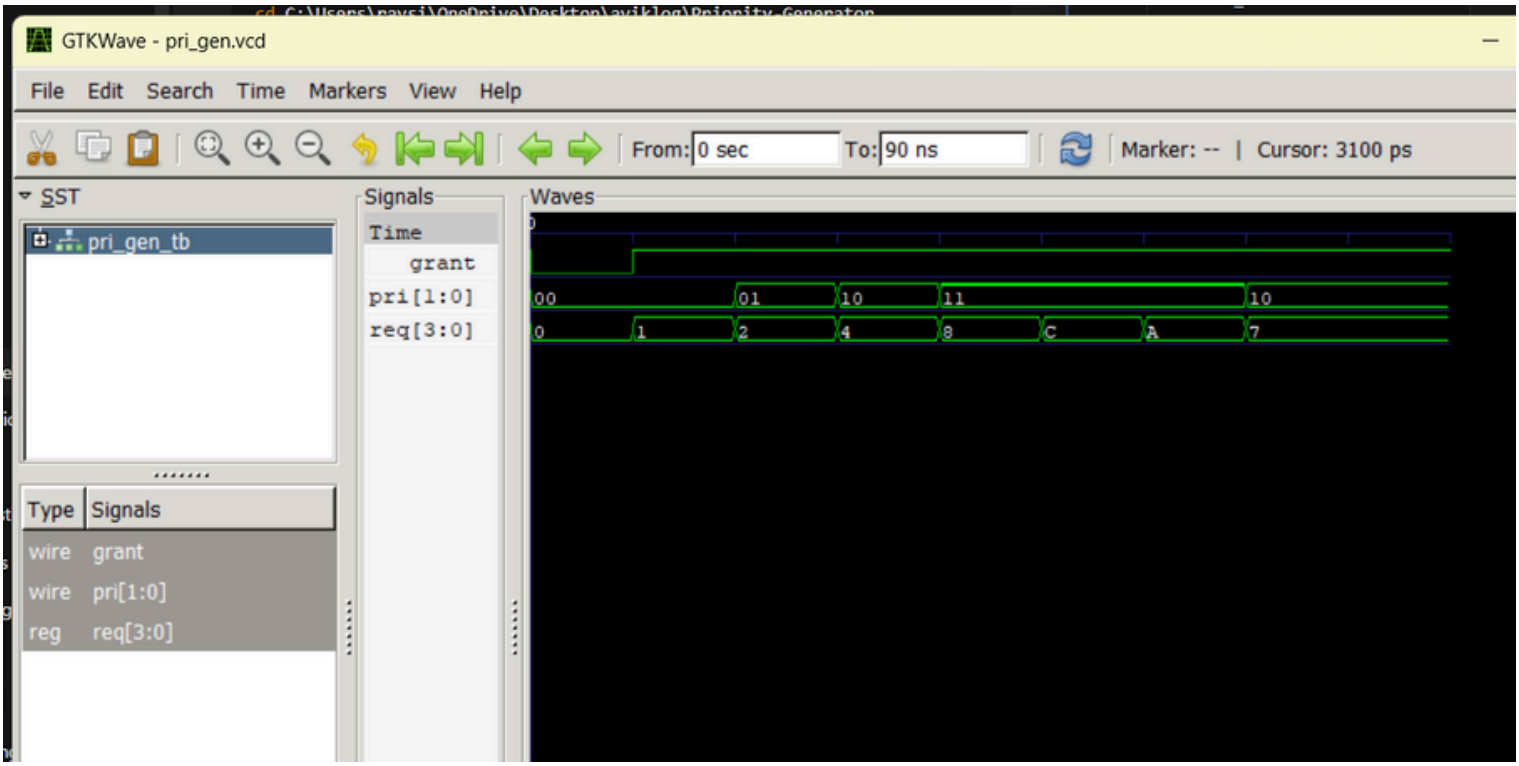


9) Priority Generator

```
module pri_gen (
    input  [3:0] req,      // 4 request lines (req[3] = highest priority)
    output reg [1:0] pri,  // 2-bit priority output
    output reg grant        // High when any request is active
);

    always @(*) begin
        // Default values
        pri = 2'b00;
        grant = 1'b0;

        // Priority checking — higher bit has higher priority
        if (req[3]) begin
            pri = 2'b11;  // Highest priority
            grant = 1'b1;
        end
        else if (req[2]) begin
            pri = 2'b10;  // Next highest
            grant = 1'b1;
        end
        else if (req[1]) begin
            pri = 2'b01;  // Medium priority
            grant = 1'b1;
        end
        else if (req[0]) begin
            pri = 2'b00;  // Lowest priority
            grant = 1'b1;
        end
        else begin
            pri = 2'b00;  // No request
            grant = 1'b0;
        end
    end

endmodule
```

```
`timescale 1ns / 1ps
module pri_gen_tb;

    reg  [3:0] req;         // Input request lines
    wire [1:0] pri;         // Output priority code
    wire grant;             // Grant signal (indicates valid request)

    // Instantiate the Design Under Test (DUT)
    pri_gen dut (
        .req(req),
        .pri(pri),
        .grant(grant)
    );

    // Dump waveform for GTKWave
    initial begin
        $dumpfile("pri_gen.vcd");      // File name for waveform data
        $dumpvars(0, pri_gen_tb);      // Dump all variables in this module
    end

    // Apply test inputs
    initial begin
        $display("Time\tRequests\tPriority\tGrant");
        $display("-------------------------------------");
```

```
        req = 4'b0000; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b0001; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b0010; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b0100; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b1000; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b1100; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b1010; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);
        req = 4'b0111; #10; $display("%0t\t%b\t\t%b\t\t%b", $time, req, pri, grant);

        #10;
        $finish;
    end

endmodule
```

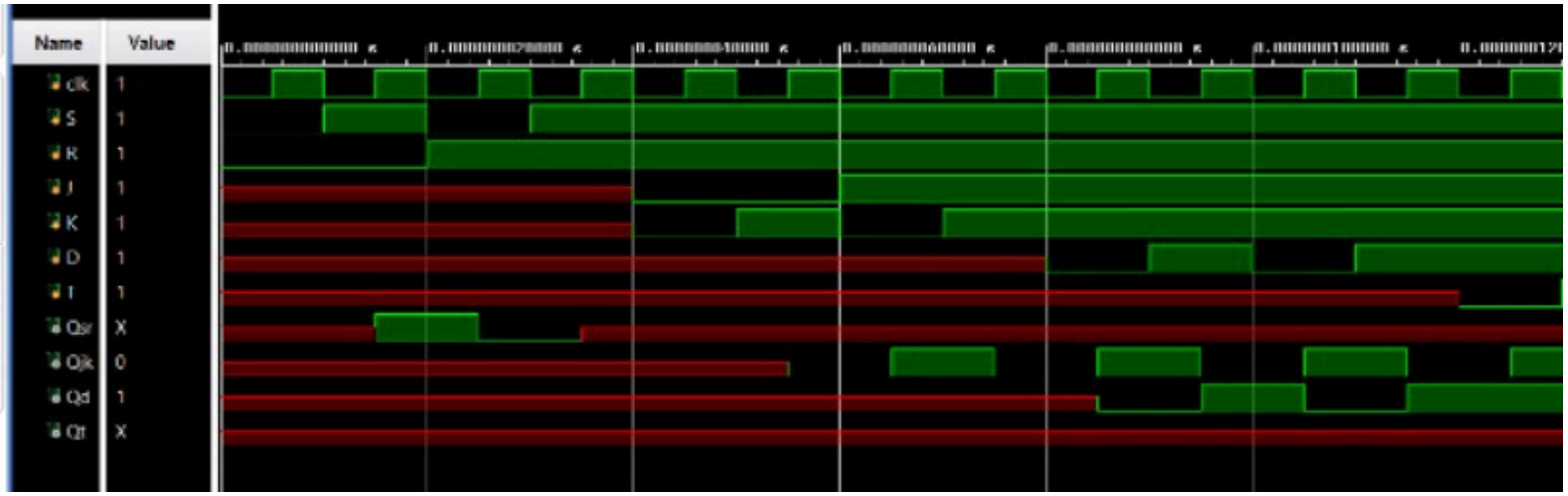


Chapter 4: Sequential Circuits

10) Flip-flops: SR, JK, T, D (behavioural)

```
module SR_JK_T;

  // Common signals
  reg clk;
  reg S, R, J, K, D, T;
  wire Qsr, Qjk, Qd, Qt;

  // Instantiate all Flip-Flops
  sr_ff sr1 (.S(S), .R(R), .clk(clk), .Q(Qsr));
  jk_ff jk1 (.J(J), .K(K), .clk(clk), .Q(Qjk));
  d_ff  d1  (.D(D), .clk(clk), .Q(Qd));
  t_ff  t1  (.T(T), .clk(clk), .Q(Qt));

  // Generate Clock
  initial begin
    clk = 0;
    forever #5 clk = ~clk;  // 10ns clock period
  end
```

```
// Test Sequences
 initial begin
   $display("\n========== Starting Simulation ==========");

   // ----- SR Flip-Flop -----
   $display("\n--- Testing SR Flip-Flop ---");
   S=0; R=0; #10;
   S=1; R=0; #10;
   S=0; R=1; #10;
   S=1; R=1; #10;

   // ----- JK Flip-Flop -----
   $display("\n--- Testing JK Flip-Flop ---");
   J=0; K=0; #10;
   J=0; K=1; #10;
   J=1; K=0; #10;
   J=1; K=1; #10;

   // ----- D Flip-Flop -----
   $display("\n--- Testing D Flip-Flop ---");
   D=0; #10;
   D=1; #10;
   D=0; #10;
   D=1; #10;

   // ----- T Flip-Flop -----
   $display("\n--- Testing T Flip-Flop ---");
   T=0; #10;
   T=1; #10;
   T=0; #10;
   T=1; #10;

   $display("\n========== Simulation Complete ==========");
   $finish;
 end
 // Monitor all outputs
 initial begin
   $monitor("Time=%0t | SR_Q=%b | JK_Q=%b | D_Q=%b | T_Q=%b", $time, Qsr, Qjk, Qd, Qt);
 end

endmodule
```



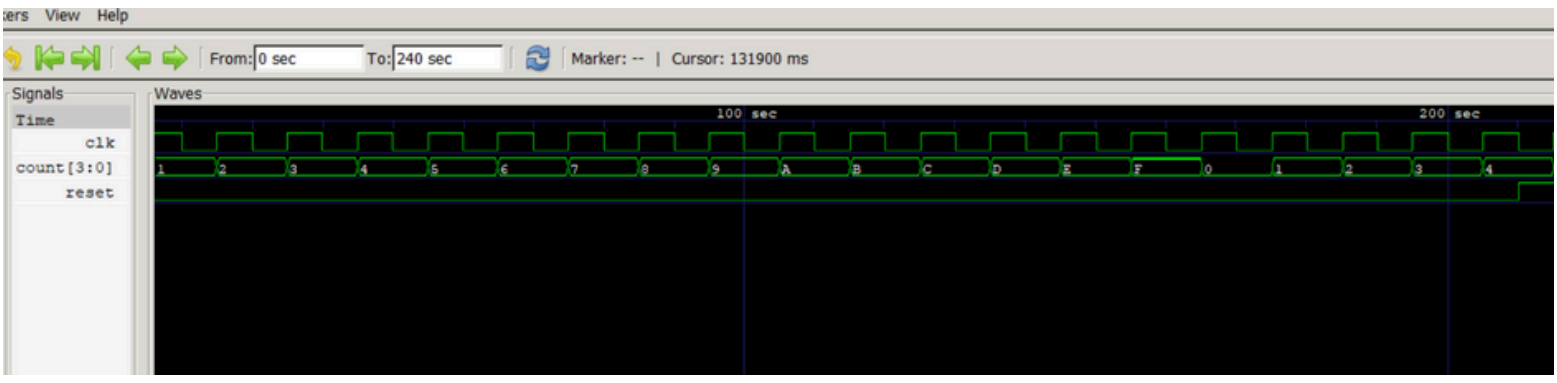
SR-JK-D-T

11) 4-bit Synchronous Counter (up counter)

```
`timescale 1ns / 1ps

// Inputs : clk - clock signal
//          reset - active high reset
// Output : count - 4-bit counter output

module sync_coun (
    input clk,            // Clock input
    input reset,          // Active high reset
    output reg [3:0] count // 4-bit counter output
);

    // Always block triggered at the rising edge of the clock
    always @(posedge clk) begin
        if (reset) begin
            count <= 4'b0000;   // Reset counter to zero
        end else begin
            count <= count + 1; // Increment count synchronously
        end
    end

endmodule
```

```
module sync_coun_tb;

    reg clk;              // Clock signal
    reg reset;            // Reset signal
    wire [3:0] count;     // Counter output

    // Instantiate the Design Under Test (DUT)
    sync_coun dut (
        .clk(clk),
        .reset(reset),
        .count(count)
    );
```

```
    // Instantiate the Design Under Test (DUT)
    sync_coun dut (
        .clk(clk),
        .reset(reset),
        .count(count)
    );

    // Generate clock signal: toggle every 5ns => 10ns period
    initial begin
        clk = 0;
        forever #5 clk = ~clk;   // Clock toggles every 5ns
    end

    // Dump waveform data for GTKWave
    initial begin
        $dumpfile("sync_coun.vcd");
        $dumpvars(0, sync_coun_tb);
    end

    // Apply reset and test the counter
    initial begin
        $display("Time\tReset\tCount");
        $display("---------------------");

        reset = 1;       // Apply reset
        #10;
        reset = 0;       // Release reset

        // Observe counting for 20 clock cycles
        repeat (20) begin
            #10;  // Wait for each clock cycle
            $display("%0t\t%b\t%b", $time, reset, count);
        end

        // Apply reset again
        reset = 1;
        #10;
        reset = 0;
        #20;

        $finish;
    end

endmodule
```

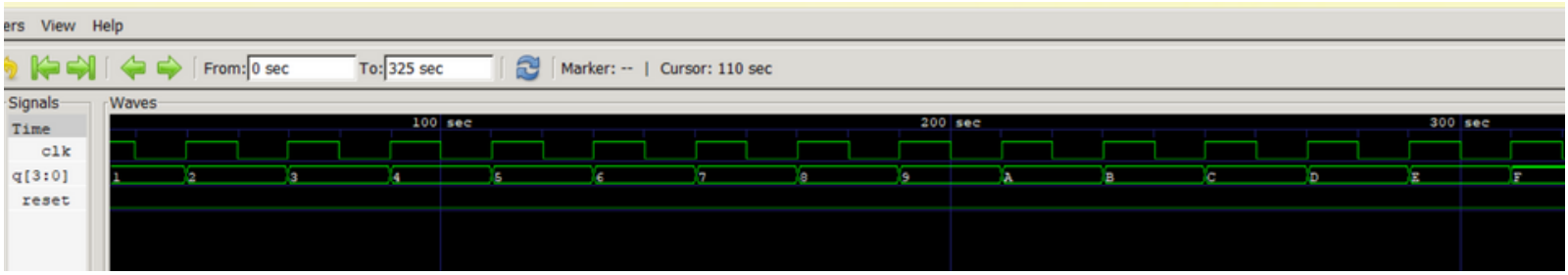


12) 4-bit Asynchronous Counter (ripple counter)

```
// 4 Bit Asynchronous Counter
// Each flip-flop toggles when the previous bit changes from 1 → 0.
// The output counts from 0000 to 1111 and then rolls over.

module async_coun(
    input clk,        // Clock input
    input reset,      // Asynchronous reset
    output [3:0] q    // 4-bit counter output
);

    // Declare four flip-flops for 4-bit output
    reg [3:0] q_reg;

    // Assign internal register to output
    assign q = q_reg;

    // Asynchronous reset and counting logic
    always @(posedge clk or posedge reset) begin
        if (reset)
            q_reg <= 4'b0000;        // Reset all bits to 0
        else begin
            q_reg[0] <= ~q_reg[0];                          // Toggle LSB every clock
            q_reg[1] <= q_reg[1] ^ q_reg[0];                // Toggle on lower bit overflow
            q_reg[2] <= q_reg[2] ^ (q_reg[0] & q_reg[1]);   // Toggle on lower 2 bits overflow
            q_reg[3] <= q_reg[3] ^ (q_reg[0] & q_reg[1] & q_reg[2]); // Toggle on lower 3 bits overflow
        end
    end

endmodule
```

```
module async_coun_tb;

    // Testbench signals
    reg clk;
    reg reset;
    wire [3:0] q;

    // Instantiate the design under test (DUT)
    async_coun dut (
        .clk(clk),
        .reset(reset),
        .q(q)
    );

    // Clock generation (period = 20ns)
    initial begin
        clk = 0;
        forever #10 clk = ~clk; // Toggle clock every 10ns
    end

    // Generate waveform dump for GTKWave
    initial begin
        $dumpfile("async_coun.vcd");   // Create VCD file for GTKWave
        $dumpvars(0, async_coun_tb);   // Dump all signals in the testbench
    end

    // Apply reset and observe counter operation
```

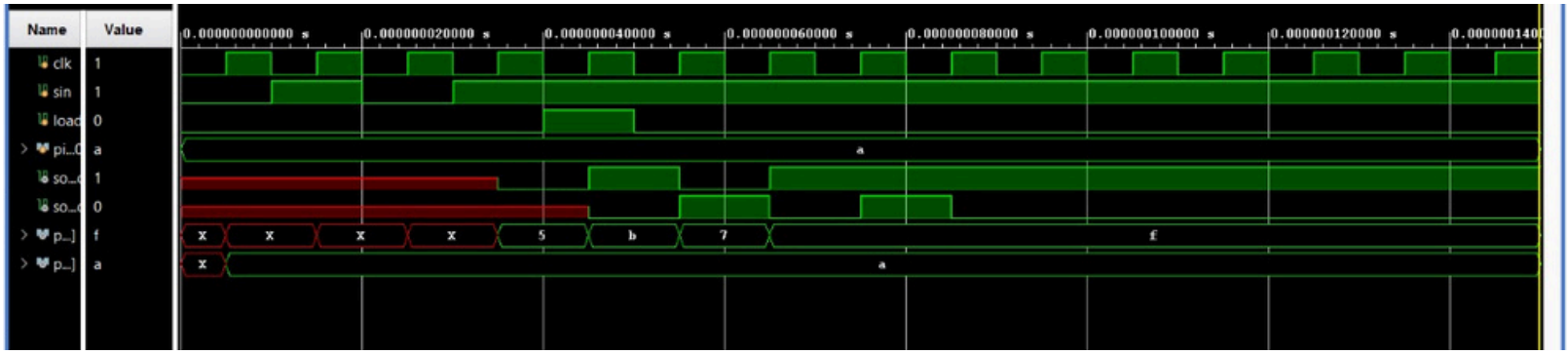
```
    initial begin
        $display("Time\tReset\tCount");
        $monitor("%0t\t%b\t%b", $time, reset, q);

        reset = 1;
        #25;                // Keep reset high for a short time
        reset = 0;

        #300;               // Run simulation for 300ns
        $finish;            // End simulation
    end

endmodule
```



Shift Register

```
// SISO Shift Register
module SISO(input clk, input sin, output sout);
  reg [3:0] shift;
  always @(posedge clk) shift <= {shift[2:0], sin};
  assign sout = shift[3];
endmodule

// SIPO Shift Register
module SIPO(input clk, input sin, output [3:0] pout);
  reg [3:0] shift;
  always @(posedge clk) shift <= {shift[2:0], sin};
  assign pout = shift;
endmodule

// PIPO Shift Register
module PIPO(input clk, input [3:0] pin, output reg [3:0] pout);
  always @(posedge clk) pout <= pin;
endmodule

// PISO Shift Register
module PISO(input clk, input load, input [3:0] pin, output reg sout);
  reg [3:0] shift;
  always @(posedge clk)
  begin
    if (load)
      shift <= pin;
    else
      shift <= {shift[2:0], 1'b0};
    sout <= shift[3];
  end
endmodule
```

```
module tb_shift_regs;
  reg clk, sin, load;
  reg [3:0] pin;
  wire sout_siso, sout_piso;
  wire [3:0] pout_sipo, pout_pipo;

  SISO siso(.clk(clk), .sin(sin), .sout(sout_siso));
  SIPO sipo(.clk(clk), .sin(sin), .pout(pout_sipo));
  PIPO pipo(.clk(clk), .pin(pin), .pout(pout_pipo));
  PISO piso(.clk(clk), .load(load), .pin(pin), .sout(sout_piso));

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

  initial begin
    sin=0; load=0; pin=4'b1010;
    #10 sin=1; #10 sin=0; #10 sin=1; #10 sin=1;
    load=1; #10 load=0;
    #100;
    $finish;
  end
endmodule
```



Mini Project Report

On

"Traffic Light Controller Design using Verilog"

Submitted for the partial fulfilment
of

B. Tech.

in

ELECTRONICS AND COMMUNICATION ENGINEERING

SUBMITTED BY:

Student Name :- Ankit Kumar Yadav
(Roll No.) :- 2300270310039

Vth Sem - IIIrd Year
Section- ECE-1

SUBMITTED TO:

Dr. Tukur Gupta
(ASSO. PROF.)
ECE DEPT.

Ajay Kumar Garg Engineering College, Ghaziabad
27th Km Milestone, Delhi-Meerut Expressway, P.O. Adhyatmik Nagar, Ghaziabad-201015

Acknowledgement

I want to express my sincere gratitude and thanks to **Prof.** (**Dr.) Neelesh Kumar Gupta (HoD, ECE Department), Ajay Kumar Garg Engineering College, Ghaziabad** for granting me permission for my industrial training in the field of **" Traffic Light Controller Design using Verilog".**

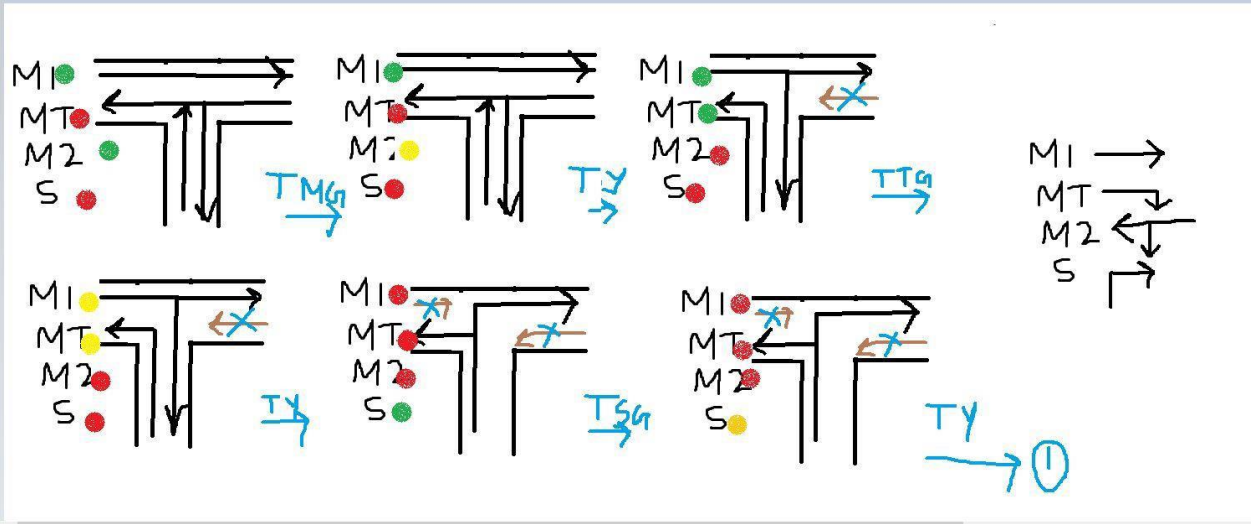
I express my sincere thanks to **Asso. Prof.** (**Dr.) Tukur Gupta** for his cooperative attitude and consistence guidance, due to which I was able to complete my training successfully.

Finally, I pay my thankful regard and gratitude to the team members for their valuable help, support and guidance.

Student Name :- Ankit Kumar Yadav
Roll No. :- 2300270310039
Vth Sem - III Year
Section- ECE-1

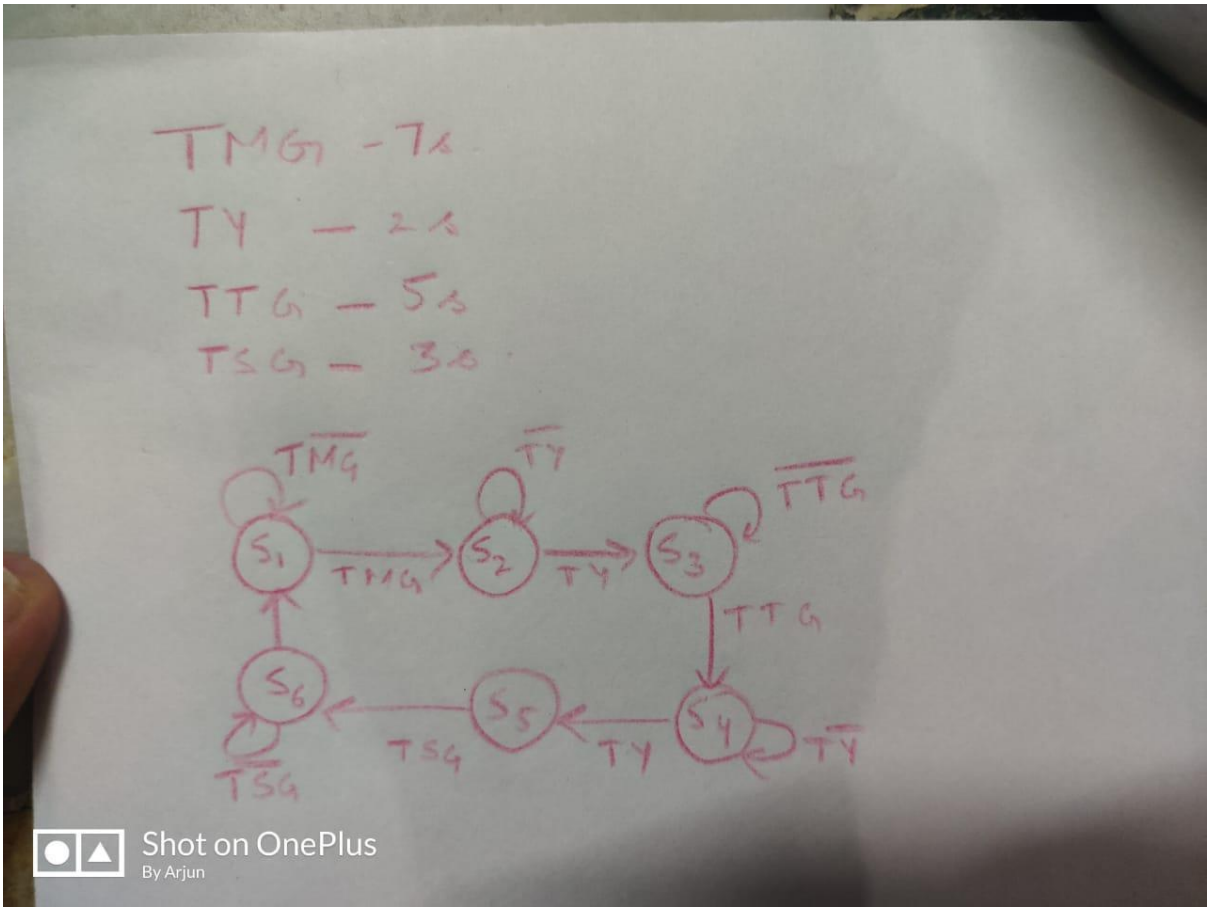
PROBLEM STATEMENT :-

The aim of the project is to design a traffic controller for a T-intersection. Let's understand the problem statement through the image given below.

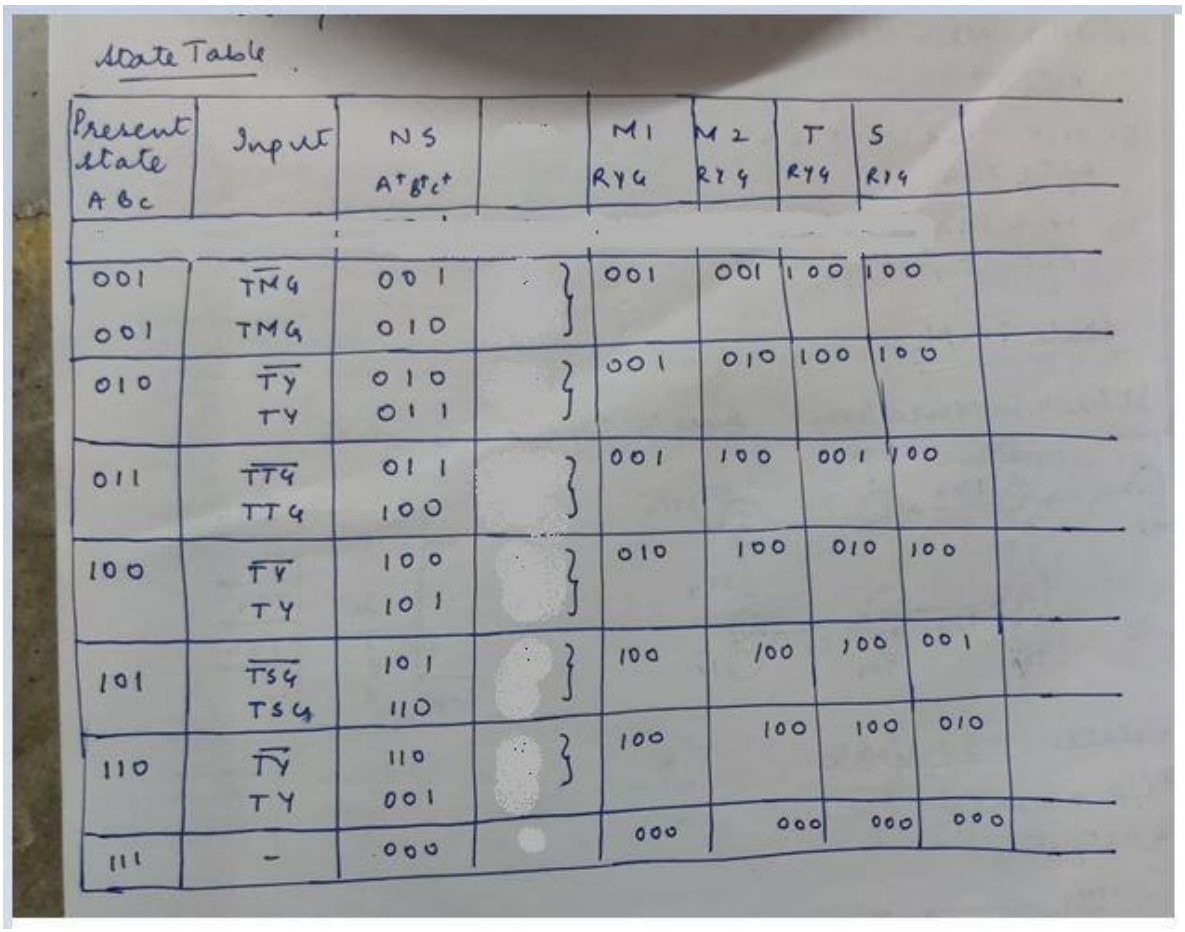


The six cases present here eventually turn to the six states .

This is the state diagram:



From the state diagram we for the state table:



State Table

Present state ABC	Input	NS $A^+B^+C^+$		M1 RYG	M2 RYG	T RYG	S RYG
001	$\overline{TMG}$	001	}	001	001	100	100
001	TMG	010					
010	$\overline{TY}$	010	}	001	010	100	100
	TY	011					
011	$\overline{TTG}$	011	}	001	100	001	100
	TTG	100					
100	$\overline{TY}$	100	}	010	100	010	100
	TY	101					
101	$\overline{TSG}$	101	}	100	100	100	001
	TSG	110					
110	$\overline{TY}$	110	}	100	100	100	010
	TY	001					
111	-	000		000	000	000	000

VERILOG CODE :-

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module Name: Traffic_Light_Controller


//////////////////////////////////////////////////////////////////////////////////
module Traffic_Light_Controller(
   input clk,rst,
   output reg [2:0]light_M1,
   output reg [2:0]light_S,
   output reg [2:0]light_MT,
   output reg [2:0]light_M2
   );
   parameter  S1=0, S2=1, S3 =2, S4=3, S5=4,S6=5;
   reg [3:0]count;
   reg[2:0] ps;
   parameter  sec7=7,sec5=5,sec2=2,sec3=3;
   always@(posedge clk or posedge rst)
      begin
      if(rst==1)
      begin
      ps<=S1;
      count<=0;
      end
      else
```

```
case(ps)
   S1: if(count<sec7)
         begin
         ps<=S1;
         count<=count+1;
         end
       else
         begin
         ps<=S2;
         count<=0;
         end
   S2: if(count<sec2)
         begin
         ps<=S2;
         count<=count+1;
         end

       else
         begin
         ps<=S3;
         count<=0;
         end
   S3: if(count<sec5)
         begin
         ps<=S3;
         count<=count+1;
```

```
        end

    else
        begin
        ps<=S4;
        count<=0;
        end
S4:if(count<sec2)
        begin
        ps<=S4;
        count<=count+1;
        end

    else
        begin
        ps<=S5;
        count<=0;
        end
S5:if(count<sec3)
        begin
        ps<=S5;
        count<=count+1;
        end

    else
        begin
```

```
            ps<=S6;
            count<=0;
            end

    S6:if(count<sec2)
            begin
            ps<=S6;
            count<=count+1;
            end

        else
            begin
            ps<=S1;
            count<=0;
            end
    default: ps<=S1;
    endcase
end

always@(ps)
begin

    case(ps)

        S1:
        begin
```

```
  light_M1<=3'b001;
  light_M2<=3'b001;
  light_MT<=3'b100;
  light_S<=3'b100;
end
S2:
begin
  light_M1<=3'b001;
  light_M2<=3'b010;
  light_MT<=3'b100;
  light_S<=3'b100;
end
S3:
begin
  light_M1<=3'b001;
  light_M2<=3'b100;
  light_MT<=3'b001;
  light_S<=3'b100;
end
S4:
begin
  light_M1<=3'b010;
  light_M2<=3'b100;
  light_MT<=3'b010;
  light_S<=3'b100;
end
```

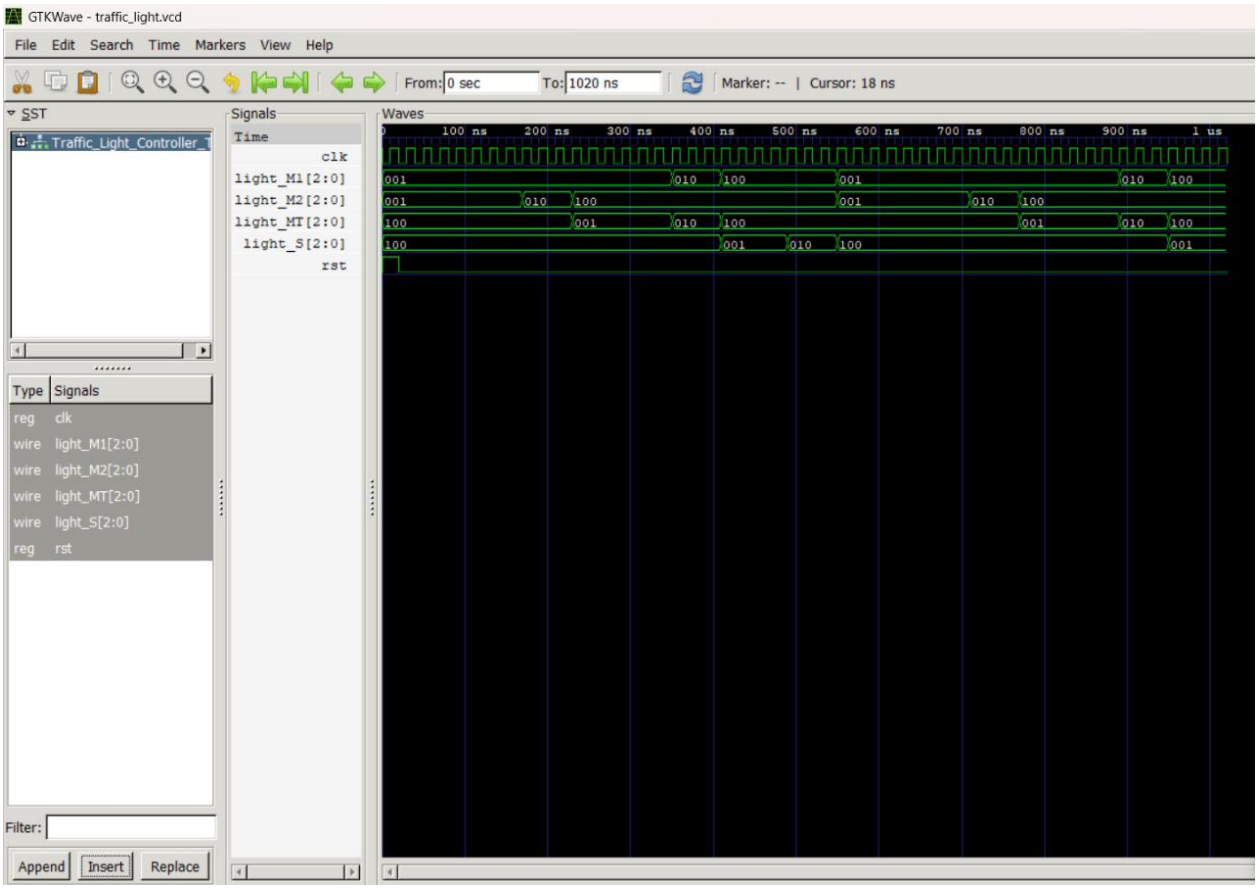
```
            S5:
            begin
              light_M1<=3'b100;
              light_M2<=3'b100;
              light_MT<=3'b100;
              light_S<=3'b001;
            end
            S6:
            begin
              light_M1<=3'b100;
              light_M2<=3'b100;
              light_MT<=3'b100;
              light_S<=3'b010;
            end
            default:
            begin
              light_M1<=3'b000;
              light_M2<=3'b000;
              light_MT<=3'b000;
              light_S<=3'b000;
            end
            endcase
        end


endmodule
```

TESTBENCH :-

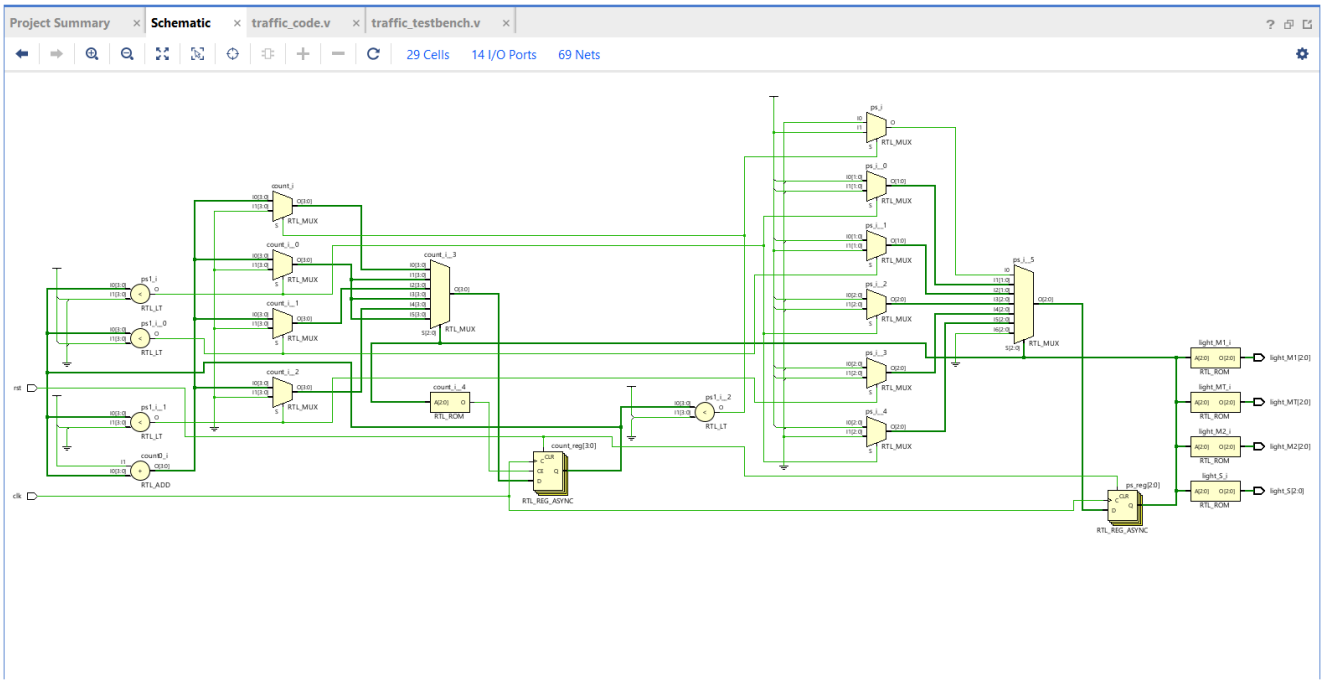
```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Module Name: Traffic_Light_Controller_TB

//////////////////////////////////////////////////////////////////////////////////


module Traffic_Light_Controller_TB;

reg clk,rst;

wire [2:0]light_M1;

wire [2:0]light_S;

wire [2:0]light_MT;

wire [2:0]light_M2;

Traffic_Light_Controller dut(.clk(clk) , .rst(rst) , .light_M1(light_M1) , .light_S(light_S)
,.light_M2(light_M2),.light_MT(light_MT)   );

initial

begin

   clk=1'b0;

   forever #(1000000000/2) clk=~clk;

end

//    initial

//    $stop;//to add ps

initial

begin

   rst=0;

   #1000000000;
```

```
    rst=1;
    #1000000000;
    rst=0;
    #(1000000000*200);
    $finish;
    end
endmodule
```

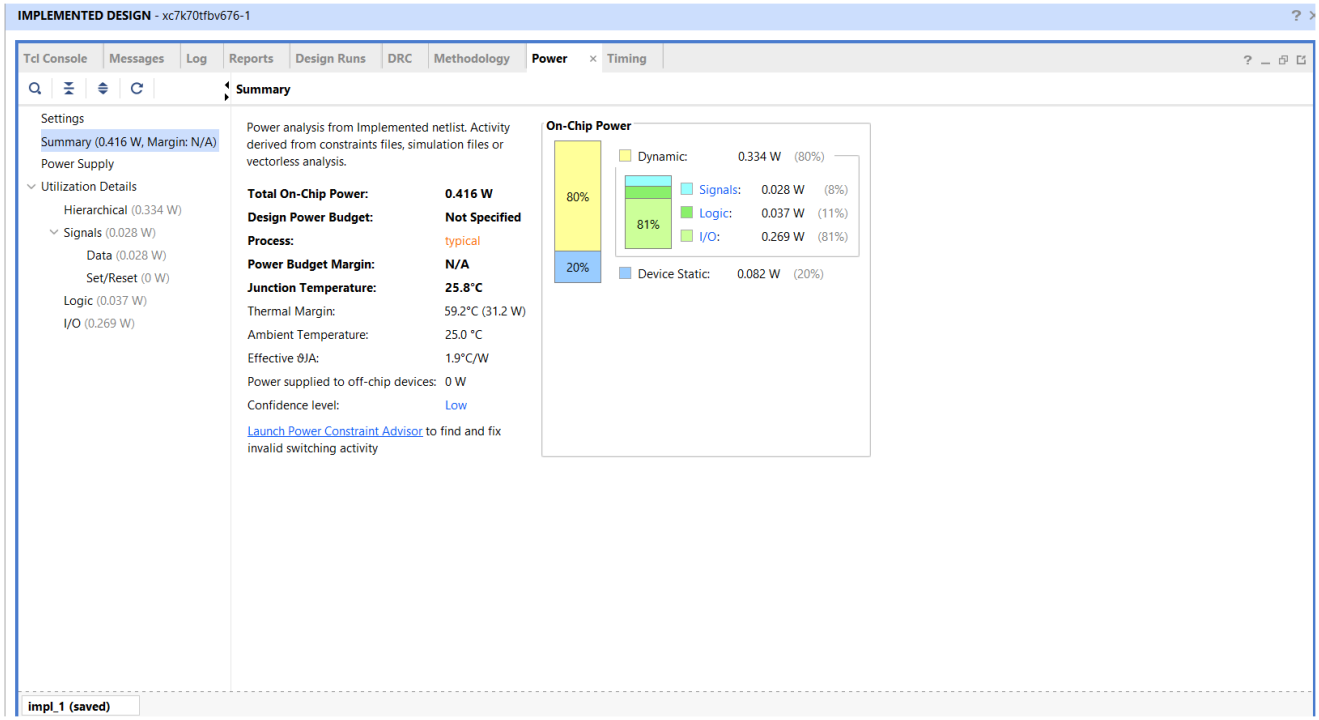
SIMULATED WAVEFORM :-



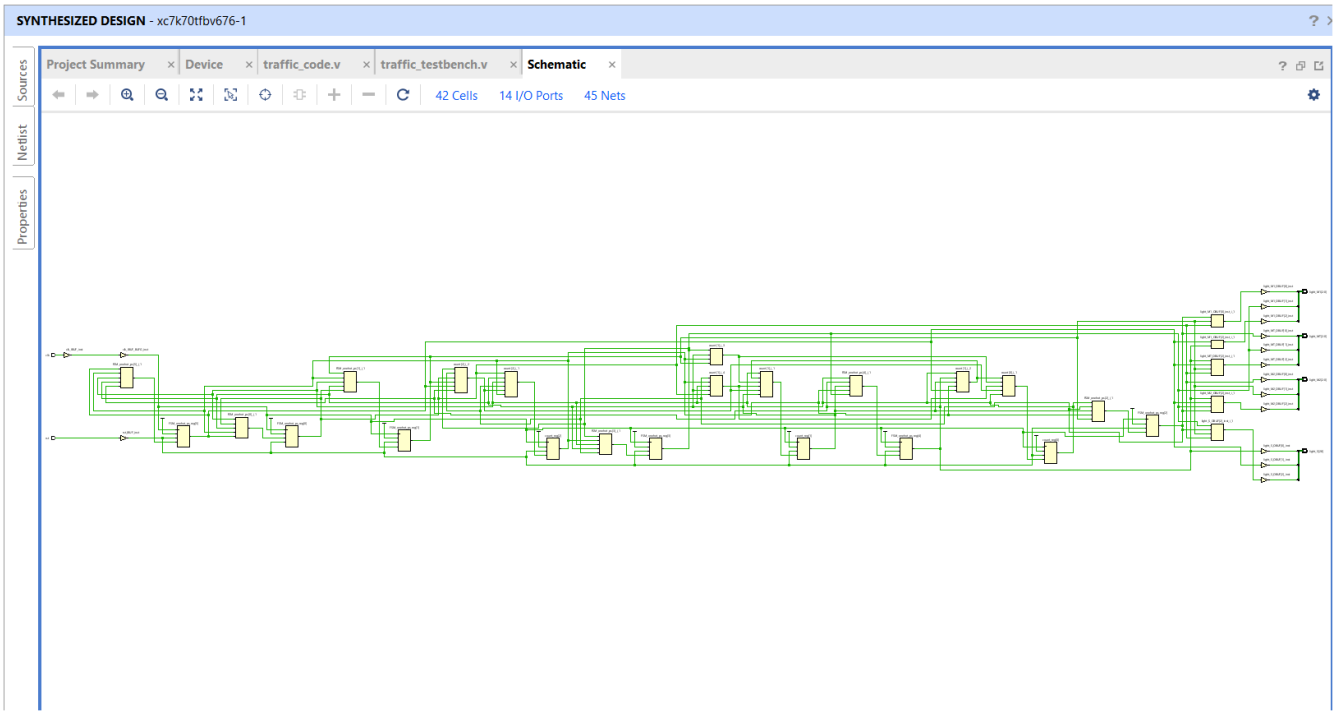
RTL-SCHEMATIC :-



POWER REPORT :-



SCHEMATIC AFTER SYNTHESIS :-



DEVICE LAYOUT AFTER IMPLEMENTATION :-