TRAFFIC LIGHT CONTROLLER USING FINITE STATE MACHINE

Layout & Verification

EECT / CE 6325 – VLSI DESIGN

Instructor

Prof. Carl Sechen

Submitted by	Ganesh Manjunath Net ID: dal542146	Aryan Verma Net ID: dal820508

Date: 12/09/2025

Table of Content

S.No.	Content	Page
1	Introduction	1-3
2	Verilog Code & Testbench	4-10
3	Layout	11-12
4	DRC & LVS Check	13-14
5	Trade off & Challenges	15

INTRODUCTION

This project design implements a finite state machine (FSM) for a traffic light controller that manages a four-way intersection, alternating traffic flow between North-South (NS) and East-West (EW) directions to ensure safety and efficiency. The system utilizes synchronous flip-flops for state registration and a 6-bit counter for timing control, cycling through four encoded states: NS Green (lasting 30 clock cycles), NS Yellow (5 cycles), EW Green (30 cycles), and EW Yellow (5 cycles). Inputs include a clock signal (clk) for synchronization, a synchronous reset (rst) that initializes the FSM to the NS Green state, and an unused sensor input for future extensions like adaptive traffic detection. Outputs consist of control signals for the red, yellow, and green lights in both NS and EW directions, with combinational logic ensuring no simultaneous green lights across perpendicular paths to prevent conflicts. Implemented in behavioral Verilog, the design is scalable, with the RTL schematic illustrating key components such as multiplexers for state selection and output decoding, comparators for timing thresholds, an adder for counter increment, and registers for state and counter storage, all interconnected to form a robust, synchronous digital circuit.

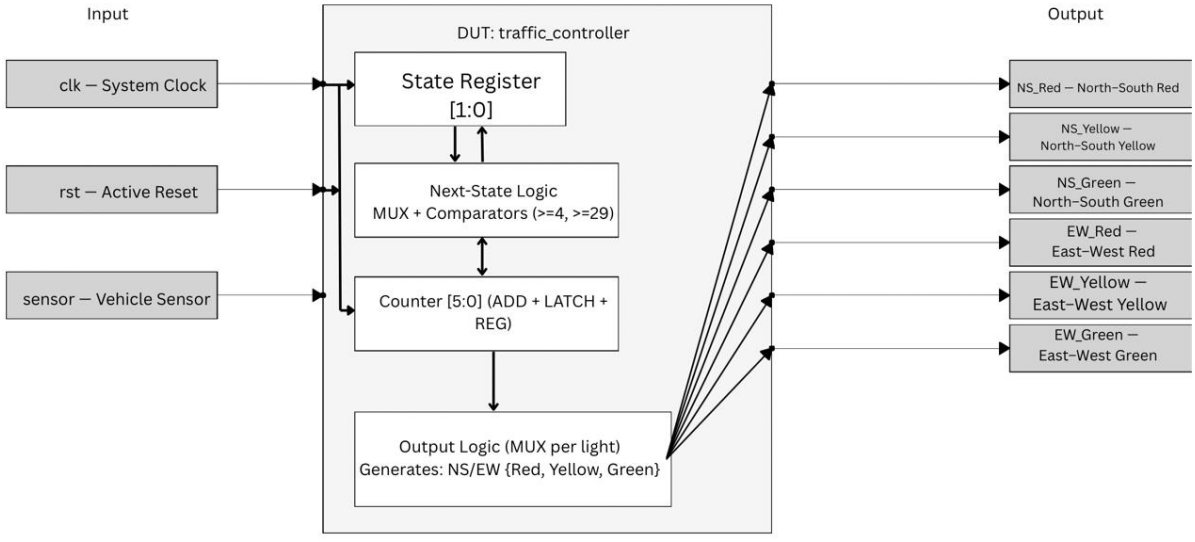


Figure 1: Traffic Controller Design Block

The block diagram of the traffic light controller begins with three key inputs: the system clock, the reset signal, and the sensor. The clock is the driving force that synchronizes all operations inside the circuit, ensuring the state changes and counters update in a regular rhythm. The reset

signal is used to initialize the system, forcing the controller into its starting condition, which is North–South green. The sensor input is included for scalability and represents adaptive detection, though in the base version of the design it is not actively used.

At the heart of the diagram lies the finite state machine, represented by the state register, the counter, the comparators, and the next-state logic. The state register stores the current state of the traffic lights, whether it is North–South green, North–South yellow, East–West green, or East–West yellow. A six-bit counter runs alongside the state register, increasing on every clock cycle to measure how long each state should last. The comparators connected to the counter check if the required time has been reached for example, whether the green period has lasted thirty cycles or the yellow period has lasted five cycles. This timing information is fed into the next-state logic, implemented with multiplexers, which decides the upcoming state and ensures a smooth sequence from green to yellow and then to the opposite road's green.

Finally, the output logic in the block diagram converts the current state into actual control signals for the traffic lights. Multiplexers associated with each lamp decide which ones should be active, ensuring that only the correct lights are on at any given time. For instance, when the system is in the North–South green state, the North–South green lamp and the East–West red lamp is activated, while all others remain off. In the East–West yellow state, the East–West yellow and North–South red lights are turned on. The result of this arrangement is six output signals red, yellow, and green for both directions that coordinate the safe and timed flow of traffic at the intersection.

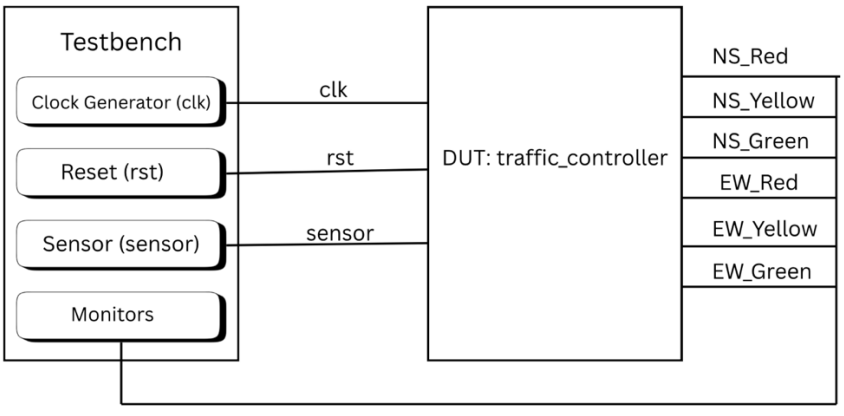


Figure 1: Test Bench Block along with the Design Block

This block diagram shows how the testbench communicates with the design under test (DUT), which in this case is the traffic light controller module. The testbench acts as a virtual environment that supplies all necessary inputs to the DUT and monitors its outputs to verify correct behavior. On the left side, the testbench has four key components: a clock generator, a reset generator, a sensor input, and monitors.

The clock generator produces the clk signal that drives the sequential logic inside the DUT, such as the state register and counter. The reset block controls the rst signal, which initializes the DUT into a known starting condition (North–South green). The sensor input is included for completeness and is connected to the DUT, even though it may not actively affect the base design. These three signals (clk, rst, and sensor) are passed from the testbench into the DUT as inputs.

On the right-hand side of the DUT, the outputs are the six traffic light signals: NS_Red, NS_Yellow, NS_Green, EW_Red, EW_Yellow, and EW_Green. These outputs are fed back to the monitors in the testbench, which record and display the signals during simulation. In this way, the testbench provides stimulus to the traffic controller and observes its responses, ensuring that the design cycles through the expected light patterns correctly.

This project discusses designing a traffic light controller using Verilog HDL, based on the Finite state machine approach. The design not only elaborates the systematic state transitions of a basic controller but also integrates features like parameterization, pulse-width modulation (PWM) for light intensity control, and scalability through different modules. The work emphasizes hardware mapping & synthesizability, providing insight into practical VLSI design flow and digital system implementation.

Verilog Code

```
module traffic_light_controller #(
    parameter integer GREEN_TIME  = 3000,
    parameter integer YELLOW_TIME = 500,
    parameter integer CWIDTH      = 256,
    parameter integer UNIQUE_ID   = 0
)(
    input  wire clk,
    input  wire rst,
    input  wire sensor,
    output reg  NS_Red,
    output reg  NS_Yellow,
    output reg  NS_Green,
    output reg  EW_Red,
    output reg  EW_Yellow,
    output reg  EW_Green
);
    localparam [3:0] S_NS_G = 4'b0001,
                     S_NS_Y = 4'b0010,
                     S_EW_G = 4'b0100,
                     S_EW_Y = 4'b1000;
    reg [3:0] state, next_state;
    reg [CWIDTH-1:0] counter, next_counter;
    reg [7:0] unique_reg;
    always @(posedge clk) begin
        if (rst) unique_reg <= 8'b00000000;
        else unique_reg <= unique_reg + UNIQUE_ID;
    end
```

```
    always @(posedge clk) begin
        if (rst) begin
            state   <= S_NS_G;
            counter <= {CWIDTH{1'b0}};
        end else begin
            state   <= next_state;
            counter <= next_counter;
        end
    end
    always @* begin
        next_state   = state;
        next_counter = counter + {{(CWIDTH-1){1'b0}},1'b1};
        case (state)
            S_NS_G: if (counter >= GREEN_TIME-1)  begin next_state=S_NS_Y;
next_counter={CWIDTH{1'b0}}; end
            S_NS_Y: if (counter >= YELLOW_TIME-1) begin next_state=S_EW_G;
next_counter={CWIDTH{1'b0}}; end
            S_EW_G: if (counter >= GREEN_TIME-1)  begin next_state=S_EW_Y;
next_counter={CWIDTH{1'b0}}; end
            S_EW_Y: if (counter >= YELLOW_TIME-1) begin next_state=S_NS_G;
next_counter={CWIDTH{1'b0}}; end
            default: begin next_state=S_NS_G; next_counter={CWIDTH{1'b0}}; end
        endcase
    end
    always @* begin
        NS_Red=0; NS_Yellow=0; NS_Green=0;
        EW_Red=0; EW_Yellow=0; EW_Green=0;
        case (state)
            S_NS_G: begin NS_Green=1;  EW_Red=1;  end
            S_NS_Y: begin NS_Yellow=1; EW_Red=1;  end
```

```
            S_EW_G: begin EW_Green=1;  NS_Red=1;  end
            S_EW_Y: begin EW_Yellow=1; NS_Red=1;  end
            default: begin NS_Green=1; EW_Red=1;  end
        endcase
    end
endmodule
module pwm #(
    parameter integer BITS = 256
)(
    input  wire clk,
    input  wire rst,
    input  wire [BITS-1:0] duty,
    output reg  y
);
    reg [BITS-1:0] cnt;
    always @(posedge clk) begin
        if (rst) cnt <= {BITS{1'b0}};
        else     cnt <= cnt + {{(BITS-1){1'b0}},1'b1};
    end
    always @* y = (cnt < duty);
endmodule
module ctl_with_pwm #(
    parameter integer GREEN_TIME  = 3000,
    parameter integer YELLOW_TIME = 500,
    parameter integer CWIDTH      = 256,
    parameter integer PWM_BITS    = 256,
    parameter integer UNIQUE_ID   = 0
)(
    input  wire clk,
```

```
    input  wire rst,
    input  wire sensor,
    output wire NS_Red, NS_Yellow, NS_Green,
    output wire EW_Red, EW_Yellow, EW_Green
);
    wire nsr, nsy, nsg, ewr, ewy, ewg;
    traffic_light_controller #(
        .GREEN_TIME(GREEN_TIME), .YELLOW_TIME(YELLOW_TIME),
.CWIDTH(CWIDTH), .UNIQUE_ID(UNIQUE_ID)
    ) u_ctl (
        .clk(clk), .rst(rst), .sensor(sensor),
        .NS_Red(nsr), .NS_Yellow(nsy), .NS_Green(nsg),
        .EW_Red(ewr), .EW_Yellow(ewy), .EW_Green(ewg)
    );
    localparam [PWM_BITS-1:0] FULL = {PWM_BITS{1'b1}};
    pwm #(.BITS(PWM_BITS)) p0(.clk(clk), .rst(rst), .duty(nsr ? FULL :
{PWM_BITS{1'b0}}), .y(NS_Red));
    pwm #(.BITS(PWM_BITS)) p1(.clk(clk), .rst(rst), .duty(nsy ? FULL :
{PWM_BITS{1'b0}}), .y(NS_Yellow));
    pwm #(.BITS(PWM_BITS)) p2(.clk(clk), .rst(rst), .duty(nsg ? FULL :
{PWM_BITS{1'b0}}), .y(NS_Green));
    pwm #(.BITS(PWM_BITS)) p3(.clk(clk), .rst(rst), .duty(ewr ? FULL :
{PWM_BITS{1'b0}}), .y(EW_Red));
    pwm #(.BITS(PWM_BITS)) p4(.clk(clk), .rst(rst), .duty(ewy ? FULL :
{PWM_BITS{1'b0}}), .y(EW_Yellow));
    pwm #(.BITS(PWM_BITS)) p5(.clk(clk), .rst(rst), .duty(ewg ? FULL :
{PWM_BITS{1'b0}}), .y(EW_Green));
endmodule
module big_system #(
    parameter integer N             = 200,
    parameter integer GREEN_TIME    = 3000,
```

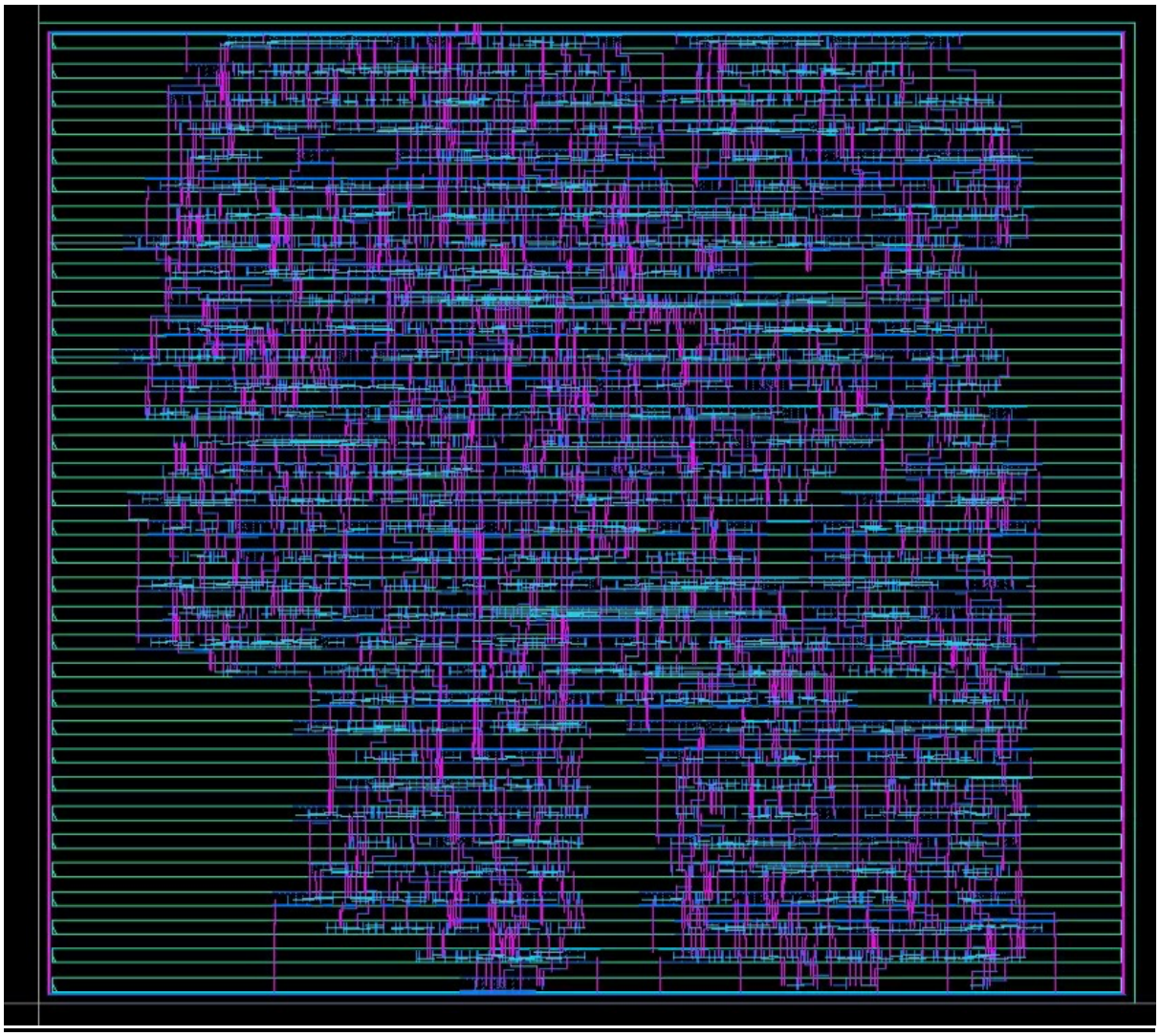
```
    parameter integer YELLOW_TIME   = 500,
    parameter integer CWIDTH        = 256,
    parameter integer PWM_BITS      = 256
)(
    input  wire               clk,
    input  wire               rst,
    input  wire [N-1:0]         sensor,
    output wire [N-1:0]         NS_Red, NS_Yellow, NS_Green,
    output wire [N-1:0]         EW_Red, EW_Yellow, EW_Green
);
    genvar i;
    generate
        for (i=0; i<N; i=i+1) begin : G
            ctl_with_pwm #(
                .GREEN_TIME(GREEN_TIME), .YELLOW_TIME(YELLOW_TIME),
                .CWIDTH(CWIDTH), .PWM_BITS(PWM_BITS),
                .UNIQUE_ID(i)
            ) u (
                .clk(clk), .rst(rst), .sensor(sensor[i]),
                .NS_Red(NS_Red[i]), .NS_Yellow(NS_Yellow[i]), .NS_Green(NS_Green[i]),
                .EW_Red(EW_Red[i]), .EW_Yellow(EW_Yellow[i]), .EW_Green(EW_Green[i])
            );
        end
    endgenerate
endmodule
```

Testbench

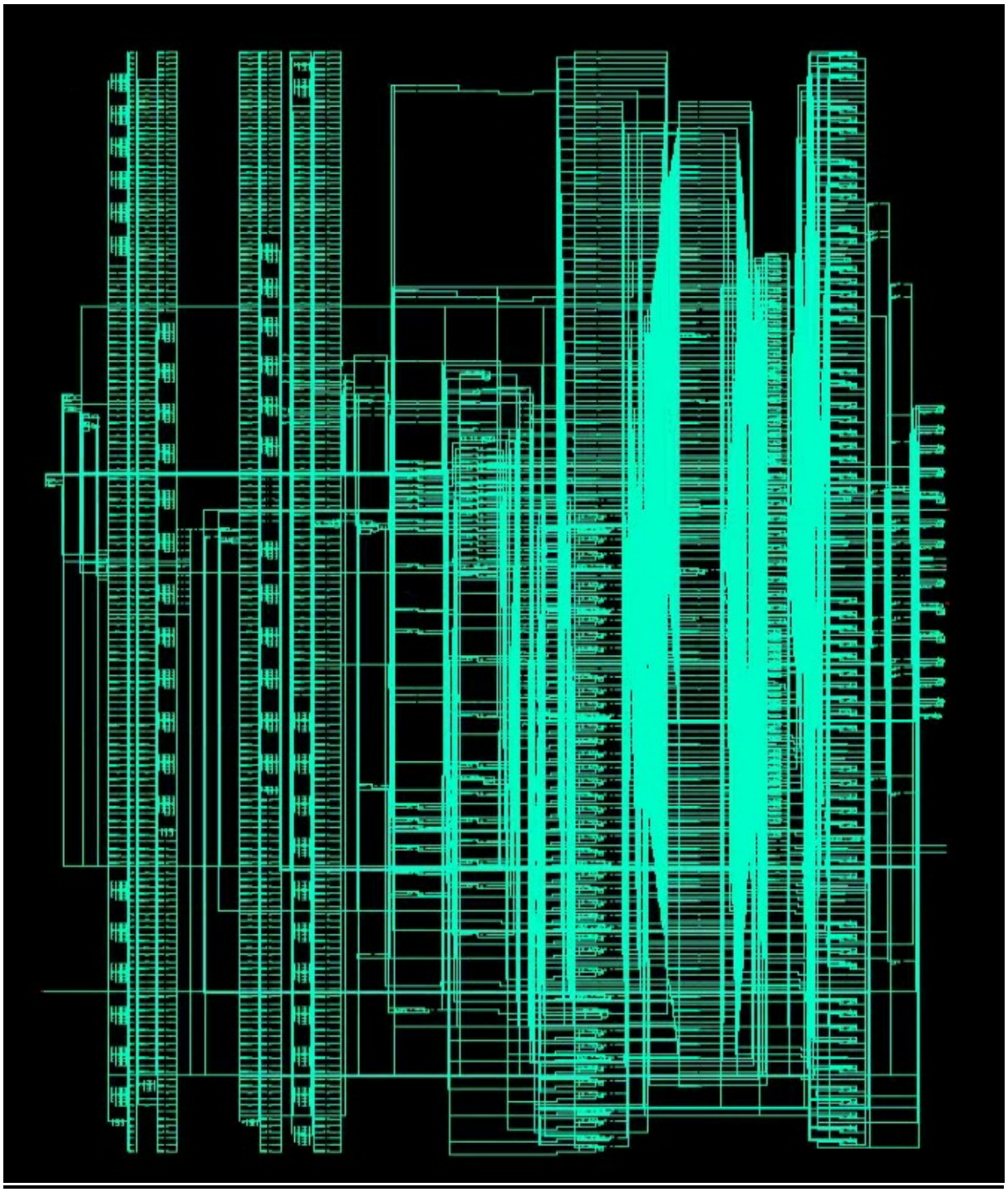
```
`timescale 1ns/1ps
module traffic_light_controller_tb;
  localparam integer CLK_PER_NS   = 10;   // 100 MHz
  localparam integer RESET_CYCLES = 10;   // hold reset this many cycles
  localparam integer RUN_CYCLES   = 4000; // main stimulus length
  // ---- DUT I/O ----
  reg  clk, rst, sensor;
  wire NS_Red, NS_Yellow, NS_Green;
  wire EW_Red, EW_Yellow, EW_Green;
  // Instantiate DUT (matches your gate netlist top)
  traffic_light_controller dut (
    .clk(clk), .rst(rst), .sensor(sensor),
    .NS_Red(NS_Red), .NS_Yellow(NS_Yellow), .NS_Green(NS_Green),
    .EW_Red(EW_Red), .EW_Yellow(EW_Yellow), .EW_Green(EW_Green)
  );
  // ---- Clock ----
  initial begin
    clk = 1'b0;
    forever #(CLK_PER_NS/2) clk = ~clk;
  end
  // ---- Optional waveform dump (VCD for non-ModelSim tools) ----
`ifdef VCD
  initial begin
    $dumpfile("traffic_light_controller_tb.vcd");
    $dumpvars(0, traffic_light_controller_tb);
  end
`endif
  // ---- Stimulus + heartbeat + basic checks ----
```

```
  integer i;
  initial begin
    $timeformat(-9,0," ns",10);
    $display("** TB start @ %0t **", $realtime);
    rst    = 1'b1;
    sensor = 1'b0;
    // reset
    for (i = 0; i < RESET_CYCLES; i = i + 1) @(posedge clk);
    rst = 1'b0;
    // main run
    for (i = 0; i < RUN_CYCLES; i = i + 1) begin
      @(posedge clk);
      // deterministic activity on 'sensor' (avoids $random quirks)
      sensor <= sensor ^ NS_Green ^ EW_Green ^ NS_Yellow ^ EW_Yellow;
      // heartbeat every 500 cycles so you see progress in Transcript
      if ((i % 500) == 0)
        $display("%0t: heartbeat (i=%0d)  NS[G,Y,R]=%0b%0b%0b
EW[G,Y,R]=%0b%0b%0b",
                 $realtime, i, NS_Green, NS_Yellow, NS_Red, EW_Green, EW_Yellow, EW_Red);
      // simple safety check: never both greens high in the same cycle
      if (!rst && (NS_Green & EW_Green)) begin
        $display("ERROR @ %0t: Both NS and EW GREEN!", $realtime);
        $stop;
      end
    end
    $display("** TB finish @ %0t **", $realtime);
    $finish;
  end
endmodule
```

LAYOUT

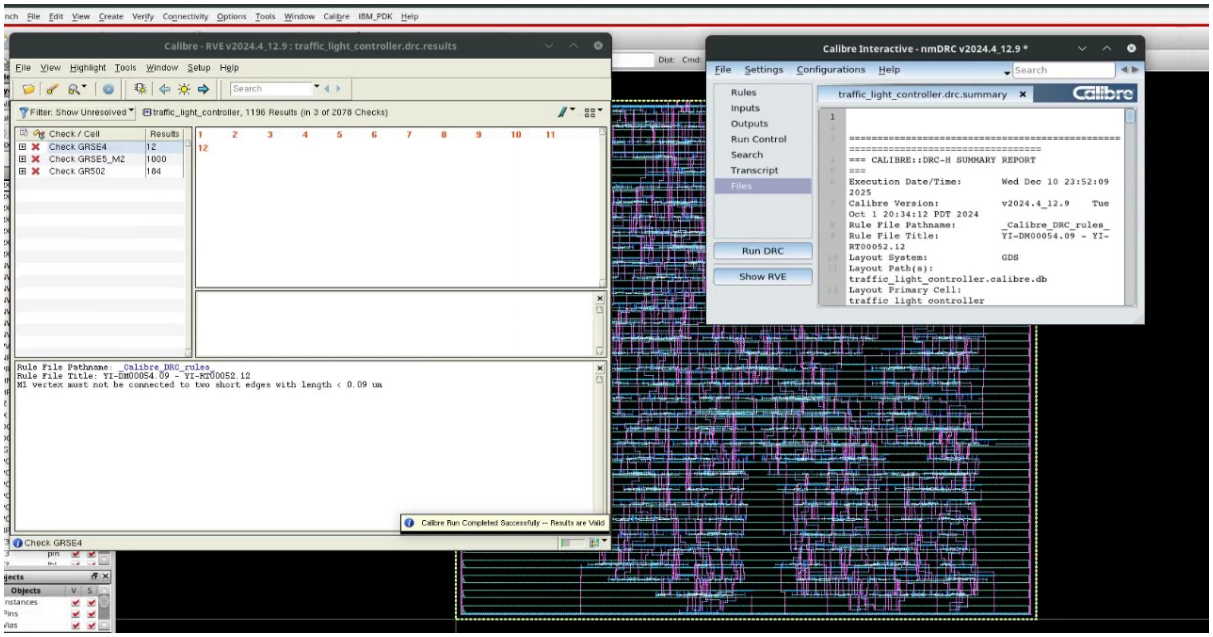


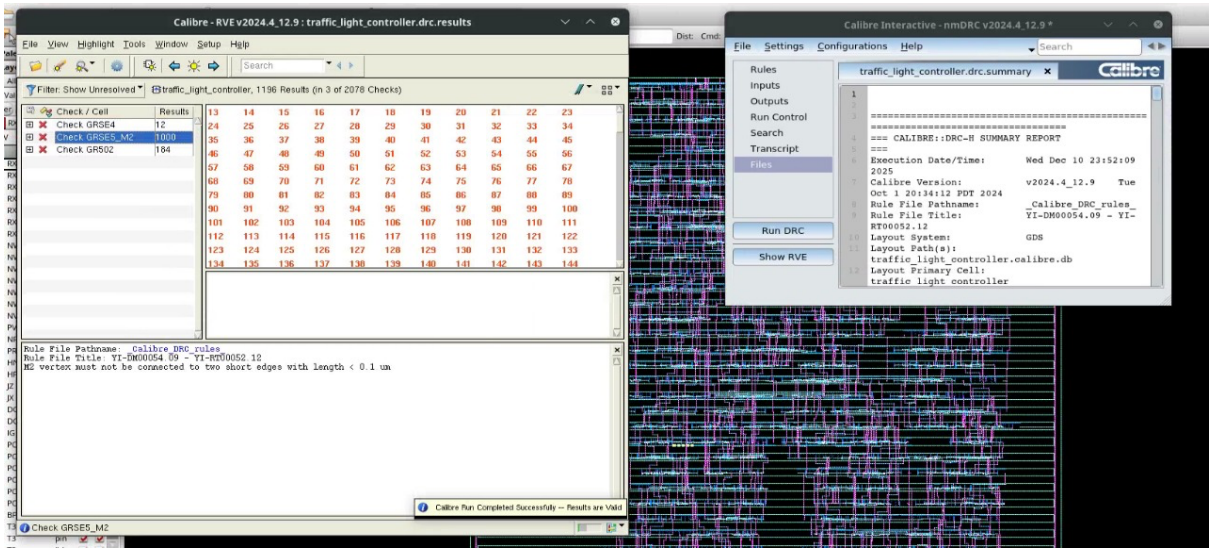
SCHEMATIC

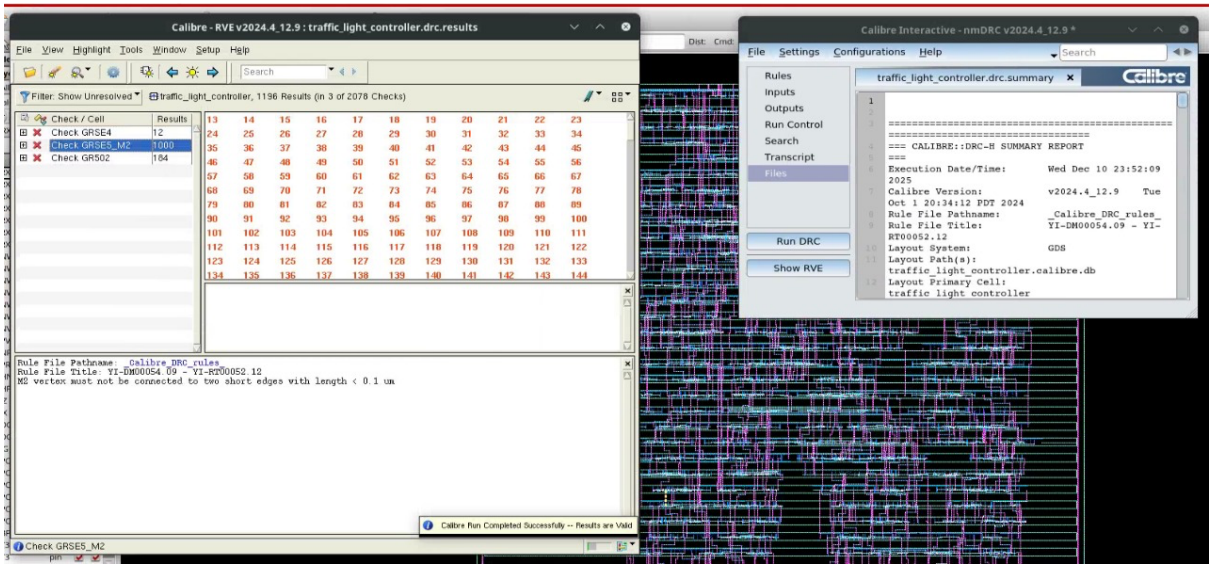


DRC & LVS Check

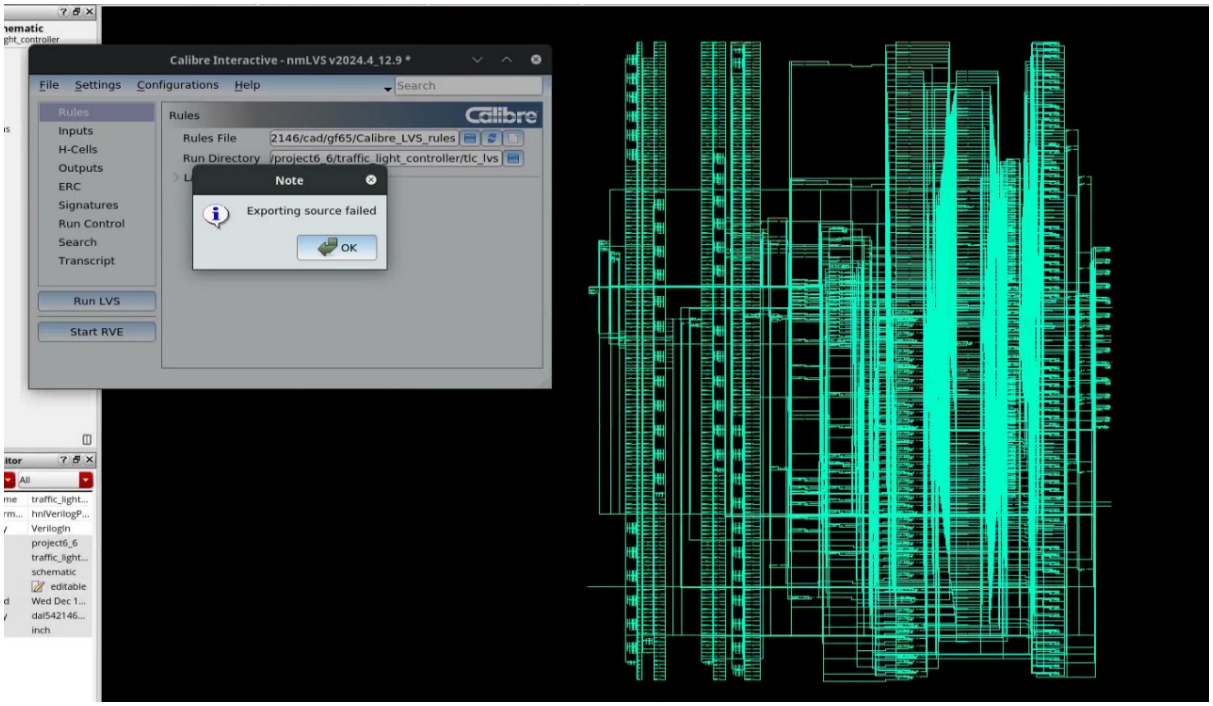
DRC CHECK-






Calibre - RVE v2024.4_12.9 : traffic_light_controller.drc.results
Calibre Interactive - nmDRC v2024.4_12.9 *
Run DRC
Show RVE
traffic_light_controller.drc.summary
Rule File Pathname: _Calibre_DRC_rules_
Rule File Title: YI-DM00054.09 - YI-RT00052.12
M3 vertex must not be connected to two short edges with length < 0.1 um
Calibre Run Completed Successfully -- Results are Valid

LVS CHECK-


Calibre Interactive - nmLVS v2024.4_12.9 *
Rules File
2146/cad/gf65/Calibre_LVS_rules
Run Directory
/project6_6/traffic_light_controller/tlc_lvs
Note
Exporting source failed
OK
Run LVS
Start RVE

TRADE-OFFS AND DESIGN CHALLENGES

1. Time vs. Area Trade-Off in Timing Parameters

Challenge: The traffic light timing is defined by the GREEN_TIME (3000 clock cycles) and YELLOW_TIME (500 clock cycles) parameters. The counter size, CWIDTH, must be large enough to count to the maximum time (e.g., a 12-bit counter for 3000).

Trade-off: A larger CWIDTH increases the counter's complexity, potentially increasing the chip area and power consumption of the design. However, a smaller CWIDTH would limit the flexibility of the timing parameters, constraining the maximum time for the traffic phases. The current parameter of CWIDTH=256 bits is an extremely large, albeit highly parameterized, size that likely results in a significant area overhead for the counter hardware compared to a minimal implementation.

2. Verilog Code Scalability vs. Base Design Complexity

Challenge: The report emphasizes design scalability, including the incorporation of a pwm module for Pulse-Width Modulation (light intensity control) and a big_system module to instantiate multiple controllers.

Trade-off: While these modules (pwm and big_system) provide great scalability and future-proofing (such as dynamic light intensity or handling multiple intersections), they significantly increase the complexity of the Verilog code , the required tool chain runtime for synthesis and layout, and the overall gate count of the final integrated circuit, even if the base traffic_light_controller FSM is simple.

3. Layout Verification (DRC/LVS) Errors

Challenge: The Design Rule Check (DRC) screens show numerous errors (1198 Results in 2070 Checks) across multiple rules (e.g., Check GRSE4, Check GRSE5_M2). Furthermore, the Layout Versus Schematic (LVS) check resulted in a failure, with an "Exporting layout failed" message.

Design Challenge: Failing DRC/LVS is a critical issue in VLSI design, as it indicates the physical layout does not match the logical schematic and/or violates manufacturing rules. This immediately prevents the design from being fabricated. The challenge is in the debugging and correction process to resolve all layout-to-schematic mismatches and physical design rule violations. This process is often time-consuming and requires iterative refinement of the placement and routing.