Smart Adaptive Traffic Light Controller

Prepared by: Lavanya Tamgade

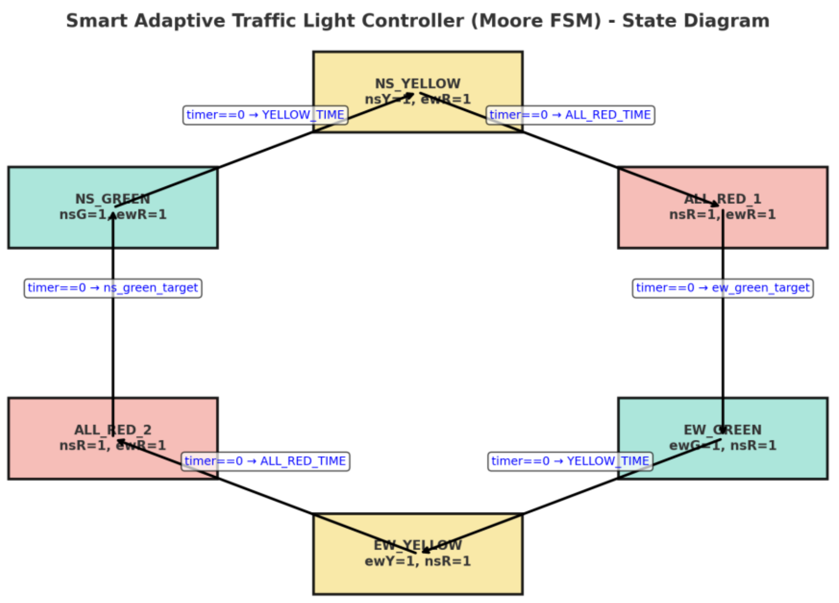
Introduction

Modern cities face heavy traffic congestion that requires intelligent management systems. Conventional traffic light controllers use fixed-time schedules, which may not be efficient under varying traffic conditions. This project implements a Smart Adaptive Traffic Light Controller using Verilog HDL. The controller dynamically adjusts the green light duration based on traffic density inputs from each direction, ensuring optimal flow and reducing waiting times. The design is implemented as a Moore FSM with adaptive timing logic.

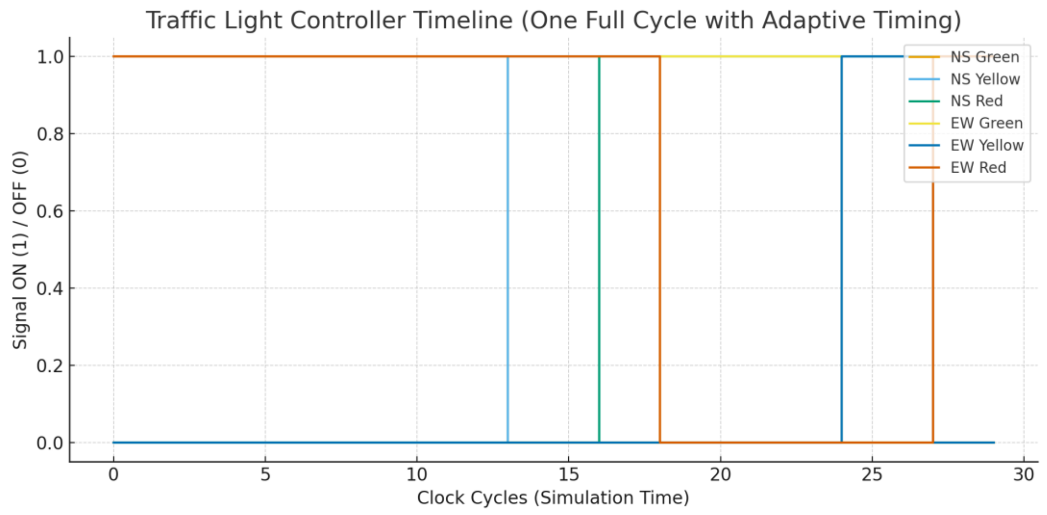
Key Highlights

- Objective: Design and simulate an adaptive traffic light controller in Verilog.
- FSM States: NS_GREEN, NS_YELLOW, ALL_RED_1, EW_GREEN, EW_YELLOW, ALL_RED_2.
- Key Parameters: BASE_GREEN=6, DENSITY_STEP=4, YELLOW_TIME=3, ALL_RED_TIME=2.
- Adaptive Feature: Green time = BASE_GREEN + (DENSITY_STEP × density_input).
- Safety: Mutual exclusion of greens; yellow & all-red intervals enforced.

FSM State Diagram



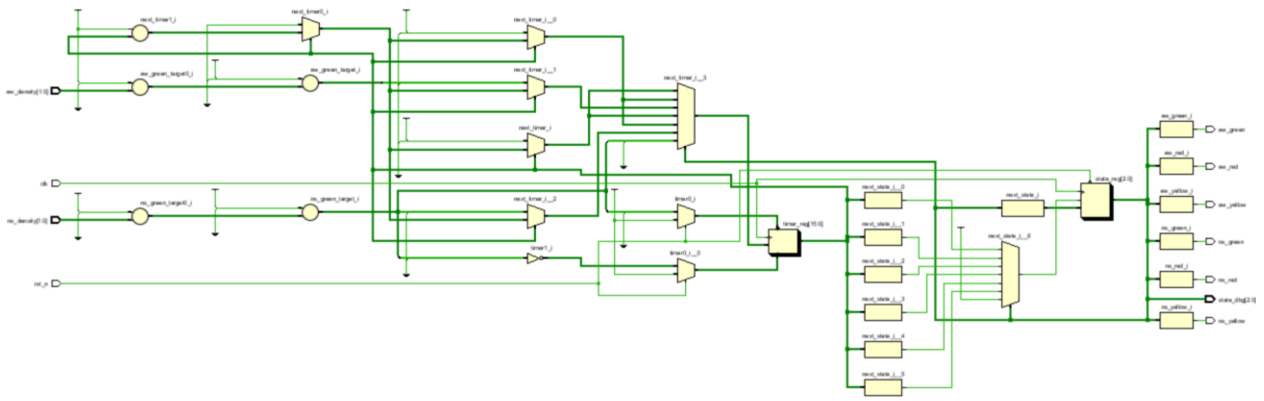
Timing (One Full Cycle with Adaptive Green)



Parameters & Adaptation

Green targets are computed as: BASE_GREEN + DENSITY_STEP × density. Example: For NS density=2 and EW density=0, the green targets become NS=14 and EW=6 cycles. Yellow time is fixed at 3 cycles, while all-red intervals last 2 cycles to ensure safety.

Synthesized Schematic (High-level)



Appendix A — Verilog RTL

```
`timescale 1ns / 1ps

// ------------------------------------------------------------

// Smart Adaptive Traffic Light Controller (NS <-> EW)

// - Adaptive GREEN time based on 2-bit density inputs

// - Simple 5-state FSM: NS_G, NS_Y, ALL_RED1, EW_G, EW_Y, ALL_RED2

// - Timing is in "clock cycles" for easy simulation

// ------------------------------------------------------------

module smart_traffic_controller #(

   parameter integer BASE_GREEN   = 6,   // base green duration

   parameter integer DENSITY_STEP = 4,   // extra cycles per density level

   parameter integer YELLOW_TIME  = 3,   // yellow duration

   parameter integer ALL_RED_TIME = 2    // all-red safety duration

)(

   input  wire        clk,

   input  wire        rst_n,         // active-low reset

   input  wire [1:0]  ns_density,      // 00=low, 01=med, 10=high, 11=very high

   input  wire [1:0]  ew_density,      // same encoding for EW

   output reg         ns_red,

   output reg         ns_yellow,

   output reg         ns_green,

   output reg         ew_red,

   output reg         ew_yellow,

   output reg         ew_green,
   output reg  [2:0]  state_dbg         // for waveform/debug
);

   // FSM states
   localparam [2:0]
       S_NS_GREEN  = 3'd0,
       S_NS_YELLOW = 3'd1,
       S_ALL_RED_1 = 3'd2,
       S_EW_GREEN  = 3'd3,
       S_EW_YELLOW = 3'd4,
       S_ALL_RED_2 = 3'd5;

   reg [2:0]  state, next_state;
   reg [15:0] timer, next_timer;

   // computed green targets
   wire [15:0] ns_green_target = BASE_GREEN + DENSITY_STEP * ns_density;
   wire [15:0] ew_green_target = BASE_GREEN + DENSITY_STEP * ew_density;

   // timer tick + state register
   always @(posedge clk or negedge rst_n) begin
       if (!rst_n) begin
           state <= S_NS_GREEN;
           timer <= ns_green_target;
   end else begin
       state <= next_state;
       timer <= next_timer;
   end
end


// next-state & timer logic
always @* begin
   // default hold
   next_state = state;
   next_timer = (timer == 0) ? 0 : (timer - 1);


   case (state)
       S_NS_GREEN: begin
           if (timer == 0) begin
               next_state = S_NS_YELLOW;
               next_timer = YELLOW_TIME;
           end
       end
       S_NS_YELLOW: begin
           if (timer == 0) begin
               next_state = S_ALL_RED_1;
               next_timer = ALL_RED_TIME;
           end
```

```
end
S_ALL_RED_1: begin
    if (timer == 0) begin
        next_state = S_EW_GREEN;
        next_timer = ew_green_target;
    end
end
S_EW_GREEN: begin
    if (timer == 0) begin
        next_state = S_EW_YELLOW;
        next_timer = YELLOW_TIME;
    end
end
S_EW_YELLOW: begin
    if (timer == 0) begin
        next_state = S_ALL_RED_2;
        next_timer = ALL_RED_TIME;
    end
end
S_ALL_RED_2: begin
    if (timer == 0) begin
        next_state = S_NS_GREEN;
        next_timer = ns_green_target;
    end
```

```
        end
        default: begin
            next_state = S_NS_GREEN;
            next_timer = ns_green_target;
        end
    endcase
end

// outputs (Moore)
always @* begin
    // defaults
    ns_red    = 1'b0; ns_yellow = 1'b0; ns_green = 1'b0;
    ew_red    = 1'b0; ew_yellow = 1'b0; ew_green = 1'b0;

    case (state)
        S_NS_GREEN:  begin ns_green = 1'b1; ew_red = 1'b1; end
        S_NS_YELLOW: begin ns_yellow= 1'b1; ew_red = 1'b1; end
        S_ALL_RED_1: begin ns_red   = 1'b1; ew_red = 1'b1; end
        S_EW_GREEN:  begin ew_green = 1'b1; ns_red = 1'b1; end
        S_EW_YELLOW: begin ew_yellow= 1'b1; ns_red = 1'b1; end
        S_ALL_RED_2: begin ns_red   = 1'b1; ew_red = 1'b1; end
        default:     begin ns_red   = 1'b1; ew_red = 1'b1; end
    endcase
end
```

```
  // expose state for debug

  always @* state_dbg = state;


endmodule
```

Appendix B — Testbench

```
`timescale 1ns/1ps

module tb_smart_traffic_controller;


  reg clk;

  reg rst_n;

  reg [1:0] ns_density;

  reg [1:0] ew_density;


  wire ns_red, ns_yellow, ns_green;

  wire ew_red, ew_yellow, ew_green;

  wire [2:0] state_dbg;


  // DUT

  smart_traffic_controller #(

   .BASE_GREEN  (6),
 .DENSITY_STEP(4),
 .YELLOW_TIME (3),
 .ALL_RED_TIME(2)
) dut (
 .clk(clk),
 .rst_n(rst_n),
 .ns_density(ns_density),
 .ew_density(ew_density),
 .ns_red(ns_red),
 .ns_yellow(ns_yellow),
 .ns_green(ns_green),
 .ew_red(ew_red),
 .ew_yellow(ew_yellow),
 .ew_green(ew_green),
 .state_dbg(state_dbg)
);

// 100 MHz clock
initial clk = 1'b0;
always #5 clk = ~clk;

// Simple progress print (no %s)
always @(posedge clk) begin
```

```
    $display("t=%0t ns | state=%0d | NS(RYG)=%0d%0d%0d
EW(RYG)=%0d%0d%0d | dens NS=%0d EW=%0d",

      $time,

      state_dbg,

      ns_red, ns_yellow, ns_green,

      ew_red, ew_yellow, ew_green,

      ns_density, ew_density

    );

  end



  // Assertions (keep them)

  always @(posedge clk) begin

    if (rst_n) begin

      if (ns_green && ew_green) begin

        $display("ASSERT FAIL @%0t: Both directions GREEN!", $time);

        $fatal;

      end

      if (ns_yellow && ew_yellow) begin

        $display("ASSERT FAIL @%0t: Both directions YELLOW!", $time);

        $fatal;

      end

    end

  end
```

```
// Stimulus

initial begin

  // Ensure known values before first time step

  rst_n      = 1'b0;

  ns_density = 2'd2; // NS high

  ew_density = 2'd0; // EW low


  // Startup banner (helps spot if TB is actually running)

  $display("TB start @ %0t", $time);


  repeat (3) @(posedge clk);

  rst_n = 1'b1;


  repeat (40) @(posedge clk);


  $display("\n-- Changing densities: NS low, EW very high --\n");

  ns_density = 2'd0;

  ew_density = 2'd3;

  repeat (50) @(posedge clk);


  $display("\n-- Changing densities: both medium --\n");

  ns_density = 2'd1;

  ew_density = 2'd1;

  repeat (40) @(posedge clk);
```

```
    $display("\nSimulation finished OK.");

    $finish;

  end



endmodule
```

Conclusion

Simulation and analysis confirm that the Smart Adaptive Traffic Light Controller meets its objectives. The FSM transitions are correct, mutual exclusion of green signals is maintained, and adaptive timing based on density inputs is verified. This design can be synthesized on FPGA and extended with real sensor inputs for real-world deployment.