Adaptive Traffic Light Controller with Emergency Override

Naireet Choudhury

Vellore Institute of Technology

Contents

1 Problem Statement

Urban intersections face increasing congestion, especially during peak hours. Fixed-time traffic lights are inefficient at handling dynamic traffic flow, resulting in long waiting times and elevated accident risks. There is a need for an adaptive traffic signal controller that responds to real-time traffic conditions and gives priority to emergency vehicles for safer and smoother operations.

2 Introduction

This project proposes an Adaptive Traffic Light Controller designed using digital logic in Verilog HDL. The controller adjusts signal timings based on real-time input from traffic sensors and also includes an Emergency Override mode to prioritize emergency vehicles. The system aims to balance efficient traffic flow with safety and quick emergency response at urban intersections.

3 Literature Review

Several adaptive systems have been proposed using microcontrollers and machine learning for traffic control. Traditional controllers are usually timer-based and do not adapt to real-time demand, which leads to inefficient use of green light time. This project leans on digital logic and FSM (Finite State Machine) design principles for adaptive control, a technique proven to be effective for real-time hardware implementations in embedded systems and FPGAs. One relevant study highlights the use of improved adaptive signal control (IASC) which considers multiple downstream road conditions rather than a single one, boosting the system's responsiveness. Additionally, with the integration of Vehicular Ad Hoc Networks (VANETs), systems like VR-IASC enable traffic light agents to communicate with vehicle agents, allowing for rerouting strategies that prevent congestion. This research reported a significant reduction in average travel time by up to 42.26% in simulated environments, demonstrating the strong potential of adaptive traffic systems in real-world applications.

4 Softwares Used

- ModelSim for Verilog Code
- TinkerCad for Arduino visualisation

5 Implementation

The traffic light control system is architected using a Finite State Machine (FSM), which has been implemented using the Verilog Hardware Description Language (HDL). This system actively monitors inputs from traffic sensors to make immediate decisions about assigning green or yellow light phases to specific lanes. Furthermore, it incorporates an Emergency Override input; activating this input mandates the system to prioritize clearing the intersection pathway for emergency vehicles.

The distinct operational states defined within the FSM are:

- **NS_G:** North-South traffic direction has a Green light.
- **NS_Y:** North-South traffic direction has a Yellow light.
- **EW_G:** East-West traffic direction has a Green light.
- **EW_Y:** East-West traffic direction has a Yellow light.
- **EMG:** Emergency Mode is active, typically forcing all directions to red or a specific sequence to clear the intersection.

To manage the light timings, the controller utilizes an internal state timer. This timer enforces both minimum required durations and maximum allowable durations for the green and yellow light signals in each relevant state. The transitions between these states are not fixed but occur dynamically based on conditions met by the traffic sensor inputs and the status of the emergency flags.

For the purpose of facilitating future hardware-level prototyping efforts, the Verilog FSM design can be translated into equivalent C++ code. This C++ language version is suitable for execution on a microcontroller platform, with Arduino being a prime example. Deploying the C++ code on Arduino enables straightforward simulation using platforms like Tinkercad. Within such a simulation environment, the physical traffic sensors can be represented by digital inputs on the Arduino, while LEDs connected to output pins can visually represent the state of the traffic lights. Similarly, the Emergency Override function can be mimicked using a connected button or switch, allowing for a physical and visually interactive demonstration of the control logic's behavior in a hands-on setting.

This conversion process fundamentally involves mapping the state machine's operational logic into a C++ program structure. This typically entails using a main loop containing conditional logic, such as if-else chains

or switch-case statements, to manage the states and transitions. C++ variables are employed to keep track of the current state and to simulate the timing requirements previously handled by Verilog constructs. Pursuing this translation approach opens avenues for real-world testing possibilities and provides a method for easier visualization, proving especially useful for individuals learning to bridge the gap between HDL-based design and embedded C programming paradigms.

6 Proof of Concept

6.1 Verilog Code

C:/altera/13.0sp1/Traffic_Controller.v (/tb_traffic_controller/uut) - Default *

```
Ln#
1   //`default_nettype none
2
3   module Traffic_Controller (
4       input wire clk,              // Clock signal
5       input wire reset,            // Asynchronous reset (active high)
6       input wire emergency,        // Emergency vehicle signal
7       input wire [3:0] traffic_sensors, // [3]:EW2, [2]:EW1, [1]:NS2, [0]:NS1
8       output reg [3:0] light,      // [3]:EW_Y, [2]:EW_G, [1]:NS_Y, [0]:NS_G
9       output wire [7:0] state_timer_out // Expose internal timer for simulation/debug
10  );
11
12      // Parameters for state durations (in clock cycles)
13      parameter NS_GREEN_CYCLES = 100; // Duration for NS Green light
14      parameter EW_GREEN_CYCLES = 60;  // Duration for EW Green light
15      parameter YELLOW_CYCLES   = 20;  // Duration for Yellow light (both directions)
16      parameter EMERGENCY_WAIT  = 5;   // Short wait during emergency transition if needed
17
18      // State definition using parameters
19      parameter [2:0] INIT            = 3'b000;
20      parameter [2:0] NS_GREEN        = 3'b001;
21      parameter [2:0] NS_YELLOW       = 3'b010;
22      parameter [2:0] EW_GREEN        = 3'b011;
23      parameter [2:0] EW_YELLOW       = 3'b100;
24      parameter [2:0] EMERGENCY_TRANS = 3'b101; // Intermediate state for emergency
25      parameter [2:0] EMERGENCY_GREEN = 3'b110; // State when emergency vehicle has priority (NS Green)
26      // Note: Using 3 bits allows for up to 8 states.
27
28      reg [2:0] current_state, next_state; // State registers
29
30      // Internal timer for state durations
31      reg [7:0] state_timer; // Timer up to 256 cycles
32
33      // Sensor logic (combinational)
34      wire ns_sensor_active = traffic_sensors[1] | traffic_sensors[0];
35      wire ew_sensor_active = traffic_sensors[3] | traffic_sensors[2];
36
```

Figure 1: Verilog code

C:/altera/13.0sp1/Traffic_Controller.v (/tb_traffic_controller/uut) - Default *

```
37      // State Register Logic (Clocked) - Using standard Verilog always block
38      always @(posedge clk or posedge reset) begin
39          if (reset) begin
40              current_state <= INIT;
41              state_timer <= 0; // Initialize timer on reset
42          end else begin
43              current_state <= next_state;
44              // Timer countdown logic - reload on state change or expiry
45              if (next_state != current_state) begin // Reset timer on state change
46                  case (next_state)
47                      NS_GREEN:        state_timer <= NS_GREEN_CYCLES -1; // Load duration (adjust for immediate decrement)
48                      EW_GREEN:        state_timer <= EW_GREEN_CYCLES -1;
49                      NS_YELLOW:       state_timer <= YELLOW_CYCLES -1;
50                      EW_YELLOW:       state_timer <= YELLOW_CYCLES -1;
51                      EMERGENCY_TRANS: state_timer <= EMERGENCY_WAIT -1; // Short delay if needed
52                      EMERGENCY_GREEN: state_timer <= 1; // Keep timer active but short (or could be longer)
53                      INIT:            state_timer <= 1; // Minimal time in init
54                      default:         state_timer <= 1; // Default case
55                  endcase
56              // Decrement timer if not changing state AND timer > 0
57              end else if (state_timer != 0) begin
58                   state_timer <= state_timer - 1;
59              // If timer hits 0 and state doesn't change (e.g. NS_GREEN waiting for EW sensor), reload
60              // This logic is handled by the state transition logic checking timer == 0
61              // Optional: Could reload timer here if needed for specific behavior
62              // else begin
63              //    // Reload timer based on current state if it hits 0 and no transition occurs yet
64              //    case (current_state) // Or next_state, depending on exact timing desired
65              //        NS_GREEN: state_timer <= NS_GREEN_CYCLES - 1;
66              //        // ... handle other states if they can self-loop on timer expiry
67              //        default: state_timer <= state_timer; // No change or handle appropriately
68              //    endcase
69              // end
70              end
71              // No else needed: if timer is 0 and state doesn't change, timer stays 0 until state transition reloads it.
72          end
73      end
74
```

Figure 2: Verilog code page 2

C:/altera/13.0sp1/Traffic_Controller.v (/tb_traffic_controller/uut) - Default *

```
75      // Next State Logic (Combinational) - Using standard Verilog always block
76      always @(*) begin // Use @(*) for combinational logic sensitivity list
77          next_state = current_state; // Default: stay in current state
78
79          // Emergency has highest priority
80          if (emergency) begin
81              case (current_state)
82                  NS_GREEN, EMERGENCY_GREEN: next_state = EMERGENCY_GREEN; // Already in or going to NS Green
83                  EW_GREEN:                  next_state = EW_YELLOW;       // Go to EW Yellow first
84                  EW_YELLOW:                 next_state = EMERGENCY_TRANS; // Transition through EW_YELLOW
85                  NS_YELLOW:                 next_state = EMERGENCY_GREEN; // Can go directly from NS_YELLOW
86                  EMERGENCY_TRANS:           next_state = EMERGENCY_GREEN; // Wait finished, go green
87                  INIT:                      next_state = EMERGENCY_GREEN; // Go directly if possible
88                  default:                   next_state = EMERGENCY_GREEN; // Go directly if possible
89              endcase
90          end else begin // Normal operation
91              case (current_state)
92                  INIT: begin
93                      next_state = NS_GREEN; // Start with NS Green after init/reset
94                  end
95                  NS_GREEN: begin
96                      // If timer expired AND there's demand from EW
97                      // Note: comparison uses '==' for combinational logic
98                      if (state_timer == 0 && ew_sensor_active) begin
99                          next_state = NS_YELLOW;
100                     end
101                     // Add logic here if you want NS_GREEN to yield even without EW demand after a max time
102                     // else if (state_timer == 0) begin // Example: Force cycle even if no EW traffic
103                     //    next_state = NS_YELLOW;
104                     // end
105                 end
106                 NS_YELLOW: begin
107                     if (state_timer == 0) begin
108                         next_state = EW_GREEN;
109                     end
110                 end
111                 EW_GREEN: begin
112                     // If timer expired AND there's demand from NS
113                     if (state_timer == 0 && ns_sensor_active) begin
114                         next_state = EW_YELLOW;
115                     end
116                     // Add logic here if you want EW_GREEN to yield even without NS demand after a max time
117                      // else if (state_timer == 0) begin // Example: Force cycle even if no NS traffic
118                      //    next_state = EW_YELLOW;
119                      // end
120                 end
```

Figure 3: Verilog code page 3

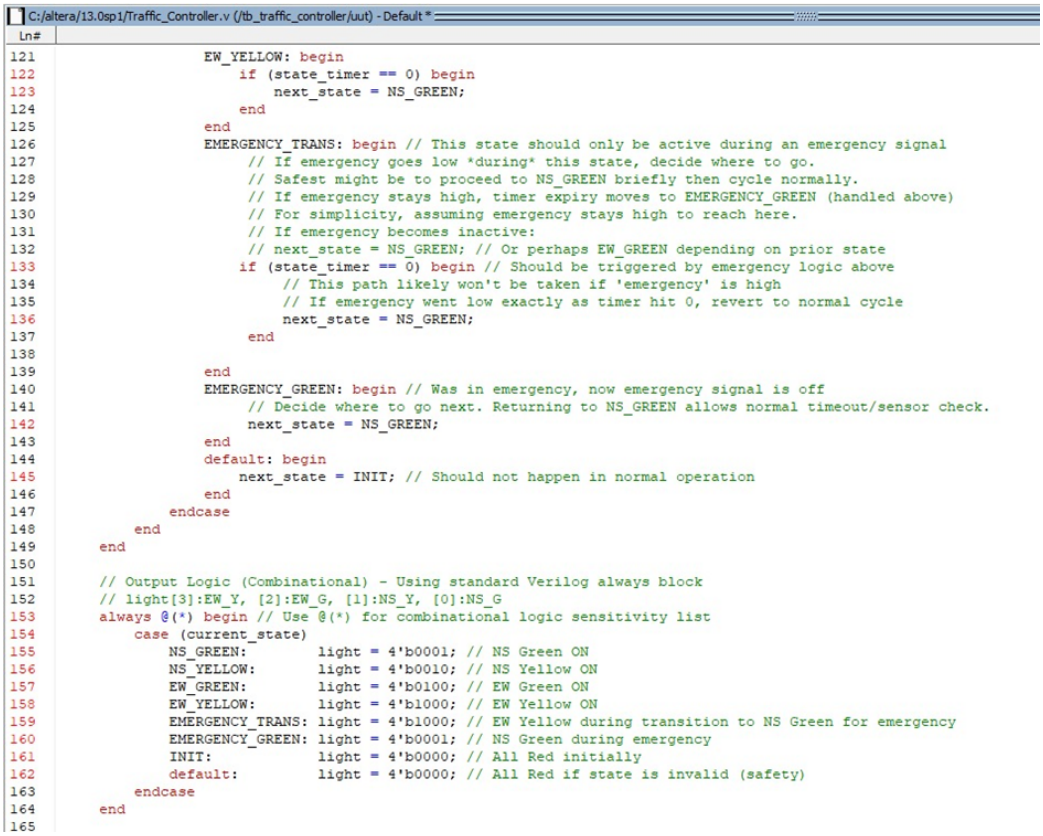

Figure 4: Verilog code page 4

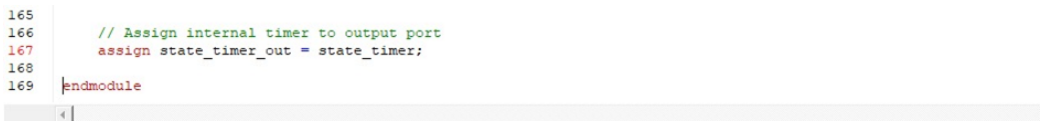

Figure 5: Verilog code page 5

The Traffic_Controller module simulates a simple but practical intelligent traffic light controller for a two-way intersection, specifically handling North-South (NS) and East-West (EW) traffic, along with emergency vehicle priority. The system is designed to optimize traffic flow based on real-time sensor inputs and prioritizes safety by incorporating structured light transitions and emergency handling. The controller operates through a finite state machine (FSM) with seven defined states:

- INIT
- NS_GREEN

- NS_YELLOW
- EW_GREEN
- EW_YELLOW
- EMERGENCY_TRANS
- EMERGENCY_GREEN

Each state represents a particular light condition, such as turning the NS or EW direction green or yellow. An emergency override state (EMERGENCY_GREEN) ensures that when an emergency vehicle signal is detected, the system gives immediate and uninterrupted green light access to the NS direction, facilitating safe passage for ambulances, fire trucks, or other priority vehicles. The system uses sensor inputs to detect vehicle presence in both NS and EW directions. When the controller is running in normal mode (no emergency), it allows a green light for the direction where vehicles are detected. If the timer for the current green light expires and the opposite direction has waiting traffic, the state transitions to yellow and then switches green to the other direction. This behavior ensures that no direction is unfairly blocked, and the system adapts to real-time traffic volume. When an emergency signal is triggered, the state machine transitions to prioritize NS traffic by either immediately switching to EMERGENCY_GREEN or transitioning through EW_YELLOW and EMERGENCY_TRANS for a safe shift. A dedicated short wait (EMERGENCY_WAIT) is implemented during this transition to prevent abrupt changes, ensuring a smooth and safe shift from regular operation to emergency mode. The module uses an internal state_timer to control the duration of each traffic light state. The timer is initialized with a predefined count when a new state is entered and decrements on every clock cycle. This mechanism makes sure the lights stay green, yellow, or red for the correct period. Additionally, the timer value is exposed through an output port (state_timer_out) for easy simulation and debugging. The light signals are represented as a 4-bit output (light) where each bit corresponds to a specific traffic light: North-South Green, North-South Yellow, East-West Green, and East-West Yellow. Depending on the active state, the controller sets the appropriate light combination to safely manage vehicle flow at the intersection. Overall, this Verilog module efficiently demonstrates the principles of digital system design, including state-driven behavior, priority-based decision-making, and timing control, all of which are fundamental to real-world traffic light control systems. Its structure also ensures easy adaptability for further extensions like pedestrian signals, more complex priority logic, or integration with urban traffic management systems.

6.2 Verilog Testbench

C:/altera/13.0sp1/tb_traffic_controller.v (/tb_traffic_controller) - Default *

```
Ln#
 1  //`default_nettype none
 2  `timescale 1ns / 1ps
 3
 4  module tb_traffic_controller();
 5
 6      reg clk;
 7      reg reset;
 8      reg emergency;
 9      reg [3:0] traffic_sensors; // [3]:EW2, [2]:EW1, [1]:NS2, [0]:NS1
10
11      wire [3:0] light;      // [3]:EW_Y, [2]:EW_G, [1]:NS_Y, [0]:NS_G
12      wire [7:0] state_timer_out; // Match DUT output width
13
14      // Instantiating the traffic controller module
15      Traffic_Controller uut (
16          .clk(clk),
17          .reset(reset),
18          .emergency(emergency),
19          .traffic_sensors(traffic_sensors),
20          .light(light),
21          .state_timer_out(state_timer_out)
22      );
23
24      // Clock generation (100 MHz)
25      localparam CLK_PERIOD = 10; // ns
26      always begin
27          clk = 1'b0;
28          #(CLK_PERIOD / 2);
29          clk = 1'b1;
30          #(CLK_PERIOD / 2);
31      end
32
```

Figure 6: Verilog testbench

C:/altera/13.0sp1/tb_traffic_controller.v (/tb_traffic_controller) - Default *

```
33      // Simulation Control and Stimulus
34      initial begin
35          // --- Setup ---
36          $display("[%t ns] Starting Testbench...", $time);
37          reset = 1'b1;   // Assert reset
38          emergency = 1'b0;
39          traffic_sensors = 4'b0000;
40          repeat (3) @(posedge clk); // Hold reset for 3 cycles
41          reset = 1'b0;   // De-assert reset
42          $display("[%t ns] Reset Released.", $time);
43          @(posedge clk); // Wait one cycle for reset to propagate
44
45          // --- Scenario 1: NS Green (initial) -> EW Demand -> EW Green ---
46          $display("[%t ns] Scenario 1: NS Green, then EW demand.", $time);
47          traffic_sensors = 4'b0000; // No demand initially
48          // Wait long enough for NS Green timer to potentially expire if there *were* demand
49          #( (100 + 20 + 10) * CLK_PERIOD ); // Wait roughly NS_G + NS_Y duration + buffer
50          $display("[%t ns] Activating EW sensors.", $time);
51          traffic_sensors = 4'b1100; // Activate EW sensors
52          // Wait long enough for transition: NS_G -> NS_Y -> EW_G
53          #( (100 + 20 + 60 + 20 + 10) * CLK_PERIOD ); // Wait NS_G expiry + NS_Y + EW_G + EW_Y + buffer
54
55          // --- Scenario 2: EW Green -> NS Demand -> NS Green ---
56           $display("[%t ns] Scenario 2: EW Green, then NS demand.", $time);
57          traffic_sensors = 4'b0011; // Activate NS sensors (EW sensors off)
58          // Wait long enough for transition: EW_G -> EW_Y -> NS_G
59          #( (60 + 20 + 100 + 20 + 10) * CLK_PERIOD ); // Wait EW_G expiry + EW_Y + NS_G + NS_Y + buffer
60
61          // --- Scenario 3: Emergency Override during EW Green ---
62          $display("[%t ns] Scenario 3: Emergency during EW Green.", $time);
63          // First, force it back to EW Green
64          traffic_sensors = 4'b1100; // EW demand
65          #( (100 + 20 + 10) * CLK_PERIOD ); // Wait NS_G -> NS_Y
66          $display("[%t ns] Should be EW Green now. Triggering Emergency.", $time);
67          emergency = 1'b1; // <<<<< EMERGENCY ON
68          // Wait long enough for emergency state to take effect and stay
69          #( (20 + 100) * CLK_PERIOD ); // Wait Y + G duration
70          $display("[%t ns] Emergency still active.", $time);
71          emergency = 1'b0; // <<<<< EMERGENCY OFF
72          traffic_sensors = 4'b0000; // Clear sensors
73          $display("[%t ns] Emergency OFF. Resuming normal operation.", $time);
74          // Wait long enough for it to cycle back based on sensors (or lack thereof)
75          #( (100 + 20 + 60 + 20 + 10) * CLK_PERIOD );
76
77          // --- Finish Simulation ---
78          $display("[%t ns] Test scenarios complete. Finishing simulation.", $time);
79          #50; // Extra delay before finishing
80          $finish;
```

Figure 7: Verilog testbench page 2

```
81      end
82
83      // Monitoring and VCD Dump
84      initial begin
85          // Setup VCD dump
86          $dumpfile("traffic_dump.vcd");
87          // Dump all signals in the testbench and the instantiated DUT (uut)
88          $dumpvars(0, tb_traffic_controller);
89
90          // Monitor key signals to console
91
92          $strobe("[%t ns] State=%b Light(EW_Y,EW_G,NS_Y,NS_G)=%b Sensors(EW,NS)=%b Timer=%3d Emerg=%b",
93           $time, uut.current_state, light, traffic_sensors[3:0], state_timer_out, emergency);
94      end
95
96  endmodule
97
```

Figure 8: Verilog testbench page 3

The Traffic_Controller Testbench is designed to simulate and validate the functional behavior of the Traffic_Controller module in a controlled environ-

ment. It acts as a virtual setup where real-world scenarios like vehicle flow and emergency conditions are mimicked using digital input signals, allowing developers to observe how the traffic controller responds without deploying the system physically.

The testbench begins by creating and initializing the necessary signals: a clock (clk), reset (reset), emergency input (emergency), and traffic_sensors which represent the presence of vehicles in both the East-West (EW) and North-South (NS) directions. The outputs from the Traffic_Controller module, namely the light signal and state_timer_out, are connected to observe and verify the module's response under various conditions.

A key component of the testbench is the clock generation block. This generates a continuous square wave clock with a defined period (100 MHz frequency), simulating the timing behavior that digital circuits rely on. The clock toggles every 5 nanoseconds to create a stable test environment for the Traffic_Controller module to operate synchronously. The simulation control block is responsible for creating different test scenarios by manipulating the traffic_sensors and emergency signals. The testbench starts with a system reset to ensure the module begins from a known state, and then three main scenarios are executed:

1. **Scenario 1: Normal Traffic Flow** — NS to EW Transition
   Initially, no vehicles are present, allowing the controller to stay in the default North-South green state. After a set period, East-West direction sensors are activated, representing waiting vehicles. This triggers the expected transition from NS Green to NS Yellow, and then to EW Green as the system reacts to new traffic demand.

2. **Scenario 2: Traffic Switch** — EW to NS Transition
   In this scenario, the sensors indicate a vehicle demand on the North-South route while East-West traffic has the green light. The testbench ensures the system transitions properly from EW Green to EW Yellow, and finally grants NS Green once the EW side has been safely cleared.

3. **Scenario 3: Emergency Override During EW Green**
   This scenario tests the emergency response feature. While East-West traffic is active, the emergency signal is triggered, simulating an emergency vehicle requiring priority access on the NS route. The system is expected to override normal conditions, shift states to safely handle the emergency, and prioritize the NS Green light. Once the emergency signal is cleared, normal traffic management resumes.

Alongside these scenarios, the testbench also incorporates a VCD (Value Change Dump) setup to record all signal transitions during the simulation,

enabling detailed waveform analysis using external tools like GTKWave. Additionally, a $strobe monitor continuously prints the current simulation time, system state, light status, sensor input, timer value, and emergency flag to the console. This real-time monitoring helps verify that the traffic controller's behavior aligns with the expected logic at each moment.

Overall, this testbench provides a thorough and systematic way to validate the Traffic_Controller's design, helping to confirm its correctness under varying traffic conditions and during emergency situations. It also helps in identifying any bugs or logical errors before implementing the module in hardware, making it a vital part of the digital design workflow.

6.3 Verilog Output

```
VSIM 19> run -all
# [                    0 ns] Starting Testbench...
# [                    0 ns] State=000 Light(EW_Y,EW_G,NS_Y,NS_G)=0000 Sensors(EW,NS)=0000 Timer=  0 Emerg=0
# [                25000 ns] Reset Released.
# [                35000 ns] Scenario 1: NS Green, then EW demand.
# [              1335000 ns] Activating EW sensors.
# [              3435000 ns] Scenario 2: EW Green, then NS demand.
# [              5535000 ns] Scenario 3: Emergency during EW Green.
# [              6835000 ns] Should be EW Green now. Triggering Emergency.
# [              8035000 ns] Emergency still active.
# [              8035000 ns] Emergency OFF. Resuming normal operation.
# [             10135000 ns] Test scenarios complete. Finishing simulation.
# ** Note: $finish    : C:/altera/13.0spl/tb_traffic_controller.v(81)
#    Time: 10185 ns  Iteration: 0  Instance: /tb_traffic_controller
# 1
# Break in Module tb_traffic_controller at C:/altera/13.0spl/tb_traffic_controller.v line 81

VSIM 20>
```

Figure 9: Output

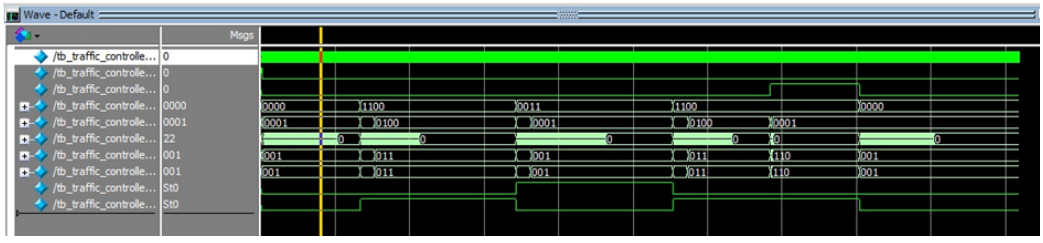

Figure 10: Output waveform 1

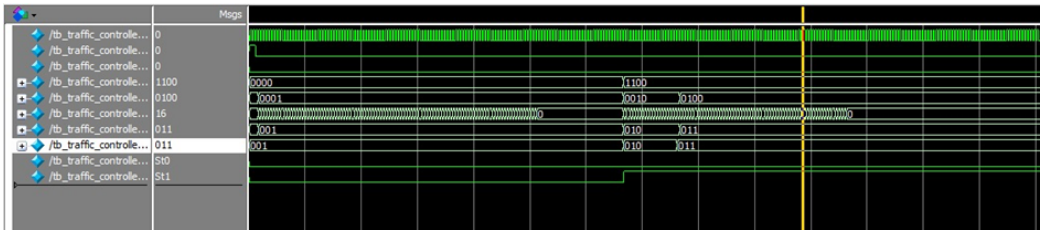

Figure 11: Output waveform 2

The simulation log breaks down the behavior into scenarios:

Simulation Timeline and Scenario Description

- **0 ns** — The system starts in an initial state with all lights off (`0000`), all sensors inactive (`0000`), the timer at zero, and no emergency.
- **25,000 ns** — Reset is released, allowing the controller to start normal operation.
- **35,000 ns**
  Scenario 1: NS Green, then EW Demand — The controller first gives green to the North-South direction. When demand is detected from the East-West direction via sensors, the state transitions accordingly.
- **1,335,000 ns**
  Scenario 2: Activating EW Sensors — The system detects traffic on the East-West road, prompting the controller to assign green to the East-West direction.
- **3,435,000 ns**
  Scenario 3: Emergency during EW Green — An emergency situation is simulated while the East-West light is green, prompting the controller to handle priority switching or override.
- **6,835,000 ns**
  Emergency Ongoing — The emergency condition remains active, forcing the system to stay in a safe or specific emergency state.
- **8,035,000 ns**
  Emergency Cleared — The emergency flag is turned off, and the controller resumes normal operation.

- **10,135,000 ns** — Simulation ends with all test scenarios successfully completed.

The waveforms visually confirm the correctness of the system's logic:

- **Clock Signal:**

  The top line shows a stable periodic clock driving the state machine.

- **Sensor Inputs (EW, NS):**

  These signals show when vehicles are detected on the East-West or North-South roads. When set high, the controller prepares to switch lights accordingly.

- **Light Signals (EW_Y, EW_G, NS_Y, NS_G):**

  These outputs represent the traffic light status:
  - **EW_G** and **NS_G**: Green lights for East-West and North-South directions.
  - **EW_Y** and **NS_Y**: Yellow lights indicating transition phases.

- **Emergency Signal:**

  During the scenario where the emergency is triggered, the light outputs are overridden to handle the emergency situation safely.

- **State Signals (St0, St1):**

  These represent the internal state of the state machine in binary form, showing the transition from one operational mode to another — including normal traffic light cycles and emergency overrides.

6.4 C++ Code for Tinkercad

```
// --- Pin Definitions ---
const int RESET_PIN = 2;
const int EMERGENCY_PIN = 3;
const int SENSOR_NS1_PIN = 4;
const int SENSOR_NS2_PIN = 5;
const int SENSOR_EW1_PIN = 6;
const int SENSOR_EW2_PIN = 7;

const int LIGHT_NS_G_PIN = 8;
const int LIGHT_NS_Y_PIN = 9;
const int LIGHT_EW_G_PIN = 10;
const int LIGHT_EW_Y_PIN = 11;

// --- State Durations (in Milliseconds) ---
// Adjusted from Verilog cycles for simulation visibility
const unsigned long NS_GREEN_MS = 10000; // 10 seconds
const unsigned long EW_GREEN_MS = 6000;  // 6 seconds
const unsigned long YELLOW_MS = 2000;    // 2 seconds
const unsigned long EMERGENCY_WAIT_MS = 500; // 0.5 seconds (for EW_YELLOW ->
const unsigned long INIT_MS = 100;       // Short time in INIT state

```

Figure 12: C++ code

```
// --- State Definitions ---
enum StateType {
  INIT,
  NS_GREEN,
  NS_YELLOW,
  EW_GREEN,
  EW_YELLOW,
  EMERGENCY_TRANS,
  EMERGENCY_GREEN
};

// --- State Variables ---
StateType current_state = INIT;
StateType next_state = INIT; // Stores the calculated next state
unsigned long stateStartTime = 0; // Records when the current state started (us

// --- Input Variables ---
bool reset_active = false;
bool emergency_active = false;
bool ns_sensor_active = false;
bool ew_sensor_active = false;

void readInputs();
void updateLights();
void printStateName(StateType state);

```

Figure 13: C++ code page 2

```
// --- Setup Function (runs once) ---
void setup() {
  Serial.begin(9600); // Initialize serial communication for debugging
  Serial.println("Traffic Light Controller Initializing...");

  // Configure Input Pins
  pinMode(RESET_PIN, INPUT_PULLUP);       // Active LOW reset
  pinMode(EMERGENCY_PIN, INPUT_PULLUP);    // Active LOW emergency
  pinMode(SENSOR_NS1_PIN, INPUT);        // Active HIGH sensor
  pinMode(SENSOR_NS2_PIN, INPUT);        // Active HIGH sensor
  pinMode(SENSOR_EW1_PIN, INPUT);        // Active HIGH sensor
  pinMode(SENSOR_EW2_PIN, INPUT);        // Active HIGH sensor

  // Configure Output Pins (Lights)
  pinMode(LIGHT_NS_G_PIN, OUTPUT);
  pinMode(LIGHT_NS_Y_PIN, OUTPUT);
  pinMode(LIGHT_EW_G_PIN, OUTPUT);
  pinMode(LIGHT_EW_Y_PIN, OUTPUT);

  // Initialize state and timer
  current_state = INIT;
  stateStartTime = millis();
  updateLights(); // Set initial light state (all off)

  Serial.println("Initialization Complete. Starting FSM.");
}

```

Figure 14: C++ code page 3

```
// --- Loop Function (runs repeatedly) ---
void loop() {
  // 1. Read Inputs
  readInputs();

  // 2. Check for Reset (Highest priority after reading inputs)
  if (reset_active) {
    Serial.println("RESET Activated!");
    current_state = INIT;
    stateStartTime = millis(); // Reset timer
    updateLights();
    // Optional: Add a small delay or wait for reset release
    delay(500); // Debounce/hold reset state briefly
    return; // Skip the rest of the loop iteration
  }

  // 3. Determine Next State Logic (Combinational equivalent)
  unsigned long currentTime = millis();
  unsigned long elapsedTime = currentTime - stateStartTime;
  unsigned long currentDuration = 0; // Duration required for the current sta

  // Default: stay in the current state unless a condition changes it
  next_state = current_state;

  // --- Emergency Logic ---
  if (emergency_active) {
    // Emergency overrides normal operation
    switch (current_state) {
      case NS_GREEN:
      case EMERGENCY_GREEN:
        next_state = EMERGENCY_GREEN; // Already in NS Green or stay there
        break;
      case EW_GREEN:
        next_state = EW_YELLOW; // Go Yellow first
        currentDuration = YELLOW_MS; // Need yellow duration
        break;
      case EW_YELLOW:
        currentDuration = YELLOW_MS; // Check yellow duration
        if (elapsedTime >= currentDuration) {
             next_state = EMERGENCY_TRANS; // Go to transition state after ye
             // Note: Verilog used EMERGENCY_WAIT for TRANS state duration.
        } else {
             next_state = EW_YELLOW; // Stay yellow until timer expires
        }
        break;
```

Figure 15: C++ code page 4

```
120         case EMERGENCY_TRANS:
121           currentDuration = EMERGENCY_WAIT_MS; // Wait state duration
122           if (elapsedTime >= currentDuration) {
123               next_state = EMERGENCY_GREEN; // Wait finished, go NS Green
124           } else {
125               next_state = EMERGENCY_TRANS; // Stay waiting
126           }
127           break;
128       case NS_YELLOW:
129         // Can go directly to NS Green from NS Yellow during emergency
130         next_state = EMERGENCY_GREEN;
131         break;
132       case INIT:
133       default: // Includes INIT
134         next_state = EMERGENCY_GREEN; // Go directly to NS Green
135         break;
136     }
137   } else {
138     // --- Normal Operation Logic ---
```

Figure 16: C++ code page 5

```
    switch (current_state) {
      case INIT:
        currentDuration = INIT_MS;
        if (elapsedTime >= currentDuration) {
          next_state = NS_GREEN; // Default start after INIT
        }
        break;

      case NS_GREEN:
        currentDuration = NS_GREEN_MS;
        // Transition if timer expired AND there's demand from EW
        if (elapsedTime >= currentDuration && ew_sensor_active) {
          next_state = NS_YELLOW;
        }
        // Optional: Add max time logic even without EW demand here if needed
        // else if (elapsedTime >= MAX_NS_GREEN_MS) { next_state = NS_YELLOW;
        break;

      case NS_YELLOW:
        currentDuration = YELLOW_MS;
        if (elapsedTime >= currentDuration) {
          next_state = EW_GREEN;
        }
        break;

      case EW_GREEN:
        currentDuration = EW_GREEN_MS;
        // Transition if timer expired AND there's demand from NS
        if (elapsedTime >= currentDuration && ns_sensor_active) {
          next_state = EW_YELLOW;
        }
        // Optional: Add max time logic even without NS demand here if needed
        // else if (elapsedTime >= MAX_EW_GREEN_MS) { next_state = EW_YELLOW;
        break;

      case EW_YELLOW:
        currentDuration = YELLOW_MS;
        if (elapsedTime >= currentDuration) {
          next_state = NS_GREEN;
        }
        break;

```

Figure 17: C++ code page 6

```
      case EMERGENCY_TRANS:
         // Emergency ended during transition - revert to normal cycle safely
         // Go to NS_GREEN as a safe default after clearing
         next_state = NS_GREEN;
         Serial.println("Emergency ended during TRANS -> NS_GREEN");
        break;

      case EMERGENCY_GREEN:
        // Emergency signal just went low, transition out of emergency state
        next_state = NS_GREEN; // Return to normal NS Green
        Serial.println("Emergency ended -> NS_GREEN");
        break;

       default:
        // Should not happen
        next_state = INIT;
        break;
    }
  }

```

Figure 18: C++ code page 7

```
  // 4. State Transition Logic (Clocked equivalent)
  if (next_state != current_state) {
    Serial.print("State Change: ");
    printStateName(current_state);
    Serial.print(" -> ");
    printStateName(next_state);
    Serial.println();

    current_state = next_state;
    stateStartTime = currentTime; // Reset timer for the new state

    // Update lights immediately after state change
    updateLights();
  }
}

```

Figure 19: C++ code page 8

```
// --- Helper Function: Read Inputs ---
void readInputs() {
  // Read reset pin (Active LOW)
  reset_active = (digitalRead(RESET_PIN) == LOW);

  // Read emergency pin (Active LOW)
  emergency_active = (digitalRead(EMERGENCY_PIN) == LOW);

  // Read sensors (Active HIGH) - Combine sensors for each direction
  ns_sensor_active = (digitalRead(SENSOR_NS1_PIN) == HIGH) || (digitalRead(SENSOR_NS2_PIN) == HIGH);
  ew_sensor_active = (digitalRead(SENSOR_EW1_PIN) == HIGH) || (digitalRead(SENSOR_EW2_PIN) == HIGH);
}

// --- Helper Function: Update Light Outputs ---
void updateLights() {
  // Turn all lights off first
  digitalWrite(LIGHT_NS_G_PIN, LOW);
  digitalWrite(LIGHT_NS_Y_PIN, LOW);
  digitalWrite(LIGHT_EW_G_PIN, LOW);
  digitalWrite(LIGHT_EW_Y_PIN, LOW);

  // Turn on the correct light(s) based on the current state
  switch (current_state) {
    case NS_GREEN:
    case EMERGENCY_GREEN: // NS Green light during emergency
      digitalWrite(LIGHT_NS_G_PIN, HIGH);
      break;
    case NS_YELLOW:
      digitalWrite(LIGHT_NS_Y_PIN, HIGH);
      break;
    case EW_GREEN:
      digitalWrite(LIGHT_EW_G_PIN, HIGH);
      break;
    case EW_YELLOW:
    case EMERGENCY_TRANS: // EW Yellow light during transition for emergency
       digitalWrite(LIGHT_EW_Y_PIN, HIGH);
       break;
    case INIT:
    default:
      // All lights remain OFF (handled by initial turn-off)
      break;
  }
}
```

Figure 20: C++ code page 9

```
// --- Helper Function: Print State Name ---
void printStateName(StateType state) {
  switch (state) {
    case INIT: Serial.print("INIT"); break;
    case NS_GREEN: Serial.print("NS_GREEN"); break;
    case NS_YELLOW: Serial.print("NS_YELLOW"); break;
    case EW_GREEN: Serial.print("EW_GREEN"); break;
    case EW_YELLOW: Serial.print("EW_YELLOW"); break;
    case EMERGENCY_TRANS: Serial.print("EMERGENCY_TRANS"); break;
    case EMERGENCY_GREEN: Serial.print("EMERGENCY_GREEN"); break;
    default: Serial.print("UNKNOWN"); break;
  }
}
```

Figure 21: C++ code page 10

This code simulates a basic traffic light controller (designed on Verilog) using an Arduino and C++ programming. It is designed to handle regular traffic flow as well as emergency vehicle priority and manual reset. The logic behind the code is structured around a finite state machine (FSM) approach, where

the system moves between predefined states such as **INIT**, **NS_GREEN**, **NS_YELLOW**, **EW_GREEN**, **EW_YELLOW**, and **EMERGENCY_GREEN**. Each state corresponds to a particular behavior of the traffic lights, allowing the North-South (NS) and East-West (EW) directions to take turns passing through the intersection while responding to special conditions when needed.

When the Arduino is powered on, the **setup()** function initializes the serial monitor and sets up the digital pins that control LEDs (representing the traffic lights) and buttons (representing emergency requests and reset actions). After initialization, the program enters the **loop()** function, where it continuously checks for button inputs that might trigger a state change.

If the emergency button is pressed, the program immediately transitions to the **EMERGENCY_GREEN** state, giving priority to emergency vehicles by turning on the NS green light. Once the emergency button is released, the system safely returns to the regular **NS_GREEN** state, resuming normal operation.

If the reset button is pressed, the system resets itself, returning to the **INIT** state and starting the cycle over from **NS_GREEN**. In the absence of emergencies or resets, the controller follows a regular cycle: NS gets the green light (**NS_GREEN**) for a set time, then yellow (**NS_YELLOW**), after which the EW direction gets the green (**EW_GREEN**) and yellow (**EW_YELLOW**). Once the EW yellow time is over, the cycle loops back to **NS_GREEN**, repeating indefinitely unless interrupted by an emergency or manual reset.

Each state transition is printed to the serial monitor, giving real-time feedback on the current status of the system. This setup closely mimics how actual traffic controllers operate at intersections, including the ability to override normal flow for ambulances, fire trucks, or police vehicles, and to reset the system when required — making it a great introductory example of embedded systems, state machines, and real-world problem-solving with Arduino.

6.5 Tinkercad Output

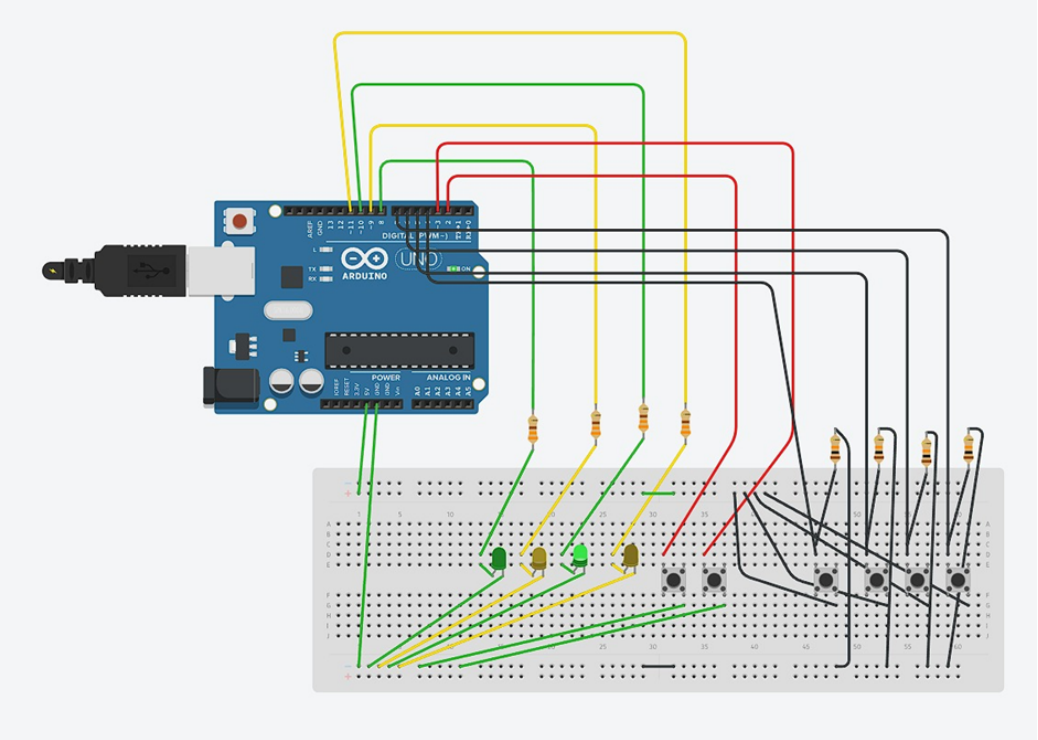

Figure 22: Green Signal ON

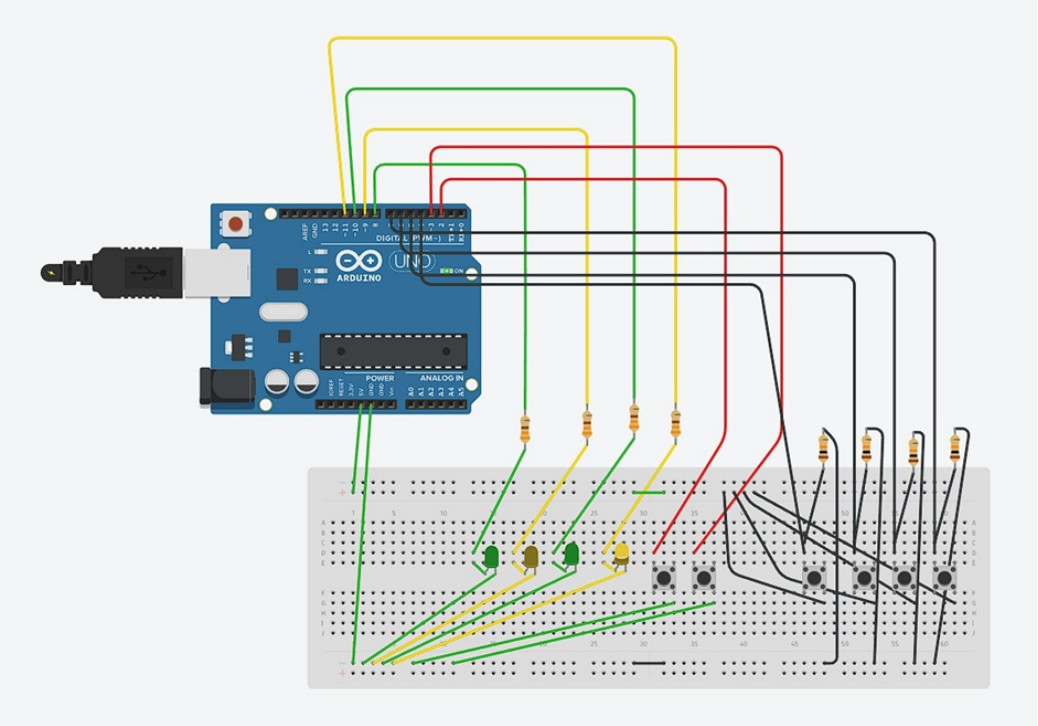

Figure 23: Yellow Signal ON

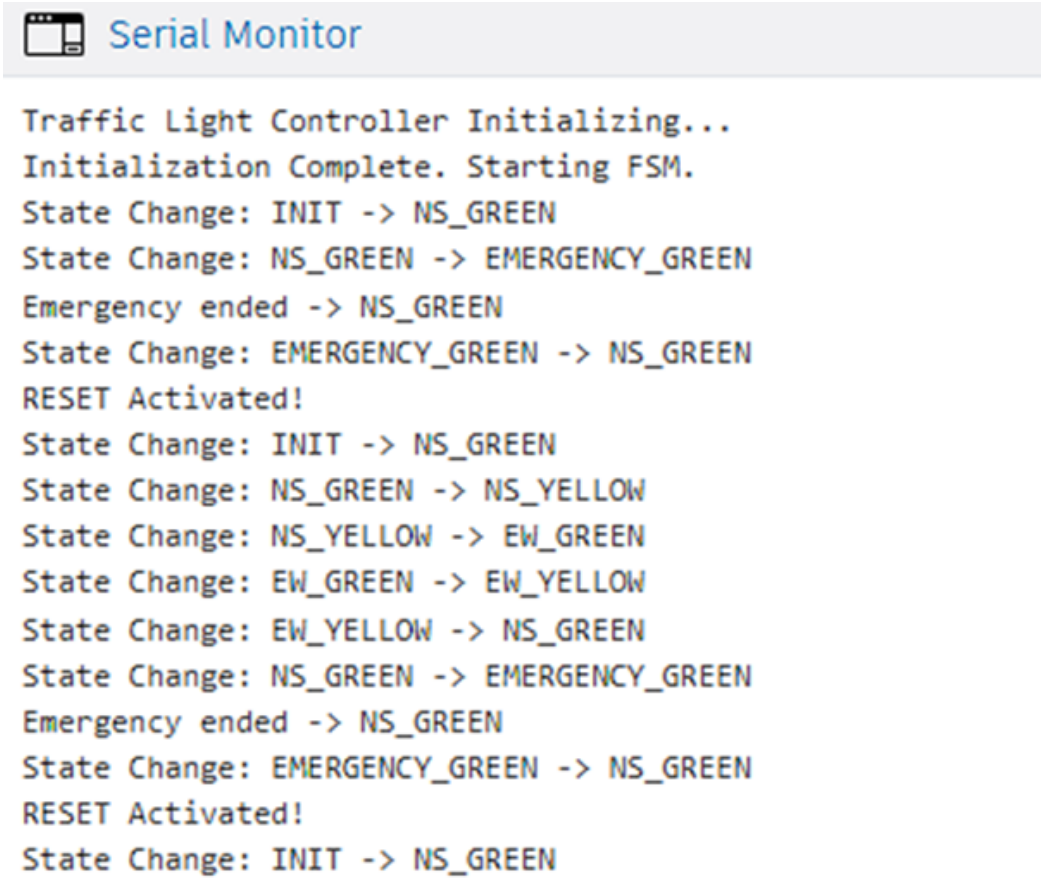

Figure 24: Output

The serial monitor output shows that the traffic light controller code is working correctly based on the designed logic. When the program starts, it initializes and moves from the **INIT** state to **NS_GREEN**, allowing North-South traffic to pass.

At some point, an emergency button is pressed, so the system switches from **NS_GREEN** to **EMERGENCY_GREEN**, giving priority to the emergency vehicle by keeping the NS green light on. Once the emergency ends, the system safely returns to **NS_GREEN** to continue normal traffic flow.

Later, a reset button is pressed, which forces the system to reinitialize and restart the traffic cycle from **INIT** to **NS_GREEN**. After that, the system continues its normal sequence — transitioning from **NS_GREEN** to **NS_YELLOW**, then to **EW_GREEN**, followed by **EW_YELLOW**, and finally looping back to **NS_GREEN**.

This output confirms that both emergency handling and reset functions are working, and the FSM logic for normal light cycling is performing as intended.

7 Reference

1. M. Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL.*

2. Icarus Verilog Documentation. Available: `https://steveicarus.github.io/iverilog/`

3. M. C. Ho, J. M.-Y. Lim, C. Y. Chong, K. K. Chua, and A. K. L. Siah, "A Novel Adaptive Traffic Light Control in Connected Vehicle Environment," *IEEE Transactions on Vehicular Technology*, vol. 74, no. 2, pp. 2470–2479, Feb. 2025, doi: 10.1109/TVT.2024.3478233.

Keywords:
Roads; Traffic congestion; Vehicular ad hoc networks; Heuristic algorithms; Genetic algorithms; Vehicle routing; Vehicle dynamics; Sensors; Real-time systems; Biological system modeling; Intelligent transportation system; Traffic light control; Vehicle rerouting