Fall 2024

Selected Topics in Computer Design-CSE416s

Final Project

Adaptive Traffic Light Controller-Part A

Team #4

Members:

Name	Code	Department
Fares Khalaf Salman Sultan	2101371	CSE
Youssef Hany Elreweny	2101449	CSE
Ahmed Ashraf Ali	2100255	CSE
Youssef Ossama Sayed	2101240	CSE

❖ Controller Specifications:

The controller is designed for use in a four-way intersection. It uses sensors readings to prioritize traffic flow dynamically. The controller operates through a finite state machine to cycle through traffic light states based on traffic conditions and timing, and an internal counter loaded with an appropriate Value based on the Traffic light.

1. Features:

- **Dynamic Traffic Management**: Adjusts traffic light durations based on real-time vehicle density from four directions (A, B, C, D).
- **Sequential State Transition**: Includes green, orange, and red states for each direction.
- **Sensor-Based Prioritization**: Uses sensor readings (Sa, Sb, Sc, Sd) to determine the next state.
- **Counter-Based Timing**: A configurable counter determines the duration of each traffic light state.

2. Inputs and Outputs:

- **Inputs**:
  - **Sa, Sb, Sc, Sd:** Two-bit signals representing traffic density in each direction (00→No cars, 01→Light Traffic, 11→Heavy Traffic).
  - **clk:** Clock signal.
  - **rst_n**: Asynchronous Active-low reset signal.
- **Outputs**:
  - **Ta, Tb, Tc, Td:** Three-bit signals representing the traffic light state for directions A, B, C, and D (001 → green, 010 →orange, 100 → red).

3. State Description:

- **Green States (Ga, Gb, Gc, Gd)**:
  - The corresponding direction has a green light.
  - Counter set to 30 seconds.
  - Transitions to orange state (Oa, Ob, Oc, Od) when the counter expires.
- **Orange States (Oa, Ob, Oc, Od)**:
  - The corresponding direction has an orange light.
  - Counter set to 3 seconds.
  - Transitions to the green state of the next prioritized direction.

4. Timing:

- Green light duration: 30 seconds (adjustable via load_value).
- Orange light duration: 3 seconds (fixed).
- Counter decrements each clock cycle.

5. Priority Rules:

- The direction with the highest traffic density has priority for the next green light.
- If multiple directions have equal density, default to a fixed priority order:

(A → B → C → D).

❖ Traffic Light Algorithm:

1. Initialization:

1. Set **current state** to **Ga** (Direction A green).
2. Load the counter with 30 for green light duration.

2. FSM Logic:

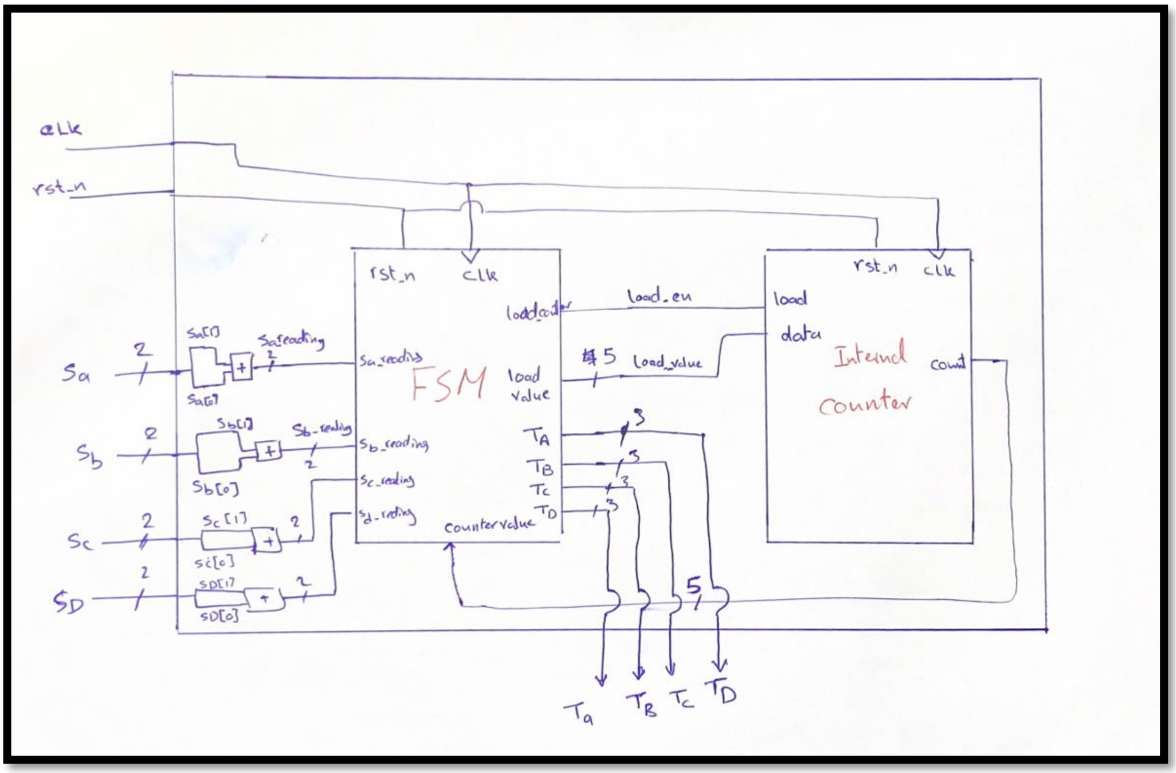
1. **Green States (Gx)**:
   - Check sensor readings:
     - If the current direction (Sa, Sb, Sc, or Sd) has the highest density, remain in the green state (Gx).
   - If the counter expires, transition to the corresponding orange state (Ox).
2. **Orange States (Ox)**:
   - Decrement the counter.
   - If the counter expires:
     - **Determine the next direction to prioritize based on sensor inputs:**
       - transition to the green state of the direction with the highest traffic density.
       - In case of ties, follow a fixed order (A → B → C → D).

3. Output Logic:

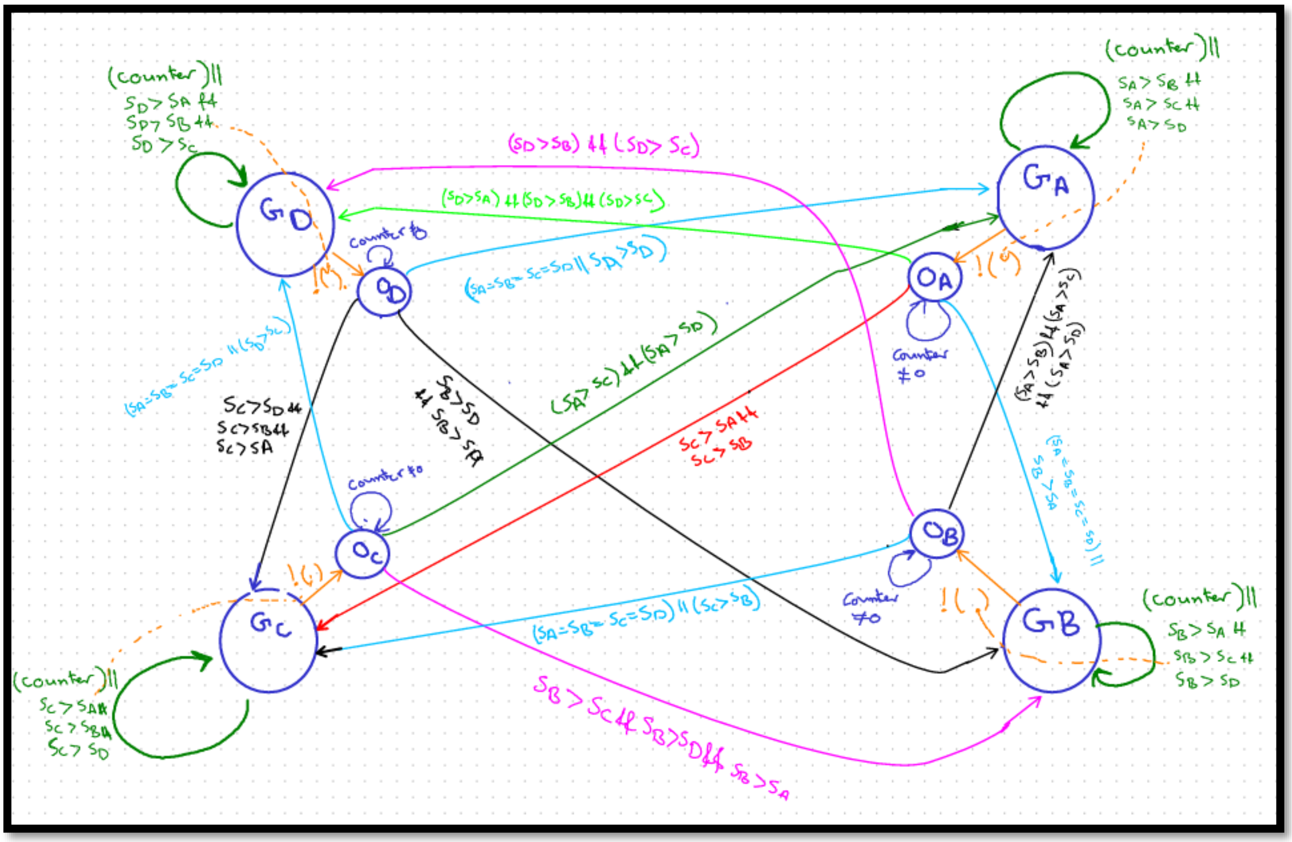
Use the current state to determine light signals:

- In Gx: The corresponding direction is green, and others are red.
- In Ox: The corresponding direction is orange, and others are red.
- When transitioning between states, reload the counter with the appropriate value (30 for green, 3 for orange).

❖ System Design:



❖ FSM Diagram:



❖ RTL code:

- Traffic_controller (FSM):

```
module Traffic_Controller (Sa,Sb,Sc,Sd,clk,rst_n,counter_value,Ta,Tb,Tc,Td,load_counter,load_value);
    parameter Ga = 3'b000 ;
    parameter Gb = 3'b001 ;
    parameter Gc = 3'b010 ;
    parameter Gd = 3'b011 ;
    parameter Oa = 3'b100 ;
    parameter Ob = 3'b101 ;
    parameter Oc = 3'b110 ;
    parameter Od = 3'b111 ;

    input clk,rst_n;
    input [1:0] Sa,Sb,Sc,Sd;  // traffic sensors
    input [4:0] counter_value;   // internal counter value
    output reg[2:0] Ta,Tb,Tc,Td;    // traffic lights  001 ->green, 010 -> orange, 100 -> red
    output load_counter;    // load enable
    output [4:0] load_value; // value to be loaded in the counter after a trasition, 30 -> green, 3 -> orange


    reg [2:0] current_state, next_state;

    //---

    // current state logic

    always @(posedge clk or negedge rst_n) begin

        if(!rst_n) begin
            current_state <= Ga;
        end
        else begin
            current_state <= next_state;
        end

    end
```

```
    // Next state logic

    always @(*) begin
        case (current_state)
            Ga: begin
                if (((Sa>Sb)&&(Sa>Sc)&&(Sa>Sd)) || counter_value !=1) begin // A is the highest priority or counter isn't finished
                    next_state <= Ga;      // remain in the same state
                end
                else begin
                    next_state <= Oa;
                end   // move to orange state in preparation to stop the traffic of this side
                end

            Gb: begin
                if (((Sb>Sa)&&(Sb>Sc)&&(Sb>Sd)) || counter_value !=1) begin // B is the highest priority or counter isn't finished
                    next_state <= Gb;      // remain in the same state
                end
                else begin
                    next_state <= Ob;
                end   // move to orange state in preparation to stop the traffic of this side
                end

            Gc: begin
                if (((Sc>Sa)&&(Sc>Sb)&&(Sc>Sd)) || counter_value !=1) begin // C is the highest priority or counter isn't finished
                    next_state <= Gc;      // remain in the same state
                end
                else begin
                    next_state <= Oc;
                end   // move to orange state in preparation to stop the traffic of this side
                end

            Gd: begin
                if (((Sd>Sa)&&(Sd>Sb)&&(Sd>Sc)) || counter_value !=1) begin // D is the highest priority or counter isn't finished
                    next_state <= Gd;      // remain in the same state
                end
                else begin
                    next_state <= Od;
                end    // move to orange state in preparation to stop the traffic of this side
                end

```

```
            Oa: begin
                if (counter_value != 1) begin
                    next_state <= Oa; // Stay in orange if counter isn't done
                end else begin
                    // Determine next state based on traffic priorities
                    if ((Sb >= Sa) && (Sb >= Sc) && (Sb >= Sd))
                        next_state <= Gb;
                    else if ((Sc >= Sa) && (Sc >= Sb) && (Sc >= Sd))
                        next_state <= Gc;
                    else
                        next_state <= Gd; // Default to Gb
                end
            end

            Ob: begin
                if (counter_value != 1)  // Stay in orange if counter isn't done
                    next_state <= Ob;
                else begin
                    // Determine next state based on traffic priorities
                    if ((Sc >= Sa) && (Sc >= Sb) && (Sc >= Sd))
                        next_state <= Gc;
                    else if ((Sd >= Sa) && (Sd >= Sb) && (Sd >= Sc))
                        next_state <= Gd;
                    else
                        next_state <= Ga; // Default to Gd
                end
            end

            Oc: begin
                if (counter_value !=1)  // Stay in orange if counter isn't done
                    next_state <= Oc;
                else begin
                    // Determine next state based on traffic priorities
                    if ((Sd >= Sa) && (Sd >= Sb) && (Sd >= Sc))
                        next_state <= Gd;
                    else if ((Sa >= Sb) && (Sa >= Sc) && (Sa >= Sd))
                        next_state <= Ga;
                    else
                        next_state <= Gb; // Default to Gd
                end
```

```
            Od: begin
                if (counter_value !=1)  // Stay in orange if counter isn't done
                    next_state = Od;
                else begin
                    // Determine next state based on traffic priorities
                    if ((Sa >= Sb) && (Sa >= Sc) && (Sa >= Sd))
                        next_state = Ga;
                    else if ((Sb >= Sa) && (Sb >= Sc) && (Sa >= Sd))
                        next_state = Gb;
                    else
                        next_state = Gc; // Default to Ga
                end
            end
        endcase
    end

    // output logic
    always @(current_state) begin
        case (current_state)
            Ga: begin
                Ta <= 3'b001;    // green
                Tb <= 3'b100;    // red
                Tc <= 3'b100;    // red
                Td <= 3'b100;    // red
            end

            Gb: begin
                Tb = 3'b001;    // green
                Ta = 3'b100;    // red
                Tc = 3'b100;    // red
                Td = 3'b100;    // red
            end

            Gc: begin
                Tc = 3'b001;    // green
                Ta = 3'b100;    // red
                Tb = 3'b100;    // red
                Td = 3'b100;    // red
            end
```

```
            Gd: begin
                Td = 3'b001;    // green
                Ta = 3'b100;    // red
                Tb = 3'b100;    // red
                Tc = 3'b100;    // red
            end

            Oa: begin
                Ta = 3'b010;    // orange
                Tb = 3'b100;    // red
                Tc = 3'b100;    // red
                Td = 3'b100;    // red
            end

            Ob: begin
                Tb = 3'b010;    // orange
                Ta = 3'b100;    // red
                Tc = 3'b100;    // red
                Td = 3'b100;    // red
            end

            Oc: begin
                Tc = 3'b010;    // orange
                Ta = 3'b100;    // red
                Tb = 3'b100;    // red
                Td = 3'b100;    // red
            end

            Od: begin
                Td = 3'b010;    // orange
                Ta = 3'b100;    // red
                Tb = 3'b100;    // red
                Tc = 3'b100;    // red
            end

            default: begin
                Ta = 3'b100;    // red
                Tb = 3'b100;    // red
                Tc = 3'b100;    // red
                Td = 3'b100;    // red
            end
        endcase
    end
    assign load_counter = (current_state !== next_state);
    assign load_value = (next_state > 3 ? 3 : 30);

endmodule
```

- **Counter:**

```verilog
module counter(clk,rst_n,load,data,count);

    input clk,load,rst_n;
    input [4:0] data;
    output reg [4:0] count;

    always@(posedge clk,negedge rst_n)
    begin
        if (!rst_n) begin
            count <= 31;
        end
        else begin
            if(load)
            count <= data;
        else
            count <= count - 1;
        end

    end
endmodule        You, last week • Next state logic
```

- **Top Module:**

```verilog
module Topmodule (
    clk,rst_n,
    Sa,Sb,Sc,Sd,
    Ta,Tb,Tc,Td
);

    input clk,rst_n;
    input [1:0] Sa,Sb,Sc,Sd;
    output [2:0]Ta,Tb,Tc,Td;

    wire [4:0] counter_output;   // internal counter current value
    wire [4:0] load_value;       // value to be loaded to the counter
    wire load_en;                // load enable

    wire [1:0] Sa_reading;       You, last week • .
    wire [1:0] Sb_reading;
    wire [1:0] Sc_reading;
    wire [1:0] Sd_reading;

    Traffic_Controller FSM (.Sa(Sa_reading),.Sb(Sb_reading),.Sc(Sc_reading),.Sd(Sd_reading),.clk(clk),.rst_n(rst_n)
    ,.counter_value(counter_output),.Ta(Ta),.Tb(Tb),.Tc(Tc),.Td(Td),.load_counter(load_en),.load_value(load_value));

    counter internal_counter (.clk(clk),.load(load_en),.data(load_value),.count(counter_output),.rst_n(rst_n));

    assign Sa_reading = Sa[0] + Sa[1];
    assign Sb_reading = Sb[0] + Sb[1];
    assign Sc_reading = Sc[0] + Sc[1];
    assign Sd_reading = Sd[0] + Sd[1];

endmodule
```

❖ Verification:

- Test Strategy for Traffic Light Controller Test bench:

Test Scenario	Objective	Inputs (Stimuli)	Expected Outputs/Behaviour
Reset Functionality	Ensure all traffic lights reset to a default state when rst_n = 0.	rst_n = 0	All traffic lights switch to the reset state (e.g., Red for all directions).
Round-Robin Scheduling	Verify fair and cyclic light transitions in all directions (no starvation).	Sa = 1, Sb = 1, Sc = 1, Sd = 1 (or similar priority inputs).	Traffic lights transition in a round-robin manner: one direction turns Green, and others remain Red.
Sensor-Based Adjustments	Confirm light timings adjust dynamically based on sensor signals (at positions 1 and 5).	Sensor at a specific lane (e.g., Sa = 3, others = 0).	Extended Green for the lane with higher priority sensor values, other lights maintain Red.
Fixed Priority Testing	Test priority-based light handling to ensure correct precedence among directions.	Assign higher priority (Sd = 3) while others are lower.	Priority lane (Sd) maintains Green until completion, other lanes wait (Red).
Dynamic Priority Changes	Check behaviour when priorities change dynamically during operation.	Vary priorities in real-time (e.g., Sa = 2, Sd = 3, etc.).	The system adapts to new priorities, transitioning traffic lights accordingly.
Edge Cases	Test system with all lanes inactive or all lanes highly active.	- Case 1: Sa = Sb = Sc = Sd = 0. - Case 2: Sa = Sb = Sc = Sd = 3.	- Case 1: All directions should remain Red. - Case 2: Ensure non-conflicting Green transitions or round-robin behaviour.
Conflicting Paths Prevention	Ensure system avoids intersecting traffic paths as per design.	Simulate conflicting paths using inputs representing multiple active lanes.	Lights for conflicting paths are never simultaneously Green.
Clock Dependency	Validate output transitions occur only on the negative clock edge (negedge clk).	Observe behaviour during clock transitions.	Outputs (lights) update only on the negative edge of clk.

Long Simulation Runs	Check system stability and robustness over extended simulation time.	Vary input patterns over hundreds of cycles.	The system continues to function correctly under long-term scenarios (no deadlocks or undefined states).
Reset Recovery	Confirm the system recovers seamlessly after coming out of reset (rst_n = 1).	rst_n = 1 after being held low.	Lights resume normal operation following the pre-defined scheduling or priority rules.

- Test bench:

```
module asic_tb ();

reg clk,rst_n;
reg [1:0] Sa,Sb,Sc,Sd;
wire [2:0] Ta,Tb,Tc,Td;

integer i;
initial begin
    clk =1;
    forever begin
        #1 clk = ~clk;
    end
end

Topmodule t1 (clk,rst_n,
    Sa,Sb,Sc,Sd,
    Ta,Tb,Tc,Td);

initial begin
    rst_n = 0;
    @(negedge clk);
    rst_n = 1;

```

```
//// Round Robin ////

// Test Case 1: All inputs are 1
Sa = 1; Sb = 1; Sc = 1; Sd = 1;
for (i = 0; i < 150; i = i + 1) begin
    @(negedge clk);
end
$display("After Test Case 1: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);

// Test Case 2: All inputs are 2
Sa = 2; Sb = 2; Sc = 2; Sd = 2;
for (i = 0; i < 150; i = i + 1) begin
    @(negedge clk);
end
$display("After Test Case 2: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);

// Test Case 3: All inputs are 3
Sa = 3; Sb = 3; Sc = 3; Sd = 3;
for (i = 0; i < 150; i = i + 1) begin
    @(negedge clk);
end
$display("After Test Case 3: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);

// Test Case 4: All inputs are 0
Sa = 0; Sb = 0; Sc = 0; Sd = 0;
for (i = 0; i < 150; i = i + 1) begin
    @(negedge clk);
end
$display("After Test Case 4: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);
```

```
# After Test Case 1: Sa=1, Sb=1, Sc=1, Sd=1 => Ta=1, Tb=4, Tc=4, Td=4
# After Test Case 2: Sa=2, Sb=2, Sc=2, Sd=2 => Ta=4, Tb=1, Tc=4, Td=4
# After Test Case 3: Sa=3, Sb=3, Sc=3, Sd=3 => Ta=4, Tb=1, Tc=4, Td=4
# After Test Case 4: Sa=0, Sb=0, Sc=0, Sd=0 => Ta=4, Tb=4, Tc=1, Td=4
```

```
//// Special Round Robin ////


// Stay at A
Sa = 3;
for ( i = 0; i<150; i=i+1) begin
    @(negedge clk);
end
$display("Special Round Robin Test 1: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);

// Round Robin between A & B
Sb = 3;
for ( i = 0; i<150; i=i+1) begin
    @(negedge clk);
end
$display("Special Round Robin Test 2: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);

// Round Robin between A & B & C
Sc = 3;
for ( i = 0; i<150; i=i+1) begin
    @(negedge clk);
end
$display("Special Round Robin Test 3: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
         Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);
```

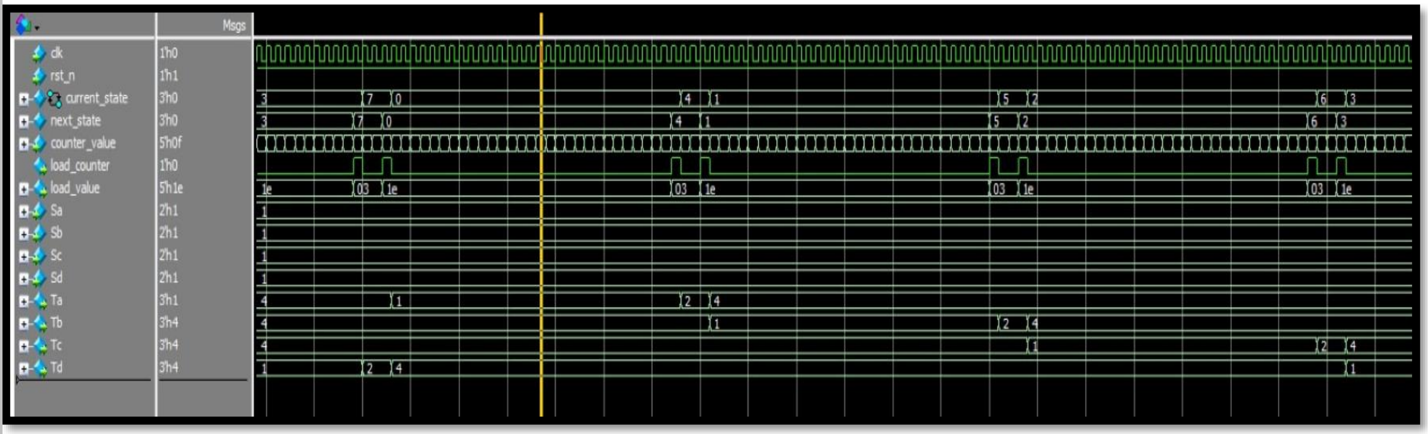
```
# Special Round Robin Test 1: Sa=3, Sb=0, Sc=0, Sd=0 => Ta=1, Tb=4, Tc=4, Td=4
# Special Round Robin Test 2: Sa=3, Sb=3, Sc=0, Sd=0 => Ta=4, Tb=1, Tc=4, Td=4
# Special Round Robin Test 3: Sa=3, Sb=3, Sc=3, Sd=0 => Ta=4, Tb=4, Tc=1, Td=4
```

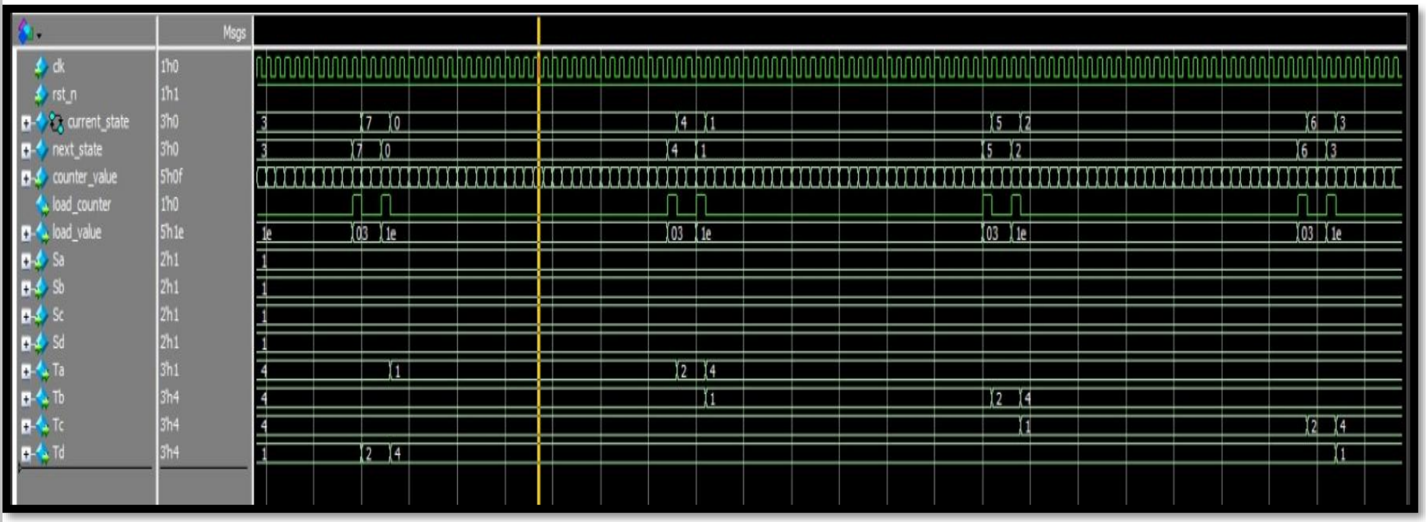
```verilog
//// Priority Given ////

//to D
Sd = 3; Sa =1;Sb =1;Sc =1;
for ( i = 0; i<65; i=i+1) begin
    @(negedge clk);
end

$display("Priority Test 1: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
            Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);


//To C
Sc = 3; Sa =1;Sb =1;Sd =1;
for ( i = 0; i<65; i=i+1) begin
    @(negedge clk);
end

$display("Priority Test 2: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
            Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);


//To B
Sb = 3; Sd = 1; Sc = 1; Sa = 1;
for (i = 0; i < 65; i = i + 1) begin
    @(negedge clk);
end
$display("Priority Test 3: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
            Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);

//To A
Sa = 3; Sd = 1; Sb = 1; Sc = 1;
for (i = 0; i < 65; i = i + 1) begin
    @(negedge clk);
end
$display("Priority Test 4: Sa=%0d, Sb=%0d, Sc=%0d, Sd=%0d => Ta=%0d, Tb=%0d, Tc=%0d, Td=%0d",
            Sa, Sb, Sc, Sd, Ta, Tb, Tc, Td);
```

```
# Priority Test 1: Sa=1, Sb=1, Sc=1, Sd=3 => Ta=4, Tb=4, Tc=4, Td=1
# Priority Test 2: Sa=1, Sb=1, Sc=3, Sd=1 => Ta=4, Tb=4, Tc=1, Td=4
# Priority Test 3: Sa=1, Sb=3, Sc=1, Sd=1 => Ta=4, Tb=1, Tc=4, Td=4
# Priority Test 4: Sa=3, Sb=1, Sc=1, Sd=1 => Ta=1, Tb=4, Tc=4, Td=4
# Test Case: Sa=2, Sb=1, Sc=0, Sd=3 => Ta=4, Tb=4, Tc=4, Td=1
# ** Note: $stop    : testbench.v(140)
#    Time: 2751 ns  Iteration: 1  Instance: /asic_tb
```

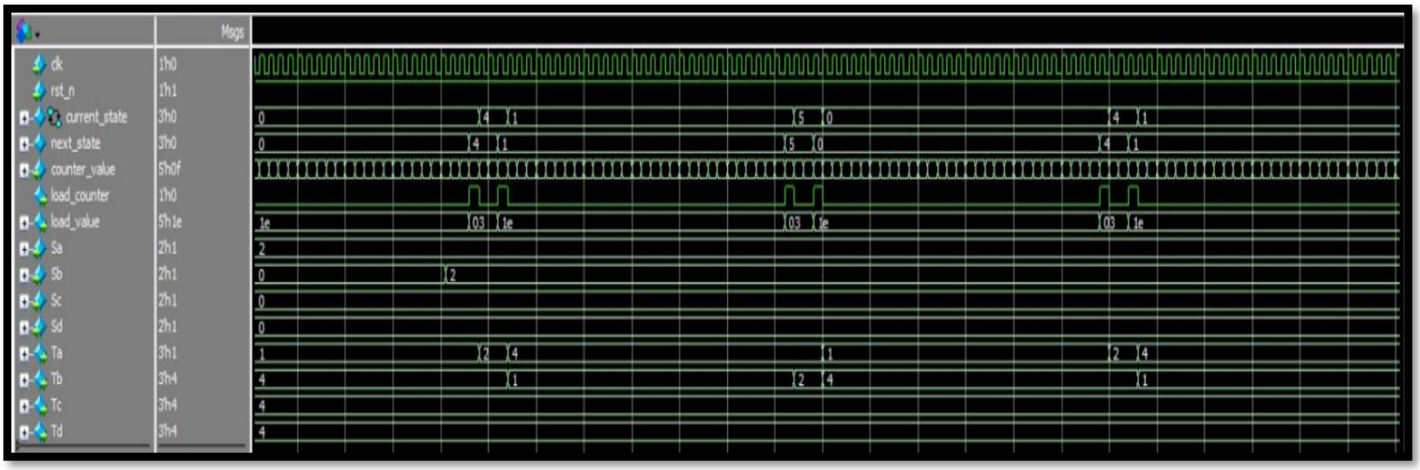
- **Waveforms:**

1- Round robin (Equal traffic):

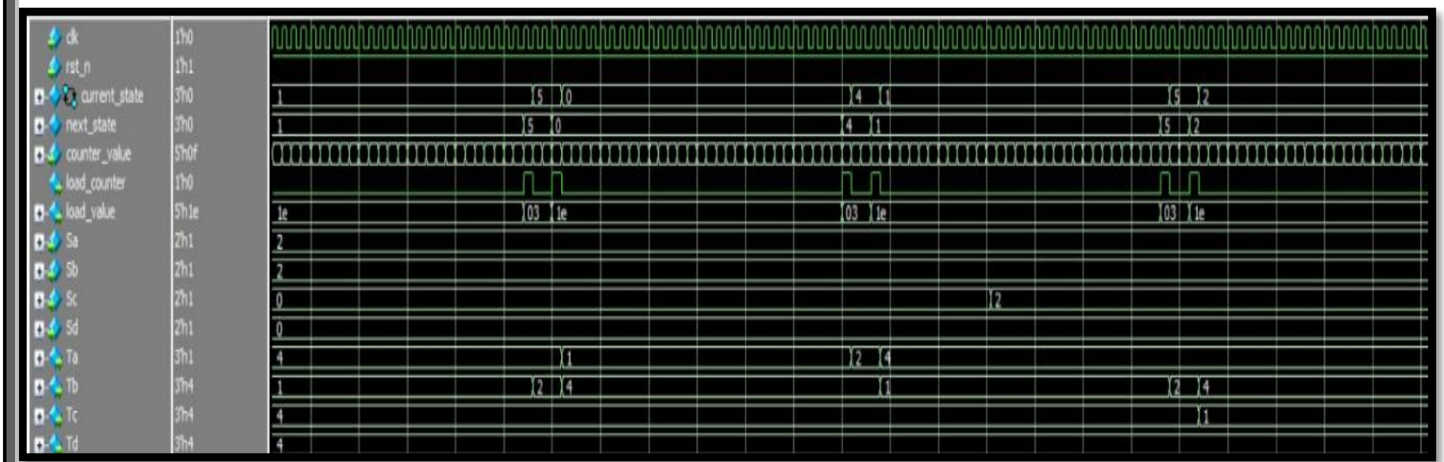


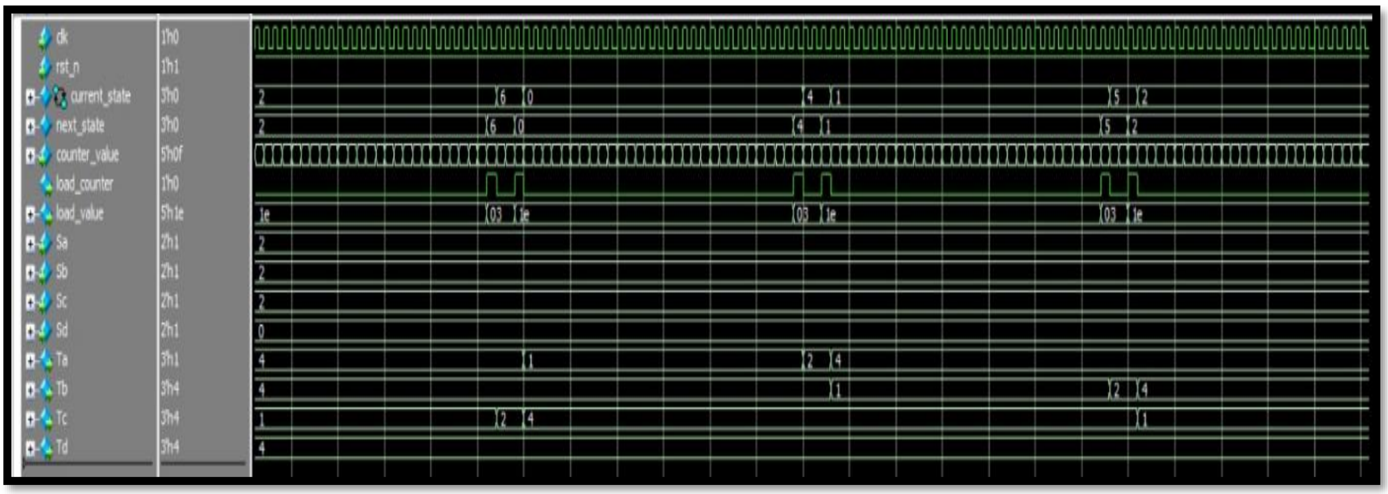


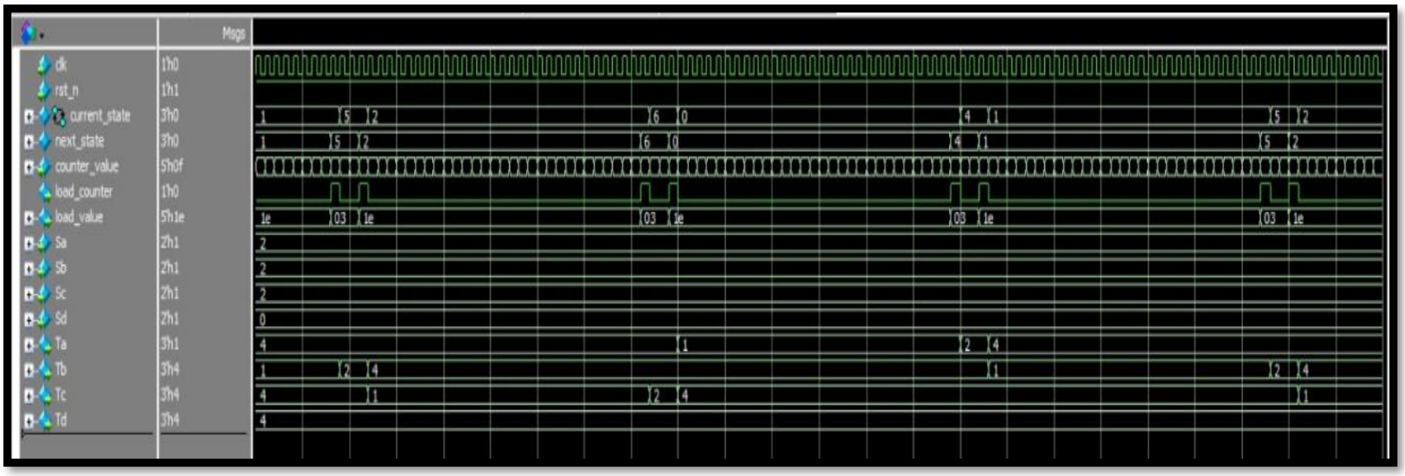
2- Round robin (Equal traffic between a&b):



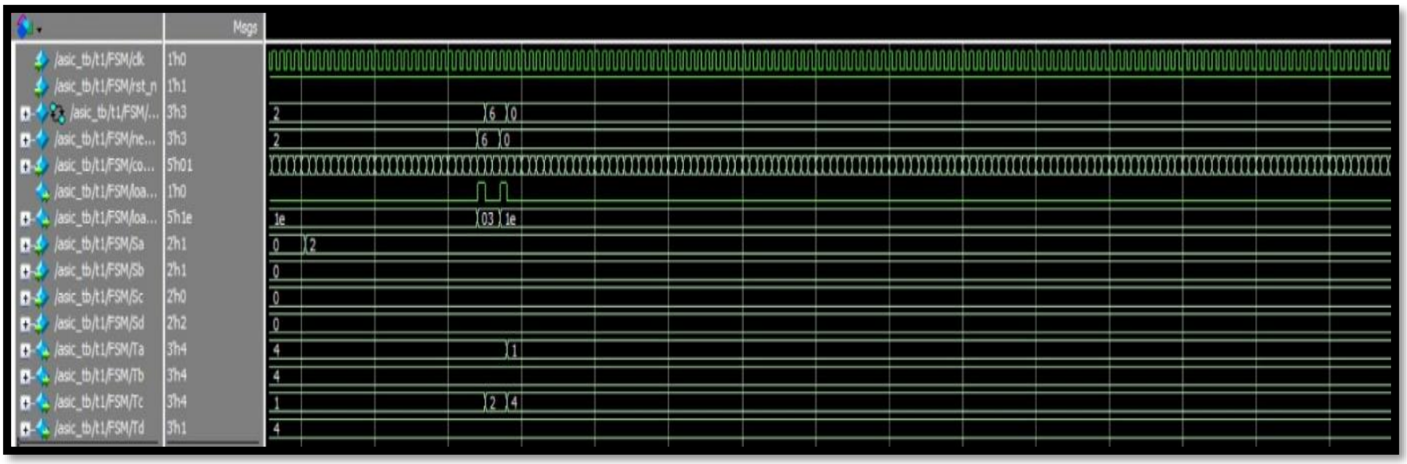
3- Round robin (Equal traffic between a&b&c):



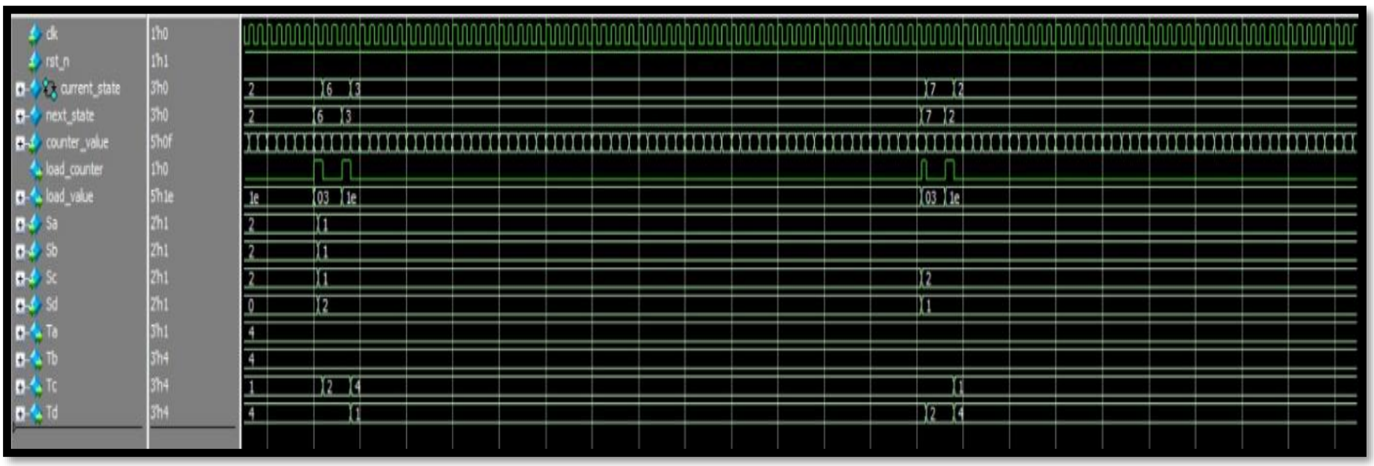




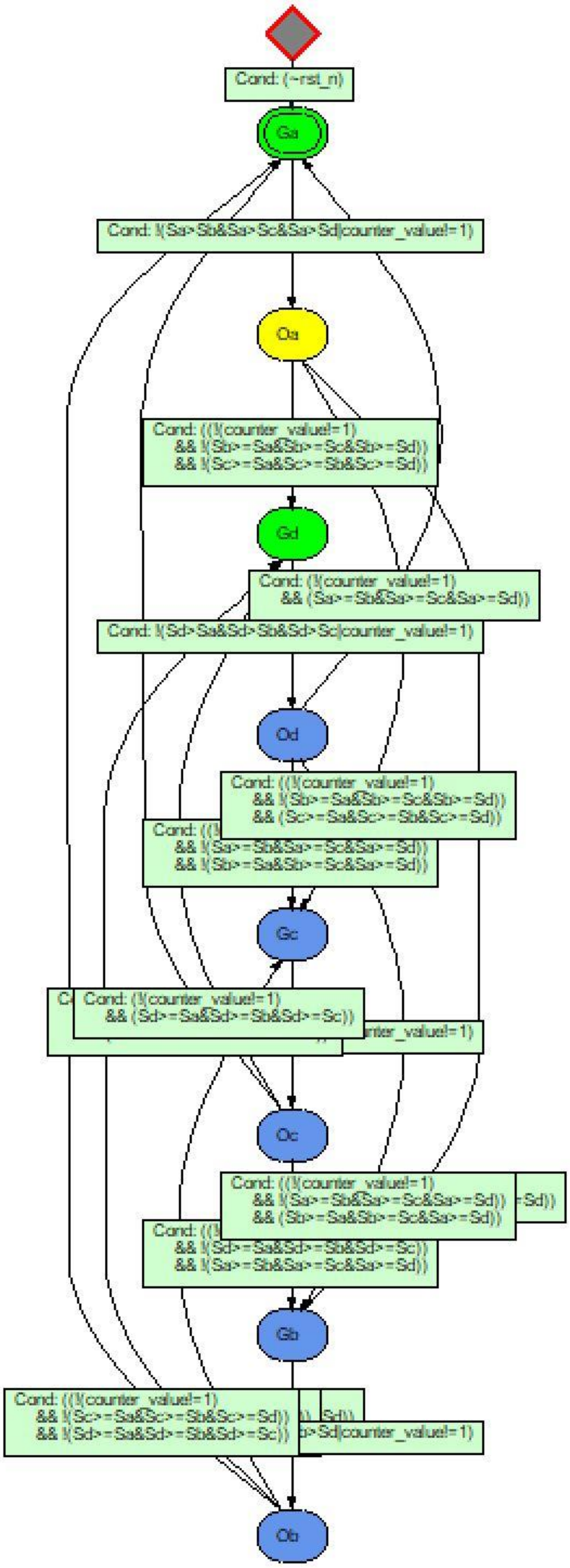
4- Stay at A:



5- Priority given to d, stay at c:



❖ FSM diagram using Questasim:



❖ Contribution Table:

Task	Name
FSM Diagram structuring	Fares , Youssef Osama , Ahmed, Youssef Hany
Design RTL coding	Fares , Youssef Osama
Test strategy, Test bench	Ahmed , Youssef Hany