Activity 04: Finite State Machines: Traffic Light Controller

Objective

Understand how finite state machines are implemented in Verilog. Implement an FSM to control traffic lights at an intersection.

Theory

Finite state machine

State Machines are one of the most common building blocks in modern digital systems. They handle everything from communications handshaking protocols to microprocessor bus wait state insertion. State machines operate at hardware speeds where software cannot compete. All too often engineers take an ad-hoc approach to state machine design.

State Machine is a flowchart like graphical notation that describes the cycle by cycle operations of an algorithm and its each step takes one clock cycle. It describes behavior rather than structure and provides a mechanism for performing systematic step-by-step design. They can be directly translated to Verilog code and used to design synchronous sequential circuits.

There are two generally accepted architectures for Synchronous State Machines. The first type considered is a state machine in which the outputs depend only on the current state. This is commonly known as a Moore machine.

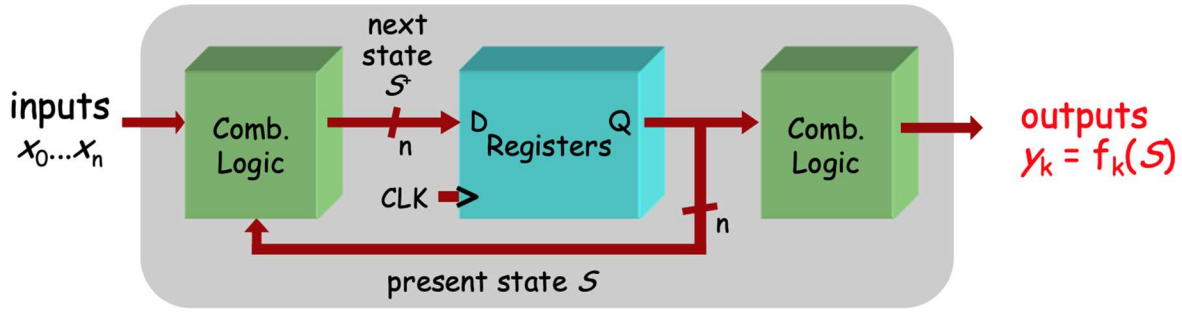

Figure 1. Moore state machine

In the second type, the outputs depend on both the current state and the input variables. This is known as a Mealy Machine

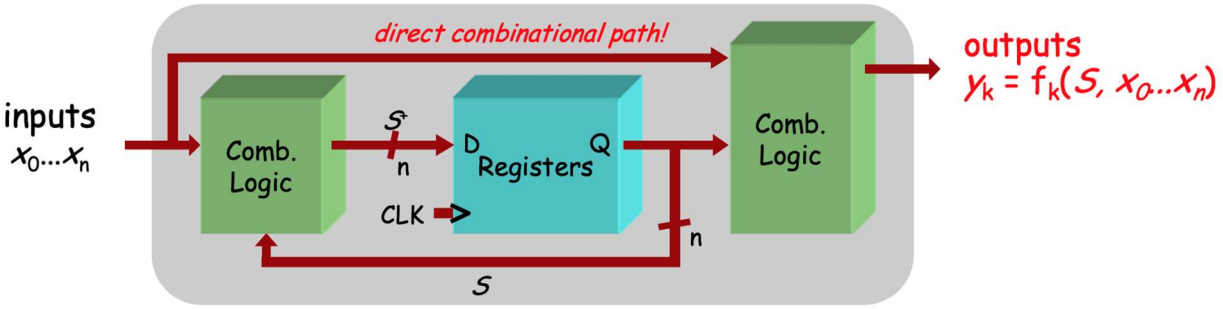

Figure 2. Mealy state machine

Traffic light control

Consider a simplified version of the intersection of Kashmir Highway and Golra Road as shown in Figure 3. Note that the traffic flows only in one direction of any of the roads. There is also a rail track that runs parallel to Kashmir Highway and crosses Golra Road. There are two traffic signals to regulate the flow of traffic. A train sensor can warn about an incoming train 30 seconds prior to its arrival and the sensor output goes off 10 seconds after the train has passed.

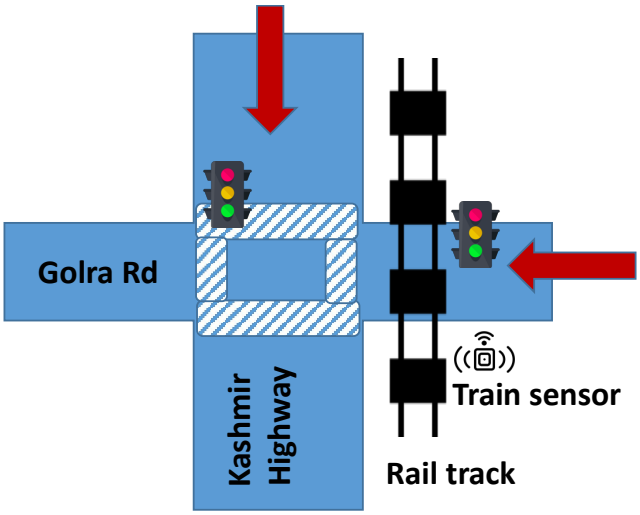

Figure 3. A simplified view of a Traffic Intersection

Activity 4a: Extract FSM from given Verilog code

Below is Verilog code implementing FSM for traffic light control at the intersection shown in Figure 3. Your objective is to read the code and understand it. Extract the state machine diagram that the code implements.

Also simulate the code using Xilinx ISE and verify that the simulation results match the behavior expected from your FSM diagram.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////
module TrafficLight(
        ////////////////////////////// inputs
                input clk,
              input rst,
        ////////////////////////////// outputs
              output reg [2:0] LED_Kashmir, LED_Golra);


////////////////////
/*Assume
      100 = Red
      010 = Yellow
      001 = Green
 */

parameter RED = 3'b100, GREEN=3'b001, YELLOW=3'b010;

// Kashmir Highway (KH), Golra (GL)
parameter S0 = 3'd0, // KH: green  ,     GL: red
          S1 = 3'd1,       // KH: yellow , GL: red
```

```
            S2 = 3'd2,       // KH: red    , GL: red
            S3 = 3'd3,       // KH: red    , GL: green
            S4 = 3'd4,       // KH: red    , GL: yellow
            S5 = 3'd5;       // KH: red    , GL: red
//Then the cycle repeats

////////////////////////////////////
reg [2:0] state, next_state;









////////////// counter

reg [3:0] count;
reg rst_counter;

//////////////////////////////////////
always@(posedge clk or posedge rst)
begin
```

```
         if(rst || rst_counter)
           count <= 4'b0000;
         else
             count <= count+1;
end
////////////////////////////////////
// state register
always@(posedge clk or posedge rst)
begin
         if(rst)
           state <= S0;
         else
             state <= next_state;
end
////////////////////////////////////
// Next state logic
always@(*)
begin
   next_state = state;
         rst_counter =0;
case(state)
S0: begin
   if(count == 4'd14) begin next_state = S1; rst_counter =1; end
end


S1: begin
```

```
   if(count == 4'd4) begin next_state = S2; rst_counter =1; end
end

S2: begin
   if(count == 4'd4) begin next_state = S3; rst_counter =1; end
end

S3: begin
   if(count == 4'd14) begin next_state = S4; rst_counter =1; end
end
S4: begin
   if(count == 4'd4) begin next_state = S5; rst_counter =1; end
end
S5: begin
if(count == 4'd4) begin next_state = S0; rst_counter =1; end
end
endcase
end
//////////////////////////////////////

// Output logic

always@(*)
begin
case(state)
S0: begin LED_Kashmir = GREEN;   LED_Golra = RED; end
```

```
S1: begin LED_Kashmir = YELLOW;  LED_Golra = RED; end

S2: begin LED_Kashmir = RED;     LED_Golra = RED; end

S3: begin LED_Kashmir = RED;     LED_Golra = GREEN; end

S4: begin LED_Kashmir = RED;     LED_Golra = YELLOW; end

S5: begin LED_Kashmir = RED;     LED_Golra = RED; end

endcase

end


endmodule
```

```
`timescale 1ns / 1ps

module Traffic_tb;

        // Inputs

        reg clk;

        reg rst;


        // Outputs

        wire [2:0] LED_Kashmir;

        wire [2:0] LED_Golra;


        // Instantiate the Unit Under Test (UUT)

        TrafficLight uut (

           .clk(clk),

           .rst(rst),

           .LED_Kashmir(LED_Kashmir),

           .LED_Golra(LED_Golra)
```

```
        );
initial begin
        clk = 0;
        forever #5 clk = ~clk;
end


initial begin
                        rst = 1;


        #10 rst = 0;


        #600;
end


endmodule
```

Questions

Answer the following questions.

Q1. Which type of FSM does the code implement? Mealy or Moore?

Q2. How many flip-flops are needed to implement the state register?

Q3. How many always blocks are used for FSM?

Q4. Consider the FSM and the counter as two separate hardware modules. Draw a block diagram showing how the FSM connects with the counter.

Lab Task 2: Extend the FSM and implement in Verilog (5 marks)

The FSM used in Task 1 does not take into account the Train Sensor. Extend the FSM to include the sensor operation. When the train comes, traffic on Golra Rd must stop, whereas, traffic on Kashmir Highway may flow. Recall that the train sensor warns about an incoming train 30 seconds prior to its arrival and the sensor output goes off 10 seconds after the train has passed.

Draw the extended FSM, implement it in Verilog, and simulate using a Verilog simulator. Include waveforms in your lab report.