# Lab 4 – State Machines

ENGIN 341 – Advanced Digital Design
University of Massachusetts Boston

## Overview

In this lab students will use State Machines to design a Traffic Light Controller for an intersection with a main street, a side street, and a pedestrian crossing. The resulting output will be displayed using a series of LEDs on a proto-board connected via ZYBO PMOD ports.

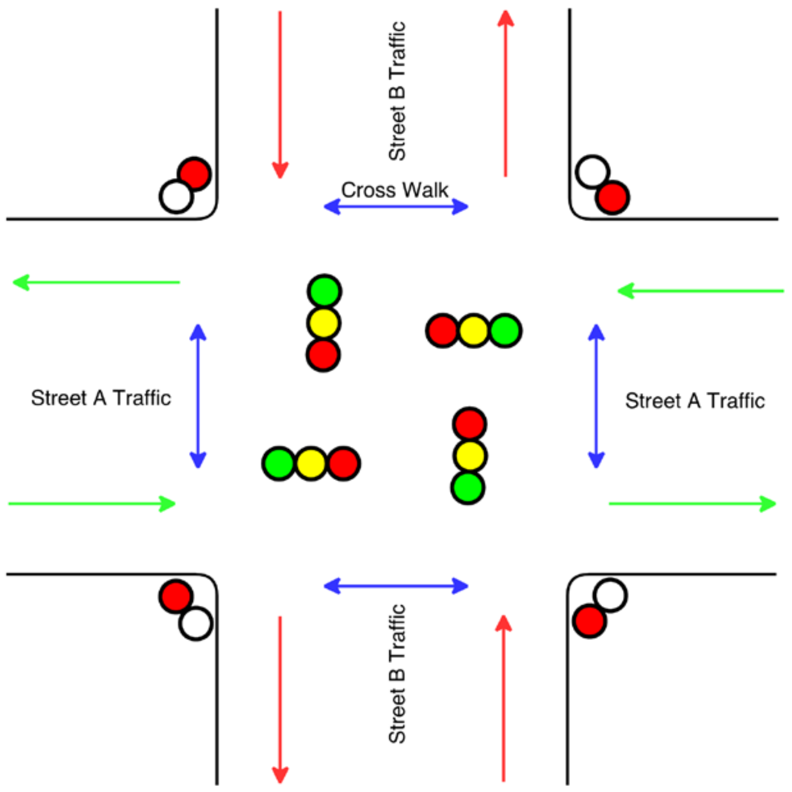

*Figure 1. Traffic Light Diagram*

Traffic Light A should consist of 3 lights: Green (Ga), Yellow (Ya), and Red (Ra)
Traffic Light B should consist of 3 lights: Green (Gb), Yellow (Yb), and Red (Rb)
The Pedestrian Crossing should consist of 2 lights: White (Ww) and Red (Rw)

**NOTE FOR 2021:** *You do NOT need to submit a report for this lab. You only need to build and implement the design. Your grade will be based on the lab demo (in class) and your submitted project code.*

The normal repeating sequence of operation for each of the three lights should be:

*Table 1. Traffic Light Sequences, When one light is Green,*

*Yellow, or White, the others must be Red.*

| **Street A** | **Street B** | **Pedestrian** |
|---|---|---|
| Green | Red | Red |
| Yellow | Red | Red |
| Red | Green | Red |
| Red | Yellow | Red |
| Red | Red | White |
| Red | Red | Red |

The timing of this sequence will be:

*Table 2. Traffic Light Sequence Timing*

| **Street A** | **Street B** | **Pedestrian** |
|---|---|---|
| Green – 4 sec | Green – 3 sec | White – 2 sec |
| Yellow – 2 sec | Yellow – 1 sec | Red – Flashes 4 seconds at 1Hz, then solid 10 sec |
| Red – 10 sec | Red – 12 sec | |

A *Maintenance Mode* will also be implemented. When *Maintenance Mode* is active, all three lights (R, Y, G) for each traffic light (A, B, P) should flash at 1Hz. When *Maintenance Mode* is switched off, all lights should reset to the starting mode of (Ga, Rb, Rw).

**Extra Credit:** Modify your design so that the Pedestrian Crossing only occurs when a "walk" button is pressed. In this case, the main sequence should only cycle through the sequence below when operating as normal. You will use the onboard pushbuttons to determine when walk has been pressed and released. Once this occurs, enable an on-board LED to indicate that the Pedestrian crossing is enabled. At the end of the next sequence, the walk sequence described in Tables 1 and 2 should occur. After the walk sequence, the on-board LED should turn off and the sequence should return to the sequence below.

| **Street A** | **Street B** | **Pedestrian** |
|---|---|---|
| Green | Red | Red |
| Yellow | Red | Red |
| Red | Green | Red |
| Red | Yellow | Red |

**Lab Work**

1. Design a state graph for the Traffic Light Controller, then convert this to a State Machine Chart. State transitions will automatically occur after the specified delay time.
   For a review of State Graphs and State Machine Charts, refer to Chapter 5, Section 5.1 in your text book.
2. Write a VHDL file that implements your state graph.
   a. Your file should have 3 inputs:
      i. Clk
      ii. Rst – Reset should return your traffic lights to the initial state (Ga, Rb, Rw)
      iii. Mode – When Mode = 1, your Traffic Controller should enter Maintenance Mode, When Mode is switched to '0', your Traffic Controller should reset to the initial state (Ga, Rb, Rw)
   b. Your design should have 8 outputs:
      i. StreetA (3 bits)
         1. Ga – Green light for traffic light A
         2. Ya – Yellow light for traffic light A
         3. Ra – Red light for traffic light A
      ii. StreetB (3 bits)
         1. Gb – Green light for traffic light B
         2. Yb – Yellow light for traffic light B
         3. Rr – Red light for traffic light B
      iii. Pedestrian (2 bits)
         1. Ww – White light for walk sign/pedestrian crossing light
         2. Rw – Red light for walk sign/pedestrian crossing light
3. Write a top level Lab4 module that connects your Traffic Controller, generic clock divider, and any other modules you choose to implement.
4. Design the constraints file using the I/O Planning tool.
5. Construct the Traffic Light circuit using the provided LEDs, resistor arrays, and a breadboard. Connect the circuit to a ZYBO PMOD using jumpers/wires, according to your constraints. Make sure your lab report includes a detailed wire list and circuit implementation diagram for your constructed circuit.

**Deliverables**

Your Lab 4 project directory, containing all sources, simulation waveforms, bit files and Lab 4 Report, are to be turned in as a zip file labeled "*ENGIN341_LAB4_LastName_FirstName.zip*".

## Part 1. State Graphs

You have been given some room for creativity with this lab. The main requirements are that the "Traffic Lights" your program controls must follow the sequence given with the specified timing, and it must implement a maintenance mode.

With this in mind, you can approach the problem in a couple of different ways. You could have a single large state machine that controls all three sets of lights, or a state machine for each set of lights. Whichever direction you go in, make sure that it is reflected in your FSM graph, and then your ASM chart. A simple example of a state graph and its state machine chart equivalent are shown below in Figures 2 and 3.

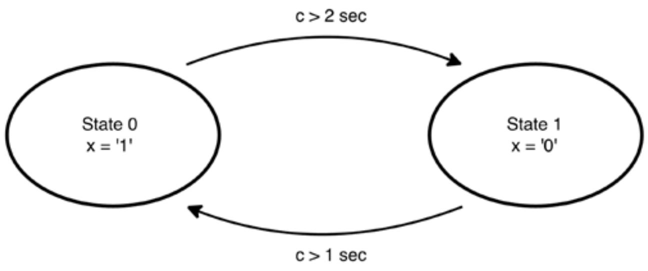

*Figure 2. A simple Finite State Machine (FSM) Graph.*

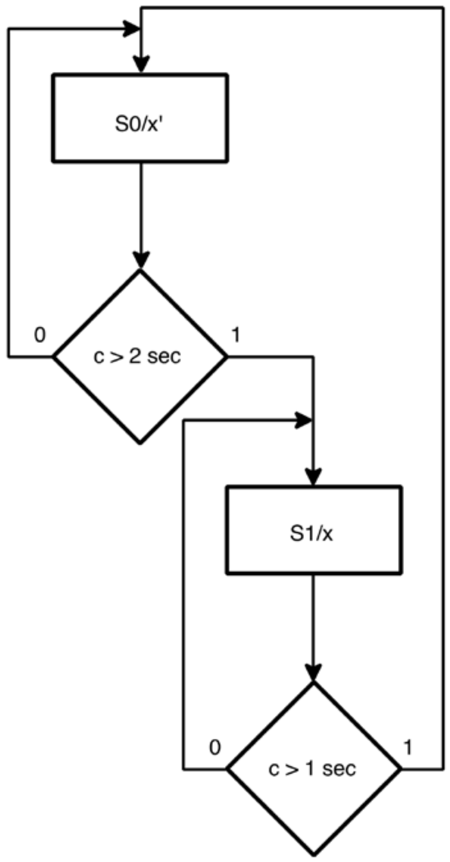

*Figure 3. An Algorithmic State Machine (ASM) Diagram Equivalent of the FSM in Figure 1.*

## Part 2. State Machines in VHDL

In VHDL, State Machines are implemented inside processes sensitive to a clock and/or a reset signal, using a combination of 'if' and 'case' statements. To the right is an example of a VHDL implementation of the simple State Machine shown in Figures 1 and 2.

```
architecture states of state_machine is
    TYPE statetype is (S0, S1);
    SIGNAL state : statetype;
    SIGNAL c : natural;

    begin
    FSM: process(clk)
    begin
        if clk'event and clk = '1' then
            case state is
                when S0 =>
                    if c > 2 then
                        state <= S1;
                        c <= 0;
                    else
                        state <= S0;
                        c <= c + 1;
                    end if;
                when S1 =>
                    if c > 1 then
                        state <= S0;
                        c <= 0;
                    else
                        state <= S1;
                        c <= c + 1;
                    end if;
            end case;
        end if;
    end process;

    X_Out: process(state)
    begin
        case state is
            when S0 => x <= '1';
            when S1 => x <= '0';
        end case;
    end process;
end states;
```

The first line to take note of is the "*TYPE statetype is (S0, S1);*". This is a *TYPE declaration*, and its purpose is to create a new enumerated data type. In this case, we have created a new data type called *statetype*, which only has two enumerated values, S0 and S1, equivalent to the integer values of 0 and 1 respectively. Any signals/variables that are declared as *statetype* can only assume one of the two (S0 or S1) enumerated values. For example, on the next line, is a signal declaration, "*SIGNAL state : statetype;". We have now created an internal signal of type *statetype*, which can change its value between S0 and S1. The *state* signal is a state register, because it only changes on a clock edge, and is used to store the present state value. The next state value of the *state* register is determined within the case statement inside the *FSM* process, based on conditional statements encapsulated by if/elsif/else statements.

This method of state encoding using enumerated data types is called enumerated state encoding. Rather than explicitly encoding each state to a series of bits, which yields hard-to-read VHDL code, especially for large FSMs, using enumerated state encoding allows Vivado's *Synthesis* tool to set the actual state bit encodings according to the synthesis options at synthesis time.

The *X_Out* process (sensitive only to the *state* signal, which in turn changes only on a rising clock edge) defines the behavior of the output signal *x*. This is an example of a Moore-type state machine. The output of Mealy-type state machines would be sensitive to both the current state and the explicit state machine inputs.

## Part 3. Top Level Design Implementation

Your top level file should, at a minimum, contain modules for your clock divider and traffic controller. You can either directly connect the output of your Traffic Controller module to the output of your Top level file. Or you can implement an LED driver.

## Part 4. Designing the Constraints

When designing your constraints file, make sure to consider which PMOD connector on the ZYBO you will be using. This will affect your pin assignments.

This design contains three inputs. For your *mode* input, you should use SW0 on the ZYBO, and for the reset, you can use one of the onboard buttons.

## Part 5. Connecting to the ZYBO PMOD Ports

You are expected to put together a “Traffic Light” circuit, containing a set of LEDs for Street A, Street B, and the Pedestrian Crossing. This is a simple circuit, but you are also expected to connect this circuit via jumpers/wires to one of the PMOD ports on the ZYBO.

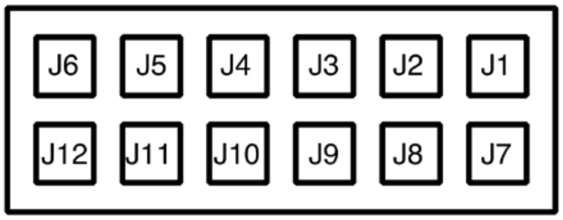

*Figure 4. Port Numbering for PMOD Connectors*

As shown in the ZYBO Reference Manual, a single PMOD connector has 12 ports. The ports you should be using are J1-J4, and J7-J10, as well as either J5 or J11 for your ground. J6 or J12 provide Vcc.

It is important that your circuit wiring and connection to the PMOD ports matches your constraint files. Refer to the ZYBO Reference Manual for the pin numbers of whichever set of PMOD ports you use.