ELE 206/COS 306 Lab 4: Introduction to Finite State Machines

Due Date: October 21st, 2020
Demonstration Required.

Tutorial Reading: Chapter 4 and Section 5.1
Tutorial Review Questions: 4.1, 4.2, 5.1, 5.2, 5.3

Introduction

This lab will cover the basics of designing a finite state machine (FSM) and implementing it in Verilog. You will be given a high-level description of a real-life control problem, and you'll go through the process of designing, implementing, and testing an FSM-based circuit that solves that problem. It is expected that you have completed the tutorial reading and tutorial review questions prior to beginning the lab. Your answers to the review questions should be a part of your write-up that is submitted at the end of the lab.

To begin, download and unzip *lab4.zip* from Blackboard, and follow the instructions in this document.

You may also wish to review the lecture slides or textbook sections covering finite state machines. This is covered in section 3.3 of the textbook. Additionally, section 5.2 of the tutorial walks through a more complex RTL design problem. Although you will not need to know about HLSMs and datapaths for this lab, it may be helpful to read through the parts describing controllers and FSMs.

A Real-Life Problem

For this lab, you will implement a controller for the traffic light at the intersection of Washington Road and Prospect Avenue. A diagram of the intersection is shown in Figure 4.1.

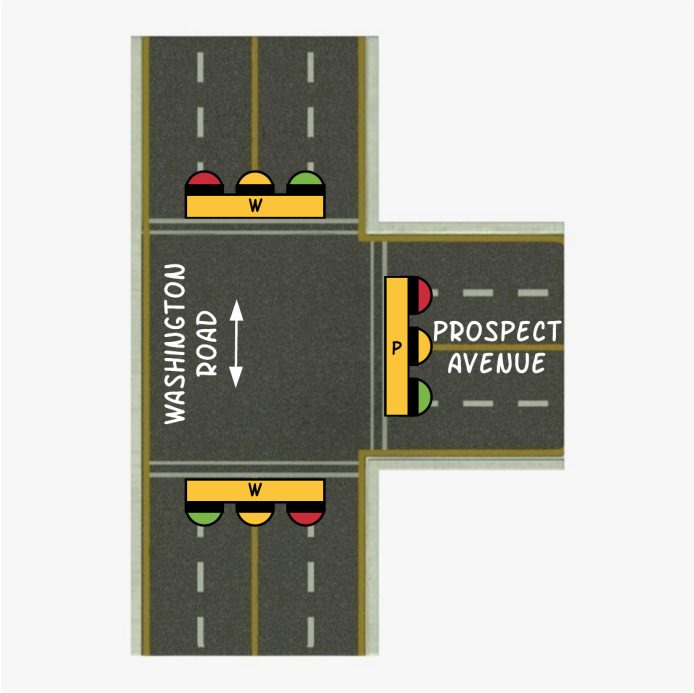

Figure 4.1: Washington-Prospect Intersection

Controller Rules

The rules for the intersection are as follows:

- Cars traveling down Washington Road have the highest priority.
- The duration of a green light for cars on Washington should be *at least* 20 seconds.
- Cars traveling down Prospect Avenue have the lowest priority. However, if there is a car waiting on Prospect, the light should change as soon as

possible *without* violating the 20-second minimum for green lights on Washington. This means if a car arrives on Prospect, and Washington has been green for more than 20 seconds, the light should begin to change immediately.

- The duration of a green light for cars on Prospect should be *exactly* 20 seconds.
- The duration of a yellow light for either road should be exactly 5 seconds.
- Assume normal stoplight patterns. Transitions are from green to yellow to red for stopping, then directly from red to green for continuing. If one direction is green or yellow, the other direction must be red.
- If the controller is reset, it will start off with a green light on Washington Road for at least 20 seconds. The controller can be reset in any state.
- Finally, the controller has a clock with a 5-second period.

If you can think of an ambiguous situation under these rules, please let the course staff know. However, feel free to make reasonable assumptions if needed – just stipulate them in your write-up.

Controller Inputs

This controller circuit will have two one-bit input signals, and no other inputs should be necessary. The first one-bit signal, `car_present`, is 1 if a car is waiting on Prospect Avenue and 0 otherwise. The other signal is `rst`, which will synchronously reset the controller whenever its value is 1. This means that the controller will only reset on a rising clock edge – not immediately when `rst` goes high.

Controller Outputs

This controller circuit will have two three-bit output signals – `light_pros[2:0]` and `light_wash[2:0]`. Each of the bits in these signals controls one of the three lights (green, yellow, red) for the named street. The mapping from bit number to light color is given in the table below:

Bit	Light
0	Red
1	Yellow
2	Green

If the bit controlling a light is 1, the light is on. If the bit controlling a light is 0, the light is off. For example, if `light_wash == 3'b100`, then the Washington Road stoplight is currently green.

Overall Controller Design

A block diagram of the controller is shown below in Figure 4.2.

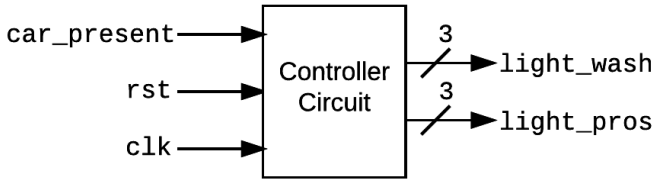

Figure 4.2: Block Diagram of Controller Circuit

The controller circuit itself will be a simple finite state machine (FSM). You can read more about FSM's in Section 3.3 of the course textbook. In this lab, you will design, implement, and test an FSM circuit that will serve as the controller circuit for our stoplight.

Part 1: Designing a Finite State Machine

Your first step for this lab is to design a simple finite state machine (FSM) that represents the control logic for our stoplight. You can draw your FSM however you want to – if you do it by hand, please take a clear photo of it and add it to your write-up.

Please follow these guidelines while creating your FSM:

- Use the rising clock edge for transitions. Assume that the clock in this scenario will have a period of 5 seconds.
- Clearly label and name your states.
- Near each state, list the value assigned to each of the outputs in that state. Alternatively, you could create a table that provides this information – whatever is easiest for you.

- Represent transitions with an arrow from one state to another. Label each transition with the logical condition under which it is taken.
- Do *not* AND the rising clock edge with every condition – we understand that all transitions will be taken synchronously on rising clock edges, so leave the clock edge out of your transition condition lists in your diagram. If a transition is always taken on the next clock edge, represent it with a simple arrow with no label.
- Leave `rst` out of all of your transition conditions, and do not draw a reset transition from every state to your initial state. Instead, simply mark your initial state clearly. When `rst` is one and a clock edge comes, the FSM will transition to the initial state automatically – regardless of which state it was in before. Removing all of the obvious transitions back to the initial state on a reset will make your FSM diagram much easier to read.

Make sure that your FSM's transitions are deterministic – that is, for each possible combination of inputs, there is only one valid transition that can be taken from each state.

Once you're sure your FSM will correctly implement the stoplight controller's behavior, move on to the next section.

Write-up Question 1:
Attach a diagram of your FSM to the writeup. This can be in any format, but if you hand-draw it, please take a clear photograph of your work. Make sure that the FSM diagram meets the guidelines given in this lab manual.

Part 2: Implementing Your FSM

Next, you will implement your FSM as a Verilog module. Make sure you've read section 4.1 of the Verilog tutorial before beginning.

Write-up Question 2:
What are the three main components of an FSM circuit? Provide a brief description of what each component does.

The source code for this lab is in your */lab4* directory. Write the code for your FSM in *Stoplight.v*. The module has already been laid out for you – just fill in the appropriate sections to match your FSM diagram.

If you're having trouble getting started, here's a list of recommended steps:

1. Add local `reg` variables for `state` and `next_state`. Ensure that the bit width of each is sufficient for the number of states in your FSM.
2. Add `localparam` variables for each state, ensuring that all states are assigned different numbers.
3. Add combinational logic for your output signals based on the value of `state`.
4. Add combinational logic to set `next_state` appropriately based on the current state and the input values.
5. Fill in the sequential logic that sets the `state` register on each rising clock edge. This block should deal with the reset condition so that the next state logic doesn't have to.

Once your FSM has been completed, you can proceed to the next section.

Testing Your Design

The */lab4* directory has two test files for your FSM. The first, *StoplightTA.t.v*, contains a simple suite of functional tests for initial testing. To run these tests against your code, open a terminal or the Windows `cmd` prompt, change directories into */lab4*, and run the following command:

```
1 iverilog -g2005 -Wall -Wno-timescale -o StoplightTATest StoplightTA.t.v
```

This command will compile your *Stoplight.v* with the provided test module. If there are any warnings or errors, fix them before proceeding.

Once compilation is complete, you can run the testing suite with the following command:

```
1 vvp StoplightTATest
```

Ensure that there are no failures. If you need waveforms for debugging, they will be dumped out as *StoplightTATest.vcd*. Open them with GTKWave:

```
1 gtkwave StoplightTATest.vcd
```

Part 3: Writing a Functional Test Case

Once your code is passing our functional test case, you will write your own functional test case to confirm that your stoplight controller is working as expected. A testbench has been created for you in *Stoplight.t.v*. Your task is to fill it in to re-create the scenario described below. Note that the time units for this simulation have been set to nanoseconds, so just assume that one nanosecond of simulation time corresponds to one second of real time. The provided clock has a 5-nanosecond period and starts off high. Assume that `car`, which is high if a car is waiting on Prospect, and `rst`, the reset signal, both start off low.

Here's the scenario. All times are given in (nano)seconds from the start of simulation:

- **Time 1** – The FSM's `rst` signal goes high.
- **Time 6** – The FSM's `rst` signal goes low.
- **Time 41** – The `car` signal goes high, indicating that a car is waiting on Prospect Avenue.
- **Time 46** – The `car` is still waiting at the light.
- **Time 51** – After seeing a green light, the car on Prospect leaves, and the `car` signal goes low.
- **Time 66** – A steady stream of cars begins to appear on Prospect, keeping the `car` signal high.
- **Time 71** – The `car` signal is still high.
- **Time 76** – One last car gets trapped on Prospect as the light switches to red. The `car` signal remains high.
- **Time 86** – The car on Prospect makes a right turn on red, and the `car` signal goes low.
- **Time 106** – A car is sighted on Prospect, and the `car` signal goes high.
- **Time 116** – The car on Prospect makes a left turn on green, and `car` goes to zero.
- **Time 131** – Another car shows up on Prospect, and `car` goes high.

- **Time 136** – The final car leaves, and `car` falls back to zero.
- **Time 140** – The simulation ends.

Translate this scenario into Verilog and write your code in the `initial` block of *Stoplight.t.v*. For every event except the first and last ones, wait one (nano)second and then check the outputs of the `Stoplight` module to ensure that the lights are showing what you expect them to show. Note that the outputs will *not* be affected by the most recent input change, as a positive clock edge has not yet occured since that input transition. We're just using times 7, 42, 47, and so on as convenient times to check the state of our stoplight. For example, at time 7, I would write the following statement:

```
// lp = Prospect, lw = Washington
if (lp !== RED || lw !== GRN) begin
  $display("Error at time %t", $time);
end
```

Once you've completed your testbench, compile it with the following command:

```
iverilog -g2005 -Wall -Wno-timescale -o StoplightTest Stoplight.t.v
```

If there are any warnings or errors, fix them before proceeding.

Once compilation is complete, you can run your testing suite with the following command:

```
vvp StoplightTest
```

Ensure that there are no failures. If you need waveforms for debugging, they will be dumped out as *StoplightTest.vcd*. Open them with GTKWave using this command:

```
gtkwave StoplightTest.vcd
```

Write-up Question 3:
Please leave a bit of feedback regarding this lab. How much time did it take, how difficult was it, did you get stuck anywhere? There are no wrong answers here – we'll be using the feedback to adjust this lab for the future.

(Optional) Verification with VCD Viewer

This part is completely optional. We have created a viewer for the VCD files your code produces. This may be useful as a final check that your code is

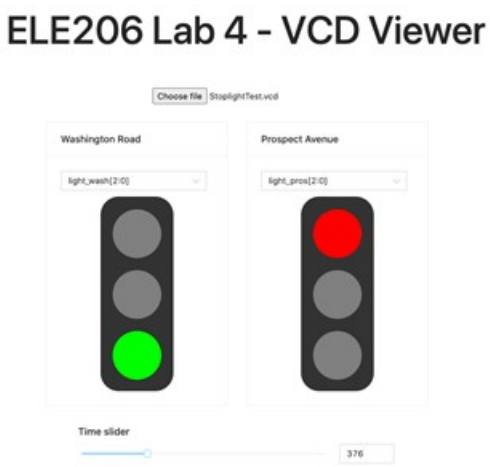

Figure 4.3: Example of an output from vcd viewer.

working as you would expect. Please note that GTKWave will still be more useful when debugging your code as the online viewer will only show the signals interpreted as traffic lights!

Here are the usage instructions:

1. Go to `https://ele206-lab4-viewer.vercel.app/`
2. Make sure you have your final VCD file. Click on "Choose file" and select your VCD file.
3. Select your output signals for Washington Road and Prospect Avenue, respectively using the drop down menus.
4. Use the Time slider to check your traffic lights are changing correctly with the correct order, intervals and colors. Your viewer should look like Fig. 4.3 after the above steps (the name of your vcd file and time may be different).

Demonstration, Write-Up, and Submission

Demonstration

For your demonstration this week, visit one of the Lab help sessions on Zoom to show a graduate TA the test case you implemented in *Stoplight.t.v*. Walk

the TA through the events that happen at the stoplight by using GTKWave to view the testbench's output waveforms.

Write-Up

For your write-up, complete the tutorial review questions assigned at the beginning of the lab and answer the 3 numbered questions from the text briefly but completely (8 questions in total). Feel free to create the write-up in any program of your choice, but save it as a PDF. Name your PDF *ELE206_lab4_netid.pdf*, with *netid* replaced with your Princeton NetID.

Submission

For submissions on Gradescope, please **only upload your PDF write-up (*ELE206_lab4_netid.pdf*)** to Gradescope by the due date (no need to create a .zip folder). Your code will be evaluated during the lab demonstration sessions on Zoom.