Lab 3: Simple Finite State Machines

Assigned: Tuesday 10/24; Due **Friday 11/11** (midnight)

Instructor: James E. Stine, Jr.

1. Introduction

This is a somewhat simpler laboratory but a crucial one. In this laboratory we will get our first introduction to sequential logic and how this works. Although sequential logic is typically a lot smaller than combinational designs, it is incredibly valuable in that it provides most of the intelligence within digital logic. That is, most computing devices have several million Boolean gates on them, but without some mechanism to control the logic, it would never work.

In this laboratory, we are going to use an idea that has been in a popular textbook that is a little older but we are going to give a new twist to it [1]. Again, although this laboratory is relatively simple the importance of this laboratory to later work will be crucial. So, it is important to try to understand the procedure in creating these systems.

The specific class of sequential that we will design in this laboratory are called Finite State Machines or FSMs. Although we cover them in a specific manner in class, they are implemented a tad different than more traditional methods. This is mainly due to the complexity in getting the Hardware Descriptive Language (HDL) to understand what we want to design.

2. Background

Until now, we have only designed combinational circuits. In this lab we will start with circuits that have distinct states. We once again use SystemVerilog to specify our design. As discussed in class, since FSMs are different in terms of their feedback, they should be integrated within a Hardware Descriptive Language (HDL) a little differently to allow the synthesizer to properly design them.

In this lab, you'll design a finite state machine to control the taillights of a 1965 Ford Thunderbird [1]. There are three lights on each side that operate in sequence to indicate the direction of a turn. Figure 1 shows the tail lights and Figure 2 shows the flashing sequence for (a) left turns and (b) right turns.

Both the car and the flashing sequence occur in various varieties in other cars as well as other vehicles (e.g., on bicycle lights). Nevertheless, this is a great exercise to learn how to design a simple Finite State Machine (FSM).

2.1 Baseline Design

Let us start with designing the state transition diagram for this FSM. Give each state a name and indicate the values of the six outputs `LC`, `LB`, `LA`, `RA`, `RB`, and `RC` in each state. Your FSM should take three inputs: reset, left, and right. The circuit should have the following properties:

- On reset, the FSM should enter a state with all lights off.
- When you press `left`, you should see `LA`, then `LA` and `LB`, then `LA`, `LB`, and `LC`, then finally all lights off again.
- This pattern should occur even if you release `left` during the sequence. If `left` is still down when you return to the lights off state, the pattern should repeat.
- The operation when `right` is active should be similar except with `RA`, `RB` and `RC`.
- It is up to you to decide what to do if the user makes `left` and `right` simultaneously true; make a choice to keep your design easy.

The job of an FSM is to do three things:

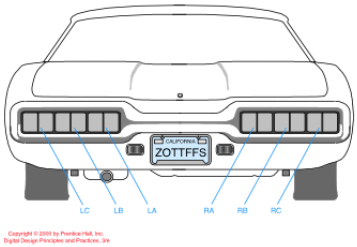

Figure 1: Thunderbird Tail Lights [1]

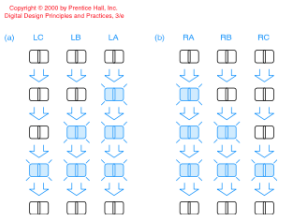

Figure 2: Flashing Sequence (shaded lights are illuminated) [1]

1. Determine the next state from the present state, and the inputs (next state logic).
2. Realize the output function based on the present state and the inputs if a Mealy FSM is used (output logic).
3. Advance from the present state to next state when a clock event arrives (state register).

As discussed in class, the state-machine diagram is an optional step, however, the state transition table is an integral and required step to start. You should start designing your state-transition table. Essentially, a state-transition table is a state-machine diagram in a table form, so that we can use our knowledge in designing combinatorial circuits to derive the Boolean equations for next state logic and output logic.

2.2 FSMs in SystemVerilog

Again, as discussed in class, synthesis engines must resort to methodologies that allow them to integrate FSMs into hardware. There is an extensive amount of literature on the integration of FSMs, but it basically boils down to the following items within the HDL. These items are basically the same for other HDLs including VHDL and Verilog.

Inside the SystemVerilog file, each FSM has to have the following distinct and separate items.

- the state register (where clock moves the next state to the present state)
- the next state logic (where the next state is determined by the present state and inputs)
- the output logic (where the outputs are determined by the present state and inputs) works the best.

Also discussed in class, the output logic and the next-state logic can be separate, but I think you will have more luck and less problems if you combine both of these elements into one item. We discussed this in class rather extensively. Normally, engineers like to have separate output and next-state logic inside a HDL as it saves space, but I believe it possibly leads to missing an output. I will leave the choice up to you. The textbook also discusses this rather extensive in Chapter 4 [2].

It is also good practice to use a naming style that clearly identifies the signals that are registered. If you want to know how every registered signal is connected. Therefore, you should name your states accordingly and not something ambiguous like `state1`.

One of the best practices for debugging FSMs is to also make sure you display your `CURRENT_STATE` and `NEXT_STATE` variables on the waveform. This is vital in debugging your FSM and making sure the output

```
module clk_div (input logic clk, input logic rst, output logic clk_en);

 logic [23:0] clk_count;

 always_ff @(posedge clk)
 //posedge defines a rising edge (transition from 0 to 1)
  begin
   if (rst)
    clk_count <= 24'h0;
   else
    clk_count <= clk_count + 1;
  end
 assign clk_en = clk_count[23];
endmodule
```

Figure 3: Clock Divider SystemVerilog File

occurs correctly. You can also output these variables to a display, such as a 7-segment display, and many engineers have this integrated into special debugging configurations for a digital design.

2.3 Asynchronous Events and Synchronizing the Clock

You have described a clock input (`clk`) in your circuit. The question is where do we get this clock from? You could be tempted to use one of the push buttons or the switches for this purpose. I brought this up in lecture when discussing the time it takes to activate a switch, such as in Family Feud^{TM} game. Events that synchronize events are fairly fast, whereas, outside occurrences are sometimes slow making putting these events into digital logic sometimes difficult.

The problem is that compared to the speed of the FPGA the change in a push button is extremely slow (in fact more than a million times slower). During the slow transition the FPGA will see many fast occurring transitions, and would interpret each of them as a clock edge. This is known as 'bounce', and usually specialized circuits (and sampling techniques) are used to prevent this. In any case, using the push buttons is not a very safe way of generating the clock.

If you look at your board, you may notice the text `sysclk_125mhz`. If fact, your board contains a 125MHz crystal oscillator circuit. The output of this oscillator is connected to the pin W5 (which is a special clock pin for this FPGA). We will simply tell he compiler to connect the net "clk" to the pin W5 in the constraints file (XDC). In this way we will have a clean clock signal.

Now we have a different practical problem: the clock is too fast. The 125 MHz means that every clock period is 8 nanoseconds long. The entire blinking sequence would be then 40 ns. This is a very short time: light travels less than 250 meters during that time. If we want to see our sequence, we need to find a way to dramatically slow down the circuit.

This can be achieved by two means. Either we implement a clock divider circuit that divides the clock by a few million times, or we can generate an enable signal every few million cycles, and then use this enable signal to control our next state transition. We will employ the former for this laboratory.

In this exercise, we will give you a small `clk_div` circuit that takes in the same `clk` and `rst` signals, and generates a `clk_en` edge signal every $8,388,608$ cycles (or every 2^{23} cycles). Considering that the main clock frequency is $125,000,000$ cycles per second, this means a `clk_en` signal is generated every 0.06711 seconds or 67.11 ms. This is a realistic clock to our light controller in action. The SystemVerilog is shown in Figure 3

The idea is pretty simple, we increment a 24-bit counter (called `clk_div`) at every clock, and set the `clk_en` to 1 when the most-significant bit (MSB) of the counter s 1. By increasing the counter size you can change the division factor as you please. Use this new `clk_en` as your clock for your FSM. Now all we have to do is to choose which buttons on the board we want to use for the control, and which LEDs we will use as taillights.

2.4 Alternate Design

We also want to make several modifications to this lab to give some different approaches. Therefore, you should also incorporate the following items into your design as a modification of the baseline laboratory.

- If both `left` and `right` are both asserted, you should have both lights (e.g., `LA`, `LB`, and `LC`) flash on and off. This would be similar to a hazard light in your car.
- The Ford Mustang's hazard lights work like the normal hazards except they produce them in sequence like our normal baseline lab. That is, you should have them do the following when in hazard mode: 1.) `LA` and `RA` 2.) add `LB` and `RB` and finally 3.) add `LC` and `RC`. You can see them on the following YouTube video: `https://youtu.be/5azgaPDvPDk`.

You should download the files in this lab from Canvas. In the distribution is the normal `top_demo` similar to Laboratory 1. A template FSM SystemVerilog file is given and you can use it to modify your FSM for this laboratory. As mentioned in previous labs, please make sure you adequately simulate your design with a testbench. Do not go to implementation without simulating your design completely on your laptop or desktop at home.

Both the clock divider and template Finite State Machine SystemVerilog (SV) file are given to you in your zip files. There are also separate `DO` files so you can run both easily. You will need, as indicated previously, a complete a state-transition table for your design. Although it is tempting to do this without writing it down, it is highly advisable to write it down. Then, just update your FSM SV file accordingly.

2.4.1 Hints/Tips

One of the challenges in this lab is that the clock for the FSM is so much slower than the clock for the FPGA. This will allow you to use a clock that is faster. Do not forget to output the `CURRENT_STATE` and `NEXT_STATE` to your waveform to check where you are with your state. It would be highly advisable to also simulate the FSM with a separate testbench. Then, create a top-level module (i.e., digital design) that incorporates the `clk_en` and the FSM and simulate with a separate testbench than the `clk_en` and the FSM. The Graphical User Interface or GUI is really useful in debugging your FSM. Another great hint is to also output both variables as `output logic` instead of `logic` and then send that output to the board somehow (e.g., 7-segment display).

As in previous labs, it is highly advisable to do all your simulation at home before you get to ENDV. The testbench methodology is so crucial to the implementation and saves you so much time. Otherwise, you are debugging both the implementation and design and just wasting time. So, make sure you have your design completely working within a testbench before you come to ENDV 360.

Also remember there are two types of Finite State Machine (FSM) designs as discussed in class: Mealy and Moore. This lab will implement a Moore-style FSM and you should define the output per each state. The book does this a little differently as mentioned in class and its far easier to define the output in each next-state.

Also, remember you can use the switches, LEDs, push buttons and 7-segment displays to help you debug the implementation. These debugging elements are very crucial to debugging your design. Also, feel free to play with the clock divider to speed or slow things down. If you get tricky, you could easily output the time (i.e. in ms or seconds) that the clock is working at on the 7-segment display, which would be a cool extra-credit option.

References

[1] John F. Wakerly, *Digital design - principles and practices*, Prentice Hall Series in computer engineering. Prentice Hall, USA, 4th edition, 2005.

[2] Sarah Harris and David Harris, *Digital Design and Computer Architecture: RISC-V Edition*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2021.