ECE 5727

Lab 2: ThunderBird Taillights

Lab Point Value: 30
Post-Lab Point Value: 35

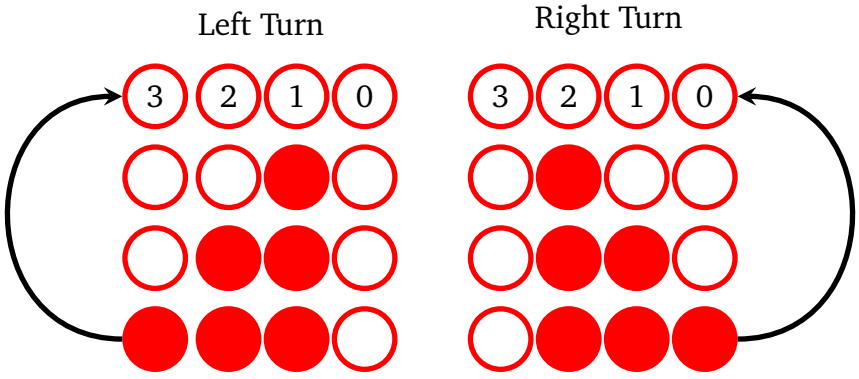
Objectives

In this lab you will:

1. Design and build a synchronous state machine.
2. Gain further familiarity with Xilinx Vivado and the design flow, especially synthesis.
3. Use the Zybo to emulate the tail lights of a 1965 Ford Thunderbird

Lab Overview

The 1965-67 Ford Thunderbird had three left and three right tail lights which flashed on in sequence to indicate left and right turns. Unfortunately, the Zybo only has 4 LEDs, so we will be using the middle two for both right and left turns.



Your task is to design the FSM controller for these 4 lights.

We will be using two of the switches to act as the **Left** and **Right** turn signal indicators for input. This is obviously imperfect, as we don't have an actual three-position switch to work with. You can assume that both switches will never be on at the same time. Additionally, we will be using one of the buttons for input, to re-create the functionality of **Hazard** lights being turned on, and you can assume this button will never be pressed when a turn signal is enabled.

- When **Left** is asserted, the lights flash in the following pattern: all off; LED[1] on; LED[2:1] on; LED[3:1] on; and then the sequence repeats.
- When **Right** is asserted, the sequence is similar, using LEDS[2:0].
- If a switch is turned off in the middle of a turning sequence, the sequence should complete (not go straight to all-off). Thus, in the middle of a **Left** or **Right** sequence, there is no need to continue checking the input until the all-off state is reached.
- If the **Hazard** button is pressed, the lights should enter hazard mode, and all four lights flash off and on in unison. The controller should remain in hazard mode until the button is pressed a second time.

Instructions

Design

1. (*5 points*) Design an FSM for this system. Draw out the state diagram. The correct design will require 8 states, one idle/off state, one for the hazards, and three each for left and right turns. Note: if you use 9 states (2 for hazard) that is OK. *Submit the diagram*
2. (*10 points*) Using the *tbird_shell.v* code provided, implement your FSM.
    - Fill in the localparam declaration with names for your states.
    - You may use either a 1 or 2-process statement machine. The shell code contains the start of a 1-process machine.
    - Make sure the slowClk is being used to clock your FSM. This way the states will change only once every 1/4 second, setting the rate of flashing for the leds.
    - The **hazardPushed** signal should be used to detect button presses. It will assert for one slowClk tick when the button is pressed down.
3. Run the code through synthesis to check for basic correctness. Other than those related to constraints, it should not generate any warnings. Note that in order to complete this step you will need to create a new Vivado project and import *tbird_shell.v, debouncer.v, clk_gen.xci* as design files.

Test & Implement

1. (*5 points*) Once your code synthesizes cleanly, it is time to testbench it. Add *tbird_tb.v* as a simulation file to the project, and run a behavioral simulation. Currently, the testbench is set up to only test the left turn signal. Modify it to fully test your design. *Submit a screenshot of your simulation waveform showing the design working correctly*.

2. (*5 points*) Now it is time to add the constraints file - *tbird.xdc*. This file informs Vivado which package pins to connect to the input/output signals of the top-level module. As Digilent has designed the board, and therefore already determined which pins are connected to which leds, switches, and buttons, they have provided a skeleton constraints file for us to modify. Each pin is described by a line such as this:

   ```
   set_property -dict {PACKAGE_PIN W13 IOSTANDARD LVCMOS33}[get_ports {sw[0]}];
   ```

   - Lines beginning with # are a comment. Remove it to un-comment the line and use it.
   - Above, we are specifying that we are referring to pin W13 (row W, column 13) on the FPGA package, and that it should be internally configured to use the CMOS 3.3V standard. The 'get_ports' command tells Vivado the name of the module port we want connected to the pin - sw[0] in this example. As this is the right-most switch, we will want to rename this to 'RIGHT'.
   - Rename another switch to 'LEFT', and select a button to be the HAZARD_BTN.
   - Finally, un-comment the top two lines dealing with the 'sysclk' signal. In the shell code, the port is labeled 'SYSCLK', so update the .xdc to reflect this. The first line, like the other lines, informs Vivado which pin the clock signal is coming in on. The second tells it what frequency and duty cycle the clock will be - for the Zybo this is an 8ns period clock with rising edge at 0ns and falling edge at 4ns which translates to being a 125MHz clock with 50% duty-cycle.

3. With the constraints file completed, it is time to generate a bitstream and load it onto the board to test.

4. (*5 points*) Demo your working TBird to me!

Lab Submission

1. Completed code including *tbird_shell.v, tbird_tb.v, tbird.xdc*

2. Screenshot of testbench

3. Diagram of your FSM - take a picture or scan it if you drew it by hand.

Taking the Lab Further

If you'd like to do more with the lab here are some ideas to try out and then discuss what you find either in person or over e-mail/Piazza:

- I lied about needing 8 states. Can you do it with fewer? Note: adding a counter is the same as using 8 states (its just the lsbs of the state register!). And in fact, its actually more inefficient since you are wasting the state created when the counter == '11'.
- Make the FSM more robust: handle the hazard button being pressed at any time (including during a turn); make it so that whichever turn was in progress when the second switch is flipped on remains running until one of the switches is flipped off;
- Make the FSM a 2-process Mealy machine if you didn't already do it that way. If you did, then make a 2-process Moore machine. To make these functionally the same, do you end up with any benefit of choosing one style over the other? What about if you make 1-process versions?
- Anything else you can think of!

Post Lab

Submit answers to the following questions, along with any supporting documentation such as updated code/testbench files and simulation waveform screenshots.

1. (*10 points*) Discuss the lab design process including:
   - What problems did you encounter and how did you solve them?
   - Explain your design decisions, including whether your FSM is a Moore or Mealy implementation and why you chose that.
2. (*20 points*)
   (a) (*5 points*) If your code is written properly, Vivado will have been able to recognize the FSM pattern. Under the 'reports' tab at the bottom of Vivado, there should be a synthesis report available. Take a look through it, and you will find where Vivado inferred the FSM and re-encoded the states. Submit the portion of the report (a screenshot is fine, or just copy-paste the text block) showing this.
   (b) (*10 points*) Now, alter your FSM and the code to use a different style - either change number of processes or between Moore/Mealy implementation - and run it through synthesis. Look through the report again - has anything changed? Is this what you expected?
   (c) (*5 points*) Vivado by default automatically chooses the encoding style it thinks best. However, this isn't always ideal (tools are not perfect). As such, we can use attributes to direct it to use the style we want. The format is:

```
(* fsm_encoding = "<style>" *)
```

where style can be one of {"sequential", "one_hot", "gray", "johnson", "auto", "none"}. This attribute is placed prior to the declaration of the state register like so:

```
(* fsm_encoding = "one_hot" *) reg [2:0] state;
```

Add an attribute to your state register to force Vivado to encode in a manner different than how it already has. Synthesize and discuss the new encoding. Be sure to include the output from the report so I know what you are talking about :).

3. (*5 points*) The other report generated from synthesis is a utilization report. Is there anything interesting about what is being used? How do the numbers compare to the overall resources available on the chip? Submit relevant portions of the report for reference.