**Lab 3: Thunderbird Lights**

**Authors:** Lauren Humpherys and Christopher Katz

**Documentation:** Captain Johnson helped us differentiate between state and output logic. He also helped us correct major errors by telling us to uncomment all switches and LED lights in the constraints file, and ground the LEDs not being used for this lab. C3C Felix Zheng verified that our RTL Schematic was correct.

**Purpose:** The purpose of this lab is to write, test and implement 6 LED lights on the Basys3 development board. The finite state machine will take four inputs: the clock, sw(0), sw(15), and btnR for reset. It will control 6 LED lights to emulate the tail lights on a Ford Thunderbird. On reset, the FSM should immediately enter a state with all lights off. When you press btnR while just sw(15) is switched on, you should then 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 logic for the right lights is similar while just sw(0) is switched on. Additionally, when both left and right switches are on your state machine should blink all lights on and off (implementing hazard lights). Finally, pushing btnL resets the clock. When the clock is not running, the machine output freezes in its current state. More specifically, holding the left button down causes any lights currently on to stay on, and any lights off to stay off.

**Prelab:** The first step in our design was to determine the Boolean logic needed to appropriately activate each of the six LED lights. This was done according to the following diagram.

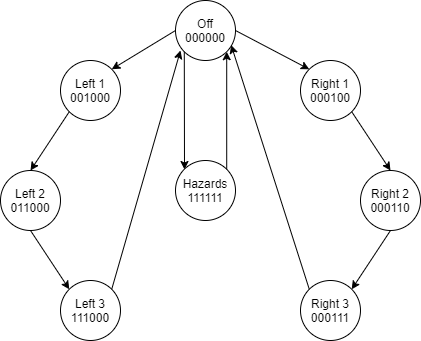
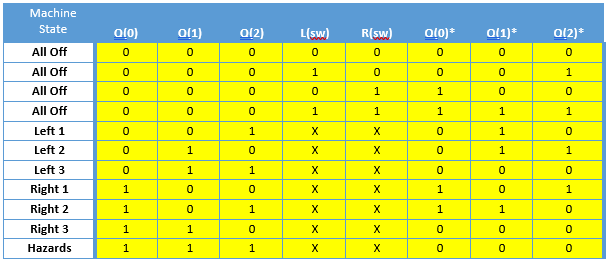


Figure 1 – Transition Diagram of Finite State Machine

Each of the possible inputs was evaluated to determine how the individual states and outputs would be affected. The results were compiled in Table 1 below. Table 1 – Showing the mapping between possible inputs and desired states

Utilizing Table 1 above, we were able to derive the following state transition equations for Q(0)\* - Q(2)\*. In order to make the implementation easier, K-maps were used to reduce the equations produced the equations below:

Q(0)\* = Q(0)'Q(1)'Q(2)'R(sw) + Q(0)Q(1)'

Q(1)\* = Q(0)'Q(1)'Q(2)'L(sw)R(sw) + Q(0)'Q(1)Q(2)' + Q(1)'Q(2)

Q(2)\* = Q(0)'Q(1)'Q(2)'L(sw) + Q(0)'Q(1)Q(2)' + Q(0)Q(1)'Q(2)'

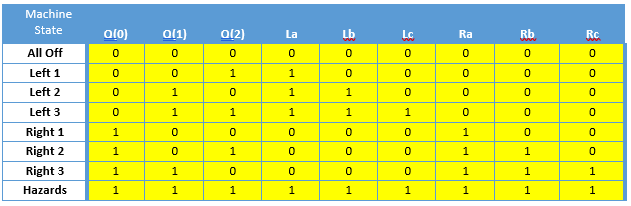


Table 2 – Showing the mapping between the states and outputs.

Utilizing Table 2 above, we were able to derive the following boolean output equations for La-Lc and Ra-Rc. In order to make the implementation easier, K-maps were used to reduce the equations and can be found with the equations below:

La = Q(1)Q(2) + Q(0)’Q(1) + Q(0)’Q(2)

Lb = Q(0)’Q(1) + Q(0)Q(1)Q(2)

Lc = Q(1)Q(2)

Ra = Q(0)

Rb = Q(0)Q(1)) + Q(0)Q(2)

Rc = Q(0)Q(1)

**Design:**  In order to implement this lab our final design needed to match the design seen in Figure 1.

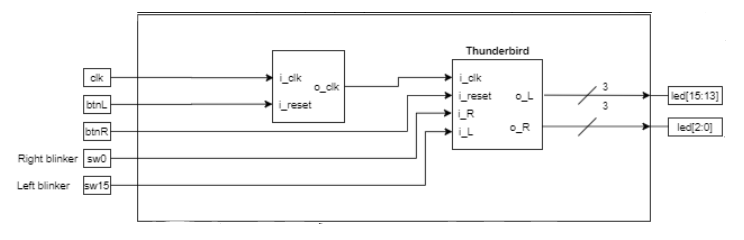


Figure 2 – Top Level Schematic

Our first step in reaching this goal was to design and test the Thunderbird FSM component separately utilizing the equations Q(0)\*-Q(2)\* above, and matching the top-level entity design above.

**Our Approach** – We set up our component file utilizing the provided class template and renamed it thunderbird\_fsm.vhd. After renaming the template we defined the entity to match Figure 2 with a three-bit Q input and a six-bit LED output. We then defined signals for Q(0)\*-Q(2)\* to align with our pre-lab equations. After implemented the Q(0)\* equation, we stopped and checked for syntax errors. At first we kept getting errors because we wrote it as Q(0)\* instead of Q\_next(0). We also got errors for writing i\_left as i\_left(sw) and i\_right as i\_right(sw) in our logic equations, but with a little help from Captain Johnson we corrected the issue. We then repeated this process with for Q(1)\* and Q(2)\*, checking syntax as we went. Next, we wrote the output logic for the six LEDs (right used led(0:2), left used led(15:13).) In the prelab, we got output logic confused with state logic and came up with the wrong equations. We were still coming up with the wrong equations during this lab, but Captain Johnson helped us correct this issue. Finally, we added the clock register process that was utilized for the Stoplight functionality in ICE4.

Our next step was to design the test bench to ensure our component worked correctly. We modified the class testbench template and renamed it thunderbird\_fsm\_tb.vhd. We declared our Thunderbird FSM and instantiated a version of it. We mapped the component to our artificially created inputs clk, reset, left and right, and outputs lights\_L and lights\_R. We then created a test process to run through all possible inputs. First we set the left and right inputs to zero, and then tested the reset button to make sure everything else is off. Then we tested the hazard state by setting the left and right inputs for 40 nanoseconds before turning them both back off again. Next we tested the right outputs by setting just the right equal to one for 40 nanoseconds before setting it back to zero again. We then did the same with the left, except doing another reset instead of turning it off. At first, led(1) was outputting the opposite output that it should have. It turns out it was because our output logic equation for o\_lights\_L(1) in our thunderbird\_fsm.vhd file was incorrect. Once the files were free of syntax and logic errors, we ran the simulation. The simulation waveform can be seen in Figure 3:



Figure 3 – Simulation Waveform

We examined the waveform and confirmed all inputs were tested, and all inputs performed correctly. For each test case, we checked these output values against the boolean values we determined in Table 2. After verifying that they matched, we were confident that our component was operating normally.

The next design step was to create the top\_level design file to match Figure 1. We started with the provided top\_basys3.vhd file provided for the lab. The entity was already defined for us, so we just needed to complete the architecture. We declared our Thunderbird FSM and Clock Divider and created a wire w\_clk to connect the two together. We then instantiated the two components, and for the Thunderbird FSM we connected i\_reset to btnR, i\_clk to w\_clk, i\_left to sw(15), i\_right to sw(0), o\_lights\_R(0) to led(0), o\_lights\_R(1) to led(1), o\_lights\_R(2)to led(2), o\_lights\_L(0)to led(13), o\_lights\_L(1) to led(14),and o\_lights\_L(2)to led(15). For the Clock component we connected i\_clk to clk, i\_reset to reset, and o\_clk to clk. Most of our errors came from us trying to map different outputs to the same LED lights, since five of our machine states involve multiple LEDs being turned on. We also corrected the statement needed to ground all of the unused LEDs between led(2) and led(13). The statement in the given testbench file was led(12 downto 3) <= (others => ‘0’); when it should have been led(12 downto 3) <= “0000000000”;.

**Final Results:** Once we finished creating our design in VHDL we looked at the RTL schematic for both the top\_basys3 design and the Thunderbird FSM design. We also took a look at the Clock Divider and its impact on the overall design. Figure 4 shows top\_basys3:

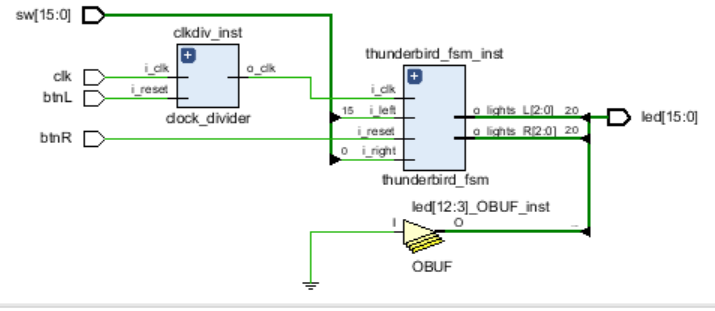


Figure 4 - Final Top Level RTL Schematic

This schematic contains the right number and type of components we expected. It looks virtually the same as our high level design made during the early steps of the project showing our top-level entity and its internal architecture, represented in Figure 2. The only difference is that it shows an output buffer that grounds all of switches grounded instead of just showing the two switches we need and leaving led(3) through led(12). We expected this outcome when we changed it to get rid of errors.

Figure 5 breaks this schematic down even further, showing the internal architecture of these components:

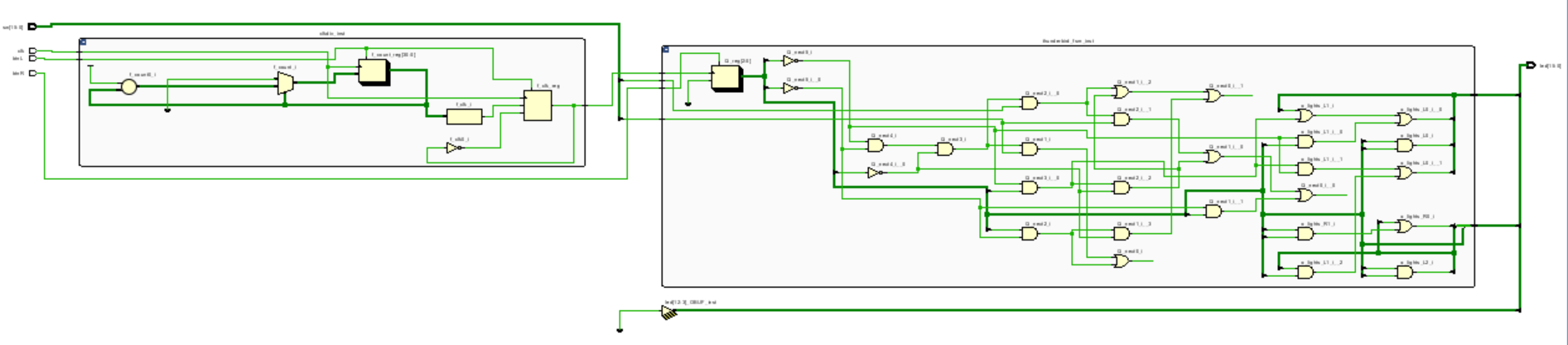


Figure 5.1 - Low - Level RTL Schematic (Clock)

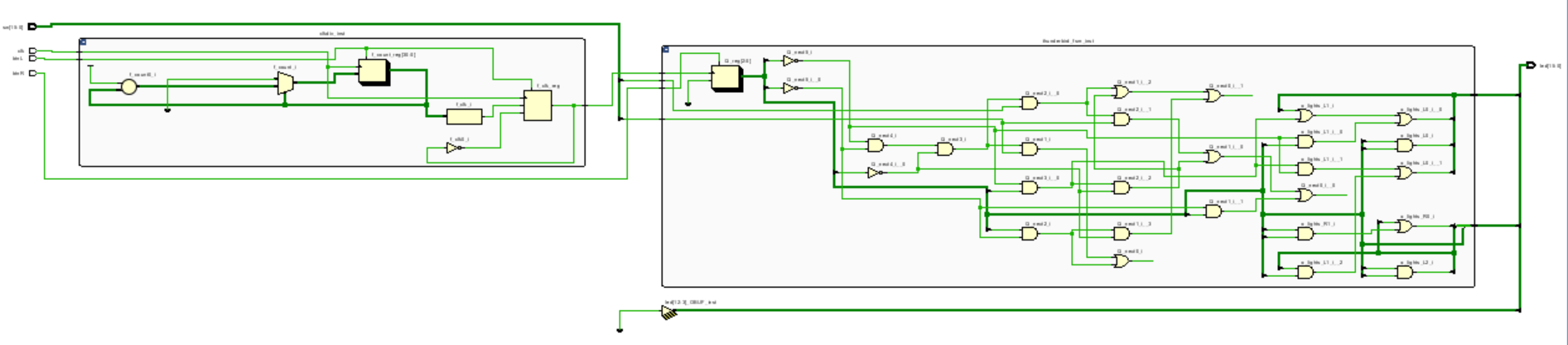


Figure 5.2 - Low - Level RTL Schematic (Thunderbird FSM)

Figure 5 is consistent with our state transition equations listed above. It also provides visual clarity for the number and types of components in the following Hierarchical RTL Component Report:

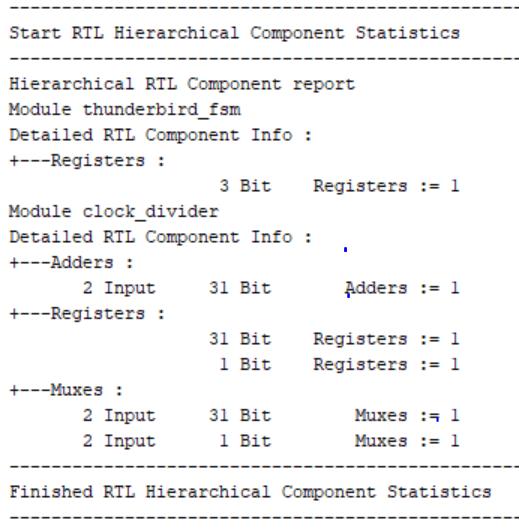


Figure 6 - Hierarchical RTL Component Report

The number and type of components in the above Hierarchical Component Report is indeed consistent with our intended design. As you can see in Figure 5.1, there is one 2-input adder increasing the count, two registers storing the count, and two muxes in the clock divider. There are also two inputs to the clock divider, which is reflected in our design. As for the Thunderbird FSM, we do use 3-bit inputs because the left and right output vectors each contain 3 bits. The interesting discovery we found in Figure 6 was the number of bits used in the clock divider components. If you look at Figure 7 below, almost all of the device utilization is put towards taking inputs and generating their respective outputs. It can be inferred that the storage and processing required to accomplish this occurs in the 31-bit components of the adders, registers, and muxes. Meanwhile, the 1-bit register would be the Look-Up Table resource being used, and the Flip-Flop resource would be the 1-bit mux.

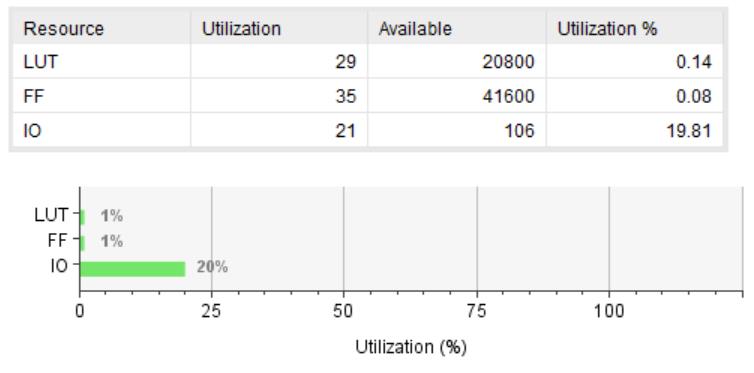


Figure 7 - Utilization Report Summary

Our final step was to generate the bitstream so that we may implement our project into hardware. Because the top\_basys3 utilized standard notation for sw and seg, we only had to uncomment the respective lines in the Basys3\_Master.xdc file. Originally we had sw[14:1] and led[12:3] commented out, but after receiving a lot of errors we uncommented all 15 switches and all 15 leds and set the ones we weren’t using to 0. We pushed the bit stream to our board and successfully demonstrated all 4 possible inputs to Captain Johnson.

**Conclusions:** In this lab, we worked a lot with the intricacies of circuits. For our output logic, we kept treating each output possibility as their own separate state by mapping them to all the LED lights they would be using. For example, in the port map of our Thunderbird FSM instantiation, we have tried o\_lights\_R(2)to led(0), led(1), and led(2)instead of just led(2). It took a while to understand that the output we were mapping here was not a separate state, but simply a mapping that would add on to the outputs already included in that vector.

**Reflection:**

* **Number of hours spent on Lab2 (Combined):** 30
* What portion of the lab was the most difficult for you? How did you overcome it?
  + The most difficult portion of this lab was fully understanding the logic behind the finite state machine, and how to create our equations and truth tables based on this logic.
* What lessons, previous assignments, or activities did you find helpful is completing this lab?
  + This lab helped us put into practice creating mealy and moore machines, which we knew we did not fully understand but did not know where to start in learning.
* What suggestions do you have for improving Lab2 in future years? Be specific. Ex: “The instructions were confusing” does not help. What parts of the instructions were confusing
  + more specificity explaining the state machines and how to create the truth table would have helped a lot.