ECE 3270 : LAB 2

STATE MACHINES

March 13, 2022

Aaron Bruner
Clemson University
Department of Electrical and Computer Engineering
ajbrune@clemson.edu

ABSTRACT

For this lab we were tasked with developing a circuit that represents a state machine. The overall idea is to model the idea of a sign in an airplane which instructs passengers whether or not they need to be wearing their seat-belt and if they can use electronics. This is an ideal example for describing a state machine. Depending on predetermined logic we can lay out logic to illuminate the sign to instruct the passengers.

1 Introduction

Airplanes have very complex systems to keep them in the air throughout their trip. One of the more simple systems is the small light above a passenger's seat that indicates whether or not they need to wear their seat-belt or if they can use electronics. This system may not be implemented as simply as we are testing in this lab but it demonstrates the overall concept.

We can best describe this circuit using a state machine. The main reason why this is the best option is because we know exactly what all states are in our system and where each state will go to next. In addition, we can be a little creative and handle unexpected input as we want to. This lab compared to the previous lab is much simpler. We are not required to implement many different circuit elements since all of our operations can be contained in one VHDL file.

2 Design

2.1 Design of State Machine

When designing a state machine we need to take into consideration all of the inputs and desired outputs for our circuit. For this system we will have an input for our clock, reset, 10k altitude, 25k altitude and smooth. As a result, we will be outputting to our passengers whether or not they need to have wear their seat-belt and if they can use their electronics. The last thing we must take into consideration is all of the states possible for our state machine. The states we can have are as follows: ground, altitude 10k, altitude 25k, smooth 1, smooth 2, smooth 3, smooth 4, and smooth 5. Thus, we have a total of eight states that are possible. Below will describe the methods of moving from stage to stage.

2.2 Tracking Inputs

In an airplane situation the pilot would be controlling the input. By this we mean the pilot would be increasing altitude or decreasing altitude. Depending on the state of the plan we can modify the lights throughout the cabin to best accompany our passengers. Considering this is not a real airplane we need to indicate to our system what is happening in the real world. Thus, we will use the switches to input data into our system.

2.3 Tracking Outputs

For our system we need to determine what state we currently are in and what the sign in the cabin needs to say. Thus, we will use two LEDs to illustrate both of these facts. LEDs four and five will be used to illustrate the sign output to our passengers. The leftmost LED will signify if the passenger needs to put away their electronics. Thus, if the LED is on then the passenger should not have their electronics out; otherwise they are allowed.

For the rightmost LED we are signifying the seat-belt. If the LED is on then the passenger should have their seat-belt on; otherwise they can take it off.

Lastly, we need to track our current state which will help debug the board code. We are simply counting from zero to eight on our LEDs depending on the current state. This information would only be used by a technician or the pilot to ensure functionality.

2.4 Design Rules

Like any circuit we need to follow specific rules for the system to operate in a manner that matches with the desired output [1]. To begin, we assume that the plane is initially on the ground. This means that we initialize our state machine to be on the ground as the first state. Next, whenever the plane passes 10,000 (25,000) feed moving in either direction, alt10k (alt25k) will pulse high for one cycle. Next, if the plane is not climbing, descending, or experiencing turbulence, the smooth signal will be set to high. Next, the state machine should set the no-electronics sign to high when the plan is below 10,000 feet, and low otherwise. Lastly, the seat-belt signal should be low only when the plane is above 25,000 feet and smooth has been asserted for at least five cycles.

Thus, we only truly have three different states for our sign display. When we are on the ground the sign will illustrate that the passengers need to have their seat-belt on and have no electronics on. If the plane is climbing or descending then the passengers can turn on their electronics but must have their seat-belt on. The only instance where the passenger can take off their seat-belt is if the plan has been smooth for five cycles.

2.5 VHDL State Machine

After we hammer out our logic on paper we can begin to implement this into VHDL. Page 53 of the lecture slide demonstrates good programming techniques for making a state machine [2]. We begin by defining our enumerated data type which helps Quartus know we are working with a state machine. For our state machine we have a total of eight data types being ground, altitude 10k, altitude 25k, smooth 1, smooth 2, smooth 3, smooth 4, and smooth 5. The next step is to setup our process based on the clock and reset input. Considering we are handling our clock on a button instead of a switch we need to detect the clock on either a rising or falling edge. Considering we always want to be changing values during the clock cycle we will detect on the rising edge.

Using the example provided in the lecture slide we will use a switch statement to describe our state machine [2]. The select statement is going to be the current state and we will change the state based on the current input. It's important to remember that with a Moore's state machine the output is purely dependent on the state and not the input value. We only change the state depending on what the current input is.

To begin, we need to follow the rule that instructs us to start our plane on the ground. Thus, if the reset is either zero or one we set the state to ground. Next, we have two different states we can transition into, the first being climbing to an altitude of 10k, or continuing to be on the ground. While on the ground we have both the light for seat-belts and no-electronics on, or for our situation tied to high.

Once we arrive at an altitude of 10k we have practically the same options, we can either continue to climb to 25k altitude or we can keep at 10k altitude. Another option we have is to return back to the ground. Once we are at an altitude of 10k we can disable the requirement for having the electronics be restricted. However, we must keep the light for having seat-belts on considering we have not reached an altitude of 25k with five smooth cycles.

After we have transitioned from 10k altitude to 25k we have similar options as before, we can either stay at 25k altitude, move up to smooth 1 which indicates that the plan is smooth, lastly we can return back to 10k altitude. This is the same idea we have seen from the time we lifted off of the ground.

Once we being to have a smooth trip we can begin to implement more creative logic. For my implementation I decided to handle turbulence as part of my design. If at any point during the five cycles to being smooth without seat-belts there happens to be turbulence we reset our counter back to 25k altitude or zero smooth. If for any reason there is an issue with the plan and we transition from smooth to 10k altitude, which should never happen. Then we are covered. The system will react to such a change normally and display the correct output.

Once we achieve smooth level five we can disable the seat-belt requirement and enjoy our flight. Assuming we continue to have a smooth flight for an infinite number of clock cycles our seat-belt requirement and no-electronics requirement lights will be off.

The above logic is best described through the state machine diagram that Quartus generates for us in figure 1. After reading over the text above the VHDL code that describes our state machine could be read like English. All it does is modifies the state based on the current state and input data that is fed into the system.

3 Results

To ensure that our board will perform as we expected we can use ModelSim to verify the output in figure 2. This image best describes what we should be seeing on our board based on the input that we provided. And after comparing these results to our state machine diagram we are seeing the desired results.

Once the code is all compiled we can program it to our board to do physical testing. As we expected, the board performs just like our ModelSim simulation. The only issue that occurred during our tests was the fact that the DE10-Lite board does not work with this lab. After testing modifications to our code, erasing the board and using testing utilities to ensure board functionality I determined that it does not work for this lab. As a result, we transitioned over to using the boards in the lab instead. Once we programmed our code on the lab boards it worked just as expected!

Once the board was programmed I could switch on the different situations and view what happens when you pulse the clock. For anything that is not explicitly described in our state machine we practically ignore the input. Say if the input was 111 for any state. We would just continue being at whatever state we currently are at until a real input is provided.

4 Conclusion

To conclude, this lab taught us many things about Quartus, ModelSim and our DE10-Lite. For starters, the DE10-Lite is incapable of performing this lab. And as a result, we needed to transfer all of our code over to the Linux machines in the lab. And as mentioned in the previous lab, doing this is a true pain and can easily cause issues if any steps are skipped. Another interesting thing to mention is that weird things happen when you hold down on the clock signal and change the switches. I am unsure if the clock is truly held to zero volts when the button is pressed. Instead, it may just lower the voltage which could cause issues down the road.

Lastly, we all now know to to generate the Quartus state diagram image which helped verify that our VHDL code was functioning as expected. This is a big added benefit to only using ModelSim to verify.

References

[1] B. Green. Lab 2: State machines. PDF, Clemson, 2022.

[2] Y. Lao. lect.5a.synchsequentiallogic. PDF, Clemson, 2022.

5 Appendix

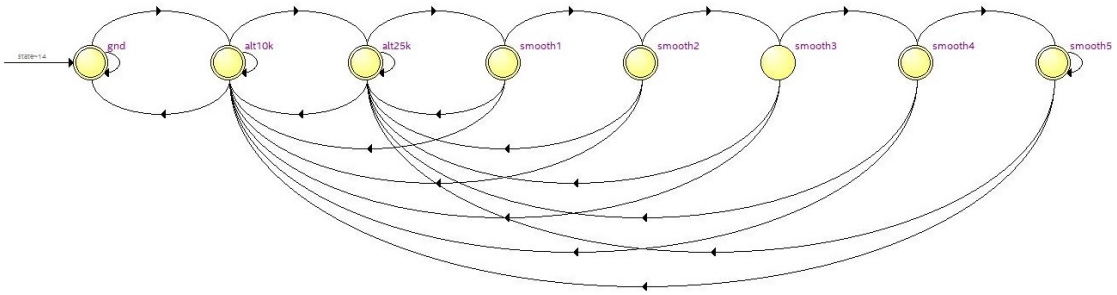

Figure 1: Quartus generated state machine diagram

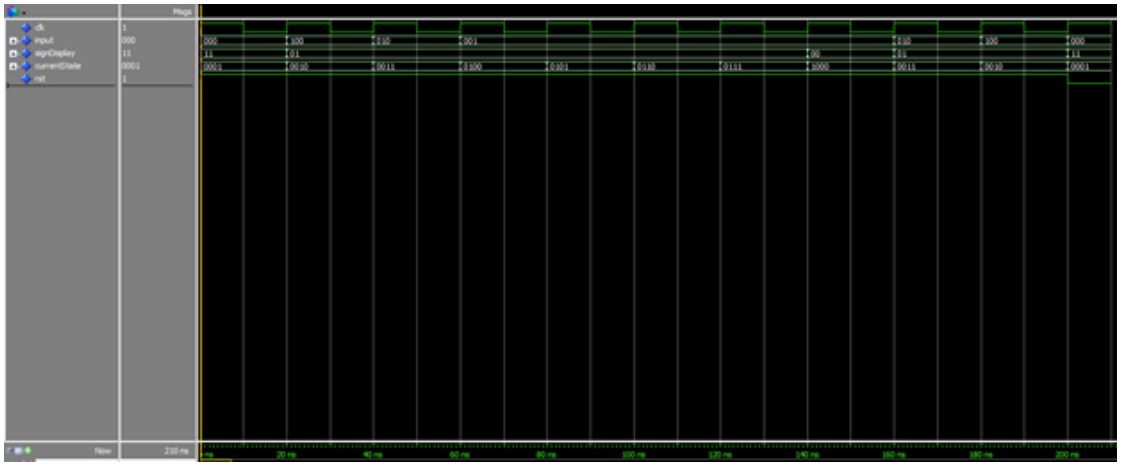

Figure 2: Simulation of VHDL code to ensure functionality