ECE 152A – Winter 2024
Lab 4

Objectives

In this lab, you will implement a Moore machine using Verilog and run it on the FPGA. The purpose of this lab is to imitate the tail light controller of a 1965 Ford Thunderbird, a car that featured sequentially flashing tail lights.

Lab Schedule

Lab Dates	**Deliverables**	**Due Date**
February 21-22	**Part 1**	March 5 11:59PM
February 28-29	**Part 2**	March 5 11:59PM
March 6-7	**Part 3**	Checkoff by the end of lab

Grading

The lab will be graded as follows for steps that are completed on time.

Part	**Weight**
Part 1	10 %
Part 2	30 %
Part 3	60 %

Please note that we take the Honor Code very seriously. You must write the Verilog Code with only your own team.

Design Strategy Outline

Part 1

Understand the behavior of the tail light controller and map the behavior of the controller to a Moore machine. Define the inputs/outputs of the circuit and draw a state diagram.

Part 2

Implement the state machine in Modelsim using Verilog. Construct a testbench and use a waveform to prove that the behavior of the controller is correct.

Part 3

Download your design onto the FPGA and wire up a circuit that demonstrates your design's functionality.

Necessary Supplies

For this lab, you will need:

- UCSB DigiLab FPGA (Information about the FPGA)
- Breadboard
- 10-input DIP Switch
- 6 Through-Hole LEDs
- Resistors for the LEDs and the switches.
- Two BNC to alligator cables

Detailed Design Strategy

Introduction

The objective of this lab is to design a controller that imitates the tail lights of a 1965 Ford Thunderbird. The tail light functions that we will look at are: left/right turn signals, brake lights, hazard lights, and running lights.

- **Turn signal** – There are three lights on each side, which flash sequentially in the turn direction.
- **Brake light** – All six lights are turned on as the brake "pedal" is pressed.
- **Hazard light** – All six lights flash on/off in unison.
- **Running light** – All lights that would normally be off are turned on at 50 % brightness.

Note that several functions can be activated simultaneously. In the following section, you will be told how to handle these situations.

Behavior Specification

Figure 1 shows a high-level block diagram of the controller that you will implement in this lab. Inputs to the circuit should be provided using the DIP switches.

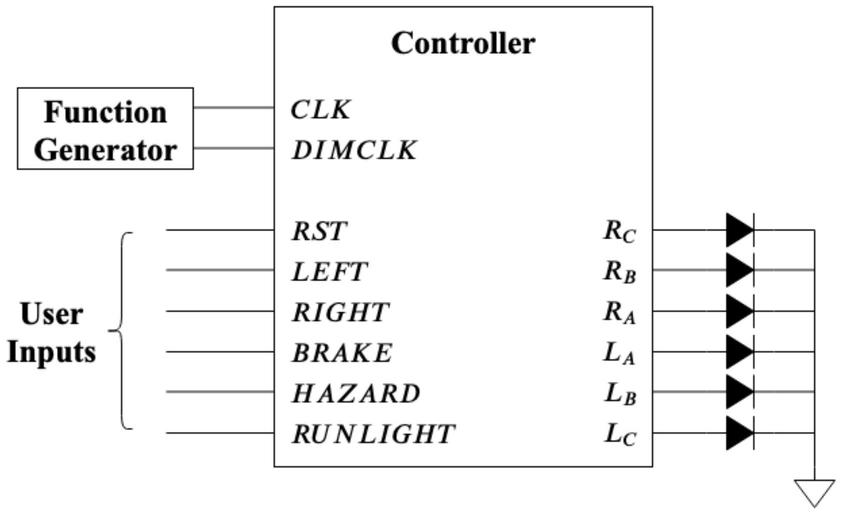
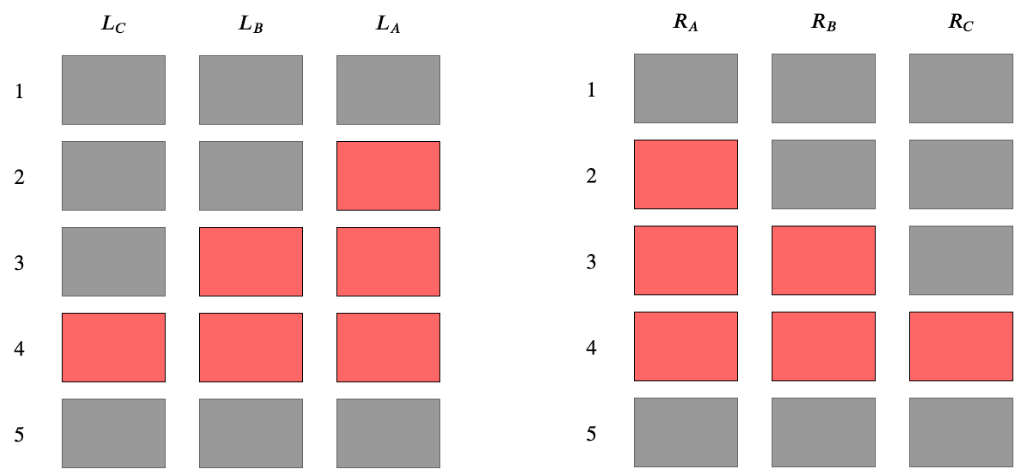


Figure 1

Left and Right Turn Signals

There are three lights on each side of the car's rear that indicate the turning direction by flashing sequentially. Initially, all three lights (on a side) are off. When a turn signal is activated, the sequence begins by turning on one light at a time sequentially in the direction of the turn. After all three lights are turned on, the sequence begins again with all lights turned off. Figure 2 demonstrates what this sequence looks like for the left (a) and right (b) turn signals – grey indicates that the light is off and red indicates that the light is on.



(a) (b)

Figure 2

The turn signals should also exhibit the following behavior.

- When the turn signal is turned off, the sequence is aborted and all lights are turned off.
- The turn signals should not be activated simultaneously, so your machine should display hazard lights when both turn signals are activated. (See the Hazard Light section.)

Brake Lights

The brake lights should exhibit the following behavior.

- While the brake input is activated, all six lights should be on.
- However, if a turn signal is also activated, the corresponding side should continue to display the sequence indicating the turning direction. For example, if both the left turn signal and brake are activated, the left side should continue to flash sequentially while the right side should have all lights on.

Hazard Lights

The hazard lights should exhibit the following behavior.

- While the hazard input is activated, all six lights should switch between on and off in unison.
- An active hazard signal should override any turn signal.
- If both turn signals are active simultaneously, the machine should display hazard lights.

Signal Precedence

It is possible that multiple input signals can be activated at the same time. In these cases, the state machine needs to prioritize certain functions over others.

- If the brake and hazard inputs are activated simultaneously, the brake lights should override the hazard lights.
- The brake lights overrides the hazard lights, however it should still comply with the turn signal if a turn signal is also activated.
- The hazard lights that result from having both turn signals on simultaneously will still have a lower precedence than the brake lights.

Running Lights

When the running light input is activated, lights that are off should now be turned on at 50 % brightness. The functionality of the other lights should not change. Since this is a digital lab, there is no direct way to generate the dimming effect. Above a certain frequency (around 100 Hz), the human eye cannot detect flickering and perceives a dimmed light. Thus, we will use a square wave with a frequency of 100 Hz to produce this dimming effect. This will be the `DIMCLK` input.

The running light function is not a part of the state machine since it's only purpose is to modify the intensity of the light. This function should be implemented using combinational logic separate from the state machine (see Part 2).

Reset

Finally, the controller should also have a reset input that will turn all lights off and overrides all other inputs. The reset can be synchronous or asynchronous.

Part 1 - State Machine Design

First, we need to map the behavior of the controller to a Moore machine. To do this, follow these steps:

1. Before you start your state machine design, answer the following questions (which are meant to help you keep the design simple).
   - What does each state represent in terms of the LED outputs and the different lighting patterns?
   - How many different lighting patterns are there?
   - What is the default state and the corresponding default output?
   - How many states should your state machine have (i.e. what should the minimum number of states be)?
2. Clearly define the state variables that you will use (and how you will encode/represent them in your code). In addition, clearly define the inputs and outputs of the controller.
3. Draw a Moore state diagram for the controller. Show all the valid states in the diagram with the values of the inputs clearly shown beside each edge connecting the nodes.

Part 2 - Verilog Implementation and Simulation

The controller can be broken down into two parts.

- State Machine – The state machine is the sequential circuit that takes care of determining what the output pattern should be as a result of the current state and the current inputs.
- Combinational Logic – This is needed to generate the actual signals for the LEDs. This logic block takes the light pattern from the state machine and drives the LEDs with different signals depending on whether the light should be off or dim (as controlled by the `RUNLIGHT` input). You need to come up with a the combinational logic for this function either through intuitive reasoning or by completing and minimizing a truth table. Note that we abstract this behavior out of the state machine because this function doesn't depend on what the state of the machine is.

Figure 3 demonstrates how the state machine and the combinational logic block make up the controller.

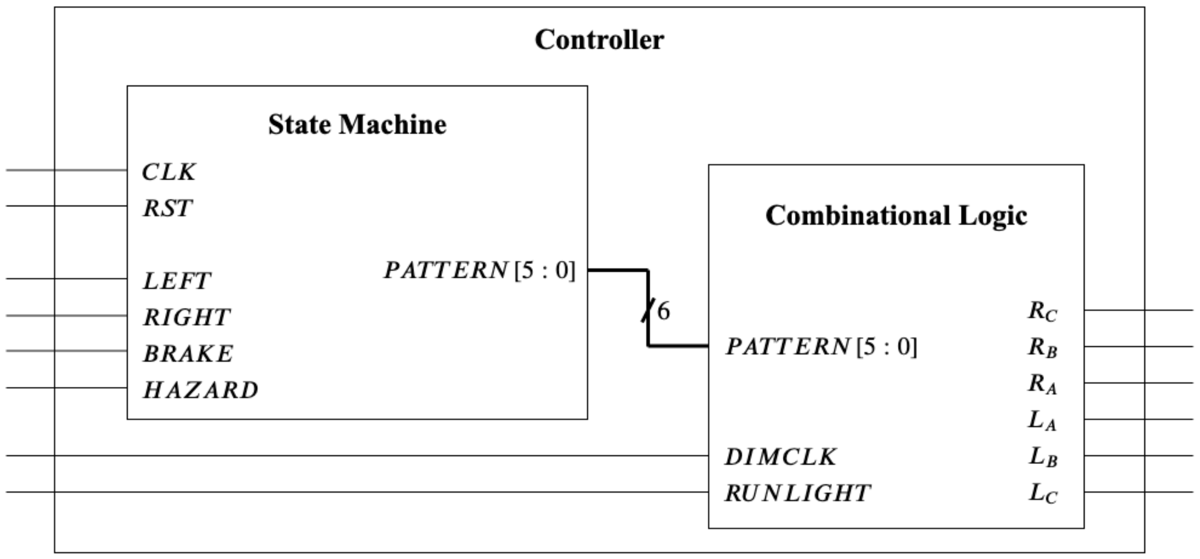


Figure 3

Keeping the structure of the controller in mind, implement the controller using Verilog. To implement the state machine, you can use always blocks and case statements (see the following pseudocode). **Note: purely using case statements for next state will result in a much larger set of cases than is reasonable. We recommend you use either nested cases or some combination of case and if/else statements**

```
// Implement transition diagram
always@(*) begin
    case({state,inputs})
        L_0:  nextState = S_another_state;
        L_1:  nextState = S_yet_another_state;
        // ...
        default:  nextState = S_default_state; // Take care of
                                               // default behavior
    endcase
end

// Moore machine outputs depend on only the current state
always@(*) begin
    case(state)
        S_0:  output = O_some_output;
        S_1:  output = O_yet_another_output;
        // ...
        default:  output = O_default_output; // Take care of
                                             // default behavior
    endcase
end

always@(posedge clk) state <= nextState;
```

After implementing your controller, construct a testbench to simulate your design. Take screenshots of your waveforms and write a paragraph or two explaining why your waveform shows that the controller functions correctly. **You should debug your code extensively during simulations so that problems in the next section are mostly limited to bugs in the hardware.**

Lab Report

Please submit a report that summarizes your work for Lab 4. The report should include:

- The work you did for Part 1.
- All of the Verilog code you wrote for the lab.
- Your waveform screenshots and the paragraphs in which you explain how your waveforms prove the functionality of the tail light controller.

The lab report is due on Canvas by Tuesday, March 5, 2024 before 11:59 PM. One report is sufficient for each group. Each partner should submit the report individually on Canvas.

Part 3 - FPGA Implementation

Synthesize your Verilog program, assign general purpose I/O pins for your circuit, and download your design onto the FPGA. **Note: The clock inputs should be connected to pins 17, 20, 89, or 92.**

Use the function generator to provide the following signals:

- `CLK` - 1 Hz square wave
- `DIMCLK` - 100 Hz square wave

If you don't remember how to generate a square wave with the function generator, please reference lab 3. So as to not damage your FPGA, double check these inputs before plugging them into your breadboard or the fpga.