ECE 152A LABORATORY 4

Objective:

This lab introduces you to sequential logic and state machines. You will design and implement a state machine and run it on the FPGA board. The purpose of the lab is to imitate the tail light controller of a 1965 Ford Thunderbird.

Deliverables and Due Dates:

- **There is NO written pre-lab assignment for Lab # 4.** (But read this handout before you come to the lab.)
- Part 1 is due in your lab section one week after the lab begins.
- **Lab # 4 checkout**: Parts 2, 3 and 4 are due two weeks after the lab begins.

(See the syllabus or the course web page for exact due dates.)

Grading:

Grading will be done as follow for the steps completed on time:

- Part 1: 20 %
- Part 2: 20 %
- Part 3: 20 %
- Part 4: 40 %

Design Strategy:

- **Part 1:**

  Our aim is to map the given word problem in Sections 4.1 and 4.2 to a Moore state machine for the tail light controller. The following are the steps to follow:
  - Clearly define:
    - The state variables you use.
    - The inputs and outputs you use.
  - Draw a state diagram for the controller.
    - Show all the valid states in the state diagram with the values of the inputs clearly shown besides each edge connecting the nodes.
  - Draw a state table for the controller.
    - Clearly show each combination of present states and inputs that you use.
    - Clearly show all don't care combinations in your state table.
  - We will NOT do K-map minimization! The point is to write down a state diagram and a state table and then implement it using behavioral Verilog.

- **Part 2:**
  - Implement the state machine in Verilog and check the correctness of the design by simulating it with Modelsim.

- **<u>Part 3:</u>**
  - Download the Verilog program onto the FPGA board and run your state machine on it.
- **<u>Part 4:</u>**
  - Interface your circuit to the computer that serves as simple input console and controls the inputs to the state machine. You have to write a small C/C++ program that generates a clock and the different signals.

4.1 Introduction

The objective of this experiment is to build a circuit that imitates the tail lights of a 1965 Ford Thunderbird. Among the tail light functions are left and right turn signals, brake lights, flashing hazard and running lights.

There are three lights on each side and for turns they operate in sequence to show the turning direction. When the hazard lights are activated, all six lights flash in unison. The brake activates all lights as long as the brake pedal is pressed. When the car's running lights are on, all lights that would normally be off are turned on at 50% brightness. Note that several functions can be activated simultaneously and the experiment description will tell you how to handle those situations.

This lab consists of four major stages. First, you will design a Moore state machine for the tail light controller. Then you will implement your state machine in Verilog, simulate and debug it on the computer. Then, you will build a simple hardware circuit using the FPGA board and 6 LEDs and run your state machine on it. Your circuit will be interfaced to the computer, which serves as simple input console and controls the inputs to the state machine. Finally, you will write a small C/C++ program that generates a clock and the different signals (left, right, brake, hazard, running lights).

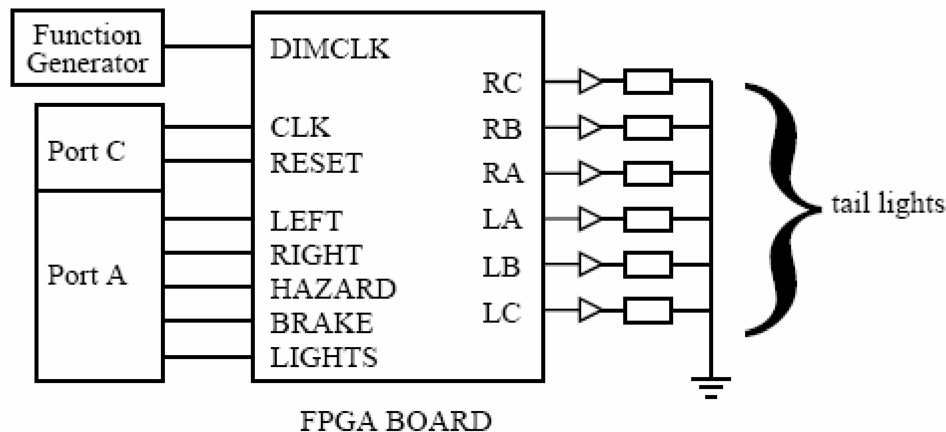

Figure 4.1: Block diagram of lab #4.

4.2 State Machine

4.2.1 Introduction

In the following sections, you will design and extend a Moore state machine to implement the different functions of the tail light controller. Inputs to the state machine are the clock CLK and the light functions as shown in Figure 4.1- you can ignore the

dimmer input DIMCLK for now. There are six outputs out of the state machine to indicate the status of the six lights.

Answer the following questions before you start your machine, they are supposed to help you keep your design simple. What does each state represent in terms of the LED outputs and their different lighting patterns? What is the default state? What the corresponding default output? How many different lighting patterns are there? How many states should your state machine have (minimum number)?

4.2.2 Left and right turn signals

Flashing Sequence

Figure 4.2 shows the flashing sequence for the left and right turns. There are three lights on each side and they operate in sequence to indicate the turning direction. Initially, all three lights on either side are dark. When a turn signal is activated, the sequence begins by activating the lights in sequence until all three lights are lit. The sequence then begins again with all lights turned off.

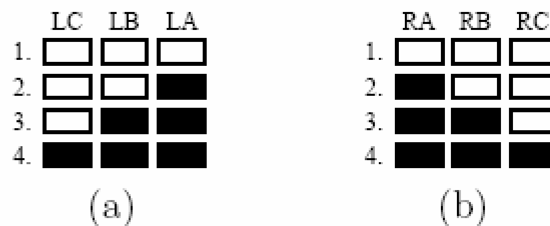

Figure 4.2: Flashing sequence for (a) left and (b) right turns.

The operation of the left and right signals is as follows:

- If the turn signal is turned off in the middle of the sequence, the sequence is aborted and all lights are turned off.
- Both turn signals could be activated simultaneously, but the machine should only execute one turn sequence. Therefore the machine has to give priority to one turn signal over the other.
- Later, when you add the hazard lights, the state machine should go to hazard when both left and right turn signals are on.

4.2.3 Brake lights

When the brake input is asserted, there are two possible cases:

- When brake is asserted, all six lights should be on.

- If any turn signal and brake are asserted, the corresponding side should continue the turning sequence. The lights on the other side should be indicating brake (all on).

4.2.4 Hazard lights

Here is the description of the hazard lights function:

- When the hazard input is asserted, the lights should flash (all on, all off) alternatingly.
- When hazard is active, it should override any turn signal.
- If both turn signals are active, the machine should go to hazard.

4.2.5 Priorities for multiple signals

It is possible that multiple input signals are active at the same time. In that case, the state machine needs to give priority to certain functions over others:

- If brake and hazard are both activated, then brake overrides the hazard lights.
- If brake is asserted, it should override hazard, but comply with a single turn signal.
- Both turn signals active indicates hazard, which still has lower priority than the brake signal.

4.2.6 Running Lights

Description

When the LIGHTS input is asserted, the tail-lights should be turned on at 50% brightness when they would normally be dark. The other functions remain unchanged.

Since this is a digital laboratory, there is no direct way to generate the dimming effect. Instead, a square wave with a 50% duty cycle and a high enough frequency will be used. This way, the lights are constantly turned on and off. Above a certain frequency (60-100Hz), the human eye cannot detect the flickering any more and perceives a dimmed intensity.

Implementation

This function is not part of the state machine in the strict sense. Your core state machine only implements the functionality of the turn signals, brake and hazard lights. To add the running lights feature, do not modify your state machine.

It is better to add some combinatorial logic to your design. Your state machine now does not drive the LEDs directly any more. Think of your state machine as a “black box" with a set of inputs and outputs (the six lights). You will create another black box that will generate the actual signals for the LEDs. As input, this black box takes control signals.

You should think of the state machine outputs as a set of signals that are inputs to your combinatorial logic. Together with the DIMCLK input (which receives the square wave), your logic should generate the final output signals to drive the actual LEDs.

The actual LED outputs can be expressed as a simple combinatorial function of the state machine outputs, the running lights input (LIGHTS) and the DIMCLK input. Develop this function through intuitive reasoning or by completing and minimizing a truth table.

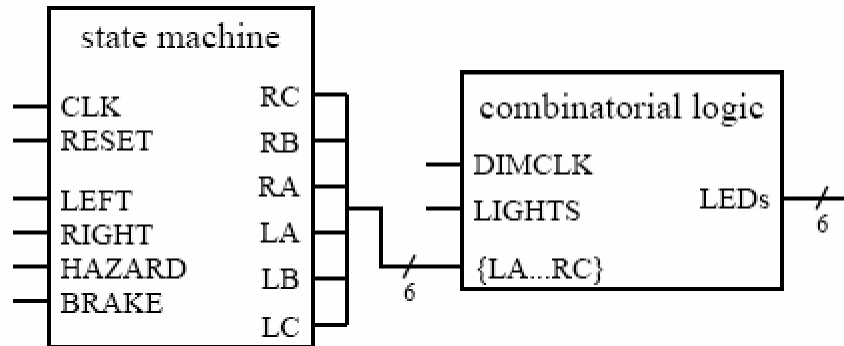

Figure 4.3: Design of the running lights logic function.

4.3 Verilog Program

4.3.1 Programming

Create a new project in Xilinx ISE and implement your state machine step by step in Verilog. The Verilog tutorial attached contains a sample Verilog program that shows how to implement a Moore state machine. It is strongly suggested that you stick closely to the given code structure -- in the past there have been problems with other coding styles.

4.3.2 Reset

Your state machine should have a well-defined default state, i.e. all lights off. To ensure that it begins in that state, you need to add a reset signal (RESET) that overrides everything else. This reset can be synchronous or asynchronous.

4.3.3 Simulation & Debugging

Before you proceed to running your state machine on the hardware, simulate it in ModelSim to verify that it operates correctly. This way you will be able to completely debug your design before you download it to the FPGA board. Once you are confident that your state machine works correctly, your debugging will be limited to circuit and FPGA board problems.

4.3.4 Downloading

Synthesize your Verilog program, do the I/O pin assignments (refer to the document "digital I/O Board" on the lab datasheets page) and then download your design onto the FPGA. If you use interface pins on the FPGA connected to the parallel port, you probably have to disconnect the board from the rest of your circuit.

4.4 Hardware

4.4.1 FPGA Board

Before you use your FPGA board or connect it to other components of your circuit, execute the test program GXSTEST (refer to the XSA50 board manual on lab datasheets page).

If the board tests fine, you can plug it into the breadboard. It usually needs a little "convincing" to fit -- apply a reasonable amount of pressure. Be careful not to bend any pins, the FPGA board is expensive!

A 9 V DC adapter supplies power to the FPGA board. Each workstation should have one. If it is missing or broken, you can also manually power the FPGA board-- refer to the XSA50 board manual on the lab datasheets page. No matter how you power the FPGA board, **you need to connect its GND pin to the common circuit ground!**

Now wire the FPGA to the other components in your circuit. You can flexibly choose which board pins to use, so your wiring will depend on that. Most of the FPGA inputs will be provided through the computer, so these will be wired to the 50-pin connector (see section 4.4.2.) **The clock inputs should be connected to the pins 15 or 18.**

The six FPGA outputs should be wired to the LEDs. However, the LEDs must be driven with buffers between the FPGA and the LEDs or with the Vcc/inverter-setup from experiment #1. Do not forget the series resistors! When you use the inverter setup from experiment #1, you can either use an inverter chip (e.g. 7404) or invert the FPGA outputs.

4.4.2 Computer I/O Interface

Plug the 50-pin connector into the breadboard. You may have to adjust its metal wires a little bit until they fit. For the connector pinout and more information on the digital I/O interface, refer to "Digital I/O Board" document on lab datasheets page. Do not forget to ground pin 50 --**all parts of your circuit need to have the same common ground!**

Your circuit should use two I/O ports:

- Port A for the function inputs (LEFT, RIGHT, BRAKE, HAZARD, LIGHTS).

- Port B or port C for the control signals (CLK, RESET).

4.4.3 Function Generator

Set the function generator to provide a square wave with a 50% duty cycle and a frequency around 100Hz. Connect the output of the oscillator to the DIMCLK input of the FPGA.

This clock is used to dim the LEDs as explained earlier.

4.5 Software

4.5.1 Introduction

Your software should be written to simulate the controls of the car. The program should not implement any functionality related to the tail light controller --these functions MUST be implemented in the state machine on the FPGA. It should take input from the console to toggle the different functions and signal those to the FPGA (without any processing).

4.5.2 Keys

Table 4.1 lists the key assignments for the different signals. Each key should immediately toggle the corresponding function, i.e. the first press should enable and the next press should disable the signal. The user should not have to press any other key besides the respective function key to perform the desired function. Your program should display the status of all signals (on/off) on the screen.

Note that it is possible to have several signals active at the same time, e.g. a turn signal, brake, hazard and lights. Your software should not assume any priority in this situation. All signals should be sent to the FPGA and the state machine should properly handle these situations.

key	signal
L	left turn
R	right turn
B	brake
H	hazard
O	running lights

Table 4.1: Key assignments for the different tail light functions.

4.5.3 Clock

While the software is managing this input and output, it must also clock the state machine via port C. The clock frequency should be about 2-3 Hz. Do not make it too slow, or your TA will fall asleep during checkout!

4.5.4 Reset

Your program should also control the reset input of the FPGA. Reset the FPGA for a short duration at the beginning of the program. During normal state machine operation, the reset must not be used, e.g. helping the state machine to turn off certain lights.