LAB 4 – Finite-State Machines

Goals

- Learn how to implement a finite-state machine using Verilog.
- Design and implement a simple circuit that emulates the blinking lights of a Ford Thunderbird.

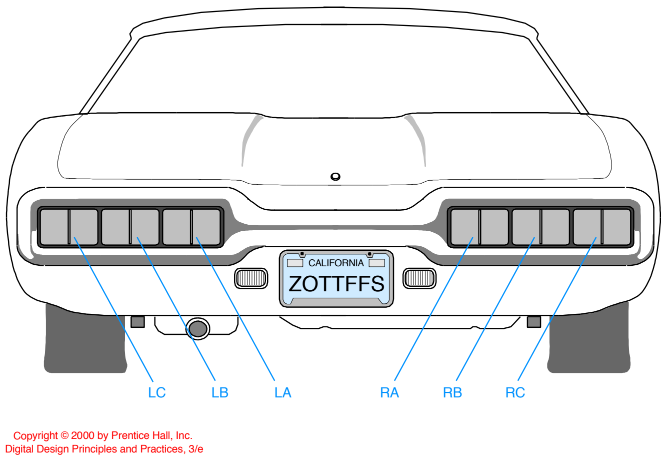
To Do

- Understand how the clock signal is derived in the FPGA board.
- Write an FSM that implements the Ford Thunderbird blinking sequence.
- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.
- **To complete the lab you must show your work before the deadline, following the instructions provided by the assistant assigned to your group. There is nothing to hand in. The required tasks are clearly marked with gray background throughout this document.**

Introduction

Until now, we have only implemented combinatorial circuits in Verilog. In this exercise, we will implement circuits with states, i.e., sequential logic circuits. Please refer to Lectures 3 and 4a to understand sequential logic.

In this lab, you will design a finite-state machine to control the tail lights of a 1965 Ford Thunderbird. There are three lights on each side that operate in sequence to indicate the direction of a turn. Figure 1 shows the tail lights and Figure 2 shows the flashing sequence for (a) left turns and (b) right turns.[1]


CALIFORNIA
ZOTTFFS
LC
LB
LA
RA
RB
RC
Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

[1] This lab is derived from an example by John Wakerly from the 3rd Edition of Digital Design.

Figure 1. Thunderbird Tail Lights

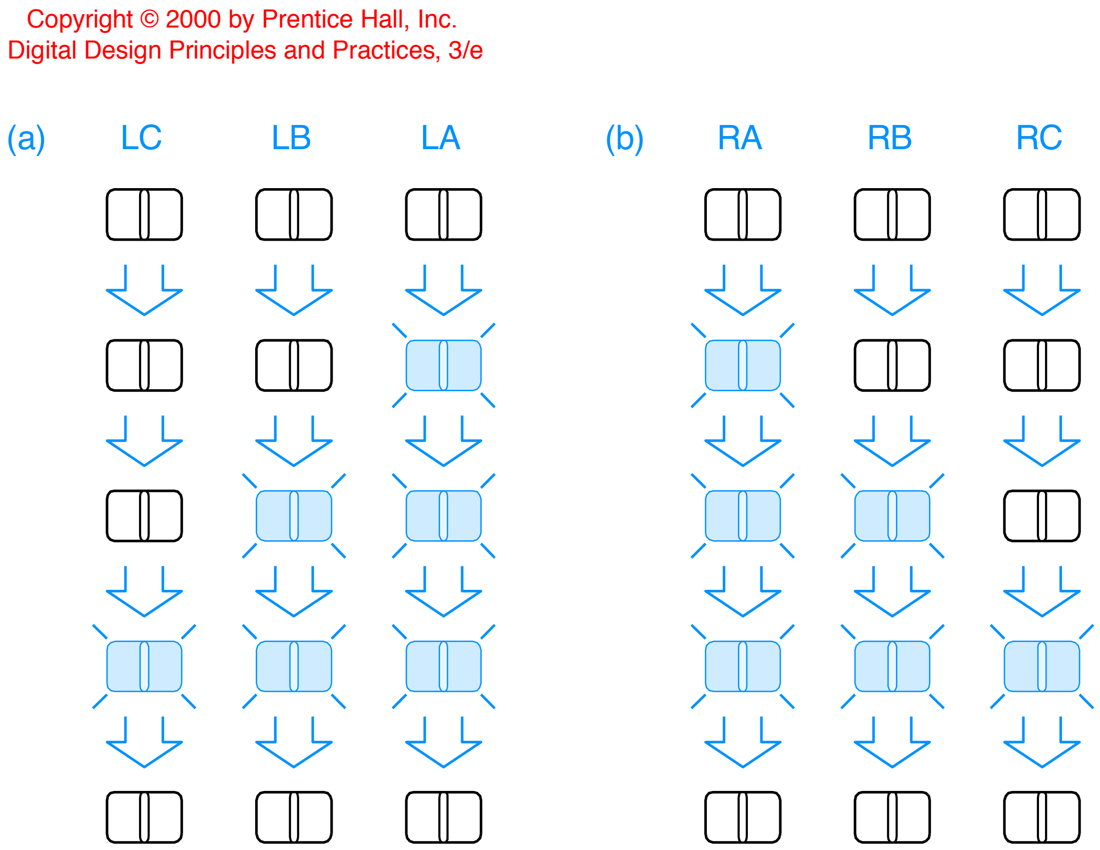

Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e
(a)
LC
LB
LA
(b)
RA
RB
RC

Figure 2. Flashing Sequence (shaded lights are illuminated)

Part 1 – FSM Design

Let us start with designing the state transition diagram for this FSM. Give each state a name and remember to assign a value to *every* output (**LC, LB, LA, RA, RB, RC**) in each state. Your FSM should take three inputs: **reset**, **left**, and **right**. The circuit should have the following properties:

- On reset, the FSM should enter a state with all lights off.
- When you press **left**, you should see LA, then LA and LB, then LA, LB, and LC, then finally all lights off again. This is illustrated in Figure 2 (a).
- 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.
- When you press **right**, you should see RA, then RA and RB, then RA, RB, and RC, then finally all lights off again. This is illustrated in Figure 2 (b).
- It is up to you to decide what to do if the user makes **left** and **right** simultaneously true. However, be sure to make a choice — this will keep your design *simple* and make writing the Verilog code a lot easier.

1a) Using a pen, draw the state transition diagram. Indicate the input conditions that cause transitions among states. Use the 1-bit signals **‘left’** and **‘right’** for the direction indicator. These will be the circuit’s inputs.

The job of an FSM is to do three things:

1. Determine the next state from the present state and the inputs (next state logic).
2. Determine the output signals based on the present state and input signals (output logic).
3. Advance from the present state to next state when a clock event arrives (state register).

In section 3.4 of the H&H textbook, you see how to prepare a state transition table. This is essentially a translation of the state diagrams into a table form so that we can use our knowledge in designing combinatorial circuits to derive the Boolean equations for next state logic and output logic.

1b) Along with your state transition diagram, add a small table describing how you will map the states to binary values.

The next step is the output mapping. There are six output signals driving the tail lights: **LA, LB, LC**, **RA, RB, RC**. The output logic table defines the values of output signals at each state. We use this description to derive our output logic.

1c) Along with your state transition diagram, add a small table or a text/pseudocode description of how you will map the states to these six output signals.

Now that we have a design, we are ready to start our project!

Part 2 – Verilog Implementation

Refer to earlier labs if you have problems with creating a new project.

A Matter of Style

The syntax of hardware description languages will allow you to write the same FSM in a number of different ways. Please refer to Lecture 5a for the best practices. We expect you to clearly separate the following three components:

- the state register (this register keeps track of the present state; at every clock cycle, the present state is replaced by the next state);
- the next state logic (where the next state is determined by the present state and current inputs);
- the output logic (where the outputs are determined by the present state and inputs);

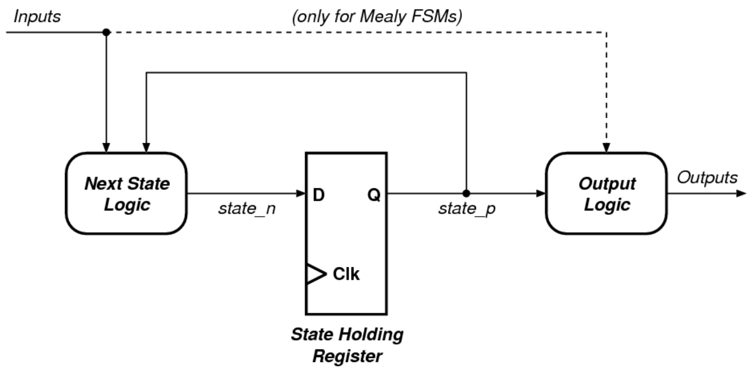


Figure 3. A simple view of an FSM.

It is also good practice to name your signals intuitively. If you want to know how every registered signal is connected, please refer to Figure 3.

In this example, there are two signals associated with the **'state'**. The next state signal is called **'state_n'** (short for "state-next") and the present state signal is called **'state_p'** (short for "state-present"). This is just an example, but it is good practice to use the same **'basename'** and to add some identifiers as a suffix to differentiate the present and next states. In this way, you are always able to distinguish which signals are registered. Note that your textbook does not follow the same naming convention.

If we use a hardware description language (e.g., Verilog or VHDL) we do not need to explicitly fill a state transition table, because we will use the compiler to first map the behavior to Boolean equations, and then automatically to logic gates.

Most importantly, **be consistent in the way you write code.** This allows you to more easily validate, reuse and share code with others.

Start Vivado and create a new project. Create a Verilog source file to implement the finite-state machine using what your knowledge of Verilog. *Be sure to include a clock (**clk**) and a reset signal (**rst**) as inputs to your circuit.*

Part 3- Implementing the Clock

You described a clock input (**clk**) in your circuit. Where does this clock come from? You might be tempted to use one of the FPGA's push buttons or switches for this purpose. This is not a good idea: these mechanical components generally do not transition from one logic-level to another one cleanly. An example of such a noisy transition is shown in Figure 4.

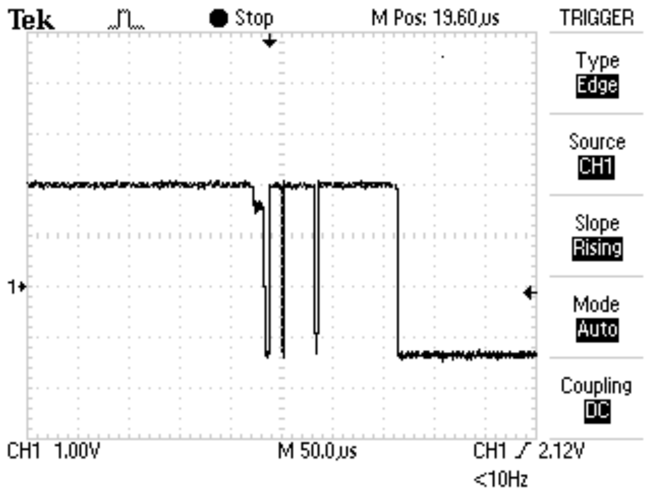


Figure 4. Push-button bounce (taken from www.labbookpages.co.uk)

What causes *bounces*, as these noisy transitions are known, is that the rate of change in a push button is very slow compared to the speed of the FPGA (more than a million times slower). During the slow transition, the FPGA sees many fast-occurring transitions and interprets each of them as a clock edge. Usually it is possible to use *debouncing circuits* and sampling techniques to prevent this. Push buttons will not work — we need a safer way to generate the clock signal.

If you look at your board, you may notice the text *"CLK100Mhz (W5)"*. In fact, your board contains a 100Mhz crystal oscillator circuit. The output of this oscillator is connected to the pin W5 (which is a special clock pin for this FPGA). We will simply make sure in our design to connect the net "clk" to the pin W5 using the constraints file (XDC). In this way, we will have a clean clock signal. We will take care of this in Part 4, when we define the constraints.

Now we have a different practical problem: the clock is too fast. 100 MHz means that every clock period is 10 nanoseconds long. The entire blinking sequence would be then 40 ns. This is a very short time: a human needs at least a few milliseconds (five orders of magnitude larger than 40 ns) to recognize an image. If we want to see our sequence, we need to find a way to dramatically slow down the circuit.

This can be achieved in two ways. We can either implement a clock divider circuit that divides the clock by a few million times, or we can generate an enable signal every few million cycles, and then use this enable signal to control our next state transition.

In this exercise, we will give you a small clock divider circuit *(clk_div)* that takes as input the *'clk'* and *'rst'* signals and generates a '*clk_en'* signal every 33'554'432 cycles (or once every 2^{25} cycles). Considering that the main clock frequency is 100'000'000 cycles per second, this means a *'clk_en'* signal is generated every 0.335s.

```
module clk_div(input clk, input rst, output clk_en);
 reg [24:0] clk_count;
 always @ (posedge clk)
 // posedge defines a rising edge (transition from 0 to 1)
  begin
   if (rst)
    clk_count <= 0;
   else
    clk_count <= clk_count + 1;
  end
 assign clk_en = &clk_count;
endmodule
```

The idea is simple: we increment a 25-bit counter (called *clk_cnt*) at every clock cycle and set *clk_en* to ‘1’ when all the bits of the counter are ‘1’. By increasing the counter size, you can change the division factor as you please.

Make the necessary changes to integrate this divider into your code. We recommend that you create a second Verilog file and add the *clk_div* as a second module to the project. Then you can instantiate the *clk_div* in your top module. In the top module, you have to modify the ‘state register’, so that it only updates the state when the *clk_en* signal is ‘1’.

We encourage you to play with the frequency of the *clk_en* signal to test how fast your eye can capture a blinking light. If you see the light not blinking but weaker than usual, it means that your eye is just averaging the brightness of the light. In this case, your eye is not detecting the blinking light.

Part 4 – Defining the Constraints

Now, all we have to do is to choose which buttons on the board we want to use for the control, and which LEDs we will use as tail lights. Therefore, we need to provide a constraints file to tell Vivado where we want to connect our pins.

Add and open a new constraint file to your project. Refer to Lab 2 for more information.

Enter the following constraints into your constraints file. Make sure that you have consistent port names in the constraints file and your top module.

```
# Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -period 10 -waveform {0 5} [get_ports clk]

# LEDs
set_property PACKAGE_PIN U14 [get_ports LC]
set_property PACKAGE_PIN U15 [get_ports LB]
set_property PACKAGE_PIN W18 [get_ports LA]
set_property PACKAGE_PIN U19 [get_ports RA]
set_property PACKAGE_PIN E19 [get_ports RB]
set_property PACKAGE_PIN U16 [get_ports RC]
set_property IOSTANDARD LVCMOS33 [get_ports {LC LB LA RA RB RC}]

# Buttons
set_property PACKAGE_PIN W19 [get_ports left]
set_property PACKAGE_PIN U18 [get_ports reset]
set_property PACKAGE_PIN T17 [get_ports right]
set_property IOSTANDARD LVCMOS33 [get_ports {left reset right}]
```

Our design is now ready to be implemented.

Using the previous labs as a guide, generate the bitstream file of your circuit and download it to the FPGA board. Show your assistant that the circuit is working correctly.

Last Words

Once again in this lab exercise, you had to make several decisions. Choosing the state encoding is such a decision:

- You could design an FSM with seven states. You may choose to decode the seven states with at least three bits. If you do this, you will save on the number of state-holding registers, but you will need a larger output decoding circuit to derive the LED controls from the state. This will be a similar to the decoder circuit that we designed in Lab 3. Instead of 4 inputs and 7 outputs, we have 3 inputs and 6 outputs.
- You could use 6 bits to decode the seven states cleverly, and directly use the output of the state register to drive the LEDs. This will save you the output decoder, but you will need twice the number of state-holding flip-flops.
- One other option is to realize the fact that the left and right blinking operations are essentially the same. Perhaps it would be possible to design just one state machine and use it twice?

There is no clear better choice. As you design more circuits, you will develop your intuition, and even your own preferences.

Until now we were able to see whether our circuits functioned correctly by directly observing them. This is only possible because our circuits had very few outputs, and it did not take much time to go through all of them. With the coming exercises, this will start to change, and we will need better methods to see if our circuit actually works.

Part 1a

Part 1b

Part 1c