

LAB 4 – Finite State Machines

Goals

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

To Do

- Understand how the clock signal is derived in the FPGA system.
- Write an FSM that implements the Ford Thunderbird blinking sequence.
- Optional: implement a dimming function, so that the lights are not only on and off, but can have intermediate levels.
- 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 have to show your work to an assistant during any lab session, there is nothing to hand in.

Introduction

Until now, we have only designed combinatorial circuits. In this lab we start with circuits with distinct states. We once again use Verilog to specify our design.

In this lab, you'll design a finite state machine to control the taillights of a 1965 Ford Thunderbird¹. There are three lights on each side that operate in sequence to indicate the direction of a turn. Figure 1 shows the taillights and Figure 2 shows the flashing sequence for (a) left turns and (b) right turns.

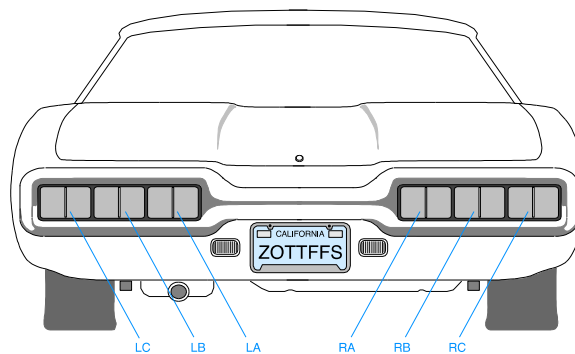


Figure 1. Thunderbird Tail Lights

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

Both the car and the flashing sequence are uncommon in Europe, but this is a good exercise to learn how to design a simple Finite State Machine (FSM).

Let us start with designing the state transition diagram for this FSM. Give each state a name and indicate the values of the six outputs LC, LB, LA, RA, RB, and RC in each state. Your FSM should take three inputs: **reset**, **left**, and **right**. The circuit should have the following properties:

- Using a pen, draw the state transition diagram on the report sheet. Indicate the input conditions that causes transitions among states. Use the signals 'left' and 'right' for the direction indicator.

1. Determine the next state from the present state, and the inputs (next state logic).
2. Realize the output function based on the present state and the inputs if a Mealy FSM is used (output logic).
3. Advance from the present state to next state when a clock event arrives (state register).

2

knowledge in designing combinatorial circuits to derive the Boolean equations for next state logic and output logic.

If we use a hardware description language (Verilog or VHDL) we do not really need to fill a state transition table, because we will use the compiler to map the behavior to first Boolean equations and then to logic gates automatically. What we have to do is to still determine how to encode the states and assign binary values to the states.

Add a small table describing, how you will map the states to binary values, to your report.

The next step is the output mapping. We have six outputs, LA, LB, LC and RA, RB, RC that each drive the light signals. The output logic table can be used to determine what happens in each state. We use this description to derive our output logic.

Add a small table or a verbal description of, how you will map the states to the 6 output lights, to your report.

Now we are essentially ready with the design and we can start our project.

Start Vivado and create a new project (you could call it Lab4). Add a new Verilog source and implement the finite state machine using what you have learned in the lecture about Verilog. Be sure to include a clock input (clk) and a reset (rst) in your circuit.

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

Question of Style

The syntax of hardware description languages will allow you to write the same FSM in a number of different ways. It has been our experience that clearly separating

- the state register (where clock moves the next state to the present state)
- the next state logic (where the next state is determined by the present state and inputs)
- the output logic (where the outputs are determined by the present state and inputs)

works the best. This style is also used in the examples in the textbook.

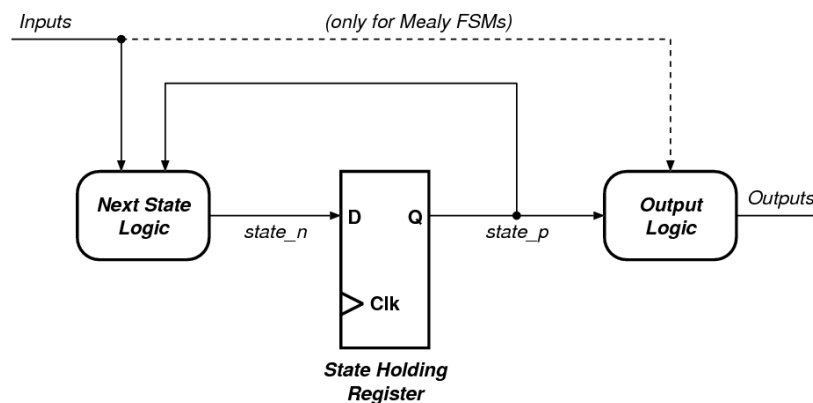


Figure 3. A simple view of an FSM.

It is also good practice to use a naming style that clearly identifies the signals that are registered. If you want to know how every registered signal is connected, please see Figure 3 as a reference.

In this example there are two signals associated with the ‘state’. The next state signal is called ‘state_n’ and the present state signal is called ‘state_p’. 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 in your textbook the examples are not in this fashion, but we still believe it makes life easier.

The important thing is to be consistent in the way you write the code. This allows you to find problems easier, makes code re-use easier, and allows others to understand your code easier.

The problem with the clock

You have described a clock input (clk) in your circuit. The question is where do we get this clock from? You could be tempted to use one of the push buttons or the switches for this purpose. This is not a good idea, because these mechanical components, generally do not transition from one logic-level to another one cleanly, but exhibit multiple transitions as seen in Figure 4.

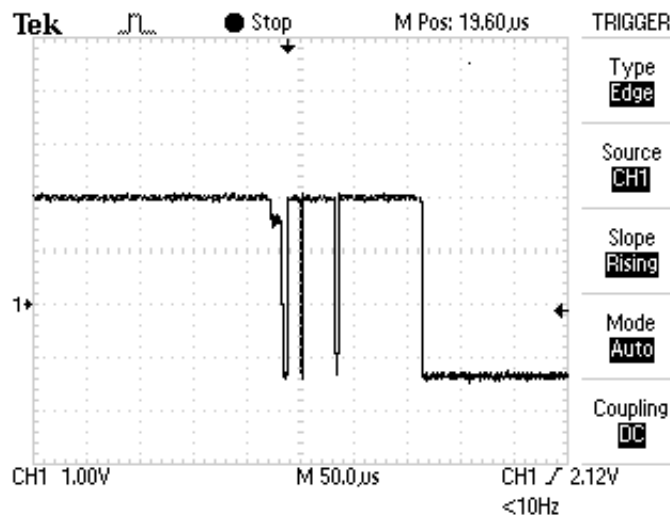


Figure 4. Push-button bounce (taken from www.labbookpages.co.uk)

The problem is that compared to the speed of the FPGA the change in a push button is very very slow (in fact more than a million times slower). During the slow transition the FPGA will see many fast occurring transitions, and would interpret each of them as a clock edge. This is known as ‘bounce’, and usually specialized circuits (and sampling techniques) are used to prevent this. In any case, using the push buttons is not a very safe way of generating the clock.

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 tell

the compiler to connect the net “clk” to the pin W5 in the constraints file (XDC). In this way we will have a clean clock signal.

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: light travels less than 250 meters during that time. If we want to see our sequence, we need to find a way to dramatically slow down the circuit.

This can be achieved by two means. Either we 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 `clk_div` circuit that takes in the same ‘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 pretty simple, we increment a 25-bit counter (called `clk_cnt`) at every clock, and set the `clk_en` to ‘1’ when all the bits of the counter is ‘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 main file. In the main file, you have to modify the ‘state register’, so that it only updates the state when the `clk_en` signal is ‘1’.

The constraints file

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 taillights. Therefore, we need to provide a constraint file to tell Vivado where we want to connect our pins.

Add and open a new constraint file to your project (Add sources, Add or create constraints). Refer to Lab 2 for more information.

Enter the following constraints:

```

# Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -add -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 old exercises as a guide, implement the circuit and map it to the FPGA board. Show an 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 decode 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 that derives the LED controls from the state. This will be a similar circuit that we designed in Lab3, 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 for driving the LEDs. This will save you the output decoder, but you will need twice the number of state holding flip-flops.

There is no clear better choice. As you continue doing more circuit design, you will develop your own preferences. 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?

Until now we were able to see whether our circuits functioned correctly by directly observing them because the circuits had very few outputs, and it did not take much time to see 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.

Digital Circuits Laboratory Exercises

Lab 4: Finite State Machines

| | | |
|--------------|--|-----------|
| Date | | Grade |
| Group Number | | |
| Names | | Assistant |
| | | |

Part 1 Draw the state transition diagram

Part 2 Draw your state encoding table

Part 3 (Signature Required) Show an assistant your code and circuit.