Lab 2: Verilog Synthesis and FSMs

University of California, Berkeley

Department of Electrical Engineering and Computer Sciences

EECS150 Components and Design Techniques for Digital Systems

John Wawrzynek, James Parker, Daiwei Li

Due February 13th 2013 @ 2PM

Contents

0 Introduction

In this lab you will be introduced to behavioral Verilog, and one of the more important tasks of the tools, logic synthesis. Completing this lab will give you experience describing hardware quickly, and at a high level. In addition you will take a close look at the logic that is produced by the CAD tools, and any inefficiencies produced in the translation from text to FPGA implementation.

In this lab we have given you the framework to build a two digit combination lock. Your job will be to implement three separate modules and complete the lock. The inputs to your lock will consist of a rotary dial and two push buttons. The rotary dial, represented by the signals `FPGA_ROTARY_INCA` and `FPGA_ROTARY_INCB`, is used to select a digit of the combination. The rotary dial push button, represented by the signal `FPGA_ROTARY_PUSH` is used to enter the digit selected. LEDs are used to indicate that the lock has been opened, and also that an incorrect combination has been entered.

1 Prelab

Note: Please make sure to complete the items in this section *before* coming to lab. You most likely will not finish the lab during your section if you do not do the prelab.

For the prelab, complete the following items:

1. Read this entire handout thoroughly. In particular, the Lab Details section gives you lots of valuable details on the circuit you must build. Please also read the Verilog FSM Tutorial. This will be helpful in understanding the FSM implementation methodology you will use in this lab and future labs.
2. Download the lab files. As always the lab distribution can be obtained by:
   ```
   wget http://inst.eecs.berkeley.edu/~cs150/sp13/lab2/lab2.tar.gz
   tar –xzvf lab2.tar.gz
   ```
3. Write your Verilog ahead of time. You are responsible for implementing `Lab2Lock` and `Lab2Counter`.
4. Answer the prelab questions.

Questions

1. How many D-type flip-flops do you think the synthesis tools will infer in `Lab2Lock`? Think about this carefully. What will D-type flip-flops be used for in this module?
2. Given the Verilog you came up with for `Lab2Counter`, what do you think the logic synthesis tools will produce? Suggest a circuit that implements `Lab2Counter`. Draw an RTL diagram of this circuit on the back of the checkoff sheet. You need not go in to great detail - simply show a very high level structure of the circuit.

2 Design Details

Overview

Figure 1 shows a diagram of the circuit we want you to implement.

Most of the modules have been provided for you; you are only responsible for implementing `Lab2Lock` and `Lab2Counter`. A short explanation of the rest of the modules in the system can be found in the following sections.

One possible point of confusion is the existence of two separate reset signals, `LockReset` and `SystemReset`. This separation exists in the FPGA_TOP_ML505.v file to allow the lock to be reset independently of the rest of the system, such as the counter, in the event that the user would like to input another combination.

The `SystemReset` will reset everything including the lock and counter. The `LockReset` will only reset the lock but preserve the state of the counter. This difference is only relevant to the top level module so you do not have to worry about it in the module you will be working with. However, you will need to show that these resets work correctly as intended after you finish your implementation.

The `SystemReset` corresponds to the button left of the GPIO LEDs. It should be marked CPU_RST.

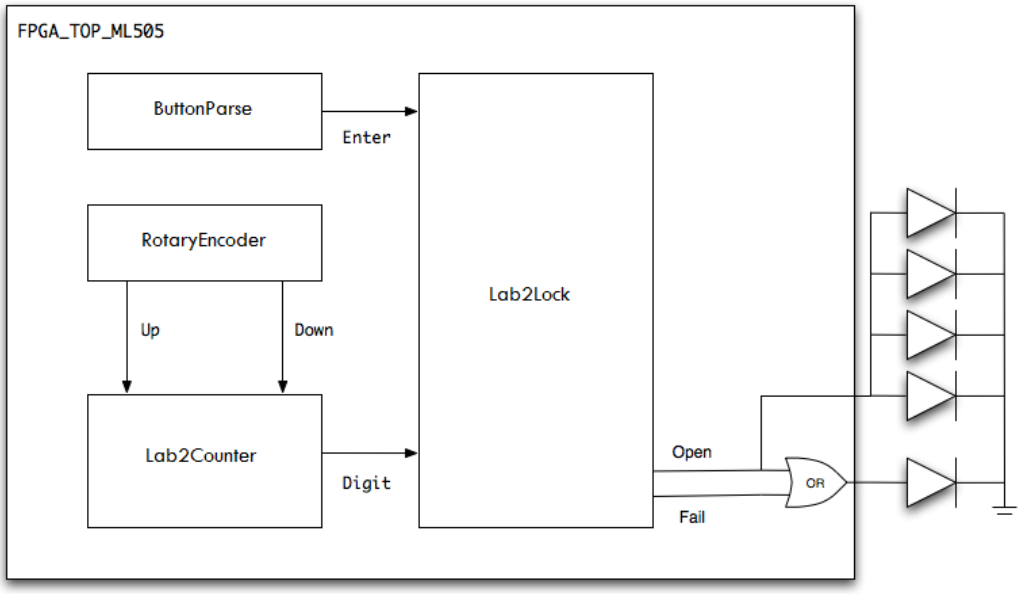


Figure 1 Block Diagram of FPGA_TOP_ML505

ButtonParse

This module has been built for you.

Raw inputs from push buttons are often very noisy, oscillating for a short time before settling on a value. This module filters its input into a clean **1-cycle pulse**. It is a fairly complex circuit so you do not need to understand the internal details of how it works.

Refer to the following table for the `ButtonParse` port specification.

Signal	Width	Dir	Description
`Clock`	1	In	The clock
`Reset`	1	In	Resets the state of the `ButtonParse`
`Enable`	1	In	Enables the output
`In`	Variable	In	Raw input from a button
`Out`	Variable	Out	Filtered single pulse output

RotaryEncoder

This module has also been built for you.

The `RotaryEncoder` module parses raw input from the rotary encoder (which is the wheel found on the lower right corner of the ML505 board). This module decodes changes in the 2-bit state of the encoder in to pulses of either the `Up` or the `Down` signal. It is important to note that **the signal from the encoder will be pulsed 4 times for each click of the wheel**. You must account for this in your counter implementation.

Refer to the following table for the `RotaryEncoder` port specification.

Signal	Width	Dir	Description
`Clock`	1	In	The clock
`Reset`	1	In	Resets the state of the `RotaryEncoder`
`A`	1	In	Raw rotary encoder signal from `FPGA_ROTARY_INCA`
`B`	1	In	Raw rotary encoder signal from `FPGA_ROTARY_INCB`
`Up`	1	Out	Pulses high during clockwise clicks of the wheel.
`Down`	1	Out	Pulses high during counter-clockwise clicks of the wheel.

Lab2Counter

We will be using the `Lab2Counter` in conjunction with the `RotaryEncoder` as a means of inputting the digits of the combination into our combination lock. We will increment the `Count` output of the lock when the wheel is clicked once counter-clockwise, and decrement when the wheel is spun clockwise.

Slightly complicating the task of designing this module is the fact that, as noted in the specification of the `RotaryEncoder`, we **only want the `Count` output to change once every 4 clicks** of the `Increment` or `Decrement` signal. You will have to figure out a way to deal with this in your design. For robustness, your design may not assume that your counter will receive 4 consecutive increment pulses or 4 consecutive decrement pulses at a time.

Signal	Width	Dir	Description
`Clock`	1	In	The clock
`Reset`	1	In	Resets the count to 0
`Increment`	1	In	Increment pulses from the `RotaryEncoder`
`Decrement`	1	In	Decrement pulses from the `RotaryEncoder`
`Count`	4	Out	The current value of the counter

Lab2Lock

This module is responsible for maintaining the state of the lock and generating outputs which indicate the lock's status (these will be pulled out to LEDs).

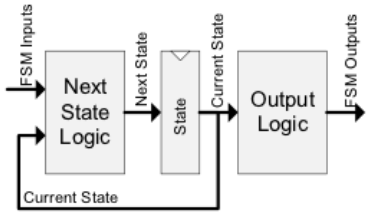
This module should detect DIGIT1 and DIGIT2 being entered in the correct order. You should use the Verilog `localparam` statement in order to define the values of these constants inside your module. The following snippet shows how this is done:

```
/* Inside the Lab2Lock module */
localparam DIGIT1 = 4'h2;
localparam DIGIT2 = 4'h3;
```

Please use the above values for the combination to your lock to make it easier to test (already done for you in the lab distribution). You may change these later if you want to test with other values but for

checkoff please use the values provided for you.

You will build this module as a finite state machine. In particular, you should build a **Moore** machine to accomplish this task. A Moore machine depends only on its current state to generate its output (not, for instance, on both its current state and its current inputs). Therefore, your `Lab2Lock` will have the following high level organization:



Remember the tips we have given you for building FSMs in Verilog. If you're not absolutely sure about how to proceed, please review the comprehensive Verilog FSM Tutorial.

Both the port specification and the state transition diagram for this module follow.

Signal	Width	Dir	Description
`Clock`	1	In	The clock
`Reset`	1	In	Resets the lock to a set state
`Enter`	1	In	Tells the lock to read in a new digit
`Digit`	4	In	A digit of the combination
`State`	3	Out	The state of the FSM, for debugging purposes
`Open`	1	Out	Indicates that the lock is open
`Fail`	1	Out	Indicates that the user has entered the incorrect combo

When you are ready to test your implementation, build your design and impact it on the board. First try entering the correct combination and see if it is successful. The current digit should be displayed on the GPIO LEDs 0 to 3 and should only increment or decrement by one value for each click of the wheel. The rotary encoder also doubles as a button that you can push inwards into the board. You can enter each digit by pushing the rotary encoder button.

A correct implementation should light up all four cardinal direction LEDs by the compass buttons on a successful combination sequence. In the event of an incorrect combination sequence, only the center compass button LED should light up indicating a failure. It should also not light up prematurely after seeing only one incorrect digit. For debugging purposes, the state of your lock is shown on GPIO LEDs 5 to 7.

The lock reset can be triggered by pressing the center compass button. Again this will only reset the lock FSM, not perform a system wide reset. The system can be reset by pressing the CPU RESET button to the left of the GPIO LEDs. Use these resets appropriately to test your implementation.

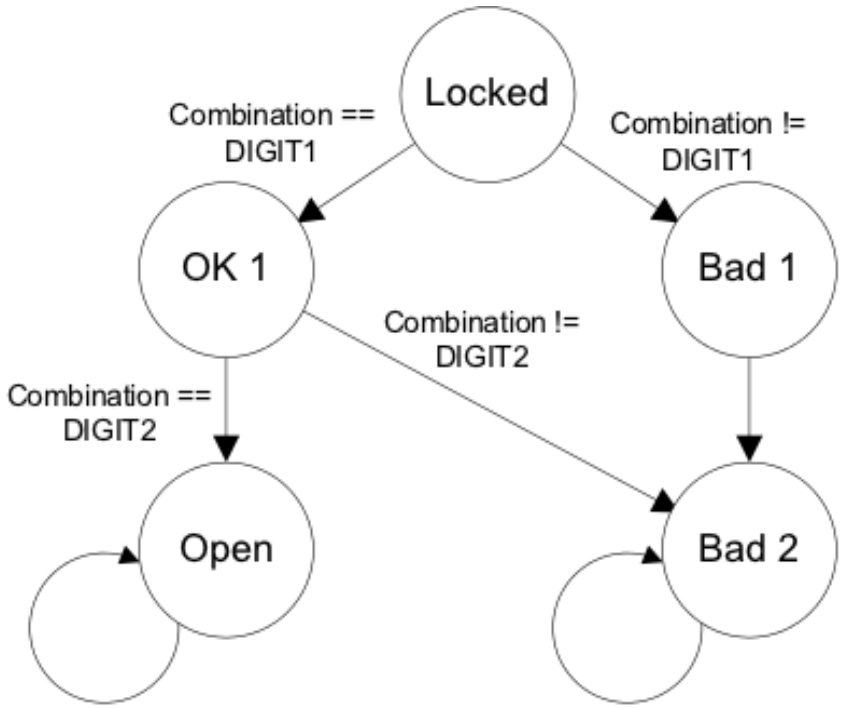


Figure 2 Lab2Lock FSM

3 Tips and Common Mistakes

Although the modules you will have to implement in this lab are fairly small and straightforward, this is the first time you will be dealing with both combinational and sequential logic in Verilog. Below are some tips and common mistakes that students make during their first time trying to write Verilog:

Combinational vs. Sequential

It is always good to look through the interface you need to deal with before you start coding. For the Lab2Lock and Lab2Counter, first figure out what state you need to preserve across clock edges if any. Then separate out what signals are relevant to the combinational aspect of the module and what signals are necessary to determine the next state values for your registers. This will help you keep track of what signals should be used when and where.

Multiple always @ Blocks

For this lab you will need to use multiple always @ <port list> blocks. One of these will be for sequential logic and the other will be for combinational logic to assign signal values. One common mistake is to assign a signal (ex. bad_signal) in both the combinational block `always @ (posedge clock)` and sequential block `always @ (*)`.

Example:

```
reg bad_signal;
always @ (posedge clock) begin
       bad_signal <= bad_inpt1;
```

```
end
always @ (*) begin
       bad_signal <= bad_input2;
end
```

This will result in a doubly driven wire since it is possibly to assign two different signals at the same time to `bad_signal`. Always try to keep all the assignments for a particular signal in one `always @` block to avoid this problem and do not assign signals in both the combinational block and sequential block.

Counter is Incrementing by 4

As stated previously in the lab, the Lab2Counter module sees 4 pulses per click of the wheel. Make sure that you correctly counted these pulses and applied appropriate truncation/shifting/etc. so that it outputs the correct bits.

4 Analysis and Checkoff

You should now have a complete and working combination lock. In order to build this circuit you did not need to specify the individual implementation of each functional unit. Instead, you provided the synthesis tool with a high level textual representation of the behavior you wanted, and the synthesizer did it's best to match this description. You now have enough experience to have seen how drastically high level synthesis can affect development times. You will now take a look at how well the synthesizer does its job.

1. In the prelab you speculated a possible circuit for synthesized Lab2Counter. Find this module in the RTL schematic and compare the circuit created by the synthesis tool to your own.

2. Collect some general information about the design. Remember you can do this by looking at the report generated after the build finishes.

    - Number of occupied SLICEs
    - Number of SLICE LUTs
    - Number of SLICE registers (used as flip-flops)

For checkoff demonstrate your working lock and counter implementation by showing both a successful combination and a failure. Also demonstrate both the lock reset and system reset capabilities and show your answers to the prelab and analysis questions.