ECEN 248 - Lab Report

Lab Number: 10

Lab Title: A Simple Digital Combination Lock

Section Number: 513

Student's Name: Abhishek Bhattacharyya

Student's UIN: 731002289

Date: 4/19/2023

TA: Hesam Mazaheri

Objectives:

The main goal of this lab is to design a prototype combination lock using a finite state machine (FSM) programmed onto the Zybo FPGA boards. This lab explores both a 3-password and a 4-password combination lock, where each password is a 4-bit binary number (digits 0-15). The correct password is “13-7-9” for the 3-password combination lock, and the desired behavior for the FSM is to start at state S0, step through states S1-S3 if presented with a correct password in each progressive state, and to unlock the lock if the correct password is presented in the last state. If an incorrect password is presented at any point, the FSM resets to state S0 and the whole combination must be entered from the start.

This lab is a first exploration into building and designing FSMs in Verilog code. The main objectives are to gain a stronger understanding of how FSMs function, how they can be used on hardware in the real world by testing them on the Zybo FPGA board, and to gain a familiarity with how to code FSMs in Verilog, which is an important basic skill used in computer engineering.

Design:

All of the deliverables for this lab were coded from the base code and instructions provided in the lab manual. Some of the code was given in full, and this code has been included here as well for completeness.

The main part of this lab was coding an FSM module for a combination lock, which was completed in the prelab. During the lab session, this code was programmed onto the Zybo FPGA board using Verilog’s hardware manager and tested for functionality.

Verilog code for all modules coded and tested is provided below.

```
`timescale 1ns / 1ps
`default_nettype none
/* This module describes the combination-lock FSM
described in the prelab using behavioral Verilog */
module combination_lock_fsm(
    output reg [1:0] state,
    output wire [3:0] Lock, // asserted when locked
    input wire Key1, // unlock button 1
    input wire Key2, // unlock button 2
    input wire [3:0] Password, // indicate number
    input wire Reset, // reset
    input wire Clk // clock
    );
    parameter S0 = 2'b00,
              S1 = 2'b01,
              S2 = 2'b10,
              S3 = 2'b11;
    assign Lock = (state == S3)? 4'b1111 : 4'b0000;
    reg[1:0] nextState; // will be driven in always block

    // here we start setting up what we see in the FSM
    always@(*)
       begin
//         state <= nextState;
        case(state)
            // here we start at s0 and see where we go from there
            S0: begin
                if(Key1 == 1'b1 & Password == 4'b1101)
                // 1101 being 13 in binary
                    nextState <= S1;
                else
                    nextState <= S0;
            end
            // here we start at s1 and see where we go from there
            S1: begin
                if(Key2 == 1'b1 & Password == 4'b0111)
                // 0111 being 7 in binary
                    nextState <= S2;
                else if (Key2 == 1'b1 & Password != 4'b0111)
                    nextState <= S0;
                else
                    nextState <= S1;
            end
            // here we start at s2 and see where we go from there
            S2: begin
                if(Key1 == 1'b1 & Password == 4'b1001)
                // 1001 being 9 in binary
                    nextState <= S3;
                else if (Key1 == 1'b1 & Password != 4'b1001)
                    nextState <= S0;
                else
                    nextState <= S2;
            end
            // here we start at s3 and see where we go from there
            S3: begin
                if(Reset)
                    nextState <= S0;
                else
                    nextState <= S3;
            end
        endcase
       end
       // here we set up the clk and reset
       always@(posedge Clk)
           if(Reset)
               state <= S0;
           else
               state <= nextState;
endmodule
```

Figure 1: *Source code for 3-password combination lock FSM*

```
`timescale 1ns / 1ps
`default_nettype none
/* This module describes the combination-lock FSM
described in the prelab using behavioral Verilog */
module combination_lock_fsm(
    output reg [2:0] state,
    output wire [3:0] Lock, // asserted when locked
    input wire Key1, // unlock button 1
    input wire Key2, // unlock button 2
    input wire [3:0] Password, // indicate number
    input wire Reset, // reset
    input wire Clk // clock
    );
    parameter S0 = 3'b000,
              S1 = 3'b001,
              S2 = 3'b010,
              S3 = 3'b011,
              S4 = 3'b100;
    assign Lock = (state == S4)? 4'b1111 : 4'b0000;
    reg[2:0] nextState; // will be driven in always block

    // here we start setting up what we see in the FSM
    always@(*)
       begin
//         state <= nextState;
        case(state)
            // here we start at s0 and see where we go from there
            S0: begin
                if(Key1 == 1'b1 & Password == 4'b1101)
                // 1101 being 13 in binary
                    nextState <= S1;
                else
                    nextState <= S0;
            end
            // here we start at s1 and see where we go from there
            S1: begin
                if(Key2 == 1'b1 & Password == 4'b0111)
                // 0111 being 7 in binary
                    nextState <= S2;
                else if (Key2 == 1'b1 & Password != 4'b0111)
                    nextState <= S0;
                else
                    nextState <= S1;
            end
            // here we start at s2 and see where we go from there
            S2: begin
                if(Key1 == 1'b1 & Password == 4'b1001)
                // 1001 being 9 in binary
                    nextState <= S3;
                else if (Key1 == 1'b1 & Password != 4'b1001)
                    nextState <= S0;
                else
                    nextState <= S2;
            end
            // here we start at s3 and see where we go from there
            S3: begin
                if(Key2 == 1'b1 & Password == 4'b1011)
                // 1011 being 11 in binary
                    nextState <= S4;
                else if (Key2 == 1'b1 & Password != 4'b1011)
                    nextState <= S0;
                else
                    nextState <= S3;
            end
            // here we start at s4 and see where we go from there
            S4: begin
                 if(Reset)
                     nextState <= S0;
                 else
                     nextState <= S4;
            end
        endcase
       end
       // here we set up the clk and reset
       always@(posedge Clk)
           if(Reset)
               state <= S0;
           else
               state <= nextState;
endmodule
```

Figure 2: *Source code for 4-password combination lock FSM*

```
/*This is the top-level module for our digital *
 *rotary combination-lock based on the diagram *
 *provide in the lab manual                    */

module combination_lock(
    output wire [3:0] LEDs,//connect to lock
    /*Let's output state for debugging!*/
    output wire [2:0] JB,
    input wire Clk,
    input wire Key1,Key2,
    input wire Reset,
    input wire [3:0]Password
);

    /*intermediate nets*/
    wire KeySync1,KeySync2, ResetSync;


    /*synchronize button inputs*/
    synchronizer syncA(KeySync1, Key1, Clk);
    synchronizer syncB(KeySync2, Key2, Clk);
    synchronizer syncC(ResetSync, Reset, Clk);

    /*wire up combination lock FSM*/
    combination_lock_fsm U1(
        .Lock(LEDs),
        .state(JB),
        .Clk(Clk),
        .Key1(KeySync1),
        .Key2(KeySync2),
        .Reset(ResetSync),
        .Password(Password)
    );


endmodule
```

Figure 3: *Source code for Combination Lock Module (as given)*

```
/*This module provides the synchronization   *
 *necessary to prevent metastability when    *
 *transitioning from an asynchronous to a    *
 *synchronous domain. In other words, when   *
 *we bring an input signal in from the FPGA  *
 *board into a clocked domain, we must do    *
 *this buffering!                            */
`timescale 1ns /1ps
module synchronizer(
     output wire OutSignal,
     input wire InSignal,
     input wire Clk
);

     /*intermediate nets*/
     reg buff0, buff1, buff2;

     always@(posedge Clk)
            begin
               buff0 <= InSignal;
               buff1 <= buff0;
               buff2  <= buff1;
            end

     assign OutSignal = buff2;

endmodule
```

Figure 4: *Source code for Synchronizer Module (as given)*

```
#LEDs
#IO_L23P_T3_35
set_property PACKAGE_PIN M14 [get_ports {LEDs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[0]}]
##IO_L23N_T3_35
set_property PACKAGE_PIN M15 [get_ports {LEDs[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[1]}]
##IO_0_35
set_property PACKAGE_PIN G14 [get_ports {LEDs[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[2]}]
##IO_L3N_T0_DQS_AD1N_35
set_property PACKAGE_PIN D18 [get_ports {LEDs[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[3]}]

##Pmod Header JB - the new pins are for JC not JB Pmod
##IO_L15N_T2_DQS_34
#set_property PACKAGE_PIN U20 [get_ports {JB[0]}]
set_property PACKAGE_PIN V15 [get_ports {JB[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {JB[0]}]
##IO_L15P_T2_DQS_34
#set_property PACKAGE_PIN T20 [get_ports {JB[1]}]
set_property PACKAGE_PIN W15 [get_ports {JB[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {JB[1]}]
##IO_L16N_T2_34
#set_property PACKAGE_PIN W20 [get_ports {JB[2]}]
set_property PACKAGE_PIN T11 [get_ports {JB[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {JB[2]}]

##BUTTONS
##IO_L20N_T3_34
#set_property PACKAGE_PIN V16 [get_ports {Reset}]
set_property PACKAGE_PIN K19 [get_ports {Reset}]
set_property IOSTANDARD LVCMOS33 [get_ports {Reset}]
##IO_L20N_T3_34
#set_property PACKAGE_PIN R18 [get_ports {Key1}]
set_property PACKAGE_PIN K18 [get_ports {Key1}]
set_property IOSTANDARD LVCMOS33 [get_ports {Key1}]
##IO_L20N_T3_34
set_property PACKAGE_PIN P16 [get_ports {Key2}]
set_property IOSTANDARD LVCMOS33 [get_ports {Key2}]

##Switches
#IO_L19N_T3_VREF_35
set_property PACKAGE_PIN G15 [get_ports {Password[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Password[0]}]
#IO_L24P_T3_34
set_property PACKAGE_PIN P15 [get_ports {Password[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Password[1]}]
#IO_L4N_T0_34
set_property PACKAGE_PIN W13 [get_ports {Password[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Password[2]}]
#IO_L9P_T1_DQS_34
set_property PACKAGE_PIN T16 [get_ports {Password[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Password[3]}]

##Clock signal
##IO_L11P_T1_SRCC_35
#set_property PACKAGE_PIN L16 [get_ports Clk]
set_property PACKAGE_PIN K17 [get_ports Clk]
set_property IOSTANDARD LVCMOS33 [get_ports Clk]
create_clock -add -name sys_clk_pin -period 20.00 -waveform {0 8} [get_ports Clk]

```

Figure 5: *XDC Design Constraint code for Combination Lock (as given)*

Results:

This lab had only one simulation section (Experiment Part 1d). Included below are the results of the simulation with the console output indicating that all tests pass.

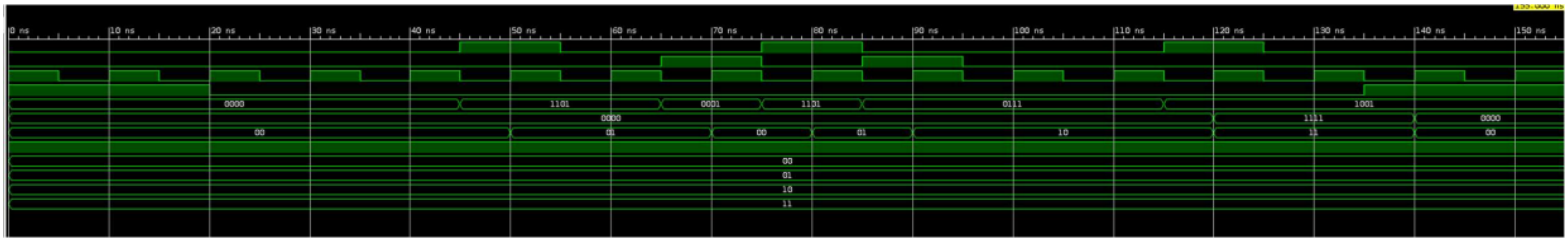

Figure 6: *Waveform for 3-Password Combination Lock FSM Simulation*

```
#       send_msg_id Add_Wave-1 WARNING "No top level
#    }
# }
# run 1000ns
All Tests Passed!!!
INFO: [USF-XSim-96] XSim completed. Design snapshot
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 7: *Console output for 3-Password Combination Lock FSM Simulation*

Conclusion:

This lab was successfully completed, with all objectives completed and deliverables completed. Being able to enter the lab with most of the code completed from the prelab allowed for a greater focus on working with the Zybo boards and understanding how the FSM works, which was extremely helpful. After importing the code from the prelab into Vivado and debugging any small errors, the testbench for the 3-password combination lock passed without issues. Testing on the Zybo board for the 3-password combination lock also went smoothly. Finally, the more secure design with the 4-password combination was implemented and tested, which also went without issue.

In all, this lab greatly helped to improve understanding of how to work with FSMs in Verilog and in real-life hardware. Being able to focus on testing and the hardware's operation rather than spending hours troubleshooting Verilog code was very beneficial and greatly improved the learning value and ease of workflow of this lab.

Post-Lab Deliverables:

1. Include the source code with comments for all modules you simulated and/or implemented in lab. You do not have to include test bench code that was provided! Code without comments will not be accepted!
    a. 3-password combination lock code
    b. 4-password combination lock code

All code for this lab is included in the *Design* section.

2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.
    a. 3-password combination lock simulation waveform

All simulation waveforms with console output are included in the *Results* section.

3. Answer all questions throughout the lab manual.
    a. Experiment Part 1, 1e: Likewise, take a look at the simulation waveform and take note of the tests that the test bench performs. Is this an exhaustive test (i.e., are all possible cases covered by our test)? Why or why not?

No, this is NOT an exhaustive test. Looking into the combination_lock_fsm_tb file, the testbench tests the one correct combination (13-7-9). However, the testbench only tests ONE incorrect password: "1".

The testbench operates like this: First, it tests if the FSM holds at S0 with no input, then enters 13 and tests if the FSM moves to state S1 as it should, then presents the wrong password "1" and tests if the FSM returns back to S0 as it should. If all of these tests pass, the testbench then enters 13, checks for state S1; enters 7, checks for state S2; and enters 9, then checks for state S3. Finally, the testbench attempts a reset, and if all the prior tests passed, the testbench reports all tests passing.

In all, only ONE wrong password is tested ("1") and the behavior of entering a wrong password is ONLY tested while the FSM is in state S1. In order for the testing to be more thorough, the testbench would need to test the basic functionality of the FSM (which it does), and also present a wrong password in each state (S0, S1, S2, and S3) and check each state resets to S0 as it should. For the test to be exhaustive, the testbench would have to check all possible input values in each state (S0 through S3). Since this amounts to $16^3 = 4096$ combinations tested, it is easy to see that testing all of these is unfeasible, which is likely why the testbench author opted to test only the most important input cases.

4. A possible attack on your combination-lock is a brute-force attack in which every possible input combination is tried.

a. Given the original design with a combination of three numbers between 0 and 15, how many possible input combinations exist?

There are 16 possible numbers to pick (1-15, and 0). Picking 3 numbers out of 16 possibilities gives $16^3 = 4096$ possible combinations (assuming numbers in the combination can be repeated).

b. How about for the modified design with a combination of four numbers?

There are 16 possible numbers to pick (1-15, and 0). Picking 4 numbers out of 16 possibilities gives $16^4 = 65536$ possible combinations (assuming numbers in the combination can be repeated).

It is clear to see that a brute force attack is untenable to do by hand. However, brute-forcing the electronic lock using another computer is possible in a realistic timeframe. One way to protect against this is by adding a timeout feature that makes the FSM pause execution for a set time (say 10 seconds) every time a wrong password is presented.

Important Student Feedback

1. What did you like most about the lab assignment and why? What did you like least about it and why?

Being able to enter the lab with most of the code completed from the prelab allowed for a greater focus on working with the Zybo boards and understanding how the FSM works, which was extremely helpful. I enjoyed being able to focus on working with the hardware and was very satisfied seeing my code work. The part I most disliked was having to ensure my code from the prelab worked and debugging it.

2. Were there any sections of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?

The lab manual was clear and concise to understand.

3. What suggestions do you have to improve the overall lab assignment?

I do not have any suggestions to improve the lab assignment.