ECE272 Lab 4
Spring 2018

Driving multi-digit LED displays
Isak Foshay

May 13^{th}, 2018

1. Introduction

This lab aims to teach students about system verilog and all of it's charm. It aims to teach students the concept of digital wires and the transfer of data through software via digital wires. This lab also teaches about state machines (However simple the state machine used is it is still a state machine). The lab also touches on persistence of vision.

2. Design

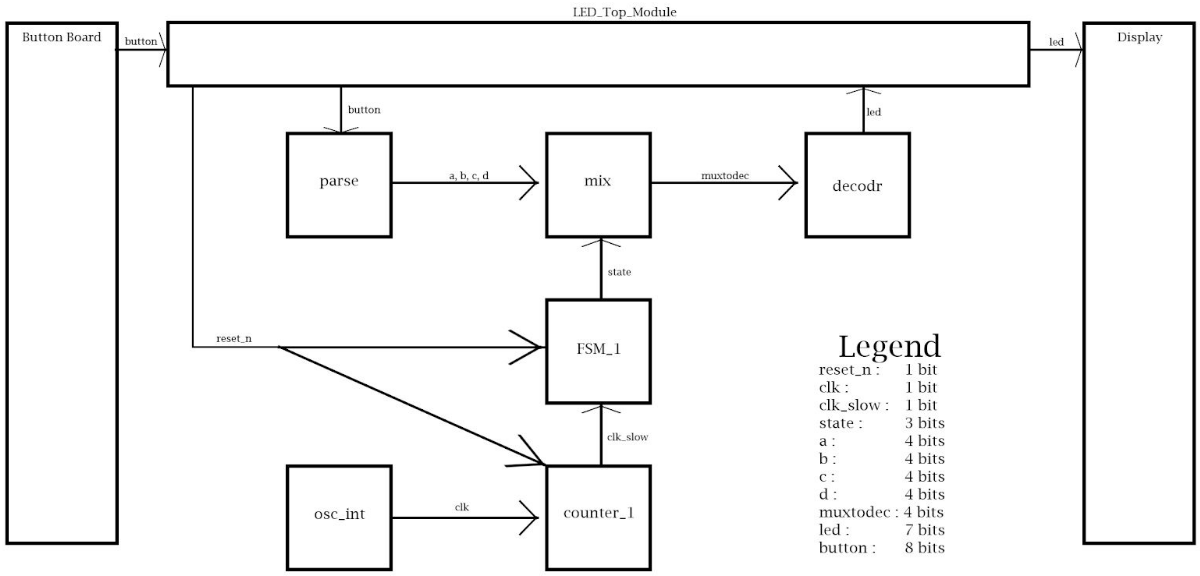

Figure 1: Design of how software works

This project took a long time so some of the names are 5th or 6th attempts at making it work so the names had devolved into semi-obscurity. Particularly mux turned into mix and decoder turned into decodr. They still get their point across and they work so I kept it as is as a tribute to the modules that finally worked. Reset comes from the top module so I had to get creative with the arrows for reset_n. I feel this accurately portrays the flow of how the software works.

	FPGA PIN	PULLMODE
button[0]	F5	UP
button[1]	E3	UP
button[2]	B1	UP
button[3]	C1	UP
button[4]	D2	UP
button[5]	E2	UP
button[6]	F2	UP
button[7]	G1	UP
reset_n	C2	UP
led[0]	G14	DOWN
led[1]	B16	DOWN
led[2]	D14	DOWN
led[3]	F14	DOWN
led[4]	D16	DOWN
led[5]	C15	DOWN
led[6]	E16	DOWN
state[0]	E7	DOWN
state[1]	E8	DOWN
state[2]	F9	DOWN

Table 1: Chosen Pins and Pull Modes

I tried to group my pins together in an at least semi consistent manner. I also made sure to assign reset_n to a pin far from my other pins I was actively using and plugging things into.

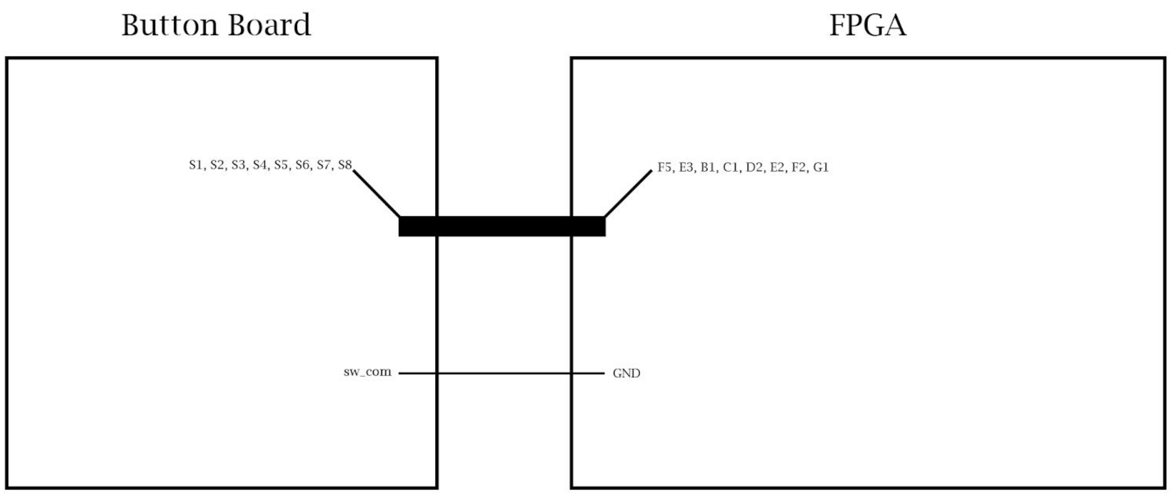

Figure 2a: Block Diagram for Hardware

This diagram shows that we are using all 8 buttons on our button board, and also shows which pin to connect each button to.

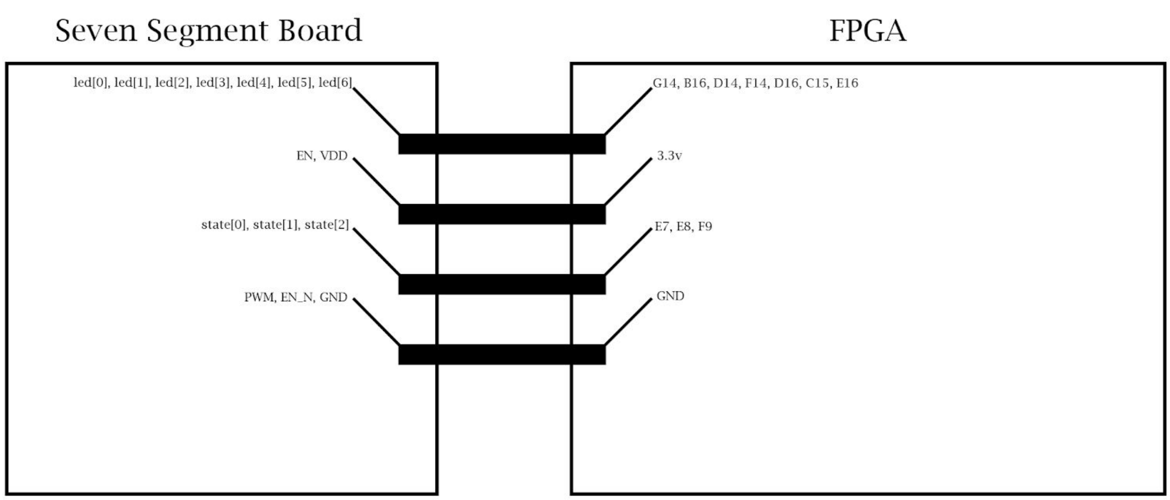

Figure 2b: Block Diagram for Hardware

This diagram shows that the state selection is no longer up to the user like lab 3 and is instead handled by the FPGA itself. It also informs whoever is setting up the wires what needs to be plugged into what.

3. Results

My result was a working seven segment display that could display on each digit and did not appear to flicker.

4. Experiment Notes

I attended 5 labs in order to finish this, as well as many many hours outside of lab. I partly blame this lab for my end of term weight gain because of how much this project stressed me out. There was little to no system verilog documentation available anywhere and it took me hours to figure out how things worked. This lab was the most difficult and stressful lab I have ever taken. But at the same time I learned the most from this lab. However, there are definitely less painful and time consuming ways to learn system verilog.

Study Questions:

1. **Describe the differences between Mealy and Moore state machines..**

   Mealy does an action when you land on the state, Moore you do an action when moving to a new state.

Appendix

Code:

```
module clock_counter(
	input logic clk_i,			//often, "tags" are added to variables to denote what they do for the user
	input logic reset_n,		//here, 'i' is used for input and 'o' for the output, while 'n' specifies
						//an active low signal ("not")
	output logic clk_o
		);

		logic [13:0] count;	//register stores the counter value so that it can be modified
					//on a clock edge. Register size needs to store as large of a
					//number as the counter reaches. Here, 2^(13+1) = 16,384.

		always_ff @ (posedge clk_i, negedge reset_n)
			begin
				count <= count + 1;	//at every positive edge, the counter is increased by 1
				if(!reset_n) //If reset_n gets pulled to ground (active low), reset count to 0
					begin
						clk_o <= 0;
						count <= 0;
					end
				else
					if(count >= 5000) //Flips the slow clock every 10000 clock cycles
						begin
							clk_o <= ~clk_o;	//Flip slow clock
							count <= 0;				//Reset the counter
						end
			end
endmodule
```

```
module LED_top_module(
	/*************************/
	/* Set inputs and outputs */
	/* to the whole FPGA here */
	/*************************/
	input logic reset_n, //be sure to set this input to PullUp, or connect the pin to 3.3V
	input logic [7:0] button,
	output logic [2:0] state,
	output logic [6:0] led
	);
		/*****************************/
		/* Set internal variables here */
		/*****************************/
		logic clk;		//used for the oscillator's 2.08 MHz clock
		logic clk_slow;	//used for slowed down, 5 Hz clock
		logic [3:0] a;
		logic [3:0] b;
		logic [3:0] c;
		logic [3:0] d;
		logic [3:0] muxtodec;
		/**********************/
		/* Define modules here */
		/**********************/
		parser parse (
		.inp(button),
		.a(a),
		.b(b),
```

```
        .c(c),
        .d(d)
        );

        mux2 mix (
        .a(a),
        .b(b),
        .c(c),
        .d(d),
        .s(state),
        .y(muxtodec));

        sevenseg decodr (
        .data(muxtodec),
        .segments(led)
        );

        //This is an instance of a special, built in module that accesses our chip's oscillator
        OSCH #("2.08") osc_int (        //"2.08" specifies the operating frequency, 2.08 MHz.
                                        //Other clock frequencies can be found in the
    MachX02's documentation
                .STDBY(1'b0),           //Specifies active state
                .OSC(clk),              //Outputs clock signal to 'clk' net
                .SEDSTDBY());           //Leaves SEDSTDBY pin unconnected


        //This module is instantiated from another file, 'Clock_Counter.sv'
        //It will take an input clock, slow it down based on parameters set inside of the module, and
        //output the new clock. Reset functionality is also built-in
        clock_counter counter_1(
                .clk_i(clk),
                .reset_n(reset_n),
                .clk_o(clk_slow));

        //This module is instantiated from another file, 'State_Machine.sv'
        //It contains a Moore state machine that will take a clock and reset, and output a state
        state_machine FSM_1(
                .clk_i(clk_slow),
                .reset_n(reset_n),
                .state(state)
                );

        /***********************************************/
        /* Add modules for:                                         */
        /* Parser          Determines the 1000's, 100's,   */
        /*                         10's and 1's place of the number*/
        /* Multiplexer     Determines which parser output  */
        /*                         to pass to the decoder             */
        /* Decoder         Convert 4-bit binary to 7-seg   */
        /*                         output for numbers 0-9             */
        /***********************************************/


endmodule
```

```
module state_machine( //example of a Moore type state machine
    input logic clk_i,
    input logic reset_n,

    output logic [2:0] state //The state outputted by this state machine
        );

        //next state register
```

```
        logic [2:0] state_n;

        //each possible value of the state register is given a unique name for easier use later
        parameter S0 = 3'b000; //First digit
        parameter S1 = 3'b001; //Second digit
        parameter S2 = 3'b011; //Third digit
        parameter S3 = 3'b100; //Fourth digit

        //asynchronous reset will set the state to the start, S0, otherwise, the state is changed
        //on the positive edge of the clock signal
        always_ff @ (posedge clk_i, negedge reset_n)
                begin
                        if(!reset_n)
                                state = S0;
                        else
                                state = state_n;

                end

        //this section defines what the next state should be for each possible state. in this
        //implementation, it simply rotates through each state automatically
        always_ff @ (*)
                begin
                        case(state)
                                S0: state_n = S1;
                                S1: state_n = S2;
                                S2: state_n = S3;
                                S3:         state_n = S0;

                                default: state_n = S0;
                        endcase
                end
endmodule
```

```
module sevenseg(
    input logic [3:0] data,
    output logic [6:0] segments );
    always @(*)
        case( data ) // 7'bABCDEFG
        0: segments = 7'b1000000;
        1: segments = 7'b1111001;
        2: segments = 7'b0100100;
        3: segments = 7'b0110000;
        4: segments = 7'b0011001;
        5: segments = 7'b0010010;
        6: segments = 7'b0000010;
        7: segments = 7'b1111000;
        8: segments = 7'b0000000;
        9: segments = 7'b0011000;
        default:segments = 7'b1111111;
    endcase
endmodule
```

```
module mux2 ( input [3:0] a,
        input [3:0] b,
        input [3:0] c,
        input [3:0] d,
        input [2:0] s,
        output logic [3:0] y);
```

```
   always @ (a or b or c or d or s)
      case (s)
         3'b100 : y = a;
         3'b011 : y = b;
         3'b001 : y = c;
                  3'b000 : y = d;
                  default:y = a;
          endcase
endmodule
```

```
module parser (
        input logic [7:0] inp,
        output logic [3:0] a,
        output logic [3:0] b,
        output logic [3:0] c,
        output logic [3:0] d);

        //assign a = (inp - (inp%1000))/1000;
        //assign b = ((inp%1000)-(inp%100))/100;
        //assign c = ((inp%100)-(inp%10))/10;
        //assign d = (inp%10);
        assign a = (inp/1000)%10;
        assign b = (inp/100)%10;
        assign c = (inp/10)%10;
        assign d = (inp % 10);
endmodule
```