ECE272 Lab 5
Spring 2018

Voltmeter

Isak Foshay

April 16th, 2018

1. Introduction

This lab aims to introduce students to Serial Peripheral Interface (SPI) and Analog To Digital (ADC) converters. We will be using the ADC as input instead of the Button Board like we have in the past.

2. Design

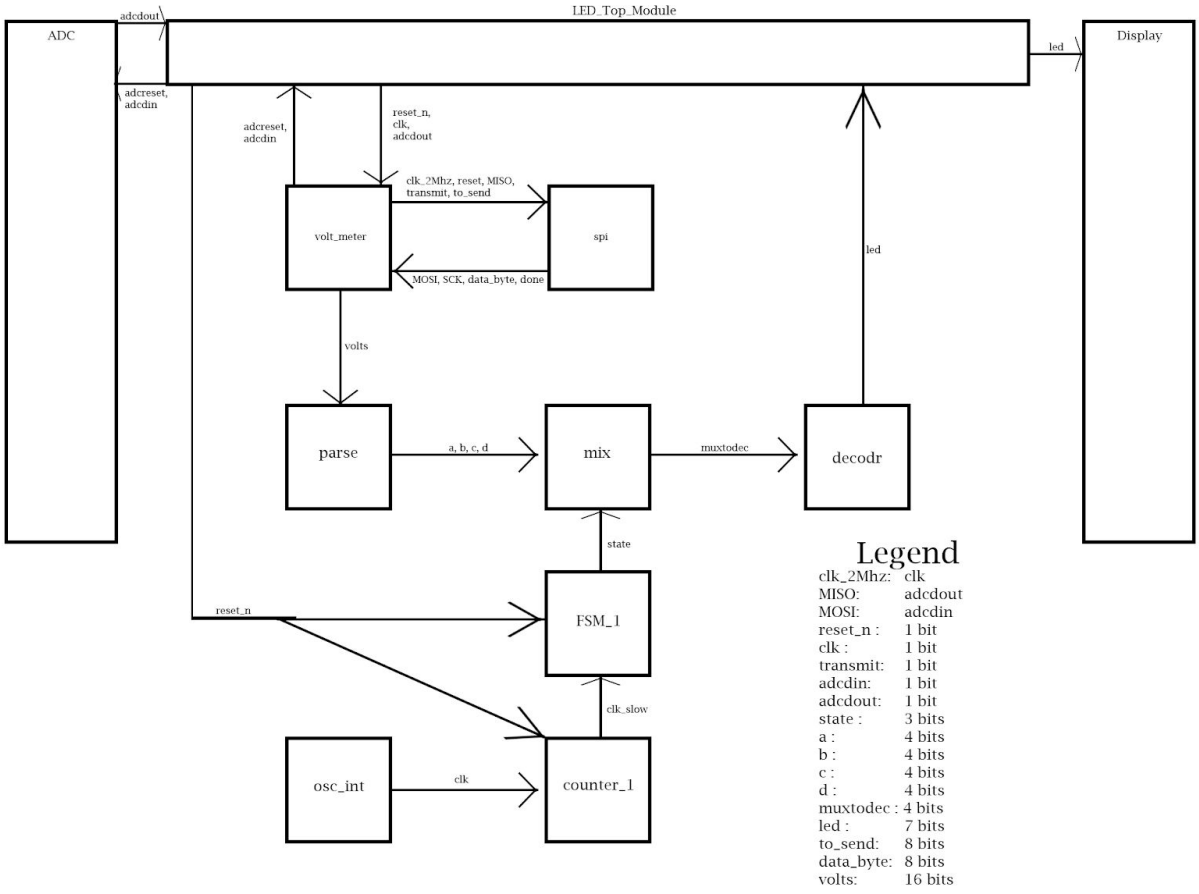

Figure 1: Design of program

This isn't the most intuitive depiction of all time but it is how my brain thinks about things so it was a good depiction for me.

	FPGA PIN	PULLMODE
adcdout	E3	DOWN
reset_n	C2	UP
adcdin	F3	UP
adcreset	R4	DOWN
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
scKlock	N6	DOWN
state[0]	E7	DOWN
state[1]	E8	DOWN
state[2]	F9	DOWN

Table 1: Chosen Pins and Pull Modes

I chose the following pins due to the similarity of the pins from the previous labs, mainly the display board. The other pins were arbitrary, yet close to one another.

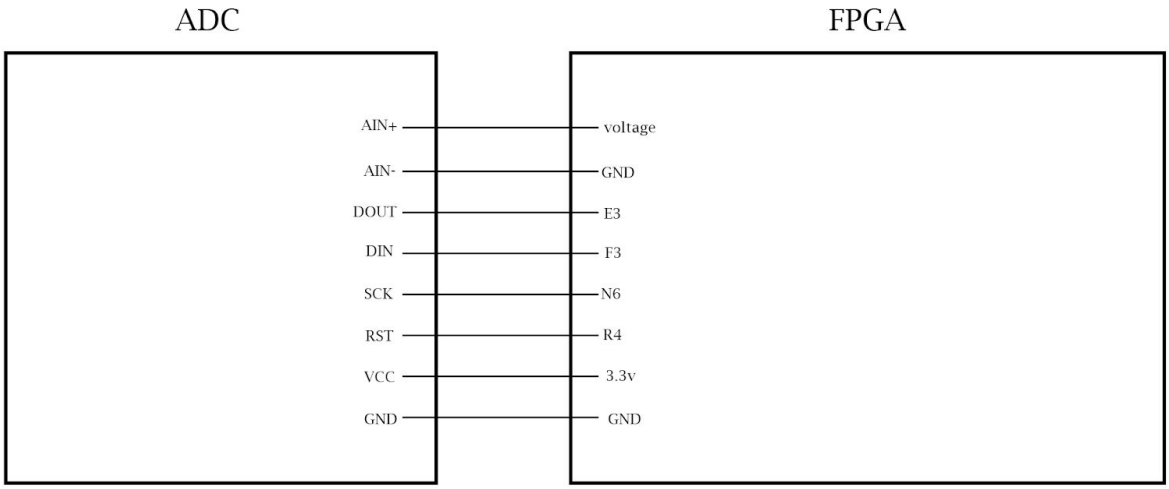

Figure 2a: Block Diagram for ADC

This diagram shows what needs to have voltage and what needs to be grounded. It also shows the pins I chose (which are grouped together). It also features the vague term “voltage” this is because it is up to the user to supply the AIN+ with the voltage they wish to be measured.

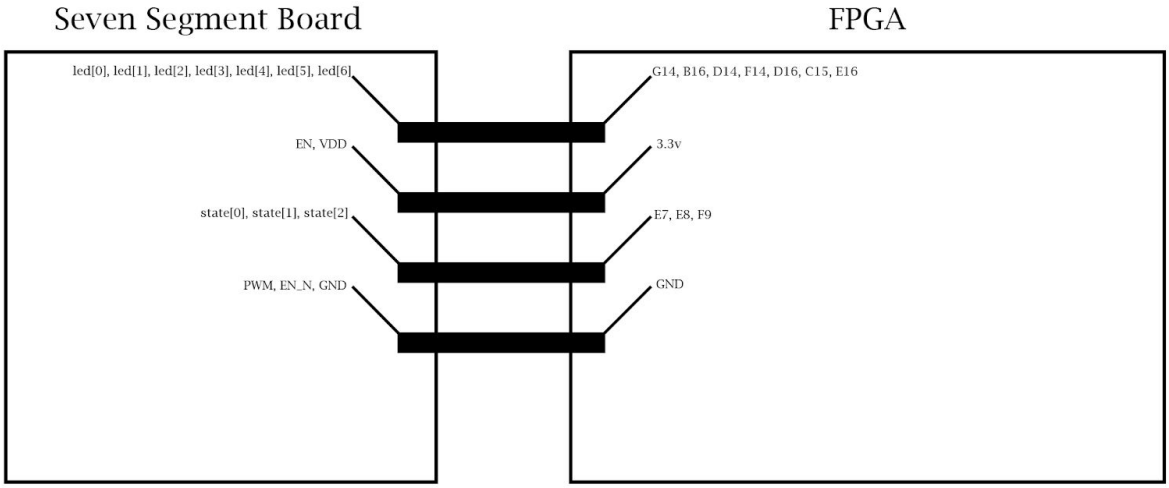

Figure 2b: Block Diagram for Display

This is the same diagram as lab 4 because I didn’t have to change anything with the display for this lab.

3. Results

The result was a working adc converter that properly displayed the output for each given voltage.

4. Experiment Notes

This lab took a while but not as long as lab 4. Learning to augment my code to fit this lab took a couple hours but in all didn't take as long as lab 4. For a long time I couldn't figure out how to give the adc different voltages and got very frustrated because I couldn't find the pins to test the voltage inputs. On the website it says to test on 3.3v, 1.2v, and ground, except, it doesn't state anywhere where someone would test 1.2v at. This is very frustrating.

Study Questions:

1. **In your own words, or in pseudocode, explain the functionality of the given adc and spi_master modules**

Adc gets clock info and miso info. adc will send raw input data over SPI. spi works by interpreting no signal on a clock edge as a zero and a high signal on a clock edge as a 1. SPI then transmits the data in two sets of 8, one set of 8 the lower 8 bits the next set of 8 the upper 8 bits. The adc is able to tell which bytes are which by how the state machine is set up in the adc module it sets the state to read_data2 in read_data1 so that read_data2 is guaranteed the next state.

Appendix

```
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
	output logic adcdin,
	output logic scKlock,
	output logic adcreset,
	output logic [2:0] state,
	input logic adcdout,
	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;
		logic [15:0] volts;
		/***********************/
		/* Define modules here */
		/***********************/

	    adc volt_meter( .reset    (reset_n),   // FPGA reset                                                    INPUT
	.clk_2Mhz (clk),   // directly from OSCH 2.08 MHz Oscillator INPUT
	.MISO     (adcdout),   // Pass through to ADC Din              INPUT   P13 Instructions wrong
	.raw_data (volts),   // 16 bit unsigned 0-5V                                                  OUTPUT
	.MOSI     (adcdin),   // Pass through to ADC Dout             OUTPUT  T6 Instructions wrong
	.adc_reset(adcreset),   // Pass through to ADC reset          OUTPUT
	.SCK      (scKlock) ); // Pass through to ADC SCK             OUTPUT  A9

		parser parse (
		.inp(volts),
		.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 [15:0] inp,
        output logic [3:0] a,
        output logic [3:0] b,
        output logic [3:0] c,
        output logic [3:0] d);
        logic [12:0] val;
        assign val = (5000*inp)/65535; //5000 (5 volts) is highest it will go (we go higher for more accuracy. divide by 65535 because that is the largest value that inp can be. so if there is
5 volts we will get 65535 then we divide and get 1 then multiply
        //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 = (val/1000)%10;
        assign b = (val/100)%10;
        assign c = (val/10)%10;
        assign d = (val % 10);
endmodule
```

```
// Author: Joshua Reed
// Class:  ECE 272, Spring 2016
// File: adc.sv
// Description:
// Unipolar volt meter implementation.
// spi_master is instantiated to communicate with the AD7705.
// A state machine is then used to setup and poll the ADC for
// voltage readings.
```

```
// The voltage that can be measured by default is 0-5V due to the layout of
// the AD7705 board, and the default gain settings.



/* instantiation template
 *
 *
 *
   adc volt_meter( .reset    ( your signals here ),   // FPGA reset
                   .clk_2Mhz ( your signals here ),   // directly from OSCH 2.08 MHz Oscillator
                   .MISO     ( your signals here ),   // Pass through to ADC Din
                   .raw_data ( your signals here ),   // 16 bit unsigned 0-5V
                   .MOSI     ( your signals here ),   // Pass through to ADC Dout
                   .adc_reset( your signals here ),   // Pass through to ADC reset
                   .SCK      ( your signals here ) ); // Pass through to ADC SCK
 *
 *
 *
*/

//=========================================
//================================ ADC ======
//=========================================
module adc( input  logic                reset, clk_2Mhz, MISO,
            output logic unsigned [15:0] raw_data,
            output logic                MOSI, adc_reset, SCK );

   logic done;
   logic transmit;
   logic unsigned [7:0] to_send;
   logic unsigned [15:0] raw_data;
   logic [7:0] data_byte;
      logic prev_done;
      logic unsigned [4:0] cnt;
      enum{CONFIGURE, RQST_COMM, READ_COMM, RQST_DATA, READ_DATA1, READ_DATA2} state;

   /* == States ============================
   *
   *      -- ( Setup ) --
   *         See p16 of the datasheet for Comm Reg bit definitions.
   *         See p17 of the datasheet for Setup Reg bit definitions.
   *         See p19 of the datasheet for Clock Reg bit definitions.
   *
   * The states should progress as:
   * 1)    a) Reset AD7705
   *              -- Set AD7705 reset to 0 then
   *              -- Set AD7705 reset to 1
   *       b) Write to comms register to select channel and
   *          set next opperation to be a write to the clock register.
   *              -- Send 00100000(32) over spi
   *       c) Write to clock register to set master clock reference,
   *          and desired update rate(50Hz).
   *          The board designed by TekBots uses a 4.192MHz oscillator.
   *          The slowest setting posible is 50Hz.
   *              -- Send 00001100(12) over spi
   *       d) Write to comms register to select channel and
   *          set next opperation to be a write to the setup register.
   *              -- Send 00010000(16) over spi
   *       d) Write to setup register to set gain, op conditions and
   *          initiate a self-calibration.
   *              -- Send 01000100(68)
   *
   *      -- ( Operational Cycle ) --
   *
   * 2)    a) Write to comms register to set up next operation as
   *          a read from the data register.
   *              -- Send 00000100(8) over spi
   *       b) If DRDY then go to request data register
   *          Else go to request comms register
   *       c) Write to comms register to set up next operation as
   *          a read from the data register.
   *              -- Send 00111000(56) over spi
   *       d) Read data byte 1
   *              -- Read spi data for 8 bits
   *       e) Read data byte 2
   *              -- Read spi data for 8 bits
   *              -- go to 2-a/start cycle again
   *
   * See p33 of the datasheet for state diagram.
   */
   always_ff @(posedge clk_2Mhz, negedge reset) begin : adc_sm
      if (~reset) begin
         state <= CONFIGURE;
         cnt <= 30;
```

```
        transmit <= 0;
        to_send <= 32;
        prev_done <= 0;
        adc_reset <= 0;
        end
    else begin
        adc_reset <= 1;
        case(state)
            CONFIGURE : begin
                // Reset ADC
                if (cnt==1500) adc_reset <= 1;
                if (cnt>4) cnt <= cnt-1;
                else if(done && (prev_done != done)) begin
                    if(cnt==4)to_send <= 12;
                    if(cnt==3)to_send <= 16;
                    if(cnt==2)to_send <= 68;
                    cnt <= cnt-1;
                    transmit <= 0;
                    end
                else transmit <= 1;

                // Transition State
                if(cnt==0) begin
                    state <= RQST_COMM;
                    transmit <= 0;
                    to_send  <= 8;
                    end
                end
            RQST_COMM : begin
                transmit <= 1;
                if (done && (prev_done != done)) begin
                    state    <= READ_COMM;
                    transmit <= 0;
                    end
                end
            READ_COMM : begin
                transmit <= 1;
                if (done && (prev_done != done)) begin
                    if (data_byte[7] == 1) begin
                        state    <= RQST_DATA;
                        to_send  <= 56; // read
                        end
                    else begin
                        state    <= RQST_COMM;
                        to_send  <= 8; // read
                        end
                    transmit <= 0;
                    end
                end
            RQST_DATA : begin
                transmit <= 1;
                if (done && prev_done!=done) begin
                    state    <= READ_DATA1;
                    transmit <= 0;
                    end
                end
            READ_DATA1 : begin
                transmit <= 1;
                if (done && prev_done!=done) begin
                    transmit  <= 0;
                    state     <= READ_DATA2;
                    raw_data[15:8] <= data_byte;                    //<-- second half of the output data
                    end
                end
            READ_DATA2 : begin
                transmit <= 1;
                if (done && prev_done!=done) begin
                    raw_data[7:0] <= data_byte;                                 //<-- first hald of the output data
                    transmit <= 0;
                    state    <= RQST_COMM;
                    to_send  <= 8;
                    end
                end
            default : state <= CONFIGURE;
            endcase
        prev_done <= done;
        end
    end

spi_master spi (
    // inputs
    .clk       ( clk_2Mhz ),
    .reset     ( reset    ),
    .MISO      ( MISO     ),
    .transmit  ( transmit ),
    .to_send   ( to_send  ),
```

```
    // outputs
    .MOSI     ( MOSI     ),
    .SCK      ( SCK      ),
    .received ( data_byte ),
    .done     ( done     ) );

endmodule


/*
 * AUTHOR: Joshua Reed
 * DATE: Nov. 25, 2015
 * DESCRIPTION: SPI master module that can be incorporated into a
 * larger project.
 * To use, load to_send with a byte of data while holding transmit low.
 * Then switch transmit high. The byte will be sent over
 * the next 16 clk cycles, and done will go high. To clear done
 * pull transmit to low.
 * MISO -> Master In Slave Out
 * MOSI -> Master Out Slave In
 * SCK -> Shift Clock
 *
 */


module spi_master(
  input clk, reset, MISO, transmit,
  input [7:0] to_send, // data to be sent
  output logic MOSI, SCK,
  output logic [7:0] received,
  output logic done );

  enum {setup, communicate, finished} state;
  logic unsigned [3:0] cnt;
  logic [7:0] hold_to_send;

  always_comb
    if (cnt % 2 == 0) SCK <= 0;
    else SCK <= 1;

  assign MOSI = hold_to_send[7]; // Send MSB

  always_comb
    if (state == finished) done <= 1;
    else done <= 0;

  always_ff @ (posedge clk, negedge reset)
    if (~reset) begin
      state <= setup;
      cnt <= 15;
      received <= 0;
      end
    else begin
      case(state)
        setup: begin
          hold_to_send <= {to_send[0], to_send[7:1]}; // Load to send reverse cycled by one
          if (transmit) state <= communicate;
          end
        communicate: begin
          cnt <= cnt-1;
          if (cnt % 2 == 1) begin // odd cycle
            if (cnt != 1) hold_to_send <= {hold_to_send[6:0], hold_to_send[7]}; // Rotate data on even number cycles
            received <= {received[6:0], MISO}; // Save current data bit in received
            end
          if (cnt == 0) begin
            state <= finished;
            received <= {received[6:0], MISO}; // Save current data bit in received
            end
          end
        finished: begin
          cnt <= 15;
          if (~transmit) state <= setup;
          end
        default: state <= setup;
        endcase
      end

  endmodule
```