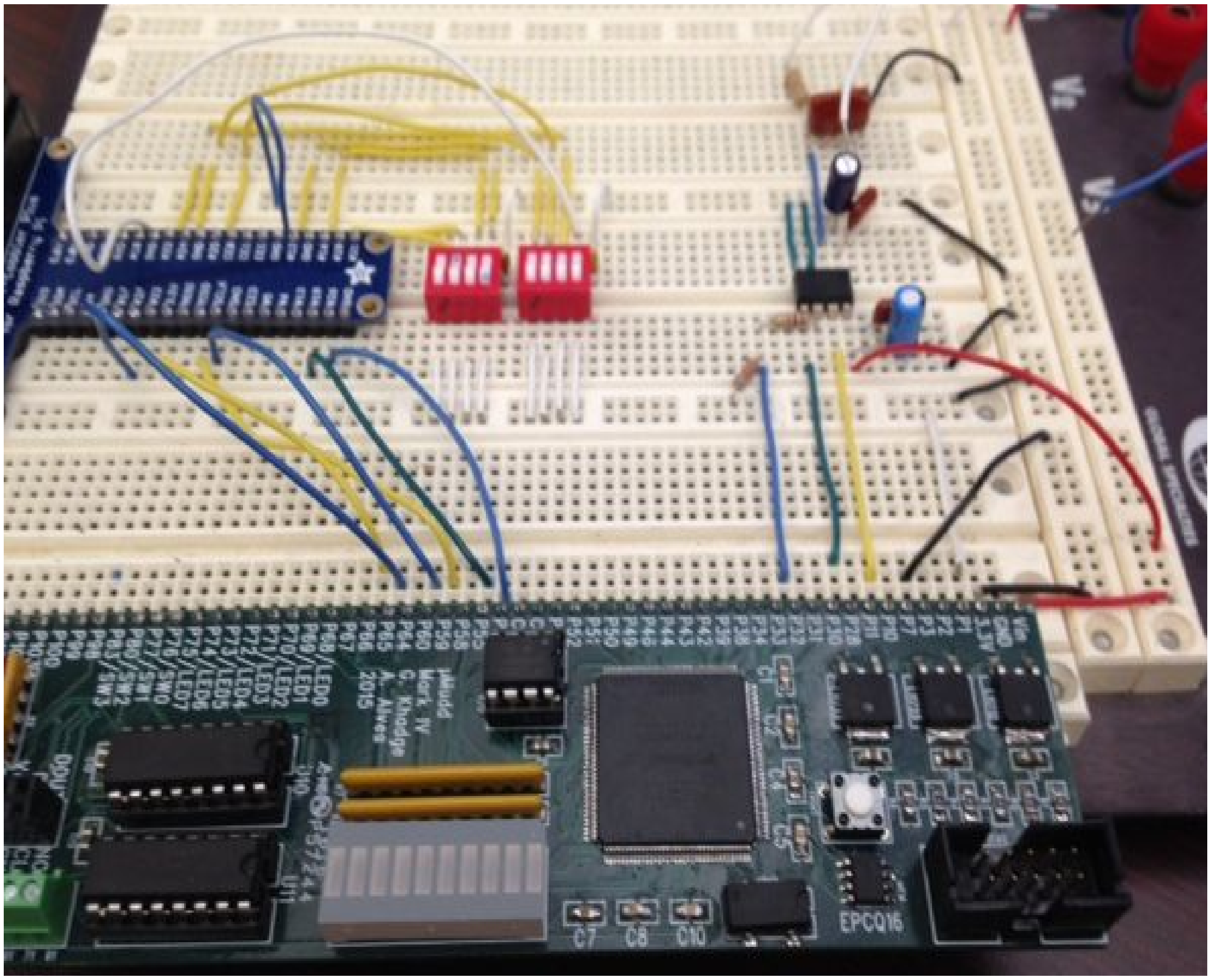
E155 Final Report: The Oscilloscope

Nick Draper & Jesus Villegas

11/29/2017



Abstract: The goal of this project is to build a rudimentary oscilloscope that has a throughput rate of 500 kHz and controls for setting the x and y range of the display. The ADS7818 converts the input signal into a digital signal and communicates this to the buffer via SPI. The buffer is designed using the FPGA in order to store the data from the ADS7818 and synchronize the output with the Raspberry Pi. The Pi receives the data via SPI and converts the values back into voltages. The Pi then graphs the data continuously with the appropriate x and y ranges.

Introduction

Modern oscilloscopes are generally costly and not very portable. Even cheap analog oscilloscopes can start at hundreds of dollars. This report documents the process for constructing a rudimentary oscilloscope using a fast sampling ADC coupled with a Raspberry Pi and an Altera Cyclone FPGA. The design will be both inexpensive relative to a professional lab oscilloscope and be portable such that a hobbyist or electrical engineering enthusiast can transport it with ease.

The basic functionality of an oscilloscope is rather straightforward. Therefore the goal of the project is to take in a continuous voltage reading, then pass this analog signal through a low-pass filter and then through an analog-to-digital converter. The ADC outputs the now digital signal at a high frequency into a buffer which is essentially the memory storage for the samples. Then have the buffer read out the stored samples at a lower frequency to a computer or any graphing capable device with controls to specify axis limits.

System Breakdown

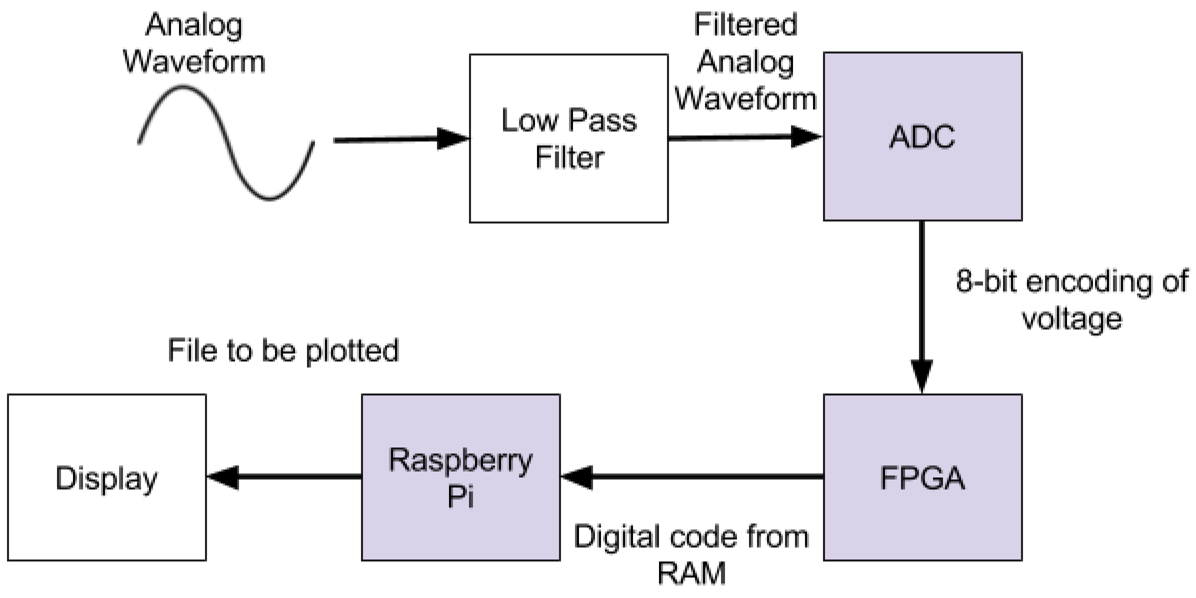


Figure 1: Block Diagram of Data Flow Through System

The system is comprised of three large scale subsystems highlighted in purple in Figure 1: the ADC, the memory buffer on the FPGA, and the Raspberry Pi. An analog waveform between zero and five volts is fed into a low pass filter, which consists of a simple RC circuit with a cutoff frequency of approximately 25 kHz. From there, the signal is then fed into the ADS7818 ADC which converts an analog voltage value in the specified range above into a 12 bit digital code.

From there, the eight most significant from the ADC are then stored into a temporary register on the Altera Cyclone IV FPGA EP4CE6 at a rate of 1.25 MHz. This transmission process between the FPGA and ADC works over SPI. When the temporary register is filled with the appropriate eight bits, a write enable flag goes high to indicate to the system to write the value to RAM. This process continually repeats until the whole RAM block, which has a size of about 16.3 Kbytes, is filled and the write pointer has reached the last address. Once the RAM buffer fills, the system then transitions to its read mode.

In read mode, the system no longer writes any values into RAM and instead sets the read pointer to the beginning of the memory block. From there the Raspberry Pi receives a signal that it should begin reading. Then while the read pointer has not reached the end the RAM, the Pi reads a one byte value each time over SPI at a rate of 100 kHz. After every read, the read address is incremented on the FPGA, and the Pi writes the value and a timestamp to a file descriptor to keep a record of the data from the current buffer. When the read address has finally reached the end of the memory, the FPGA signal the Pi through a GPIO to graph the current data. The Pi then uses a program called Gnuplot to plot the data contained in the file. A software timer delay displays the plot for a duration and then flushes the data in the file and sends a reset signal to the FPGA to reset all addresses and states and the whole process repeats. There are also two sets of quad dip switches that control the X-axis and Y-axis scaling of the plotted data.

New Hardware

This project uses the ADS7818 because it has the highest frequency speed of any DIP packaged ADC which is preferable due to its straightforward implementation.The ADS7818 is a successive approximation analog-to-digital converter which means that the conversion is done via a binary search through all quantization levels. In other words, the ADC approximates a discrete value for the voltage and then converts this approximate digital signal back to analog. Then a comparator takes in the original analog signal and the reconstructed analog signal in order to test the validity of the approximation. Due to this process of approximation, when the ADC is prompted over SPI to begin sending data it sends a 16-bit package where the first two bits and the last two bits do not encode any relevant data but rather all of the data is encoded into the middle 12 bits.

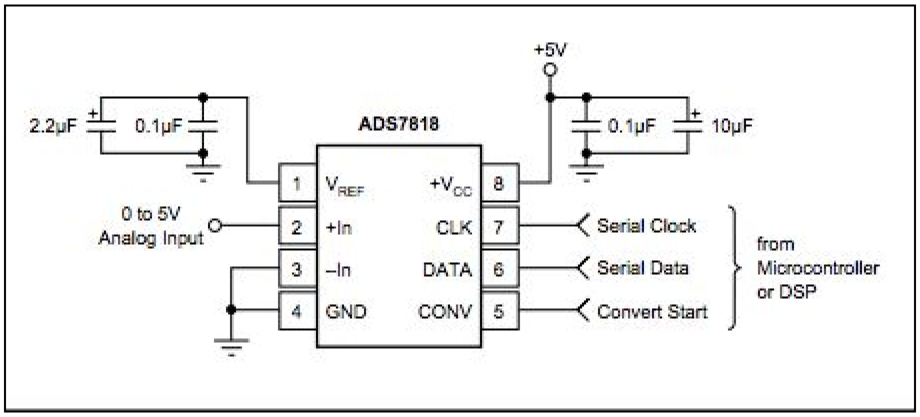
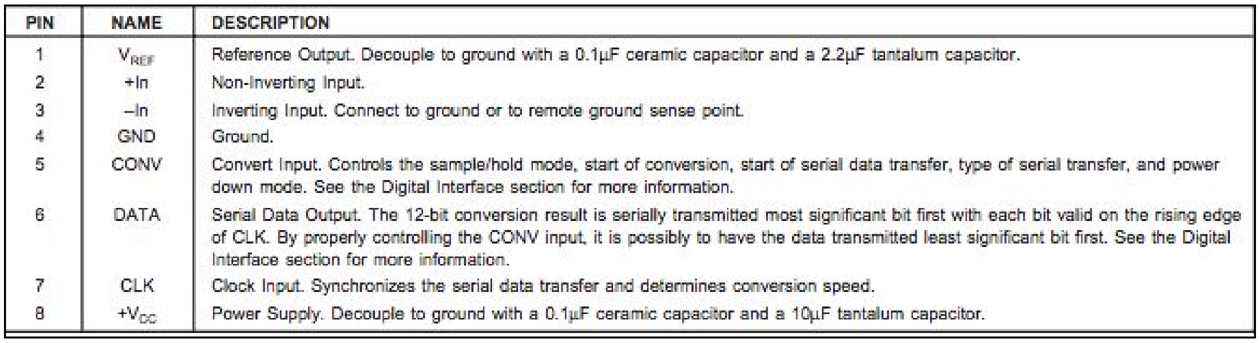


Figure 2: Operation Schematic [2]



PIN	NAME	DESCRIPTION
1	V_{REF}	Reference Output. Decouple to ground with a 0.1µF ceramic capacitor and a 2.2µF tantalum capacitor.
2	+In	Non-Inverting Input.
3	–In	Inverting Input. Connect to ground or to remote ground sense point.
4	GND	Ground.
5	CONV	Convert Input. Controls the sample/hold mode, start of conversion, start of serial data transfer, type of serial transfer, and power down mode. See the Digital Interface section for more information.
6	DATA	Serial Data Output. The 12-bit conversion result is serially transmitted most significant bit first with each bit valid on the rising edge of CLK. By properly controlling the CONV input, it is possibly to have the data transmitted least significant bit first. See the Digital Interface section for more information.
7	CLK	Clock Input. Synchronizes the serial data transfer and determines conversion speed.
8	$+V_{CC}$	Power Supply. Decouple to ground with a 0.1µF ceramic capacitor and a 10µF tantalum capacitor.

Figure 3: Pin Reference Table [2]

Figure 2 above, shows the schematic for the ADS7818 circuit used in this project. This schematic allows for only positive analog inputs since the inverted input is connected to ground. This device requires three signals communication, CLK, DATA, and CONV, to transfer data over the FPGA. The CLK signal is an input signal which sets the clock speed of the ADC. The minimum clock speed is of 200 kHz with a 12.5 kHz throughput rate while the maximum clock speed is 8 MHz with a 500 kHz throughput rate. In this project the FPGA outputs a 1.25 MHz clock signal to the ADC. The DATA signal outputs voltage in 16-bits with a 12-bit resolution. Lastly, CONV acts as a trigger in order to the signal the ADC to stop sampling and begin the process of converting the signal and sending it over DATA.

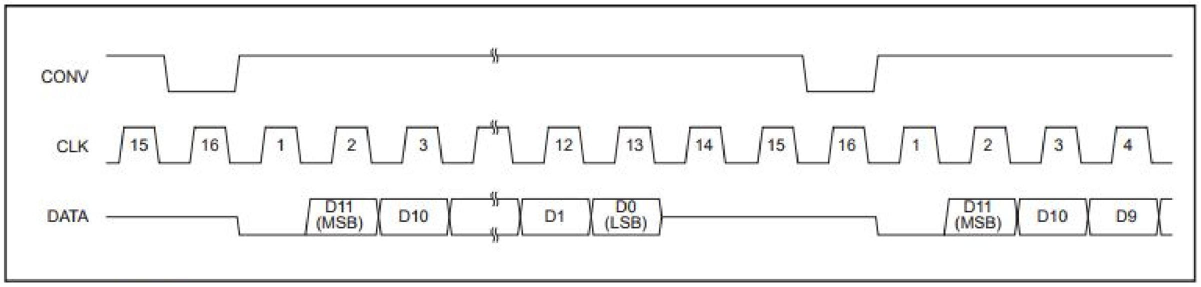


Figure 4: Timing Diagram for Communication Protocol with FPGA [2]

Two important notes about this component. This project instead implements the DSP timing diagram from the ADS7818, which is shown above, and this timing sequence works with the SPI module on the FPGA and communicates the data over correctly. The next is that the data sheet indicates to read on the posedge of the CLK however this created an issue where the LSB of a previous reading carried over as the MSB of the next reading causing a 2.5 V shift in some values.

In order to convert the values output by the ADC back into voltages, divide the values by 4096, which is the maximum for a 12-bit number, and then multiply it by the reference voltage. While this covers a general explanation of implementation, the data sheet which is widely available on the internet should be referenced for any further guidance.

Microcontroller (Raspberry Pi) Design

The Raspberry Pi performs four main functions: 1) Converts the ADC values into voltages while also using a counter to approximate the corresponding time value, 2) Writes the voltage and corresponding time value into a text file, 3) Reads and decodes the the dip switches in order to accordingly set the x and y range or terminate the program, and 4) Graph the values in the text file using gnuplot.

1) In order to read in the ADC values from the FPGA, functions were written using SPI. The Pi initiates communication using the SPIinit() fucntion from EasyPio.h [1]. The CS bit is set low and the SCLK is set to 100 kHz. However since the project requires that the FPGA be the master, GPIO pins FULL and DONE_READING are set as inputs and PI_GRAPH_DONE is set as an output. These auxiliary signals allowed the FPGA to function as a pseudo-slave. The FULL signal goes HIGH in order to trigger the spiSendReceive() function [1]. The ADC value is converted, the corresponding time value is approximated.

2) Once the value has been converted and the time value approximated, it is written to a text file. This is repeated until the DONE_READING signal goes HIGH indicating all the values have been read out of memory.

3) Once DONE_READING goes HIGH then the Pi reads the switch values and accordingly sets the x and y range using the fprintf function to write to the gnuplot command line. In this case the commands are set xrange and set yrange.[3] However, if SWITCH_KILL is HIGH then the program will not enter the DONE_READING condition, and the program will close gnuplot with the fprintf function and gnuplot command clear and exit.

4) Once the x-range and y-range have been set the plot is generated using the fprintf function and the gnuplot command plot; then the gnuplot command pause is used in order to pause gnuplot while the program continue running in order to read new data from the FPGA. [3] This allows for the graph to continuously update but also causes a slight delay between switch input and display change. After gnuplot is paused, then PI_GRAPH_DONE is set HIGH in order to

appropriately reset registers within both the memory and pi communication modules so that data is sent over whenever it is available once again.

FPGA (Altera Cyclone IV EP4CE6) Design

The FPGA has three modules each of which performs a crucial function to the system. The first module called adc_com in shown in Appendix C is responsible for the communication and data retrieval from the ADC. This module is comprised of a state machine with 16 states. In the zeroth state, the default, the adc_conv pin raised high. When the system transitions to state one, the adc_conv pin is set low. The system then enters state two which corresponds with cycle one in Figure 4 for the timing diagram. This state has no relevant data coming over the MISO line and thus can be neglected. When the module enters state three, on the negative edge of the clock the most significant bit is sent on the MISO line. This happens for the next 11 cycles since the ADC sends 12 bits of data. After all the bits have been sent out, the adc_conv flag is raised back high and then the write_enable flag is raised high. This indicates to the memory module to write the eight most significant bits of this voltage reading to RAM. Note that that write_enable pin is high for one cycle as to not write the same values multiple times to RAM.

The second module handles all the memory interfacing operations and is appropriately declared memory. This module configures a large block of memory, in this case about 16 Kbytes, as single-port memory shown here in Figure 5.

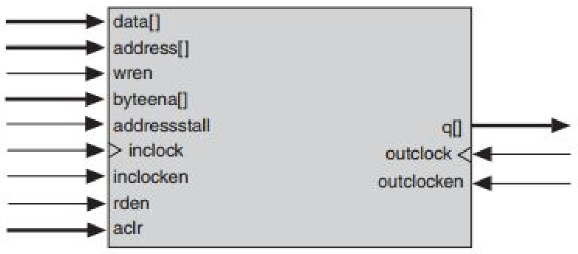


Figure 5: Single-Port Memory Block Diagram

The memory takes in two clocks, one which is synchronized with the ADC clock, the inclock, and the other being a serial clock coming from the Raspberry Pi, the outclock. It also takes in a write_enable flag, one byte of write_data. Its outputs are one byte of read_data, a 14 bit read address, a full flag, and a done_reading flag. The memory logic is quite simple. It functions as a three state state machine. In its zeroth state, it just holds and does nothing. In state one, it is in the write state. Here the modules accepts write_data from the adc_com module and will write the data to RAM when the write_enable flag is high. After every write, the write pointer is incremented. Once the write pointer reaches the end of the RAM block, the system transitions to state two, read mode. In read mode, the module will not write values anymore and assigns read_data to whatever the value in memory is at the read address it is given. After every read, which is comprised of one byte, the read pointer is also incremented. Once the read pointer has reached the end of the RAM block, the system enters the zeroth state in which it does nothing until instructed by the Pi.

The final module is the pi_spi one. This module handles the transmission of data from RAM to the Pi using the SPI protocol. In this module, the read address is generated and sent to the memory module. It then is given a byte of value, read_data, from which it writes one bit of read_data every serial clock cycle to the MISO line. After all eight bits have been sent, the read address is incremented. Note that the Pi will send over only eight serial clock cycles at a time and the read mode is handled by the Pi. When the FPGA enters the zeroth state, the Pi then graphs its current data set. Once the Pi finishes graphing its data with a delay, it raises a reset pin high which resets all addresses on the FPGA and causes it to enter write mode.

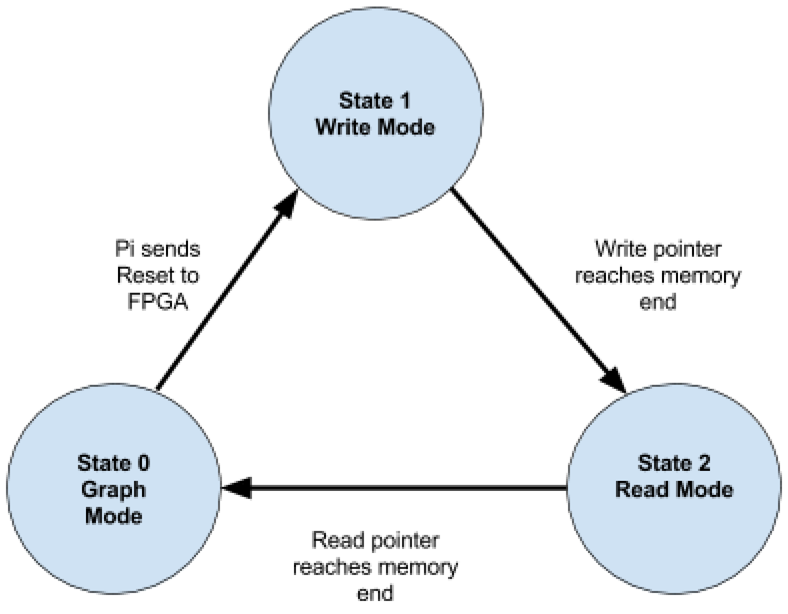


Figure 6: Overall State Machine of System

Results and Future Work

The final project works well and meets all the design parameters laid out in the project proposal. It can successfully graph all DC voltages between zero and five volts with very reasonable accuracy. It can also graph sinusoidal waves up to 23 kHz with good accuracy and can handle five volt peak to peak waves all the way down to 200 mV peak to peak waves with fair accuracy. The system can also handle triangle wave inputs as well as square waves. Though the frequency of the square waves was more limited due to the large harmonics of its sharp edges. Since the low pass filter has a corner frequency of only about 25 kHz, the sharpness of the square waves did not show as well as expected. Though it does match what an actual oscilloscope measures for the same signal through a low pass filter.

For future work, the system can be improved by adding a programmable gain amplifier (PGA). For signals smaller than 200 mV peak to peak, the system could amplify such voltages with a PGA and have a wider variety. Other improvements include constructing a triggering module to plot as opposed to just a software delay. As well as a fast fourier transform display to improve functionality.

References

[1] E155: Microprocessor-Based Systems. Harvey Mudd College, 2016. Web: http://pages.hmc.edu/harris/class/e155/

[2] “12-Bit High Speed Low Power Sampling ANALOG-TO-DIGITAL CONVERTER”. Burr Brown. May 2000. https://datasheet.lcsc.com/szlcsc/ADS7818E-250_C122808.pdf

[3] “gnuplot 5.0 An Interactive Plotting Program” Williams & Kelley. August 2017 http://www.gnuplot.info/docs_5.0/gnuplot.pdf

Parts List

Part	Source	Vendor Part #	Quantity	Price (per unit)
ADS7818 ADC	Digikey	ADS7818P-ND	5	$6.41

Appendix A: Raspberry Pi 3B C Code

```
// plot.c
// jvillegas@g.hmc.edu, ndraper@g.hmc.edu 11/10/2017
//
// Receives data from an FPGA which acts as a buffer
// for a hobbyist oscilloscope and plots the data

#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include "EasyPIO.h"

#define DATA_REQ 26
#define DONE_READING 27
#define PI_GRAPH_DONE 22
#define FULL 19
#define HIGH 1
#define LOW 0

#define SW_1 12 //MSB of closest switch
#define SW_2 16
#define SW_3 20
#define SW_4 21 //LSB

#define SW_KILL 18 //closes gnuplot MSB of farthest switch
#define SW_7 23
#define SW_6 24
#define SW_5 25 //LSB

int main(void)
{
	char spi_data = 0;
	int spi_data_shift = 0;
	double times = 0;
	double volt = 0;
	int counter = 0;

	// Speed of the SCLK from the Pi when reading SPI
	int sclk = 100000;
	// Speed of the clock sent to the ADC, set by the FPGA
```

```
double adc_clock = 1250000.0;
// time spacing between each voltage reading
double time_div = 16.0/adc_clock;

// file descriptor for both
char *data_files = "data0.dat";
FILE *data = fopen(data_files, "w");
FILE *gnuplot = popen ("gnuplot", "w");

pioInit();
// setup SPI on the pi with speed of sclk
spiInit(sclk, 0);
// GPIO initialization and setup
pinMode(DONE_READING, INPUT);
pinMode(PI_GRAPH_DONE, OUTPUT);
pinMode(FULL, INPUT);
pinMode(RESET, OUTPUT);
pinMode(SW_1, INPUT);
pinMode(SW_2, INPUT);
pinMode(SW_3, INPUT);
pinMode(SW_4, INPUT);
pinMode(SW_5, INPUT);
pinMode(SW_6, INPUT);
pinMode(SW_7, INPUT);
pinMode(SW_KILL, INPUT);

digitalWrite(RESET, HIGH);
printf("RESET\n");
delayMillis(3000);
digitalWrite(RESET, LOW);

while(1) {
        // If the kill switch is on, close the gnuplot windows
        if(digitalRead(SW_KILL)) {
                fprintf(gnuplot, "clear\n");
                fprintf(gnuplot, "exit\n");
        }
        // If the DONE_READING pin is high, plot the data we have read
        // from RAM
        else if(digitalRead(DONE_READING)) {
                fclose(data);

                // Conditionals for how to set the X-axis scales on the plot
```

```
if(digitalRead(SW_1)) {
        fprintf(gnuplot, "set xrange[0:.0001]\n");
}
else if(digitalRead(SW_2)) {
        fprintf(gnuplot, "set xrange[0:.001]\n");
}
else if(digitalRead(SW_3)) {
        fprintf(gnuplot, "set xrange[0:.01]\n");
}
else if(digitalRead(SW_4)) {
        fprintf(gnuplot, "set xrange[0:.1]\n");
}
else {
        fprintf(gnuplot, "set xrange[0:.001]\n");
}

// Conditionals for how to set the Y-axis scales on the plot
if(digitalRead(SW_5)) {
        fprintf(gnuplot, "set yrange[0:.5]\n");
}
else if(digitalRead(SW_6)) {
        fprintf(gnuplot, "set yrange[0:2.5]\n");
}
else if(digitalRead(SW_7)) {
        fprintf(gnuplot, "set yrange[0:5]\n");
}
else {
        fprintf(gnuplot, "Set yrange[0:5]\n");
}

// update the plot and keep it up for 2 seconds with delay
fprintf(gnuplot, "plot '%s' notitle smooth csplines\n", data_files);
fflush(gnuplot);
fprintf(gnuplot, "pause 1\n");
delayMillis(2000);
printf("if: %d\n", counter);

// reset the counter and then reopen the data file to refill it
counter = 0;
data = fopen(data_files, "w");
// Reset the addresses on the FPGA
digitalWrite(PI_GRAPH_DONE, HIGH);
printf("RESET\n");
```

```
                spiSendReceive(0);
                digitalWrite(PI_GRAPH_DONE, LOW);
            }
            // if the RAM buffer is full execute conditional block
            else if(digitalRead(FULL)) {
                // grab the data from a single address in RAM
                spi_data = spiSendReceive(0);
                printf("else if: %d, byte: %d\n", counter, spi_data);
                spi_data_shift = (int)spi_data << 4;
                volt = (double)spi_data_shift/4096.0 * 5.0;
                // correctly update the time step
                times = time_div * (double)counter;
                // write the time and voltage to the data file
                fprintf(data, "%lf %lf\n", times, volt);
                counter++;
            }
            else {
                printf("else: %d\n", counter);
                continue;
            }
        } // end while
        return 0;
}
```

Appendix B: Verilog Code

```
// oscope.sv
// jvillegas@g.hmc.edu, ndraper@g.hmc.edu 11/10/2017

/*
 * Create an oscilloscope that reads in an analog waveform which
 * is then passed to an ADC. The FPGA will then sample the data
 * from the ADC and write it to a buffer in its on board RAM.
 * Once the buffer fills, the FPGA will stop reading writing
 * values to its memory and then signal the Pi to read values
 * from the RAM buffer. Once the Pi reads all the data, it will
 * signal the FPGA to read from the ADC and refill the buffer.
 */
module oscope(input logic osc_clk,
              input logic reset,
              input logic adc_data,
              input logic pi_graph_done,
              input logic sclk,
              output logic adc_clk,
              output logic adc_conv,
              output logic full,
              output logic done_reading,
              output logic miso,
              output logic [7:0] read_led);

      logic [7:0] write_data, read_data;
      logic [13:0] read_adr;
      logic write_enable;

      // Instantiation of the adc_com module
      adc_com adc1(osc_clk, reset, adc_data, adc_clk, adc_conv, write_enable, write_data);
      // Instantiation of the memory module
      memory m1(adc_clk, reset, write_enable, write_data, read_adr, pi_graph_done,
                read_data, done_reading, full, read_led);
      // Instantiation of the pi_spi module
      pi_spi s1(sclk, reset, miso, pi_graph_done, read_adr, read_data);

endmodule

// Module used to communicate with the ADC, collects what data to write to RAM
// as well as a write enable flag for when to write
```

```
module adc_com(input logic osc_clk,
                        input logic reset,
                        input logic adc_data,
                        output logic adc_clk,
                        output logic adc_conv,
                        output logic write_enable,
                        output logic [7:0] write_data);

        // State declarations
        typedef enum logic [4:0] {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
                                        S12, S13, S14, S15, S16, S17, S18, S19} statetype;
        statetype state, nextstate;

        // temp register to hold adc data
        logic [11:0] temp_data;
        // counter for adc_clk
        logic [31:0] counter;

        // oscillator clock to set the adc_clk
        always_ff @(posedge osc_clk, posedge reset)
                if (reset) counter <= 0;
                else counter <= counter + 1;

        always_ff @(negedge adc_clk, posedge reset)
        begin
                if (reset) state <= S0;
                // state transition and loading of temp_data
                else
                begin
                        state <= nextstate;
                        case (state)
                                // do nothing
                                S0:     temp_data <= 0;
                                // conv goes low, do nothing
                                S1:             ;
                                // junk cycle no data, do nothing
                                S2:             ;
                                // MSB D11
                                S3:     temp_data[11] <= adc_data;
                                S4:     temp_data[10] <= adc_data;
                                S5:     temp_data[9] <= adc_data;
                                S6:     temp_data[8] <= adc_data;
                                S7:     temp_data[7] <= adc_data;
```

```
					S8:	temp_data[6] <= adc_data;
					S9:	temp_data[5] <= adc_data;
					S10:	temp_data[4] <= adc_data;
					S11:	temp_data[3] <= adc_data;
					S12:	temp_data[2] <= adc_data;
					S13:	temp_data[1] <= adc_data;
					// LSB D0
					S14:	temp_data[0] <= adc_data;
					// Raises the conv pin and write enable no data to collect
					S15:	;
					default:		; // do nothing
				endcase
			end
		end

		// state transition logic, adc_conv logic, write_enable logic,
		always_comb
		begin
			case (state)
				// default conv is high and no writing
				S0:		begin
							adc_conv = 1;
							write_enable = 0;
							nextstate = S1;
						end
				// first time conv goes low
				S1:		begin
							adc_conv = 0;
							write_enable = 0;
							nextstate = S2;
						end
				// junk cycle no data comes in yet
				S2:		begin
							nextstate = S3;
							adc_conv = 0;
							write_enable = 0;
						end
				// D11 (MSB) comes in at this point
				S3:		begin
							nextstate = S4;
							adc_conv = 0;
							write_enable = 0;
						end
```

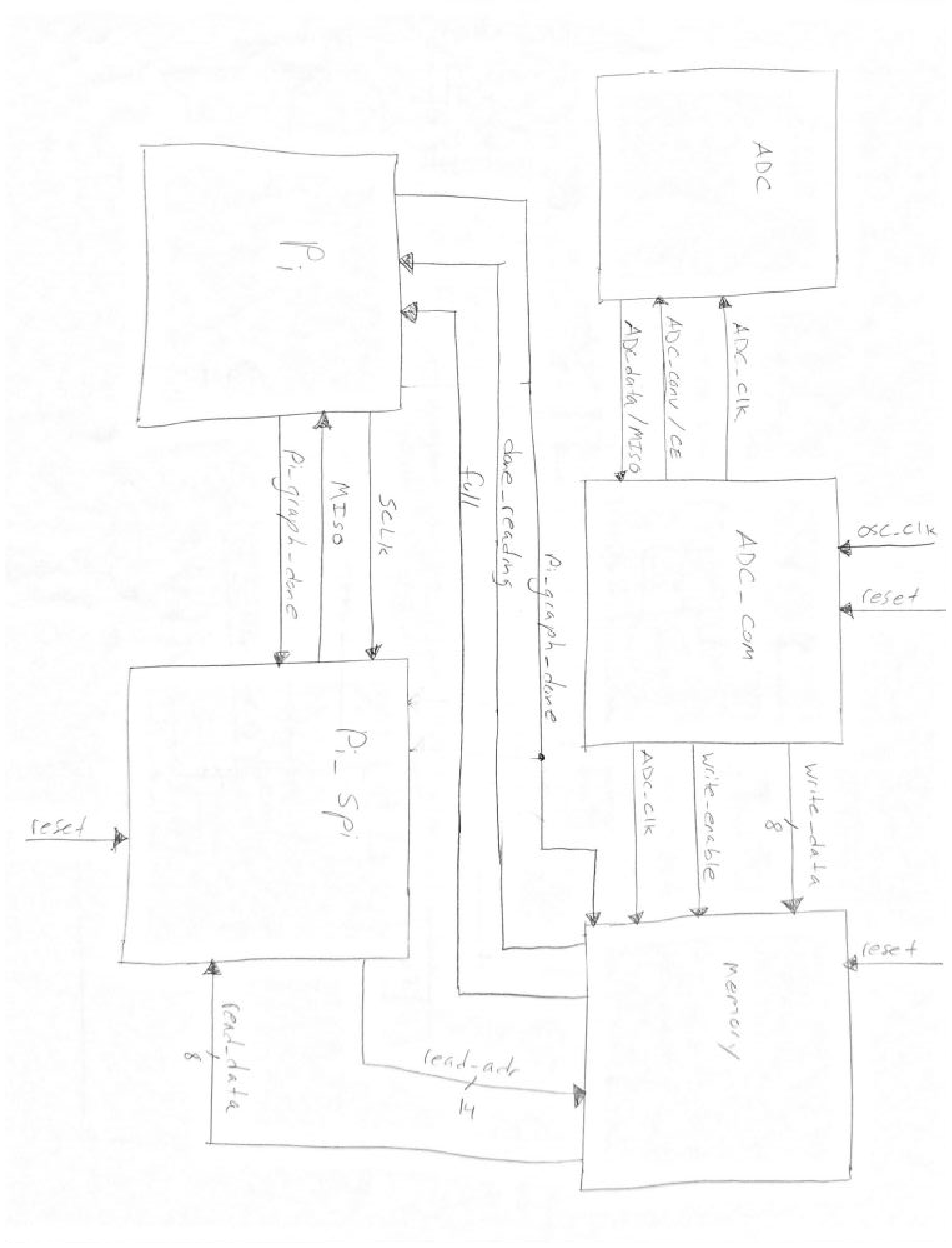
```
// D10
S4:         begin
                  nextstate = S5;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D9
S5:         begin
                  nextstate = S6;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D8
S6:         begin
                  nextstate = S7;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D7
S7:         begin
                  nextstate = S8;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D6
S8:         begin
                  nextstate = S9;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D5
S9:         begin
                  nextstate = S10;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D4
S10:        begin
                  nextstate = S11;
                  adc_conv = 0;
                  write_enable = 0;
            end
// D3
```

```
                    S11:            begin
                                            nextstate = S12;
                                            adc_conv = 0;
                                            write_enable = 0;
                                    end
                    // D2
                    S12:            begin
                                            nextstate = S13;
                                                    adc_conv = 0;
                                            write_enable = 0;
                                    end
                    // D1
                    S13:            begin
                                            nextstate = S14;
                                            adc_conv = 0;
                                            write_enable = 0;
                                    end
                    // D0 (LSB) comes in
                    S14:            begin
                                            nextstate = S15;
                                            adc_conv = 0;
                                            write_enable = 0;
                                    end
                    // Conv goes high and so do does write enable
                    // then we repeat states
                    S15:            begin
                                            nextstate = S0;
                                            adc_conv = 1;
                                            write_enable = 1;
                                    end

                    default:        begin
                                            nextstate = S0;
                                            adc_conv = 1;
                                            write_enable = 0;
                                    end
            endcase // state
        end

        assign adc_clk = counter[4];
        assign write_data = temp_data[11:4];
endmodule
```

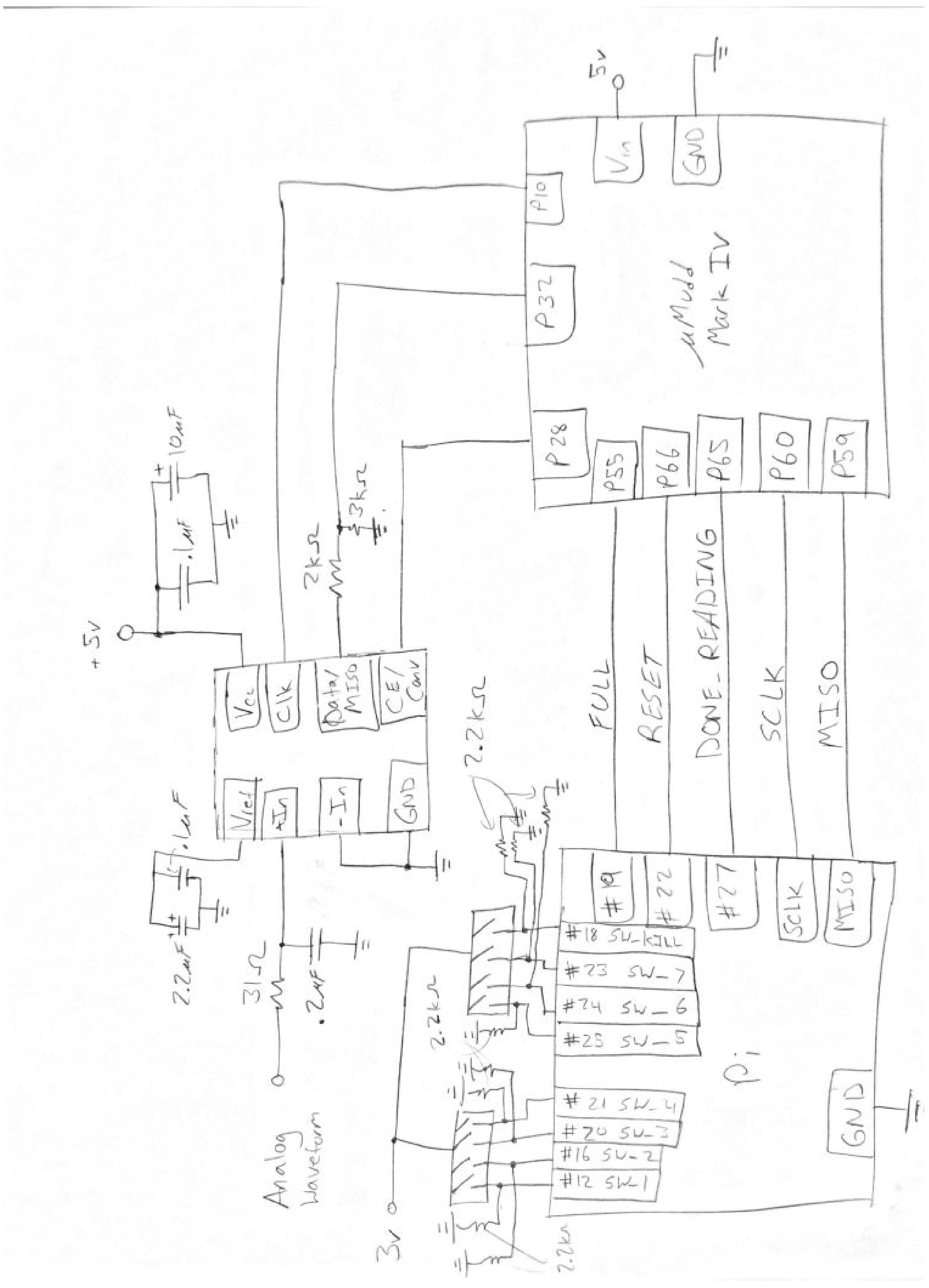
```
// This module handles the writing to memory and reading from memory.
// We write only when the write enable flag is high and
module memory(input logic adc_clk,
                      input logic reset,
                      input logic write_enable,
                      input logic [7:0] write_data,
                      input logic [13:0] read_adr,
                      input logic pi_graph_done,
                      output logic [7:0] read_data,
                      output logic done_reading,
                      output logic full,
                      output logic [7:0] read_led);

        logic [13:0] write_adr;
        logic [7:0] mem[16383:0];

        // full should only be high if the write address has reached
        // the end of the buffer and only be reset when the write_adr changes
        always_ff @(posedge adc_clk, posedge reset)
                if (reset) full <= 0;
                else if (pi_graph_done) full <= 0;
                else if (write_adr == 16382) full <= 1;
                else full <= full;

        // logic for writing the data to memory and appropriately changing the
        // writing pointer
        always_ff @(posedge adc_clk, posedge reset)
                if (reset) write_adr <= 0;
                else if (full) write_adr <= write_adr;
                else if (pi_graph_done) write_adr <= 0;
                else
                begin
                        if (write_enable & !full)
                        begin
                                mem[write_adr] <= write_data;
                                write_adr <= write_adr + 1;
                        end
                        else ; // do nothing
                end

        assign read_data = mem[read_adr];
        assign done_reading = (read_adr == 16382);
        assign read_led = write_data;
```

```
endmodule

// Module to communicate with the Pi and send read_data over MISO to the FPGA,
// also handle the read address generator to grab data from memory
module pi_spi( input logic sclk,
               input logic reset,
               output logic miso,
               input logic pi_graph_done,
               output logic [13:0] read_adr,
               input logic [7:0] read_data);

    logic reset_adr;

    typedef enum logic[3:0] {S1, S2, S3, S4, S5, S6, S7, S8} statetype;
    statetype state, nextstate;

    // next state transition
    always_ff @(posedge sclk, posedge reset)
        if (reset) state <= S8;
        else if (pi_graph_done) state <= S8;
        else state <= nextstate;

    // get read data to miso
    always_ff @(negedge sclk, posedge reset)
        if (reset) miso <= 0;
        else
            case (state)
                S1: miso <= read_data[6];
                S2: miso <= read_data[5];
                S3: miso <= read_data[4];
                S4: miso <= read_data[3];
                S5: miso <= read_data[2];
                S6: miso <= read_data[1];
                S7: miso <= read_data[0];
                S8: miso <= read_data[7];
                default: miso <= 0;
            Endcase

    // logic to control the read address
    always_ff @(posedge sclk, posedge reset)
        if (reset) read_adr <= 0;
        else if (pi_graph_done) read_adr <= 0;
        else if (state == S7) read_adr <= read_adr + 1;
        else read_adr <= read_adr;

    // nextstate logic
    always_comb
        case (state)
            S8: nextstate = S1;
            S1: nextstate = S2;
            S2: nextstate = S3;
            S3: nextstate = S4;
            S4: nextstate = S5;
            S5: nextstate = S6;
            S6: nextstate = S7;
            S7: nextstate = S8;
            default: nextstate = S8;
        endcase
endmodule
```

Appendix C: System Block Diagram



Appendix D: Circuit Schematic



Appendix E: FPGA Pin Assignments

Digital Logic	**FPGA Pin Number**
Oscillator Clock	Pin 88
ADC Clock	Pin 10
ADC Data / ADC MISO	Pin 32
ADC CE / Conversion	Pin 28
Full Flag	Pin 55
Reset / Pi Graph Done Flag	Pin 66
Done Reading Flag	Pin 65
Pi Serial Clock	Pin 60
Pi MISO	Pin 59

Appendix F: Raspberry Pi 3B Pin Assignments

Digital Logic	**Raspberry Pi Number**
Full Flag	Pin 19
Done Reading Flag	Pin 27
Reset / Pi Graph Done Flag	Pin 22
Pi Serial Clock	Pin SCLK
Pi MISO	Pin MISO
X Axis 0 - .0001	Pin 12
X Axis 0 - .001	Pin 16
X Axis 0 - .01	Pin 20

X Axis 0 - .1	Pin 21
Y Axis 0 - .5	Pin 25
Y Axis 0 - 2.5	Pin 24
Y Axis 0 - 5	Pin 23
Kill Program Switch	Pin 18