# MicrOscope

Alex Echeverria & Isabel Martos-Repath

E155 Final Project, Fall 2017

## Abstract

Oscilloscopes are a popular and useful part of electronics benchtop equipment- but generally far too expensive for the average college student to own. With this project, the team sought to design and build a simple, low-cost oscilloscope made primarily out of materials used in the E155: Microprocessor Systems course, that future students could to use to augment their dorm room electronics bench setups (and save them the long, chilly walk to the Parsons Building basement). The project, MicrOscope, takes in an analog signal ranging from 0 to 5 V and up to a frequency of 19 kHz, and processes it to be displayed on a webpage. The input signal is filtered through a model of a 10x scope probe and a Sallen Key filter, then an FPGA collects the data from an ADC. The FPGA checks the data against the user's chosen trigger level, filters it (if the user selects to use the high or low pass filters), and then communicates with a Raspberry Pi. The Raspberry Pi plots the signal on a web page according to user's selected inputs regarding time per division, voltage per division, and noise level.

## Introduction

The team's project is an oscilloscope that takes an analog input, passes it through analog filters meant to approximate a 10x scope probe and a Sallen-Key filter to prevent aliasing, filters it again using high and low pass filters implemented on an FPGA, and finally displays the signal on a web page using a Raspberry Pi. There are also a myriad of inputs the user can select using switches to control how they want their signal to display: trigger level (a threshold for which they want the signal to be above if it is to appear onscreen), volts/division, time/division, noise level. Below, in Figure 1, is a block diagram of how the oscilloscope is organized as well as a flowchart showing how a signal will progress through the system in Figure 2.

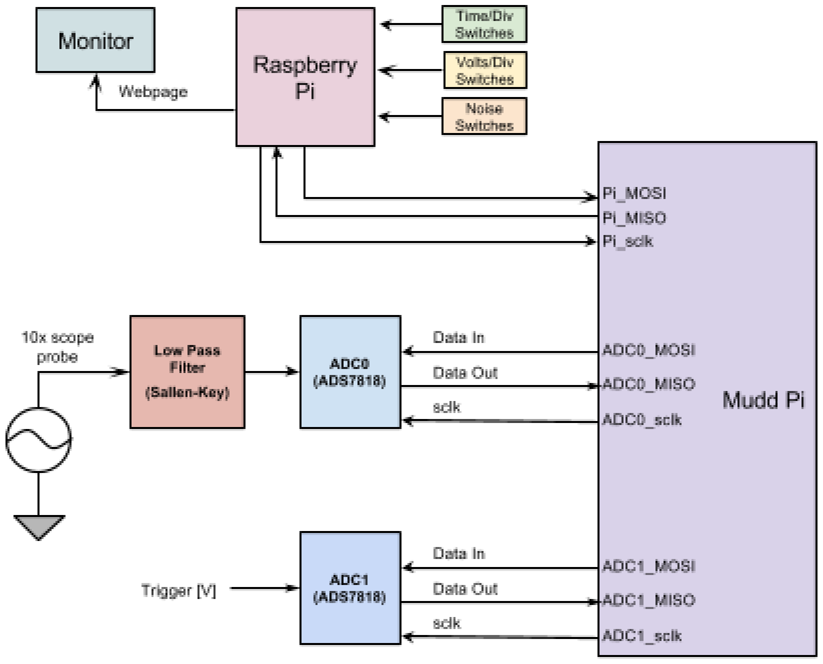


Figure 1: Block diagram of whole system

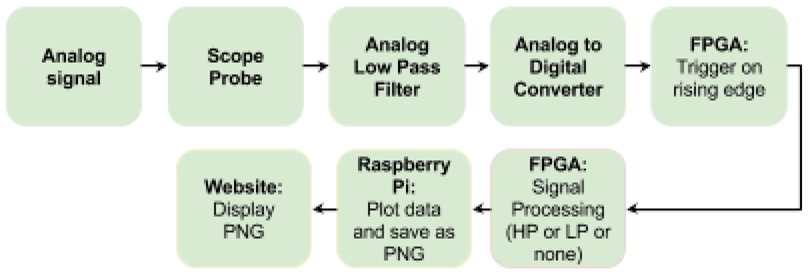

Analog signal
Scope Probe
Analog Low Pass Filter
Analog to Digital Converter
FPGA: Trigger on rising edge
FPGA: Signal Processing (HP or LP or none)
Raspberry Pi: Plot data and save as PNG
Website: Display PNG

Figure 2: System overview of MicrOscope; how signal progresses from input to output

## New Hardware

The project uses a ADS7818 analog to digital converter (ADC) that can sample up to 500kb/s and measure voltages ranging from 0 to 5 volts [10]. The ADC requires 8 connections, the full schematic is in the *Schematics* section of this report, and can be interfaced using a SPI communication protocol.

## Schematic

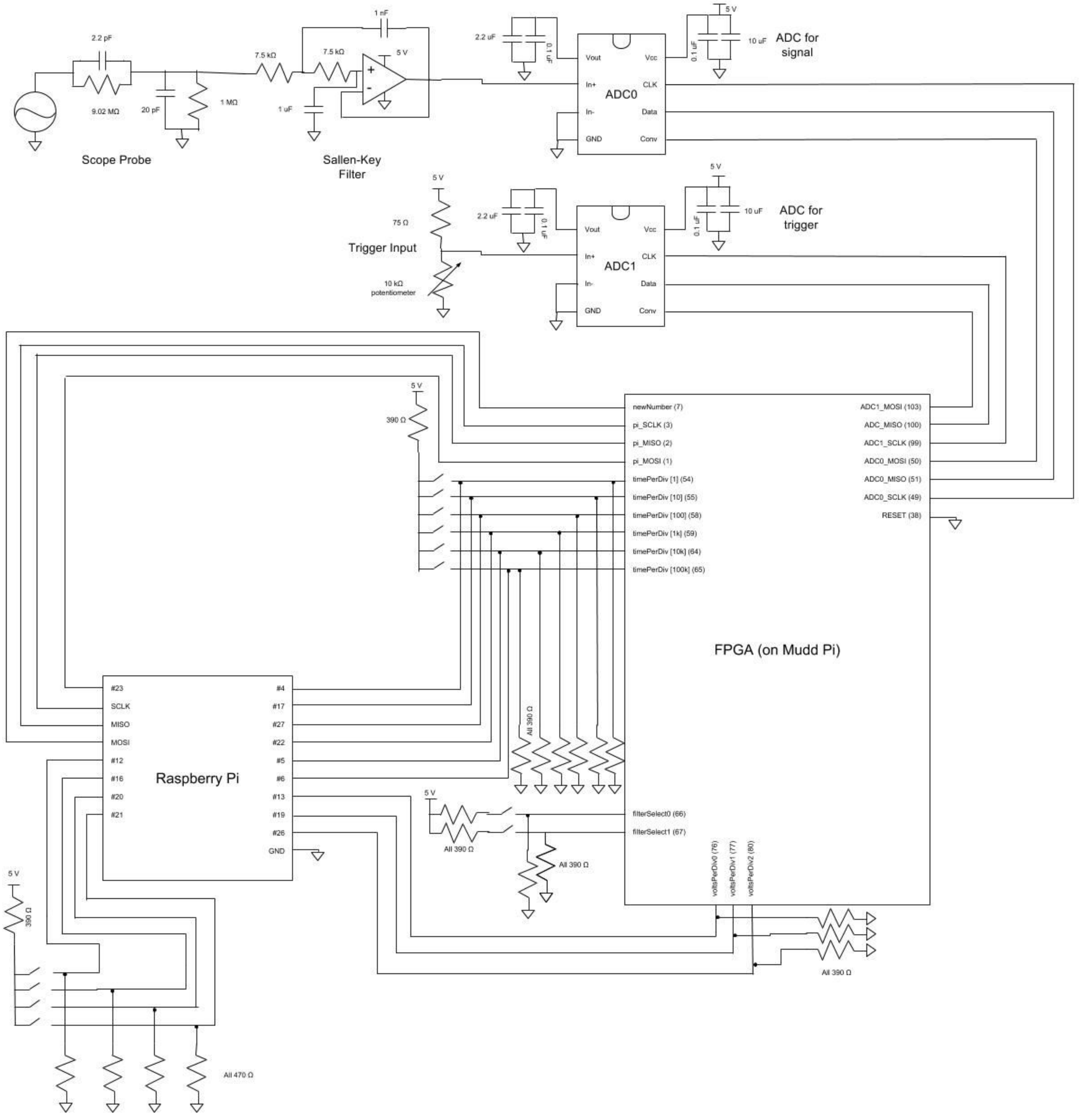


Figure 3: Schematic of overall design

## FPGA Design

### *ADC Communication*

After being passed through the 10x scope probe and the anti-aliasing Sallen Key filter, the analog signal needs to be converted to be handled by the FPGA, which is done by one of the ADS7818 ADCs. Other than the filtering, the same needs to happen for the user's selected trigger value, which comes out of a voltage divider also as an analog signal. This module allows the Mudd Pi board to communicate with the ADS7818 ADCs used. The FPGA communicates to the ADCs using a SPI interface. According to the timing diagram of the ADS7818 found in its datasheet, the FPGA master needs to drive the slave ADC's CONV pin low for 15 cycles and then read data from the MISO wire [10]. The clock speed used in this operation was 1.25MHz. A diagram of the SystemVerilog module implemented is shown in Figure 4.

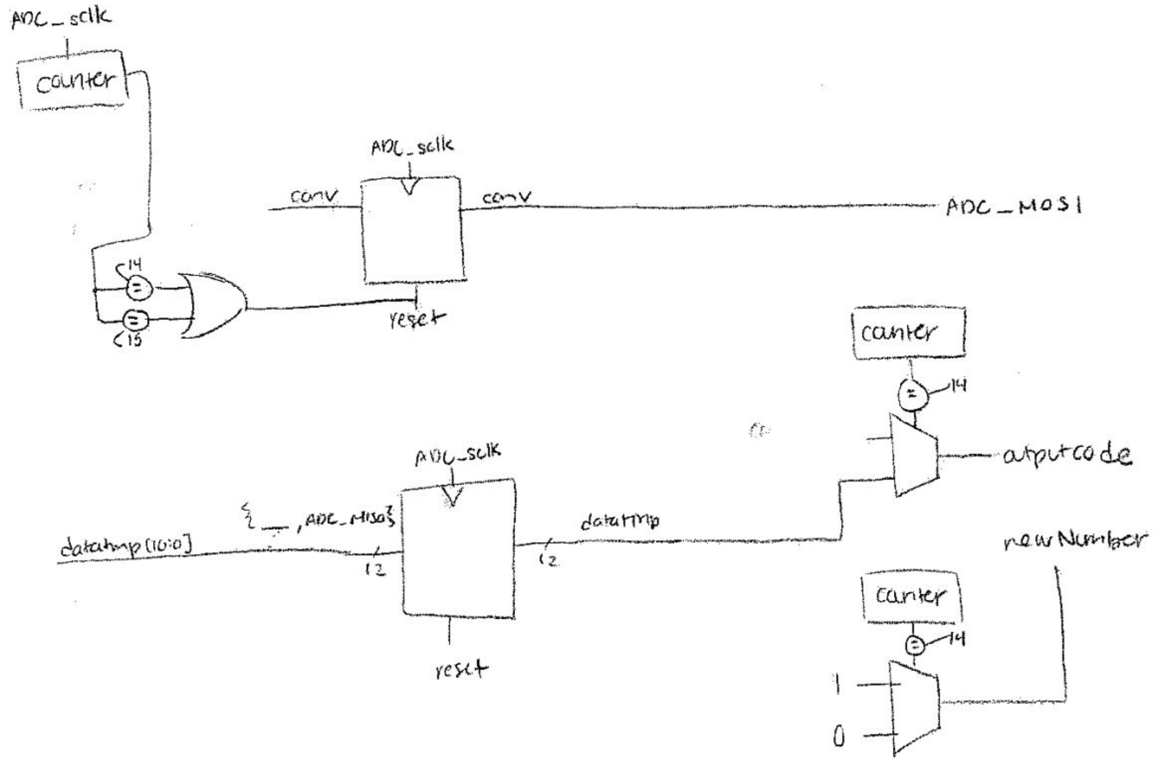


Figure 4: Design for SystemVerilog ADC.sv module used to communicate with ADS7818. Inputs are ADC_sclk, ADC_MISO, reset. Outputs are ADC_MOSI , newNumber, and outputcode.

### *Digital Filtering*

Now the user may want to filter the signal. This module allows the user to run the input signal through a digital high pass filter, digital low pass filter if chosen. The user toggles DIP switches which the Mudd Pi boards. The system implemented two digital filters that can be used by setting 3 DIP switches. The filters used were IIR filters, one low pass the other high pass. The corner frequencies were set by adjusting a constant, alpha, in two difference equations for digital low pass and high pass filters. The equations are shown in Figure 5. The formula used to calculate the corner frequency as well as the formulas in Figure 5 were taken from Wikipedia [4].

```
for i from 1 to n
    y[i] := y[i-1] + α * (x[i] - y[i-1])
```

(a) Digital Low Pass Filter Difference equation

```
for i from 1 to n
  y[i] := α * (y[i-1] + x[i] - x[i-1])
```

(b) Digital High Pass Filter Difference equation

Figure 5: Difference equations used for (a) Digital Low and (b) High Pass Filters

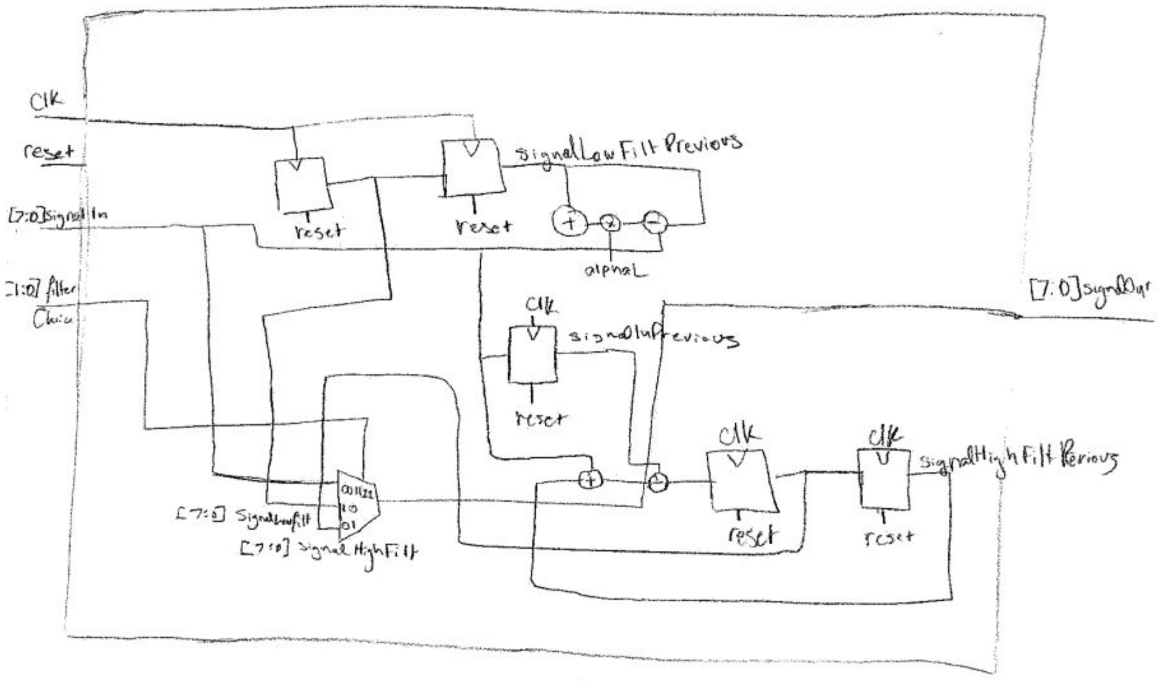


Figure 6: Design for SystemVerilog filter.sv module used to implement filtering. Inputs are clk, reset, signalIn, filter choice. Output is signalOut.

The team stored the y[n-1] and x[n-1] term in difference equations using a flip flop. The team used a mux controlled by the DIP switches in order to determine what the signalOut output should be.

*Triggering*

This module is used to read a signal based on the trigger level set by the user; this a common feature for oscilloscopes. The system used a rising edge triggering scheme. It worked by checking to see if a signal exceeded a trigger threshold value, and then enabling the output a certain number of times based on the current time scale chosen which is one-hot encoded, and then repeating. Figure 7 depicts the SystemVerilog module implemented.

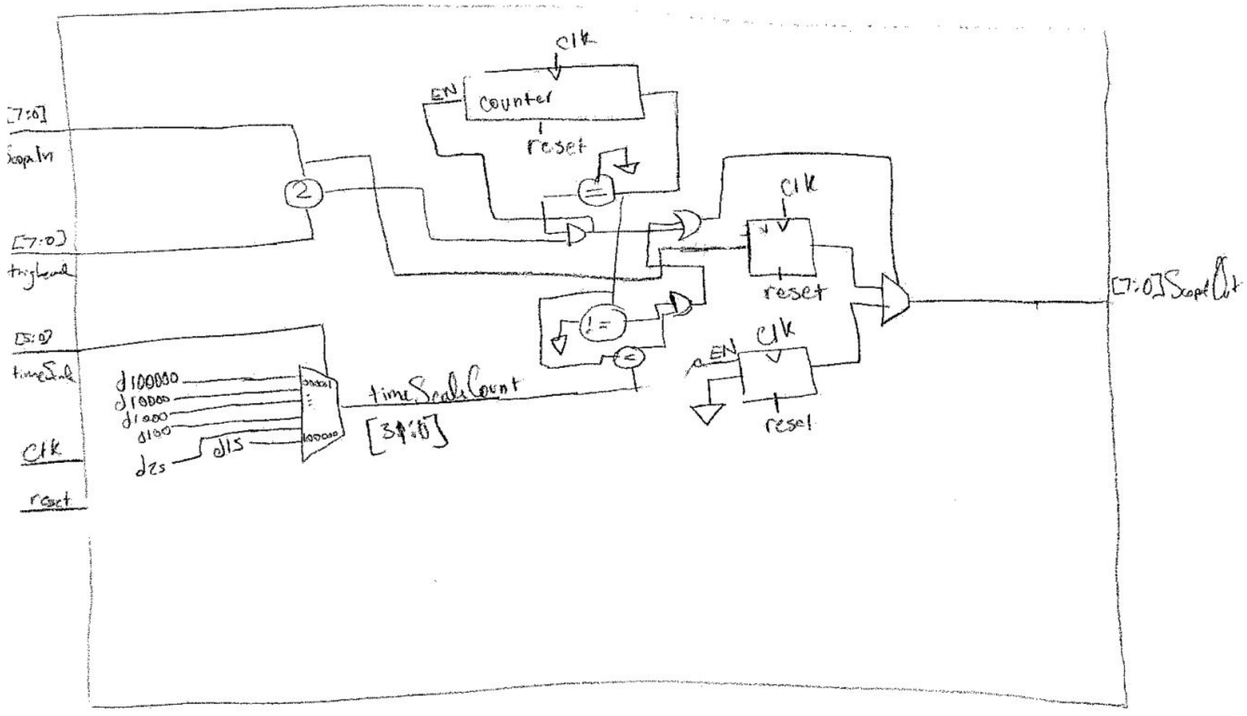


Figure 7: Design for SystemVerilog trig.sv module used to handle user's trigger level input. Inputs are scopeIn, trigLevel, timeScale, clk, and reset. Outputs is scopeOut.

The team used a counter in order count the number of samples passed through when a signal is greater than the trigger level. The team then used a mux to determine if the signal should be allowed to pass through or nothing is allowed to pass through.

*Volts Scale*

This module allows the user to set the voltage scale of the input signal using DIP switches. The scales are one-hot encoded. The system has three volts scales available: 0V to 5V, 0V to 3V, and 3V to 5V. This was accomplished with the voltsScale module. The module takes in two inputs, a 3-bit bus that are attached to the Mudd Pi's onboard DIP switches,voltsScale, and a 8-bit bus data input,signalIn. The module has one 8-bit output, signalOut. The switch inputs what scale to use, the default case is 0 to 5V scale which allows the signal to pass through without any alterations. The second setting checks to see if the input is ever larger than 3V, if it is signalOut is forced to 3 V. Similarly, the third setting to see if the input is ever less than three volts, if signalOut is forced to 3 V. Figure 8 is diagram of the module implemented in Verilog.

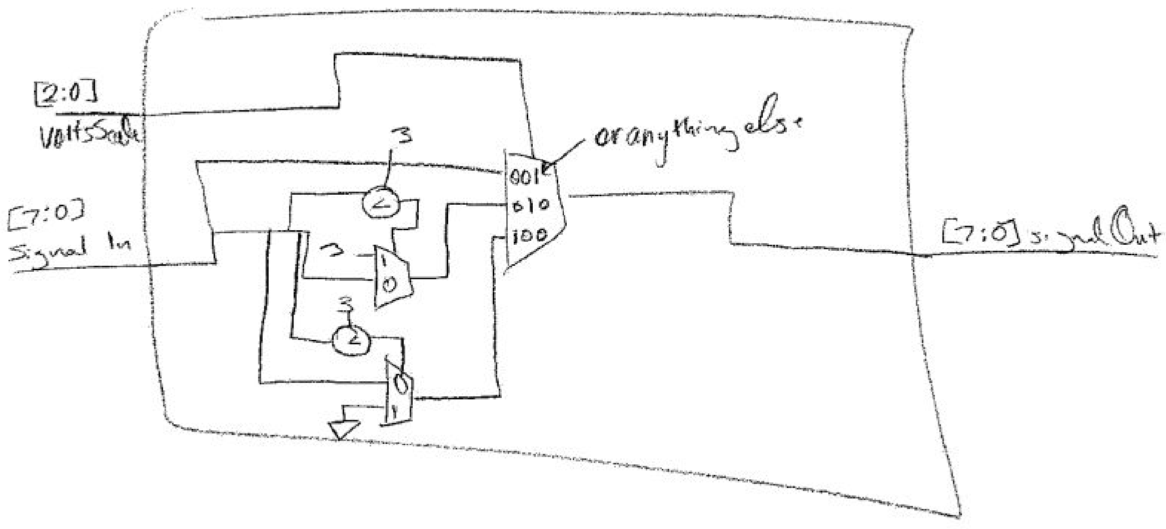


Figure 8: Design for SystemVerilog voltsScale.sv module used to scale the input signal. Inputs are voltsScale, signalIn. Output is signalOut.

*Buffer*

The team used a asynchronous first in first out (FIFO) block to store data in a buffer because the team could not guarantee the same clock for the SPI communication for the ADC and the FPGA and the SPI communication for the FPGA and Raspberry Pi. Originally, the team had tried to design everything with one clock, but needed to switch plans and include this FIFO buffer. As the buffer was not the main focus of the project, the team opted to use an existing FIFO block [1].

*Raspberry Pi Communication*

Finally, the signal data has made its way from being converted from an analog to a digital signal in the ADC, collected over SPI into the FPGA, filtered through a low pass or high pass filter, checked to see if it surpasses the user selected trigger level, and is now being stored in the FIFO buffer. The data now needs to plotted, which is handled by the Raspberry Pi and discussed in the upcoming section. The data is passed from the FIFO buffer in the FPGA to the Pi using SPI, where the Pi functions as the master and the FPGA as the slave. The module pi.sv, the structure of which is show below in Figure 9, builds the slave module required to implement SPI.

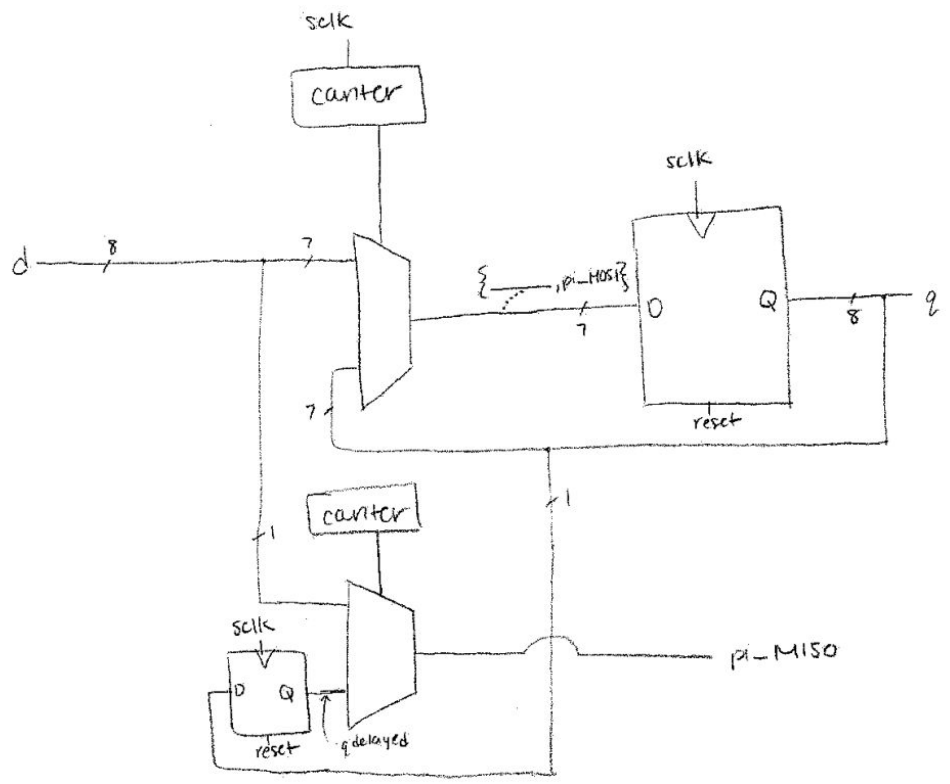


Figure 9: Design for SystemVerilog pi.sv module used to act as the slave for SPI between the Pi master and FPGA slave. Inputs are sclk, reset, d (voltage to send) and pi_MOSI. Outputs are pi_MISO and q (voltage sent).

## Raspberry Pi Design

The Raspberry Pi was used to plot the data received from the Mudd Pi board on a web page. A diagram that depicts how the overall code in the Raspberry Pi is structured is shown in Figure 10.

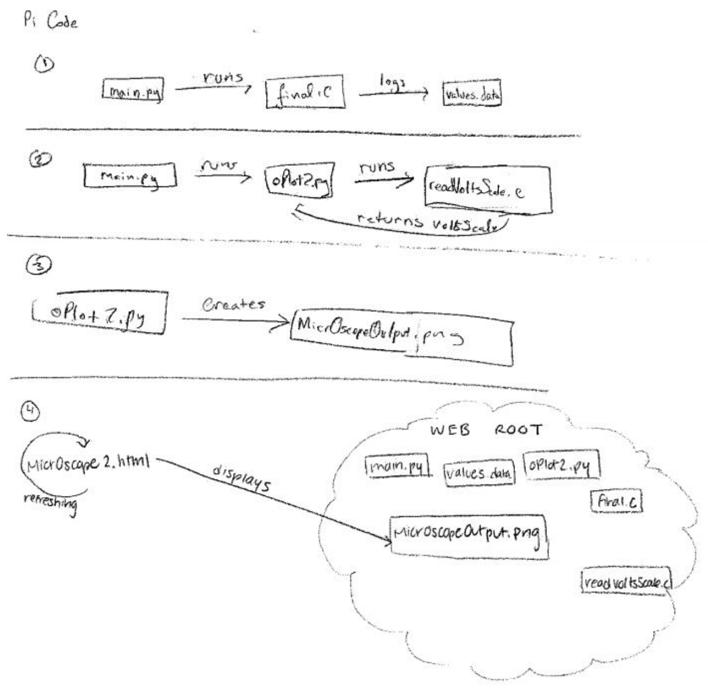


Figure 10: Diagram explaining the flow of programs called by the Raspberry Pi during Operation

*Communication*

To first get the signal data, the Raspberry Pi communicated with the Mudd Pi board using SPI communication protocol. In the program final.c, SPI was implemented to communicate with the FPGA using the EasyPIO.h library [6]. After taking into account various user inputs regarding time per division process and noise filtering, which are discussed in further depth below, the data was stored in values.data.

*Plotting*

The program readVoltsScale.c was written to take the user's choice for volts per division into account (which they choose by setting a physical set of DIP switches). Next, final.c calls this program, and also looks at the user's choices for time scale and noise level configurations from other sets of DIP switches. Figure 11, shown below, details a block diagram of how final.c progressed through to logging signal data into values.data. When oPlot2.py runs, it uses gnuplot-py, a plotting tool, to create a python gnuplot object, label the plot with titles and axes, and read values.data for points to plot. The plot is then saved as a PNG file into the web root of the Raspberry Pi, which for simplicity (albeit not security) is where all of the Pi code is being stored.

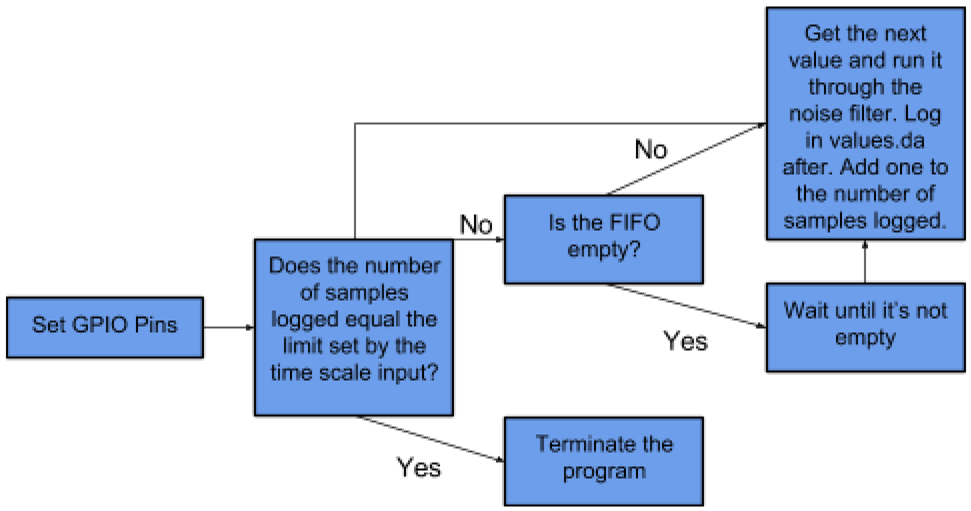

Set GPIO Pins
Does the number of samples logged equal the limit set by the time scale input?
No
Is the FIFO empty?
No
Get the next value and run it through the noise filter. Log in values.da after. Add one to the number of samples logged.
Yes
Wait until it's not empty
Yes
Terminate the program

Figure 11: A block diagram of how data was logged into values.data

*Noise Filtering*

Noise filtering on the Raspberry Pi, as is mentioned above, was accomplished by comparing the current value and previous value recorded using final.c. If the current value was significantly larger or significantly smaller than the previous value recorded (the specific delta tolerance is adjustable by the user with the use of DIP switches), the current value displayed is thrown out and the previous value is plotted. Figure 12 depicts this filtering method.

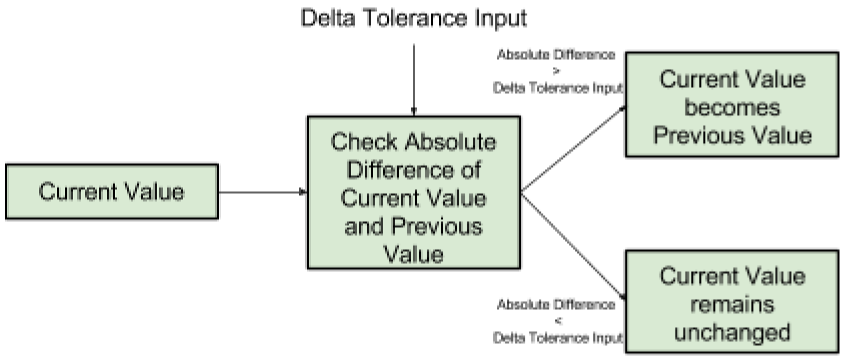

Delta Tolerance Input
Current Value
Check Absolute Difference of Current Value and Previous Value
Absolute Difference > Delta Tolerance Input
Current Value becomes Previous Value
Absolute Difference < Delta Tolerance Input
Current Value remains unchanged

Figure 12: Block diagram depicting how data was processed to take noise into account

*Web Design*

The oscilloscope output was designed to be viewed on a web page. The program main.py serves as the main module on the Pi, running final.c and oPlot2.py over and over which continually asks the FPGA for data over SPI, processes it according to the user inputs, stores it in values.data, creates a plot using gnuplot-py using what is in values.data, and writes over a PNG file to update the output plot. A program, MicrOscope2.html, creates a webpage that the user can access to see MicrOscope's output. It works by displaying the PNG of the output plot, and then automatically refreshing every 0.1 seconds to show the most recent data.

## Results

The system was capable of reporting accurate DC levels of signals, change voltage/division scales, time/division scales, and perform high pass filtering, and triggering. The signal displayed on screen would vary with frequency even if the input signal did not, the team later discovered that this was caused by a timing issue with Raspberry Pi read requests. For future endeavors in low-budget oscilloscope design, a printed circuit board is recommended as they are capable of running faster clocks and thus faster sampling rates by analog to digital converters.

## References

[1] Asynchronous FIFO Block:
http://www.asic-world.com/examples/verilog/asyn_fifo.html
[2] Sallen Key Filter Design Tool:
http://sim.okawa-denshi.jp/en/OPstool.php
[3] Altera Forum Page used to help for ADC.sv
https://alteraforum.com/forum/showthread.php?t=20814
[4] Wikipedia pages on low and high pass filters
https://en.wikipedia.org/wiki/Low-pass_filter#Discrete-time_realization
https://en.wikipedia.org/wiki/High-pass_filter
[6] E155 Class Website
[7] Harris, D. M., Harris, S. L. *Digital Design and Computer Architecture: ARM Edition.* 2016.
[8] gnuplot Resource:
https://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html
[9] HTML Resource:
https://www.w3schools.com/tags/att_meta_http_equiv.asp
[10] ADC7818 Datasheet:
https://www.ti.com/lit/ds/symlink/ads7818.pdf

## Parts List

| **Part** | **Source** | **Vendor Part#** | **Price** |
|---|---|---|---|
| Analog Digital Converter | Digikey | ADS7818 | $6.41 |
| DIP Switches | E155 Cabinet | N/A | N/A |
| 10 kOhm Potentiometer | Electronics Lab Cabinet | N/A | N/A |
| Operational Amplifier | Electronics Lab Cabinet | MCP6002 | N/A |
| Resistors, Capacitors, Wires | E155 & Electronics Lab Cabinets | N/A | N/A |

| | | | |
|---|---|---|---|
| Mudd Pi Board | E155 Course | N/A | $7 |
| Raspberry Pi | E155 Course | N/A | $35 |

## Appendices

### Appendix A: FPGA SystemVerilog Code

**microscope.sv**

```
///////////////////////////////////////////////
// E155 Final Project: MicrOscope
// I. Martos-Repath & A. Echeverria
// First written on 18 November 2017
// Updated 4 December 2017
///////////////////////////////////////////////

/////////////////////////
// ADC_FPGA_SPImaster module
// Function: FPGA (acting as master) gets output codes from ADC
(acting as slave)
/////////////////////////
module microscope(input logic clk, reset,
ADC0_MISO,ADC1_MISO,pi_sclk,pi_MOSI,
                                input logic [5:0]timeScale,
                                input logic [2:0]voltsScale,
                                input logic [1:0]filterSelect,
                                output logic [7:0]led,
                                output logic ADC0_sclk,
ADC0_MOSI,ADC1_sclk, ADC1_MOSI,pi_MISO,empty);
     logic [7:0]scopeOut,signalOutWide;
     logic [7:0]untriggeredSignal;
     logic [7:0]q,d;
     logic [11:0]scopeIn;
     logic newNumber;
     logic ADC_sclk;
     logic [11:0]trigLevel;
     logic [31:0]counter; //counter for clock divider to make sclk
     //logic to generate sclk to pass to ADC slave
     always_ff@(posedge clk, posedge reset) begin
          if(reset) counter <= 32'b0;
```

```
           else begin
           counter <= counter + 1;
           end
      end
      assign led = trigLevel[11:4]; //Used to visually see trigger
value
      assign ADC_sclk = counter[4];
      assign ADC0_sclk  = ADC_sclk;
      assign ADC1_sclk  = ADC_sclk;
      //Read data from ADC's
      ADC triggerLevel(ADC_sclk,reset,ADC0_MISO,ADC0_MOSI,trigLevel);
      ADC
channel1(ADC_sclk,reset,ADC1_MISO,ADC1_MOSI,scopeIn,newNumber);
      //Filter input signal if user sets switches
      filter
lowOrNot(newNumber,reset,scopeIn[11:4],filterSelect,untriggeredSignal
);
      //Check to see if signal passes threshold trigger value
      trig
trigger(untriggeredSignal,trigLevel[11:4],timeScale,ADC_sclk,reset,sc
opeOut);
      //Set Voltage Scale
      voltsScale scaleOutput(voltsScale,scopeOut,signalOutWide);
      logic full;
      logic writeEn_in;
      assign writeEn_in = 1;
      //Buffer used to write data values and read them
      aFifo buffer(d, empty, pi_MOSI, pi_sclk, signalOutWide,full,
writeEn_in,newNumber,reset);
      //SPI Slave module used to interface with Raspberry Pi
      pi(pi_sclk,reset,pi_MOSI,pi_MISO,d,q);
endmodule
```

**ADC.sv**

```
module ADC(input logic ADC_sclk,reset, ADC_MISO,
                  output logic ADC_MOSI,
                  output logic [11:0]outputcode,
                  output logic newNumber);
      logic conv; //needs to go low to start collecting from ADC
      logic [3:0]count; //counter for negative edges of sclk
      logic en;
      logic [11:0]datatmp;
```

```
	//The clock counter starts at 0, so clock is from 0 to 15
instead of 1-16
	always_ff@(posedge ADC_sclk, posedge reset)
		begin
			if(reset) count <= 0;
			else count <= count + 1;
		end
	//Assert the CONV signal
	always_ff@(negedge ADC_sclk, posedge reset)
		begin
			if(reset) conv <= 0;
			else begin
				if((count==4'd14)||(count==4'd15)) conv <= 0;
				else conv <= 1;
			end
		end
	//Read the serial data into a 12-bit register
	always_ff@(negedge ADC_sclk, posedge reset)
		begin
			if(reset) outputcode <= 0;
			else begin
				datatmp <= {datatmp[10:0],ADC_MISO};
				if(count == 4'd14)
				begin
					newNumber <= 1;
					outputcode <=datatmp;
				end
				else newNumber <= 0;
			end
		end
	assign ADC_MOSI = conv;
endmodule
```

**aFifo.sv**

```
//==========================================
// Function : Asynchronous FIFO (w/ 2 asynchronous clocks).
// Coder    : Alex Claros F.
// Date     : 15/May/2005.
// Notes    : This implementation is based on the article
//            'Asynchronous FIFO in Virtex-II FPGAs'
//            writen by Peter Alfke. This TechXclusive
//            article can be downloaded from the
//            Xilinx website. It has some minor modifications.
```

```
//==========================================
module aFifo
  #(parameter    DATA_WIDTH    = 8,
                 ADDRESS_WIDTH = 8,
                 FIFO_DEPTH    = (8 << ADDRESS_WIDTH))
     //Reading port
    (output logic  [DATA_WIDTH-1:0]        Data_out,
     output logic                          Empty_out,
     input logic                          ReadEn_in,
     input logic                          RClk,
     //Writing port.
     input logic  [DATA_WIDTH-1:0]        Data_in,
     output logic                          Full_out,
     input logic                          WriteEn_in,
     input logic                          WClk,
     input logic                          Clear_in);
    /////Internal connections & variables//////
    logic   [DATA_WIDTH-1:0]             Mem [FIFO_DEPTH-1:0];
    logic  [ADDRESS_WIDTH-1:0]          pNextWordToWrite,
pNextWordToRead;
    logic                               EqualAddresses;
    logic                               NextWriteAddressEn,
NextReadAddressEn;
    logic                               Set_Status, Rst_Status;
    logic                                Status;
    logic                               PresetFull, PresetEmpty;
    //////////////Code///////////////
    //Data ports logic:
    //(Uses a dual-port RAM).
    //'Data_out' logic:
    always @ (posedge RClk)
        if (ReadEn_in & !Empty_out)
            Data_out <= Mem[pNextWordToRead];
    //'Data_in' logic:
    always @ (posedge WClk)
        if (WriteEn_in & !Full_out)
            Mem[pNextWordToWrite] <= Data_in;
    //Fifo addresses support logic:
    //'Next Addresses' enable logic:
    assign NextWriteAddressEn = WriteEn_in & ~Full_out;
    assign NextReadAddressEn  = ReadEn_in  & ~Empty_out;
    //Addreses (Gray counters) logic:
    GrayCounter GrayCounter_pWr
```

```
        (.GrayCount_out(pNextWordToWrite),
         .Enable_in(NextWriteAddressEn),
         .Clear_in(Clear_in),
         .Clk(WClk)
        );
    GrayCounter GrayCounter_pRd
        (.GrayCount_out(pNextWordToRead),
         .Enable_in(NextReadAddressEn),
         .Clear_in(Clear_in),
         .Clk(RClk)
        );
    //'EqualAddresses' logic:
    assign EqualAddresses = (pNextWordToWrite == pNextWordToRead);
    //'Quadrant selectors' logic:
    assign Set_Status = (pNextWordToWrite[ADDRESS_WIDTH-2] ~^
pNextWordToRead[ADDRESS_WIDTH-1]) &
                         (pNextWordToWrite[ADDRESS_WIDTH-1] ^
pNextWordToRead[ADDRESS_WIDTH-2]);
    assign Rst_Status = (pNextWordToWrite[ADDRESS_WIDTH-2] ^
pNextWordToRead[ADDRESS_WIDTH-1]) &
                         (pNextWordToWrite[ADDRESS_WIDTH-1] ~^
pNextWordToRead[ADDRESS_WIDTH-2]);
    //'Status' latch logic:
    always @ (Set_Status, Rst_Status, Clear_in) //D Latch w/
Asynchronous Clear & Preset.
        if (Rst_Status | Clear_in)
            Status = 0;  //Going 'Empty'.
        else if (Set_Status)
            Status = 1;  //Going 'Full'.
    //'Full_out' logic for the writing port:
    assign PresetFull = Status & EqualAddresses;  //'Full' Fifo.
    always @ (posedge WClk, posedge PresetFull) //D Flip-Flop w/
Asynchronous Preset.
        if (PresetFull)
            Full_out <= 1;
        else
            Full_out <= 0;
    //'Empty_out' logic for the reading port:
    assign PresetEmpty = ~Status & EqualAddresses;  //'Empty' Fifo.
    always @ (posedge RClk, posedge PresetEmpty)  //D Flip-Flop w/
Asynchronous Preset.
        if (PresetEmpty)
            Empty_out <= 1;
        else
            Empty_out <= 0;
endmodule
```

**graycounter.sv**

```
//===========================================
// Function : Code Gray counter.
// Coder    : Alex Claros F.
// Date     : 15/May/2005.
//========================================
`timescale 1ns/1ps
module GrayCounter
   #(parameter   COUNTER_WIDTH = 4)
    (output reg  [COUNTER_WIDTH-1:0]    GrayCount_out,  //'Gray' code
count output.
     input wire                        Enable_in,  //Count enable.
     input wire                        Clear_in,   //Count reset.
     input wire                        Clk);
    /////////Internal connections & variables///////
    reg    [COUNTER_WIDTH-1:0]         BinaryCount;

    /////////Code//////////////////////
    always @ (posedge Clk)
        if (Clear_in) begin
            BinaryCount   <= {COUNTER_WIDTH{1'b 0}} + 1;  //Gray
count begins @ '1' with
            GrayCount_out <= {COUNTER_WIDTH{1'b 0}};      // first
'Enable_in'.
        end
        else if (Enable_in) begin
            BinaryCount   <= BinaryCount + 1;
            GrayCount_out <= {BinaryCount[COUNTER_WIDTH-1],
                              BinaryCount[COUNTER_WIDTH-2:0] ^
BinaryCount[COUNTER_WIDTH-1:1]};
        end
endmodule
```

**Filter.sv**

```
module filter(input logic clk, reset,
                       input logic [7:0]signalIn,
                       input logic [1:0]filterChoice,
                       output logic [7:0]signalOut);
logic [7:0]signalLowFilt;
logic [7:0]signalHighFilt;
always_comb
begin
     case(filterChoice)
               2'b00 : signalOut = signalIn;
               2'b10 : signalOut = signalLowFilt;
               2'b01 : signalOut = signalHighFilt;
               default : signalOut = signalIn;
     endcase
end
//Low Pass Filter
logic [63:0]alphaL;
assign alphaL = 0.1; //Gives us corner frequency of 40027.70716
logic [7:0]signalLowFiltPrevious;
always_ff@(posedge clk, posedge reset) begin
          if(reset) signalLowFilt <= 0;
          else begin
               signalLowFilt <=
signalLowFiltPrevious+(alphaL*(signalIn-signalLowFiltPrevious));
               signalLowFiltPrevious <= signalLowFilt;
          end
     end
//High Pass filter
logic [63:0]alphaH;
assign alphaH = 0.9; //Gives us corner frequency of 40027.70716
logic [7:0]signalHighFiltPrevious;
logic [7:0]signalInPrevious;
always_ff@(posedge clk, posedge reset) begin
          if(reset) begin
               signalHighFilt <= 0;
               signalInPrevious <= 0;
          end
          else begin
               signalHighFilt <=
alphaH*(signalHighFiltPrevious+signalIn-signalInPrevious);
               signalInPrevious <= signalIn;
               signalHighFiltPrevious <= signalHighFilt;
          end
end
endmodule
```

**pi.sv**

```
module pi(input logic sclk, reset, //from Pi master
                    input logic pi_MOSI,
                    output logic pi_MISO,
                    input logic [7:0]d, //voltage to send
                    output logic [7:0]q); //voltage sent
     logic [2:0]counter; //need to be able to count up to 8
     logic qdelayed;
     always_ff@(negedge sclk, posedge reset)
          if(reset) counter <= 0;
          else counter <= counter + 3'b1;
     always_ff@(posedge sclk)
          q <= (counter == 0)? {d[6:0], pi_MOSI}: {q[6:0], pi_MOSI};
     always_ff@(negedge sclk)
          qdelayed = q[7];
     assign pi_MISO = (counter == 0) ? d[7] : qdelayed;
endmodule
```

**trig.sv**

```
module trig(input logic [7:0]scopeIn,trigLevel,
                    input logic [5:0]timeScale,
                    input logic clk,reset,
                    output logic [7:0]scopeOut);

logic [7:0]scopeInTenX;
assign scopeInTenX = scopeIn;
logic [31:0] writeCount;
logic [31:0] timeScaleCount;
always_comb
begin
     case(timeScale)
               6'b000001 : timeScaleCount = 32'd100000;
               6'b000010 : timeScaleCount = 32'd10000;
               6'b000100 : timeScaleCount = 32'd1000;
               6'b001000 : timeScaleCount = 32'd100;
               6'b010000 : timeScaleCount = 32'd25;
               6'b100000 : timeScaleCount = 32'd15;
               default : timeScaleCount = 32'd200;
     endcase
end
always_ff@(posedge clk, posedge reset) begin
     if(reset) begin
          scopeOut = 0;
          writeCount <= 32'b0;
     end
     else begin
          if((scopeInTenX >= trigLevel)&&(writeCount==32'b0)) begin
               scopeOut <= scopeInTenX;
               writeCount <= writeCount + 1;
          end
          else if((writeCount != 32'd0)&& (writeCount <
timeScaleCount)) begin
          writeCount <= writeCount + 1;
          scopeOut <= scopeInTenX;
          end
          else begin
               writeCount <= 0;
               scopeOut <= 0;
          end
     end
end
endmodule
```

**voltsScale.sv**

```
module voltsScale(input logic [2:0]voltsScale,
                           input logic [7:0]signalIn,
                           output logic [7:0]signalOut);

always_comb
begin
     case(voltsScale)
               3'b001 : signalOut = signalIn;
               3'b010 : begin
                              if(signalIn > 12'd3) signalOut =
12'd3;
                              else signalOut = signalIn;
                              end
               3'b100 : begin
                              if(signalIn < 12'd3) signalOut = 0;
                              else signalOut = signalIn;
                              end
               default : signalOut = signalIn;
     endcase
end
endmodule
```

## Appendix B: Raspberry Pi C, Python, HTML Code

**main.py**

```
#! /usr/bin/env python
import os //for command line inputs
if __name__ == '__main__':
     while(1):
          //Get data from FPGA
          os.system("sudo ./final")
          //Plot it
          os.system("python oPlot2.py")
```

**final.c**

```
#include <stdio.h>
#include "EasyPIO.h"
void setPins(){
       pinMode(4,INPUT);
       pinMode(17,INPUT);
       pinMode(27,INPUT);
       pinMode(22,INPUT);
       pinMode(5,INPUT);
       pinMode(6,INPUT);
       pinMode(24,INPUT);
       pinMode(18,INPUT);
       pinMode(23,OUTPUT);
       pinMode(12,INPUT);//delta = 0.05 //use for DC values
       pinMode(16,INPUT);//delta = 0.5
       pinMode(20,INPUT);//delta = 1.5
       pinMode(21,INPUT);//delta = 4.5
}
 int main(void){
   float voltage;
   char data;
   int empty = 0;
   /*Intialize PIO*/
   pioInit();
   spiInit(8440000,0); //Run this clock much faster than the FPGA to
ensure buffer doesn't fill
   pinMode(23,INPUT);
   int timeScale[6];
   int delta[4];
   int timeCountTrue;
   int timeCount = 0;
   float deltaT = 0.0000128; //From sampling rate of ~79 kHz
   float timeToPlot = 0;
   float lastValue = 0;
   float deltaTolerance = 0;
   setPins();
   while(1){
     timeCount = 0;
     if(timeCountTrue == 1){
           printf("Done with one round of measurements\nStarting
over...\n");
           return 0;
           timeCountTrue = 0;
     }
     while(timeCountTrue == 0){
          while(empty == 1){
               empty = digitalRead(23);
               printf("The buffer is empty\n");
          }
          //Update sensitivity
          delta[0] = digitalRead(12);
          printf("delta[0] is %d\n",delta[0]);
          delta[1] = digitalRead(16);
          delta[2] = digitalRead(20);
          delta[3] = digitalRead(21);

          if(delta[0] == 1) deltaTolerance = 0.05;
          else if(delta[1] == 1) deltaTolerance = 0.5;
          else if(delta[2] == 1) deltaTolerance = 1.5;
          else if(delta[3] == 1) deltaTolerance = 4.5;
          else deltaTolerance = 10;

          lastValue = voltage;
          data =  (spiSendReceive('1'));
          voltage = data*0.01960784313;
          if(timeCount<5) voltage = voltage;
          else{
               if((lastValue-voltage)> deltaTolerance) voltage =
lastValue;
               if((voltage-lastValue)> deltaTolerance) voltage =
lastValue;
          }
          printf("voltage is %f\n",voltage);
          printf("data is %d\n",data);
          timeScale[2] = digitalRead(27);
          //Check timeScale
            if(timeScale[5] == 1){
                timeCountTrue = (timeCount > 4);
            }
            else if(timeScale[4] == 1){
                timeCountTrue = (timeCount > 8);
            }
            else if(timeScale[3] == 1){
                timeCountTrue = (timeCount > 79);
            }
            else if(timeScale[2] == 1){
                timeCountTrue = (timeCount > 782);
                printf("timeCountTrue is %d\n",timeCountTrue);
            }
            else if(timeScale[1] == 1){
                timeCountTrue = (timeCount > 7813);
            }
            else if(timeScale[0] == 1){
                timeCountTrue = (timeCount > 78125);
                }
          timeToPlot = timeCount*deltaT; //plot time based on sample
number and sampling rate
          timeCount = timeCount + 1;
          printf("timeCount is %d\n",timeCount);
          FILE * fp;
          // open the file for writing
          if(timeCount < 2) fp = fopen("values.data","w");
          else fp = fopen ("values.data","a");
          fprintf (fp, "%f %f\n",timeToPlot,voltage);
          // close the file
          fclose (fp);

             timeScale[0] = digitalRead(4);
             timeScale[1] = digitalRead(17);
             timeScale[2] = digitalRead(27);
             timeScale[3] = digitalRead(22);
             timeScale[4] = digitalRead(5);
             timeScale[5] = digitalRead(6);
             printf("timeScale[2] is %d\n",timeScale[2]);

     }
```

```
   }
   return 0;
}
```

**oPlot2.py**

```
#! /usr/bin/env python
""" oPlot.py -- Displays Oscilloscope Output
Run this program by typing python oPlot2.py
"""
from numpy import *
import Gnuplot, Gnuplot.funcutils
import commands
def display():
      """Demonstrate the Gnuplot package."""
      #Get y Axis Bounds
      yAxisBound = commands.getstatusoutput('sudo ./readVoltsScale')
      #Create a plot of current Oscilloscope Output
      g = Gnuplot.Gnuplot(debug=1)
      #Setup title, axis labels etc.
      g.title('MicrOscope')
      g('set style data linespoints')
      g('set grid')
      databuff = Gnuplot.File("values.data",using='1:2')
      g.plot(databuff)
      #Setting the volts/div
      if(yAxisBound[1] == '13'):
            g('set yrange ["0":"5"]')
      elif(yAxisBound[1] == '19'):
            g('set yrange ["0":"2.5"]')
      elif(yAxisBound[1] =='26'):
            g('set yrange ["2.5":"5"]')
      else:
            g('set yrange ["0":"5"]')
      #Generate the PNG
      g.hardcopy(filename="micrOscopeOutput.png",terminal="png")
if __name__ == '__main__':
      display()
```

**readVoltsScale.c**

```
#include <stdio.h>
#include "EasyPIO.h"
void setPins(){
        //TimeScale Pins
```

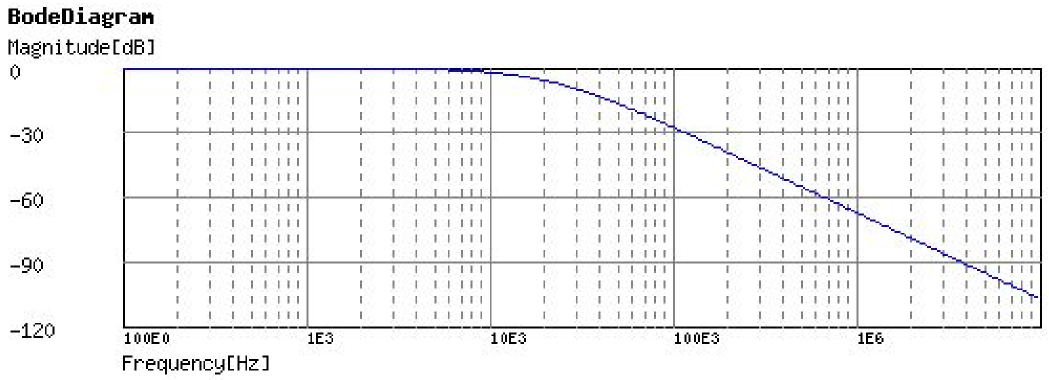
```
    pinMode(13,INPUT);
    pinMode(19,INPUT);
    pinMode(26,INPUT);
}
 int main(void){
  /*Intialize PIO*/
  pioInit();
  int voltScale[3];
  setPins();
  if(digitalRead(13)==1) printf("13"); //0 to 5 V
  else if(digitalRead(19)==1)printf("19"); //0 to 3 V
  else if(digitalRead(26)==1)printf("26"); //3 to 5 v
  return 0;
}
```

**MicrOscope2.html**

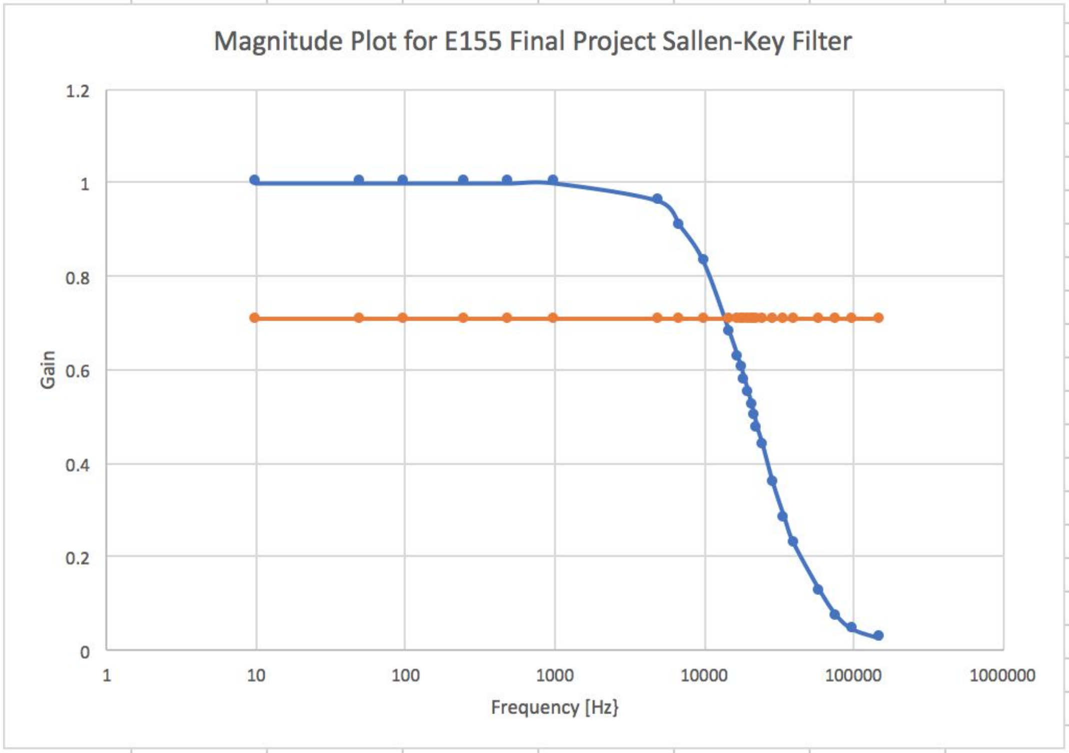
```
<!DOCTYPE html>
<html>
<head>
        <title>MicrOscope</title>
        <meta http-equiv="content-type" content="text-html;charset=utf-8">
        <meta http-equiv="refresh" content="5">
</head>
<body>
        <H2>MicrOscope Output Page</H2>
        <p><img src="micrOscopeOutput.png"></p>
</body>
```

## Appendix C: Analog Filter Verification

- ADC0 sclk, ADC1 sclk runs at 1.25 MHz.
- There's a new number every 14 clock cycles, according to how team has written SPI for FPGA to ADCs.
- This means the sampling rate for the input signal is $\frac{1.25\,MHz}{14} = 89\ kHz$ or $\frac{1\ sample}{11.2\ \mu s}$
- Thus to meet Nyquist requirement, the fastest possible signal the user can enter is 44.5 kHz.
- The Sallen-Key filter is set to have a corner frequency of 21 kHz, well below the maximum possible frequency that meets Nyquist.
  - Designed Bode Plot below using http://sim.okawa-denshi.jp/en/OPstool.php



  - 
  - Experimentally verified Sallen-Key filter by measuring a Bode Plot, as seen below. Corner frequency was slightly less than designed value, at around 19 kHz.



  - 
- The digital filters inside the FPGA both have corner frequencies of 1.3 kHz.