High-Frequency Portable Digital Storage Oscilloscope

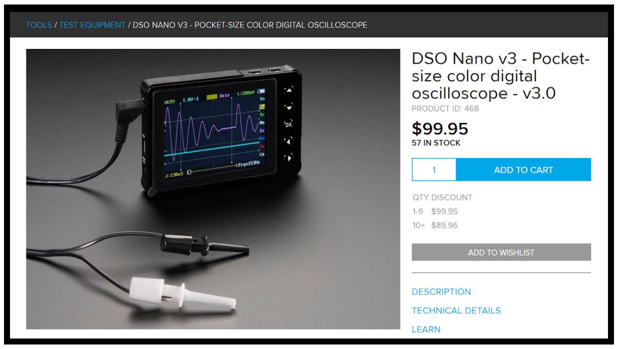
Trevor Fung

E155 Microprocessors 2017

Abstract:
Signal visualization is an important aspect of debugging, testing, and understanding in electronics. One of the most common tools to achieve such visualization is an oscilloscope. This project's purpose was to test various techniques used in high-frequency oscilloscopes and create a low-cost prototype. In this design, an analog front end feeds the input signal into two time-interleaved ADCs. The Cyclone IV FPGA controls the ADCs and stores the 16383 8-bit output words into RAM. A Raspberry Pi 3B in headless mode can then access those values via SPI running at 500 kHz, and print out a plot via gnuplot.

Introduction

Oscilloscopes are incredibly useful tools for debugging, testing, and understanding circuits and electronic devices. The motivation for this project came from an attempt to buy a small, portable, and cheap oscilloscope and the resulting dissatisfaction with the speeds (that is, bandwidths) available commercially. A good example of this problem is the DSO Nano v3: an attractive handheld oscilloscope for under $100, but with a bandwidth of only 100 kHz.



TOOLS / TEST EQUIPMENT / DSO NANO V3 - POCKET-SIZE COLOR DIGITAL OSCILLOSCOPE

DSO Nano v3 - Pocket-size color digital oscilloscope - v3.0

PRODUCT ID: 468

$99.95

57 IN STOCK

1 ADD TO CART

QTY DISCOUNT

1-9 $99.95

10+ $89.96

ADD TO WISHLIST

DESCRIPTION

TECHNICAL DETAILS

LEARN

Figure 1: The DSO Nano v3, as seen on adafruit.com [1]

This is hardly suitable for the probing the digital realm, where clocks are frequently in the the mega- or gigahertz. Some intuition into how much it costs to probe such a signal can be found by looking at Keysight Technology's oscilloscope pricing, as seen below.

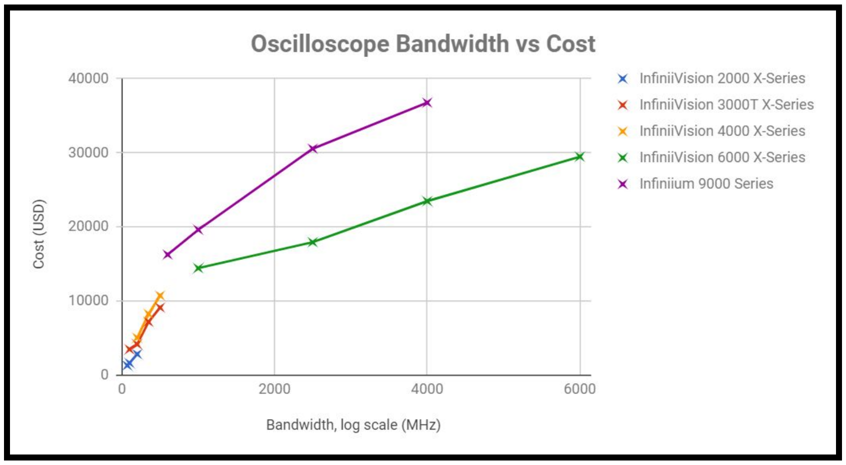


Figure 2: Pricing of 2-channel Keysight Oscilloscopes, retrieved 12/7/17 [2]

The graph looks nearly logarithmic, and it can be seen that in the sub-GHz range the cost to increase bandwidth is fairly steep. So exploring the process of making an oscilloscope faster and faster without the steep increase in cost in that range is a worthwhile exercise.

To understand where this cost comes from, an understanding of how the oscilloscope functions is necessary. There are three kinds of oscilloscope: the analog oscilloscope, digital storage oscilloscope (DSO), and digital phosphor oscilloscope [3]. The digital storage oscilloscope is the most common of the three, and the easiest to understand. The block diagram of a DSO is shown below:

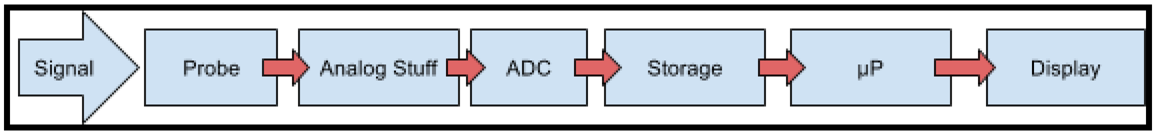


Figure 3: Block diagram of a typical DSO

The signal comes to the oscilloscope through a probe, which ideally acts to get the signal in without significant loading of the circuit under test. The analog block usually contains an amplifier or attenuator to get the signal sufficiently within the ADC's input range, and an anti-aliasing filter to ensure no unexpected high-frequency signals cause measurement errors. There may also be protective circuitry, AC-coupling switches, and other useful features. The ADC (Analog-to-Digital Converter) is arguably the most important part of the oscilloscope, as its specs largely determine the resolution (in number of bits) and bandwidth (in Hertz) of the entire scope. The ADC's reading is then placed in some kind of storage, and a microprocessor takes care of display and other high-level functions like averaging, FFT, and triggering. This does not cover all oscilloscope features or how they all work, but is sufficient for understanding how this project works.

Design Process

The goal of this project was to build a portable, cheap oscilloscope capable of, as a minimum, capturing a 5 V_{PP}, 100 kHz sine wave with no offset. The goals for this oscilloscope are tabulated below.

Specification	**Test Conditions**	**Spec**
Maximum Signal Frequency	50 mV_{PP} sine wave, 0 V_{DC} offset	100 kHz
Maximum Input Voltage	1 kHz sine wave, no DC offset	2.5 V
Minimum Input	1 kHz sine wave, no DC offset	-2.5
Input Bias Current	25°C	<1 µA
Voltage Precision	5 V_{PP} 1 kHz sine wave, 0 V_{DC} offset	< 100 mV

Table 1: Goals for oscilloscope's specs

In order to hit all these goals, the components needed to be small, cheap, and have the ability to function at higher frequencies. The block diagram for this project's design is shown below:

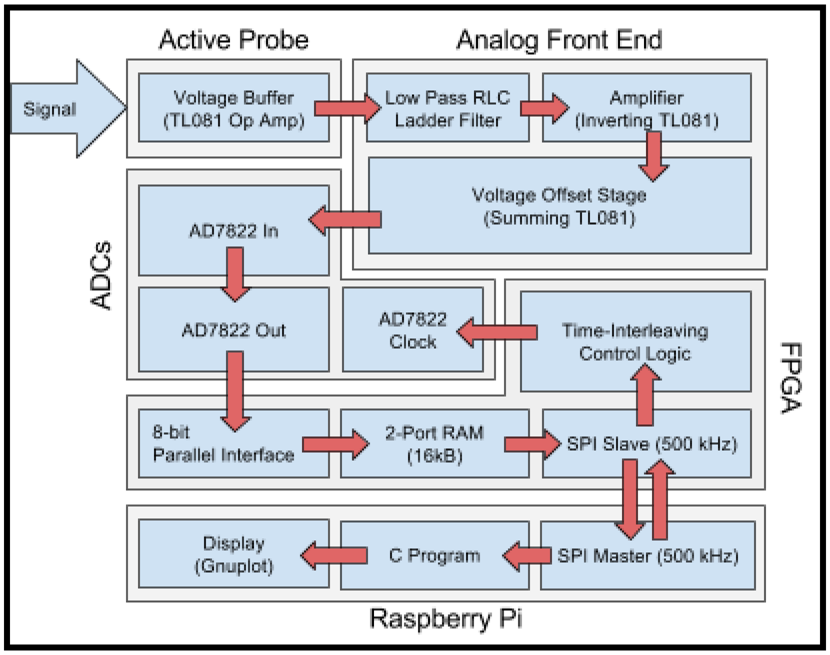


Figure 4: Block diagram of this oscilloscope

By comparing this to the block diagram for a general DSO, one might note the following devices filling the following functions:

- A voltage buffer acting as a probe
- Anti-aliasing filter and various amplifiers doing analog stuff
- Two 8-bit, 2 Msps AD7822 doing analog-to-digital conversion
- A Cyclone IV EP4CE6E22C8 FPGA storing the incoming data and controlling the ADCs
- A Raspberry Pi 3B communicating with the FPGA and plotting the data

The op amps, ladder filter, FPGA, and Pi were chosen based on availability, and the ADCs were the fastest through-hole ADC available online at the time. All the components fit nicely onto a portable breadboard, and collectively cost about $100. The ADCs are 8-bit, which in this case corresponds to 50 mV precision after a signal passes through the analog attenuation and shifting. They also can sample at up to 2 MHz, which is far more than twice the desired input frequency and therefore satisfactory.

Because loading the circuit under test affects the signal to measure and visualize, it is desirable to use a probing method that doesn't cause loading. I started by trying to use resistors and capacitors to create the equivalent circuit of a 10x passive probe connected to an oscilloscope. This worked fine in simulation, and failed only at ~300 MHz, so it wouldn't be a limiting factor in terms of speed. The circuit diagram is shown below.

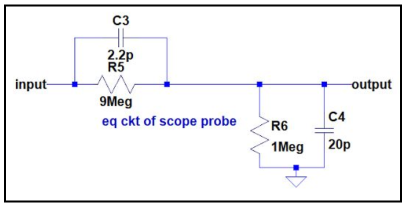


Figure 5: Equivalent circuit of 10x oscilloscope probe

I found that immediately feeding the output into the anti-aliasing filter was a pain, so I separated the two stages with a voltage buffer/unity gain buffer made with a TL081 JFET-input op amp, chosen for the very low input bias current. I realized soon after that the equivalent circuit of the probe wasn't actually doing much, and went back to researching existing probes. During my initial research I had completely neglected active probes, which unlike passive probes are not limited to only passive elements. I learned active probes trade off input voltage ranges for higher speed (see below).

Probe Type	Typical Useful Frequency Range	Typical Maximum Input Voltage
Passive, High Impedance (1 MOhm)	0 to 50 MHz	600 Volts
Passive, Low Impedance (500 Ohms)	0 to 8 GHz	20 Volts
Active	0 to 2 GHz	10 Volts

Figure 6: Comparison table of probe types [4]

As my input voltage range is already limited heavily by the ADC, this trade-off is fine. I then scrapped the passive probe idea, and am just using a voltage buffer as my input stage (see below).

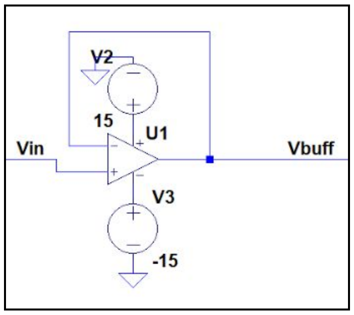
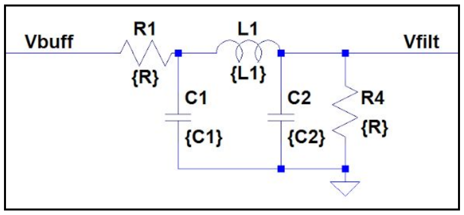


Figure 7: Active probe design used

According to the Shannon-Nyquist Theorem, to avoid any aliasing, signals with input frequencies greater than the ADC's sampling frequency divided by 2 ($f_s/2$) should be attenuated. To that end, I used a third-order Butterworth RLC ladder filter, chosen for the flat frequency response in the passband, fair amount of attenuation (~ -18 dB/octave), and ease of design (roughly speaking, two capacitors of a value, one inductor of twice that value, and two identical resistors). Because the ADC's f_s is 2 MHz, I chose the corner frequency of the filter to be ~500 kHz in order to have a -21 dB attenuation at 1 MHz. I then chose the value of R to work with the maximum output current of the TL081 buffering stage. The topology and values are shown below. Additionally, this filter contributes a gain of ½ in the passband.



R	C1	L1	C2
5 kΩ	60 pF	3 mH	60 pF

Figure 8: Butterworth RLC ladder filter design

Because the input range of the ADC is only 0 to 2 V when V_{CC} = 3.3 V, I need to center the input voltage at 1 V. I will also further attenuate the signal by ½ to ensure that the input stays away from the ADC's limits. The circuit is shown below:

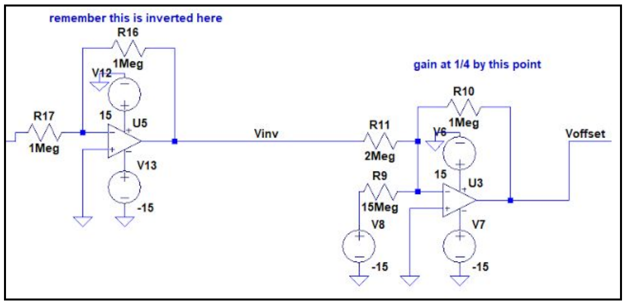


Figure 9: Attenuator and shifter circuit

The TL081 was again chosen, this time for its 3 MHz unity gain bandwidth, which is more than enough for this application here. Because the summing amplifier stage inverts the output and then adds it to the offset, an initial inversion (the first stage on the left) is needed to undo the second inversion. The second stage adds 15/15 = 1 V and the doubly-inverted signal multiplied by ½ to produce the final output for the ADC. Below are example waveforms after every part of the analog front end. The design, once built, was found to have the characteristics simulated.

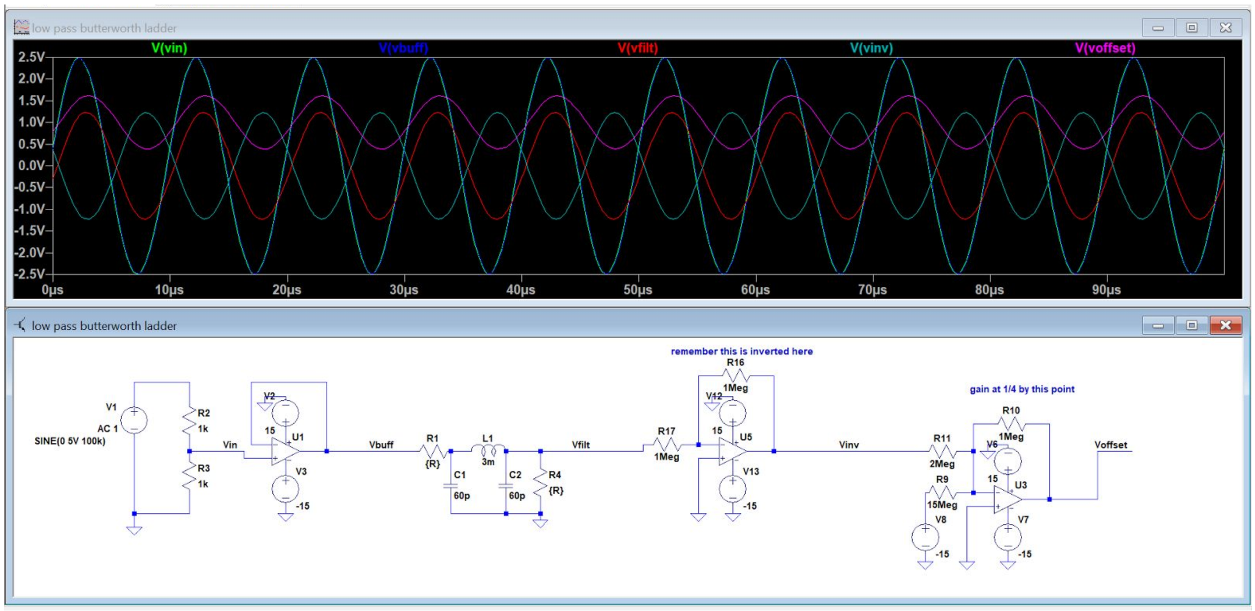


Figure 10: Fully-functioning analog front end

This analog front end then feeds into two AD7822s, which are alternate sampling (a technique known as time-interleaving) to effectively create a single ADC with twice their individual sampling rates. Because the clock of the FPGA is only 40 MHz, I was only able to get the effective ADC sampling at 3.3 Msps, which is still far more efficient than a single ADC. The setup diagram for the ADC is shown below.

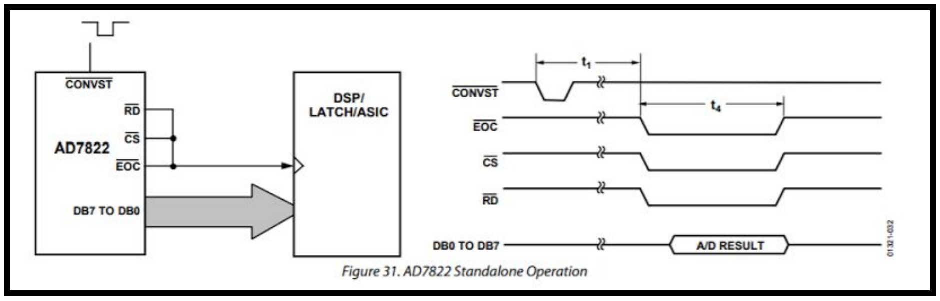


Figure 10: ADC timing diagram [5]

The minimum possible time between ADC trigger pulses is 540 ns, which corresponds to 23 cycles on the FPGA. By extending this counter to 24 and sending a pulse to an ADC every 12, we can trigger the ADCs in an alternating fashion. By muxing between the two ADCs and selecting the currently valid output, we achieve time-interleaving.

The output of that mux is then placed into a two-port RAM created with the Altera MegaFunction Wizard. The RAM can hold 16384 8-bit words, and so the write address is controlled by a counter which resets when the RAM is full or when the Pi calls for a new set of data over SPI. Writing to the RAM is RAM is disabled when the Pi calls for a read of the RAM. Because I wanted the ADCs to operate at a fairly fast clock that the Pi's SPI might not be able to match, the RAM has independent read and write clocks. The write clock is the FPGA's 40 MHz clock, and the read clock is the SPI's 500 kHz sclk. Reading out of the RAM is controlled by another counter that starts incrementing once the Pi starts reading over SPI.

The SPI module takes as input sclk, miso, mosi, and a started signal which tells all the FPGA logic that a transmission is about to take place. After the started signal goes low, an edge detector clears the RAM and RAM address counters. The RAM then fills up and sends a signal to say that it's full to the sclk domain. The SPI module then starts reading from RAM, shifting out bit by bit on a shift register, and incrementing the read counter. A block diagram of all the FPGA logic is shown below, and the Verilog code can be found in the **Appendix A**.

The Pi's C program uses the GPIO and SPI functions defined earlier in class, and just writes low to the started pin and initiates SPI enough times to read the entire RAM, storing it in an array. The program then writes out the array into a .dat file, which can be plotted on gnuplot. The Pi's code can be found in **Appendix B**. To capture an oscilloscope trace, the user resets the FPGA, runs the compiled C program, and has gnuplot plot out the output file.

Results

The specs achieved with this design are tabulated below. All desired thresholds were reached.

Spec	**Test Conditions**	**Spec**	**Theoretical limit and notes**
Maximum Signal Frequency	50 mV$_{PP}$ sine wave, 0 V$_{DC}$ offset	100 kHz	~500 kHz, which is the -3 dB corner of the anti-aliasing filter, and ¼ of the ADC sampling frequency
Maximum Input Voltage	1 kHz sine wave, no DC offset	2.5 V	4V, because the front end attenuates by ¼ then adds 1 V$_{DC}$, and the max

			input to the ADC is 2 V
Minimum Input	1 kHz sine wave, no DC offset	-2.5	-4 V, because with ¼ attenuation and 1 V_{DC} offset, it becomes 0V, which is the min input to the ADC
Input Bias Current	25°C	30 pA typical	Taken from TL081's datasheet, as that's the very first part of the front end. Should mean that inadvertent circuit loading isn't an issue
Voltage Precision	5 V_{PP} 1 kHz sine wave, 0 V_{DC} offset	50 mV	31 mV, as the ADC is 8-bit, so with a 2 V input range that's a precision of ~7.8 mV. Divide by the attenuation of ¼ to get ~31 mV.

Table 2: Specifications of final design

Because this design was created with the idea of speed in mind, it's important to discuss the limitations of the current design. While the design did satisfy the goal of a 100 kHz sine wave, it is important to note that it cannot fully represent a 100 kHz square wave, as seen below:

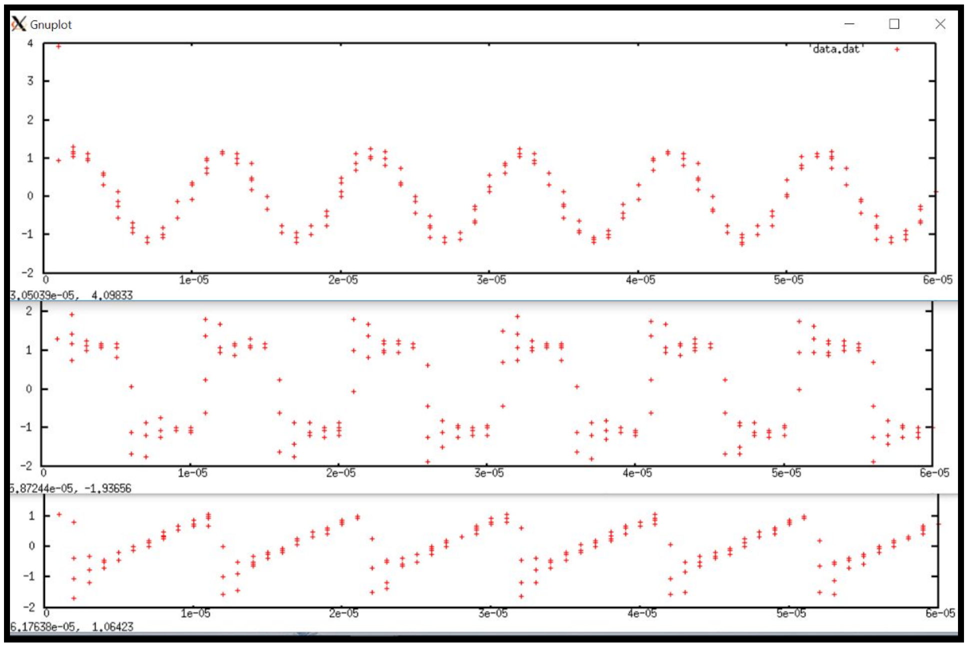


Figure 11: A sine wave, square wave, and sawtooth as seen by the oscilloscope at 2.5 V_{PP}, 100 kHz, no offset

This is because visualizing a square wave requires properly capturing the harmonics, and so the bandwidth of an oscilloscope should be five times the desired digital signal's frequency [6]. So to go faster, I note the limitations of the current system below:

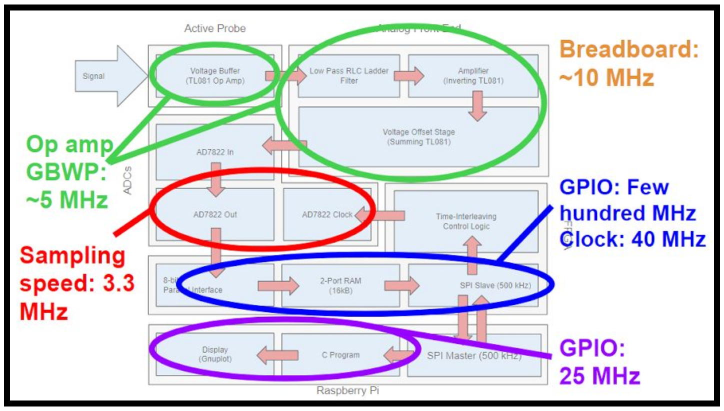


Figure 12: Current system speed limits

Getting ADCs with higher sampling frequencies and bandwidths and op amps with larger gain bandwidth products are clear steps. Also designing a PCB would remove the breadboard's built-in speed limit. Because of the current design's two clock domains, the Pi's limits can be negated by always running SPI at some manageable speed. Interleaving more ADCs is another option. Impedance matching in the analog front end would also start to become appreciable at higher frequencies.

Although this design fulfills all the goals set out in the project proposal, it does have several flaws. First, it effectively acts as a camera, only displaying data when triggered by the C program running because I could not figure out how to display a live graph over SSH or on a webpage hosted by the Pi. To display, the user also has to run gnuplot from the command line, as I didn't put the command to plot inside the C program itself. Furthermore, the ADC's RAM doesn't clear out after the first run properly, and so before every scope capture the user must reset the FPGA's board. Also, the first three data points captured by the FPGA and every ~2000th point after is the maximum value possible, which seems to indicate a systemic error. Also, gnuplot would stack points that were "close enough" in time, which makes the graph look a little weird. None of these issues are deal breakers, but collectively serve to relegate this project to the "prototype" realm.

If I had more time, in addition to cleaning up the aforementioned bugs, I would've added more features. I would've added switches and potentiometers to the breadboard for AC coupling and triggering functions. I had ordered a digital potentiometer to make the input amplifier adjustable from the Pi, but unfortunately it did not arrive in time. I also would've liked to have the Pi display through a live-updating webpage. I would've also liked to have put the entire front end and FPGA on a PCB, and put the entire thing in a plastic case or the like.

References

[1] https://www.adafruit.com/product/468
[2] https://www.keysight.com/en/pcx-x2015004/oscilloscopes?cc=US&lc=eng&tab=all&state=3
[3] http://ecee.colorado.edu/~mcclurel/txyzscopes.pdf
[4] http://teledynelecroy.com/doc/probes-probing
[5] http://www.mouser.com/ds/2/609/AD7822_7825_7829-877543.pdf
[6] https://www.mouser.com/pdfdocs/Tektronix12_things_to_consider1.pdf

Parts List

Part	Source	Vendor Part #	Price
ADC (x2)	Mouser.com	AD7822	$11.99 each
Op Amp	Stock Room	TL081	$0.51 each
Various standard resistors and capacitors	Stock Room	n/a	<$1 total

Parts listed are all those not provided as part of the class. Full BOM for MuddPi board and Raspberry Pi can be found at http://pages.hmc.edu/harris/class/e155/MuddPi_MarkIV-BOM.xls and https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md respectively

Appendix A

```
module testbench();
	logic clk;
	logic reset;
	logic EOCbar1;
	logic EOCbar2;
	logic [7:0] dataIn1;
	logic [7:0] dataIn2;
	logic sclk;
	logic mosi;
	logic starterPin;
	logic CSbar;
	logic miso;
	logic CONVSTbar1;
	logic CONVSTbar2;
	logic [7:0] toLEDs;

	buffer
dut(clk,reset,EOCbar1,EOCbar2,dataIn1,dataIn2,sclk,mosi,starterPin,CSbar,miso,CONVSTbar1,CONVSTbar2,toLEDs);

	//clk wave
	initial begin
		clk = 1'b0;
		forever #1 clk = ~clk;
	end

	//sclk wave
	initial begin
		sclk = 1'b0;
		repeat(16) @(posedge clk);
		forever #80 sclk = ~sclk;
	end

	//reset wave
	initial begin
		reset = 1'b1;
		repeat(3) @(posedge sclk);
		reset = 1'b0;
	end

	//CONVSTbar1 wave
```

```
initial begin
	//initial values
	EOCbar1 = 1'b1;
	dataIn1 = 8'b00000000;
	@(negedge reset); //wait for reset
		forever begin
		@(negedge CONVSTbar1) begin
			repeat(16) @(posedge clk);
			EOCbar1 = 1'b0;
			repeat(4) @(posedge clk);
			EOCbar1 = 1'b1;
			dataIn1 = 8'b10000001;
		end
	end
end

//CONVSTbar2 wave
initial begin
	//initial values
	EOCbar2 = 1'b1;
	dataIn2 = 8'b00000000;
	@(negedge reset); //wait for reset
		forever begin
		@(negedge CONVSTbar2) begin
			repeat(16) @(posedge clk);
			EOCbar2 = 1'b0;
			repeat(4) @(posedge clk);
			EOCbar2 = 1'b1;
			dataIn2 = 8'b10000001;
		end
	end
end

//starterPin wave
initial begin
	starterPin = 1'b0;
	repeat(8) @(posedge sclk);
	starterPin = 1'b1;
	forever begin
		repeat(2500) @(posedge sclk); //2500 kinda arbitrary, but basically waiting for buffer to fill
		starterPin = 1'b0;
		repeat(294920) @(posedge sclk); //over how long it should take to pass off entire word
		starterPin = 1'b1; //not super sure about these timings, would not trust entirely
	end
end

//CSbar and mosi wave
initial begin
	CSbar = 1'b1;
	mosi = 1'b0;
	repeat(8) @(posedge sclk);
	forever begin
		repeat(2510) @(posedge sclk); //arbitrary, but basically waiting for buffer to fill
		repeat(16383) begin
			CSbar = 1'b0;
			repeat(17) @(posedge sclk);
			CSbar = 1'b1;
```

```
                                        @(posedge sclk);
                                end
                        end
                end
endmodule

/* /E155 Final Project: Oscilloscope/
                Trevor Fung (tfung@hmc.edu)
                11/26/17
                Module: buffer
                This is the master module that links together
                the control logic for the ADC, the buffer,
                and the SPI connection to the Pi.
*/
module buffer(input logic clk,
                                        input logic reset,
                                        input logic EOCbar1,
                                        input logic EOCbar2,
                                        input logic [7:0] dataIn1,
                                        input logic [7:0] dataIn2,
                                        input sclk,
                                        input mosi,
                                        input starterPin,
                                        input CSbar,
                                        output miso,
                                        output logic CONVSTbar1,
                                        output logic CONVSTbar2,
                                        output logic [7:0] toLEDs); //toLEDs formerly known as dataOut
                logic [7:0] tempQ, toMux1, toMux2, ramOut;
                logic [14:0] readRamCounter;
                logic start, synch, synchdStart, ramFull, ramFullSynchd,wren;
                assign toLEDs = tempQ;

                //Synchronizer to FPGA clk domain for start
                always_ff @(posedge clk)
                begin
                        synch <= start;
                        synchdStart <= synch;
                end

                ///Counter for timing on ADCs
                logic [5:0] counter; //to hold 24
                //simple counter that resets
                // after counting to 23
                always_ff @(posedge clk)
                begin
                        counter <= counter + 6'b1;
                        if((counter > 22) | reset)
                        begin
                                counter <= 0;
                        end
                end

                ///Counter for ram writing
                logic [14:0] writeRamCounter; //to hold addresses 0-16383
                //only want to write to RAM if RAM not full
                assign wren = !writeRamCounter[14] & ((counter == 0) | (counter == 12));
```

```
		//simple counter that stops
		// after counting to 16383
		always_ff @(posedge clk)
		begin
			if(reset | synchdStart) writeRamCounter <= 0;
			else if(wren) writeRamCounter <= writeRamCounter + 15'b1;
		end

		//Synchronizer to tell SPI clk domain RAM is full
		always_ff @(posedge sclk)
		begin
			ramFull <= writeRamCounter[14];
			ramFullSynchd <= ramFull;
		end

		ADCcontrol1 parallelInterface1(clk,reset,EOCbar1,dataIn1,counter,CONVSTbar1,toMux1);
		ADCcontrol2 parallelInterface2(clk,reset,EOCbar2,dataIn2,counter,CONVSTbar2,toMux2);
		mux2 muxer(clk,toMux1,toMux2,counter,tempQ);
//		FIFO circularBuffer(clk,tempQ);
		ram storage(.data(tempQ), //can only store 16384 8-bit words - enough for 500 periods of 100kHz @ 3.3Mhz sampling
						.rdaddress(readRamCounter[13:0]),
						.rdclock(sclk),
						.wraddress(writeRamCounter[13:0]),
						.wrclock(clk),
						.wren(wren),
						.q(ramOut) );
		spiSlave spi(reset,sclk,starterPin,CSbar,miso,mosi,ramOut,ramFullSynchd,start,readRamCounter);

endmodule


/* /E155 Final Project: Oscilloscope/
		Trevor Fung (tfung@hmc.edu)
		11/26/17
		Module: ADCcontrol1
		This is one of the master module that sends out an
		appropriate CONVSTbar signal, and buffers the
		data seen with registers
*/
module ADCcontrol1(input logic clk,
							input logic reset,
							input logic EOCbar1,
							input logic [7:0] dataIn1,
							input logic [5:0] counter,
							output logic CONVSTbar1,
							output logic [7:0] toMux1);

		//making CONVSTbar only pulse low once per 24 clock cycles
		assign CONVSTbar1 = ~(counter == 0);

		logic dataGood1;
		assign dataGood1 = (counter == 23) & EOCbar1;

		always_ff @(posedge clk)
		begin
			if(reset) toMux1 <= 0;
			else if(dataGood1)
			begin
				toMux1 <= dataIn1;
			end
		end
endmodule

/* /E155 Final Project: Oscilloscope/
		Trevor Fung (tfung@hmc.edu)
		11/26/17
		Module: ADCcontrol2
		This is one of the master module that sends out an
		appropriate CONVSTbar signal, and buffers the
		data seen with registers
*/
module ADCcontrol2(input logic clk,
							input logic reset,
							input logic EOCbar2,
							input logic [7:0] dataIn2,
							input logic [5:0] counter,
							output logic CONVSTbar2,
							output logic [7:0] toMux2);

		//making CONVSTbar only pulse low once per 24 clock cycles
		assign CONVSTbar2 = ~(counter == 12);

		logic dataGood2;
		assign dataGood2 = (counter == 11) & EOCbar2;

		always_ff @(posedge clk)
		begin
			if(reset) toMux2 <= 0;
			else if(dataGood2)
			begin
				toMux2 <= dataIn2;
			end
		end
endmodule

/* /E155 Final Project: Oscilloscope/
		Trevor Fung (tfung@hmc.edu)
		11/26/17
		Module: mux2
		Standard 2-input mux with register attached. The mux
		picks between the inputs depending on which one should be
		valid, and a register only allows the valid input to go out
		to the buffer.
*/
module mux2(input logic clk,
					input logic [7:0] input1,
					input logic [7:0] input2,
					input logic [5:0] counter,
					output logic [7:0] out);

		always_ff @(posedge clk)
		begin
			if(counter == 23)
			begin
```

```
				out <= input1;
			end
			else if(counter == 11)
			begin
				out <= input2;
			end
		end
endmodule

/* /E155 Final Project: Oscilloscope/
		Trevor Fung (tfung@hmc.edu)
		12/4/17
		Module: spiSlave
		Takes care of SPI to the Pi. A rising edge on starterPin indicates the
		beginning of a cycle. start is an output signal
		that goes high for a few clock cycles upon seeing mosi
		go high; it acts as a reset/clear for the ram address counters.
		After RAM fills up, starts to shift out on miso
*/
module spiSlave(input logic reset,
						input logic sclk,
						input logic starterPin,
						input logic CSbar,
						output logic miso,
						input logic mosi, //irrelevant on this end
						input logic [7:0] ramOut,
						input logic ramFullSynchd,
						output logic start,
						output logic [14:0] readRamCounter);
		logic synchq, starterPinSynchd,edgeDetectorq;
		//Synchronizer on starterPin
		always_ff @(posedge sclk)
		begin
			synchq <= starterPin;
			starterPinSynchd <= synchq;
		end

		//posedge detector on starterPin
		assign start = starterPinSynchd & ~edgeDetectorq;
		always_ff @(posedge sclk)
		begin
			edgeDetectorq <= starterPinSynchd;
		end

//		///Counter for ram reading to hold addresses 0-16383
//		//simple counter that keeps track of what address we're outputting
//		always_ff @(posedge sclk)
//		begin
//			if(reset | start) readRamCounter <= 0;
//		end

		//count number of bits in current transmission
		logic [3:0] bitCount;
		always_ff @(posedge sclk)
		begin
			if(CSbar | reset) bitCount <= 0;
			else bitCount <= bitCount + 4'b1;
```

```
	end

	logic [7:0] shiftReg;
	always_ff @(posedge sclk)
	begin
		if(reset | start) readRamCounter <= 0;
		else if( (bitCount == 8) & ramFullSynchd)
		begin
			shiftReg <= ramOut;
			if(!readRamCounter[14]) readRamCounter <= readRamCounter + 15'b1;
		end
		else shiftReg <= {shiftReg[6:0],1'b0};
	end
	assign miso = shiftReg[7];
endmodule



// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: smallerRAM.v
// Megafunction Name(s):
// 			altsyncram
//
// Simulation Library Files(s):
// 			altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
// ************************************************************


//Copyright (C) 1991-2013 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module ram (
		data,
		rdaddress,
		rdclock,
		wraddress,
		wrclock,
		wren,
		q);

		input	[7:0]  data;
		input	[13:0]  rdaddress;
		input	  rdclock;
		input	[13:0]  wraddress;
		input	  wrclock;
		input	  wren;
		output	[7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
		tri1	  wrclock;
		tri0	  wren;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

		wire [7:0] sub_wire0;
		wire [7:0] q = sub_wire0[7:0];

		altsyncram	altsyncram_component (
					.address_a (wraddress),
					.clock0 (wrclock),
					.data_a (data),
					.wren_a (wren),
					.address_b (rdaddress),
					.clock1 (rdclock),
					.q_b (sub_wire0),
					.aclr0 (1'b0),
					.aclr1 (1'b0),
					.addressstall_a (1'b0),
					.addressstall_b (1'b0),
					.byteena_a (1'b1),
					.byteena_b (1'b1),
					.clocken0 (1'b1),
					.clocken1 (1'b1),
					.clocken2 (1'b1),
					.clocken3 (1'b1),
					.data_b ({8{1'b1}}),
					.eccstatus (),
					.q_a (),
					.rden_a (1'b1),
					.rden_b (1'b1),
					.wren_b (1'b0));
		defparam
			altsyncram_component.address_aclr_b = "NONE",
			altsyncram_component.address_reg_b = "CLOCK1",
			altsyncram_component.clock_enable_input_a = "BYPASS",
			altsyncram_component.clock_enable_input_b = "BYPASS",
			altsyncram_component.clock_enable_output_b = "BYPASS",
                    altsyncram_component.intended_device_family = "Cyclone IV E",
                    altsyncram_component.lpm_type = "altsyncram",
                    altsyncram_component.numwords_a = 16384,
                    altsyncram_component.numwords_b = 16384,
                    altsyncram_component.operation_mode = "DUAL_PORT",
                    altsyncram_component.outdata_aclr_b = "NONE",
                    altsyncram_component.outdata_reg_b = "CLOCK1",
                    altsyncram_component.power_up_uninitialized = "FALSE",
                    altsyncram_component.widthad_a = 14,
                    altsyncram_component.widthad_b = 14,
                    altsyncram_component.width_a = 8,
                    altsyncram_component.width_b = 8,
                    altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC "0"
// Retrieval info: PRIVATE: CLRq NUMERIC "0"
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRrren NUMERIC "0"
// Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRwren NUMERIC "0"
// Retrieval info: PRIVATE: Clock NUMERIC "1"
// Retrieval info: PRIVATE: Clock_A NUMERIC "0"
// Retrieval info: PRIVATE: Clock_B NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_B"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone IV E"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MEMSIZE NUMERIC "131072"
// Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING ""
// Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "2"
// Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "2"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "1"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
// Retrieval info: PRIVATE: REGrren NUMERIC "1"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
// Retrieval info: PRIVATE: VarWidth NUMERIC "0"
// Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "8"
// Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: ADDRESS_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK1"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone IV E"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16384"
// Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "16384"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "DUAL_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK1"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "14"
// Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "14"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_B NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL "data[7..0]"
// Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
// Retrieval info: USED_PORT: rdaddress 0 0 14 0 INPUT NODEFVAL "rdaddress[13..0]"
// Retrieval info: USED_PORT: rdclock 0 0 0 0 INPUT NODEFVAL "rdclock"
// Retrieval info: USED_PORT: wraddress 0 0 14 0 INPUT NODEFVAL "wraddress[13..0]"
// Retrieval info: USED_PORT: wrclock 0 0 0 0 INPUT VCC "wrclock"
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT GND "wren"
// Retrieval info: CONNECT: @address_a 0 0 14 0 wraddress 0 0 14 0
// Retrieval info: CONNECT: @address_b 0 0 14 0 rdaddress 0 0 14 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 wrclock 0 0 0 0
// Retrieval info: CONNECT: @clock1 0 0 0 0 rdclock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 8 0 data 0 0 8 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 8 0 @q_b 0 0 8 0
// Retrieval info: GEN_FILE: TYPE_NORMAL smallerRAM.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL smallerRAM.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL smallerRAM.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL smallerRAM.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL smallerRAM_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL smallerRAM_bb.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

Appendix B

```
//scopeCapture.c
#include "scopeGPIO.h"
#include <time.h>

//runs one full capture on the scope
int main(void){
        pioInit();
        spiInit(500000,0); //500kHz SPI
        pinMode(STARTERPIN, OUTPUT);

        digitalWrite(STARTERPIN,1); //use this pin to start overall conversation
        nanosleep((const struct timespec[]){{0,5000000L}},NULL); //wait 5ms to fill RAM
        digitalWrite(STARTERPIN,0); //no longer needed

        char output[16384]; //array to store 8-bit data
        short rawOutput; //raw output is 16-bit
        int i,j;
        for(i = 0; i<16384; ++i){
                rawOutput = spiSendReceive16(0xD000); //doesn't matter what we send
                rawOutput &= 0x00FF; //masking out non-valid bits, slightly unnecessary
                output[i] = (char)rawOutput;
        }

        for(j = 0; j<16384; ++j){
                printf("Content-type: text/html\n\n");
                printf("Value: %d\n",output[i]);
        }
        return 0;
}

<!-- scope.html, an attempt at doing scopeCapture on a webpage -->
<!DOCTYPE html>
<!DOCTYPE html>
<html>
<head>
    <title>Oscilloscope Control Page</title>
    <meta http-equiv="content-type" content="text-html;charset=utf-8">
```

```
</head>
<body>
	<form action="cgi-bin/scopeCapture" method="POST" target="tar1">
		<input type="submit" value="Click me">
	</form>
	<iframe id="tar1" name="tar1">
</body>
```

```
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

/////////////////////////////////////////////////////////////////////
// Constants
/////////////////////////////////////////////////////////////////////

// GPIO FSEL Types
#define INPUT  0
#define OUTPUT 1
#define ALT0   4
#define ALT1   5
#define ALT2   6
#define ALT3   7
#define ALT4   3
#define ALT5   2

// Physical addresses
#define BCM2836_PERI_BASE        0x3F000000
#define GPIO_BASE               (BCM2836_PERI_BASE + 0x200000)
#define SPI0_BASE               (BCM2836_PERI_BASE + 0x204000)
#define BLOCK_SIZE (4*1024)

#define STARTERPIN 21

volatile unsigned int *gpio; //pointer to base of gpio
volatile unsigned int *spi;  //pointer to base of spi registers

/////////////////////////////////////////////////////////////////////
// SPI Registers
/////////////////////////////////////////////////////////////////////

typedef struct
{
        unsigned CS                  :2;
        unsigned CPHA                :1;
        unsigned CPOL                :1;
        unsigned CLEAR               :2;
        unsigned CSPOL               :1;
        unsigned TA                  :1;
        unsigned DMAEN               :1;
        unsigned INTD                :1;
        unsigned INTR                :1;
        unsigned ADCS                :1;
        unsigned REN                 :1;
        unsigned LEN                 :1;
        unsigned LMONO               :1;
        unsigned TE_EN               :1;
        unsigned DONE                :1;
        unsigned RXD                 :1;
        unsigned TXD                 :1;
        unsigned RXR                 :1;
        unsigned RXF                 :1;
        unsigned CSPOL0   :1;
        unsigned CSPOL1   :1;
        unsigned CSPOL2   :1;
        unsigned DMA_LEN          :1;
        unsigned LEN_LONG         :1;
        unsigned                  :6;
}spi0csbits;
#define SPI0CSbits (* (volatile spi0csbits*) (spi + 0))
#define SPI0CS (* (volatile unsigned int *) (spi + 0))

#define SPI0FIFO (* (volatile unsigned int *) (spi + 1))
#define SPI0CLK (* (volatile unsigned int *) (spi + 2))
#define SPI0DLEN (* (volatile unsigned int *) (spi + 3))

void pioInit() {
        int  mem_fd;
        void *reg_map;

        // /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
        if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
            printf("can't open /dev/mem \n");
            exit(-1);
        }

        reg_map = mmap(
          NULL,          //Address at which to start local mapping (null means don't-care)
   BLOCK_SIZE,      //Size of mapped memory block
   PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
   MAP_SHARED,      // This program does not have exclusive access to this memory
   mem_fd,          // Map to /dev/mem
   GPIO_BASE);      // Offset to GPIO peripheral

        if (reg_map == MAP_FAILED) {
   printf("gpio mmap error %d\n", (int)reg_map);
   close(mem_fd);
   exit(-1);
  }

        gpio = (volatile unsigned *)reg_map;

        reg_map = mmap(
          NULL,          //Address at which to start local mapping (null means don't-care)
   BLOCK_SIZE,      //Size of mapped memory block
   PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
   MAP_SHARED,      // This program does not have exclusive access to this memory
   mem_fd,          // Map to /dev/mem
   SPI0_BASE);      // Offset to SPI peripheral

  if (reg_map == MAP_FAILED) {
   printf("spi mmap error %d\n", (int)reg_map);
   close(mem_fd);
   exit(-1);
  }

  spi = (volatile unsigned *)reg_map;

}
```

```
// assign a pin to a mode based on its function
void pinMode(int pin, int function)
{
        unsigned index, shift;

        // make sure the range of the pin to is [0, 53]
        if (pin > 53 || pin < 0) {
            printf("Pin out of range, must be 0-53 \n");
            exit(-1);
         }

        // find the position of the gpio pin
        index = pin / 10;
        shift = (pin % 10) * 3;

        // put function to the found positions
        gpio[index] &= ~(((~function) & 7) << shift);
        gpio[index] |= function << shift;
}

// write HIGH or LOW to the specified pin
void digitalWrite(int pin, int val)
{
        unsigned set, clr;

        // make sure range of the pin to is [0, 53]
        if (pin > 53 || pin < 0) {
            printf("Pin out of range, must be 0-53 \n");
            exit(-1);
         }

        // find the correct bits to write
        set = pin < 32 ? 7 : 8;
        clr = pin < 32 ? 10: 11;

        // set pin based on val
        if (val)
                gpio[set] = (0x1) << (pin % 32);
        else
                gpio[clr] = (0x1) << (pin % 32);
}

/////////////////////////////////////////////////////////////////////
// SPI Functions
/////////////////////////////////////////////////////////////////////

void spiInit(int freq, int settings) {
   //set GPIO 8 (CE), 9 (MISO), 10 (MOSI), 11 (SCLK) alt fxn 0 (SPI0)
   pinMode(8, ALT0);
   pinMode(9, ALT0);
   pinMode(10, ALT0);
   pinMode(11, ALT0);

   //Note: clock divisor will be rounded to the nearest power of 2
   SPI0CLK = 250000000/freq;   // set SPI clock to 250MHz / freq
   SPI0CS = settings;
   SPI0CSbits.TA = 1;          // turn SPI on with the "transfer active" bit
}

char spiSendReceive(char send){
   SPI0FIFO = send;            // send data to slave
   while(!SPI0CSbits.DONE);    // wait until SPI transmission complete
   return SPI0FIFO;            // return received data
}

short spiSendReceive16(short send) {
   short rec;
   SPI0CSbits.TA = 1;          // turn SPI on with the "transfer active" bit
   rec = spiSendReceive((send & 0xFF00) >> 8); // send data MSB first
   rec = (rec << 8) | spiSendReceive(send & 0xFF);
   SPI0CSbits.TA = 0;          // turn off SPI
   return rec;
}
```