Digital Storage Oscilloscope
6.111 Final Project Project Report

Rolando A. Gonzalez

December 2021

1 Brief

My 6.111 project goal was to implement a fully-fledged digital storage oscilloscope (DSO) on a Nexys-4 field-programmable-gate-array (FPGA), mainly using the on-board 1 mega-samples per second (MSPS) 12-bit analog-to-digital converter (XADC). The main function of this DSO should be to display sampled waveforms onto a VGA-compatible display accompanied by a graphical user interface that would reasonably allow the user to visually deduce a waveform's characteristics and toggle settings to appropriately scale an incoming waveform to fit in the oscilloscope display.

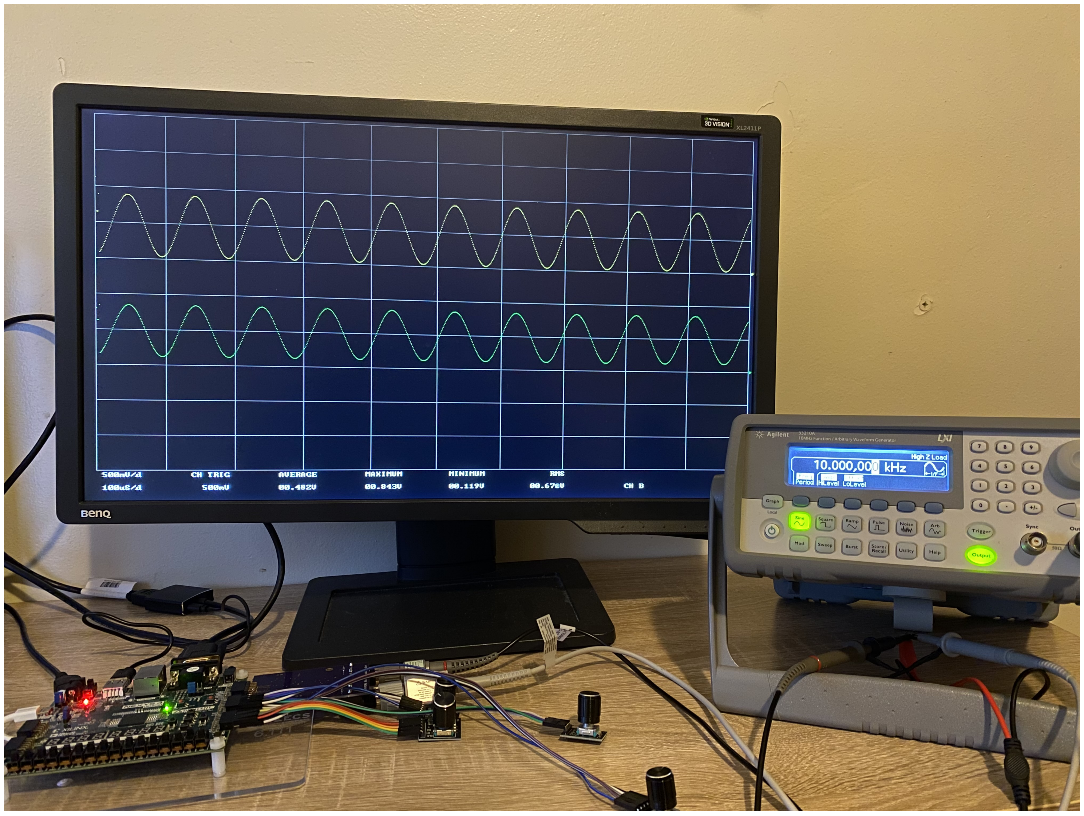


Figure 1: A picture of the FPGA digital storage oscilloscope in action, displaying a 10kHz 900mVpp wave

2 Design Overview

As depicted in the system-wide block diagram (figure 2), the main FPGA verilog code builds a pipeline that goes from the ADC, through a block that interprets collected samples and builds them into a waveform, along another stage that draws all graphical elements on the screen, and out via the VGA display port after being drawn into a cohesive picture.

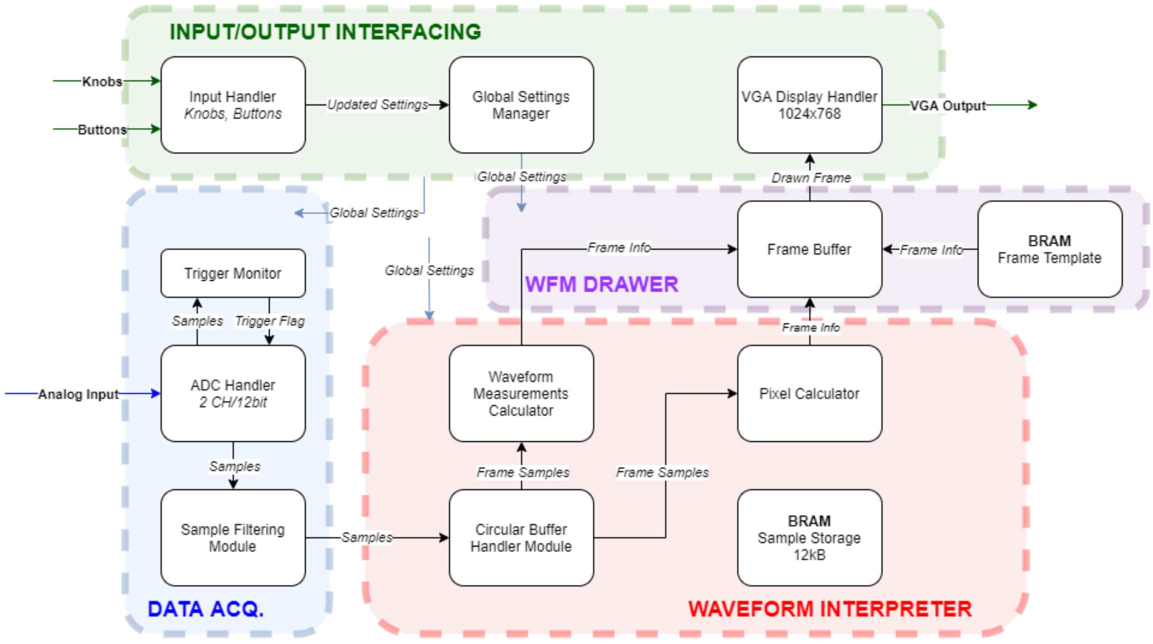


Figure 2: System wide block diagram

In theory, the design should be sample-count agnostic, taking it as a parameter and counting up to N cycles. The sampling stage should funnel samples at f_s[1] (possibly with delays in-between samples in order to elongate the time scale), and the interpreter stage will always store N/2 cycles until the DSO triggers[2] and then store another N/2 samples. The resultant math (to obtain measurements and scale samples to the display) is done over N samples– the results of which are fed into the combinational logic that draws the scope graphical interface. For a 1000x700 grid, N would be 1000. The entire pipeline should be created and designed for a single channel, in order to parallel the same design for an additional channel.

Two block random access memories (BRAMs) would exist in the system, both with enough space to hold a sample for each pixel column in the grid (1000). The data acquisition and waveform stages would each each use a clock faster than the one used in the waveform drawer/VGA display stages, as VGA needs a 65MHz clock to handle the horizontal and vertical syncing.

3 The Result

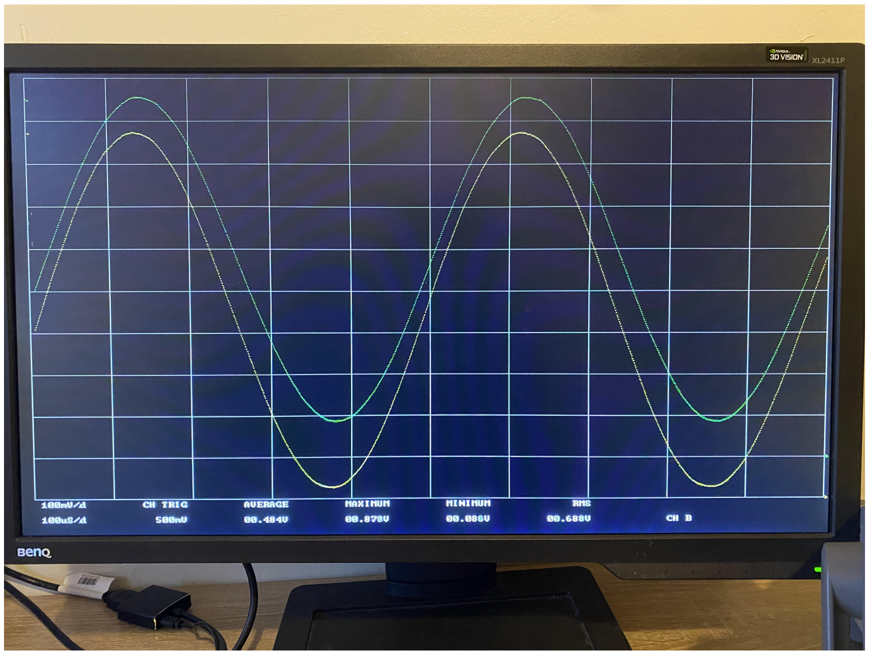
In the final product, the completed oscilloscope displays two waveforms in a 1000x700 grid on a VGA-capable display. A user can toggle multiple parameters to scale the incoming waveforms and can set the oscilloscope to trigger on a a rising or falling edge at a given voltage point for one of the active channels. Text below the waveform display grid relays information such as current scaling settings and measurements of the currently active channel. Additionally, a pmod-compatible[3] analog front end (AFE) was designed, fabricated, and assembled to directly connect passive oscilloscope probes to the FPGA. The board properly terminates two

[1]In this case, 1MSPS.
[2]The event in which a waveform crosses a specific voltage threshold.
[3]Pmod compatibility here referring to directly connecting to the 2x6pin connectors on the Nexys4 board.

1:10 probes, re-scales the signal to obtain a 1:1 or 1:50 ratio relative to the signal being probed, and offsets the signal to fit within the full 1.0V range of the XADC, all while being powered by the Nexys4 board's +3V3 rail.

Here's a summary of the oscilloscope's feature-set:

Final DSO Feature List		
Feature	Description	How to Toggle
2 Channel Sampling	Samples two waveforms at 1MSPS each.	Always active
Triggering Type	Determine whether to trigger on a rising/falling edge or disable triggering altogether.	Switches
Triggering Threshold	Set voltage level to trigger on (2mV increments).	Rotary Encoder
Voltage Scaling	Scale voltage per division between 17mV/div up to 1V/div.	Rotary Encoder Knob
Time Scaling	Scale voltage per division between 100uS/div up to 1S/div.	Rotary Encoder Knob
Vertical Offsetting	Move displayed waveforms vertically. Each channel has its own vertical offset.	Rotary Encoder Knob + Switch
External AFE Interfacing	Toggle relays on/off on external PMOD daughter board to add attenuation or AC couple incoming signals.	Switches
Oscilloscope GUI	Oscilloscope displays numeric global settings (timescale, voltage scale), relative GND for each signal, and numeric measurement values for one channel at a time.	Always active
Waveform Measurements	Oscilloscope records minimum, maximum, average, and root mean square of both channels. Only one is displayed at a time (toggle-able).	Switch
Sample Filtering	Pass all collected samples through a low pass filter to smooth out sudden transitions (works best with square waves).	Switch



```
100mV/d     CH TRIG     AVERAGE     MAXIMUM     MINIMUM     RMS
100uS/d     500mV       00.484V     00.878V     00.086V     00.688V     CH B
```

Figure 3: A closeup of the oscilloscope graphical interface, displaying a 900mVpp sine

4 Sub-module Specifics

This section is meant to describe the each sub-module's actions at a high-level, while chronicling insights obtained during development and some musings regarding potential improvements. To reiterate, the project was designed to scale up to multiple channels (2) with the same set of core modules, so some modules will be referred to as just working with a single channel/waveform. All relevant system verilog code can be found in the appendix.

4.1 Data Acquisition

This stage handles interfacing with the ADC and monitoring incoming samples in the event that the samples reflect that the incoming waveform has crossed a specific voltage threshold as it goes from low to high (rising edge) or high to low (falling edge). Additionally, the samples can be channeled through a low pass filter instead of being directly fed into the next stage.

4.1.1 XADC Handler

The XADC is set to simultaneous sampling mode, operating both ADCs in lock step to sample two external analog inputs and storing those samples in two status registers. This functionality is locked to specific sets of XADC channels, so auxiliary channels 3 and 11 (which are connected to the JXADC port) were used. Because the ADCs run in lock step, both status registers are updated at the exact same time.

XADC Characteristics	
Characteristic	Value
Sampling Speed	1M samples per second
Sample Resolution	12 bit
Voltage Step	0.244 mV
Startup Channel Select	Simultaneous Selection
Timing Mode	Continuous

A finite-state-machine (FSM) waits for the XADC to signal that new samples have been captured, and communicates with through the DRP interface[4] to unload the samples from both channels' sample storage registers. The FSM is always waiting for new samples, and responds to the always-active XADC signaling for new data stored in its registers.

In order to reduce the sampling frequency, we sample every other N samples to reduce sampling frequency f_s to $\frac{f_s}{N}$. One thing to note is that reducing the sampling frequency like so still means that at higher time divisions (e.g., 100uS to **1mS**), our Nyquist frequency does not stay at the original 500kHz when sampling at 1MSps. But I figured that looking at high frequency waveforms with a large time scale was not a relevant use case, so that method seemed fine. Here's the timescales I achieved with this module:

Timescale and Sampling Rate sampling every other N samples			
N	Sampling Rate ($\frac{f_s}{N}$)	Time Between Samples	Time/Div
0	1MSPS	1μS	100μS/div
10	100KSPS	10μS	1mS/div
100	10KSPS	100μS	10mS/div
1000	1KSPS	1mS	100mS/div
10000	100SPS	10mS	1S/div

4.1.2 Trigger Monitor

The trigger takes any 12bit value and monitors the incoming sample feed to see if the 12bit valued threshold is reached or crossed. It also detects whether a signal edge is rising or falling and only triggers on one kind of edge, in order to prevent aliasing on periodic signals that cross the threshold in both ways.

[4]See the *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter* user guide regarding exact DRP sequencing plus timings.

The trigger works by looking at the three most recent samples. It determines whether the signal is rising or falling by using the first and third most recent samples, and it determines whether the threshold is reached by looking at the second most recent sample, with some slack in the event that the signal does not exactly cross our threshold. The problem with this design is that it's not very noise immune, working just fine with our AFE but potentially failing to work if noise muddles the transition type or even causes sequences of rapid triggers as it crosses the threshold endlessly. A good solution might be to expand the buffer in order to average out a trend versus looking at just three consecutive samples.

4.1.3 Sampling Filtering Module

The optional filtering block is included to limit the bandwidth of the scope's current sampling rate to $\frac{f_s}{10}$ using a 30th order low pass filter created using Matlab's fir1 tool[5] using the following code: *round(fir1(30,.2)*1024)*.

For a Nyquist frequency of 500kHz, the resultant coefficients should produce a 30th order LPF with a cutoff of 100kHz. The coefficients were placed in the fir compiler Vivado IP, which created a module I could selectively funnel samples through for each channel.

4.2 Waveform Interpreter

The waveform interpreter receives filtered/unfiltered 12-bit samples and stores them in a buffer. Nothing too exciting if we're constantly updating the DSO display. In the event that we only want to display when the waveform triggers the DSO, then the buffer needs to store the most recent N/2 samples and then collect the rest (N/2) when triggered. A circular buffer is used here to solve this issue.

The module also calculates measurements of a given collection of samples: average, minimum, maximum, and the root-mean-square. There are probably many ways of doing this, but this isn't a super critical stage. Timing between frames is long enough and the sample count is small enough for there to be room for inefficiency[6]. This module captures all N samples and then reads them again to do math on that set of N samples.

4.2.1 Circular Buffer Handler Module

To store the N/2 most recent samples in an N sized buffer, the buffer is constantly written to and the start address of the most recent N/2 samples is maintained and updated. This module is connected to the trigger, and stores the next N/2 samples while keeping the same N/2 start address. This start address is passed on to the waveform drawer so it knows where to start reading the BRAM (and correctly display the samples in order).

4.2.2 Pixel Calculator and Waveform Measurements Calculator

After N samples have been collected, we read all N samples again and do the following action per measurement:

Measurements: How do they work?	
Measurement	Description
Maximum/Minimum	Start with two 12 bit registers set to 0 (maximum) or 4095 (minimum) and update them if the next sample is larger or smaller respectively.
Average	Add all samples in a $\log_2(1000 * 2^{12})$-bit sized buffer and divide over 1000 at the end.
Root Mean Square	Add the square of all given samples in a $\log_2(1000 * 2^{12} * 2^{12})$-bit sized buffer and divide over 1000 at the end and then take the square root.
Pixel Calculation	Divide a given sample over a specific scaling value to maintain a specific voltage/division ratio and calculate its position on the grid.

[5]`https://www.mathworks.com/help/signal/ref/fir1.html`

[6]Not that there needs to be any. Sampling and doing the math in parallel is the immediate better option, but that was not implemented here.

Vivado IPs for dividers and square root calculators were used in order to complete said functions over multiple cycles. I'm sharing below what scalar values I used, and what voltage steps and voltage per division (a division being 70 pixels) they result in.

Pixel Scaling Values			
Scaling Value	Voltage Step / Pixel	Resultant Voltage/Div	Displayed Voltage/Div
1	.244mV	17mV/div	17mV/div
3	.732mV	51mV/div	50mV/div
6	1.46mV	103mV/div	100mV/div
12	2.93mV	205mV/div	200mV/div
17	4.15mV	290mV/div	300mV/div
29	7.08mV	496mV/div	500mV/div
44	10.7mV	752mV/div	750mV/div
59	14.4mV	1.08V/div	1.00V/div

The pixel scaling is close enough to where on the screen, the resultant waveform seems pretty close to the advertised values. But again, they're approximations due to the limitation of working with a 700-pixel tall space.

4.3 Waveform Drawer

The oscilloscope graphical interface consists of a 1000x700 grid where the waveforms from channel A (yellow) and channel B (green) are measured, text reflecting certain settings/measurements, and markers noting the relative GND of the currently measured waveforms. The reason for the 1000x700 dimensions are twofold– the grid houses 10 divisions in either direction (vertical/horizontal) and required to be multiples of 10, while needing to leave extra space in order be able to place text and markers outside of the grid frame instead of cluttering the grid itself.

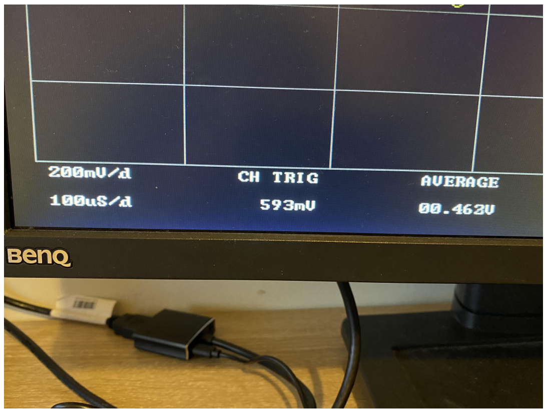
This module outputs the required signals used to drive a VGA display. Such a module was shamelessly borrowed from lab 3[7] of this class. Ideally, a higher resolution could've been used, and the module should be able to naturally scale by tweaking all the parameterized modules accordingly to fit more samples and scale appropriately to a larger resolution. That was not implemented nor tested, though.

The grid is entirely combinational logic, drawing a gray pixel when reaching equally spaced intervals in the x direction and the y direction. The module polls from the global channel offset setting and the pixel location BRAM to draw the GND marker and the pixel for each channel respectively. Note that the other modules are signaled for when this module is polling the BRAMs in order to not collide read and write cycles and potentially write two waveforms at the same time, which looks like the ghost of another waveform is appearing as you trigger on one waveform[8].

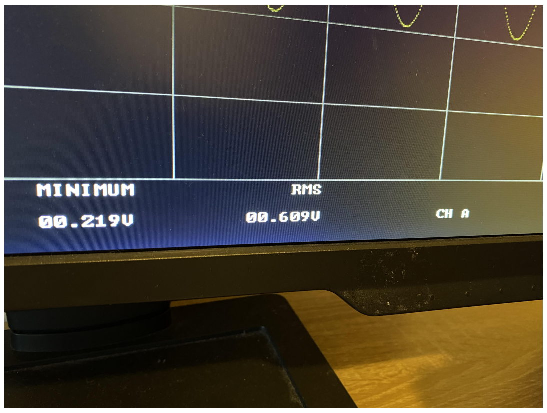
For text, I wrote a module that converts 12-bit values into ASCII characters and plugs them into the alphanumerical ROM display modules I got the code from here[9]. I combine these into a module that takes in info to convert and display alongside the display module's hcount/vcount signals to draw the expected text on the screen.

I stuck to whatever functional rudimentary grid I had originally drawn, but this definitely could use a face-lift. I prioritized getting a functional display/graphical interface up and running, but adding dash markers between divisions and thinking more about the UI/UX aspect of this screen would definitely be a nice-to-have-done.

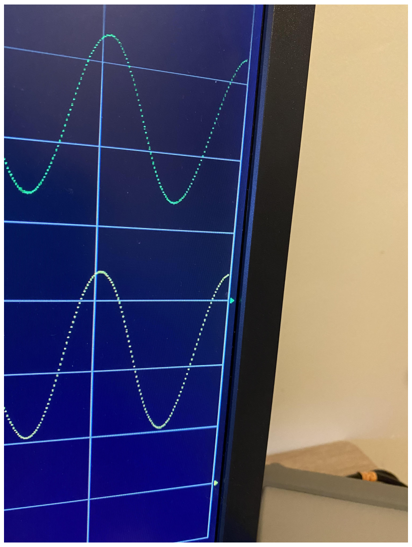
[7] `https://6111.io/F21/labs/lab03`
[8] This was a huge headache, but this sort of inter-module communication helps a lot!
[9] `https://web.mit.edu/6.111/volume2/www/f2021/cstringdisp.v`



(a) Time scale, voltage scale and measurements



(b) More measurements plus channel being triggered on and measured



(c) Reference GND for channels A/B located on the right

Figure 4: Oscilloscope GUI Additions

4.4 Input/Output Interfacing

The input handler manages inputs by the four rotary encoders: one for the horizontal time scaling, another for the vertical voltage scaling, an additional one for the trigger setting, and a final one for the cursor movement and placement. This stage is pretty straightforward.

4.4.1 Input Handling

Keyes KY-040 rotary encoders (figure 5) were used to cycle through available settings on the DSO. Their power terminals (+/GND) were hooked up to the +3V3 VCC and GND terminals on the JA and JB PMOD ports. All other signal I/O (encoder pins A,B and the button) were mapped as inputs and debounced.

The demonstration code found in the Digilent reference manual[10] for Digilent's own PMOD rotary encoder board was moderately tweaked and used to interface with the rotary encoders. Knob transitions, denoted by a specific sequence of A/B transitions, increase or decrease registers containing settings affecting the sampling pipeline.

[10]`https://digilent.com/reference/pmod/pmodenc/start`

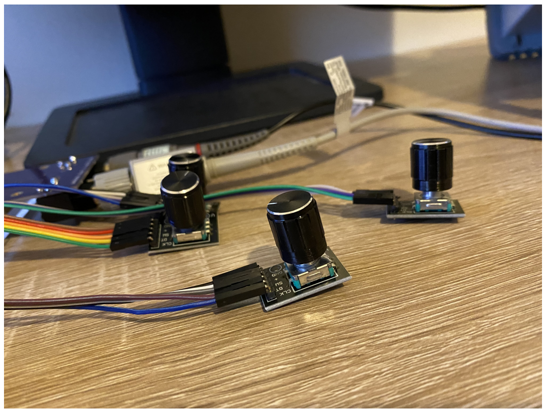


Figure 5: Four rotary encoders were connected to the FPGA with jumper wires

4.4.2 Global Settings

The interface module holds settings used throughout other modules, and manages the following global settings:

Oscilloscope Global Settings	
Option	Description
Time Scale	Scales the time between (horizontal) grid divisions. Goes from 100uS to 1S per division.
Voltage Scaling	Scales the voltage between (vertical) grid divisions. Goes from 17mV to 1V per division.
Channel Measure Select	Sets either channel A or channel B as the active channel. The active channel is triggered on and has its measurements displayed on the bottom.
Channel Vertical Offset	Offsets active channel signal ”ground” vertically between top and horizontal edges of the screen.
Trigger Threshold	Sets trigger threshold for current active channel in mV. Goes from 0mV to 998mV in 2mV increments.
Trigger Edge Toggle	Switches between trigger event: rising or falling edge.
Manual Trigger Enable	Forces an active trigger. Oscilloscope displays waveforms as fast as it interprets them.
Filter Enable	Enables low pass filtering in sampling stage to smooth out fast edges.
AC Coupling Enable	Enables AC coupling for both channels through the external AFE.
Attenuation Enable	Enables a 1:50 reduction for each channel through the external AFE.

4.5 The Pmod Analog Front End

The analog front end is an external daughter printed circuit board assembly (PCBA) that connects to the JXADC port on the Nexys4 board. It contains three stages: one to terminate a 10:1 passive probe properly with a $1M\Omega$ resistor with a toggle-able AC coupling path, an additional toggle-able 1:50 reduction stage, and a final 1:10 gain output stage that additionally provides a +0V5 offset to center a signal between the XADCs voltage reference (+1V0) and GND. See appendix for schematic and design file references.

The toggle-able features (AC coupling, passive reduction) were designed by using a +3V3 relay and laying out two parallel signal paths between which the relay could choose to route an incoming analog signal through. The relay would be turned on using a simple +3V3 logic N-channel MOSFET with a conservatively high enough power rating (an SOT323 2N7002).

For the gain stage (figure 8), a total gain of 10 was split between two op-amp circuits– a non-inverting topology yielding a gain of 5, and a differential amplifying topology yielding a gain of 2. Originally, the

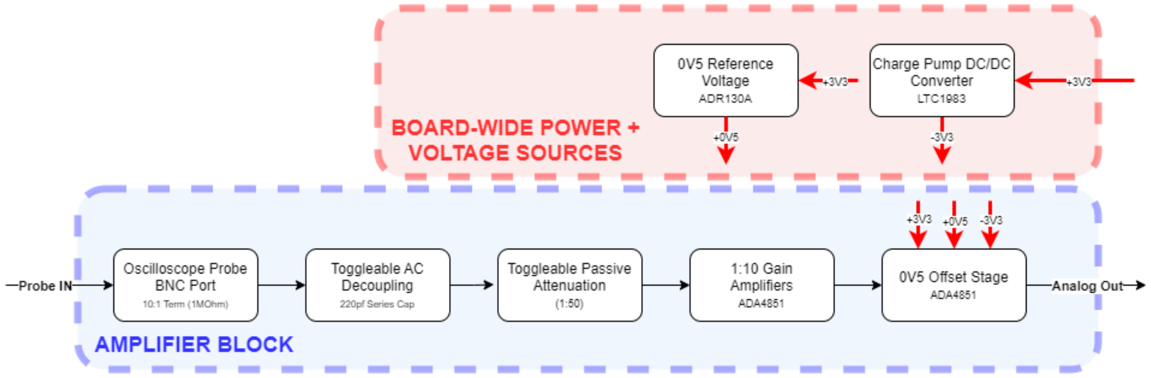


Figure 6: Analog front end block diagram

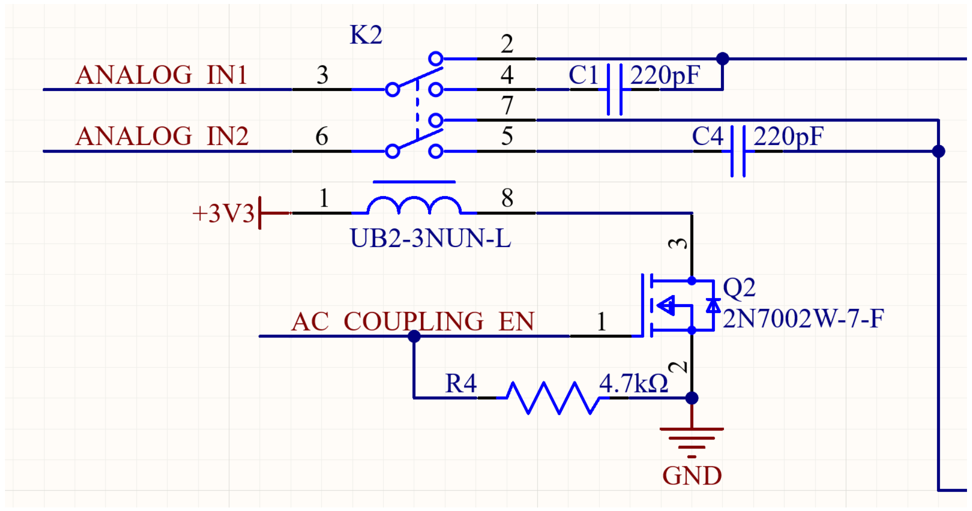


Figure 7: Selectable AC coupling through a +3V3 relay (a high pass filter with $f_c \approx 723Hz$)

gain was cascaded between two stages to increase the bandwidth of each stage[11] but such a concern should really only be apparent at higher frequencies beyond the scope of this scope. Both stages use small resistors to ignore the ADA4851's relatively high input bias current. The second stage uses a differential amplifying topology, which results in the following transfer function:

$$R_A = R_{11} = R_9$$

$$R_B = R_{13} = R_8$$

$$V_{OUT} = 0.5 + (\frac{R_A}{R_B})V_{IN}$$

The ADA4851[12] was chosen due to its relatively low power consumption, low cost, high bandwidth, and low voltage offset (0.6mV). To improve the performance of the system, swap out the part for a higher cost part that can provide a more aggressive rail-to-rail output[13], and one with a smaller input bias current to prevent unintentionally offsetting our signal.

Due to the input bias current of this op-amp, the buffer staged right after the input termination actually creates a large input offset voltage. The ADA4851 has a typical input bias current around $1\mu A$, which across

[11]The ADA4851 has a bandwidth of 130MHz at unity gain, for reference.

[12]`https://www.analog.com/media/en/technical-documentation/data-sheets/ADA4851-1_4851-2_4851-4.pdf`

[13]The ADA4851's output swings to within 60 mV of either rail.

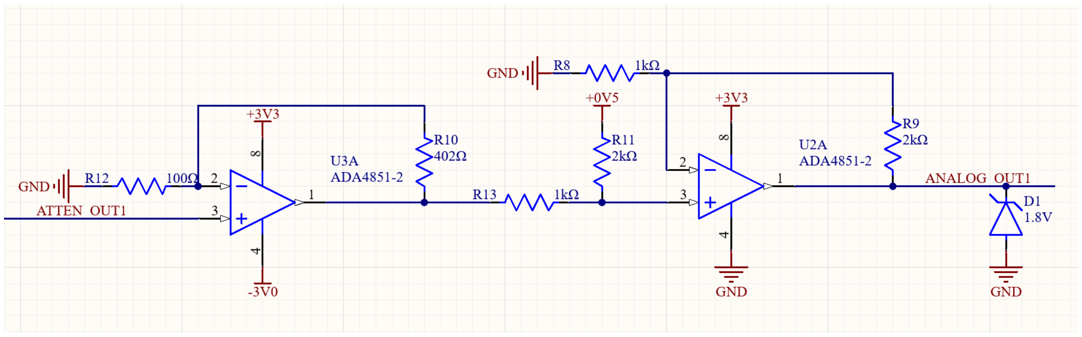


Figure 8: Two op-amp gain stage, the differential amplifier topology at the end inserts a +0V5 offset

the $1M\Omega$ resistor yields a whopping 1V! A quick fix was to swap out U1 for an ADA4891, a pin-to-pin replacement with a bias current of about 50pA at worst[14].

A charge pump converter (LTC1983) was added to enable the AC-coupling of signals by allowing two of the op-amps to swing below GND (down to -3V0). A +0V5 voltage reference output (ADA130) was also included to provide the final offset voltage reference for the differential amplifying topology at the end. To reiterate, a +0V5 offset would center a signal between the XADC's V_{REF} and GND.

Since the AFE adds a +0V5 offset to any incoming probed signal, readings are distorted by said offset. The intention was for the FPGA to respond to this in software and compensate measurements accordingly, but this was never implemented. In the 1:50 attenuation case, selecting and activating this option would increase the voltage scaling and all the measured values by a factor of 50. In theory, this would increase the V_{IN} width this module could operate with– but it would still be rated for about $\tilde{6}$Vpp (the op-amp supplies) versus the theoretical max of 50Vpp.

See the following table (referencing figure 9) for the board's top 2x6-pin pinout:

Pmod AFE Pinout		
AFE Pin	JXADC Pin	Pin Function
Pin 1	VCC	Nexys4 VCC (+3V3)
Pin 2	GND	Nexys4 GND
Pin 3	A18	XADC Channel 2 negative terminal/GND
Pin 4	A15	Not Connected
Pin 5	A16	AC coupling EN signal
Pin 6	A14	XADC Channel 1 negative terminal/GND
Pin 7	VCC	Nexys4 VCC (+3V3)
Pin 8	GND	Nexys4 GND
Pin 9	A13	XADC Channel 2 positive terminal
Pin 10	B16	Channel 2 attenuation EN signal
Pin 11	A15	Channel 1 attenuation EN signal
Pin 12	A13	XADC Channel 1 positive terminal

One thing to note about the pinout is that it doesn't line up with pmod port as-is. It should optimally be mirrored across the y-axis (relative to figure 9), else the board is improperly connected. This is a minor layout problem that can be solved with proper rerouting to better match the Nexys4 pmod pinout. The current pmod AFE pinout, while wrong, can be "fixed" by soldering the right angle connector to protrude from the bottom side of the board like in figure 10.

Fully loaded, the board takes up around 160mA[15] of current, which falls below the 1A pmod spec as the Nexys4 on-board power converter can supply enough current. The low quiescent current also gives this board plenty of headroom in the event that the op-amps saturate at +3V3 for extended periods of time.

[14]The tradeoff? It has an input offset voltage of around 10mV at worst. :(

[15]Most of it comes from turning on a relay (46.7ma/ea) as the op-amps take in about 19mA of quiescent current, plus whatever inefficiency the negative charge pump presents.

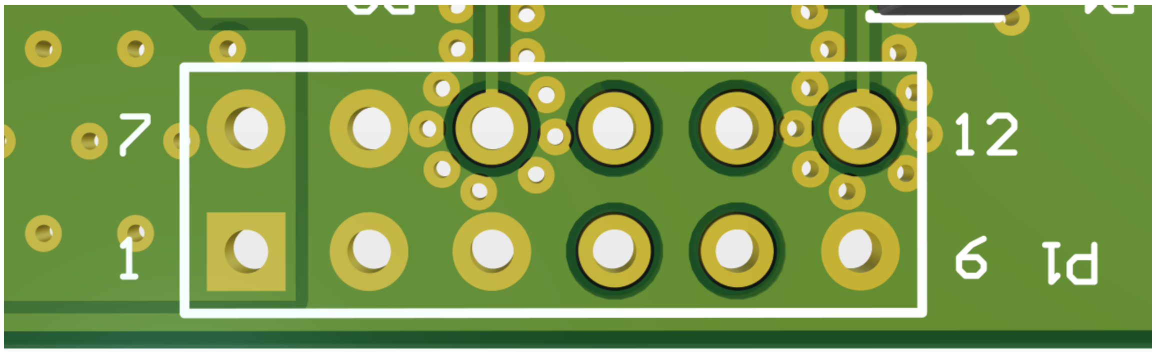


Figure 9: Analog front end pmod connector pinout top side view (right angle connector connects here)

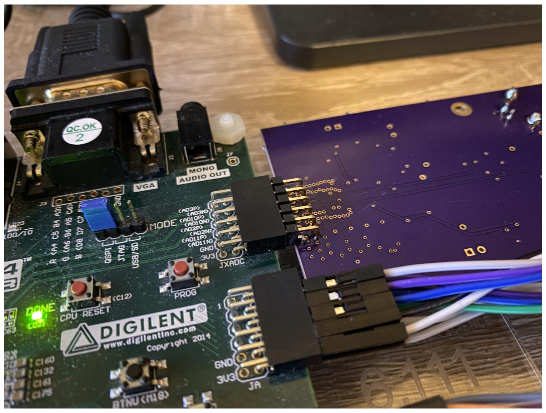


Figure 10: Pmod AFE connected to the Nexys4 board

5 Appendix

5.1 System Verilog Code

5.1.1 Top Level Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: rolandog
//
// Revision:
// Revision 0.01 - File Created
// top_level.sv
//
// connects all modules together
//
// misc logic:
//  -stages samples between modules to avoid writing to a bram while another module reads from it
//  -given global channel select variable, pipelines appropriate measurements to text drawer
//
//////////////////////////////////////////////////////////////////////////////////


module top_level(   input wire clk_100mhz,
                    input wire adc_ch3n,
                    input wire adc_ch3p,
                    input wire adc_ch11n,
                    input wire adc_ch11p,
                    input wire[15:0] sw,
                    input wire encoder_A1, encoder_B1, encoder_btn1,
                    input wire encoder_A2, encoder_B2, encoder_btn2,
                    input wire encoder_A3, encoder_B3, encoder_btn3,
                    input wire encoder_A4, encoder_B4, encoder_btn4,
                    output logic ac_coupling_enable,
                    output logic attenuation_enable1,
                    output logic attenuation_enable2,
                    output logic[3:0] vga_r,
                    output logic[3:0] vga_b,
                    output logic[3:0] vga_g,
                    output logic vga_hs,
                    output logic vga_vs,
                    input wire btnc, btnu, btnd, btnr, btnl,
                    output logic [15:0] led
    );
    //global variables
    logic global_reset;
    assign global_reset = btnc;
    logic manual_trigger; //0 = no effect, 1 = trigger manually activated
    logic attenuated_input; //0 = no attenuation, 1 = pmod afe 1:50 attn
    logic filter_enable; //0 = no filter, 1 = filter
    logic channel_select; //0 = CHA, 1 = CHB
    logic edge_toggle; //0 = rising, 1 = falling


    logic clk_65mhz;
    // create 65mhz system clock, happens to match 1024 x 768 XVGA timing
    clk_wiz_65 clkdivider(.clk_in(clk_100mhz), .clk_out(clk_65mhz));



    //CHANNEL A
    logic [11:0] sampleA;
    logic sample_readyA;
    logic triggerA;
    logic draw_waveformA;
    logic [9:0] start_addressA;
    logic [11:0] averageA;
    logic [11:0] maximumA;
    logic [11:0] minimumA;
    logic [11:0] rmsA;


    //CHANNEL B
    logic [11:0] sampleB;
    logic sample_readyB;
    logic triggerB;
    logic draw_waveformB;
    logic [9:0] start_addressB;
    logic [11:0] averageB;
    logic [11:0] maximumB;
    logic [11:0] minimumB;
    logic [11:0] rmsB;

    //settings
    logic [2:0] sample_speed;
    logic [3:0] voltage_scaling;


    //bram blocks
    logic [9:0] sample_bramA_addrA;
    logic [9:0] addr_A2;
    logic [11:0] bramA_in;
    logic [11:0] bramA_out;
    logic bramA_write;

    logic [9:0] sample_bramA_addrB;
    logic [9:0] addr_B2;
    logic [11:0] bramB_in;
    logic [11:0] bramB_out;
    logic bramB_write;

    logic [9:0] hcountA;
    logic [9:0] hcountB;
    logic [9:0] hcount_mem;
    logic [9:0] vcount_pixel_locationA1;
    logic [9:0] vcount_pixel_locationA2;
    logic [9:0] vcount_pixel_locationB1;
    logic [9:0] vcount_pixel_locationB2;

    logic run_mode;
    logic waveform_trigger_A;
    assign waveform_trigger_A = (triggerA || manual_trigger || run_mode);
    logic waveform_trigger_B;
    assign waveform_trigger_B = (triggerB || manual_trigger || run_mode);

```

```

    //text
    logic [15:0] text_bram_address_write;
    logic text_bram_pixel_write;
    logic [15:0] text_bram_address_read;
    logic text_bram_pixel_read;
    logic bram_text_write;

    //measurement
    logic [11:0] average;
    logic [11:0] maximum;
    logic [11:0] minimum;
    logic [11:0] rms;

    logic [9:0] channel_a_offset;
    logic [9:0] channel_b_offset;
    logic [11:0] threshold;

    //drawing
    logic drawing_finish;

    //DATA ACQUISITION STAGE
    data_acquisition data_acq(.clock_in(clk_100mhz), .reset_in(global_reset),
                              .sample_speed_setting(sample_speed),
                              .sampleA_out(sampleA), .sampleB_out(sampleB),
                              .vauxn3(adc_ch3n),.vauxp3(adc_ch3p),
                              .vauxn11(adc_ch11n),.vauxp11(adc_ch11p),
                              .sample_readyA_out(sample_readyA),
                              .sample_readyB_out(sample_readyB),
                              .triggerA_out(triggerA), .triggerB_out(triggerB),
                              .trigger_threshold(threshold),
                              .channel_select(channel_select),
                              .edge_toggle(edge_toggle),
                              .filter_enable(filter_enable));

    //reset interpreter on timescale change
    logic [3:0] current_sample_speed;
    logic waveform_interpreter_reset;

    always_ff @(posedge clk_100mhz) begin
        current_sample_speed <= sample_speed;
        waveform_interpreter_reset <= (sample_speed == current_sample_speed) ? global_reset : 1'b1;
    end

    //MATH, DRAWING, AND INTERPRETATION STAGE

    //buffer pixel locations only when screen is not displaying waveforms
    logic [9:0] vcount_pixel_location_bufferB;
    logic [9:0] vcount_pixel_location_bufferA;
    logic [9:0] addr_buffer;
    logic [9:0] buffer_write_en;
    localparam OSCOPE_BUFFER_SIZE = 1000;


    always_ff @(posedge clk_100mhz) begin
        if ((draw_waveformA || draw_waveformB)) begin //drawing_finish
            buffer_write_en <= 1'b1;
            addr_buffer <= 10'b0;
        end else begin
            buffer_write_en <= (addr_buffer == OSCOPE_BUFFER_SIZE - 1) ? 1'b0 : buffer_write_en;
            addr_buffer <= (addr_buffer == OSCOPE_BUFFER_SIZE - 1) ? 10'b0 : addr_buffer + 1;
        end
    end

    waveform_interpreter wfm_intA(.clock_in(clk_100mhz), .reset_in(waveform_interpreter_reset),
                          .sample(sampleA),
                          .sample_ready(sample_readyA),
                          .trigger(waveform_trigger_A),
                          .draw_waveform(draw_waveformA),
                          .average(averageA), .minimum(minimumA), .maximum(maximumA),.rms(rmsA),
                          .addr(sample_bramA_addrA),.addr2(addr_A2),
                          .bram_in(bramA_in), .bram_out(bramA_out),
                          .bram_write(bramA_write),
                          .channel_offset(channel_a_offset),
                          .voltage_scaling_setting(voltage_scaling),
                          .hcount(hcountA), .vcount_pixel_location(vcount_pixel_locationA1));

    blk_mem_channel_a sample_bramA (.addra(sample_bramA_addrA), .clka(clk_100mhz),
                                   .dina(bramA_in), .wea(bramA_write),
                                   .addrb(addr_A2), .clkb(clk_100mhz),
                                   .doutb(bramA_out), .web(0), .enb(1));

    blk_mem_channel_a_gui oscope_bramA_buffer (.addra(hcountA), .clka(clk_100mhz),
                                              .dina(vcount_pixel_locationA1), .wea(1), .ena(1),
                                              .addrb(addr_buffer), .clkb(clk_100mhz),
                                              .doutb(vcount_pixel_location_bufferA), .enb(1));

    blk_mem_channel_a_gui oscope_bramA (.addra(addr_buffer), .clka(clk_100mhz),
                                       .dina(vcount_pixel_location_bufferA),
                                       .wea(buffer_write_en), .ena(1),
                                       .addrb(hcount_mem), .clkb(clk_65mhz),
                                       .doutb(vcount_pixel_locationA2), .enb(1));

    waveform_interpreter wfm_intB(.clock_in(clk_100mhz), .reset_in(global_reset),
                         .sample(sampleB),
                         .sample_ready(sample_readyB),
                         .trigger(waveform_trigger_B),
                         .draw_waveform(draw_waveformB),
                         .average(averageB), .minimum(minimumB),
                         .maximum(maximumB),.rms(rmsB),
                         .addr(sample_bramA_addrB), .addr2(addr_B2),
                         .bram_in(bramB_in), .bram_out(bramB_out),
                         .bram_write(bramB_write),
                         .channel_offset(channel_b_offset),
                         .voltage_scaling_setting(voltage_scaling),
                         .hcount(hcountB), .vcount_pixel_location(vcount_pixel_locationB1));

    blk_mem_channel_b sample_bramB (.addra(sample_bramA_addrB), .clka(clk_100mhz),
                                   .dina(bramB_in), .wea(bramB_write),
                                   .addrb(addr_B2), .clkb(clk_100mhz),
                                   .doutb(bramB_out), .web(0), .enb(1));

    blk_mem_channel_b_gui oscope_bramB_buffer (.addra(hcountB), .clka(clk_100mhz),
                                              .dina(vcount_pixel_locationB1), .wea(1), .ena(1),
                                              .addrb(addr_buffer), .clkb(clk_100mhz),
                                              .doutb(vcount_pixel_location_bufferB), .enb(1));

    blk_mem_channel_b_gui oscope_bramB (.addra(addr_buffer), .clka(clk_100mhz),
                                       .dina(vcount_pixel_location_bufferB),
                                       .wea(buffer_write_en), .ena(1),
                                       .addrb(hcount_mem), .clkb(clk_65mhz),
                                       .doutb(vcount_pixel_locationB2), .enb(1));

    waveform_drawer wfm_draw(.clock_in(clk_65mhz), .reset_in(global_reset), .threshold(threshold),
                          .draw_waveformA(draw_waveformA), .draw_waveformB(draw_waveformB),
                          .vga_r(vga_r), .vga_b(vga_b),
                          .vga_g(vga_g), .vga_hs(vga_hs), .vga_vs(vga_vs),
                          .vcount_pixel_locationA(vcount_pixel_locationA2),
                          .vcount_pixel_locationB(vcount_pixel_locationB2),
                          .channel_offset_A(channel_a_offset),
                          .channel_offset_B(channel_b_offset),
                          .hcount_mem(hcount_mem),
                          .text_bram_address_read(text_bram_address_read),
                          .text_bram_pixel_read(text_bram_pixel_read),
                          .drawing_finish(drawing_finish));

    display_text_handler text_handler (.clock_in(clk_100mhz), .reset_in(global_reset),
                                      .attenuated_in(attenuated_input),
                                      .text_address(text_bram_address_write),
                                      .average_in(average), .maximum_in(maximum),
                                      .minimum_in(minimum), .rms_in(rms),
                                      .sample_speed_setting(sample_speed),
                                      .voltage_scaling_setting(voltage_scaling),
                                      .pixel_out(text_bram_pixel_write),
                                      .bram_text_write(bram_text_write),
                                      .threshold_in(threshold), .channel_select(channel_select));

    bram_screen_text oscope_gui_text_bram (.addra(text_bram_address_write), .clka(clk_100mhz),
                                          .dina(text_bram_pixel_write), .wea(bram_text_write), .ena(1),
                                          .addrb(text_bram_address_read), .clkb(clk_65mhz),
                                          .doutb(text_bram_pixel_read), .enb(1));


    //INTERFACING

    always_comb begin
        average = (channel_select) ? averageB : averageA;
        maximum = (channel_select) ? maximumB : maximumA;
        minimum = (channel_select) ? minimumB : minimumA;
        rms = (channel_select) ? rmsB : rmsA;
    end

    interfacing_handler interface_IO (.clock_in(clk_100mhz), .reset_in(global_reset),
                                      .btnu(btnu), .sw_in(sw),
                                      .encoder_A1(encoder_A1), .encoder_B1(encoder_B1),
                                      .encoder_btn1(encoder_btn1),
                                      .encoder_A2(encoder_A2), .encoder_B2(encoder_B2),
                                      .encoder_btn2(encoder_btn2),
                                      .encoder_A3(encoder_A3), .encoder_B3(encoder_B3),
                                      .encoder_btn3(encoder_btn3),
                                      .encoder_A4(encoder_A4), .encoder_B4(encoder_B4),
                                      .encoder_btn4(encoder_btn4),
                                      .sample_speed_setting(sample_speed),
                                      .voltage_scaling_setting(voltage_scaling),
                                      .channel_a_offset(channel_a_offset),
                                      .channel_b_offset(channel_b_offset),
                                      .led_in(led), .threshold(threshold),
                                      .run_mode(run_mode), .edge_toggle(edge_toggle),
                                      .ac_coupling_enable(ac_coupling_enable),
                                      .attenuation_enable1(attenuation_enable1),
                                      .attenuation_enable2(attenuation_enable2),
                                      .channel_measure_select(channel_select),
                                      .manual_trigger(manual_trigger),.filter_enable(filter_enable));


endmodule //top_level
```

5.1.2 Data Acquisition Stage Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: rolandog
//
// Revision:
// Revision 0.01 - File Created
// data_acquistion.sv
//
//  interfaces with internal XADC module
//      -simultaneous dual channel sampling mode (both channels at 1MS/sec)
//      -can delay incoming samples (by skipping N-1 samples) to obtain a sampling rate of fs/N
//
//  employs a low pass filter to smooth out rough edges
//      -fir filter with a Wn of .1
//      -works okay for straight rising edges
//
//  contains a trigger monitor for each channel
//      -3 sample buffer, checks if middle sample meets threshold and other two samples denote trend
//          (rising, falling)
//
//  see interfacing.sv for settings applicable to this module
//
//////////////////////////////////////////////////////////////////////////////////


module data_acquisition( input wire clock_in,
                    input wire reset_in,
                    input wire [2:0] sample_speed_setting,
                    input wire vauxn3, vauxp3, vauxn11, vauxp11,
                    input wire filter_enable,
                    input wire channel_select,
                    input wire edge_toggle,
                    input wire [11:0] trigger_threshold,
                    output logic [11:0] sampleA_out,
                    output logic [11:0] sampleB_out,
                    output logic sample_readyA_out,
                    output logic sample_readyB_out,
                    output logic triggerA_out,
                    output logic triggerB_out
                    );

    //variables
    logic [15:0] sample_counter;
    logic [15:0] sample_count;
    logic sample_count_reached;

    logic [15:0] adc_data;
    logic [6:0] adc_address;
    logic samples_ready;
    logic data_enable;
    logic data_ready;

    logic [11:0] sampled_adc_data;

    logic [11:0] sampleA;
    logic [11:0] sampleA_temp;
    logic [11:0] sampleB;
    logic [11:0] sampleB_temp;
    logic sample_readyA;
    logic sample_readyB;
    logic triggerA;
    logic triggerB;

    localparam CHANNEL_A = 7'h13;
    localparam CHANNEL_B = 7'h1B;

    //XADC
    xadc_wiz input_adc (.dclk_in(clock_in), .daddr_in(adc_address), .reset_in(0),
                        .vauxn3(vauxn3),.vauxp3(vauxp3),
                        .vauxn11(vauxn11),.vauxp11(vauxp11),
                        .vp_in(0),.vn_in(0),
                        .di_in(16'b0),
                        .eoc_out(samples_ready),
                        .do_out(adc_data),.drdy_out(data_ready),
                        .den_in(data_enable), .dwe_in(0));


    //sample count LUT
    always_comb begin
        case(sample_speed_setting)
            3'b000: sample_count = 0;  //1MSPS
            3'b001: sample_count = 10; //100kSPS
            3'b010: sample_count = 100; //10kSPS
            3'b011: sample_count = 1000; //1kSPS
            3'b100: sample_count = 10000; //100SPS
            default: sample_count = 0; //1MSPS
        endcase
    end

    assign sampled_adc_data = adc_data[15:4]; //offset binary

    //sampling
    assign sample_count_reached = (sample_counter == sample_count);

    //passing on samples state machine
    //waiting -> channelA -> channelB -> waiting

    //all states
    enum {WAITING, READ_CHANNEL_A, CHANNEL_A_SAMPLING,
          READ_CHANNEL_B, CHANNEL_B_SAMPLING, SEND} state, next;

    //output logic
    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            sampleA <= 12'b0;
            sampleB <= 12'b0;
            sampleA_temp <= 12'b0;
            sampleB_temp <= 12'b0;
            sample_readyA <= 1'b0;
            sample_readyB <= 1'b0;
            data_enable <= 1'b0;
            sample_counter <= 16'b0;
        end else begin
            case (next)
            WAITING: begin
                    //wait for samples
                    sampleA <= 12'b0;
                    sampleB <= 12'b0;
                    sample_readyA <= 1'b0;
                    sample_readyB <= 1'b0;
                    adc_address <= CHANNEL_A;
                  end
            READ_CHANNEL_A: begin
                    //ping DRP with channel A data, toggle DEN high for one CLK cycle
                    data_enable <= (state == WAITING) ? 1'b1 : 1'b0;
                  end
            CHANNEL_A_SAMPLING: begin
                    //read CHA
                    sampleA_temp <= sampled_adc_data;
                  end
            READ_CHANNEL_B: begin
                    //ping DRP with channel B data, toggle DEN high for one CLK cycle
                    data_enable <= (state == CHANNEL_A_SAMPLING) ? 1'b1 : 1'b0;
                    adc_address <= CHANNEL_B;
                end
            CHANNEL_B_SAMPLING: begin
                    //read CHB
                    sampleB_temp <= sampled_adc_data;
                end
            SEND: begin
                    //send CHA, CHB
                    sampleA <= sample_count_reached ? sampleA_temp : 0;
                    sampleB <= sample_count_reached ? sampleB_temp : 0;
                    sample_readyA <= sample_count_reached ? 1 : 0;
                    sample_readyB <= sample_count_reached ? 1 : 0;
                    sample_counter <= sample_count_reached ? 0 : sample_counter + 1;
                end
            default: begin
                    //huh, well that wasn't supposed to happen -- wait for samples
                    sampleA <= 12'b0;
                    sampleB <= 12'b0;
                    sample_readyA <= 1'b0;
                    sample_readyB <= 1'b0;
                end
            endcase
        end
    end

    //state transition and logic
    always_comb begin
        case (state)
        WAITING: next = (samples_ready) ? READ_CHANNEL_A : WAITING;
        READ_CHANNEL_A: next = (data_ready) ? CHANNEL_A_SAMPLING : READ_CHANNEL_A;
        CHANNEL_A_SAMPLING: next = READ_CHANNEL_B;
        READ_CHANNEL_B: next = (data_ready) ? CHANNEL_B_SAMPLING : READ_CHANNEL_B;
        CHANNEL_B_SAMPLING: next = SEND;
        SEND: next = WAITING;
        default: next = WAITING;
        endcase
    end

    always_ff @(posedge clock_in) begin
        state <= reset_in ? WAITING : next;
    end

    //channel select logic
    always_comb begin
        triggerA_out = (channel_select) ? triggerB : triggerA;
        triggerB_out = (channel_select) ? triggerB : triggerA;
    end

    //trigger modules
    trigger_monitor t1 (.clock_in(clock_in), .reset_in(reset_in),
                        .threshold(trigger_threshold), .sample(sampleA_out),
                        .edge_setting(edge_toggle),
                        .new_sample(sample_readyA_out), .triggered(triggerA)); //channel A
    trigger_monitor t2 (.clock_in(clock_in), .reset_in(reset_in),
                        .threshold(trigger_threshold), .sample(sampleB_out),
                        .edge_setting(edge_toggle),
                        .new_sample(sample_readyB_out), .triggered(triggerB)); //channel B

    //fir filters -- Wn = .100
    logic [11:0] sampleA_filtered;
    logic sample_readyA_filtered;
    fir_compiler_input fir_filterA (.aclk(clock_in),
                                    .s_axis_data_tvalid(sample_readyA),
                                    .s_axis_data_tdata(sampleA),
                                    .m_axis_data_tdata({4'b0, sampleA_filtered}),
                                    .m_axis_data_tvalid(sample_readyA_filtered));

    logic [11:0] sampleB_filtered;
    logic sample_readyB_filtered;
    fir_compiler_input fir_filterB (.aclk(clock_in),
                                    .s_axis_data_tvalid(sample_readyB),
                                    .s_axis_data_tdata(sampleB),
                                    .m_axis_data_tdata({4'b0, sampleB_filtered}),
                                    .m_axis_data_tvalid(sample_readyB_filtered));
    //filter logic-- when filter is enabled, pipeline samples through filter
    always_comb begin
        sampleA_out = (filter_enable) ? sampleA_filtered << 2 : sampleA;
        sampleB_out = (filter_enable) ? sampleB_filtered << 2 : sampleB;
        sample_readyA_out = (filter_enable) ? sample_readyA_filtered : sample_readyA;
        sample_readyB_out = (filter_enable) ? sample_readyB_filtered : sample_readyB;
    end

endmodule

//monitors incoming samples, outputs a high on triggered to signal a low-to-high transition
//  sequentially cycles through 3 samples,
//  checks if first and third sample denote a rising/falling trend and
//  whether the second sample hits the threshold value within SLACK
module trigger_monitor( input wire clock_in,
                    input wire reset_in,
                    input wire [11:0] threshold,
                    input wire [11:0] sample,
                    input wire new_sample,
                    input wire edge_setting,
                    output logic triggered
    );

    localparam SLACK = 5;
    logic [11:0] first_sample;
    logic [11:0] second_sample;
    logic [11:0] third_sample;

    logic edge_detect; //rising = 0, falling = 1
    assign edge_type = edge_setting ? (third_sample > first_sample) : (third_sample < first_sample);

    //trigger if trend is increasing/decreasing, sample near threshold
    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            triggered <= 0;
            first_sample <= 11'b0;
            second_sample <= 11'b0;
            third_sample <= 11'b0;
        end else begin
           triggered <= (second_sample <= threshold + SLACK &&
                         second_sample >= threshold - SLACK) ? edge_type : 0; //increasing
           first_sample <= (new_sample) ? sample : first_sample;
           second_sample <= (new_sample) ? first_sample : second_sample;
           third_sample <= (new_sample) ? second_sample : third_sample;
        end
    end




endmodule //trigger_monitor
```

5.1.3 Waveform Interpretation Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: rolandog
//
// Revision:
// Revision 0.01 - File Created
// waveform_interpreter.sv
//
// measures samples from a channel, stores them into circular BRAM buffer the size of the screen
//   (1000 x 12 bit samples)
// captures N/2 samples, waits for trigger, captures N/2 samples for a total of N samples
//
// when N samples are reached:
//  calculates pixel placement on display for each sample and
//  passes it into a buffer going to the oscilloscope drawer module (1000 x 10 bit placements)
//  goes over all 1000 samples and infers average, min/max and rms values
//
//////////////////////////////////////////////////////////////////////////////////


module waveform_interpreter (input wire clock_in,
                    input wire reset_in,
                    input wire [11:0] sample,
                    input wire sample_ready,
                    input wire trigger,
                    input wire [3:0] voltage_scaling_setting,
                    output logic [9:0] addr,
                    output logic [9:0] addr2,
                    output logic [11:0] bram_in,
                    input wire [11:0] bram_out,
                    input wire [9:0] channel_offset,
                    output logic bram_write,
                    output logic [9:0] hcount,
                    output logic [9:0] vcount_pixel_location,
                    output logic draw_waveform,
                    output logic [11:0] average,
                    output logic [11:0] minimum,
                    output logic [11:0] maximum,
                    output logic [11:0] rms);
    //bram variables
    logic [9:0] start_address_out; //passes on BRAM start address
    logic signal_math_handler; //signals to start measuring across N samples
    //passing on to bram
    waveform_manager manager (.clock_in(clock_in), .reset_in(reset_in),
                              .sample(sample), .sample_ready(sample_ready),
                              .trigger(trigger), .bram_in(bram_in),
                              .bram_write(bram_write),
                              .conversion_done(draw_waveform),
                              .draw_waveform(signal_math_handler),
                              .addr_out(addr), .start_addr_out(start_address_out));

    waveform_math_handler handler (.clock_in(clock_in), .reset_in(reset_in),
                                   .begin_conversion(signal_math_handler),
                                   .start_addr(start_address_out),
                                   .bram_out(bram_out),
                                   .voltage_scaling(voltage_scaling_setting),
                                   .conversion_done(draw_waveform),
                                   .addr_out(addr2), .channel_offset(channel_offset),
                                   .average_out(average),
                                   .maximum_out(maximum),.minimum_out(minimum),
                                   .rms_out(rms),
                                   .hcount_out(hcount),
                                   .vcount_pixel_location_out(vcount_pixel_location));

endmodule //waveform_interpreter

//handles the math and interfacing with a bram
module waveform_manager(input wire clock_in,
                    input wire reset_in,
                    input wire [11:0] sample,
                    input wire sample_ready,
                    input wire trigger,
                    input wire conversion_done,
                    output logic [11:0] bram_in,
                    output logic bram_write,
                    output logic draw_waveform,
                    output logic [9:0] addr_out,
                    output logic [9:0] start_addr_out
                    );
    //update these depending on screen size
    localparam MAX_ADDRESS = 999;
    localparam HALF_SAMPLE_COUNT = 500;
    localparam SAMPLE_COUNT = 1000;

    //measurement variables
    logic [9:0] sample_count;
    logic [9:0] start_addr;

    //measurement fsm
    //keep track of N/2 samples, when trigger collect all N and update
    //all states
    enum {START, SAMPLING, UPDATE} state, next;

    //events
    logic done;
    assign done = (sample_count == SAMPLE_COUNT);


    //output logic
    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            start_addr <= 10'b0;
            addr_out <= 10'b0;
            draw_waveform <= 10'b0;
            start_addr_out <= 10'b0;
            sample_count <= 10'b0;
        end else begin
            case (next)
            START: begin
                    //begin
                    draw_waveform <= 1'b0;
                    sample_count <= 10'b0;
                  end
            SAMPLING: begin
                    bram_write <= 1'b1;
                    //update only if new sample
                    if (sample_ready) begin
                        //new sample
                        bram_in <= sample;
                        //sample count handling
                        if (trigger && (sample_count == HALF_SAMPLE_COUNT)) begin
                            sample_count <= sample_count + 10'b1; //add one
                        end else if ((sample_count == HALF_SAMPLE_COUNT)) begin
                            //keep same sample count
                            sample_count <= sample_count ;
                            //add to start address
                            start_addr <= (start_addr == MAX_ADDRESS) ? 10'b0 : start_addr + 10'b1;
                        end else begin
                            sample_count <= sample_count + 10'b1; //add one
                        end

                        //update address for next sample
                        addr_out <= (addr_out == MAX_ADDRESS) ? 10'b0 : addr_out + 10'b1;
                    end else begin
                        //do nothing
                    end
                  end
            UPDATE: begin
                    //pass along signaling that we're done
                    bram_write <= 1'b0;
                    start_addr_out <= start_addr;
                    draw_waveform <= 1'b1;
                  end
            default: begin
                    //huh, well that wasn't supposed to happen -- wait for samples
                    start_addr <= 10'b0;
                    addr_out <= 10'b0;
                    draw_waveform <= 1'b0;
                  end
            endcase
        end
    end

    //state transition and logic
    always_comb begin
        case (state)
        START: next = SAMPLING;
        SAMPLING: next = (done) ? UPDATE : SAMPLING;
        UPDATE: next = (conversion_done) ? START : UPDATE;
        default: next = START;
        endcase
    end

    always_ff @(posedge clock_in) begin
        state <= reset_in ? START : next;
    end
endmodule //waveform_manager

module waveform_math_handler(input wire clock_in,
                    input wire reset_in,
                    input wire begin_conversion,
                    input wire [9:0] start_addr,
                    input wire [11:0] bram_out,
                    input wire [3:0] voltage_scaling,
                    input wire [9:0] channel_offset,
                    output logic conversion_done,
                    output logic [9:0] addr_out,
                    output logic [11:0] average_out,
                    output logic [11:0] maximum_out,
                    output logic [11:0] minimum_out,
                    output logic [11:0] rms_out,
                    output logic [9:0] hcount_out,
                    output logic [9:0] vcount_pixel_location_out);

    //measurement variables
    logic [9:0] sample_count;
    assign hcount_out = sample_count;
    logic [21:0] average;  //stores a max value of 900*2^12
    logic [11:0] maximum;
    logic [11:0] minimum;
    logic [33:0] rms;     //stores a max value of 900*2^12*2^12
    logic [24:0] bram_squared;     //stores a max value of 2^12*2^12

    //update these depending on screen size
    localparam MAX_ADDRESS = 999;
    localparam SAMPLE_COUNT = 1000;


    //voltage scaling options
    logic [7:0] scaler;
    always_comb begin
        case (voltage_scaling)
            3'b000: scaler = 1; //17mV/div
            3'b001: scaler = 3; //50mV/div
            3'b010: scaler = 6; //100mV/div
            3'b011: scaler = 12; //200mV/div
            3'b100: scaler = 17; //300mV/div
            3'b101: scaler = 29; //500mV/div
            3'b110: scaler = 44; //750mV/div
            3'b111: scaler = 59; //1V/div
            default: scaler = 0;
        endcase
    end

    //////////DIVIDER, SQRT MODULES
    //divider for vcount
    logic divider_data_valid;
    logic divider_done;
    logic [11:0] dividend;
    logic [11:0] quotient;
    logic [23:0] divider_data_out;
    assign quotient = divider_data_out[19:8];

    vcount_divisor div (.aclk(clock_in), .s_axis_divisor_tdata(scaler),
                        .s_axis_divisor_tvalid(divider_data_valid),
                        .s_axis_dividend_tdata({4'b0, dividend}),
                        .s_axis_dividend_tvalid(divider_data_valid),
                        .m_axis_dout_tdata(divider_data_out), .m_axis_dout_tvalid(divider_done));

    //divider for average, rms
    logic [9:0] average_divisor;
    assign average_divisor = SAMPLE_COUNT;

    logic avg_div_data_valid;
    logic avg_div_done;
    logic [23:0] avg_div;
    logic [21:0] avg_div_result;
    logic [39:0] avg_div_data_out;
    assign avg_div_result = avg_div_data_out[37:16];

    average_divisor_mean div_avg (.aclk(clock_in), .s_axis_divisor_tdata(average_divisor),
                                  .s_axis_divisor_tvalid(avg_div_data_valid),
                                  .s_axis_dividend_tdata(avg_div),
                                  .s_axis_dividend_tvalid(avg_div_data_valid),
                                  .m_axis_dout_tdata(avg_div_data_out),
                                  .m_axis_dout_tvalid(avg_div_done));

    logic rms_div_data_valid;
    logic rms_div_done;
    logic [33:0] rms_div;
    logic [33:0] rms_div_result;
    logic [55:0] rms_div_data_out;
    assign rms_div_result = rms_div_data_out[49:16];

    rms_divisor_mean div_rms (.aclk(clock_in),
                              .s_axis_divisor_tdata(average_divisor),
                              .s_axis_divisor_tvalid(rms_div_data_valid),
                              .s_axis_dividend_tdata({6'b0, rms_div}),
                              .s_axis_dividend_tvalid(rms_div_data_valid),
                              .m_axis_dout_tdata(rms_div_data_out),
                              .m_axis_dout_tvalid(rms_div_done));

    logic rms_sqrt_done;
    logic [11:0] rms_sqrt_result;
    logic [17:0] rms_sqrt_data_out;
    assign rms_sqrt_result = rms_sqrt_data_out[11:0];

    cordic_sqrt sqrt_rms (.aclk(clock_in),
                          .s_axis_cartesian_tdata(rms_div_result),
                          .s_axis_cartesian_tvalid(rms_div_done),
                          .m_axis_dout_tdata(rms_sqrt_data_out),
                          .m_axis_dout_tvalid(rms_sqrt_done));

    logic math_done;
    assign math_done = (avg_div_done && rms_div_done && rms_sqrt_done);


    //measurement fsm
    //keep track of N/2 samples, when trigger collect all N and update
    //all states
    enum {WAITING, TAKE_IN_ADDRESS, MEASURING, DIVIDING, SENDING, UPDATING, MATH, DONE} state, next;

    //output logic
    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            average <= 22'b0;
            maximum <= 12'b0;
            minimum <= 12'b1111_1111_1111;
            rms <= 34'b0;
            average_out <= 22'b0;
            maximum_out <= 12'b0;
            minimum_out <= 12'b0;
            rms_out <= 11'b0;
            bram_squared <= 25'b0;
            addr_out <= start_addr;
        end else begin
            case (next)
            WAITING: begin
                    //wait for all samples to be taken in
                    conversion_done <= 1'b0;
                    average <= 22'b0;
                    maximum <= 12'b0;
                    minimum <= 12'b1111_1111_1111;
                    rms <= 34'b0;
                    sample_count <= 10'b0;
                  end
            TAKE_IN_ADDRESS:begin
                    addr_out <= start_addr;
                  end
            MEASURING: begin
                    //new sample
                    average <= average + bram_out;
                    maximum <= (bram_out > maximum) ? bram_out : maximum;
                    minimum <= (bram_out < minimum) ? bram_out : minimum;
                    bram_squared <= (bram_out * bram_out);

                    //vcount division
                    dividend <= bram_out;

                  end
            DIVIDING: begin
                    //new sample
                    rms <= bram_squared + rms;
                    divider_data_valid <= 1'b1;
                  end
            SENDING: begin
                    //send vcount location to BRAM
                    divider_data_valid <= 1'b0;
                    //check if pixel is out of bounds
                    if (quotient >= 8 + channel_offset) begin
                        //out of bounds
                        vcount_pixel_location_out <= 0;
                    end else begin
                        vcount_pixel_location_out <= channel_offset - quotient;
                    end
                  end
            UPDATING: begin
                    //update address for next sample
                    addr_out <= (addr_out == MAX_ADDRESS) ? 10'b0 : addr_out + 10'b1;
                    sample_count <= sample_count + 10'b1;
                  end
            MATH: begin
                    avg_div <= average;
                    avg_div_data_valid <= 1'b1;
                    rms_div <= rms;
                    rms_div_data_valid <= 1'b1;
                    //update addresses if not triggered
                  end
            DONE: begin
                     conversion_done <= 1'b1;
                     average_out <= avg_div_result;
                     maximum_out <= maximum;
                     minimum_out <= minimum;
                     rms_out <= rms_sqrt_result;
```

```

                      //turn off math modules
                      avg_div_data_valid <= 1'b0;
                      rms_div_data_valid <= 1'b0;
                   end
             default: begin
                     //huh, well that wasn't supposed to happen -- wait for samples
                     average <= 22'b0;
                     maximum <= 12'b0;
                     minimum <= 12'b0;
                     rms <= 12'b0;
                     addr_out <= start_addr;
                   end
             endcase
         end
     end

     //state transition and logic
     always_comb begin
         case (state)
         WAITING: next = (begin_conversion) ? TAKE_IN_ADDRESS : WAITING;
         TAKE_IN_ADDRESS: next = MEASURING;
         MEASURING: next = DIVIDING;
         DIVIDING: next = (divider_done) ? SENDING : DIVIDING;
         SENDING: next = UPDATING;
         UPDATING: next = (sample_count == SAMPLE_COUNT) ? MATH : MEASURING;
         MATH: next = (math_done) ? DONE : MATH;
         DONE: next = WAITING;
         default: next = WAITING;
         endcase
     end

     always_ff @(posedge clock_in) begin
         state <= reset_in ? WAITING : next;
     end

endmodule//waveform_math_handler
```

5.1.4 Waveform Drawing Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: rolandog
//
// Revision:
// Revision 0.01 - File Created
// waveform_drawer.sv
//  connects to an external vga display using xvga module by gim
//  contains combinational logic to draw grid, oscilloscope gui elements and waveforms
//    NOTE: waveform pixel locations are preemptively calculated
//  reads waveform/text elements from external brams as frame updates
//
//  grid is 1000x700
//  mockup:
//   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
//  |                                                    |
//  |\/\/waveformB\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/|
//  |                                                    |gndB
//  |\/\/waveformA\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/|
//  |                                                    |gndA
//  |                                                    |
//  |_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _|
//  time    |                 | measurement desc |
//  voltage | trig_threshold  | measurements     |    current active channel
//////////////////////////////////////////////////////////////////////////////////


module waveform_drawer(input wire clock_in,
            input wire reset_in,
            input wire draw_waveformA,
            input wire draw_waveformB,
            input wire [9:0] vcount_pixel_locationA,
            input wire [9:0] vcount_pixel_locationB,
            input wire [9:0] channel_offset_A,
            input wire [9:0] channel_offset_B,
            input wire text_bram_pixel_read,
            input wire [11:0] threshold,
            output logic [15:0] text_bram_address_read,
            output logic[3:0] vga_r,
            output logic[3:0] vga_b,
            output logic[3:0] vga_g,
            output logic vga_hs,
            output logic vga_vs,
            output logic [9:0] hcount_mem,
            output logic drawing_finish
    );

    //variables
    logic [10:0] hcount;    // pixel on current line
    logic [9:0] vcount;     // line number
    logic hsync, vsync, blank; //control signals for vga
    logic [11:0] pixel;
    logic [11:0] rgb;

    //signal when no longer reading from oscilloscope bram
    always_ff @(posedge clock_in) begin
        drawing_finish <= (vcount == 710) ? 1'b1 : 1'b0;
    end

    //xvga video signals module
    xvga xvga_manager (.vclock_in(clock_in), .hcount_out(hcount), .vcount_out(vcount),
                       .vsync_out(vsync), .hsync_out(hsync), .blank_out(blank));


    logic [11:0] gui_pixel;
    oscilloscope_gui_gen o_gui(.vclock_in(clock_in),
                               .hcount(hcount), .vcount(vcount),
                               .vcount_pixel_locationA(vcount_pixel_locationA),
                               .vcount_pixel_locationB(vcount_pixel_locationB),
                               .hcount_mem(hcount_mem),
                               .text_bram_address_read(text_bram_address_read),
                               .text_bram_pixel_read(text_bram_pixel_read),
                               .channel_offset_A(channel_offset_A),
                               .channel_offset_B(channel_offset_B),
                               .pixel_out(gui_pixel), .threshold(threshold));

    //buffer, mux for screen
    logic b,hs,vs;
    always_ff @(posedge clock_in) begin
        // default: gui
        hs <= hsync;
        vs <= vsync;
        b <= blank;
        rgb <= gui_pixel;
    end

    // the following lines are required for the Nexys4 VGA circuit - do not change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;


endmodule //waveform_drawer

//draw a 1000 by 700 grid
module oscilloscope_gui_gen(input wire vclock_in,
            input wire [10:0] hcount,    // pixel number on current line
            input wire [9:0] vcount,     // line number
            input wire [9:0] vcount_pixel_locationA,
            input wire [9:0] vcount_pixel_locationB,
            input wire text_bram_pixel_read,
            input wire [11:0] threshold,
            input wire [9:0] channel_offset_A,
            input wire [9:0] channel_offset_B,
            output logic [15:0] text_bram_address_read,
            output logic [9:0] hcount_mem,
            output logic [11:0] pixel_out
        );

    //color pixels
    localparam WHITE = 12'b1111_1111_1111;
    localparam GRAY = 12'b1000_1000_1000;
    localparam YELLOW = 12'b1111_1111_0000;
    localparam GREEN = 12'b0000_1111_0000;
    localparam RED = 12'b1111_0000_0000;

    //grid
    logic [11:0] grid_pixel;
    logic [11:0] text_pixel;
    localparam HORIZONTAL_STEP = 100;
    localparam HORIZONTAL_OFFSET = 10;
    localparam VERTICAL_STEP = 70;
    localparam VERTICAL_OFFSET = 8;

    assign grid_pixel = ((hcount == 0*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 1*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 2*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 3*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 4*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 5*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 6*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 7*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 8*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 9*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         hcount == 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET ||
                         vcount == 0*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 1*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 2*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 3*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 4*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 5*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 6*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 7*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 8*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 9*VERTICAL_STEP + VERTICAL_OFFSET ||
                         vcount == 10*VERTICAL_STEP + VERTICAL_OFFSET) &&
                         !(hcount < 0*HORIZONTAL_STEP + HORIZONTAL_OFFSET) &&
                         !(hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET) &&
                         !(vcount < 0*VERTICAL_STEP + VERTICAL_OFFSET) &&
                         !(vcount > 10*VERTICAL_STEP + VERTICAL_OFFSET))? GRAY : 0;

    logic [11:0] channel_gnd_pixelA;
    logic [11:0] channel_gnd_pixelB;

    //draws A gnd arrow
    assign channel_gnd_pixelA = (((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 5) &&
                                 (vcount == channel_offset_A)) ||
                                 ((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 4) &&
                                 (vcount == channel_offset_A + 1 || vcount == channel_offset_A - 1 )) ||
                                 ((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 3) &&
                                 (vcount == channel_offset_A + 2 || vcount == channel_offset_A - 2 )) ||
                                 ((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 2) &&
                                 (vcount == channel_offset_A + 3 || vcount == channel_offset_A - 3 ))) ?
                                 YELLOW : 0;
    //draws B gnd arrow
    assign channel_gnd_pixelB = (((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 5) &&
                                 (vcount == channel_offset_B))||
                                 ((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                  hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 4) &&
                                 (vcount == channel_offset_B + 1 || vcount ==  channel_offset_B - 1 )) ||
                                 ((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 3) &&
                                 (vcount == channel_offset_B + 2 || vcount == channel_offset_B - 2 )) ||
                                 ((hcount > 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1 &&
                                 hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 2) &&
                                 (vcount == channel_offset_B + 3 || vcount == channel_offset_B - 3 ))) ?
                                 GREEN : 0;

    //oscilloscope channel
    logic [9:0] hcount_mem_count;

    always_ff @(posedge vclock_in) begin
        if (hcount == 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET + 1) begin
            hcount_mem_count <= 0;
        end else begin
            hcount_mem <= hcount_mem_count;
            hcount_mem_count <= (hcount >= 5 &&
                                 hcount <= 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET) ?
                                 hcount_mem_count + 1: 0;
        end
    end

    logic [11:0] channelA_pixel;
    logic [11:0] channelB_pixel;
    logic [9:0] vcountA;
    logic [9:0] vcountB;

    always_ff @(posedge vclock_in) begin
        vcountA <= vcount_pixel_locationA;
        vcountB <= vcount_pixel_locationB;

    end

    assign channelA_pixel = (vcountA == vcount && hcount > HORIZONTAL_OFFSET &&
                             hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET) ? YELLOW : 0;
    assign channelB_pixel = (vcountB == vcount && hcount > HORIZONTAL_OFFSET &&
                             hcount < 10*HORIZONTAL_STEP + HORIZONTAL_OFFSET) ? GREEN : 0;

    //text on screen


    localparam TEXT_OFFSET = 710;
    logic [16:0] next_text_bram_address;
    logic [10:0] next_hcount;
    assign next_hcount = (hcount == 1023) ? 11'b0 : hcount + 1;
    logic [9:0] next_vcount;
    assign next_vcount = (hcount == 1023) ? vcount + 1 : vcount;


    //address, pixel
    always_ff @(posedge vclock_in) begin
        if (vcount >= TEXT_OFFSET &&  hcount <= 1024) begin
            text_pixel <= (text_bram_pixel_read) ? WHITE : 0;
            text_bram_address_read <= ((next_vcount - TEXT_OFFSET) * 1024) + next_hcount;
        end else begin
            text_pixel <= 0;
            text_bram_address_read <= ((next_vcount - TEXT_OFFSET) * 1024) ;
        end
    end
    assign pixel_out = (channelA_pixel || channelB_pixel) ? ( channelA_pixel | channelB_pixel) :
                       text_pixel | grid_pixel | channel_gnd_pixelA | channel_gnd_pixelB; //draw

endmodule //oscilloscope_gui_gen

////////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                                ---- HORIZONTAL -----     ------VERTICAL -----
//                                Active                    Active
//                    Freq        Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz    25.175      640     16    96   48       480    11    2   31
//   800x600, 60Hz    40.000      800     40   128   88       600     1    4   23
//   1024x768, 60Hz   65.000      1024    24   136  160       768     3    6   29
//   1280x1024, 60Hz  108.00      1280    48   112  248       768     1    3   38
//   1280x720p 60Hz   75.25       1280    72    80  216       720     3    5   30
//   1920x1080 60Hz   148.5       1920    88    44  148      1080     4    5   36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////////////////

module xvga(input wire vclock_in,
            output logic [10:0] hcount_out,    // pixel number on current line
            output logic [9:0] vcount_out,     // line number
            output logic vsync_out, hsync_out,
            output logic blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   logic hblank,vblank;
   logic hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   logic vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP - 1)); // 805

   // sync and blanking
   logic next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

`default_nettype wire //xvga
```

5.1.5 Interfacing IO Code

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: rolandog
//
// Revision:
// Revision 0.01 - File Created
// interfacing.sv
//
// handles all I/O tied to the FPGA, decoding buttons and handling rotary encoders
// additionally controls logic interfacing to external pmod afe
//
// settings:
//  - sample_speed_setting: determines timescale, incremented/decremented by knob
//      0:  100uS/div (lowest possible)
//      1:    1mS/div
//      2:   10mS/div
//      3:  100mS/div
//      4:    1S/div (highest realized setting)
//  - voltage_scaling_setting : determines voltage scaling, incremented/decremented by knob
//      0:  17mV/div
//      1:  50mV/div
//      2: 100mV/div
//      3: 200mV/div
//      4: 300mV/div
//      5: 500mV/div
//      6: 750mV/div
//      7: 1.00V/div
//  - channel_X_offset : determines channel X vertical offset
//  - threshold : sets trigger threshold for current active channel
//  - manual_trigger / run_mode : select whether to disable triggering (auto)
//  - filter_enable : enables FIR (100kHz BW) filtering for both channels
//  - channel_measure_select : selects which channel outputs to display on screen,
//                             also selects which channel is triggered on
//      0: A
//      1: B
//  - ac_coupling_enable : enables external pmod afe ac coupling routing
//  - attenuation_enable1/2 : enables external pmod afe 1:50 attenuation for channels A/B
//  - edge_toggle : determines which type of edge to trigger on
//      0: rising
//      1: falling
```

```
//
/////////////////////////////////////////////////////////////////////////////////


module interfacing_handler(input wire clock_in,
                    input wire reset_in,
                    input wire btnu,
                    input wire[15:0] sw_in,
                    output logic [15:0] led_in,
                    input wire encoder_A1, encoder_B1, encoder_btn1,
                    input wire encoder_A2, encoder_B2, encoder_btn2,
                    input wire encoder_A3, encoder_B3, encoder_btn3,
                    input wire encoder_A4, encoder_B4, encoder_btn4,
                    output logic [2:0] sample_speed_setting,
                    output logic [3:0] voltage_scaling_setting,
                    output logic [9:0] channel_a_offset,
                    output logic [9:0] channel_b_offset,
                    output logic [11:0] threshold,
                    output logic manual_trigger,
                    output logic filter_enable,
                    output logic channel_measure_select,
                    output logic run_mode,
                    output logic ac_coupling_enable,
                    output logic attenuation_enable1,
                    output logic attenuation_enable2,
                    output logic edge_toggle
    );

    //variables
    logic encoder_button1;
    logic encoder_button2;
    logic encoder_button3;
    logic encoder_button4;

    logic enca1;
    logic encb1;
    logic enca2;
    logic encb2;
    logic enca3;
    logic encb3;
    logic enca4;
    logic encb4;

    logic channel_offset_select;

    //parameters
    parameter ONE_HZ_PERIOD = 25_000_000;
    parameter DEBOUNCE_COUNT = 1000000;
    parameter DECODER_DEBOUNCE_COUNT = 10000;

    parameter CHANNEL_OFFSET = 8 + 70*4; // or 8 + 70*7
    //debounced incoming signals
    debounce #(.DEBOUNCE_COUNT(DEBOUNCE_COUNT)) db1 (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(btnu),.clean_out(manual_trigger));          //manual trigger

    debounce #(.DEBOUNCE_COUNT(DEBOUNCE_COUNT)) db_encoder1_btn (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(~encoder_btn1),.clean_out(encoder_button1)); //encoder btn 1
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder1_a (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_A1),.clean_out(enca1));             //encoder A 1
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder1_b (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_B1),.clean_out(encb1));             //encoder B 1



    debounce #(.DEBOUNCE_COUNT(DEBOUNCE_COUNT)) db_encoder2_btn (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(~encoder_btn2),.clean_out(encoder_button2)); //encoder btn 2
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder2_a (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_A2),.clean_out(enca2));             //encoder A 2
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder2_b (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_B2),.clean_out(encb2));             //encoder B 2

    debounce #(.DEBOUNCE_COUNT(DEBOUNCE_COUNT)) db_encoder3 (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(~encoder_btn3),.clean_out(encoder_button3)); //encoder btn 3
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder3_a (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_A3),.clean_out(enca3));             //encoder A 3
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder3_b (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_B3),.clean_out(encb3));             //encoder B 3

    debounce #(.DEBOUNCE_COUNT(DEBOUNCE_COUNT)) db_encoder4 (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(~encoder_btn4),.clean_out(encoder_button4)); //encoder btn 4
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder4_a (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_A4),.clean_out(enca4));              //encoder A 4
    debounce #(.DEBOUNCE_COUNT(DECODER_DEBOUNCE_COUNT)) db_encoder4_b (.reset_in(reset_in),
               .clock_in(clock_in), .noisy_in(encoder_B4),.clean_out(encb4));              //encoder B 4

    assign attenuation_enable1 = sw_in[14];
    assign attenuation_enable2 = sw_in[14];
    assign ac_coupling_enable = sw_in[13];
    assign channel_offset_select = sw_in[2];
    assign led_in[0] = encoder_button1;
    assign led_in[1] = encoder_button2;
    assign filter_enable = sw_in[0];
    assign channel_measure_select = sw_in[1];
    assign run_mode = sw_in[15];
    assign edge_toggle = sw_in[3];


    //ENCODERS
    logic reduce_offset;
    logic add_offset;
    logic add_threshold;
    logic reduce_threshold;
    logic add_time;
    logic sub_time;

    encoder_handler encoder1 (.clk(clock_in), .A(enca1), .B(encb1), .RST(encoder_button1),
                              .EncOut(voltage_scaling_setting));
    logic [4:0] encoder_tally2;
    encoder_handler encoder2 (.clk(clock_in), .A(enca2), .B(encb2), .RST(encoder_button2),
                              .ADD(add_offset), .SUB(reduce_offset));

    logic [4:0] encoder_tally3;
    encoder_handler encoder3 (.clk(clock_in), .A(enca3), .B(encb3), .RST(encoder_button3),
                              .ADD(add_threshold), .SUB(reduce_threshold));
    logic [4:0] encoder_tally4;
    encoder_handler encoder4 (.clk(clock_in), .A(enca4), .B(encb4), .RST(encoder_button4),
                              .ADD(add_time), .SUB(sub_time));

    //CHANNEL OFFSETS
    parameter CHANNEL_A_OFFSET = 8 + 70*4;
    parameter CHANNEL_B_OFFSET = 8 + 70*7;

    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            channel_a_offset <= CHANNEL_A_OFFSET;
            channel_b_offset <= CHANNEL_B_OFFSET;
        end else begin
            if (add_offset) begin
                if (channel_offset_select) begin
                    channel_b_offset <= (channel_b_offset == 708) ?
                                            channel_b_offset : channel_b_offset + 10;
                end else begin
                    channel_a_offset <= (channel_a_offset == 708) ?
                                            channel_a_offset : channel_a_offset + 10;
                end
            end else if (reduce_offset) begin
                if (channel_offset_select) begin
                    channel_b_offset <= (channel_b_offset == 8) ? channel_b_offset : channel_b_offset - 10;
                end else begin
                    channel_a_offset <= (channel_a_offset == 8) ? channel_a_offset : channel_a_offset - 10;
                end
            end

        end
    end

    //TRIGGER THRESHOLDING
    parameter CHANNEL_THRESHOLD = 2048;

    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            threshold <= CHANNEL_THRESHOLD;
        end else begin
            if (reduce_threshold) begin
                threshold <= (threshold == 4095) ? threshold : threshold + 16;
            end else if (add_threshold) begin
                threshold <= (threshold == 0) ? threshold : threshold - 16;
            end else begin
                threshold <= encoder_button3 ? CHANNEL_THRESHOLD : threshold;
            end
        end
    end
```

```

    //TIMESCALE SELECT
    always_ff @(posedge clock_in) begin
        if (reset_in) begin
            sample_speed_setting <= 3'b0;
        end else begin
            if (sub_time) begin
                sample_speed_setting <= (sample_speed_setting == 3'b100) ?
                                        sample_speed_setting : sample_speed_setting + 1;
            end else if (add_time) begin
                sample_speed_setting <= (sample_speed_setting == 3'b000) ?
                                        sample_speed_setting : sample_speed_setting - 1;
            end else begin
                sample_speed_setting <= encoder_button4 ? 0 : sample_speed_setting;
            end
        end
    end
endmodule //interfacing


///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
///////////////////////////////////////////////////////////////////////////////


`default_nettype none

module debounce (input wire reset_in, clock_in, noisy_in,
                 output logic clean_out);
   parameter DEBOUNCE_COUNT = 1000000;
   logic [19:0] count;
   logic new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count == DEBOUNCE_COUNT) clean_out <= new_input;
     else count <= count+1;


endmodule

`default_nettype wire //debounce
```

5.2 Waveform Demonstrations

See waveforms for demonstration of triggering, time/voltage scaling change, and measurement capabilities. All photos were taken while the DSO was triggering.

5.2.1 10kHz 500mVpp Waveforms

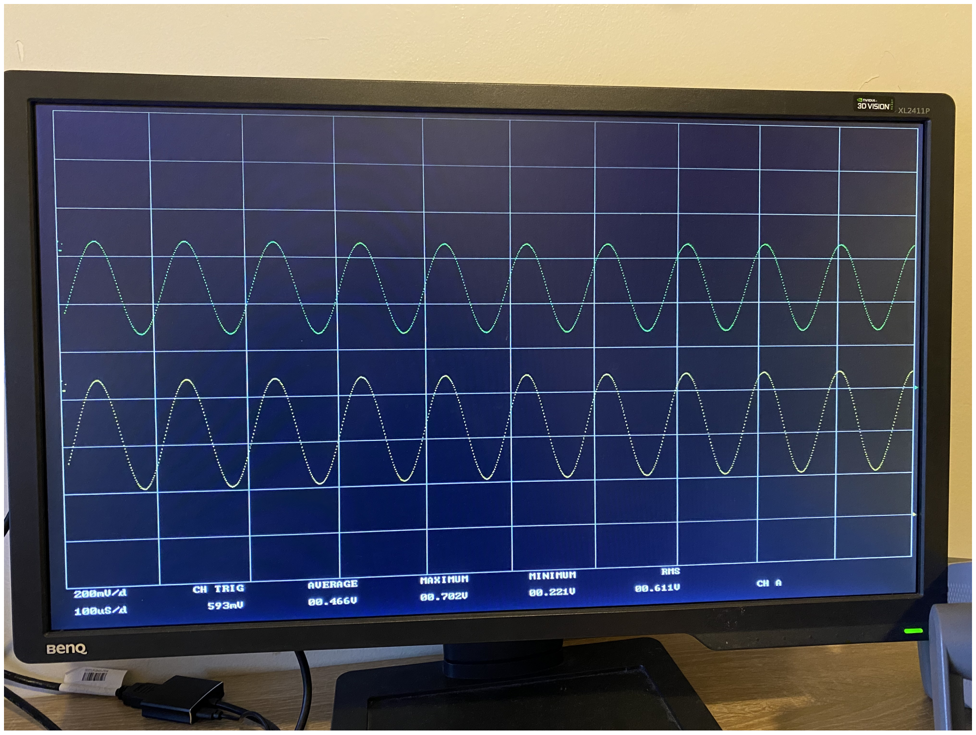

```
200mV/d    CH TRIG   AVERAGE   MAXIMUM   MINIMUM   RMS       CH A
100uS/d    593mV     00.466V   00.702V   00.221V   00.611V
```

Figure 11: Scoping on a 10kHz 500mVpp sine wave

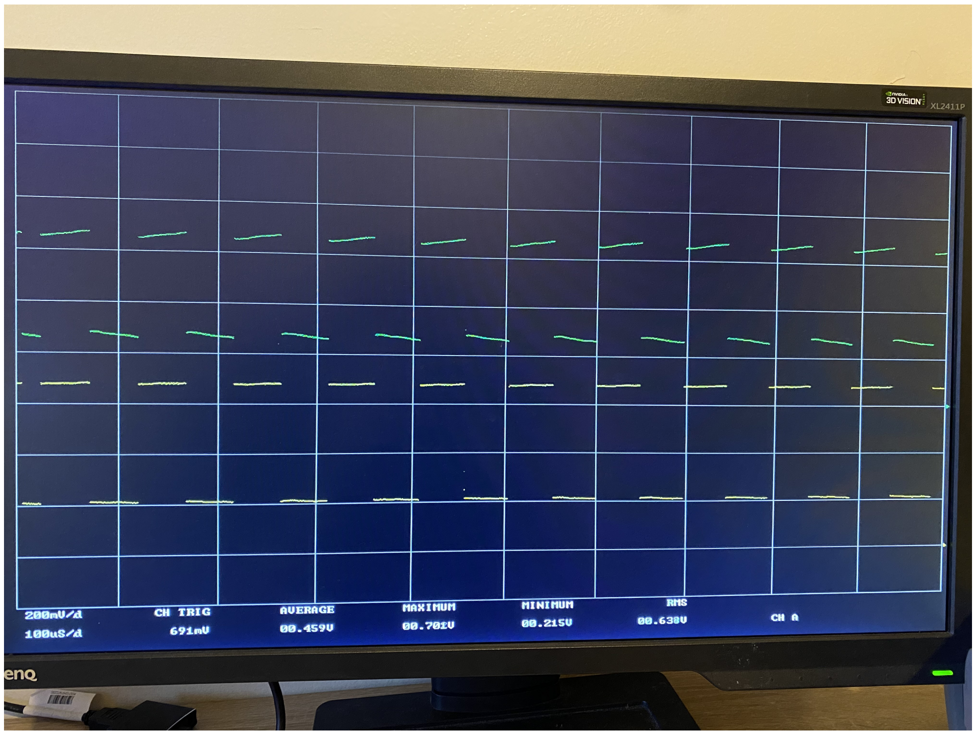

```
200mV/d    CH TRIG   AVERAGE   MAXIMUM   MINIMUM   RMS       CH A
100uS/d    691mV     00.459V   00.701V   00.215V   00.638V
```

Figure 12: Scoping on a 10kHz 500mVpp square wave (need to tweak probe capacitance on CHB)

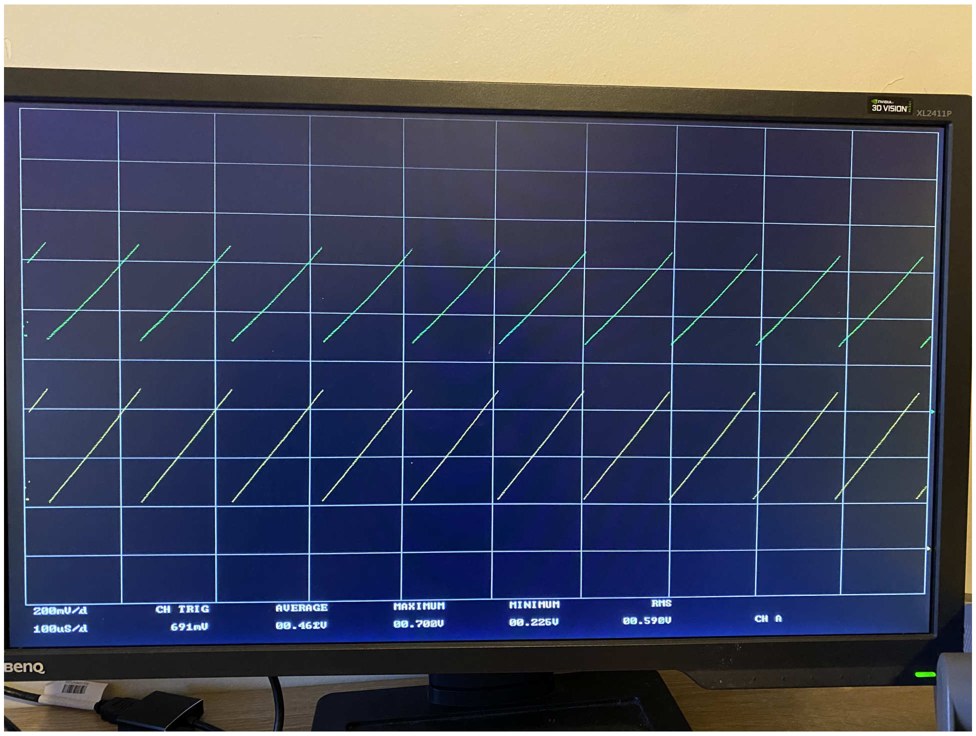


Figure 13: Scoping on a 10kHz 500mVpp ramp wave

5.2.2 1kHz 500mVpp Waveforms - Rising vs Falling Edge

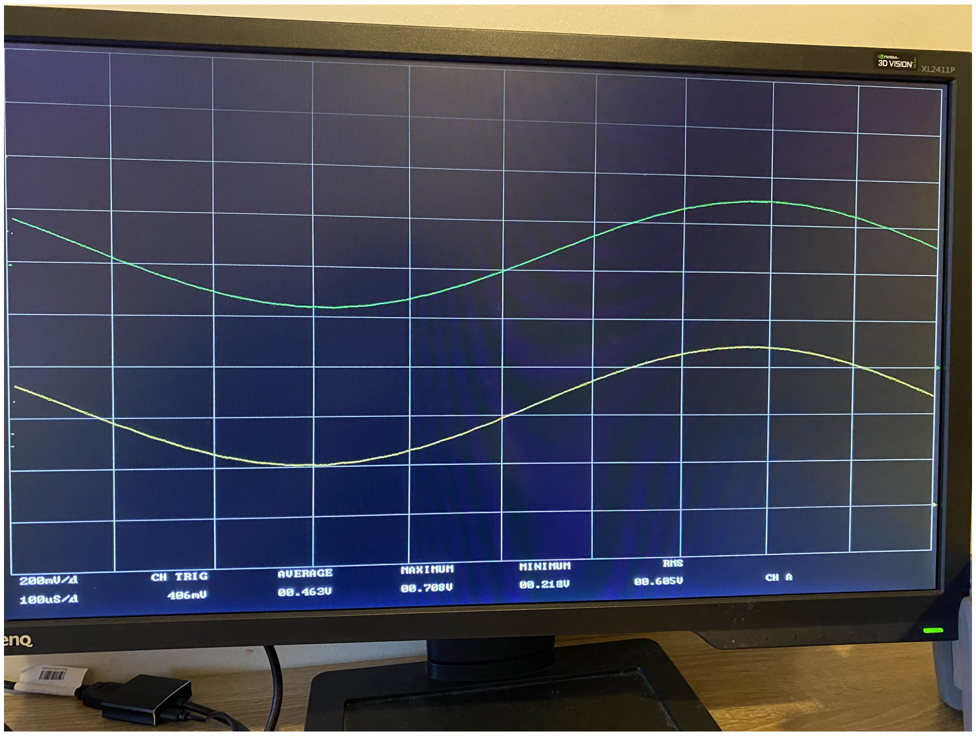


Figure 14: Scoping on a 1kHz 500mVpp wave, trigger on rising edge

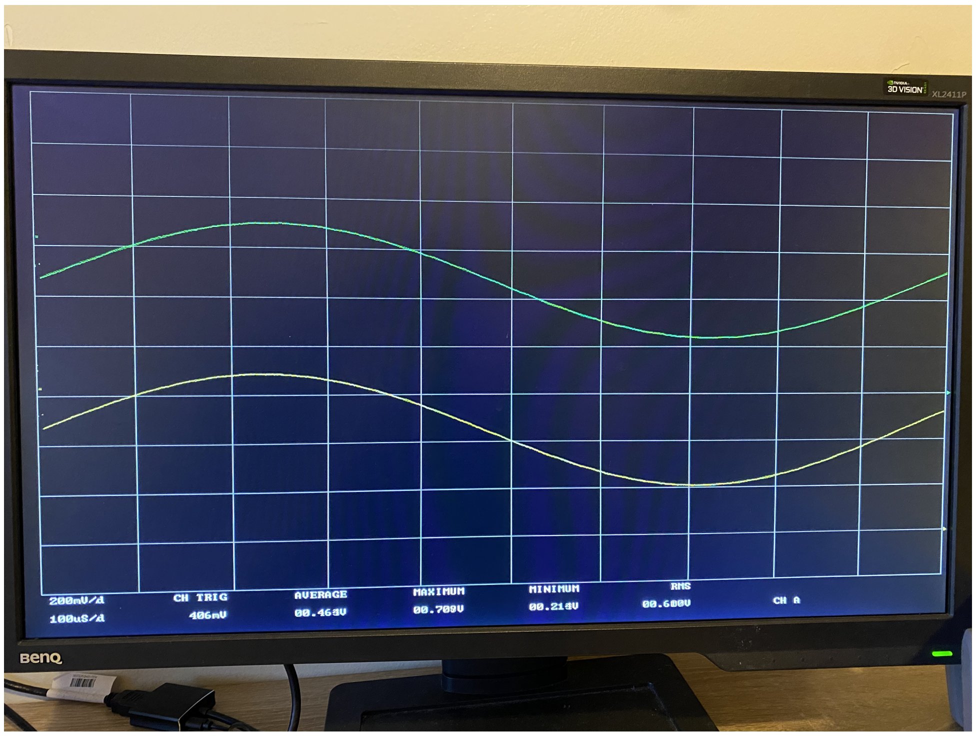


Figure 15: Scoping on a 1kHz 500mVpp wave, trigger on falling edge

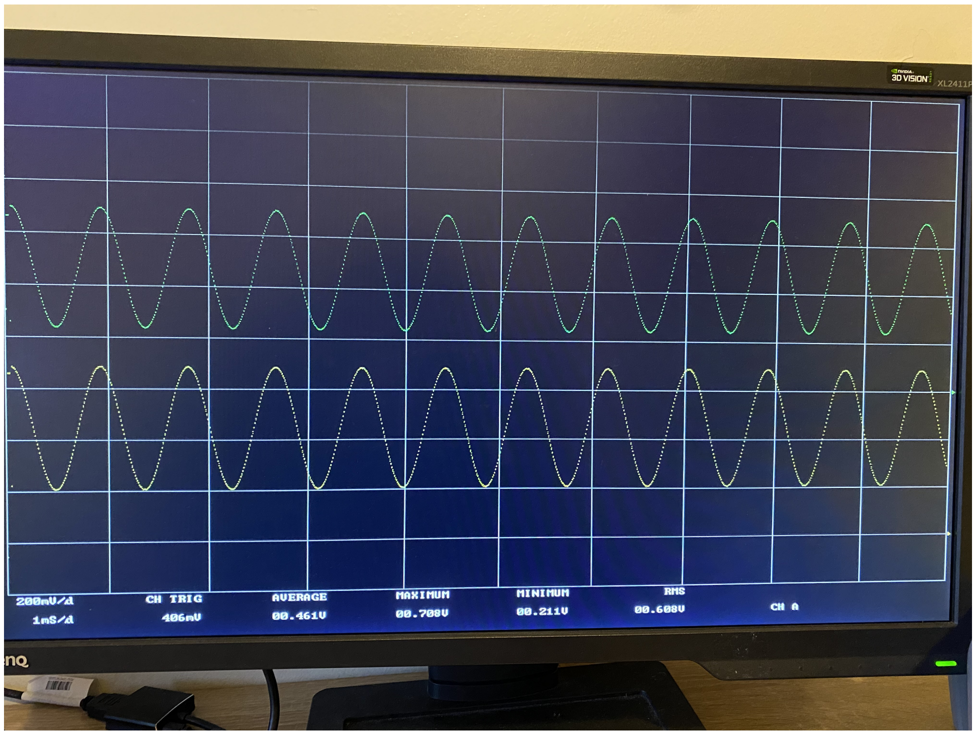


Figure 16: Scoping on a same wave as before, zoomed out 10x in timescale

5.3 External Pmod Analog Front End Design Documentation

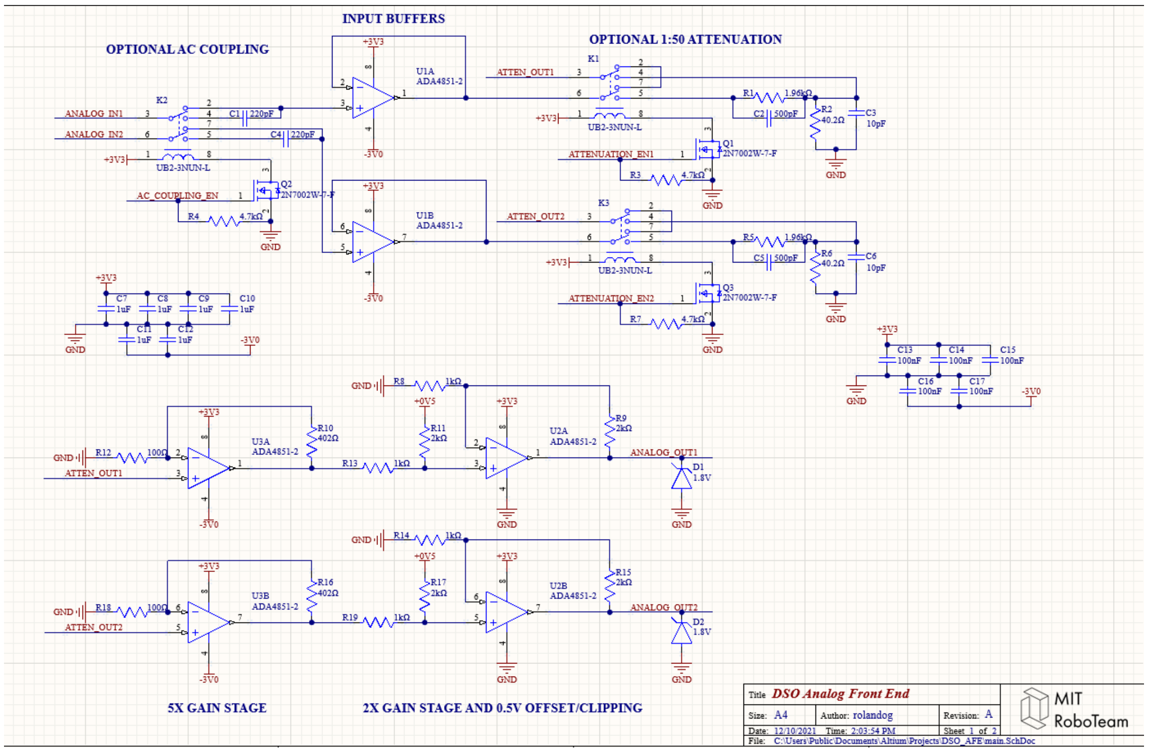


Figure 17: Pmod AFE board analog signal pipeline schematic

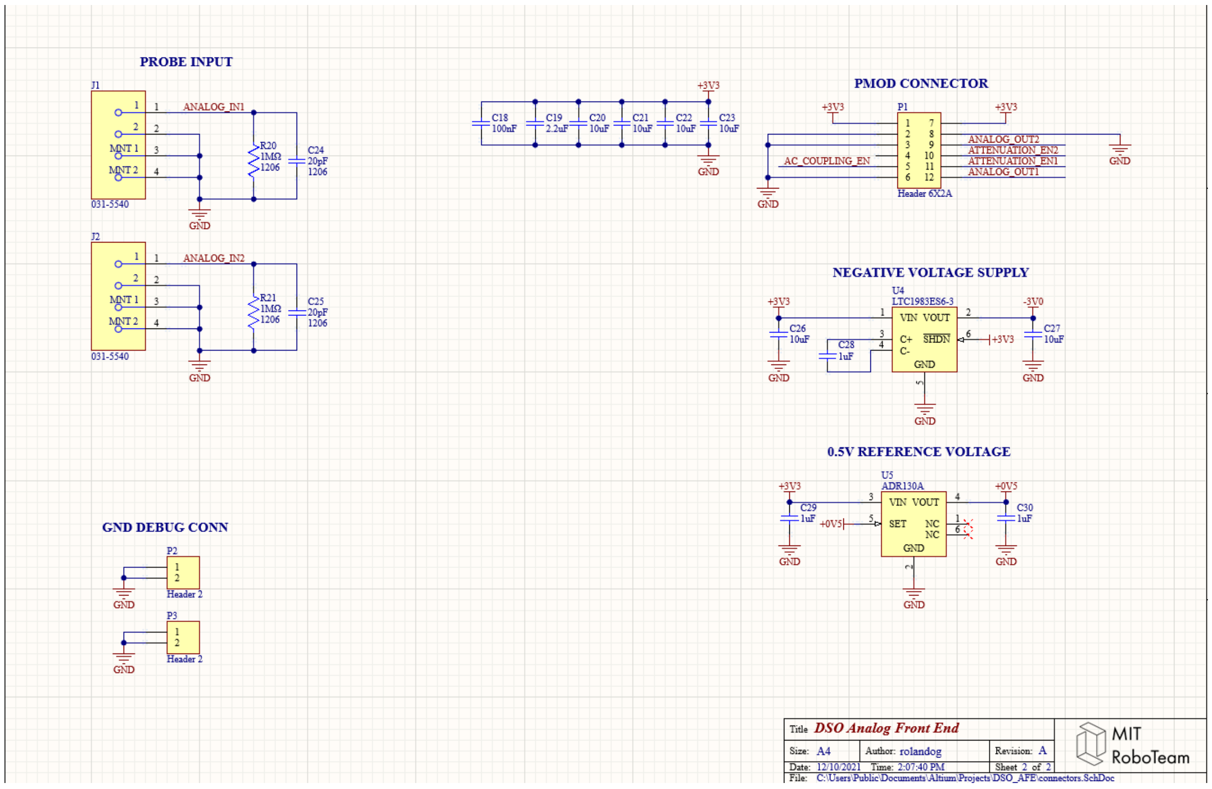
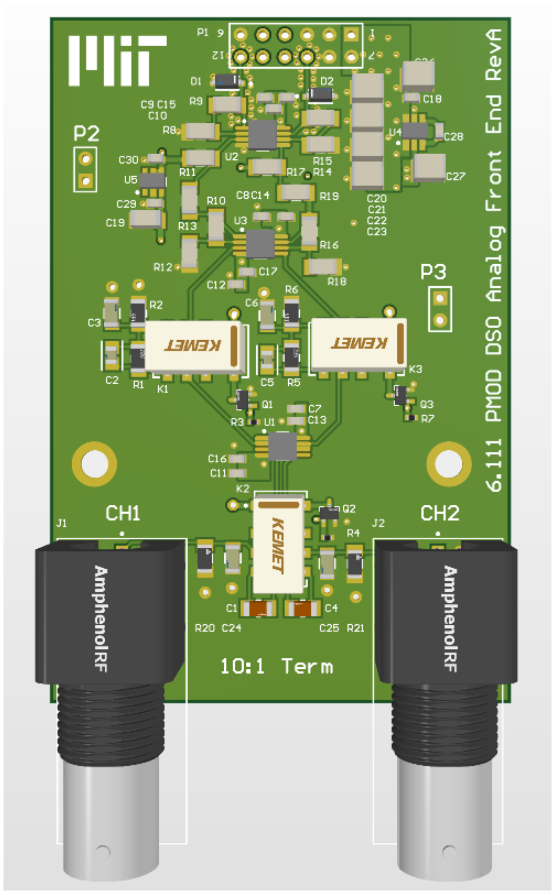
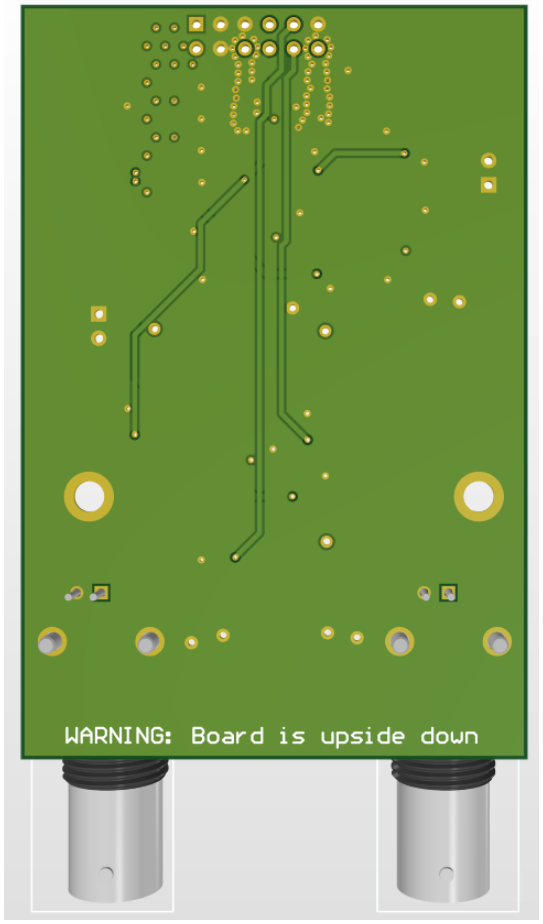


Figure 18: Pmod AFE board connectors and power schematic

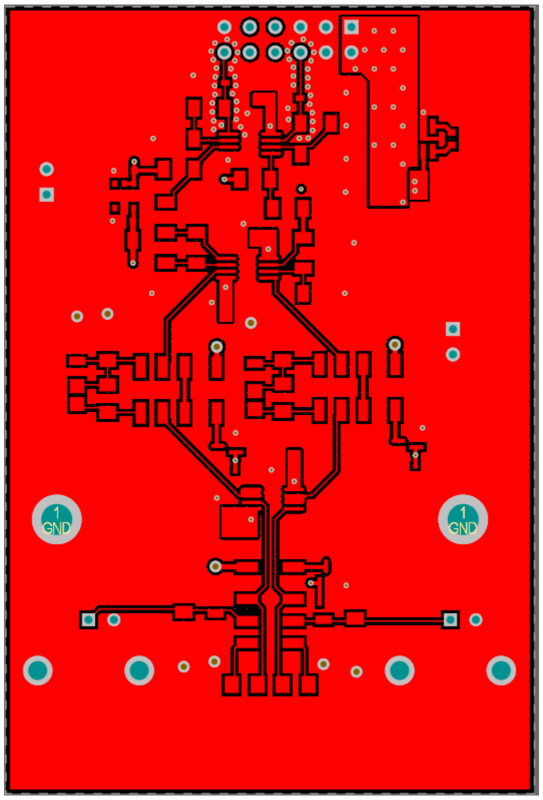


(a) Pmod AFE 3D render, front side

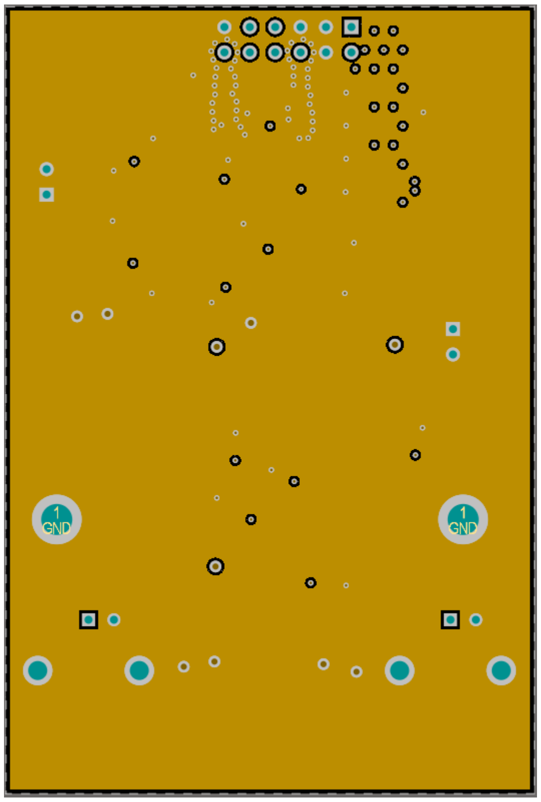


(b) Pmod AFE 3D render, back side

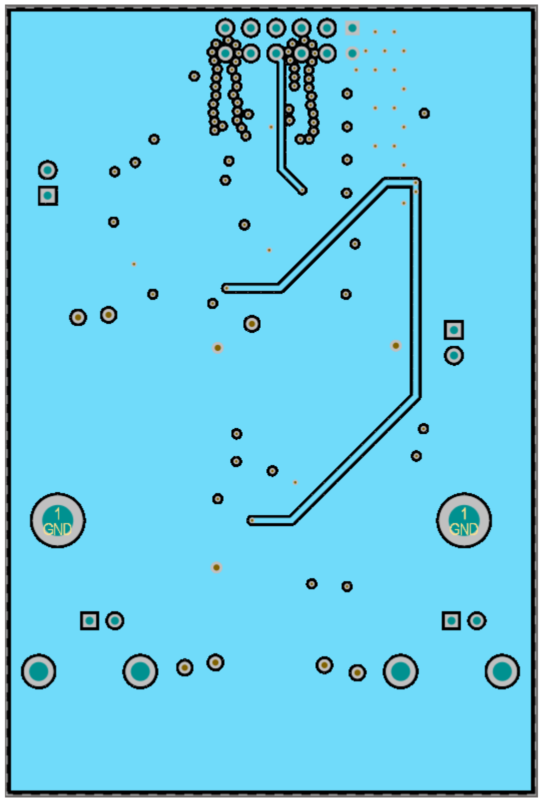
Figure 19: Pmod AFE PCBA 3D Render



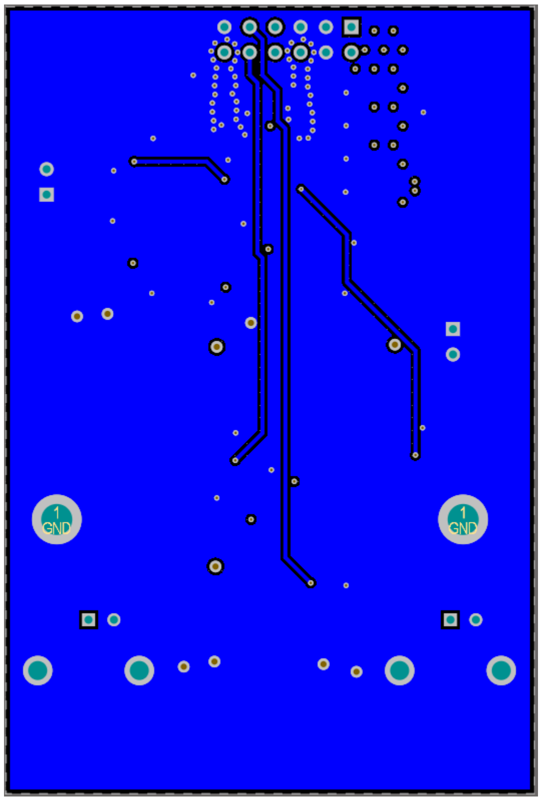
(a) Pmod AFE PCB top layer



(b) Pmod AFE PCB midtop layer



(c) Pmod AFE PCB midbottom layer



(d) Pmod AFE PCB bottom layer

Figure 20: Pmod AFE board copper layers layout