FPGA Oscilloscope Design

Will Buchta

B Term 2023

ECE3999: Independent Study

Table of Contents

Abstract

I have worked on many complex projects in the past, but started to realize that the end-product wasn't 'useful' to the real world. Sure, I learned the steps of how to design complex systems, but I hadn't applied my skills in a way to produce something genuinely impressive to those in industry. Thus, I started looking for possible projects that would keep me entertained, but also impress potential employers in the future.

This project was aimed at developing the skills necessary to design complex FPGA centered printed circuit boards (PCBs), and the digital systems running on them. I started with a goal of making a functional digital oscilloscope and progressed through analog front-end design to creating necessary peripherals, and finally digital system design with Verilog. The current milestone that I am working on is debugging the digital system at various layers of abstraction.

Introduction

Small scale consumers suffered heavily during the chip shortage starting in 2020, with Xilinx FPGAs and SoCs becoming unobtanium. Once the low-end FPGAs started coming back in stock, I started thinking about this project. The first step in designing was creating the analog front end, which will be discussed in more detail later. Given that most FPGAs have a standard out-of-the-box clock speed of 100MHz, I chose an ADC with a maximum sample rate of 105Msps to allow for easy acquisition. In addition to the analog front end, I included 256MB of DDR3 SDRAM, VGA video output, a seven-segment display clock, an STM32 for a USB keyboard or mouse, and composite video decoder. The PCB itself is an 8 layer, 150x110mm board, with 10 distinct power rails. See Figure 1 below for a high-level block diagram:

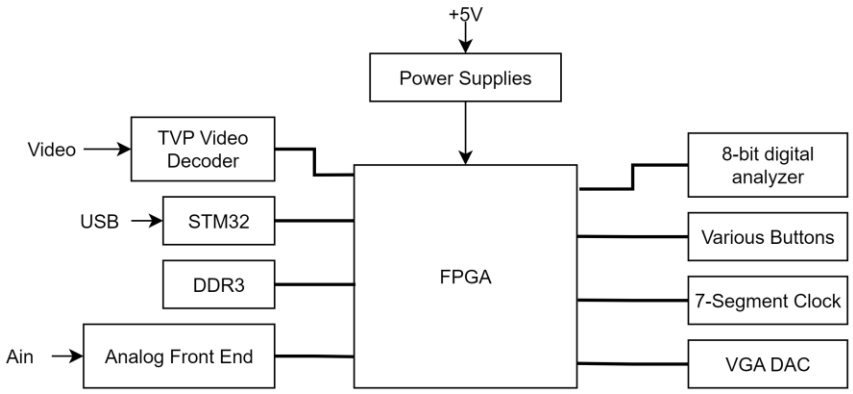


Figure 1: High level board block diagram

General Design

Estimating the required FPGA fabric resources for any project is difficult beforehand, as the system inherently changes size through the development cycle. I originally started designing the board with the low range Spartan-7 25 variant but moved up to the Artix-7 50 after some consideration. This upgrade allowed for faster memory, more supported IP blocks, and headroom when writing HDL, while not hitting significantly in the cost department. A main concern of mine was the physical manufacturability of the board – going through months of design for the board to turn out a dud would not be ideal. As this was my first PCB design with ball grid array (BGA) packages, I erred on the side of caution at all steps. I chose the Artix-7 IC with a pin pitch of 1mm (vs the standard 0.8mm), allowing more clearance for trace breakout. This also allowed two traces be broken out between pads in some tight areas, greatly easing layout complexity. See a diagram from UG1099, page 14 [5]:

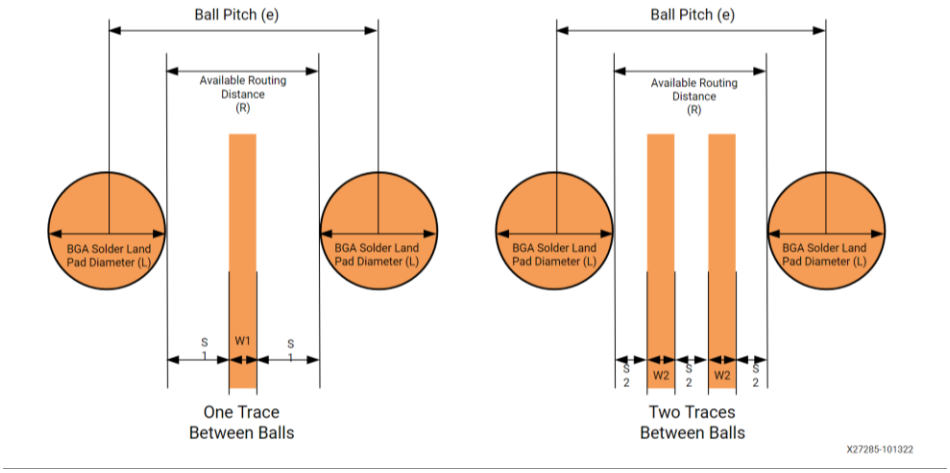


Figure 2: BGA package trace breakout

The first step in designing the schematic was to include the schematic symbol and PCB footprint of the chosen FPGA. Unfortunately, it is hard to find these readily available online. Xilinx provides text files for each FPGA, detailing the type, name, and function for each. Since this chip has 256 pins, designing them in by hand was not an option. Instead, I created a python script to parse through the given text file, create symbols based on grouping (i.e., pins in bank 15 go here, 14 there, separate symbols for power and ground), while generating the footprint. All required mechanical information for footprint creation was found in UG1099 [5].

FPGA Configuration

The 7 series line of FPGAs have a lot of required 'boiler plate' components to even have the devices boot up. This includes *at least* three power rails, some sort of JTAG interface, SPI flash for configuration, a clock, and a wide range of decoupling capacitors. To choose the flash configuration, I consulted the user guides, specifically UG470, "*7 Series FPGAs Configuration*" [6]. There is a lot of nitty-gritty detail for complex configuration schemes therein which are out of the scope of this document; suffice it to say that I designed the board to be programmable by either JTAG or SPI flash, configurable with a jumper. I also made sure the flash chip I chose was supported by the chosen FPGA. To get the required decoupling, I consulted UG483, "*7 Series FPGAs PCB Design Guide*"[4], which details the recommended size and number of decoupling capacitors per pin bank. The most complex step of the configuration was implementing the JTAG interface, achieved through the industry standard USB-JTAG FT2232HQ IC. This IC is very versatile and supports many different communication interfaces, causing quite a headache. I was able to create the connection diagram with the help of the Typical Applications section in its datasheet [8], its pin descriptions table, as well as an example design from one of Xilinx's evaluation boards. In addition to the configuration SPI flash, I added a second identical flash chip on board for general use. Other general-purpose components on the configuration section

of the schematic include ESD protection, user pushbuttons, mounting holes, and an LED indicating when the device is fully configured.

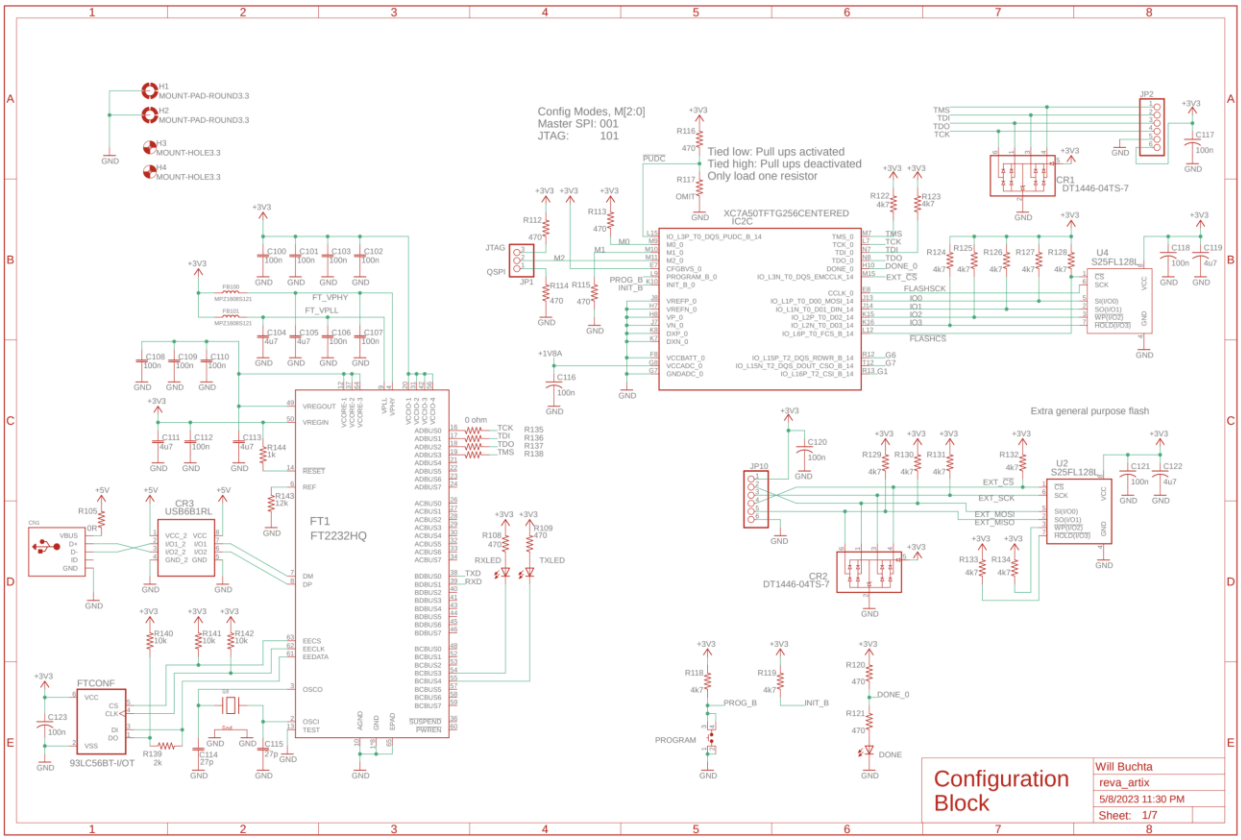


Figure 3: Configuration block of schematic.

Another thing that I like to do when creating a multi-page schematic design is to specially label all passive components; this makes routing a lot easier, as cross referencing between layout and the schematic is simple. All component names are specified as the type, then 3 digits, with the leading digit corresponding to the schematic page. The first resistor on board is “R100”, the first capacitor is “C100”, and so on. On a side note, the order of schematic pages does not indicate the chronological order that I designed this board in. For example, page 2 details FPGA power and VGA output, but was one of the last items to design.

DDR3 Memory

The next logical part to include was a DDR memory IC, given that it is the by far the most complex component. Vivado provides a memory interface generator wizard (MIG, UG586 [9]), listing compatible memory IC part numbers. This wizard also generates a pinout, which was later copied over to the schematic. One important bug that I figured out after quite some time was that the base FPGA clock source must be in a bank in the same “column” as the DDR interface; this is due to the physical internal structure of the clock tree. Sourcing the clock from a different column introduces too much jitter for the high frequency transactions.

From the MIG, I elected to use a 16-bit data interface as opposed to the simpler 8-bit. This was indeed much more work to layout but provided double the bandwidth. From the list of compatible ICs, I chose one that was in stock on Digikey and had a decent capacity of 2Gbit / 256MB. When finishing the wizard, an I/O constraints file was generated, which I carefully copied over to the schematic. A fun little trick to help when routing DDR interfaces is called “bit swapping”; it is possible to swap two bits in a data byte group to ease layout.

I specifically did not include termination resistors for the address/command and control pins, as the complexity would have been far too high. It is standard practice to include when routing between multiple ICs, but I made sure to keep the FPGA and DDR chip as close as possible. Adding termination resistors would have required a power rail equal to half the DDR power supply, at 0.75V, along with several other components. The data lines have internal termination resistors.

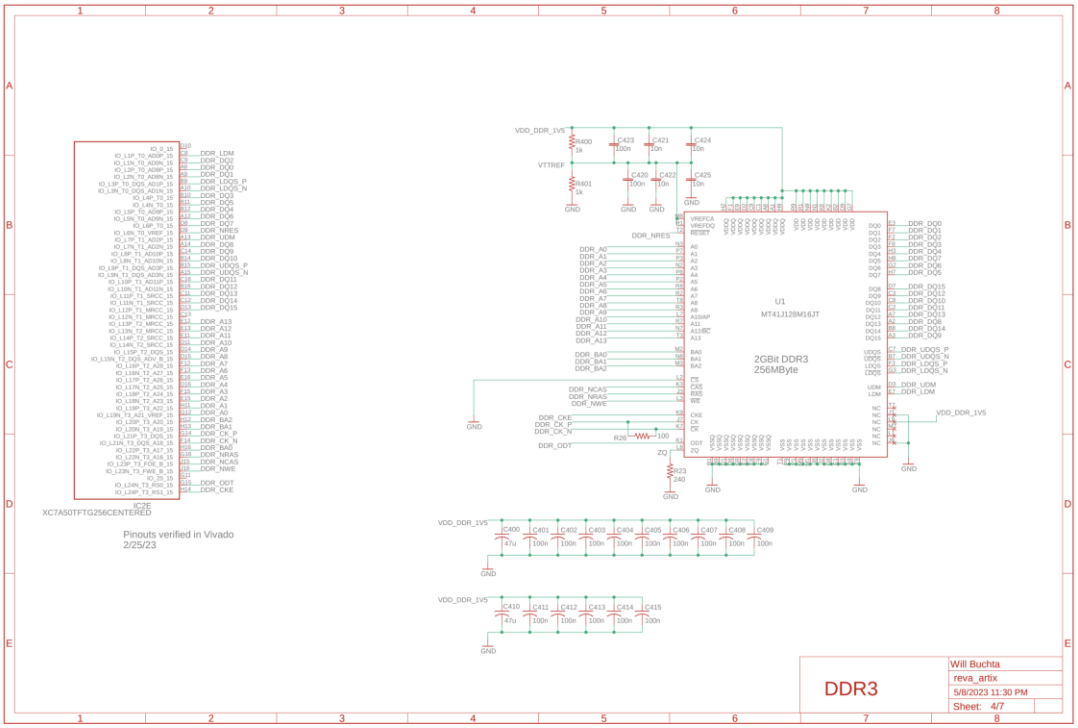


Figure 4: DDR3-FPGA connection diagram

The layout of the DDR was quite difficult and took several days with multiple restarts. For starters, there are 96 pins with a 0.8mm pitch, pushing the physical manufacturing constraints – I was not able to place vias between the pads. Secondly, ***extreme*** care is needed when routing. Below are some of the design ideas that need to be met for successful 400MHz operation:

- All traces in a group should be routed on the same layer (eg D7-D0, D15-D8, address groups).
- All traces must have a controlled impedance.
- When a signal trace changes reference planes, a ground via is needed to accompany the high frequency return current [1]. Effectively one ground via near a signal via or group of signal vias.
- Differential traces must be matched in time relative to other traces, as well as inner-pair length matching.
- Package delays must be considered. As the FPGA is a wire-bonded integrated circuit, each BGA pin is connected internally to a wire which then connects to actual silicon. This signal may then pass through more internal silicon, introducing further distance the signal must travel. For lower speed interfaces, this does not matter. However, at the high frequencies of DDR memory interfaces, the window for data is extremely tight. It is imperative that all bits of a byte group reach the destination at the same time, and are setup before the clock – this is measured by the so-called "eye diagram". When routing these interfaces, there is a time budget associated, starting at the period of the clock signal. From this period, setup and hold times are deducted, as well as noise, among other things. See Figure 5 for an example. Note that this is for a clock rate of 167MHz, and the interface designed for the Oscilloscope is at 400MHz, a data rate of 800MHz.

Table 18. Eye Diagram Budget for Address, Command, Control—DLL Method

Skew Contributor	Skew Item	Value	Unit	Comment
Total budget	Nominal clock period description	6000	ps	Ideal Eye. Applied clock rate of 167 MHz, data rate = 333 MHz.
DDR-related	Setup/hold time requirements at memory	1600	ps	Directly from DDR data sheet
Interconnect related	ISSI, crosstalk, loading degradation	950	ps	Based on a single bank of memory with five (x16) chips
	Clock skew in system	50	ps	Ideally could be smaller if clock rules are followed
	Capacitive mismatch on inputs	25	ps	—
	V_{REF} reduction	50	ps	—
Interconnect related (continued)	Termination mismatch	15	ps	—
	Trace matching with group	150	ps	Assumes max trace delta within group of 0.75 inches * 180 ps/in
85xx related	Eye movement [1]	300	ps	Nominal amount. DLL in normal operation.
	Clock skew	150	ps	Amount of clock skew on-chip
	Clock to out spread	2000	ps	Assumes tco_min = 1 ns Assumes tco_max = 3 ns Operation: DLL method used
eye_margin	Extra margin on the eye diagram after all worst case parameters are included [2]	710	ps	—

Figure 5: Eye diagram budget [2].

To account for package delays, Xilinx provides a breakdown for each pin of all offered FPGAs. From this, the XC7A50T in the FTG256 package was generated and used. Some PCB software allows for these delays to be integrated directly in the environment, but the software I used for the project does not. Instead, the signal speed based on different layers had to be considered by hand – a tricky task. There are several factors that go into determining the actual required length of each trace, starting with signal propagation time. It is not that the traces to all pins have to be the same length, it's that the traces must be matched in *time*.

To talk about propagation, it is first necessary to detail layer stackup. The PCB stackup I used was a standard stackup taken from the manufacturer's website, which detailed dielectric thicknesses and effective dielectric constant. The actual ordering of layers I chose is beyond the scope of this document; suffice it to say that the first step for signal integrity is a good stackup. Notice in Figure 6 how there are no two adjacent signal layers – if the electric field of two high frequency traces interfere with each other in the same dielectric layer, hideous amounts of crosstalk occur. Thus, all signal layers are separated by at least one ground plane.

The stackup information was fed into the Saturn PCB Toolkit [3], a useful PCB calculator. With the goal of a 50 Ohm impedance, the trace width was varied, producing the following table:

Layer of Stackup	Trace width(mil)	tprop (ps/in)	Impedance	1/((ps/in)/1000) = mil/ps
Signal	12	147.2	50.8	6.79
7628 0.18mm				
Ground				
Core 0.3mm				
Signal	7.6	175.5	49.95	5.70
7628 0.18mm				
Ground				
Core 0.3mm				
Power				
7628 0.18mm				
Signal	7.6	175.5	49.95	5.70
Core 0.3mm				
Ground				
7628 0.18mm				
Signal	12	147.2	50.8	6.79

Figure 6: Stackup and trace delays

Notice how the signals on internal layers travel at a slower speed than those on the outer layers.

The next step in determining final trace lengths was to find the distance of the trace with the longest internal package delay, which would set the total time delay for each trace. This pin was found from the generated package delay table and routed first. Once the length of this trace was measured, the total time delay was calculated, and from there, the required length of all

other traces could be calculated using an excel spreadsheet (easier said than done). Below is a sample of the spreadsheet. The highlighted cells in the Signal Name column indicate where traces had to be routed on different layers.

IO Bank	Pin Number	Site Type	Signal Name	total delay (ps)	Average Trace Delay (ps)	required length(mm)	required length (mils)
15	E12	IO_L13P_T2_MRCC_15	ddr3_addr[13]	298.5	89.6195	36.02	1418
15	E13	IO_L13N_T2_MRCC_15	ddr3_addr[12]	298.5	92.9575	35.45	1396
15	E11	IO_L14P_T2_SRCC_15	ddr3_addr[11]	298.5	107.417	32.96	1297
15	D11	IO_L14N_T2_SRCC_15	ddr3_addr[10]	298.5	104.6365	33.43	1316
15	D14	IO_L15P_T2_DQS_15	ddr3_addr[9]	298.5	91.3155	35.73	1407
15	D15	IO_L15N_T2_DQS_ADV_B_	ddr3_addr[8]	298.5	93.818	35.30	1390
15	F12	IO_L16P_T2_A28_15	ddr3_addr[7]	298.5	95.639	29.37	1156
15	F13	IO_L16N_T2_A27_15	ddr3_addr[6]	298.5	84.962	36.83	1450
15	E16	IO_L17P_T2_A26_15	ddr3_addr[5]	298.5	98.005	34.58	1361
15	D16	IO_L17N_T2_A25_15	ddr3_addr[4]	298.5	97.4525	34.67	1365
15	F15	IO_L18P_T2_A24_15	ddr3_addr[3]	298.5	97.465	34.67	1365
15	E15	IO_L18N_T2_A23_15	ddr3_addr[2]	298.5	98.059	29.02	1143
15	H11	IO_L19P_T3_A22_15	ddr3_addr[1]	298.5	87.1145	30.60	1205
15	G12	IO_L19N_T3_A21_VREF_15	ddr3_addr[0]	298.5	84.0405	31.05	1222
15	H12	IO_L20P_T3_A20_15	ddr3_ba[2]	298.5	71.842	39.09	1539
15	H13	IO_L20N_T3_A19_15	ddr3_ba[1]	298.5	63.4445	40.54	1596
15	H16	IO_L22P_T3_A17_15	ddr3_ba[0]	298.5	80.6895	31.53	1242
15	F14	IO_L21N_T3_DQS_A18_15	ddr3_ck_n[0]	298.5	69.0915	39.57	1558
15	G14	IO_L21P_T3_DQS_15	ddr3_ck_p[0]	298.5	68.432	39.68	1562
15	H14	IO_L24P_T3_RS1_15	ddr3_cke[0]	298.5	80.432	37.61	1481

Figure 7: Trace length calculation

From this spreadsheet, traces were finally routed.

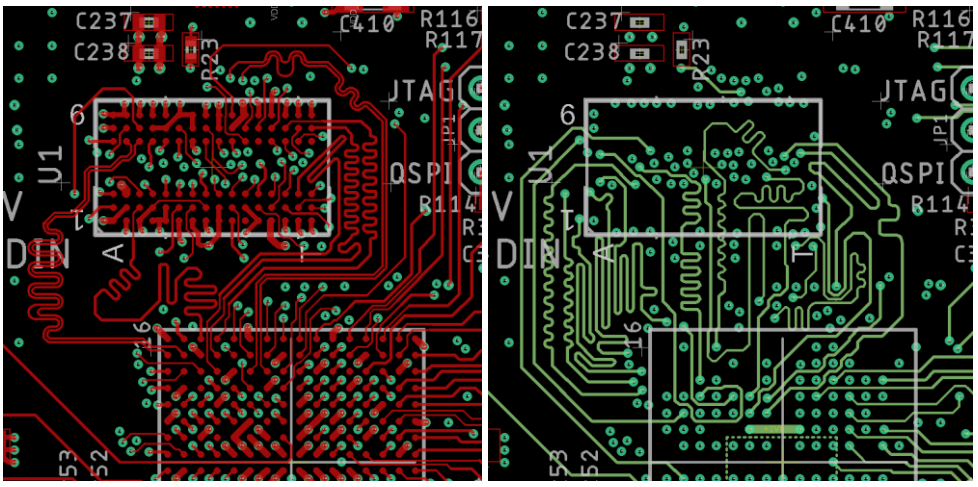


Figure 8: Top two routing layers

The traces were lengthened by a trick called “meandering,” where squiggles are added in the path.

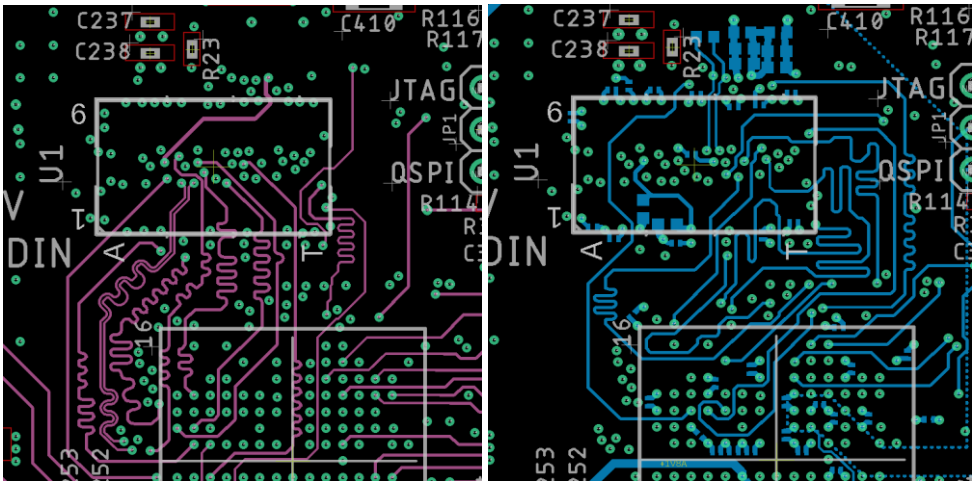


Figure 9: Bottom two routing layers.

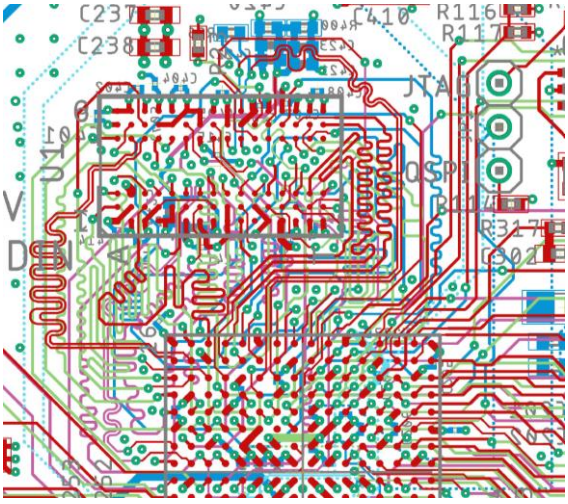


Figure 10: All layers visible at once

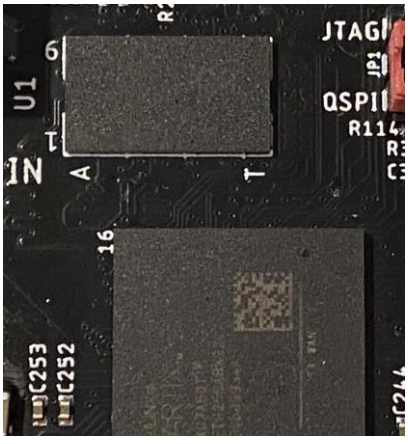


Figure 11: Assembled DDR3

Decoupling and VGA Output

As mentioned in the configuration section, one of the boiler plate requirements for FPGAs is a specified number of decoupling capacitors, described in User Guide 483 [4]. This was implemented on the second page of the schematic. One area that I differed from the user guide was to add extra small capacitance high frequency capacitors, so that each pin would have one. These capacitors were to be mounted on the bottom side of the board, directly under each power pin, providing local decoupling. Due to size constraints, they would have to be in the 0201 package.

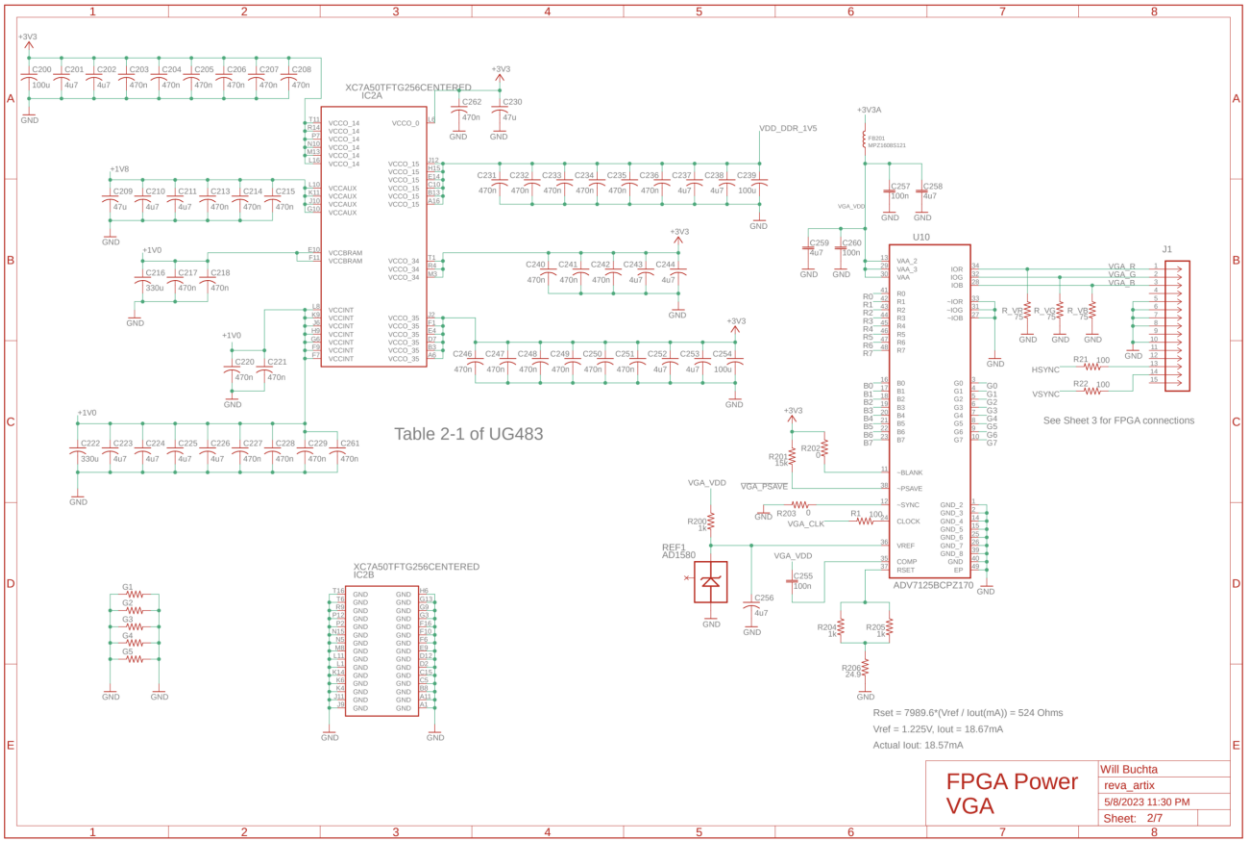


Figure 12: Power supply pins and VGA output

For video output, I chose the ADV7125, a versatile composite video DAC. I originally wished to use an HDMI output, but to buy the required video chips, the design must be licensed by HDMI licensing agents, which was simply not worth it.

The interface is quite simple; 24 bits of RGB data, a pixel clock, as well as horizontal and vertical synchronization pulses. Some evaluation boards use resistor divider DACs for a total of 12 bits of color, but I chose to go with the professional option. Care was taken to provide localized noise reduction with the use of a ferrite bead on the power rails, even after being fed by an LDO (see the power supply section). The voltage reference and Rset values seen towards the bottom left of the symbol were taken from values and equations in the datasheet [10]. The reference and

Rset work together to set the full-scale output of the three current DACs in the chip. With this, a resolution of 1920x1080p is possible, with a pixel clock of 148.5MHz. One trick that I repeated multiple times throughout this design was that of keeping BOM down; for example, instead of having a 525 Ohm resistor from Rset to ground, I used two 1k's in parallel with a 24.9Ohm in series. I also placed many 0 Ohm resistors to allow for signal re-routing after the board had been manufactured.

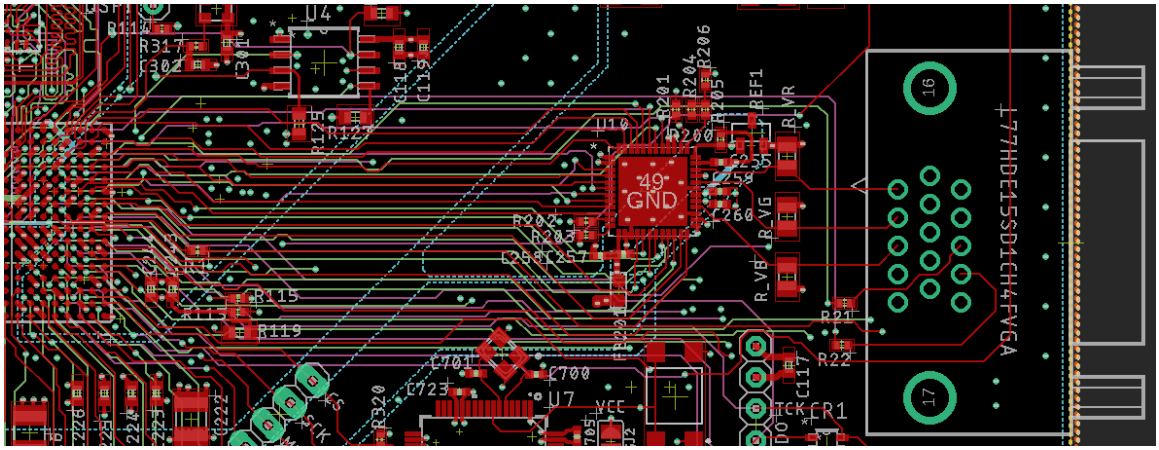


Figure 13: Routing of the VGA output

After completing the DDR routing, VGA routing was a walk in the park. I was able to avoid any signal trace overlap by constraining each bit of the interface to any pin of the FPGA. This was an iterative process, where traces were routed from the VGA DAC backwards to the FPGA, and pins were constrained along the way.

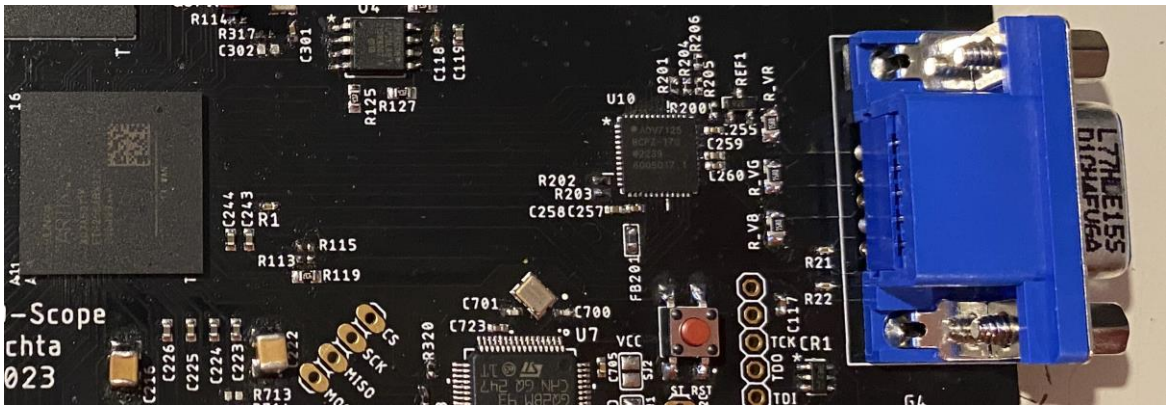


Figure 14: Completed VGA section

Analog front end design

The next design challenge was to design the analog front end, which would provide the actual voltage-to-bits needed in an oscilloscope. This was arguably more challenging than the DDR interface, as the DDR was almost entirely pre-generated, whereas the analog portion was designed independently. Starting off with goals, I wanted a relatively slow (for modern oscilloscopes) ADC to make interfacing easier. The cutting-edge DACs and ADCs today require ultra high-speed serial transceivers, which was way above my paygrade at the time of designing. Since the speed of the ADC could be lowered, bits could be increased, so I settled with a 105MSPS, 10-bit, pipelined ADC from Analog Devices, featuring a parallel data interface. 105MSPS leads to a nyquist input bandwidth of around 50MHz.

The ADC has an input range of +/-1V differential, leading to roughly 2mV per bit at the inputs. Since a scope with a limited input of 1V is useless for the vast majority of measurements, The signal can be divided down by a resistor divider scheme. With a total resistance of 1MOhm (standard), attenuations of 1, 5, or 20 can be achieved. The signal is then routed to a gain/buffer stage, where the input signal can either be doubled or just buffered. Note that the op amps chosen (ADA4899) are capable of being “multiplexed”. This allows for final attenuations of 0.5, 1, 2.5, 5, 10, or 20x. Thus, a maximum input of +/-20V at ~20mV/bit can be realized. To route Vin through these attenuations, high frequency relays were used. Analog switches were investigated as an alternative, but none were capable of the high input voltages at the required bandwidth. Relays were also added to allow for 50 Ohm termination or AC coupling. All relays are driven with NPN transistors, with diodes across the coils to curb high voltage spikes.

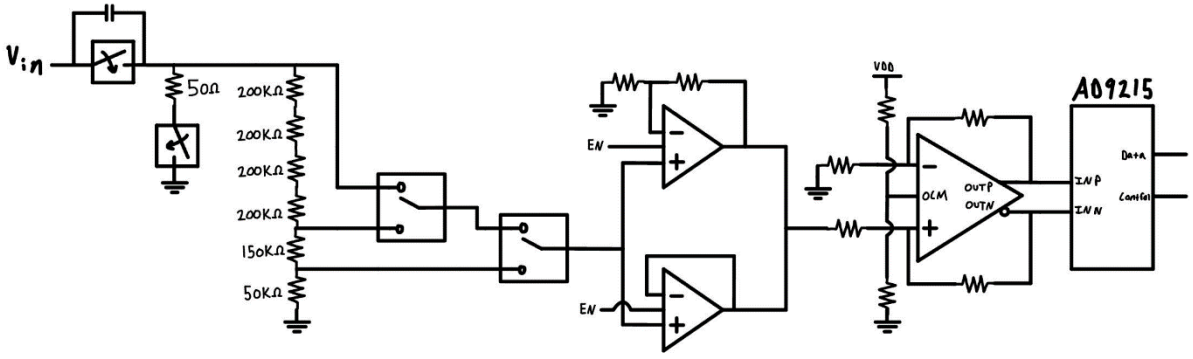
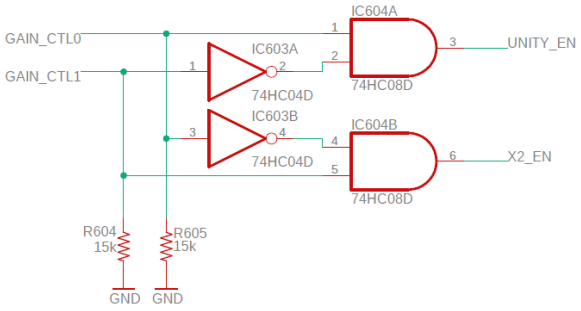


Figure 15: Simplified data acquisition front end

Since the input signal is single-ended and can be positive or negative, it needs to be converted to a differential signal via a fully differential op amp, as seen directly before the ADC. The common mode output voltage of this op amp is equal to half of the voltage supply rail, which provides the best performance for the ADC (as specified in the datasheet [11]).

To control which gain/buffer op amp would be active, discrete logic components were added to ensure operational safety. If the enable pins were controlled directly by the FPGA, a bug could

cause both to be on at the same time, leading to a short between the outputs and a broken op amp. Thus, the following scheme was designed:



G1	G0	X1_EN	X2_EN
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	0

Figure 16: Control of op amps

Looking back, I would modify the buffer/gain stage. The ADA4899 is ultrafast and has good gain vs bandwidth flatness, but the input bias current is significant at 100nA. Since there is a large source impedance when using the resistor divider attenuator, in the worst case, nearly a volt of error can be attained – unacceptable. One way to fix this problem is to use an op amp with a much lower input bias current. This may result in a tradeoff of lower bandwidth.

One thing to note is the voltage supply for the op amps; They are rated for +/-5V operation, but that was not achievable based on the main 5V from the barrel jack power. A large supply voltage (>3.3V) was necessary to prevent the op amps from clipping.

The digital portion of the data acquisition system does not fall under “analog,” but it is relevant to this section. 8 pins from the FPGA were broken out to a level shifter IC, meant to provide some isolation from the real world and any possibly damaging signals.

The last feature of the analog front end is the inclusion of a voltage DAC. The DAC I chose has a full-scale output of 0V to its supply (3.3V). Since an output with both positive and negative capabilities is desirable, some buffering was added to achieve functionality.

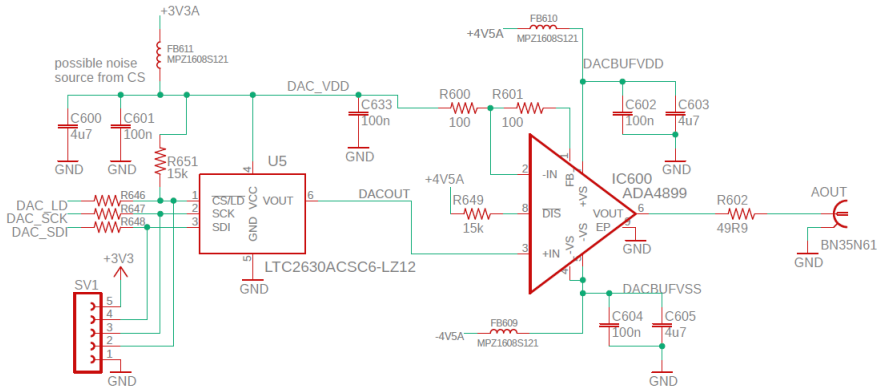


Figure 17: DAC for waveform generation

I would like to note that the entire front end was simulated in LTSpice (see Figure 18):

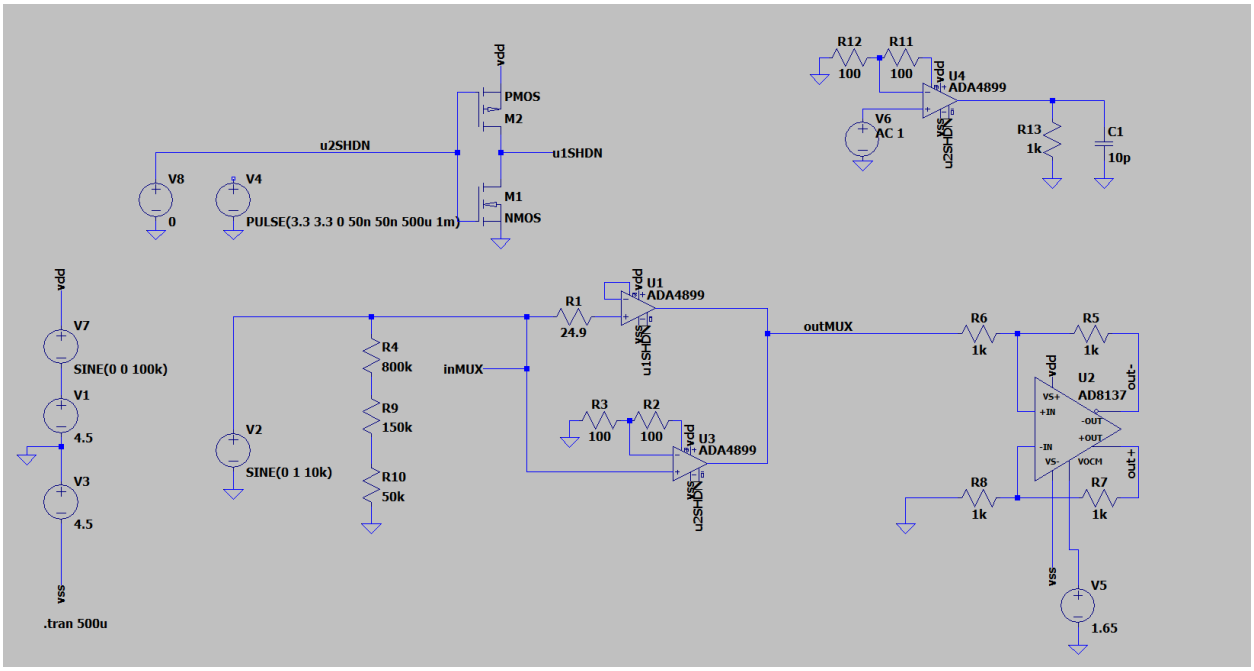


Figure 18: Simulation of analog front end

In terms of layout, the analog section is as far away from the rest of the board as possible, with ground “fencing vias” placed between the two (see Figure 19).

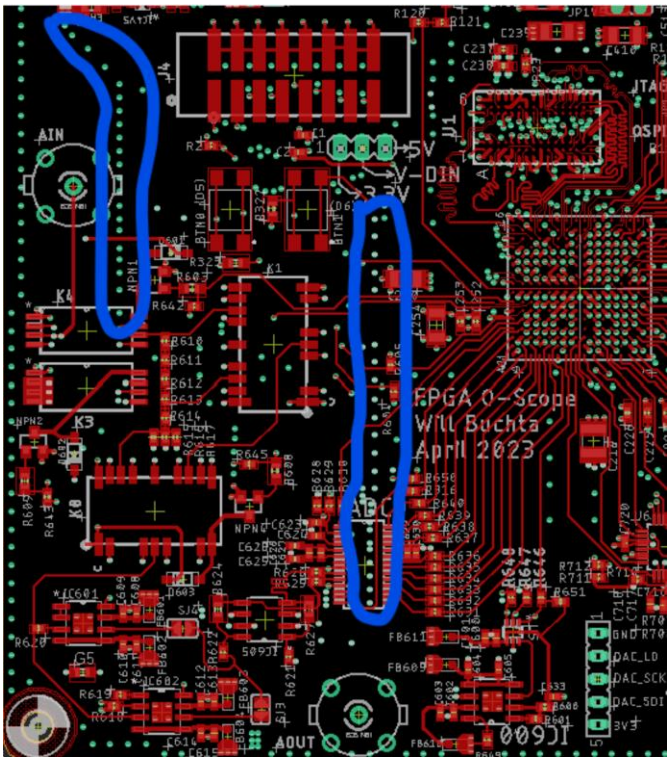


Figure 19: Fencing vias to increase isolation

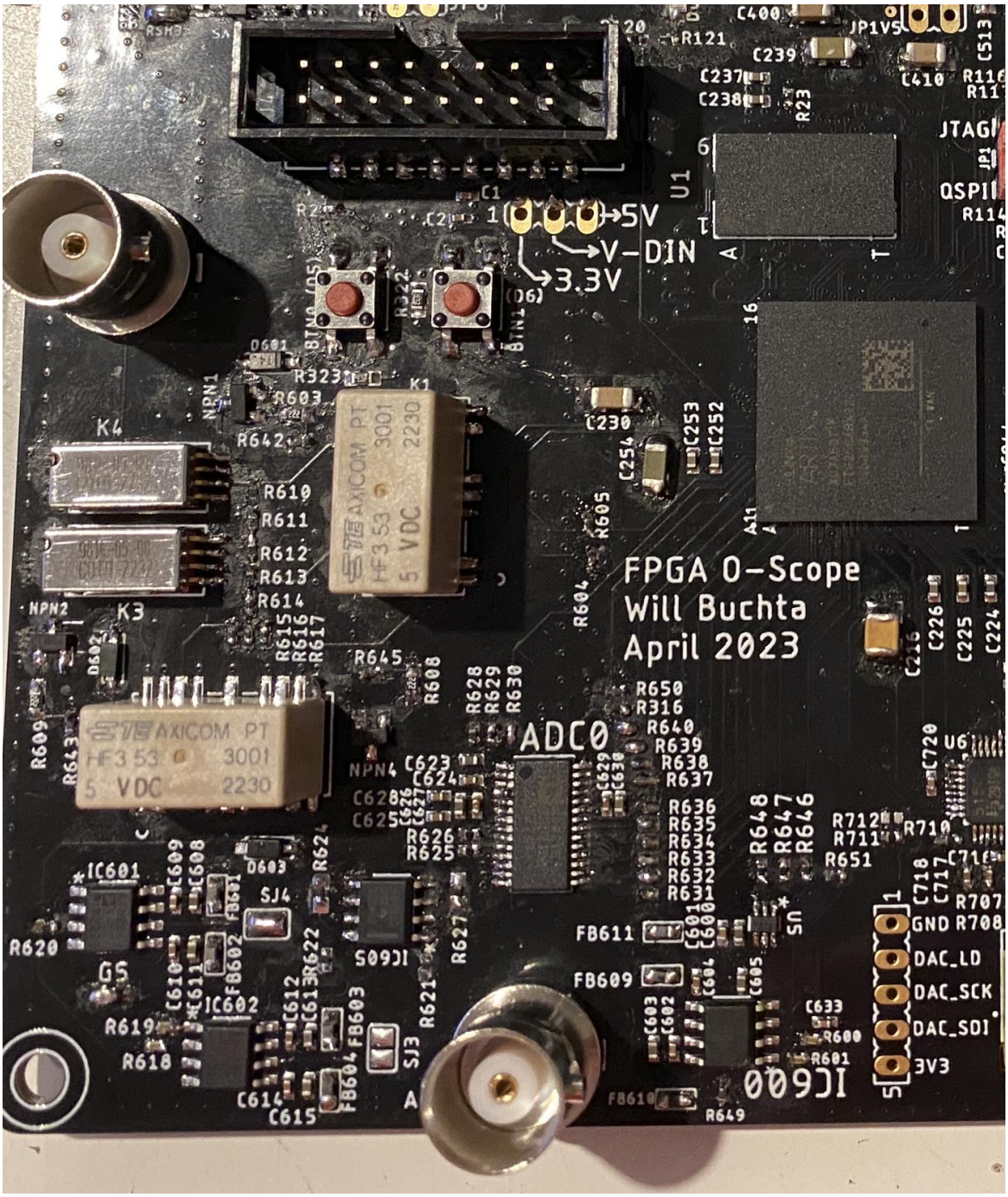


Figure 20: Assembled Analog Front end

Power Supply Scheme

With the large majority of the board designed, it was time to design the power supply. This was no easy task; as mentioned previously, there are 10 different power rails. The FPGA itself needs 3.3V for I/O, 1.5V for DDR, 1.8V for internal logic, and 1.0V for the core. Various peripherals require 1.8V at low noise. An analog 3.3V is required for the ADC, as well as +/-4.5V for the op amps in the analog front end. To generate the low noise -4.5V, an additional -5V rail was necessary. All current consumptions were estimated, and *significant* overhead was added to the estimates.

To this end, all the digital power rails were supplied from the quad buck ADP5052. This is a very complicated PMIC with many independent variables to sort through when designing. Luckily, there were some recommended part numbers and values for components at various current consumptions and voltages. This IC also allowed for power supply sequencing, which is recommended by Xilinx; the core voltage (1.0V) should be brought up first, then auxiliary (1.8V), and finally IO supplies (1.5V, 3.3V). Each analog power rail was fed by an independent low noise linear dropout regulator (LDO). Every power rail has a shunt resistor in series with it to allow for easy measurement of current consumption.

To generate the -4.5V rail, an inverting regulator was first designed to provide -5V. This was then fed into a negative LDO, which provided a low noise rail.

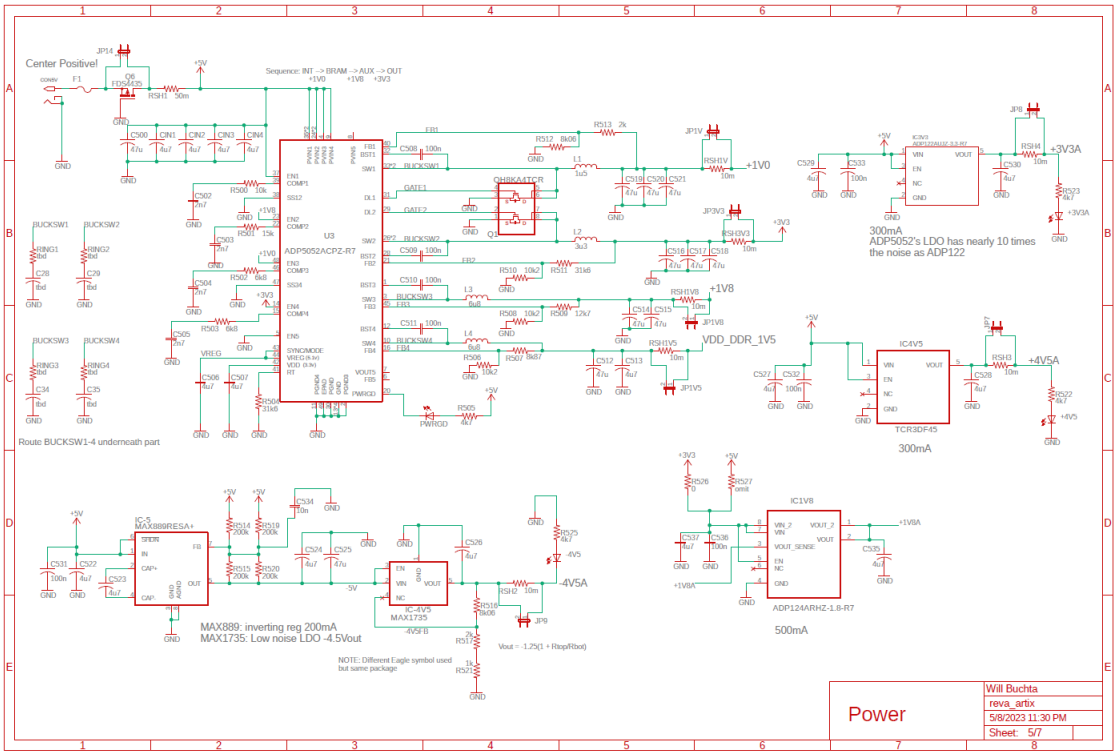


Figure 21: Power supplies

When performing layout, the analog and digital power rails were explicitly kept away from each other to eliminate any noise creeping in. In Figure 22, the analog portion is on the left.

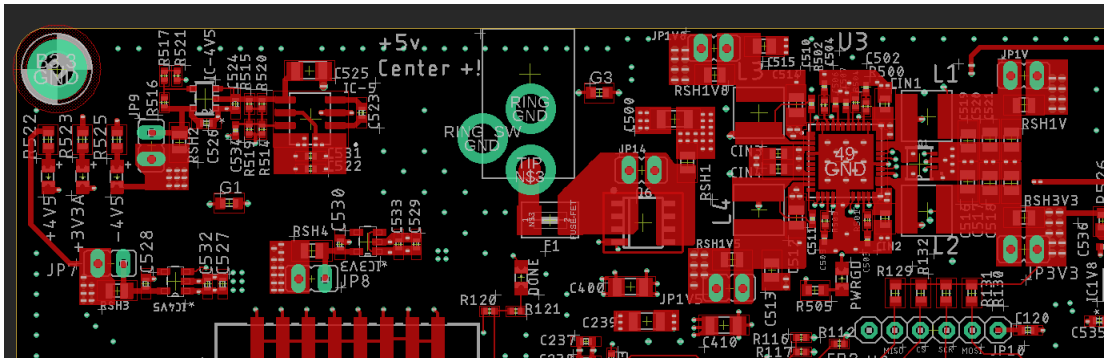


Figure 22: Power Supply layout

Also included are several LEDs indicating if / which power rails are functional. Unfortunately, I misinterpreted the operation of the PGOOD pin on the ADP5052, leading to some post-fabrication rewiring:

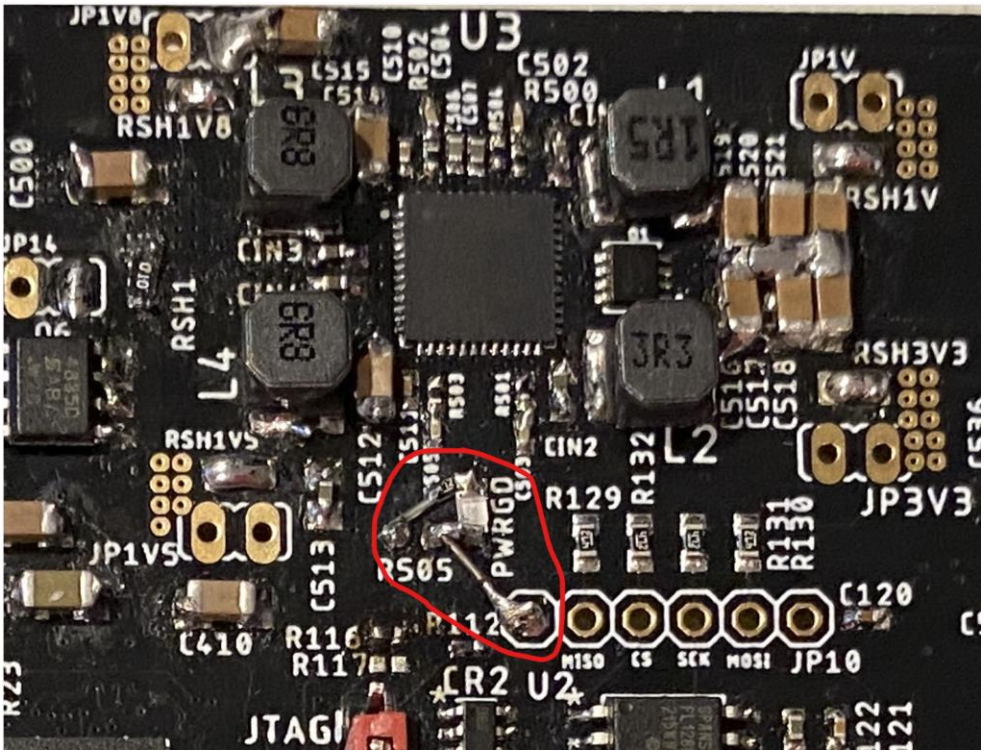


Figure 23: Fudge wires

There were several ideas to keep in mind while performing layout, each of which form the basis for an entire article or application note. Among other app notes, Phil's Lab on YouTube provides a great video detailing the design and layout considerations of switching regulators [12]. The ADP5052 also provides an example layout in the datasheet [13].

As shown in Figure 6, an entire layer is dedicated to power delivery. This is shown in Figure 24 below:

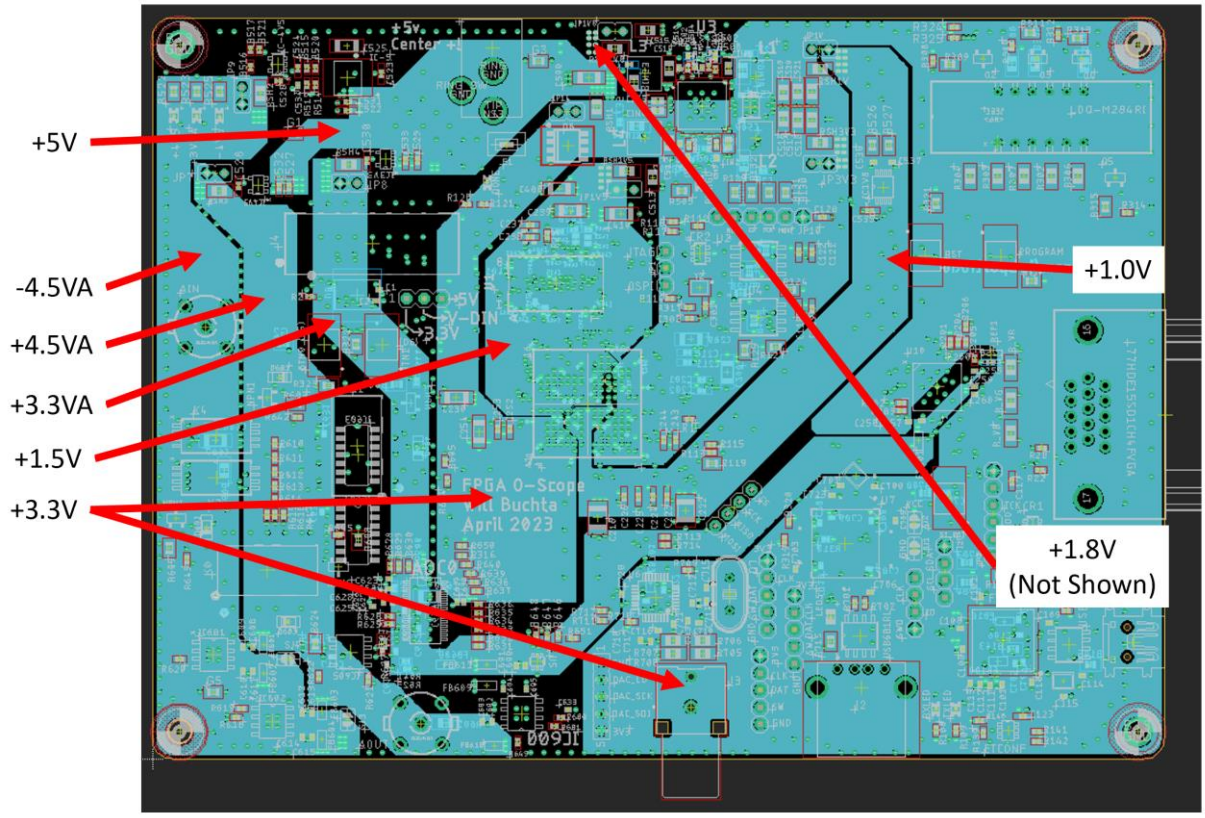


Figure 24: Power delivery layer

One feature to note is that for large boards, warping can occur during manufacturing. Since copper has different thermal expansion properties than the dielectric, it is important to have layers with copper pours cover symmetric portions of the layer. In the case of Figure 24, nearly the entire layer is filled in.

Miscellaneous Components

The last main sections of the board are the general FPGA-peripheral connections, STM32, and composite video decoder. There is not too much to go over; 90% of all peripherals end up in one of the three FPGA banks shown. Notable connections include the 100MHz source clock and 7-segment display capable of showing 16 bits of data.

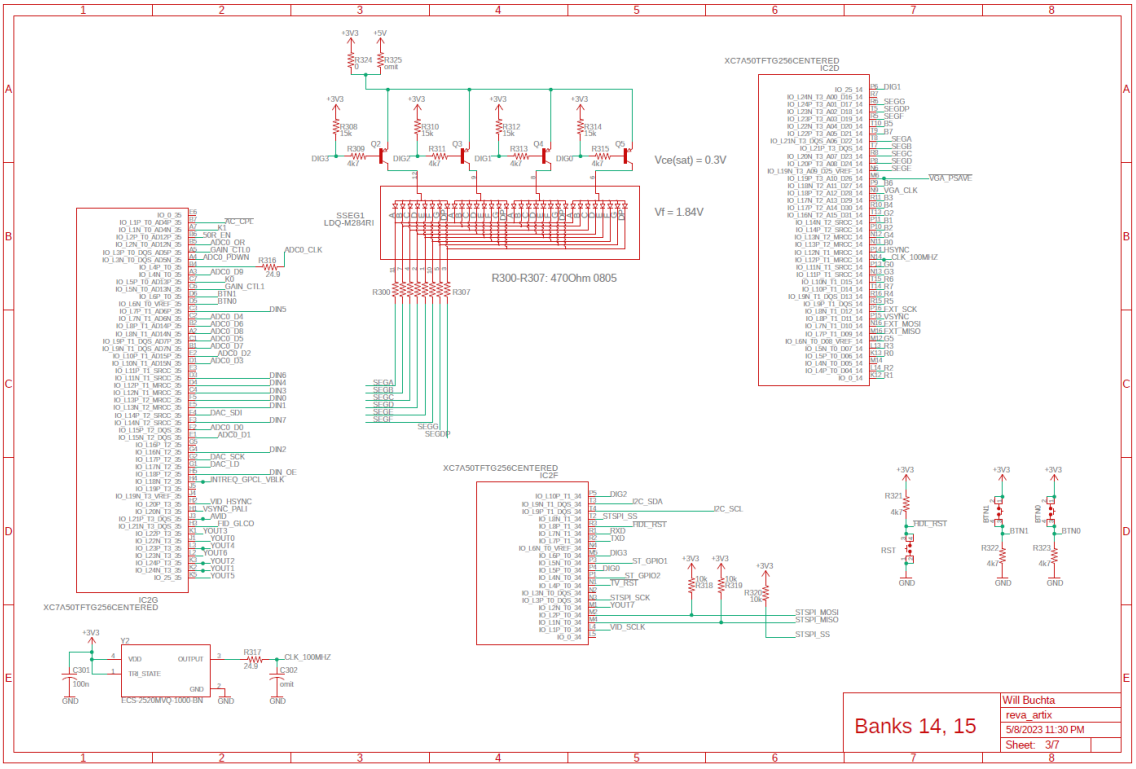


Figure 25: FPGA to peripheral connections

The last page of the schematic details a composite video decoder, STM32, and 8-bit level shifter as discussed in the analog front-end section. The composite video was a last-minute inclusion as I was curious about applying the board to be used as a composite-VGA converter. The STM32 is meant to be used as an interface to the real world; it can act as a USB controller, so a mouse or keyboard could be attached to control various settings on the oscilloscope. Unfortunately, I have not gotten to implement either the STM32 or decoder in software due to the inherent complexity. If I could, I would split this task off to another engineer who could spend more time on it.

The STM32 line of microcontrollers is powerful and comes in a vast variety of form factors and functionality. There is very little boilerplate associated with them, as just a reset button, external crystal (optional), boot mode configuration, and debug interface is needed.

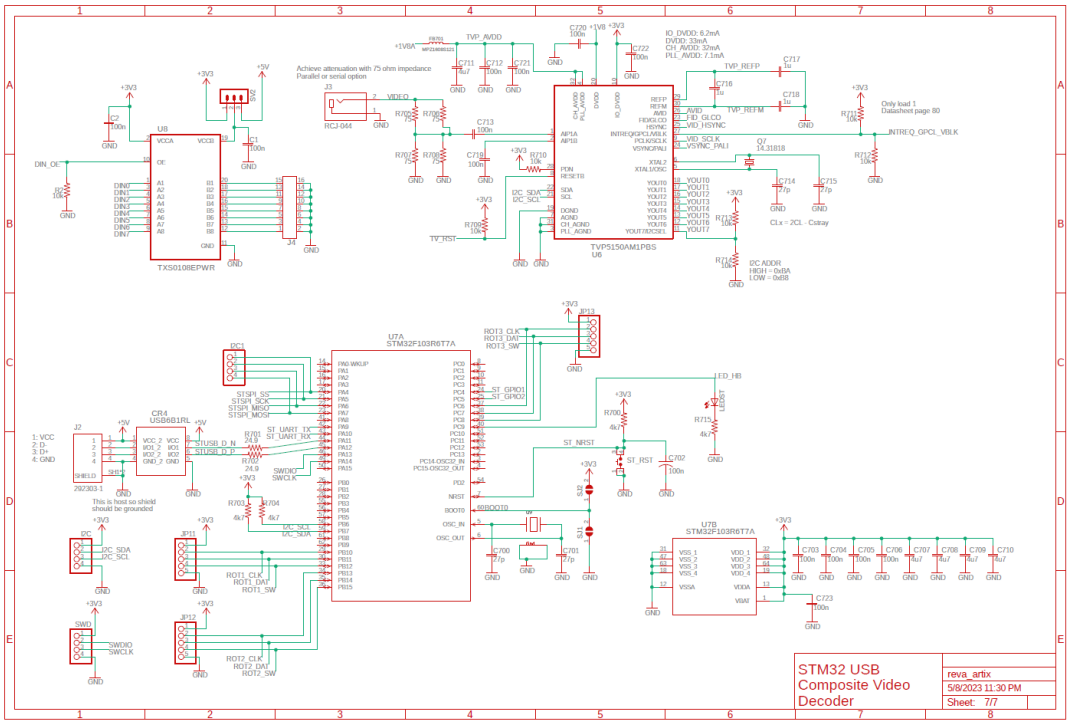


Figure 26: STM32, Composite video decoder, and level shifter.

Assembly

Once I had completed the board layout, I deliberately waited a week before placing the fabrication order. This was to make sure I wouldn't think of anything new to add, and to fix any bugs that I thought of in the meantime. The quote I got for 10 PCBs was several hundred dollars, so a re-spin would be quite unpleasant. Once I was certain the design was ready, I shipped the CAD data overseas to PCBWay in Shenzhen. The boards arrived a few weeks later; it was very satisfying to see them in person after several months of design.

The next challenge was to get all the parts on the board; I have lots of experience soldering, but there was no chance I would be able to get the DDR and FPGA done by hand. Several other challenges arose as well, such as soldering 0201 capacitors and hundreds of 0402 components. Therefore, I reached out to several local PCB assembly houses (Greater Boston area) for quotes. Unfortunately, since I only planned on getting one prototype board done, almost the entire cost to assemble was in setup charges and up-front costs. It was not feasible to get every part assembled, as the BOM was just over 100 unique components. I settled for a reduced assembly BOM, comprising of parts that I was not able to do myself. This included packages with hidden pins, no pins, parts with exposed pads, and parts with many fine-pitch pins. I also elected to get the 0201 capacitors assembled, 100nF capacitors, and other common value components that had a significant quantity. In the end, this cut the cost down by roughly half.

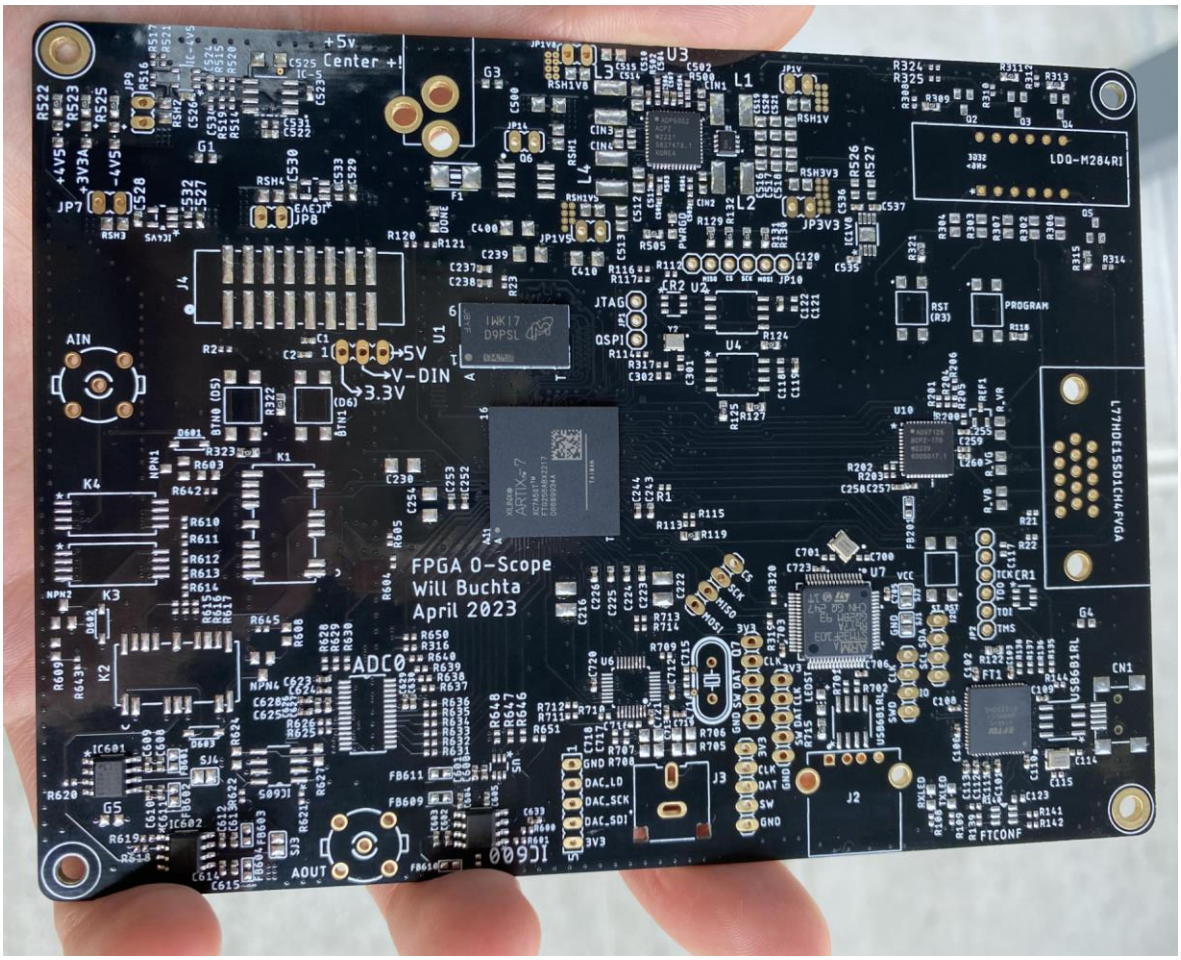


Figure 27: Partially assembled board

One unfortunate byproduct of the partial-assembly process was that every pad on board now had solder on it, as seen in Figure 27. This meant that for every component I was going to hand solder, I first had to wick any solder off – a very tedious process.

The rest of the assembly process took approximately 20 hours of work. Figure 28 is the fully assembled board.

The next crucial step in the process was hardware bring up. This included several simple tests of basic functionality of various peripherals. The very first step was to power the board and verify all power supplies were up and running. This was by far the most stressful, as a short could take several hours to find and shake out and could possibly render the board useless. Thankfully, all rails were operational and within tolerance – a huge relief.

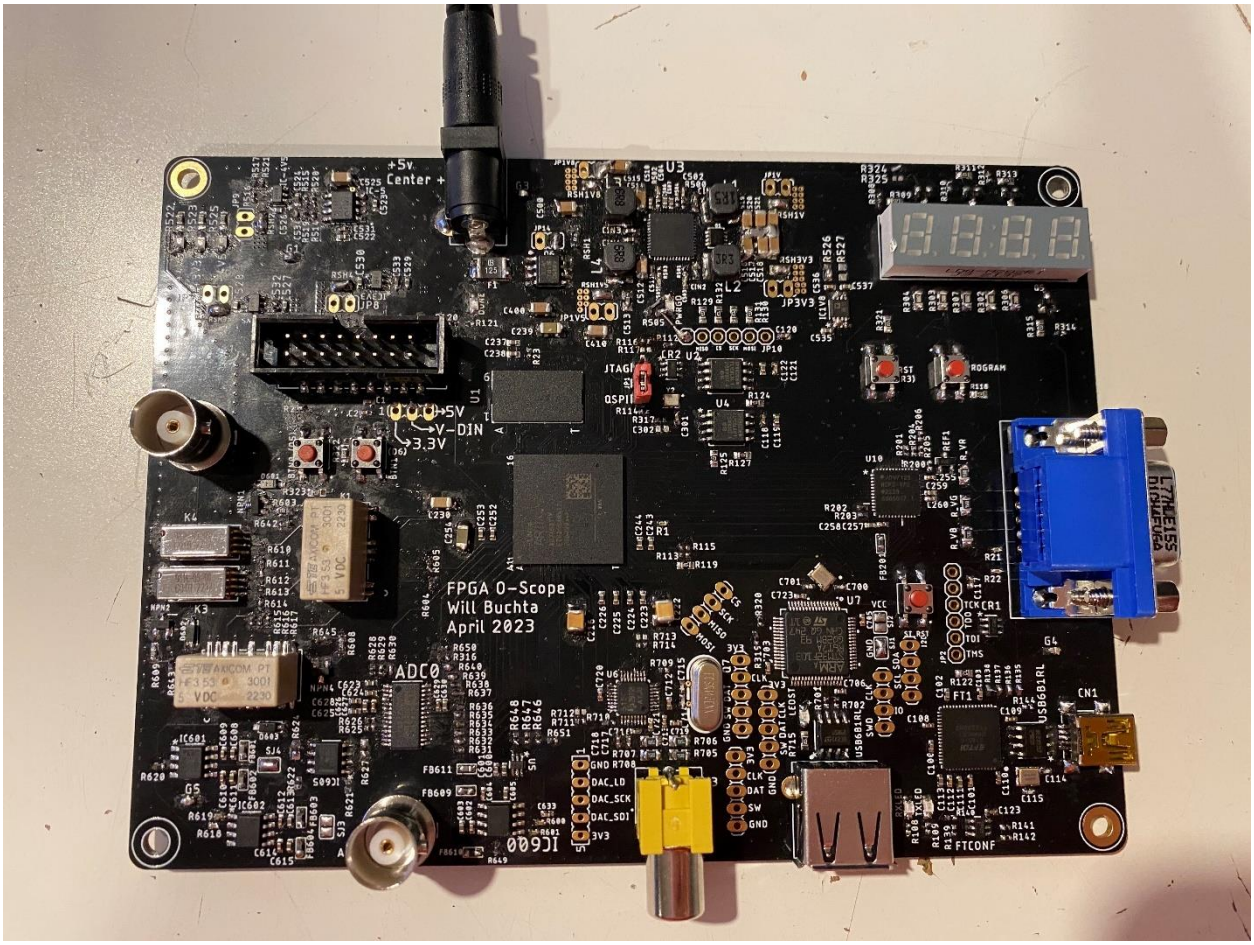


Figure 28: Fully Assembled board

The first FPGA test that I wanted to run was to get the seven-segment display working. This would demonstrate USB-JTAG functionality, FPGA configuration and flash functionality, FPGA core, clock, and I/O functionality, and of course the display functionality. I created a very simple design in Verilog to control the multiplexing of digits to show a given hexadecimal number.

To use the USB-JTAG functionality of the FT2232HQ, I had to flash its configuration EEPROM with the FT_PROG tool, available from their website [14].

When plugged in, Vivado recognized the device, and I was able to load the program into the FPGA. Pictured below is the number 0x6EEF being displayed. This was also a huge relief, as many of the core functionalities were verified from the test.

I also created a simple VGA timing controller, which output solid color bars at a 1080p resolution.

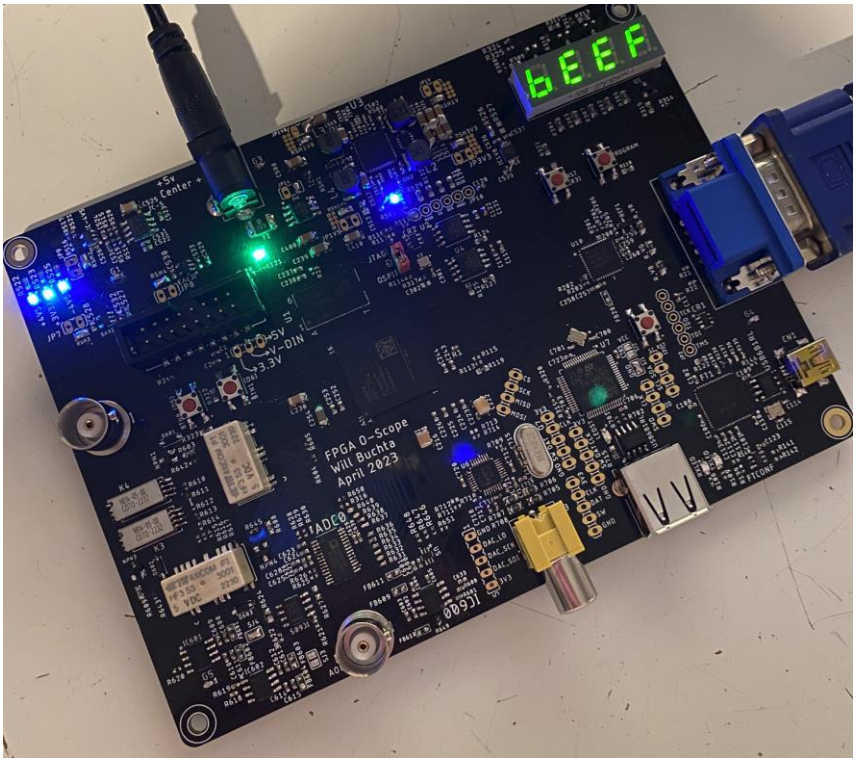


Figure 29: 7 segment display working

AXI Lite 7-Segment Module

Before moving on to big block-diagram based designs, I needed to learn how to write modules with an AXI interface. Understanding the basics of this protocol was necessary to use custom hardware in a Microblaze based design. The simplest way that I saw to do this was with the seven-segment display – a simple memory mapped register would control the value shown.

Vivado provides a tool to create and customize AXI based modules with pre-generated boiler plate code, but naturally, the integration of this tool into actual designs is broken and has been for several years. After thorough forum searching, it was discovered that a workaround to this was to copy and paste the boiler plate AXI modules into the current working project and edit them directly, instead of in the “Create and package new IP” wizard.

The design itself is straightforward; there is a top module which can house multiple AXI interfaces, and lower-level hardware implementation modules for each AXI interface. I implemented my Verilog for the display inside the lower level AXI Lite transaction module. See Figure 30 for a rough block diagram. The point of Figure 30 is not to show the actual implementation of the AXI-Lite protocol; it’s meant to show the general idea of the module. Circled on the left and right are the inputs and outputs for the interface. There are 4 actual RTL data registers along with one address register. The output from the 0^{th} data register’s lower 16 bits feeds into the seven-segment display module, as outlined.

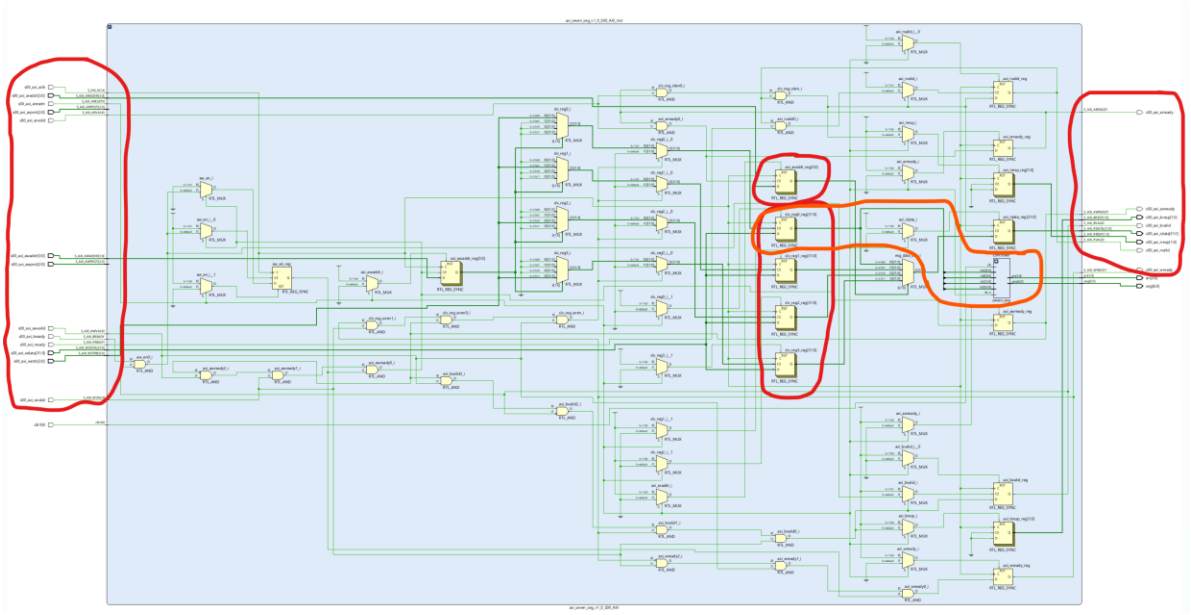


Figure 30: First Axi-Lite module

When programming, setting the value of the display is as simple as dereferencing a pointer; all registers are hardware memory mapped. I did not write a testbench for this module as verification was straightforward.

Digital System Design

DDR3 Verification

The next immediate area to verify was DDR3. Ideally, the FPGA would get samples from the ADC, store them in memory, process the data, then update the display output. To verify the memory, I created a simple Microblaze block diagram (Figure 31) with the bare-minimum necessary to boot. This included clock and reset inputs, MIG, UART (through the FT2232, without needing to disconnect JTAG – a very nice feature), and a simple AXI-Lite interface for the seven-segment display. This Vivado project became the basis for all future projects; custom hardware or other IP could be added easily. I created an HDL wrapper of the design, synthesized it, and exported the bitstream to Vitis (as well as the hardware description file, .xsa).

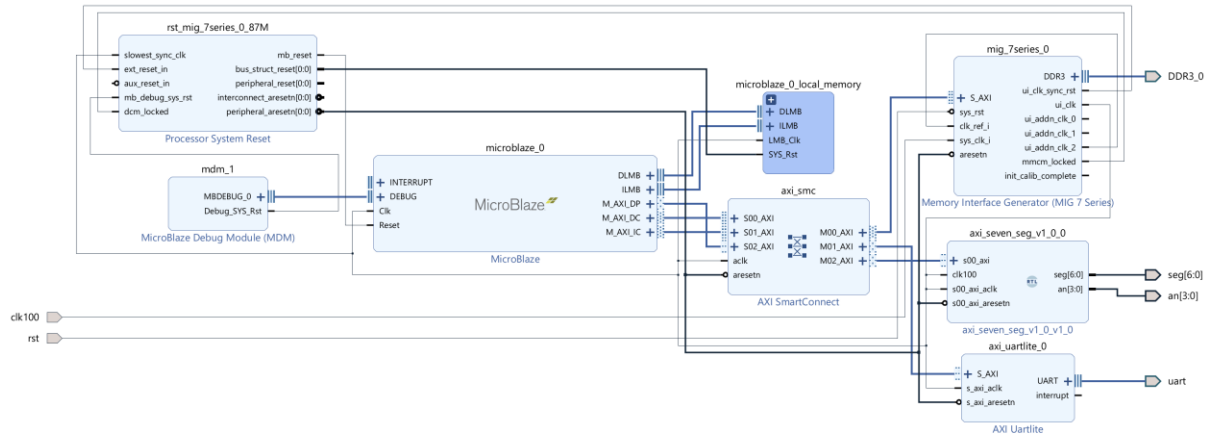


Figure 31: block design for DDR3 verification. Basis for all Microblaze designs

One of the options when creating a new Vitis project is a program that tests various memory regions present in the hardware – including the MIG. I generated the memory test code, compiled, downloaded to the FPGA, and got positive results indicating the DDR was functional. This was also a relief, given that the DDR took the longest to design and had the highest likelihood of error.

Video Test Pattern Generator

Another major area I wanted to verify before diving into data acquisition was the video IP blocks provided by Xilinx. To get started, I searched through forums and online tutorials. Xilinx provides blocks for video timing signal generation, AXI Stream to video, and AXI Stream video generation modules. All three of these modules were placed into their own clock domain with the addition of an MMCM/PLL Clock Wizard IP. This did create the issue of crossing clock domains when programming, but the Xilinx AXI-SmartConnect IP took care of this.

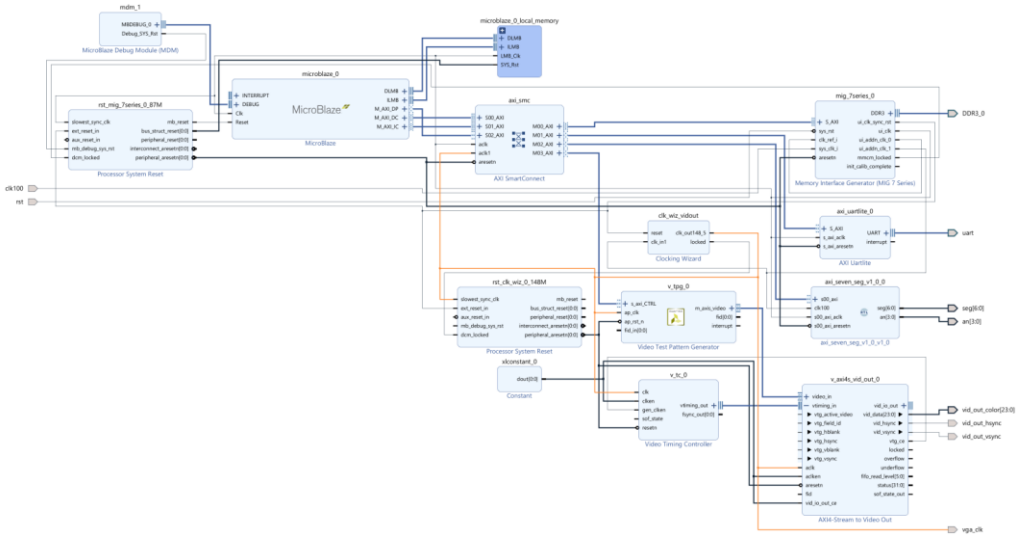


Figure 32: Video generation using Xilinx IP

Again, an HDL wrapper was created, the project was exported to Vitis, and a basic example program was run to generate color bars.

Data Acquisition IP: Interfacing with the AD9215

When scoping out this project, I underestimated the complexity and challenge the data acquisition hardware layer would pose. I have gone through multiple different iterations of prototypes, beginning with layers of hierarchical state machines, to AXI-Lite memory mapped ADC samples, to an AXI-Stream (AXIS) module with AXI-Lite configuration and control. The final design that I settled on (after about half a dozen other attempts) has everything controlled by memory mapped registers. The module I designed had to control all relays, op amps, and the ADC. Based on the current state of relays and gain stage, the module also had to convert the samples into floating point numbers, as opposed to in software (the hardware performs the following: V_{sample} = (Bits/512)*attenuation. A high-level block diagram of the module is shown in Figure 33:

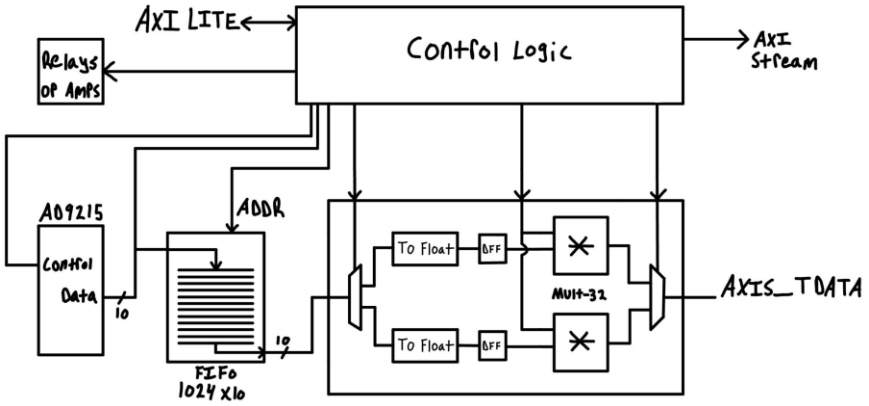


Figure 33: Data Acquisition module

The FIFO, control logic, and float conversion modules were written from scratch. The 32-bit hardware multiplier was modified from an online repository [15].

The FIFO (`reg [9:0] fifoMem [0:1023]`) is implemented using block rams, which are good for narrow, deep arrays. There are several different registers used as pointers into this memory to implement FIFO functionality.

The control logic is in charge of triggering, controlling attenuation, writing samples to the FIFO, advancing the sample-to-voltage conversion, and communication with a higher level AXI master (like a Microblaze). It is also in charge of monitoring the 'OR' signal from the ADC, which indicates if the input has exceeded the current input range. If 'OR' is ever high, the attenuation is switched to x20 in an attempt to prevent damage to the ADC. The general flow of data in the system is as follows:

1. When signaled from software, the FIFO will begin reading samples. The module will not be able to monitor for a trigger until after the first half of the FIFO is full; the idea is to have the 512th sample be at the trigger index.
2. Once at least 512 samples have been recorded, the trigger functionality will be enabled. Three trigger modes are supported: rising edge, falling edge, and auto. If in auto trigger, the module will proceed to step 3. Otherwise, the module will continually monitor the current and previous samples to determine if an edge has occurred. The trigger level is set in software as actual bits, calculated based on the attenuation.
3. After a trigger event, the next 512 samples will be recorded in the FIFO. Once finished, the AXI Stream portion is 'primed' to account for pipelining delays. The module will then set a flag indicating that is has finished sampling and is ready to be streamed.
4. The AXIS master logic will then spit out 32-bit floating-point samples, 1 per clock cycle.

An extensive test bench was created for this module. See below for an example run:

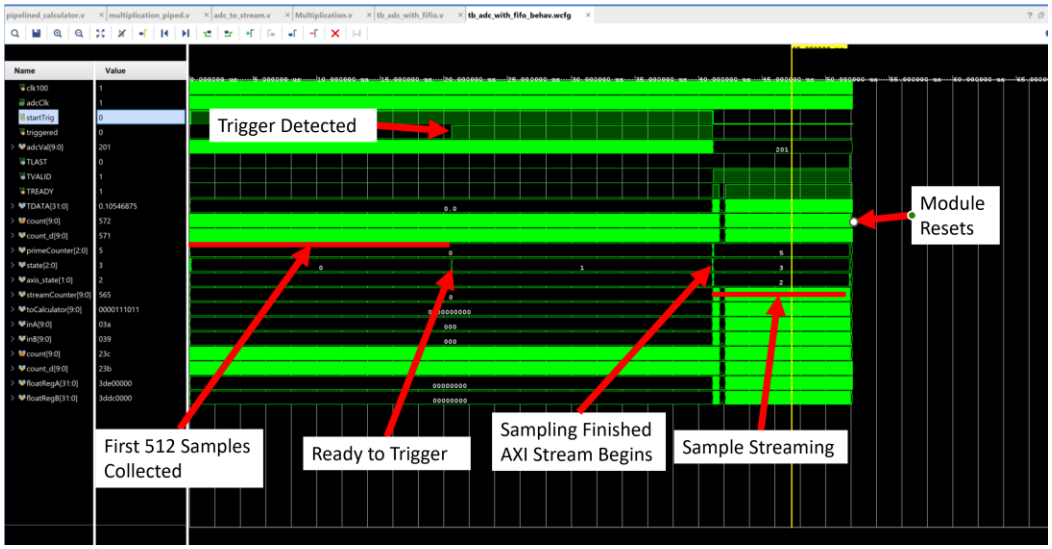


Figure 34: Test Bench of module

Another important feature that I implemented was that of scaling the ADC sampling frequency. At 100MSPS, a 1024 sample record length only allows for 10.24 microseconds to be recorded. The ADC recommends a minimum sample rate of 5MSPS (I assume that as it is pipelined with sample/hold circuitry, a lower frequency could result in lost charge and error), so logic was implemented to allow a clock rate divider anywhere from 1-20 (from the 100MHz base clock). A feature to add or modify in the future would be to allow for "lower" sample rates by only recording every n^{th} sample.

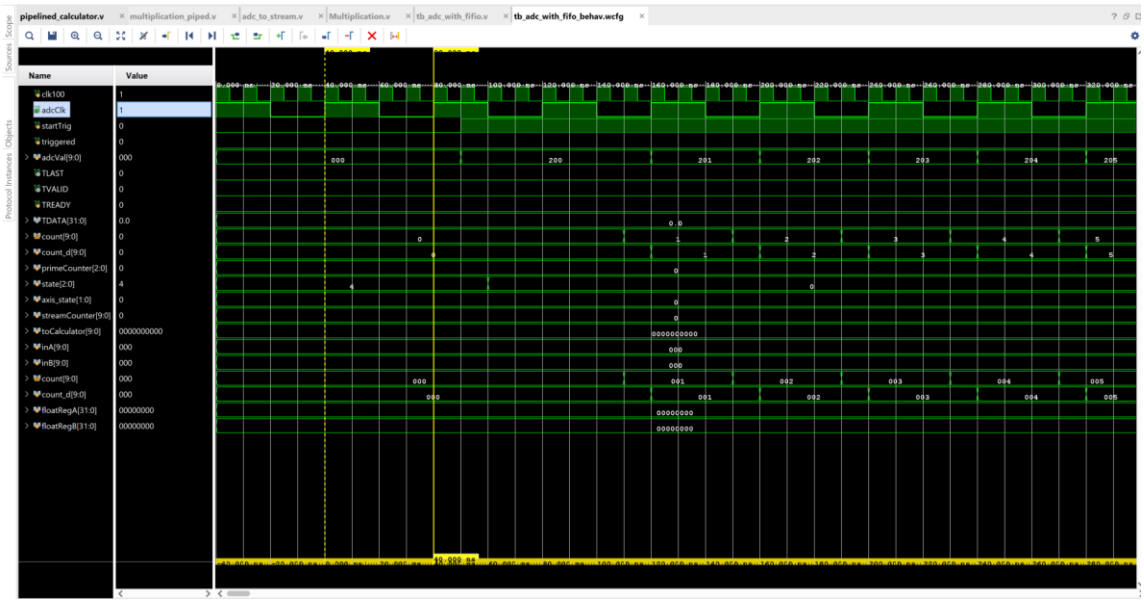


Figure 35: An example of running the ADC clock at a lower frequency (25MHz)

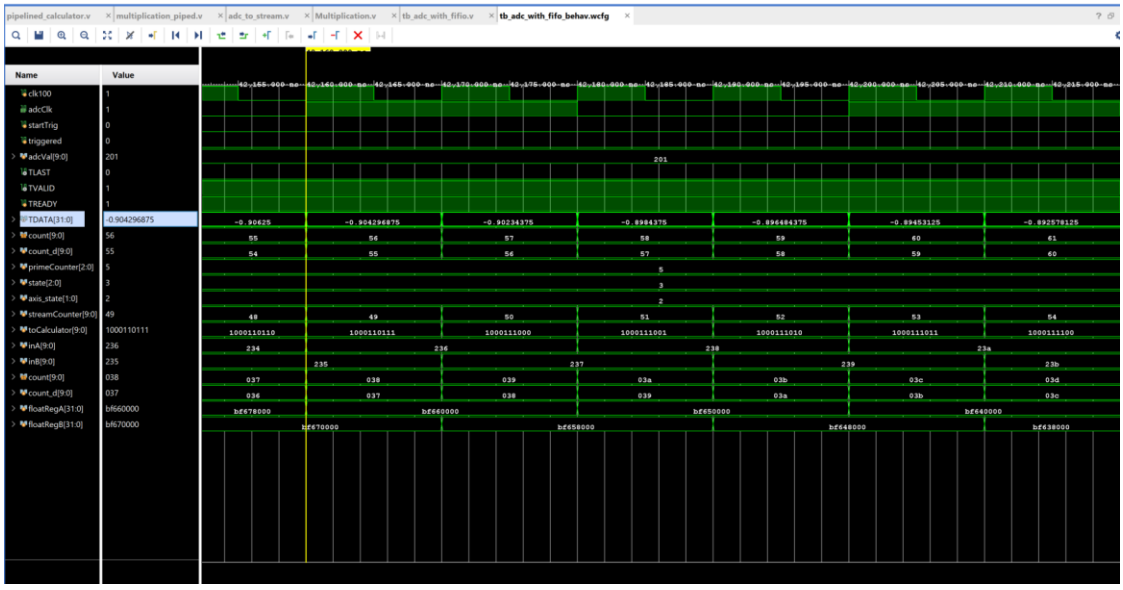


Figure 36: Floating point values being passed through TDATA of AXI Stream

The idea is to then have the AXIS output of this module feed into a DMA controller, which would move samples from the FIFO into DDR3. The whole block diagram of the digital system is shown below. Video DMA has been added to interface between the DDR3 and AXIS to video module.

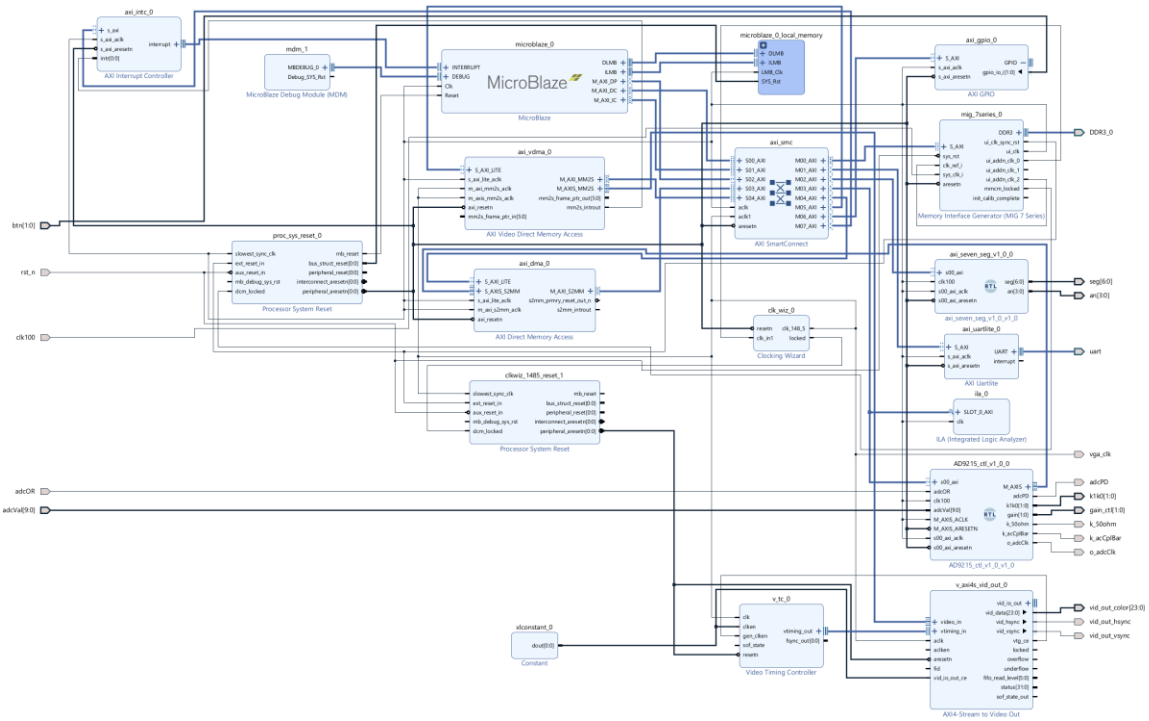


Figure 37: Complete Digital System

The state of this project is currently trying to implement sample gathering functionality; I am facing several small hardware bugs such as variations of when it is possible to start sampling, issues reading registers while debugging, and documentation trouble.

Conclusion

This project has been fun. I started out with the goal of creating custom FPGA PCB designs to implement certain tools, such as a basic digital oscilloscope. Throughout the past year, I learned advanced PCB design topics, the general flow of complex projects, and an idea of how much time and effort each stage of a project requires. I gained more experience with Verilog, AXI Stream and AXI Lite protocols, plus the design of FIFOs and hardware/software codesign. Looking back, the fact that the board even turned on without shorting constituted a victory. This project has also given real world experience such as learning the manufacturing process of making electronic products, and tradeoffs that need to be made for cost. In the end, talking about this board was a big topic during my interviews for a hardware internship at Nvidia, who ended up giving me an offer. One general goal I have when working on any project is to make the next one more complex and sophisticated than the previous; this eventually leads to very complicated and impressive designs.

Bibliography

[1] https://www.protoexpress.com/blog/current-return-path-signal-integrity/

[2] NXP AN2582: https://www.nxp.com/docs/en/application-note/AN2582.pdf

[3] Saturn PCB: https://saturnpcb.com/saturn-pcb-toolkit/

[4] UG483: https://docs.xilinx.com/v/u/en-US/ug483_7Series_PCB

[5] UG1099: https://docs.xilinx.com/r/en-US/ug1099-bga-device-design-rules

[6] UG470: https://docs.xilinx.com/v/u/en-US/ug470_7Series_Config

[8] FT2232 datasheet: https://ftdichip.com/wp-content/uploads/2020/07/DS_FT2232H.pdf

[9] UG586: https://docs.xilinx.com/v/u/en-US/ug586_7Series_MIS

[10] ADV7125 Datasheet: https://www.analog.com/media/en/technical-documentation/data-sheets/adv7125.pdf

[11] AD9215 datasheet: https://www.analog.com/media/en/technical-documentation/data-sheets/AD9215.pdf

[12] https://www.youtube.com/watch?v=AmfLhT5SntE

[13] ADP5052 datasheet: https://www.analog.com/media/en/technical-documentation/data-sheets/adp5052.pdf

[14] FT_PROG: https://ftdichip.com/utilities/

[15] Floating point multiplication in Verilog: https://github.com/nishthaparashar/Floating-Point-ALU-in-Verilog/tree/master