Final Project Report

PYNQ-182 - Music Visualization in Hardware

Karl Swanson & Peter Albanese

Project Description

The concept of music visualization dates back to 1590, in which poet Gregorio Comanini described the works of Renaissance artist Gieuseppe Arcimboldo in creating a “Color Organ”[2]. This original Color Organ was a massive hydro-mechanically powered musical instrument which would display various colors by moving curtains over colored glass panes[1].

Today, color organs are an everyday commodity, from a basic search on Amazon, it is easy to find light and color organ kits within a suitable price range. Color organs, or music visualizers as we will denote them in this report are most often seen in concerts, dance halls, or clubs. These visualizers however are far more expensive than your average Amazon kit. Professional music visualizers utilize more complicated algorithms to analyze music frequency and display interesting patterns. These professional setups can cost upwards of $100,000 for a customizable LED matrix, crazy!

Our project will explore the feasibility of implementing a Music Visualizer on the PYNQ Z2. We believe that by implementing the more complicated music visualization algorithms in hardware, such as FFT (Fast Fourier Transform) for frequency analysis, bandpass filtering, and moving average filters, the cost of a professional system could be dramatically reduced.

The major focus of our project is the analysis of FFT in hardware versus software. From initial research, we found an FFT implementation already available as IP on the PYNQ board, which will be utilized for our project.

Progress Report

PL Implementation /HLS

Initially, we intended to implement a FFT algorithm in High Level Synthesis (HLS) ourselves, and set to accomplishing this task. Following research on FFT, we pursued development using the Cooley-Tukey algorithm. The Cooley-Tukey algorithm is one of the most common FFT algorithms used today, and makes use of a divide and conquer approach which completes a FFT using recursion[3]. This rather clever approach splits the initial “problem”, in our case a 1024-byte array of uint8_t values into subcases. The subcases are split on even and odd values of the array.

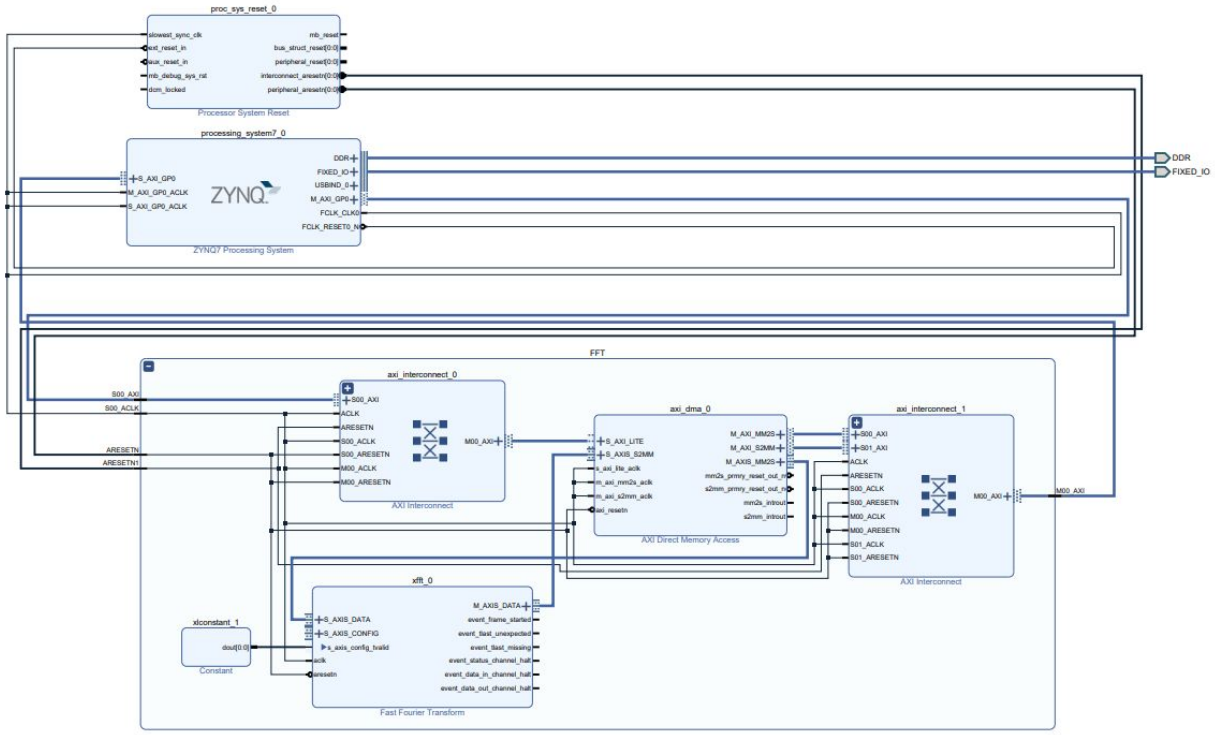

Figure 1 - FFT With AXI DMA and Interconnects

It seemed that this algorithm was the perfect approach to obtaining frequency values of an audio sample. Given our determined constraints, we knew that we would always feed an audio buffer of 1024 bytes. Given the divide and conquer nature of the Cooley-Tukey algorithm, one of the major restrictions is that it is limited to inputs that are a power of two. So our constraints fit the algorithm perfectly. Following our implementation of the algorithm in HLS we created and ran a test bench to validate the algorithm, and everything worked as intended. Unfortunately, once we went to synthesize the C code we hit a major roadblock; HLS cannot synthesize recursive code.

Given this roadblock, we attempted to move to an iterative approach to solve the FFT problem, but struggled in translating the Cooley-Tukey from recursive code. In another attempt we looked into a simple discrete Fourier transform utilizing a Left Hand Riemann Sum algorithm for integration. While working on the implementation of the discrete Fourier transform, we discovered that the Vivado suite includes it's own FFT IP block. Moving forward we decided to use the provided block, as it is fully optimized for hardware.

That being said, we still wanted to do some sort of HLS exploration for our project, so we decided to implement an FFT algorithm described in the class book. The algorithm described there took some time to get working with the correct pragmas, and is wildly inefficient compared to the IP block provided by Vivado, but it was interesting to optimize. We performed various

optimizations on the code including unrolling and pipelining, and in the end determined that the best optimization involved pipelining the inner loop of our FFT. The inner loop of our FFT code is responsible for calculating the phasor values of the FFT with sine and cosine functions. We have included our pareto graph and table below describing our optimization attempts.

Algorithm	Latency	FPGA Area	DSP48E	FF	LUT
None	222338	25934	75	12013	18434
Pipeline first outer loop only	4342	121064	879	32996	33164
Pipeline last loop only	210000	25960	75	12050	18460
Pipeline inner loop	15618	16810	49	8333	11910
Pipeline inner loop, unroll outer loop	14730	75917	410	33913	34917
Pipeline inner loop, unroll outer loop 32	15620	49354	229	23844	26454

Figure 2 - Optimization Table

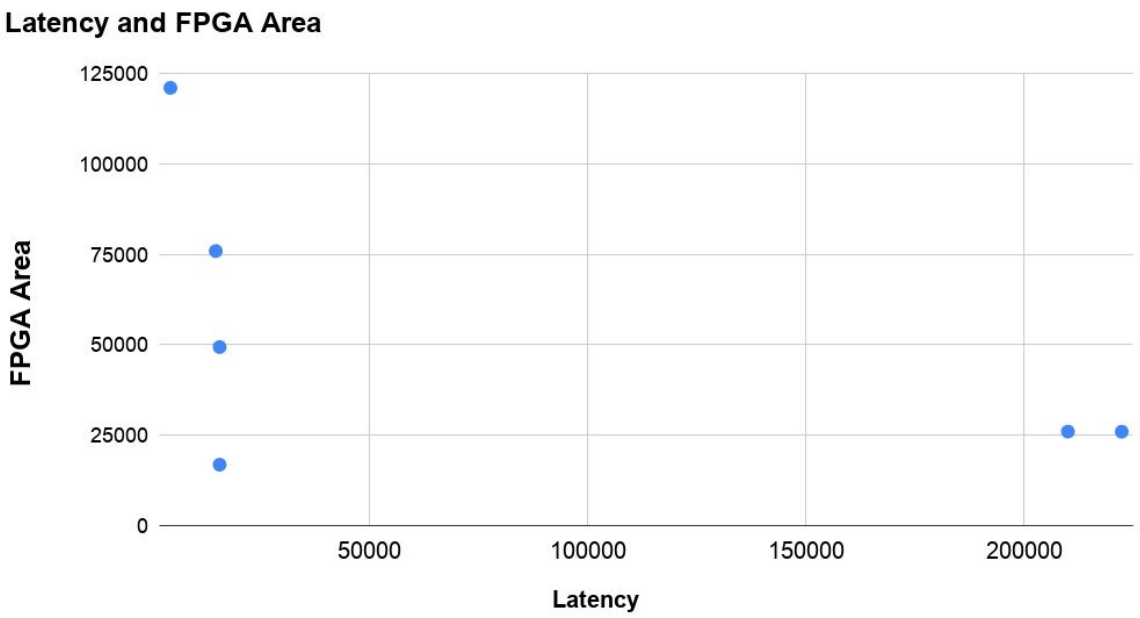

Figure 3 - Pareto Graph

Our work with the Vivado FFT IP had some issues with regards to DMA access. We have been in contact with the TA Vicky, attempting to resolve the issue. For the longest time, we would

receive the following error when attempting to access DMA “RuntimeWarning: coroutine '_DMAChannel.wait_async' was never awaited after removing the cwd from sys.path.”. This may have been due to how we were setting our configuration constant. After fixing the configuration of the FFT IP block we were able to get this working.

Upon further experimentation with the FFT IP, we made good progress with the DMA data transfer. Firstly, we discovered an issue with our ARESET lines in the block diagram where the lines for interconnects and peripherals were swapped. Fixing this issue solved the program hanging issue when we ran the overlay in Jupyter Notebooks. Now that we could run the overlay without errors and transfer data, we looked deeper into the FFT IP block and its available options. From there, we investigated the TDATA format for sending data over the DMA. Since FFTs operate on complex numbers (real and imaginary parts), we had to use this data format to stream our values into the hardware.

Field Name	Width	Padded	Description
XN_RE	b_{xn}	Yes	Real component (b_{xn} = 8 - 34) in twos complement or single precision floating-point format.
XN_IM	b_{xn}	Yes	Imaginary component (b_{xn} = 8 - 34) in twos complement or single precision floating-point format.

Table 1 - TDATA Real and Imaginary Parts

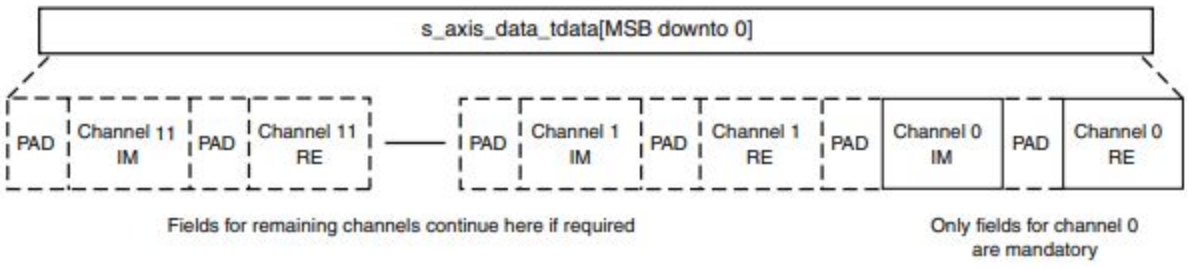

Figure 4 - TDATA Expected Format

While this format was necessary for us to transfer the correct data to and from the FFT, it was extremely difficult to work with. Since Python is an untyped language, it becomes very difficult to move data to a desired format. Whereas C or C++ allows casting and provides the tools to easily manipulate data at the bit level, Python’s higher level approach often hides important low-level content like this. The input buffer was expecting an alternating cycle of imaginary and real numbers for each data frame. Each number was to be a 12 bit integer, padded to 16 bits. Thus making each frame (imaginary and real combined) 32 bits in total. Since our input to the FFT was audio frames from a .wav file, all of the values were real numbers. As such, we had to create an input buffer with alternating real frame data and zeros for the imaginary portions. All of this data needed to be 12 bits, padded to 16. This was accomplished with some convoluted numpy array manipulation, but ultimately worked.

Now that this was functional, we took a look at some of the FFT IP options to modify. We discovered that we could use multiple channels to send and receive data, which would greatly speed up the data transfer operation. This, however, would require a fair bit more formatting to get the data in the correct orientation to be processed by the FFT block. Since this is not as much of a hardware issue as a software issue, we decided to forgo this modification for now. Adding multiple channels could result in a theoretical speed up to 12 times faster than the current transfer. There is definitely room for improvement here, and we may choose to look into this if we continue work on this project in the future. Additionally, we looked into the array input size for the FFT block. Currently, we are using a size of 1024 for the input, but there are options to process much larger sets of data all at once. This was quickly shot down due to the limitations of the DMA transfer, which is limited to a maximum size of 1024 for one transfer. Since we would still have to do a large number of transfers for each calculation, the gain in speed from this change would be marginal.

Once all of this was sorted and functional, we ran into another issue which some of the other groups had seen in the past. It is not normally possible to use the GPIO outputs (which we need to control our LED strip) while using another overlay on the PYNQ board. To solve this, we found a tutorial for rebuilding the PYNQ base overlay [12]. Following these guidelines, we were able to extend the base overlay to include our FFT IP so that we could still use the GPIO pins while also transferring data to and from our FFT in hardware. This was accomplished by first cloning the repository for the base overlay.

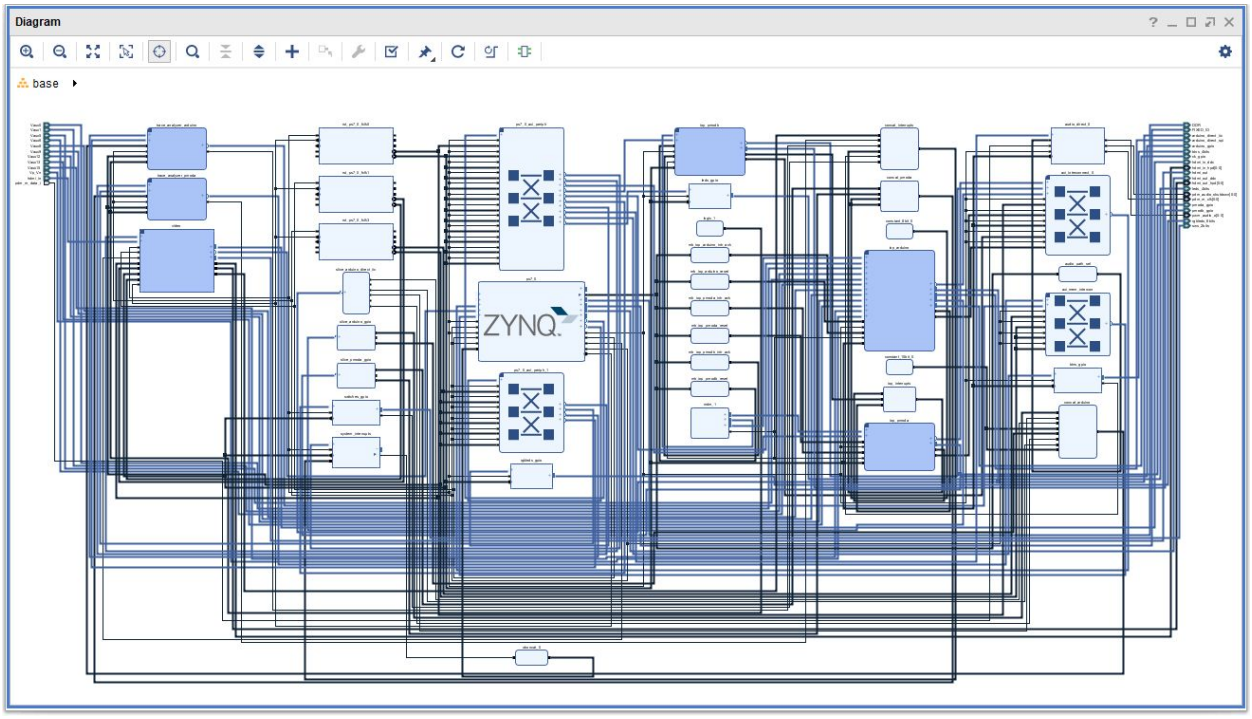

Figure 5 - Base Overlay Block Design

The .tcl files provided in the base overlay were then used as sources to set up the project in Vivado and create the block design. From here, we were able to create and implement our own FFT module to drop into the design. The program could then be compiled into a bitstream which provided us with the new base overlay files to run on the board. To interact with the new IP in the updated base overlay, we could either write a custom driver for the board or simply write to specific registers to utilize the new FFT block. After going through these steps, we were unfortunately unable to get our new block working in the modified base overlay on the board. We did not want to go too far beyond this level of editing the base overlay for fear of messing up the software on the board and not being able to run our other code. Ultimately, this prevented us from running our full system (with the LED strip) together. Our PL FFT is functional, the data can be displayed via the matplot graph in software, and our LED display system is functional, with the input data being processed strictly in software, but the combination unfortunately does not work.

PS Implementation

Our original plan for the PS implementation was to stream audio input from a microphone via the LINE IN port on the PYNQ Z2 board, process it with an FFT, map the frequencies to colors, and display the colors on a matplot graph. This plan remained mostly unchanged, aside from the audio input.

Once we got to work with the PYNQ Z2 board, we quickly realized that streaming audio into the device was not supported by the base overlay. The audio library offers a direct stream from audio input to output, but does not provide access to the data as it passes through the device. To remedy this, we looked into external libraries created by other PYNQ Z2 users. The issue with this, however, was that they had to overwrite some base library files. Since they were created by unverified sources and did not have concrete version information, we did not want to potentially ruin the base overlay files and struggle to reinstate them from home. As such, we simplified the audio problem to recording data as a wav file, and processing the recorded data. Additionally, through testing we discovered that the LINE IN port did not support microphone input. It can only be used for input from a direct auxiliary source (e.g. a cell phone playing music through the headphone jack). The HP MIC port was found to work with our headphones and microphone, so we moved to using that port instead.

Despite our ability to now record our own test data for the system, we ran into another issue with processing the saved audio files. When unpacking the data from the files, you need to specify the exact number of frames to extract and the type of the data. The python wave library provides you with the number of frames in a file, as well as the number of raw bytes of data available. By dividing these two values, you can find the number of bits per frame, thus giving you the data type. This can also be confirmed by dividing the total number of unpacked values by the sample rate, which yields the duration of the file in seconds. Most wav files we found online stored data as int2 or int4. The audio base overlay in PYNQ apparently stores data as

int6. This is not a supported (or generally valid) type for unpacking and processing. This inconvenient format has led us to use example wav files found online for our testing. This issue was fixed with a snippet of code in our final implementation, so we are now able to use our own recorded sound files.

Aside from some minor difficulties in correctly splitting our data and running it through an FFT to get a resulting frequency, the rest of the PS code was fairly straightforward. We were able to take the frequency data, generate an RGB set from it, and animate a colored circle on a python matplot. The FFT portion of the code has since been moved to the hardware FFT block in Vivado, but we needed to generate the software version for timing comparison. The matplot graph has also been swapped out for a physical LED strip to display the output, which is a much nicer display.

Offboard Light Strip Control

A large part of our project has focused on using the PYNQ Z2 board to control a LED light strip with on-board GPIO. The PYNQ Z2 offers “full” control of Raspberry Pi and Arduino GPIO including digital pins, analog pins, and power/ground rails[7]. Given prior experience working with Arduino systems, we opted to use the analog pins available on the PYNQ to control logic level N-channel MOSFETs. These MOSFETS then in turn connect to a 12V rail on a breadboard, and when biased on the gate, output the ~+12V from the drain pin to a LED strip control pin, either R, G, or B, with these letters corresponding to Red, Green, and Blue. The Arduino Analog pins connect to the gates of the MOSFETs, based off the voltage provided by them 0-5V, the MOSFET would then act as a variable resistor, and output variable power to the LED control pins. This would allow us to control the intensity of the individual red, green, and blue LEDs on the strip, thus gaining the ability to display any of the ~16 million colors in the RGB spectrum.

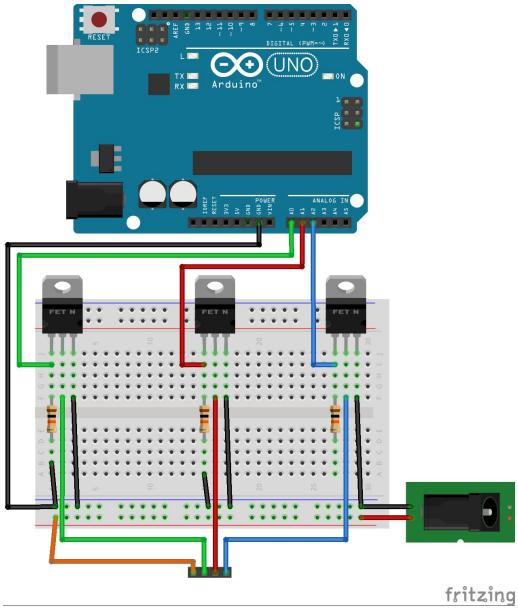

Figure 6 - Fritzing Diagram of Circuit

Unfortunately after working with the PYNQ GPIO library, we came to the realization that the analog pins on both the Arduino and RPI connectors could only be used to **read** analog values from 0-3.3V, which was quite misleading. The PYNQ Z2 itself has no DAC onboard, with the exception of the 24-bit DAC utilized for audio output. We posted on the PYNQ forums to find a way around this problem: https://discuss.pynq.io/t/analog-output-on-pynq-z2/1059

Following discussion on the forum and outside research, we found the PMOD DA4 module, which would allow us to connect a DAC directly to the PYNQ and control it with the provided example: link.

Purchasing the PMOD DA4 at this time was not possible due to shipping restrictions with the COVID-19 pandemic, as such we were not able to pursue control of the light strip with analog output.

To continue with our project, we decided to use the digital pins (capable of outputting 3.3V max) on the PYNQ board so we could get discrete red, green, and blue colors on the strip. Moving forward, audio frequency ranges will be clamped to appropriate values and mapped to a specific color. Again here, we ran into some problems as the MOSFETs we purchased were intended for 0-5V biasing. With a 3.3V bias, some of the MOSFETs would work, while others would not. To combat this, we made use of an LCR component tester to find which of our MOSFETs had the lowest terminal voltage. We selected 3 with Vt in range 3.2-3.4V. Moving forward we will see what we can do, possibly offsetting the gate voltage on the MOSFETs.

Division of Work

A large portion of this project was completed through pair programming. We spent a large number of hours online, screen sharing on Discord and sharing code in Github. That being said, the work was largely split along the PS/PL line. For issues in the PS, Peter would usually be the lead, while issues in the PL were usually led by Karl.

	Work
Group Effort	● Recursive to iterative FFT in hardware ● Textbook FFT and optimization ● Block diagram and debugging ● Code for interacting with LED strip
Peter	● Unpacking and processing .wav files ● FFT in software ● Frequency calculations
Karl	● Electronics setup for LED strip ● DMA stream ● Rebuilding base overlay

Table 2 - Breakdown of Work (Detailed below)

Even the more individual work we did often had input or assistance from the other person since the entire project interleaves itself. However, the following will be an approximate breakdown of what was done by each member. For Peter, the individual tasks were unpacking and processing .wav files, implementing the FFT calculation in software, and implementing the frequency calculations which could be mapped to a set of RGB values. The .wav file handling included issues like recording audio to the device in a usable format, and unpacking the unconventional int6 data type used by the PYNQ audio functions. The FFT and frequency calculations involved testing on numerous sets of data to validate results, as well as figuring out how to iterate over a static set of data to extract frequencies at different time intervals. For Karl, the main individual tasks were assembling the electronics setup for the LED strip, implementing the DMA stream, and rebuilding the PYNQ base overlay. The LED setup required a large amount of ingenuity and component testing with the parts he had due to the inability to order components online (COVID-19 restrictions). The DMA stream involved solving issues with the block diagram, as well as settings and quirks of Vivado, in order to send complex number data between the hardware and software. Lastly, the rebuilding of the base overlay with our own FFT block inserted was a very long process (with many hours spent generating bitstreams) in order to give us the capability of running the FFT calculation and accessing the GPIO pins simultaneously.

Group work spanned the entirety of the project, but the main tasks we worked on together were converting a recursive FFT to an iterative FFT in hardware, implementing and optimizing the textbook FFT, creating and debugging the block diagram, and writing code for interfacing with the LED strip through the GPIO pins. Our initial implementation of the FFT in hardware was a recursive version, which we quickly realized would not be supported by Vivado. As such, we spent a good amount of time converting this code to an iterative approach, which ultimately did not work. From there, we decided to implement the textbook's version of an FFT (as suggested by the professor) and spent time optimizing the area and latency of that design. We worked together to design the block diagram for our setup, running into issues with the code hanging in Jupyter Notebooks, and eventually tracking down the sources of these issues. Finally, we spent time testing the LED strip with our code, figuring out which colors we could achieve given our limited supplies and modified plans. We mapped these colors to frequency ranges and made tweaks until we were satisfied with the result. Once again, we each contributed bits of help and information to almost all parts of the project, from formatting and processing data sent through the DMA stream, to animating our frequency data in an aesthetic output. While our online setup was not exactly ideal, we were able to work together through this situation to ultimately reach a point where we were satisfied with our final result.

Sources

[1] https://en.wikipedia.org/wiki/Color_organ
[2] https://www.sandlotscience.com/giuseppe-arcimboldo/
[3] https://en.wikipedia.org/wiki/Fast_Fourier_transform
[4] https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/
[5] http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf
[6] https://en.wikipedia.org/wiki/Discrete_Fourier_transform
[7] https://pynq.readthedocs.io/en/v1.4/7_peripherals.html#arduino-connector
[8] https://www.audiocheck.net/audiofrequencysignalgenerator_sinetone.php
[9] https://docs.python.org/3.3/library/struct.html
[10] https://buildmedia.readthedocs.org/media/pdf/pynq/latest/pynq.pdf
[11] https://ccrma.stanford.edu/~mromaine/220a/fp/sound-examples.html
[12] https://discuss.pynq.io/t/tutorial-rebuilding-the-pynq-base-overlay/61
[13] https://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_1/pg109-xfft.pdf