6.111 Project Reflection
Mitchell Gu | Guitar Hero: Fast Fourier Edition

It's been several weeks since the beginning of Ryan and I's FPGA Guitar Hero project, and while the end product has remained the same, there have been many exciting changes to our implementation plans. Most of these changes are in my domain of the project, the audio processing portion. The audio processing half of the project involves digitizing in an electric guitar's pickup signal, performing note recognition on it, and passing the note data in serialized form to the game logic. In this report I will give the implementation changes and the challenges that motivated them, an overview of the current progress, and my plan moving forward.

By far the biggest change to audio processing has been switching from a 6.111 Labkit platform to the Nexys 4. Initially, we anticipated using the 6.111 Labkit for audio processing because its integrated AC'97 audio codec is well suited for digitizing incoming guitar data. Furthermore, we already had experience using the AC'97 in Lab 5 of the class and could base our implementation on Gim's fast fourier transform demo running on the Labkit.

Upon starting working with the Labkit, I increasingly found the decade-old development tools cumbersome. For example, the support for Virtex II FPGA used on the Labkit was abandoned in Xilinx's FFT module generator on ISE (the development environment) 10.1. In order to generate FFT modules for the Labkit, I had to create them with ISE 8.1, migrate them over to ISE 10.1, and try to get the module recognized in my existing project. Not only was the process inconsistent and time-intensive, but I figured if I was going to learn FPGA development for future projects, I wanted to learn with the most up-to-date technology and tools.

The more I looked into the Vivado development environment and Nexys 4 platform, the more I realized that the advantages we had seen in the Labkit had more powerful parallels on the newer platform. For audio input, the Nexys 4 had a built-in 12-bit ADC capable of 1 million samples per second. Xilinx's newer Vivado software also offered tight integration of pre-built IP (modules) with the Nexys 4 hardware, including the ADC. The Nexys 4's plentiful digital signal processing (DSP) slices and faster processing speed gave me more freedom in how implemented the note recognition. Finally, the Vivado software made it easy to string IP modules together, reconfigure them, and pre-synthesize them to cut down on compile times.

With knowledge of these tools, I've successfully displayed a fairly optimal real-time, low-latency frequency spectrum of a guitar pickup input onto a VGA monitor. I was able to achieve a low noise floor, high frequency resolution, and sharp peaks through several design decisions. Firstly, I took advantage of the ADC's 1Msps sample rate to oversample by a factor of 256. This technique increases the signal to noise ratio while increasing the bit width of a sample from 12 bits to 16 bits. Compared to a 16x oversampling factor, the difference in background noise is very audible. The oversampling was also useful for effectively lowering my sample rate to the 4kHz range. After performing a 4096-point FFT on these samples, the frequency spectrum was in an ideal range of 0 to 2kHz. Since the guitar would only output fundamental frequencies up

to around 1khz, this range made efficient use of the FFT's output while providing a detailed 1Hz bin resolution.

The next task was to identify which notes were present in a given frequency spectrum. My first instinct was to implement a peak detection algorithm that would look at each frequency of the 37 possible guitar notes and determine if there was a peak there. While this was feasible, I had concerns about how difficult it would be to calibrate the peak detection to the right sensitivity. I thought this would involve a lot of time-consuming iterations to test with an actual guitar. I had also been researching online common techniques for note detection, and it seemed like spectral peak detection was generally not regarded as a great technique. I'm sure it would have been a lot easier in our context because the signal from the guitar pickups was very pure, but I had a more elegant solution in mind that would capitalize on the FPGA's parallel computing power.

Instead of detecting peaks, for each of the 37 notes I would play the note on the guitar correctly, then save the spectral profile that the FPGA produced on the SD card. Then, I would parse the SD card data dumps, load those reference spectra into block RAM in the FPGA, and compare the current spectrum with each reference spectra. The comparison would be a correlation metric inspired by the 6.02 class which performs a dot product of the current spectrum and a reference spectrum and normalizes by the magnitude of each. After some research, I've determined this approach is very feasible because each DSP slice is perfectly suited to perform a multiply-and-accumulate procedure that is required for performing a dot product. This approach seems more elegant to me because it's data-driven and only requires one parameter to tune: the threshold correlation value to consider a note played. Furthermore, the profiles don't have to be single notes. I could record reference profiles for common chords and recognize entire chords as well.

So far, I am halfway into this plan; I've successfully been able to write 16 profiles in one session into 16 SD card data slots, which makes it easy to take several samples. I also wrote quick Python scripts to read from the data dumps, plot a histogram to confirm they're correct, average across different slots, and generate a .coe coefficients file that I can initialize certain block RAM instances on the FPGA with. The process of figuring this out was a great opportunity to learn about how communication with an SD card works, and how to manipulate raw binary data within unformatted sectors of an SD card. From here, the remaining work involves implementing the dot product and correlation metric calculation logic, then setting a threshold and outputting identified notes serially for Ryan's game logic to process.

I've really enjoyed how much I've learned about FPGA development so far, but I'm still a good bit behind where I'd like to be to finish this project. Around a week remains to work on the final project, but I'm still finishing up the note recognition. I hope to finish the audio processing in the next day or two, then shift my focus to the graphics side of the game FPGA. Little to nothing has been done yet for graphics, so I would like to put together a module for displaying modern fonts on the game screen, as well as basic game graphical elements such as notes and strings.