FPGA Music Experience

Rhian Chavez & Miles Johnson

December 2019

Contents

1 Introduction

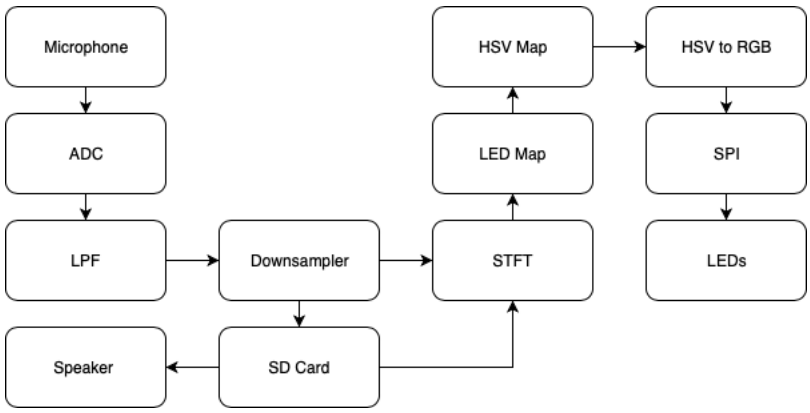

Figure 1: System Block Diagram

We decided to use the Nexys 4 DDR Artix-7 FPGA running at a 100Mhz clock rate to record, playback, and visualize the frequency spectrum of any sound (particularly music) in an aesthetically pleasing manner. We used the Dot-Star LED strips shown below to display the audible frequency spectrum of sound recorded to an SD card and played back through a speaker (or passed directly from the microphone to the LED frequency visualization). The brightness of each LED is proportional to the energy in it's associated frequency bin (after the energy reaches a certain threshold, so that the LED isn't just constantly dim, ruining the experience). After an LED has been illuminated, if it is not again told to be illuminated by the next round of frequency information, it slowly decays brightness (over about 300 milliseconds).

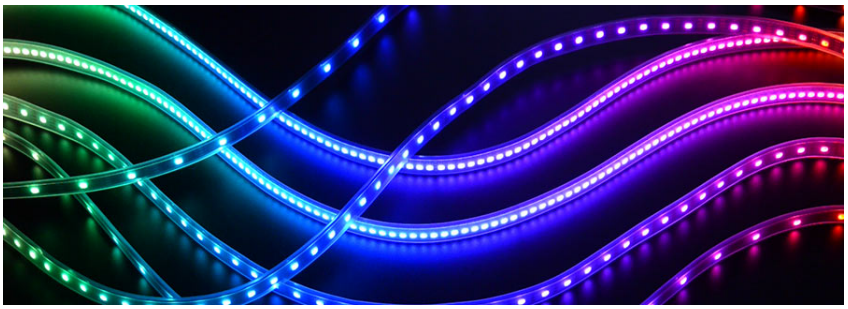

Figure 2: LED strips

We attempted to not only show the short time Fourier transform (STFT) on the LED strips, but sweep around the HSV color wheel and only display significantly powerful frequency bins simultaneously. We also stacked many

same lenght LED strips on top of each other, varying the threshold to turn on the LEDs (associated with each frequency bin) as the strips distanced from the vertical center (displaying not only frequency but also amplitude information in a much more apparent way). The following sections describe the technical details of our implementation on a module level.

2 SD Card Interface - Miles

It took three iterations to arrive at an SD Card interface which recorded and played back sound to an adequate extent. By this I mean that the input audio data to the SD sounds identical (to my ears) to the output audio data from the SD when both are played back through the aux output.

In this section I will outline my approaches to each iteration, with the last being a more detailed explanation of the final implementation, then breifly explain my implementation of rewinding on the SD card. First, I will start with a brief explanation of how the SD card reading and writing functions.

2.1 SD card read/write

The SD card reads and writes in 512 byte blocks. By this I mean the following: when the SD card is in the idle state, one may assert the write or read command to begin a 512 byte write or read operation starting at a given address. I will now explain how these 512 byte write and read operations function.

Both operations function by reading/writing a byte at a time. For the write operation, the SD will signal when the next byte should be present on the data input, then begin writing that byte the next clock cycle. For the read operation, the SD will signal when the next byte is available on the data out terminal. For both operations, the rate at which the 512 bytes are written or read is entirely determined by the SD card, and the SD card does not perform these operations at a constant rate. For both of these operations, once 512 bytes have been written or read, the SD card returns to the idle state.

Through various testing, it seems to me that, when clocked at 25 MHz, a 512 byte write operation usually takes about 150,000 clock cycles, with a maximum of about 525,000 clock cycles. Every now and then there is a 1-1.5 second delay, but this only ever seemed to happen immediately after I programmed the fpga. For our purposes, these 1-1.5 second delays just ended up resulting in a short delay in music playback every once in a while.

I should also note here that the SD card must be clocked at 25mhz, whereas the rest of our project is clocked as 100mhz. This did not seem to cause much trouble when we combined the separate parts.

2.2 First Implementation

I began testing the SD reading and writing by using sd_contoller.v and the 7 segement display to write real-time seconds to the sd card as fast as it could

take the information, and read back as fast as it could read back. To do this, I continually asserted read/write as long as a corresponding button was being pressed on the fpga. It seemed write operations were consistently slower than read operations, so I ended up continuing with that assumption for the duration of the project.

My first approach to solving this discrepency between read and write speeds was to alternate between writing audio data and time data to the SD. I wrote code that would alternate between inputing current audio data and the number of clock cycles between audio data. I wrote a module which took a 512 byte block from the SD as an input, interpreting the first value as audio data, the second as time data, etc., and outputting each audio data for the corresponding number of clock cycles given by the corresponding time data.

After testing this module through simulation, I tried to implement it on the actual SD card. Nothing worked as it was supposed to, and eventually I realized that tracking the time between every single byte might be a bit overcomplicated and unnecessary. This lead to my second implementation.

2.3 Second Implementation

I decided to instead record the time of each entire 512 byte block in BRAM, and split that time equally over each individual byte. I implemented a counter to count the number of clock cycles between the start of each 512 byte block, and recorded that time to a BRAM module.

After some debugging, this module seemed to work fairly well. The hardest part in this implementation was trying to include all the necessary timing data into BRAM. After trying to increase the write width of BRAM, it seemed to me that BRAM could only write a single byte at a time. The problem with this was that the number of clock cycles ended up being a 24 bit value. First I tried just taking the top 8 bits, but this was not accurate enough. I ended up taking advantage of the fact that it takes the SD card a while to read or write 512 bytes. I stored each 24 bit value at 3 different addresses in BRAM, writing a protocol for reading/writing these in groups of three.

To test this implementation, I recorded tenths of seconds to the SD, and played back real time seconds on one side of the 7 segement display, and the data output from the SD on the other side. Visually, I could tell no difference, so I moved on to testing with actual audio data.

Recording and playing back audio, the general structure was audible, but there was a lot more static than the input. I concluded that within each 512 byte block the SD card must write fairly irregularly. One solution would be to try to make my previous implementation work, but luckily at this point I already had an idea of how long it usually or maximally takes the SD to write/read a 512 byte block, leading to my third implementation.

2.4 Third and Final Implementation

For this implementation, I added my implementation of a FIFO to regulate the incoming audio data. Instead of continuously writing to the fpga, I created an interface which made use of two 512 byte registers. Incoming audio data is sampled at about 24khz to fill up the first 512 byte register, and the SD card interface writes from the second 512 byte register at its own pace. Once the first register has been filled up and the second register has been completely read, I update the data from the second register with the first registers data, start refilling the first register with current audio data, and begin the next 512 byte write operation to the SD card.

Based on the write times I saw when testing the SD, it seemed to me that 24khz was slow enough that the SD would almost always finish writing before the first register was full. In this way, all of the 24khz sampled data should be retained. To read from the SD, I used the same stucture from my second implementation, splitting the time of each 512 byte block evenly between bytes. This made more sense now since it was written at a constant sampling rate rather than the arbitrary SD write rate. Note that the time for each 512 byte block should now be identical in this implementation. Interestingly, when I tried to use a constant number of clock cycles per 512 byte block, which I calculated by hand, it consistently played back too fast. This was most likely a mathematical error on my part, but luckily I already had the BRAM interface from the previous section. Using this interface to record the time for each 512 byte block as before, the playback sounded correct.

I will briefly summarize the final implementation in the following two paragraphs:

For writing audio data to the SD, 512 bytes of data are sampled from the ADC at 24khz and recorded in a register. Concurrently, 512 bytes of data are written to the SD from a second register. The number of clock cycles for both to complete is recorded in three bytes of BRAM. Once the first is full and the second has been completely written to the SD, the first register is written to the second, write addresses are updated, and the process repeats until the user stops pressing the write button.

For reading audio data from the SD, a 512 byte read operation from the SD is started at the same time as a 3 byte read operation from BRAM. The 512 bytes are read to a register, and concurrently a second 512 byte register outputs data at a constant rate. This rate is such that it finishes outputting all 512 bytes after the number of clock cycles given by the 3 byte value read from BRAM. After this number of clock cycles has passed, the first register is written to the second register, read addresses are updated, and the process repeats until the user stops pressing the read button.

2.5 Rewinding

Once I had implemented the SD interface, it was simple to implement functionality which allows the user to rewind through recorded audio data.

When reading from the SD, I specify addresses to read from which are multiples of 512, as well as corresponding addresses to read time data from BRAM which are multiples of 3. When the user presses the rewind button (btnu), I decrease the SD address by 50*512=25600, and the BRAM address by 3*50=150, which corresponds to rewinding by about a second.

3 Oversampling & Filtering Input - Miles

Sampling the data at 24khz already creates noise, so we first sample at 48khz and apply a filter from lab 5 to decrease noise. I updated the coefficients in this filter to only filter less since we are sampling much faster than in lab 5. When playing back the audio, we use the same filter to reconstruct a 48khz signal in the same way as described in lab 5.

In the FFT module we would only like to analyze frequencies up to 4khz, so we under-sample further before sending audio data there. We briefly attempted filtering data further before sending it to the FFT, but saw no difference in that time.

Significant time was spend on these filters due to our use of a mic not well suited to the recording and playback of audio. Many interesting and illogical filters were used to make this mic sound slightly better and look better on the lights. However, after the design checkoff, we switched to the lab 5 mic, and all of these filters were useless. We defaulted back to similar filters to lab 5, which sounded and looked much better.

4 Frequency Analysis - Rhian

Frequency analysis was done using a 128 bin running short time Fourier transform. 8 bit real valued data was provided to the Fourier transform input at consistent rate from either the ADC or the SD Card (taking audio directly from the microphone or from the saved data, respectively).

The STFT outputs both the real and imaginary parts of each 128 frequency bins for a given set of data, and provided each frequency bin output at a variable rate (on the order of 100 MHz maximum, but slowed down to a rate dependent on the input frequency which was approximately 8KHz). Since we were interested in the magnitude of each bin of the STFT, I wrote a module which squared and summed the real and imaginary parts of each frequency bin, as well as finally took to the square root and output it to the rest of the system (providing the magnitude of the frequency content in each bin).

A visualization of the running STFT in this module can be shown in the form of a spectrogram (shown below). The vertical axis varies in frequency, the horizontal axis varies in time, and the color varies in magnitude. This representation shows how the frequency content of a signal varies over time, each different column representing an individual STFT.

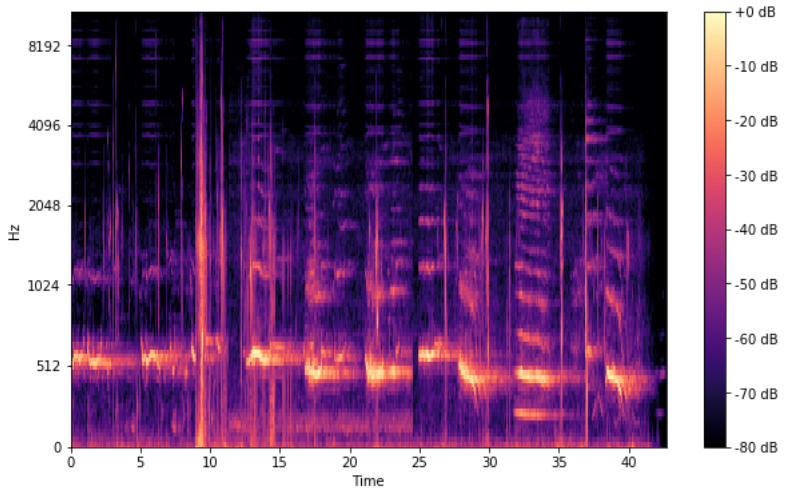

Figure 3: Spectrogram

Like all of the other modules which I wrote, the frequency analysis simply raised a single clock cycle flag every time a new output was ready. This module provided output in the form of the magnitude and bin number of a particular STFT bin and had a throughput of 1 per clock cycle. The output magnitude was an unsigned 27 bit number, but typically, only the bottom 9 bits had any relevant information.

If I were to do this section again, I would guarantee that the output of the STFT was less than 8 bits in magnitude so that I knew the maximum bit depth of the resulting data that passed through and out of this module to the rest of the visualization signal path. The lack of this control resulted in significant guess and check processes (to threshold frequency bin power to turn on an LED, etc). It would have also been much easier to implement beat detection (by taking the discrete derivative of the power in the lowest frequency bins and determining if it is large enough to warrant a beat) with more determined knowledge about the typical magnitude of the STFT bins.

5 STFT to LED Mapping - Rhian

The output of the STFT is 128 bins, but the LED strips we used only had 72 LEDs per strip. Therefore, if we were going to display the audible frequency spectrum on them from the STFT, we needed to map the 128 bin output to the 72 LEDs. We also wanted to display the lower frequencies at the horizontal center of each strip, and have horizontal distance from the center increase with increasing frequency (further constraining the number of unique frequency bin energies we could display).

The solution I ended up using mapped the lowest frequency bins of STFT to

a single LED towards the center of each strip, and averaged 2 STFT frequency bins for each of the remaining LEDs. This method does create a sense of non-linearity to the frequency visualization experience, but was very straightforward to implement.

The input to this module was the frequency bin magnitude and number from the STFT. The maximum latency was 2 clock cycles.

6 HSV Assignment - Rhian

Not only did each LED on each strip need to be associated with one to two frequency bins, they also needed an associated color and brightness. In order to implement the desired "decay" of each LED, it was necessary to keep their previous color and brightness in a 72x8 bit BRAM so that the previous brightness could be compared against incoming frequency information and decayed if that frequency information was not significant enough.

We desired that the color of the LEDs should sweep through the HSV color wheel as a function of time, so we needed to save and manipulate their color/brightness in an HSV format. The HSV color mapping scheme is depicted below for clarity.

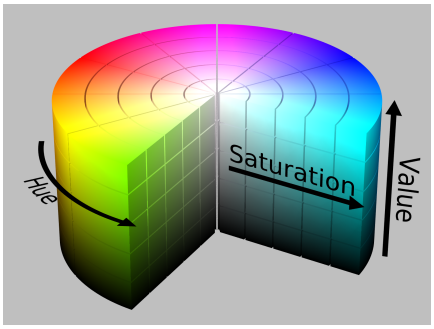

Figure 4: HSV Color Cylinder

In order to implement the constant hue sweeping and also assign value to each LED asynchronously, the HSV assignment module (hue_sweep_listen.sv) constantly listens for new LED frequency data and if that data is greater than a set threshold (set by switches on the DDR4, and greater for LED strips further from the vertical center), then it assigns that frequency data to the "value" of the associated LED. This new frequency data (if sufficiently energetic) is stored in BRAM for the decay process to commence upon future iterations.

Information from this module is then sent to the rest of the LED frequency signal chain at a constant rate since it the HSV values are held in BRAM that is asynchronously written and read from.

7 HSV to RGB - Rhian

Unfortunately, the data type that the Dot-Star LED strips take over SPI is RGB color space as opposed to HSV, so I needed to create a module to convert from HSV to RGB. The process for converting between the two color spaces is straightforward from a mathematical perspective, but rather tedious to implement in verilog due to the necessity of a division operation. I was able to successfully implement this module by pipelining the 5 steps of math that take place in the conversion, (simultaneously passing a long relevant information at each time for each stage of the pipelined system.)

This module has a latency of 5 clock cycles and a maximum throughput of 1 per clock cycle.

A visual representation of RGB color space is shown below for clarification.

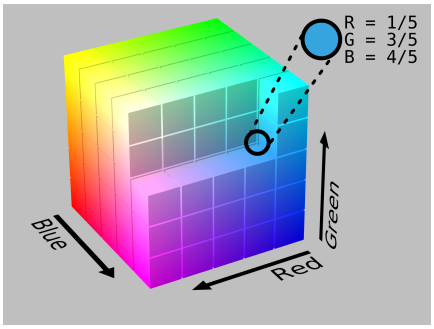

Figure 5: RGB Color Space

Data from this module was then passed to the SPI communication interface which commanded each LED strip.

8 LED SPI Scheme - Rhian

The Dot-Star LED strips have their own communication scheme which I had to implement on the DDR4 in order to properly communicate with them. Each strip requires 5V power, clock, and data. I chose to use a 50% duty cycle clock at an approximately 500KHz frequency. Upon the rising clock edge a new data bit was read on the data line.

Below the communication scheme of the LED strips is shown graphically:

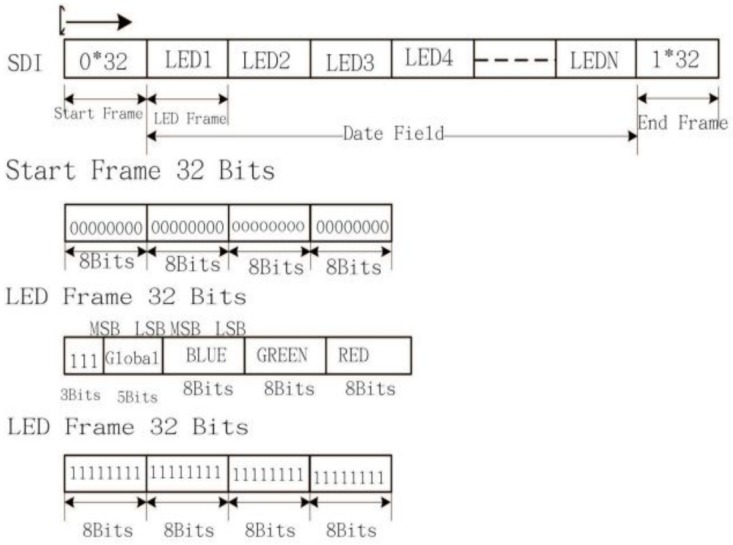

Figure 6: LED SPI Scheme

As you can see, the data line begins with 32 low bits and ends with 32 high bits. In between is all the data for each LED, each containing 32 bits of information per LED. Each RGB value was 8 bits (which was exactly what I designed the HSV to RGB module to output.

While implementing the system, I noticed that my color brightness was never what I told it to be, and finally realized that although the order of each color word "RGB" was switch in the LED frame, the bits within each word were not! This required some fun debugging to figure out.

9 Conclusion

Perhaps the most important lesson we learned during this project is to make sure you are using adequate equipment before trying to fix your code. We spent a lot of time trying to account for the static our original mic was creating, which all went away when we switched mics.

On the coding side, one take away is that it is often better to try out a simpler method first, since it may be accurate enough for your purposes, or easy to modify. In our implementation of the SD card interface, trying to time each individual byte took a good chunk of our time and amounted to nothing. Timing the entire 512 byte block was much easier to do, and while it was not adequate, it was relatively simple to implement a FIFO which made it adequate.

Another takeaway is that testing the interfaces between modules is impor-

tant. For the FFT and lights interface, each module was written and tested separately. They all seemed to work, but once we tested them on the actual fpga, we ended up finding multiple issues with the module as a whole. It would have saved a decent amount of time if we had written test benches for the interfaces between modules as well.

Overall, for anyone looking to repeat this project or do something similar, we have a few tips. First, use an adequate microphone. Second, the SD usually takes at most around 500,000 clock cycles to read/write a 512 byte block, so it should be relatively safe to write sound data up to 24khz to it. Third, give yourself time to play with the frequency range for the lights (decreasing it to 4000hz made ours look a lot better).

10 Appendix

10.1 Associated Verilog

Verilog modules by author (posted on MIT 6.111 course website):

Rhian & Miles

- top_level.sv

Rhian

- fft_mag.sv
- squarer.sv
- freq_bin_to_led_num.sv
- hue_sweep_listen.sv
- hsv_to_rgb.sv
- rgb_to_lights.sv
- beat_detect.sv

Miles

- SD_interface.sv
- fir31.sv
- coeffs31.sv