# Finding, Parsing, and Generating Audio Transcriptions

https://github.com/woxsao/6.111-Music-Transcription-Project

Monica Chan
*Department of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, USA
mochan@mit.edu

Cynthia Zhang
*Department of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, USA
zcynthia@mit.edu

Ian Hueston
*Department of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, USA
ianh7095@mit.edu

***Abstract*—This is a music transcription system. Provided music tones, the system is able to detect and determine which note was played, and then transcribe it live on a musical staff, displayed on a monitor via HDMI. The system will be able to transcribe notes ranging from C4 to A5, rhythms from eighth notes or rests to whole notes or rests. The system will be able to handle music up to 124 BPM. The system is comprised of three main components: audio processing, note and rhythm determination, and visual display.**

***Index Terms*—Digital systems, Field programmable gate arrays, signal processing, audio processing, Fourier Transform**

## I. Introduction

A music transcription system is one heavily dependent on signal processing. A large part of the risks of this project project lie in the audio pre-processing stage, where it is critical that audio data filtered properly, or else excess frequencies may muddle the audio data. Because the project depends on precise detection of frequencies, it is also essential that the FFT is highly accurate. To accomplish these tasks, we implemented an FIR low-pass filter that will filter out any high frequency content that our system does not want or need. We then implemented a Hanning window to improve the accuracy of our frequency domain analysis. This process was the main challenge we foresaw, as our team has little to no experience with signal processing. The rest of the system involves figuring out notes, note values, and where to display each note.

## II. Hardware Components/Overview

This system requires little hardware. The primary components are the Real Digital Urbana FPGA board, and a monitor, connected via an HDMI cable. The microphone will be the default digital PDM microphone on the Urbana FPGA board. The block diagram for our system can be seen in Appendix 1.

## III. Clocking/Sampling Rate Summary

The majority of the audio processing system runs on a 69.632 MHz clock. This value is determined via the sampling rate, as it is 32 times greater than the microphone sampling rate of 2.176 MHz. The digital display system runs on a 74.25 MHz clock.

The main design constraint we had was a maximal BPM which we chose to be 160 BPM. This corresponds to a frequency of 2.667 Hz, so we designed the audo preprocessing to be able to output note information to the image processing stage at around 4Hz. The justification for each of the samples is as follows:

- Audio Processing Clock rate: 69.632MHz. This was chosen to be 32 times the microphone sample rate because the low pass filters require 32 clock cycles to compute a filter output for a single microphone sample.
- Microphone Sample rate: 2.176MHz
- Input frequency FFT: 17KHz. This is because we are downsampling a factor of 128 from 2.176MHz.
- Output of peak finder: 4.15Hz. The FFT has 4096 bins, and so the peak finder has to find the maximum of the 4096 bins. So $17000/4096 = 4.15$Hz.

Since the output of our peak finder is at 4.15 Hz, we meet the design requirements.

## IV. Sampling and Data Preprocessing

### A. Microphone, Low-Pass Filter, and Decimation (Chan)

Our microphone will sample at a rate of 2.176MHz, with a clock of 69.632 MHz. This sample rate and clock frequency was chosen because the filter we designed requires 29 clock cycles to calculate the output from a single input from the microphone. The clock cycle is 32 times the microphone

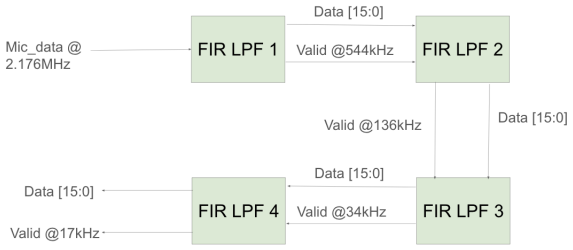



Fig. 1. Diagram of the chained FIR filters with sample frequencies. Notice the first 3 decimators downsample by a factor of 4, but the last one downsamples by 2, giving a total dowsampling of 128.

sample rate to give us sufficient clock cycles to calculate the filtered output before a new microphone sample comes in.

$$F(s) = \sum_{i=0}^{28} c(i) \cdot s(28 - i) \tag{1}$$

The goal of the filter is to filter down to a rate of 17kHz and filter out all audio higher than 8kHz, since the range of frequencies we are representing are between 261.63 Hz and 880 Hz. We do this by creating 4 29 tap FIR (Finite Impulse Response) filters, each with a cutoff frequency of 0.125. Eq. 1 shows the math for a FIR filter, where $c$ is the coefficient array, and $s$ is a length 29 array that stores the last 29 samples recorded by the microphone. Three of the four decimators downsample by a factor of 4, and the last one downsamples by a factor of 2. Together, chaining 4 of these filter/decimators together downsamples by a factor of 128, to give us a sampling frequency of 17kHz. Fig. 2 shows this cascaded chain of filters with the frequencies.

Each individual filter was a 29 tap filter, whose taps were designed with TFilter, an online FIR filter designer. The cutoff frequency is at 0.125 the sample rate, with a roll-off starting at 0.06 the sample rate [1]. This gives us an overall cutoff of $\frac{1}{256}$ the input sample rate, around 8.5kHz.

Our system is designed to represent an 8 bit amplitude wave, but the decimator stages all have an output of 16 bits to account for bit growth between the stages of the filter. At the output of the fourth filter, we take the top 8 bits of the 16 bits as the input into the Hanning Window module.

The testbenched output of our FIR filter can be seen in Fig. 3. Since we're using a PDM microphone, the actual input signal from the microphone looks like our pdm_out signal on

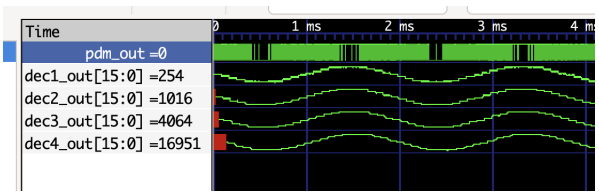



Fig. 2. Current cascaded decimator output

the diagram, and our filters successfully reconstruct the sine wave.

Significant time was spent debugging audio quality on the output of the filter/decimation chain. This problem is three-fold: Bit growth, bit selection of the fourth decimator, and input ternary to the filter chain. The latter 2 are discussed in section III-E. For bit growth, we discovered that the filter we chose leads to a gain of 4 at every filter stage, that is, for an input of value 1, the FIR filter would output 4. Therefore, by the end of the 4 filters, we would end up with a total gain of 256. Initially, we designed each filter stage to downshift by 2 (divide by 4). However, this led to removing critical audio information at each stage. Ultimately, we decided to preserve the bit growth between stages and take the top 8 bits at the very end.

### *B. Hanning Window (Zhang)*

The FFT will use 4096 samples. The purpose of the Hanning window is to pinch down the corners of the window of samples in order to prevent any windowing artifacts and prevent spectral leakage. For a given window of samples, we multiply each value by a coefficient. The following equation generates the coefficients for a Hann window:

$$w(n) = 0.5(1 - \cos(2\pi \frac{n}{N})), \quad 0 \leq n \leq N \tag{2}$$

$n$ represents which sample number we are on (e.g. 2106th sample in the window). $N$ relates to the size of the window $L$, where $L - 1 = N$. Because our window width is 4096, $N = 4095$.

To implement this in SystemVerilog, we needed additional memory. While the original approach to calculate coefficients was to use real types (to accommodate non-integer values), this was eventually shot down as we discovered that reals cannot be synthesized. We then pivoted to pre-calculating the coefficients and storing them in a BRAM. The values are scaled up in order to do integer math. The coefficients are calculated and scaled as follows:

$$w(n) = \text{round}(0.5(1 - \cos(2\pi \frac{n}{4095})) \cdot 2^{24}), \quad 0 \leq n \leq 4095 \tag{3}$$

The coefficient is scaled by 24 to ensure that extremely small coefficients will still be reflected when stored as an integer. These calculated coefficient values are then stored in hex as a 6 digit value. The BRAM has a width of 28 (an extra 4 bits of 0 are prepended to the beginning to prevent signage issues) and depth of 4096. When calculating the output value of the Hanning window, the audio data value is multiplied by the corresponding coefficient retrieved from the BRAM, and then arithmetically shifted to the right by 24 (to undo the scaling).

The two-cycle delay for fetching from the BRAM presented an alignment challenge. In order to maintain proper alignment of audio data to coefficient, we pipeline the audio data sample so that it remains within the system while the BRAM is retrieving a coefficient.

The resulting calculation is a scaled value according to Hann coefficients, where within a window of 4096, the corners are pinched to zero, as seen in Fig. 3.

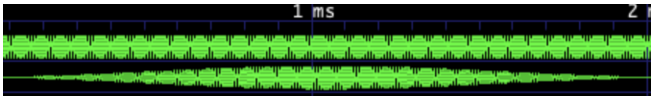



Fig. 3. Window of data before and after the Hann windowing filter.

## V. Note determination

### A. FFT (Zhang)

The FFT module uses the Xilinx IP, generated by Vivado as an *.xci file. The inputs are 16 bits wide, comprised of a real part that is the 8-bit wide outputs from the Hanning window and an imaginary part that is 8-bits of 0. The output is 48 bits wide, with 24 bits real, 24 bits imaginary. The following is the specified Vivado configuration:

- transform length: 4096
- target clock frequency: 70 MHz
- target data throughput: 1 Msps
- data width and phase factor width: 8 bits.
- scaling options: unscaled
- output order: natural

With audio samples at 17kHz as input to the FFT and 4096 samples per FFT, the system achieves a bin width of 4.15Hz and output frequency resolution of 4.15Hz. This should be sufficient for identifying notes from C4 to A5, where the narrowest difference in frequency is about 16Hz.

We chose the unscaled option as the output at 8 signed bits for imaginary and real was not precise enough to determine peaks. The unscaled option results the output in being 24 bits each for imaginary and real. This allowed for sufficient precision for determining peaks.

### B. Peak Finder (Zhang)

The peak finder module determines which frequency bin the input audio belongs in. The module takes the 4096 samples of 48 bit data from the FFT and outputs an index representing the bin that the audio data best fits in. To determine where the peaks are, we need to take the magnitude of the outputs of the FFT. To implement this, we simply square the top 24 bits and the bottom 24 bits by performing signed multiplication. Performing unsigned multiplication was an unfortunate bug that took significant time to debug.

To find the peak of the 4096 outputs of the FFT, we keep a running counter up to 4096 for each output sample of the FFT, a running maximum that keeps track of the max value seen so far, and a running maximum index that keeps track of which index has thus far been the largest in magnitude.

While the FFT outputs 4096 bins, we do not need to observe each of these. The magnitude calculations and running tallies only run for 300 of the values, as the range of frequencies we are interested in are limited to 261.63Hz (C4) to 880Hz (A5). This means we are only interested in the first 213 bins. As a result, the peak finder module outputs a frequency bin index after processing 300 of the FFT output values.

### C. Integration of the Audio Processing Pipeline (Zhang, Chan)

The low-pass filter and decimators were developed separately from the FFT portion of the audio processing pipeline. In the development phase, the low-pass filter and decimators were frequently tested with audio output in order to ensure that the filters were performing as expected to a listener. The Hanning window, FFT, and peak finder were tested using testbenches to observe the general shape of the wave. However, during the integration stage, we found that there is often a mismatch in expected shape and audio output.

For example, for audio output, taking the bottom 8 bits was necessary for clear audio output. However, the Hanning window required the output to be the top 8 bits instead, as the bottom bits would break the shape of the wave. The difference in wave forms can be observed in Fig. 4.

Another instance of mismatched expectations was the input to the FIR Filter. Since the microphone is a PDM microphone, we wrote a ternary statement that changed the input to the filter based on whether the input was a 0 or a 1. To achieve clear audio output, we would threshold values of 0 from the microphone to 0, and 128 otherwise. However, for the FFT to work as intended, we needed to threshold to -127 in the case the microphone input was 0. We suspect the 127 / 0 combination did not work due to filter biasing, although we never found the exact reason.

To test the audio processing pipeline without the visual display output, we utilized the seven segment display on the FPGA board. This was used to observe intermediary values of the pipeline, such as the output of the fourth decimator, the output of the Hanning window, and output of the peak finder. This helped isolate problems when trying to troubleshoot on hardware. It also allowed us to verify the FFT was outputting correct values, as we could verify that the expected index was produced by the peak finder based on various input tones. The troubleshooting process involved feeding 750 Hz waves directly into various points of the pipeline, and then pivoting to using the microphone with pure tones.

### D. Note Lookup Table (Chan)

A Python script was used to generate all the bin cutoff indices for each note. First, we copied all the 22 notes' frequency values into an array, and then applied the following equation to determine its bin $b$

$$b = \lfloor f/4.15 \rfloor \tag{4}$$

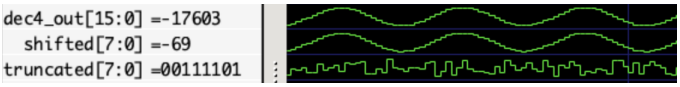



Fig. 4. The top wave is the output of the fourth decimator. To observe the 16-bit waveform as an 8-bit wave, we tested the shape of taking both the top 8 bits (arithmetic shift right by 8) and bottom 8 bits (truncating the top 8 bits). The shifted shape can be observed in the second row, while the truncated shape can be observed in the third. Clearly, the second row better maintains the shape of the wave, but for some reason, the third row sounds better as an audio output.

where $f$ is the frequency, and 4.15 is the bin width of our FFT. We saved these bin indices and made a 2D array in the System Verilog module. This was stored as a 2D array rather than a BRAM because the array itself is quite short so we did not need the capabilities of the BRAM. Since each item in the lookup table stores the lowest bin index corresponding to a certain note, we would look up the largest bin index in the lookup table that the input was still greater than.

## VI. Sprites and Image Representation (Hueston)

To be able to display the notes on a screen, a sprite sheet was made to represent all note types that were needed. The sprite sheet contains 11 32x100 pixel sprites representing the notes, rests and other features used in the display Fig. 5. The sprites used include whole, half, quarter, and eighth notes and rest, the sharp and natural symbols, and the treble clef. The sprites are arranged vertically and the memory address is shifted by a factor of 32*100 depending on what frame is displayed.

### A. Image Sprite

Image_sprite is the main module that handles the image representation in the system. It takes in notes and processes the information it has to display pixels for the HDMI signal. This module includes writing notes to BRAM and determining which type of note to display and where, along with any additions such as sharps, naturals, treble clefs, and staff lines. The module then outputs a color to be sent in the HDMI signal. The sequence of how the output is determined is as such: based on what area of a measure hcount and vcount are currently on, that measure is pulled from the BRAM. Based on what area of eighth note block hcount and vcount are in, the note in that block of the measure is compared with other notes in the measure to determine if this block should be shown and if so, which note/rest type should represent it. From that it shifts the note by its displacement from middle C and reads from memory to get the color to be sent to the screen.

### B. Staff Lines

There are 5 systems in our sheet music, each made up of 4 measures. The writing area covers (96,100) to (1152,600). The staff lines appear on any line in the writing area where vcount == x33, x41, x49, x57, and x65, with smaller ledger lines appearing above(x25) and below(x73) the staff of A4 and C4/C#4 respectfully. The measure lines appear within the staff lines at every the end of each measure in the drawing zone. There is also the treble clef which is rendered a block before the note writing blocks begin.

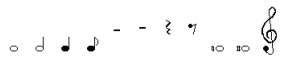


Fig. 5. Sprites used in the image_sprite module. Each sprite has dimensions $100 \times 32$. They're stored in the sprite_mem BRAM.

### C. Note Writing

The image sprite module takes in notes and stores them in BRAM. This BRAM is 8*6 bits wide, for the 8, 6 bit notes in a measure, and 20 entries deep, for the 20 measures. Given the input BPM, in which 60, 80, or 120bpm can be chosen via switches, the module will take in the current note at eighth note intervals and add it to the current written measure, then it moves to the next eighth note. There is a also a metronome that flashes an led at each eighth note to help keep people in time with the transription. When a measure is completed, the module starts writing to the next measure in the BRAM. This writing automatically stops writing when all possible notes have been written. This note writing process is catalyzed by flipping a switch on the FPGA. When reactivated, the BRAM is reset to prepare to be written again.

### D. Note Locator

The note locator module inputs the note that is currently being displayed in the frame, and determines where on the staff the notes should be. All of the note sprites are placed at C4 by default, and for each note above in letter of C4, its displacement is increased by 4, which increases the address looked at by 128. For C4/C#4 and rests, displacement is 0.

### E. Duration Detection and Memory Addressing

The image_sprite module can display any sequence of notes[1] it is given in whole, half, quarter, and eighth notes and rests. Each note is a 6-bit number, in groups of 8 for each measure. In a 6-bit note, bits 0 to 4 represent one of our 22 notes, 6'b100000 for C4 up to 6'b110101 for A5. Bit 5 represents whether a given note is a note or a rest, 1 for notes and 0 for rests. When hcount and vcount are at a certain note's block, that note is compared to all of the other notes in the measure to determine if the note should be shown, and if so, which type of note should be displayed.

Consider a measure such as in Fig. 6 with a note sequence of C,C,C,C,D,D,0,D. In this case, these notes are displayed as a half note C, a quarter note D, an eighth rest, and an eighth note D. There are some restrictions upon which notes can be shown at certain parts. The general rule is that a note type can appear if the nth eighth note of the measure mod the eighth note value of the note (eg. 1 for eighth note, 4 for half note) == 0. This is generally for musical cleanliness, so there aren't quarter and half notes starting on ands and the like. When a note larger than an eighth note is displayed, the first note in that note's group is shown and the other blocks that are part of the note are not shown.

There is also the possibility of showing a sharp or natural symbol. These symbols adhere with the rules of sharp and natural symbols. For a given note, it will look backward in the measure to look for other notes of the same letter. For a sharp note, if the last played note of that letter in the measure is sharp, no symbol will be displayed, and if the last played note of that letter in the measure is natural, or there is no

[1]"notes" refers to notes or rests

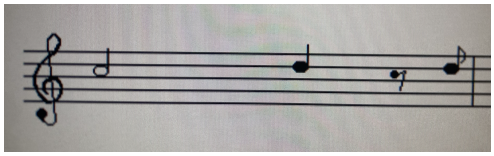


Fig. 6. Test notes to show the duration detection working.

previous note of that letter, the sharp symbol is shown. This is well shown in Fig. 7 For a natural note, if the last played note of that letter in the measure is sharp, the natural symbol will be displayed, and if the last played note of that letter in the measure is natural, or there is no previous note of that letter, no symbol is shown.

For indexing into our memory, two calls are made. One is for notes/rests, and the other is for sharps/naturals the addresses come in the form:

$$\begin{aligned}&(\text{frame\_of\_sprite}) \cdot 3200\\ &+ \text{row\_of\_block} \cdot 32\\ &+ \text{column\_of\_block}\\ &+ \text{vertical\_displacement} \cdot 32\end{aligned} \tag{5}$$

When receiving the data from the memory calls along with knowledge of the staff lines, the image_sprite module outputs the staff line data, sharp data, and note data anded so that when any of them is low, the screen shows black in that pixel.

Two series of example notes have been made to demonstrate the system working Fig. 8, Fig. 9. These are generated with microphone data.

## VII. Results

### A. System performance

The music transcription system is able to consistently identify music tones C4 to A5. It is also able to approximate note durations. The system is capable of identifying pure tones with high accuracy, and non-pure instrument-produced tones with slightly lower accuracy. While the system is able to *identify* the range of tones we aimed to, it seems to struggle with *detecting* the tones. Tones often have to be played directly into the microphone, or else the system will not detect it.

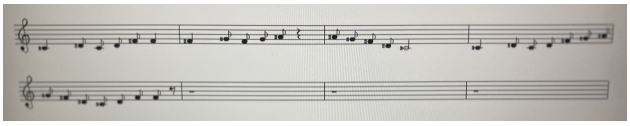


Fig. 7. Korean Folk Song: Arirang. This demonstrates sharp and natural logic consistent with real sheet music.

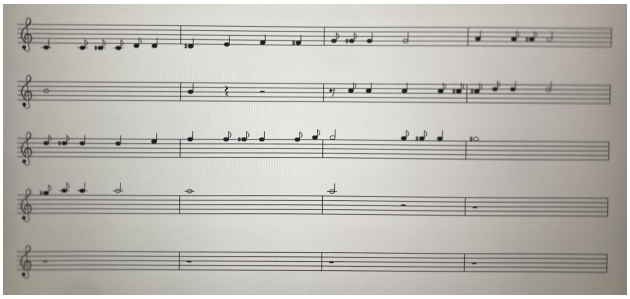


Fig. 8. All notes that can be displayed. From C4 to A5

Fig. 9. Hot Cross Buns in C Major.

### B. Memory Utilization

Our project is visual heavy, and so we made sure to be conservative with our memory. From the post_synth_timing.rpt file, we observe the system to be utilizing only 12.67% of total BRAM memory. BRAMs we used in our project include:

- Hanning window coefficients: 28 * 4096 bits
- note sprite storage: 32 * 1100 bits
- played notes storage: 48 * 20 bits
- FFT IP: 11 BRAMs

We save significant memory with the note sprite storage, as sheet music is only black and white, and so we can store the sprite colors as one bit values, which also eliminates the need for a palette to be stored in the memory. Therefore, a $32\times100$ dimension sprite can be represented with 3200 bits.

### C. Timing

In the audio processing pipeline, we noticed that we had mild negative slack due to the large multiplication calculations necessary from the FIR filters. As a result, we changed clocks from the initial 139.264 MHz to 69.632 MHz. This brought the audio processing section's slack to positive 4.315 ns.

When integrating the visual display portion of the project, we had significant negative slack. This mostly arose from the combinational logic required to render the measure logic as well as clock mixing. A large portion of the slack fix was making the note memory a BRAM rather than its initial implementation, which was a 2D array. Accessing and slicing into this array was expensive, and we reduced the slack from -25 ns to -5 just with that fix alone. The rest of the fixes were due to clock mixing. We fixed the clock mixing (the 74.25 MHz pixel clock and the 69.632 MHz microphone clock) by running the always_ff block in image_sprite at the 69.632 Mhz clock. The clocks are reconciled by using the note_mem BRAM, where its first port that we use for writing is clocked at 69.632 MHz, whereas its second port used for reading is clocked at 74.25MHz. We were able to bring the slack to -0.070 ns, so it's still negative, but it's much closer to what we want. Regardless, the system works as expected even with the negative slack.

## VIII. Takeaways and Next Steps

One issue we faced when testing the system with a real piano is that the system would only hear the initial press of the key, and then fail to detect the resonating tone of the rest of the

press. These are likely related issues and could be fixed in the future by using a better microphone, perhaps an external one that can be placed closer to the piano. We could also consider better bit selection methods out of the FIR filter, since maybe we are filtering out critical data in that last stage when we take the top 8 bits.

We can expand the robustness of the system by expanding the range of notes and type of musical symbols that can be displayed. For example, one issue we noticed is when, for example, a two-beat note starts on the last beat of a measure. In such a case, we display two quarter notes, split across the two measures. However, in proper sheet music, a tie would connect these two quarter notes to indicate the note extends across measures. Our existing infrastructure of the code would allow for this to be added relatively easily.

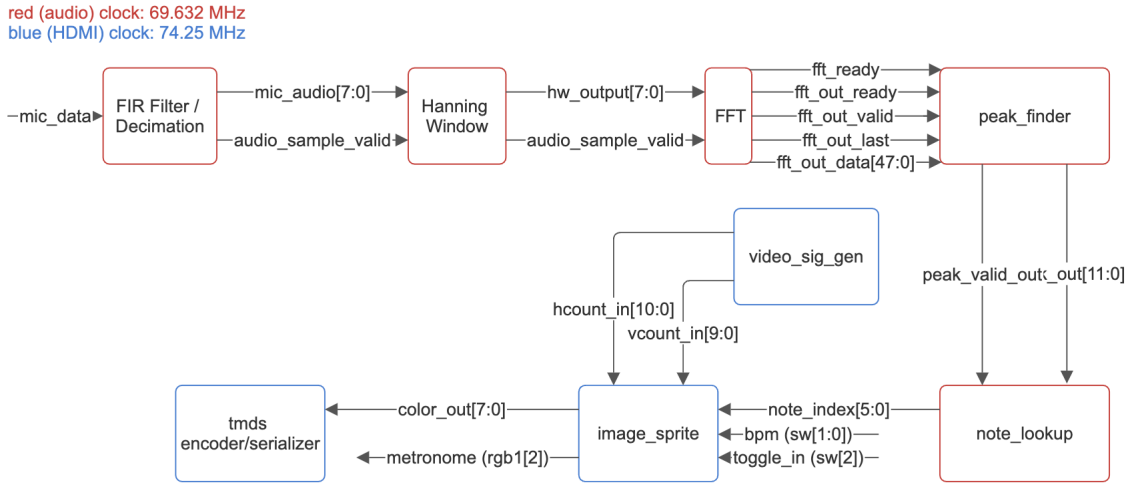
## IX. Contributions

Monica Chan worked on the FIR filters/decimators and note lookup modules. Ian Hueston worked on the note locator, image sprite, and video generation modules. Cynthia Zhang worked on the Hanning window, FFT, and peak finder modules. Monica Chan and Cynthia Zhang spearheaded the project research, design, and planning. Zhang and Chan were also responsible for integrating the audio pre-processing pipeline. Chan and Hueston were responsible for integrating the downstream image sprite and fixing the timing issues. All members contributed to the report and diagrams. Special thanks to Joseph Feld, Adrianna Wojtyna, and Joe Steinmeyer for much assistance in the project and understanding basics of signal processing.

## References

[1] P. Isza, “TFilter - free online FIR filter design,” TFilter - Free online FIR filter design, http://t-filter.engineerjs.com/ (accessed Nov. 22, 2023).

# X. Appendix A


red (audio) clock: 69.632 MHz
blue (HDMI) clock: 74.25 MHz
mic_data
FIR Filter / Decimation
mic_audio[7:0]
audio_sample_valid
Hanning Window
hw_output[7:0]
audio_sample_valid
FFT
fft_ready
fft_out_ready
fft_out_valid
fft_out_last
fft_out_data[47:0]
peak_finder
video_sig_gen
hcount_in[10:0]
vcount_in[9:0]
peak_valid_out
_out[11:0]
tmds encoder/serializer
color_out[7:0]
metronome (rgb1[2])
image_sprite
note_index[5:0]
bpm (sw[1:0])
toggle_in (sw[2])
note_lookup