# FPGAAAuto-Tune

MIT 6.111 Final Project Report

Kika Arias & Elaine Ng

# Motivation

Auto-Tune is an audio effect software program that was created in 1997 by Andy Hildebrand. It has been used by music artists to adjust off pitch singing, but became popularized by noted artists like Cher and T-pain to add overexaggerated effects to their voices. Most auto-tuning systems implement pitch correction through the "Time-Domain Pitch Synchronous Overlap and Add" (TD-PSOLA). This method corrects frequencies of frames of audio by resampling and piecing the resampled frames together. The advantages of TD-PSOLA are its simplicity and computational efficiency.

However, we were inspired by the material we learned in 6.003 to attempt to implement auto-tuning in a less conventional way, by using a frequency domain approach. In this approach, we convert the time-domain signal to its frequency representation in order to identify the dominant pitch and correct it. This approach is not commonly used since it is computationally intensive and requires a large memory storage. But with the resources provided by the FPGA, we wanted to attempt this alternative method of auto-tune.

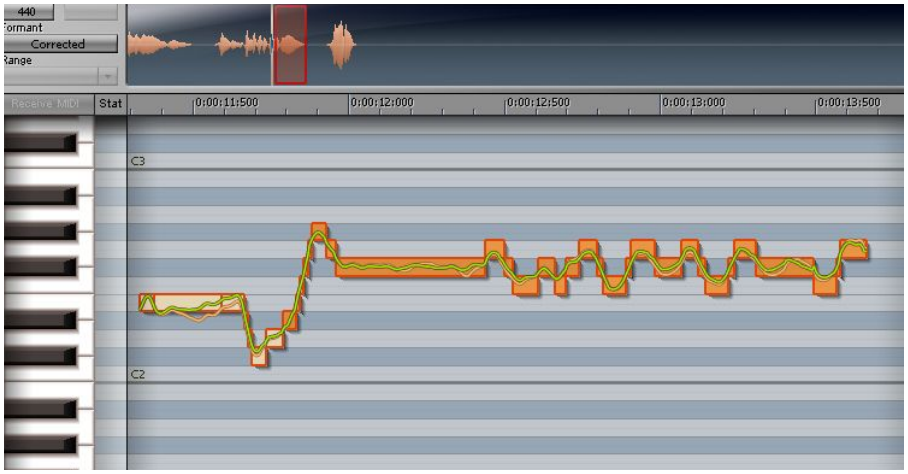


*Figure 1. Waves-Tune plugin for pitch correction (Auto-Tune for $60!)*

# Overview of System

Our system has four main modules which work together to produce an auto-tuning effect. At a high level, the audio is put into the STFT and once we get the frequency content it goes into both a peak detector and memory. The data stored in memory is used to display a graph of frequency over time (spectrogram). The peak detector analyzes the frequency content of one window and finds the frequency bin with the greatest magnitude. Once the maximum is identified, a sine tone at that frequency is generated and output to a speaker.

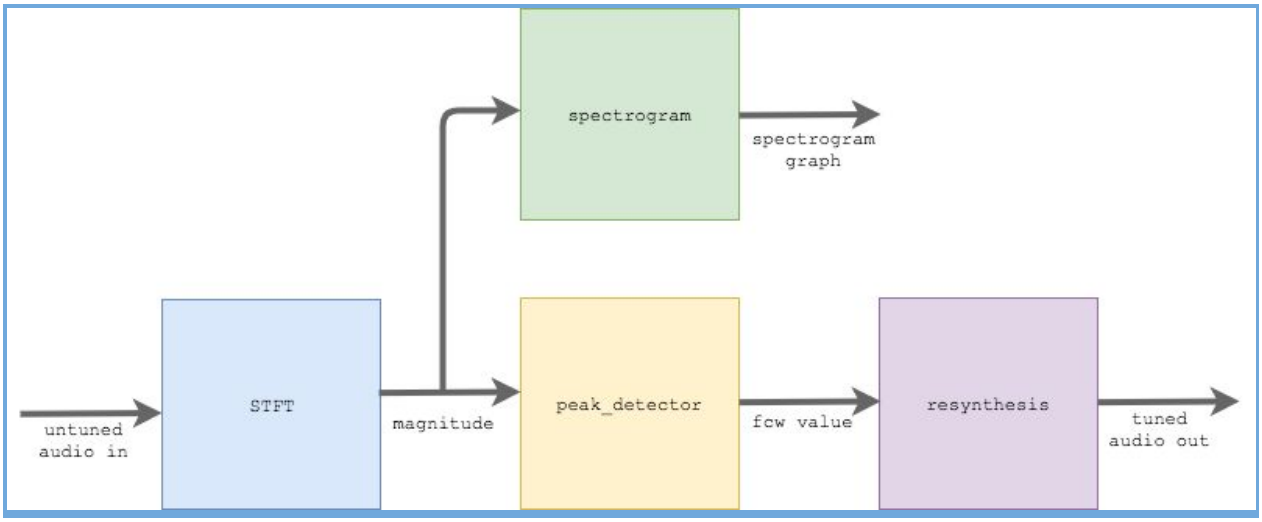


*Figure 2. High level block design describes how the system goes from audio input to visual display and audio output*

In addition to this basic system, we designed other features for testing and adding fun audio effects to the final result. We utilized the FPGA's switches to toggle between generated pure sine tones at different frequencies and microphone audio as inputs to the system. We also created effects that output a harmonized chord, a sine tone an octave higher, and a sine tone an octave lower than the original signal.

Together, all of these components give us one system that takes in an audio signal, rounds it to the nearest note on the western scale, and outputs a generated sine tone at that note's frequency. For this project Elaine was responsible for the STFT, resynthesis module, and the extra audio effects, while Kika was responsible for the peak detector module, audio inputs and outputs, and visualization. While the modules were developed separately, we spent a majority of our time collaborating together in order to help each other debug and understand how inputs were moving through our system as a whole. We believe this allowed us to have a relatively smooth integration period, and better cohesive understanding as a team. In the following sections, we will describe each of the four main components of our system and the different challenges we faced as we were developing this project.

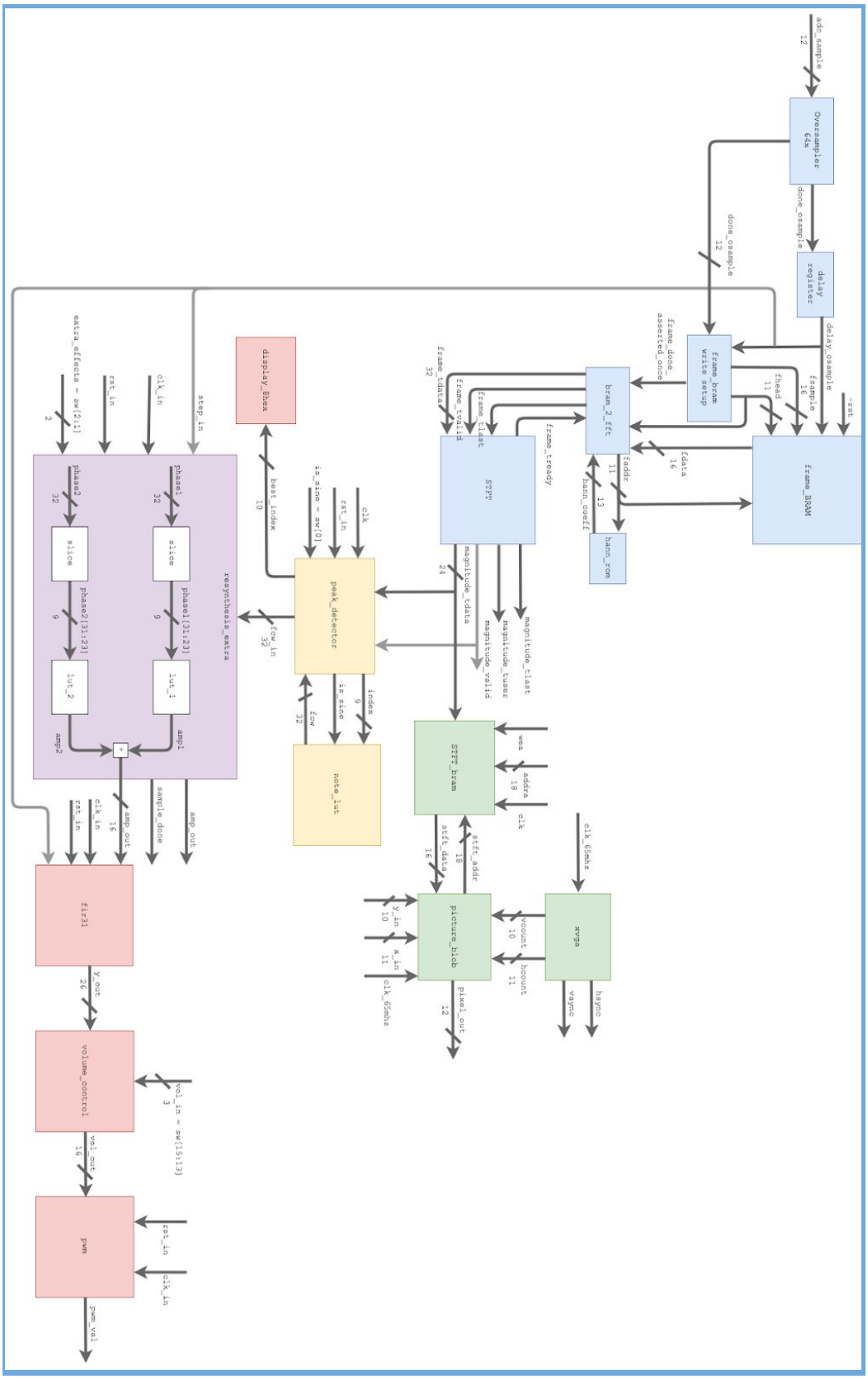


*Figure 3. Total system  block design describes how the entire system is integrated.*

# Details of Main Modules

## Frame Construction and STFT (Elaine)

We ultimately decided to implement real-time auto-tuning by processing the samples of input audio in frames of length 2048-samples. Our audio processing is based on determining the dominant frequency in each frame through the Short-Time Fourier Transform (STFT). Essentially, instead of saving all of the samples and taking the FFT of the entire signal, we can obtain local frequency information by just looking at the Fast Fourier Transform (FFT) of each frame. The *STFT_fsm* module is based on the NEXYS-4 FFT demo that was provided to us [1]. On a high level, this module oversamples the incoming data by 64x, builds up a 2048-sample frame in BRAM, and performs FFT on the frame.

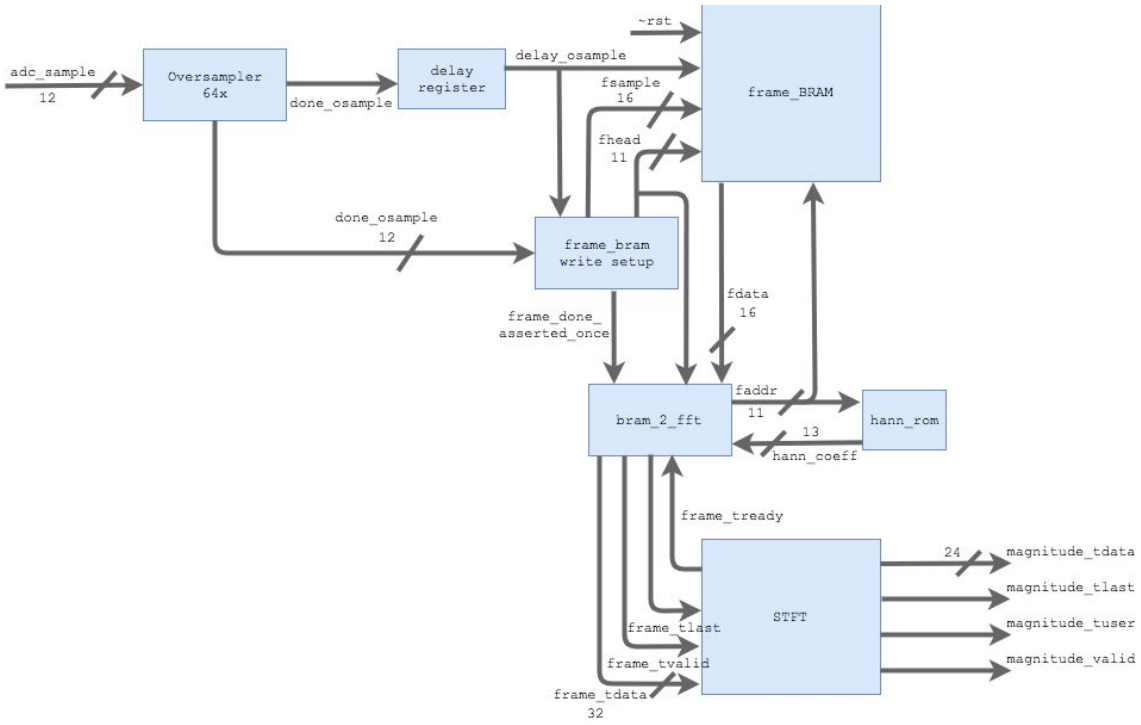


*Figure 4. STFT_fsm module block diagram with submodules.*

### Oversampling

Although samples are coming from the ADC at 1Msps, we needed the sampling rate of the signal into the FFT to be much smaller to meet a target ~8Hz resolution since frequency resolution is sampling rate/transform length. (8Hz is just enough to differentiate C#3 and D3, which is around the bottom range for a trained bass singer). The demo uses a 4096-point FFT, but to halve the size of the frame BRAM and the time to construct the frame, we used a 2048-point FFT. With a transform length of 2048 and 8Hz resolution, the sampling rate had to be downsampled by a factor of 64 (a target sampling rate of ~15.625kHz).

Instead of downsampling (taking a sample every 64 sample triggers and 12'd0 otherwise), we used the oversampler method in the demo. To oversample the signal by 64x, the signal is sampled 64 times the target sampling rate and averaged over the extra cycles. The *oversampler* module outputs this averaged sample $(osample)$ every 64 sample triggers and effectively makes the input 1MHz sample trigger the faster 64x sampling rate. Besides improving our frequency resolution, oversampling by 64x also increases the incoming 12-bit precision to 15 bits of precision. The extra 3 bits comes from the 8x increase in signal-to-noise ratio (SNR).

The oversampled sampling rate of 15.625kHz meant we now had a Nyquist frequency (half of the sampling rate) of 7.8125kHz. This is the maximum frequency that our system could produce before aliasing occurs. To avoid aliasing, we limited our input audio to signals with frequencies that humans could sing. The highest note we were concerned with was C6 and corresponds to a frequency of 1046.5Hz, well below the Nyquist frequency.

For testing purposes, we also made an option for the *STFT_fsm* to be able to take pure sine waves generated from the FPGA itself and a 16x oversampler. Samples from the *sine_generator* module were not limited to the 1Msps sampling rate, and the 16x oversampler reduced the wait time by 4x. The 64x oversampler made simulations run for a long time, because we had to wait for 64 sample triggers (6400 clock cycles).

### Building the Frame

Most of the logic of the *STFT_fsm* module is for writing the samples in the frame BRAM and negotiating done and start signals for the *oversampler* and *bram_2_fft* modules. As an $osample$ from the *oversampler* module is ready, which is represented by the output $done\_osample$ signal, we appended a 1'b0 and wrote it to frame BRAM. Because during the first 2 sample triggers the *oversampler* module does not output valid data, we delayed the $done\_osample$ signal by 2 sample triggers using a shift register, and this delayed signal became the write enable signal for frame BRAM.

In the demo, the start signal for the *bram_2_fft* module to read samples from frame BRAM to the FFT block was a $vsynch$ signal because the FFT only needed to update as fast as it needed to be displayed on a VGA monitor [1]. Since we were going for real-time audio processing, we wanted the frames to be sent one after the other, meaning at a sampling rate of approximately 104MHz/2048 samples. When a sample is written into frame BRAM, an 11 bit $sample\_counter$ is incremented. A $frame\_done$ signal is asserted when $sample\_counter$ is 2048 and frame BRAM is full. We could not make this $frame\_done$ signal the start signal of *bram_2_fft* exactly, because this $frame\_done$ signal is high when the sample_counter is full and 64 sample triggers after that. This 6400 clock cycle pulse is much longer than the time needed for the FFT block to perform its calculation. To avoid having multiple spectrograms per frame, we made the start signal $frame\_done\_asserted$ which is the rising edge of $frame\_done$.

## FFT

The *bram_2_fft* module handles the input into the FFT block and handshake signals of the FFT block. The module is normally in a $!sending$ state. When the start signal is asserted, the read address $(addr)$ and the number of samples sent $(send\_count)$ are resetted. The module then goes to the sending state where $addr$ is incremented and the samples from frame BRAM are passed as inputs until the last sample in memory is read. Before *bram_2_fft* sends the sample to the FFT block, it is multiplied by the coefficient in the Hann window corresponding to its address. We chose to multiply the frame by a Hann function to smooth out the edges of the frame. Without the Hann function, the discontinuities introduced by windowing with a rectangular pulse, which is equivalent to our process of creating the frame, would show up as unwanted artifacts in the FFT.

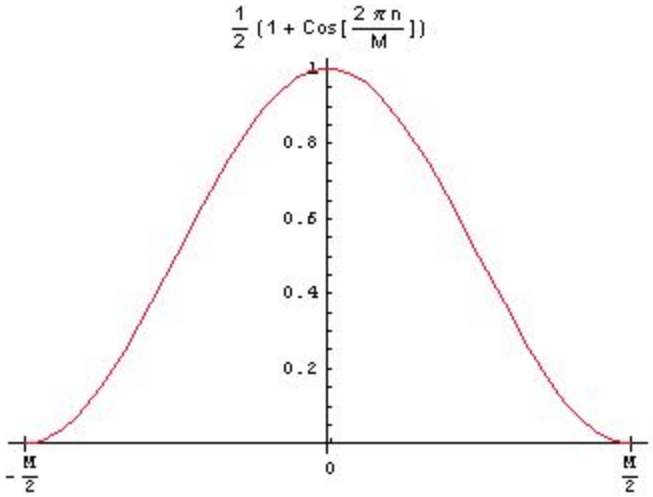


*Figure 5. Hann window function*

The FFT block design is taken from the demo. Since there was not much documentation about the provided block design, we spent a long time trying to understand how this all worked, and we would like to provide additional details we figured out for the reader. On a high level, the FFT block consists of several IP blocks: an FFT core, 2 slices, 2 multipliers, an adder, a square root CORDIC, and an AXI-4 register slice to delay the control signals from the FFT until the CORDIC receives data. Data transfer for the FFT core and CORDIC are handled through the AXI protocol in which data is only sent when both the source of data's VALID signal and the receiver's READY signal are both high. The handshake signals for reading data from frame BRAM to the FFT $(frame\_tvalid$ and $m\_axis\_data\_tready)$ are handled by *bram_2_fft*. The AXI-4 stream register slice controls the VALID signals of the CORDIC. Eventually, the CORDIC's $magnitude\_tvalid$ signal is used by modules like *peak_detector* to indicate when the *STFT_fsm* module is outputting valid data.

There are also ways to configure the FFT core besides changing the transform length [2]. We could decide to use a "Pipelined, Streaming I/O" architecture or "Radix-Burst" architectures. The Radix-Burst architectures result in a smaller core size but more latency. We chose to use the Pipelined, Streaming I/O architecture at the expense of core size because it allows us to

simultaneously process the current frame, load a new frame, and output the results from the previous frame. This significantly reduces latency which is we need for realtime auto-tuning. There is also an option to make the output "Bit/Digit, Reversed Order" or "Natural Order" [2]. We chose to keep the output in "Natural Order" i.e. have the output in increasing index order. Making the output "Bit/Digit, Reversed Order" can slightly reduce the latency, but we still had to convert the output back into "Natural Order" for modules downstream. We also kept the Throttle Scheme its "Non-Real Time" option [2]. If we had used the "Real Time" Throttle Scheme, the FFT core would no longer be using standard AXI protocol.

The FFT block also takes a configuration signal, $s\_axis\_config\_tdata$ whose format is shown in Figure 4 [2]. The only necessary signal is the FWD/INV signal, which is 1 for the forward FFT and 0 for the inverse FFT. Our project only uses the forward FFT, so we set $s\_axis\_config\_tdata$ to always be 1.

1. (optional) NFFT plus padding
2. (optional) CP_LEN plus padding
3. FWD/INV
4. (optional) SCALE_SCH

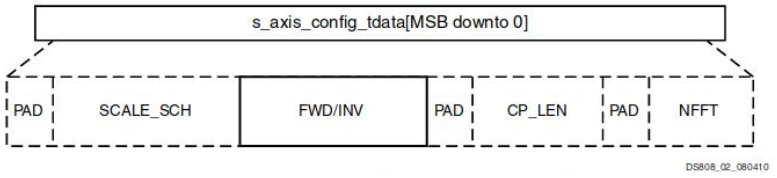


*Figure 6. $s\_axis\_config\_tdata$ format from FFT Core Documentation*

The FFT core has an input data width of 16 bits, which means it represents the real part of the FFT in 16 bits and the imaginary part of the FFT in 16 bits. To be precise, the FFT core expects a 32 bit input and a 32 bit output, where the top 16 bits represent the imaginary part and the lower 16 bits represent the real part [2]. Since we were dealing with real audio signals, we padded the incoming 16-bit samples from frame BRAM with 16 zeros. The blocks after the FFT core separate the real and imaginary parts of the output of the FFT with the slices and square each part with the multipliers. The squares are summed in an adder, and the sum is sent through a square root CORDIC to obtain the magnitude of the FFT. This magnitude is written into STFT BRAM which the *spectrogram* module accesses, and is also directly passed to the *peak detector* module.

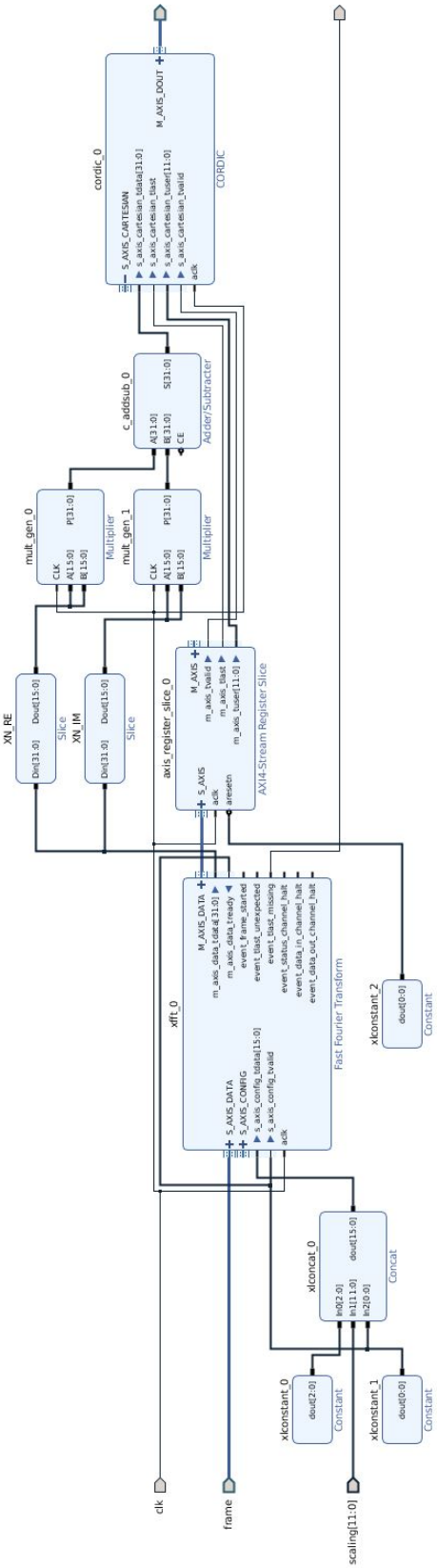


*Figure 7. FFT Block Design.*

# Peak Detector (Kika)

Once the STFT is done processing a frame of data, which is signaled by the assertion of $t\_valid$, it will start streaming out one bin of data every clock cycle. These 2,048 consecutive frequency bins are input directly to the peak detector, which compares the values and tracks which bin contains the dominant frequency. Once that bin number is identified, we use a look-up table to trace it to a frequency control word ($fcw$) which is used to increment the phase accumulator in the later resynthesis module.

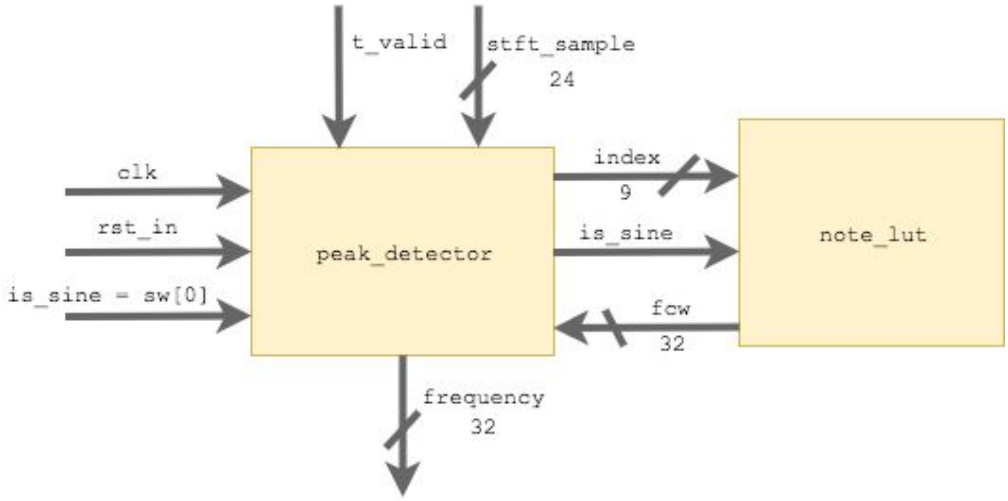


*Figure 8. Peak_detector module block diagram showing interaction with note_lut*

## Peak Detection

The peak detector module takes in a 24 bit value which represents the magnitude of the FFT at one frequency bin. It relies on $t\_valid$ to dictate when to start storing and comparing values. The peak detector stores 3 values in registers: $current\_val$, which is the value at the current time step, $prev\_val1$, which is the value at the previous time step t+1, and $prev\_val2$, which is the value at t+2 two time steps prior. There are eight separate conditions that must be met in order for the peak detector to set a bin and its value to be the dominant frequency bin for that window. If there is no such bin that meets this criteria for a window, the peak detector will keep the previous window's highest magnitude as its maximum.

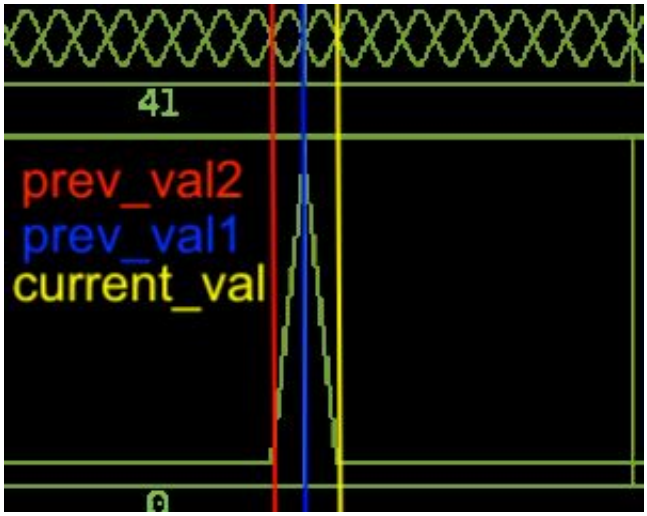


*Figure 9. Graphical relationship between the locations of prev_val1, prev_val2, and current_val in the magnitude_tdata output*

The first two conditions constrain the bin number as being greater than 10 and less than 141. Setting a lower bound on the bin number allows us to avoid setting bin 0, which represents the DC offset, and its neighboring bins as the maximum. We assume that any frequency below 76 Hz will not be intentionally sung or played into our microphone and thus the peak should never appear in the first 10 bins. For the case of bin 0, the DC offset is not indicative of the frequency content within the signal itself. Even if the value in bin 0 is higher than all other bins, since it does not represent any frequency it should not be taken into consideration for our peak detector. Bin 141 is chosen as the upper bound since we chose 1046.50 Hz as the highest frequency we wanted to detect. Although bin 141 represents a frequency that is above 1046.50 Hz, having it as our upper limit gives us the ability to tune notes higher than 1046.5 Hz down to our maximum.

The next three constraints are on $prev\_val1$. It is compared against two other values, $highest\_val$, which is the largest magnitude seen up to that point in the window, and $prev\_highest\_val$, which is the largest magnitude seen in the previous window. $prev\_val1$ must be greater than $highest\_val - 5000$, have a value of greater than $prev\_highest\_val - 2000$, and generally have a magnitude greater than 15,000. These thresholds were tuned by probing the output of the STFT with the ILA and serve to make the system resistant to noise (further rationale is provided in the challenges section).

The final three constraints are on a value labeled as $current\_sum$, which is the sum of $prev\_val2$ and $current\_val$. $current\_sum$ represents the two values on either side of the peak. Since with the FFT there is a smearing effect caused by windowing, we used this to our advantage as an additional characteristic to help identify true peaks. When looking at the ILA output, a consistent trend showed that when we probed a bin that should contain a true peak, the magnitudes proceeding and following that bin have higher than normal values. Occasionally, due to noise (or

possibly aliasing), there were bins in the same window that had higher magnitudes than the bin we can calculated to have the highest value. However, the bins that fell on either side of these unexpected peaks would have significantly lower values than the true peak. For this reason, similar to the thresholds used for $prev\_val1$, $current\_sum$ is compared against $highest\_sum$ and $prev\_highest\_sum$ which represent the largest sum seen thus far in the current window and the previous window respectively. $current\_sum$ must be greater than 30, $highest\_sum$, and $prev\_highest\_sum - 5$. These thresholds were also set by probing the ILA.

If all of these constraints were met, the $highest\_val$, $highest\_sum$, and $best\_index$ are set to the $prev\_val1$, $current\_sum$, and $counter - 1$ respectively. When $counter$ reaches 2048 (the number of samples in a window), $highest\_val$, $pre\_val1$, $prev\_val2$, and $current\_val$ are reset while $prev\_highest\_val$ and $prev\_highest\_sum$ are updated. The note_lut module is initialized within the peak detector module and it takes $best\_index$ as an input and outputs an $fcw$ which is streamed to the resynthesis module to generate a sine wave.

### Note LUT

The note_lut module interacts with the peak detector by taking in a 9 bit index and outputting a 32 bit $fcw$ value. There is an additional input is_sine, which is 1 when our input is a sine wave output by the sine generator, or 0 if the input is coming from the external microphone. Based on the input $is\_sine$, note_lut outputs a different $fcw$ to accommodate the differences in sampling rate between the two inputs. The $fcw$ is output back to the peak detector since it is contained within the peak detector module.

The lookup table was generated by a python script and allows us to map bins to frequencies ranging from B2 to C6 (123.47 Hz to 1046.50 Hz). Each bin has a width of 7.63 Hz, which is calculated by dividing the sampling rate ($f_s$) of 15,625 Hz by the number of bins. This formula is derived by understanding that the range of frequencies goes from $-\frac{f_s}{2}$ to $\frac{f_s}{2}$ since the maximum detectable frequency is the nyquist frequency, and the lowest detectable is the negative of that same frequency. The total range of frequency is thus fs, and this makes the size of each bin 7.63 Hz. This bin size defines what the lowest detectable frequency is for our system. Taking into account numeric rounding, the lowest change in notes we can detect is B2 to C3, which is why this is the lower threshold for our LUT. In our system, any note below this threshold is mapped to B2. While the definitive upper bound on the highest frequency we are capable of detecting is approximately 7,800 Hz, we decided to cap it at C6 since 1046.5Hz is generally considered to be the upper end of the vocal range for a classically trained soprano singer.

This lookup table allows us to make a conversion from a bin number, to a frequency, to an $fcw$. Using python, we combined the last step of converting from frequency to $fcw$ to avoid the computational cost of creating an additional frequency to FCW lookup table. (The calculations used to convert frequencies to $fcw$ values will be discussed in the next section on resynthesis).

We include two different $fcw$ values for each frequency in order to accommodate the different sampling rates of the two types of inputs. To convert bin number to frequency, for the table's creation, we multiplied the bin number by 7.63 Hz (the width of the bin). Then, this frequency was mapped to its nearest frequency in the standard western scale [3]. Since the highest frequency we plan on detecting is 1046.5 Hz, if a majority of the frequency content lies above that value (in a bin number higher than 133), then it gets mapped to 1046.5 Hz, which is encapsulated by the default case. A similar logic was used for accommodating our lower frequency threshold.

# Resynthesis (Elaine)

The *resynthesis* module constructs a new signal for a frame at the corrected frequency in sine tones. It takes an input $fcw$ (corresponding to the correct frequency) from the *peak detector* module, and uses it to adjust the step size the $phase$ (an index into a sine lookup table) is incrementing by.

A note about timing here: the system is always playing back something, and the *resynthesis* module is always outputting a sine wave at the audio sampling rate of 15.625kHz (if the input comes from the mic) or 16.250kHz (if the input is from the FPGA sine generator). The durations of notes in the input signal are preserved because the samples passed through *peak_detector*, *resynthesis*, and *playback* all maintain the frame structure created back in *STFT_fsm*. All these modules after the *STFT_fsm* take less time than the time *STFT_fsm* needs to construct the next frame. So the FFT calculation, peak detection, resynthesis of corrected signal for the current frame happen before the next frame is even constructed, thus this system outputs the corrected audio at least frame-by-frame. If there are fast note changes that happen within a single frame, this system cannot faithfully resolve or correct those notes.

## Sine Generator

The *resynthesis* module is mostly the *sine_generator* module from Lab 5A, which generates an entirely positive sine wave from a 32-point, 12-bit lookup table. Because the sine wave would go through a 64x oversampler, we needed more than 64 points in the table. Our table has 512 points with 12-bit depth, which means it has values from 0 to 4095. A sine wave at a specific frequency corresponds to cycling through the table at a particular step size [4]. This step size is related to how the sine generator represents the continuous range of angular values between 0 and $2\pi$ as integers from 0 to $2^{32}$. A 32-bit register $phase$, which stores the current phase, is incremented by $phase\_incr$ at every clock cycle. The correct phase increment is given by the frequency control word or $fcw$:

$$fcw = \frac{2^{32}(target\ frequency)}{sampling\ rate}$$

For the *resynthesis* module, the peak detector provides the $fcw$ corresponding to the correct frequency. At each clock cycle, the top 9 bits of $phase$ is used to index into the sine lookup table.

## Extra Effects

We also implemented a few extra audio effects for our stretch goal. Depending on switches 1 and 2, we could have: (0) no effect and normal playback, (1) harmonize by a major third, (2) chipmunk effect, and (3) Darth Vader effect.

To harmonize by a major third, resynthesis outputs a sample that consists of an amplitude of the sine wave at the correct frequency and an amplitude of a sine wave of the frequency a major

3rd above the correct frequency. For a fixed frequency $f$, the frequency corresponding to a note major 3rd above is $\frac{5}{4}f$. Since $fcw$ is proportional to $f$, this means that we just need to grab sine values from a second sine lookup table with the phase incrementing by $\frac{5}{4}(fcw)$. The chipmunk effect is playback at an octave higher than the correct frequency. For a fixed frequency $f$, the frequency corresponding to note an octave above is $2f$. The module grabs values from a single sine lookup table with the phase incrementing by $2(fcw)$. Finally, the Darth Vader effect is playback at an octave lower than the correct frequency. For a fixed frequency $f$, the frequency corresponding to a note an octave below is $\frac{f}{2}$. Thus, the module grabs values from a single sine lookup table with the phase incrementing by $\frac{fcw}{2}$.

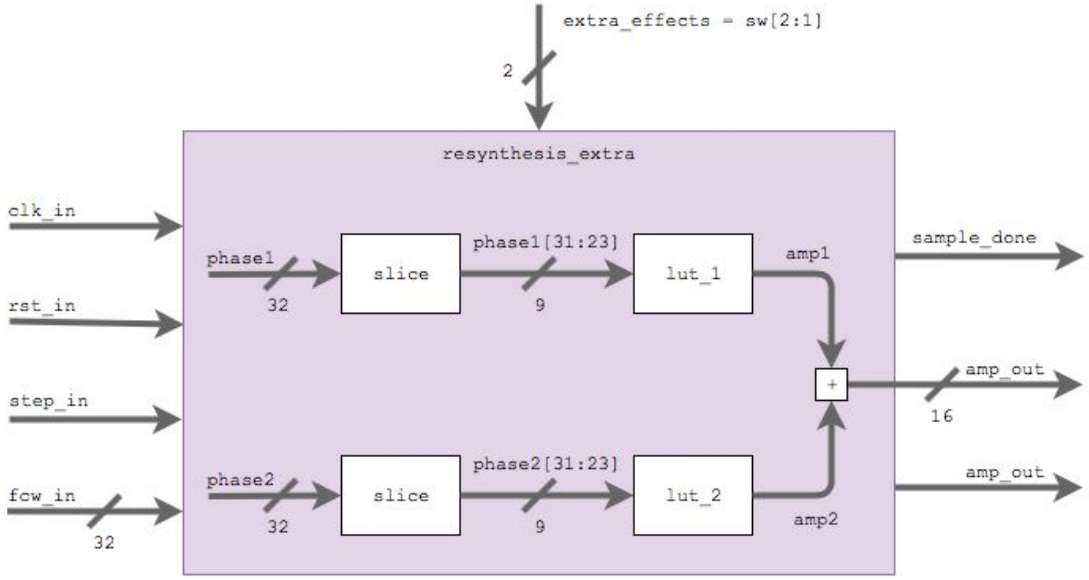


*Figure 10. Simplified resynthesis module.*

# Visualization (Kika)

The visualization module communicates with the STFT bram through the toplevel module by requesting addresses and receiving a 16 bit value as an input. That value is then mapped to a color simply by splitting the lower 12 bits into three groups of 4 bits for red, green, and blue. Colors with a red or yellowish hue map to a higher magnitude, while blue (specifically dark blue) represents lower magnitude values. The color values for each pixel is then output to the display via the xvga module provided to us in a prior lab.

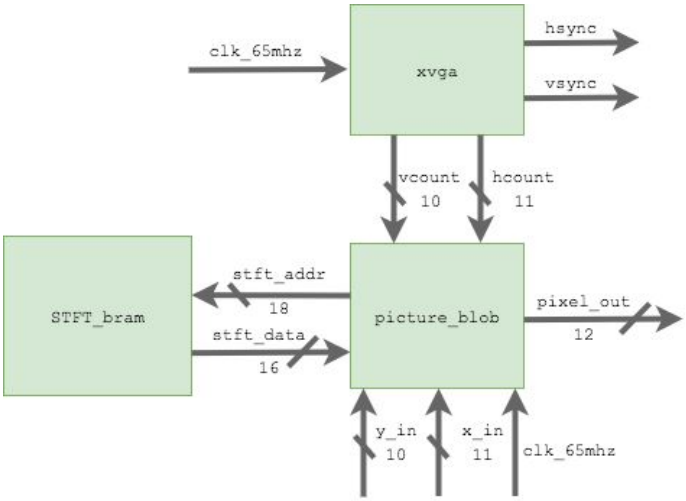


*Figure 11. Block diagram of how the spectrogram visualization interacts with xvga and the STFT_bram*

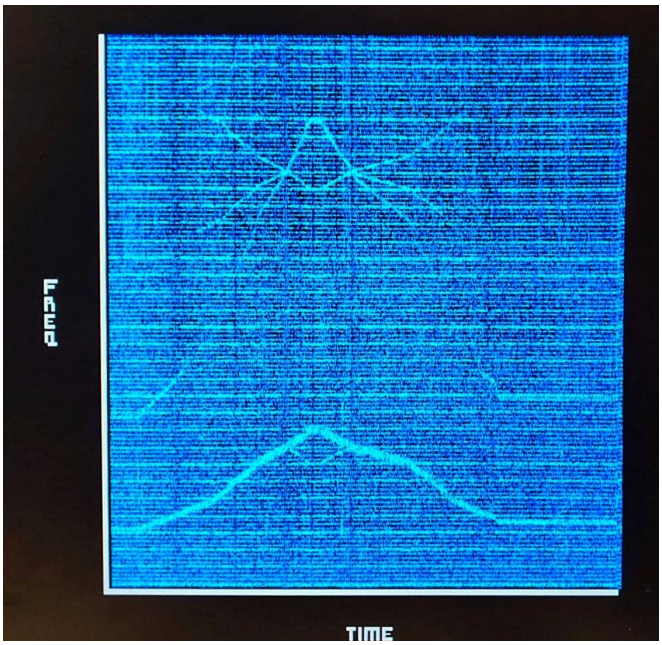


*Figure 12. The spectrogram drawn on the monitor after performing an upward and downward frequency sweep with a single frequency played at the very end. The thicker light blue represents values with the highest magnitude. Each pixel represents a single frequency bin.*

## STFT BRAM

The STFT bram has a depth of 262,144 values. This corresponds to an image of 512x512 pixels in size. This size is the largest we could make the spectrogram since both height and width have to be a power of two, and we were constrained by the size of the monitor which is 1024x768 pixels. When the STFT begins outputting a window of data, each bin is stored in consecutive positions in the bram. Only the first 512 bins are stored, since the relevant frequency content is present in these bins. An alternative strategy would be to downsample the bins in a window by 4, however, since each bin is represented by a single pixel, we felt as though downsampling had the potential to erase important information from the spectrogram and since the highest bin considered by the peak detector is bin 140, drawing the first 512 bins of a window would be a sufficient representation of the relevant STFT data. Once 512 windows have been written into bram, the STFT bram simply starts overwriting the existing values, which provides our spectrogram with the ability to draw and update itself in real time.

View the Spectrogram Drawing Video

*Watch the generation of the spectrogram as you hear the audio sweep up and down. The spectrogram draws in real time and accurately depicts the rise and fall in the frequency over time.*

## Request Address

Since the spectrogram values are stored in a one dimensional format, while representing a two dimensional image, we need to have a way to map $hcount\_in$ and $vcount\_in$ to a specific address in the STFT bram. Since the vga display first draws a horizontal row, and then moves to the next line, while the STFT bram is written to in columns, the formula for the request address is as follows:

$$specgram\_request\_address =$$

$$= ((hcount\_in - x\_in + 1) * (512) - (vcount\_in - y\_in + 1 + 270) + hcount\_in)$$

With this holistic formula, it can be difficult to see how $hcount\_in$ and $vcount\_in$ relate to the address. A simplified version looks as follows:

$$specgram\_request\_address = ((hcount\_in + 1) * (512) - (vcount\_in + 1))$$

The top left corner of the spectrogram should correspond to index 511 in the STFT bram, while the bottom left corner corresponds to index 0. The next column begins with bram index $specgram\_request\_address = 512 * 2 - 1$ and ends with $specgram\_request\_address = 512$. This pattern repeats as we move through the spectrogram image and is the rationale for the derivation of this equation.

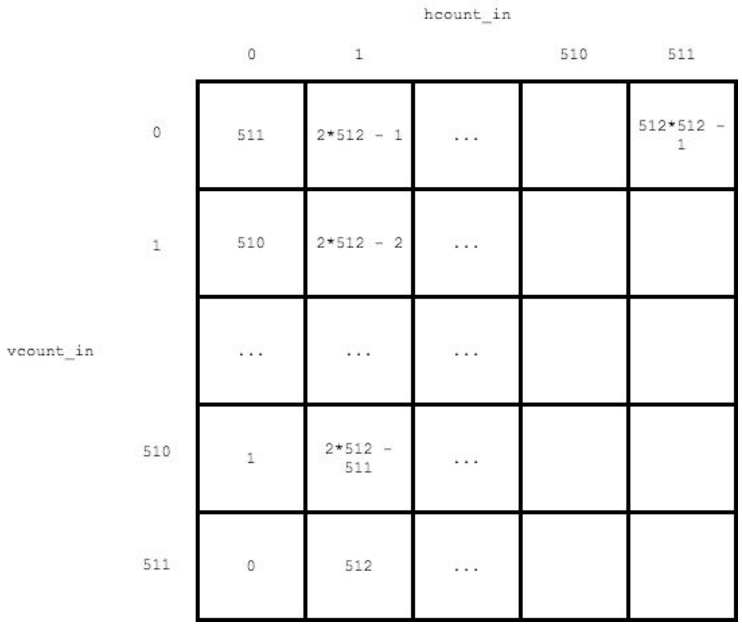


*Figure 13. As hcount_in and vcount_in change, the desired output of the specgram_request_address follows a distinct pattern which allows us to derive the previous equation.*

The additional $hcount\_in - 270$ factor which is added to the original equation is to help offset any latency that is added to the system from having to fetch the address from bram and draw it to the monitor. This will be discussed further in the section on challenges.

# Challenges

## Elaine

Most of my challenges came from the *STFT_fsm* module. When I first began working on this project, I thought this module would be much easier to implement because it was largely based on the FFT demo. In reality, my efforts in this project was spent making this module work to a higher level of precision that the rest of the modules in this project downstream required. This module was particularly difficult to debug because it would take so long to run simulations. At the sampling rates and transform lengths the final system uses, because the frame construction takes at least 6400 clock cycles, the simulation would sometimes take close to an hour to run. That is why for testing purposes, I also made a 16x oversampler and the option of using the input sine generator. When we finally could run simulations on simple signals like DC signals and pure sine waves, we noticed that the FFT output would not be correct even after multiplying the windows by a Hann function. As shown in Figure 6, the output of the FFT of a sine wave (which should just be two symmetric peaks with some smearing) is a bunch of peaks spaced periodically. I was confused why I got this behavior because we were smoothing out the frames and avoiding aliasing (by using pure sine waves at frequencies lower than the Nyquist). The correct peaks were always there, but usually the FFT gave more peaks than it should have. Sometimes those extra peaks were comparable, even larger than, in magnitude to the correct peaks, so it made the *peak_detector*'s job more difficult. The last thing I tried was to add an input FIR filter just after the oversampler to certainly get rid of frequencies that would alias. Unfortunately, the filter kept cutting out too much of the signal even at very low cutoffs, so I could not get that to work.

If I had more time, I would probably have delved more deeply into the timing of the different modules that made up *STFT_fsm* module and Xilinx's documentation on the FFT core. Many parts of this module were still black boxes, and I probably would have an easier time debugging if I knew more deeply what every part was doing. We realized while integrating that there were timing issues, but by then we had so many modules to test to figure out where those came from. I fixed a few timing issues such as the extra delay for the oversampler to finish before samples could be written to frame BRAM through test benches for the separate modules that made up *STFT_fsm* module. During integration, we could not feasibly run simulations anymore, so we tried to use the spectrogram, the segment display, and the ILA. Often what we saw on these different displays would mismatch, and it was difficult to pinpoint where the errors and extra delays came from.

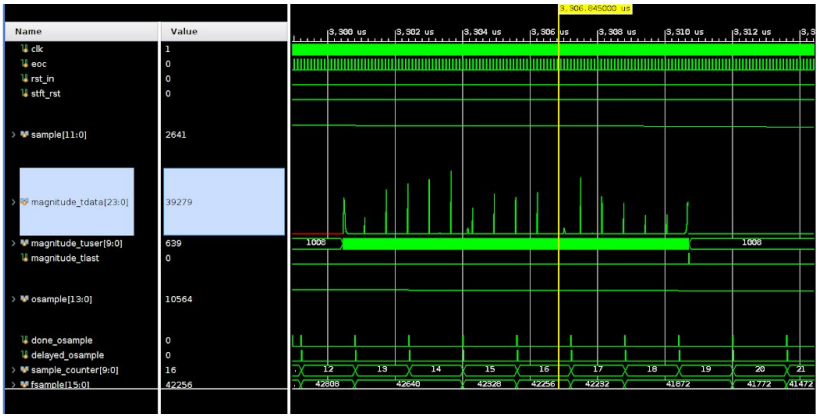


*Figure 14. Simulation of the output of STFT_fsm with a sine wave input. Notice the extra periodic peaks.*

Kika

I think a lot of my challenges began occurring once we started integrating the system. Originally I had a very simplified view of my modules, and while each one worked exactly how I had expected when it was on its own, once plugged into the larger system, timing delay and the variability of the FFT outputs forced me to have to think of "hacky" solutions to the problems I encountered. The $hcount\_in - 270$ factor that is added to the $specgram\_request\_address$ in the visualization module is one of the first examples of this. Although in testing the spectrogram was drawing horizontal lines (which is the expected output from a sine wave), with the output of the STFT it began drawing diagonally.

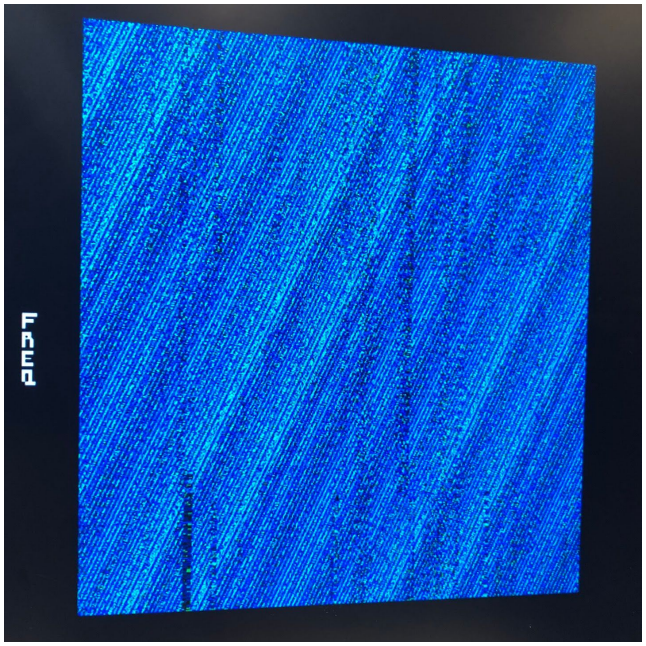


*Figure 15. A diagonally drawn and very bad looking spectrogram which is likely the result of timing created by timing delays and the noisy output of the FFT core.*

Finding a solution for this problem was not a major issue, however, realizing that this was a problem took a significant amount of time since we relied on the previous testing to verify that the spectrogram would still work. I'm not entirely satisfied with this hack as a solution to the diagonal drawing. We believe that the source of this problem is pipelining, but without sufficient time to investigate, we are unable to provide a proper solution to this problem. I believe the hack solution has left the spectrogram with horizontal line artifacts which appear to be present even when there is no audio input whatsoever.

As discussed in Elaine's section, the multiple peaks that were output in one frame by the FFT core make the peak detector's job a lot more difficult than expected as well. Originally the peak detector was just a very basic comparison of the magnitude of each frequency bin in a window to find the maximum. However, even with a constant generated sine wave as an input, the FFT core would output drastically different values every window. This made the peak detector's output flicker around a lot trying to chase down which bin had the exact maximum value.

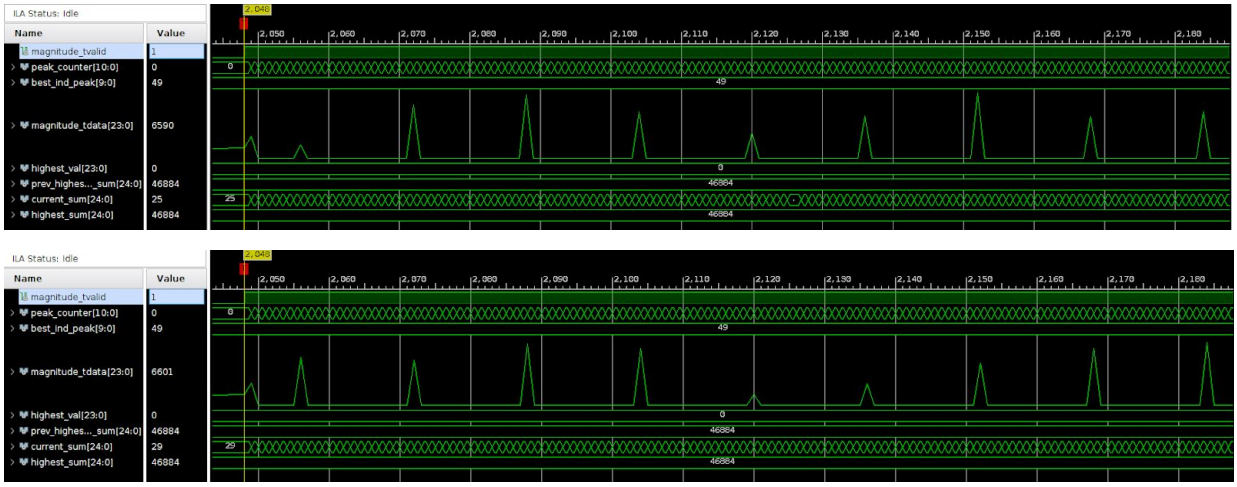


*Figure 16. The FFT core output magnitude_t_data shows different values for two frames generated by a constant sine wave input. In these two examples, the bin number of the highest peak changes (best_ind_peak does not change due to the constraints and thresholding detailed in the section on peak detection).*

This prompted the creation of adding in additional constraints to the peak detector in order to make the peak change less often. Although these new constraints accomplished their goal of making the frequency shift less often, they produced an undesirable effect of sometimes not being sensitive enough to intentional changes in frequency. This trade-off is incredibly unfortunate considering that for a true real time auto-tune, the system should be responsive to even subtle changes in frequency. Nevertheless, we chose to stick with a less sensitive outcome since it allowed us to demonstrate the system's overall capabilities. My suspicion is that without the square root cordic, we would have been able to make a better peak detector and even a better spectrogram since the large magnitude values would be even more pronounced. Unfortunately, it was too late when I came to this realization.

# Lessons Learned and Advice for Future Projects

As evidenced by the challenges we faced, we wish we had more time. We have a few regrets, but all in all we enjoyed the challenge of attempting to create auto-tune and learned many valuable things.

1. **Make test benches and simulations for individual modules and as you integrate.** Simulations not only helped us determine if individual modules were functioning by themselves, but also if they worked with other modules. For example, when we were investigating why *STFT_fsm* was not outputting valid data. By looking at all the signals through the different stages in simulation, we were able to trace it to a bit-depth mistake in *bram_2_fft*. When running simulations, it is helpful to use test signals that the FPGA can easily generate and have easy transforms, such as DC offsets and sine waves. It is also helpful to make the audio sampling rate much higher than 16.25kHz and smaller transform lengths.
2. **Timing is important.** Figure out how long individual modules take and make sure you give enough time for each module to perform its logic. It is best to think about timing and additional pipelining as you connect modules rather than blindly integrating everything into a system that glitches. Simulations help with this!
3. **Instead of the oversampler, we wanted to try directly downsampling the input signal and passing it through an input FIR filter.** Although we thought we avoided aliasing, during checkoff Joe mentioned that some of the behavior in the higher frequency region of our spectrogram looked like aliasing. Since this project is mostly concerned with relatively low frequency content, we think it might have been better if we had just downsampled (take a sample every 64 clock cycles and 16'd0 otherwise). We think cutting everything off above the Nyquist using an FIR filter would definitely get rid of aliasing and a lot of the noise we were seeing and probably lead to a cleaner FFT.
4. **Instead of using the square root CORDIC, output the square of the magnitude.** An interesting last minute realization that we made was that the square root CORDIC was potentially reducing our ability to detect peaks and output a nice spectrogram. We kept the CORDIC due to timing issues that we were running into with the AXI protocol. Had we been able to work with the magnitude squared, it would have made our peaks more pronounced, which would give more defined coloration on the spectrogram since we could potentially see red at peaks instead of the light blue. It would also increase the difference in magnitude of maximum values which could have both negative or positive side effects depending on if the FFT is outputting the maximum value in the correct frequency bin.
5. **Cover the microphone circuit with a soundproof box.** The system is actually very sensitive to all kinds and levels of audio. It was difficult to test and auto-tune input signals because the microphone would pick up noise and sounds from around the lab. We think covering

the microphone circuit along with the source of input audio inside a soundproof box would help.

6. **Try resynthesizing a general signal at the correct frequency instead of sine tones.** We had a plan for implementing our stretch goal of a more generalized *resynthesis* module that would require a few changes in the current structure of the system. A general signal has harmonics in addition to the fundamental frequency, so the peak detector would have to detect and pass the peaks and correct $fcw$ for the harmonics as well. Pitch shifting for this general signal means to not only shift the fundamental to its correct location, but also to shift each of the harmonics by an amount that depends on the shift in fundamental, but is different for each harmonic. We were going to create a filter for the STFT of the frame of samples we are trying to tune that consists of "delta functions" at the correct frequency bins for the fundamental and harmonics. Then we were going to convolve this filter with the STFT of the frame, store that result in another BRAM. And through a method similar to the process outlined in *STFT_fsm* the data from this new BRAM would be sent to an inverse FFT core to generate the samples of pitch corrected audio.
7. **Be ok with change and don't be afraid to ask for help.** We think a lot of the integration would have moved a lot faster had we not been hesitant to ask for help, and also more willing to test for bugs and issues in unexpected areas. A majority of the time, we had assumed the problem was contained in one part of the system when in fact it was a completely different part. We got stuck being hesitant to changing staff code and implementations that had previously worked. This was unfortunately where we lost a lot of time.
8. **Get a partner that complements your faults.** One of the things that we enjoy about working with each other is how different we are. We have very different skill sets and ways of approaching problems. Even though we were responsible for different modules in the system, we ended up working together a lot when helping each other debug, and obviously while integrating. We were able to provide each other with fresh perspectives and gain a general understanding of the whole system by the end of the project.

# Appendix

## References


[1] Mitchell Gu's Nexys 4_FFT Demo

[2] FFT Core documentation

[3] Note to frequency table

[4] The simplest way to make a sine generator


## Verilog Source Files

Link to public GitHub repository: https://github.mit.edu/elaineng/fpga_autotune

## Acknowledgements


We would like to thank Gim, Joe, Victor, and Sarah, for being extremely helpful in helping us complete our final project. We had fun living in the 111 lab all semester.