F.A.V.S.

FPGA Audio Visual Synthesizer

Andrew Sepulveda
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
andrews9@mit.edu

Jenzel Freeman
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
jenzel@mit.edu

I. Abstract

A. Problem

We would like to implement an additive synthesizer that is controlled via a MIDI In signal. The synthesizer should be able to produce sine, square, triangle, and sawtooth waves. Furthermore, we would like to do some analysis on the chords being played on the synthesizer and use this analysis to control animations on an LED Matrix.

B. Challenges

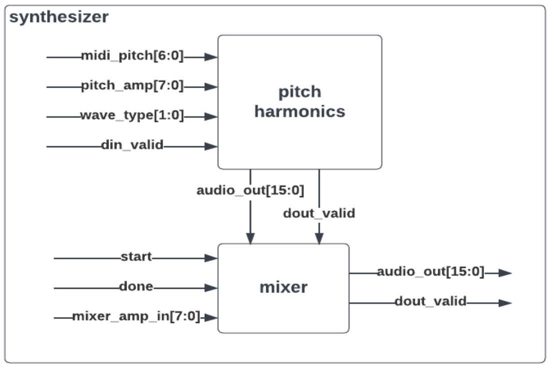
One challenge with developing a synthesizer is being able to reproduce a pure sine tone. While this may seem not so difficult, analog-to-digital conversion and resource limitation can introduce unwanted disturbances and harmonics in the produced audio. Also, our Nexys board does not come with MIDI In ports, so we will need to build the circuitry to allow for MIDI communication. Moreover, there are no drivers or libraries to work with when communicating to the LED Matrix thus any graphic that we would like to display onto the LED Matrix will require us to work from ”scratch” and implement all behavior ourselves.

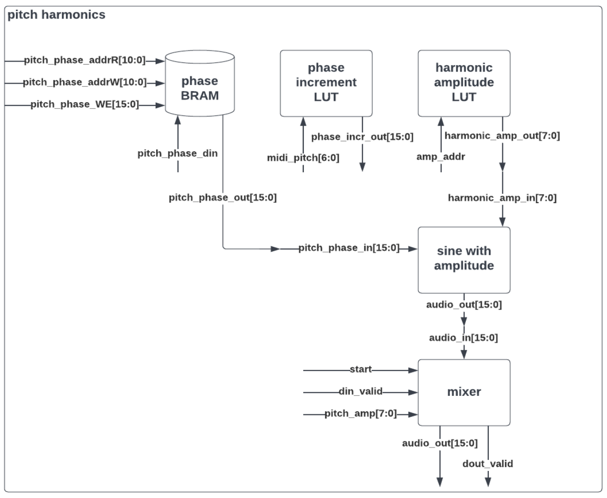
C. Requirements

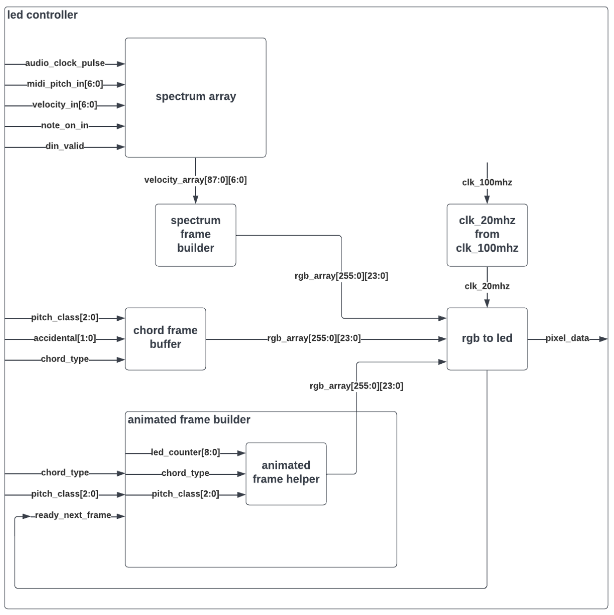
Our system should be able to produce 4 types of waves (sine, square, triangle, sawtooth). It should allow for 88 distinct pitches to be produced, which match the 88 pitches found on a full-size piano. The system should be able to correctly detect when any major or minor triad is currently being played. With the ability to detect these chords, the system should be able to display animations on an LED matrix depending on the chord detected. These animations should include displaying the name of any given major/minor triad being played as well as displaying an estimation of the frequency spectrum of the notes actively being produced by the synthesizer. Lastly, the system should receive MIDI In data from a digital keyboard.

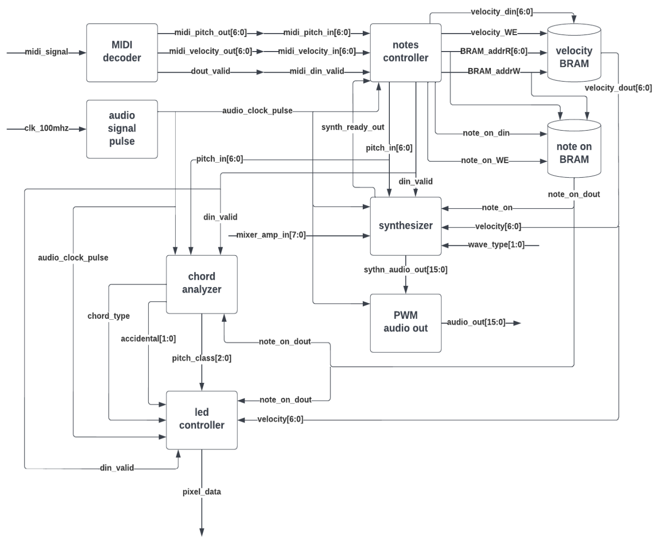
II. High-level Description

The status for all 88 pitches is controlled by a **Notes Controller** module. The **Synthesizer** module produces and outputs audio data every 44.1KHz. At the beginning of each 44.1KHz cycle, the **Synthesizer** is fed data regarding which notes are on and off and outputs the previously computed audio data to the speakers. The LED Controller has two main components associated with it: building the arrays to be displayed on the LED Matrix and the method in which the FPGA communicates with the LED Matrix. The **spectrum_frame_builder**, **animated_frame_builder**, and **chord_frame_builder** modules all function to build the arrays given data collected from the synthesizer. The **rgb_to_led** module then selects which of these arrays to use and sends the corresponding data to the LED Panel. It does this by considering one bit of the rgb code of one LED of the 256 LEDs on the panel at a time. All remaining block diagrams will be explained in great deal further along in this report.









III. Audio Clock Generation (Andrew)

To begin the process of audio synthesis, we need to determine at what rate we would like to produce and output audio. We decided to generate audio output at 44.1KHz which is the standard sampling rate for CDs. To do this, we have a module *audio_signal_pulse* running on the system clock of 100 MHz to produce a 44.1KHz clock pulse signal. It stores an internal 32-bit counter that increments by 1,894,081 every clock cycle. Why 1,894,081? To get a 44.1KHz clock pulse from the 100MHz clock, the audio clock signal should go high once every $\frac{10^8}{44100} = 2267.57$ cycles of the system clock. Since our counter is 32 bits wide, we set our counter increment to be $\frac{2^{32}}{10^8/44100} = 1,894,080.57 \approx 1,894,081$. Each time the counter overflows, the audio clock signal goes high for 1 system clock cycle. This works because every 2^{32} clock cycles, there will be $1,894,081$ overflows and thus the audio clock outputs a high signal every $\frac{2^{32}}{1,894,081} = 2267.57$ system clock cycles.

IV. Amplitude Modification (Andrew)

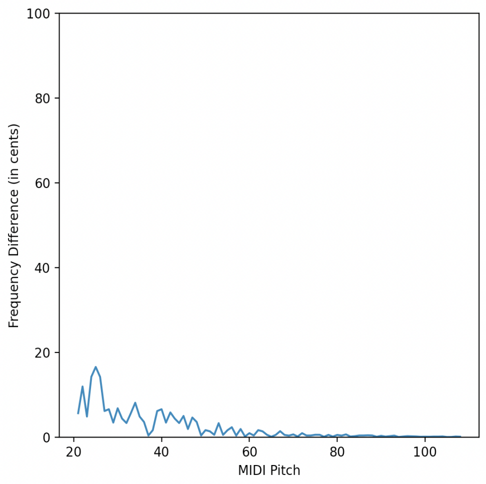
Since the audio data from the sine wave BRAM is scaled to its maximum values (-2^{15} to $2^{15} - 1$) we would like to scale its magnitude down. This will help prevent from clipping when adding together audio data which could overflow. To do this, we use an *Amplitude* module to multiply the audio data by an 8-bit signed value (from -128 to 127) and shift to the right by 7 bits to divide by 128. Thus, we are scaling the original audio data by multiplying by a fraction $\frac{X}{128}$ where $-128 \leq X \leq 127$. We allow X to be signed so that future calculations (i.e. triangle and sawtooth wave generation) will be possible.

V. Sine Wave Generation (Andrew)

To generate a sine wave, we use a BRAM to store 2^{14} samples of 16-bit audio data. The BRAM stores a quarter of a sine wave and we use symmetry to determine the sine value out. We use a 16-bit phase value, where 0 maps to 0 radians and $2^{16} - 1$ maps to just less than 2π radians. By using the symmetry of a sine wave, we can see that the 2 most significant bits of the phase correspond to which quarter of the sine wave the phase corresponds to. For example, a top two bits of 00 corresponds to a phase in the first quarter and a top two bits of 11 corresponds to a phase in the last quarter. Thus to get the corresponding sine value for the corresponding phase, we use the lower 14 bits of the phase as the address into our BRAM and adjust the resulting output according to the top two bits.

To produce a sine wave of any frequency, we step through the wavetable at different step sizes. In general, to produce a sine wave of frequency f, we use the step size $\frac{f}{F_s} \times 2^{16}$ where F_s is the sampling rate of 44.1KHz. If we want to produce a sine wave of 440 Hz, we would need a step size of $\frac{440}{44100} \times 2^{16} = 653.87 \approx 654$. We have precomputed the step size required for all 88 pitches on a piano and stored them in a look-up-table for quick access.

By using a BRAM with 2^{14} samples, we are able to produce highly accurate pure sine tones for all 88 pitches on a full piano. Figure 1 shows the difference between the desired pitch and the produced pitch, measured in cents where a cent is a unit of measurement for the ratio between two frequencies. As we can see, the lower pitches have the highest error (with MIDI pitch 25 having the highest error of 16.53 cents) and the higher pitches have the least error (with all pitches higher than midi pitch 67 having an error of less than 1 cent). It is important that the error be measured in cents since frequency doesn't increase linearly for consecutive pitches whereas cents does. For example, the difference in frequency between pitch 80 (830.61Hz), pitch 81 (880Hz), and pitch 82 (932.33Hz) is 49.39Hz and 52.33Hz, whereas the cents between each pair of consecutive pitches is always 100.



VI. Note Controller (Andrew)

Our *Note Controller* module is responsible for reading pitch and velocity data coming out of **MIDI Decoder** and storing this data in a **Velocity BRAM** and a **Note On BRAM**. Furthermore, **Note Controller** is responsible for reading out the data stored in these BRAMs for the **Synthesizer** and **Chord Analyzer** modules.

When **Note Controller** receives valid data from **MIDI Decoder**, it interprets a velocity of 0 as a note-off signal for the corresponding pitch. Thus, it will store a 0 in the **Note On BRAM** for the corresponding pitch to indicate the pitch is off. However, it does not update the velocity in the **Velocity BRAM** to be 0 for the corresponding pitch since we would like to remember this velocity even after the pitch is turned off. When **Note Controller** receives a non-zero velocity, it interprets this as a note-on signal and updates both BRAMs to indicate the corresponding pitch is on.

When **Note Controller** sees the 44.1KHz signal *audio_signal_pulse* go high, it immediately begins reading out the data from the BRAMs to the **Synthesizer** and **Chord Analyzer** modules. After reading out one sample, the module waits for a *synth_ready* signal from the **Synthesizer** and then reads out the next data stored in the BRAMs. Once it has read out the data corresponding to MIDI pitch 108 (the highest pitch on a piano), the **Notes Controller** module returns to an idle state waiting for the next high *audio_signal_pulse* to repeat the cycle.

VII. Synthesizer (Andrew)

The **Synthesizer** module controls the production of audio. It accomplishes this with the help of the **Pitch Harmonics** module and the **Synth Mixer** module.

Whenever **Synthesizer** sees the 44.1KHz *audio_signal_pulse* go high, it prepares the **Synth Mixer** for receiving data by setting the *mixer_start* wire high. Also, **Synthesizer** enters a "Reading" state and outputs a high *synth_ready* signal to let the **Note Controller** module know it is ready to receive velocity and note-on data. While in the Reading state, the synthesizer checks to see if it has received any valid data. If the data is valid but reads that the note is off, the synthesizer keeps its *synth_ready* signal high and waits for the next valid data. If the data is valid and the note is on, it sets *synth_ready* to 0 and passes the valid data to the module **Pitch Harmonics**. **Pitch Harmonics** will take in the *pitch*, *velocity*, and *wave_type* data from the synthesizer and produce 16-bit signed audio data that represents the corresponding pitch along with 16 of its harmonics. The *wave_type* given to **Pitch Harmonics** will scale the harmonics for a given pitch to produce sine waves, square waves, triangle waves, and sawtooth Once **Pitch Harmonics** finishes producing audio data, the Synthesizer will pass this data to the **Synth Mixer** module and the **Synthesizer** sets *synth_ready* high once again.

The **Synthesizer** continues this cycle until it receives valid data for MIDI pitch 108. After this, **Synthesizer** will set the wire *mixer_done* high to indicate to **Synth Mixer** that it will no longer be receiving audio data and to compute a final *audio_out* data that will be played on the speakers. To prevent any loss of data, all audio data passed into **Synth Mixer** is added together in a register of size $16+7=23$ bits. This prevents any clipping of audio since all audio passed in is 16-bits and there are a total of 88 notes, so even if all 88 notes have audio data corresponding to the highest possible value of $(2^{16}-1)$, $(2^{16}-1)\times 88 < 2^{23}-1$ so the data in the register should never overflow. However, we don't want to output 23 bits of audio, so we use an **Amplitude** module to decrease the data to 16-bits.

VIII. Pitch Harmonics (Andrew)

As mentioned earlier, the module **Pitch Harmonics** combines 16 harmonics for a given pitch and scales the amplitude of each harmonic by the *wave_type* signal. In real life, acoustic instruments generate harmonics when producing sound, so we would like for our synthesizer to mimic this. We will begin discussing how this module works by discussing the modules within **Pitch Harmonics**.

A. Pitch Phase BRAM

The **Pitch Phase BRAM** stores the 16-bit phase for each pitch and its 16 harmonics. Since there are 88 total pitches, the **Pitch Phase BRAM** has a depth of $88\times 16 = 1408$. The phases for all 16 harmonics corresponding to a certain pitch are stored sequentially. For example, the lowest MIDI pitch, 21, has its first harmonic phase stored at address 0, its second harmonic phase stored at address 1, and so on with its last harmonic phase stored at address 15.

B. Phase Increment Look-Up Table

To determine how much to increase a pitch's phase by, we made a look-up table that takes in a MIDI pitch from 21 to 108 and outputs a 16-bit phase increment value. Since harmonics are integer multiples of a fundamental frequency, then the phase increment for a corresponding harmonic is just an integer multiple of the phase increment for the fundamental pitch. For example, since MIDI pitch 60 has a phase increment of 389, the phase increment for pitch 60's second harmonic is $2 \times 389 = 778$. In general, to calculate the phase increment f_h for a given harmonic h, $f_h = h \times f_0$ where f_0 is the phase increment for the fundamental pitch.

C. Harmonic Amplitude Look-Up Table

The purpose of generating 16 harmonics for every pitch is so that we can produce waves other than just a sine wave. To produce a certain type of wave (sine, square, triangle, and sawtooth), we use the **Harmonic Amplitude Look-Up Table** to determine the amplitude for each harmonic. To do this, the *wave_type* signal is merged with the current harmonic number to produce an address to look into the table. The table will output the corresponding amplitude for the given harmonic in order to correctly produce the desired *wave_type*.

D. Sine With Amplitude

This module merges the **Sine Wave BRAM** and **Amplitude** modules into one, since it is a common operation in our system to get the sine value for a phase and then scale it. This module takes in a 16-bit phase and an 8-bit signed amplitude to produce a scaled 16-bit audio value.

E. Harmonic Mixer

Similar to the **Synth Mixer** module, this adds up the 16 produced harmonics and scales their sum to produce a 16-bit signed output.

F. Generating Pitch Harmonics

Now that we are familiar with the modules of **Pitch Harmonics**, we can discuss how it produces harmonic audio. To begin, the module must first receive a data-valid signal from the **Synthesizer** module. When it receives valid data, it sets a register *current_harmonic* to 0 and increments it by 1 every clock cycle until it reaches 15. The pitch data given to **Pitch Harmonics** will be passed to **Phase Increment LUT** to get the phase increment for the fundamental pitch. Furthermore, the input for **Pitch Phase BRAM** and **Harmonic Amplitude LUT** are derived from the *current_harmonic* register. The outputs of all of these modules are piped so that **Sine With Amplitude** receives the correct *pitch_phase_in* and *harmonic_amplitude_in* at the correct time. **Sine With Amplitude** passes its output to the **Harmonic Mixer** module which knows when it has collected all 16 harmonics. Once **Harmonic Mixer** has calculated the mixed harmonic signal, it sends a *dout_valid* signal high to let the **Synthesizer** module know it has completed and to move to the next pitch.

To update the **Pitch Phase BRAM**, the output of **Pitch Phase BRAM** is added to the corresponding phase increment and written back into **Pitch Phase BRAM**. What we are essentially doing is calculating the phase for the next audio clock cycle and storing it in the BRAM. Then, once the next audio clock cycle occurs, the phase is read out of the BRAM and used to calculate the audio data, and we once again compute the next phase and store it in the BRAM.

IX. PWM Audio (Andrew)

To convert our digital audio data into sound, we pass the data to a **PWM Audio** module. Typical PWM modules will take in audio data and use a counter that increments by 1 each clock cycle so that whenever the counter is less than the audio data the PWM module outputs a high signal and a low signal otherwise. This produces a square wave, which can introduce artifacts such as unwanted harmonics. What our **PWM Audio** module does is something similar, but instead of deciding whether to output a high or low signal based on the counter value, we reverse the counter value and output a high signal if the reversed counter value is less than or equal to the audio data. The idea behind this is that this method will push any distortions into high-frequency ranges that cannot be heard by humans. This helps eliminate any unwanted frequencies and produce a much more pure sound.

X. Chord Analyzer (Andrew)

To display effects on our LED Matrix, we analyze which notes are currently being played and determine if they build a chord. Our system can detect 24 types of chord triads, with 12 chords having a ”Major” quality and 12 chords having a ”Minor” quality. When the **Chord Analyzer** module sees *audio_signal_pulse* goes high, it enters a ”Reading” state. While in this state, if **Chord Analyzer** receives valid data indicating a pitch is off, it does nothing. If it receives valid data that indicates a pitch is on, we calculate the MIDI pitch number $\bmod 12$ and use the result as an index into a 12-bit register. We set the value at that index to 1, indicating that this pitch class is currently being played. There are 12 distinct pitch classes, and since chords can be described by just the pitch classes that make them, we only need to keep track of 12 registers. Once **Chord Analyzer** has received valid data for pitch 108, it passes the 12-bit register into a **Chord Look-Up Table** which will return a non-zero value if the 12-bit register describes a major or minor triad.

XI. LED Controller (Jenzel)

A. LED Clock Generation

The LED's utilized in this project (WS2812b) require data to be sent at speeds of 800Kbps, thus it is necessary to generate a clock of around 20mhz so that this condition may be satisfied. The **clk20mhz_from_clk100mhz** module completes this task by using an internal counter to wait every 5 cycles of the system's *clk_100mhz* to output a high signal. On the following clock cycle, the counter is reset and the module outputs a low signal to allow for the process to repeat, thus effectively creating a 20mhz clock from the system's *clk_100mhz*.

B. Chord Frame Builder

When displaying graphics onto the LED array, it is necessary to have a method of generating the 2 dimensional array in Verilog that can account for the 256 LEDs on the panel as well as the 24 bit *rgb_code* associated with each individual LED. The **chord_frame_builder** module takes in the 3 bit *pitch_class*, 2 bit *accidental*, and 1 bit *chord_type* derived from the **chord_analyzer** module to dictate what *rgb_array* to generate. It does this using purely combinational logic and, more specifically, case statements that make direct assignments to the elements (LEDs) in the *rgb_array* based on the three aforementioned inputs. For instance, if the *pitch_class* is 3'b001 that corresponds to a "C" note and thus the first 88 LEDs will be directly assigned (no loops) to represent a "C". Other values of *pitch_class* may result in a "D", "E", "F", "G", "A", or "B" to be assigned to the LEDs. The same process occurs for *accidental* such that it assigns the following 80 LEDs to either represent a sharp, flat or nothing at all if natural. Lastly, *chord_type* will determine whether a capital "M" or a lowercase "m", to represent a major or minor chord respectively, is assigned to the remaining 88 LEDs. The *rgb_array* created in this module is assigned to the 0th index of *rgb_arrays* in the top level.

C. Spectrum Frame Builder

The **spectrum_frame_builder** module employs the **spectrum_array** module which functions in collecting the velocities of all 88 notes every 44.1khz. The **spectrum_array** module accomplishes this by using a simple state machine where upon a pulse from *audio_clk_pulse* (occurs every 44.1khz) it enters a collecting state that checks whether the data for a note from the **notes_controller** is valid (*din_valid*) and whether the note is on (*note_on* from **note_on_BRAM**). If valid and on then the *velocity_in* of that note will be stored in *velocity_array*, otherwise nothing is stored. Once all 88 notes have been processed, the FSM will return to its resting state where it awaits a new pulse from *audio_clk_pulse*. Now that all 88 notes' velocities are stored in *velocity_array*, the **spectrum_frame_builder** module can use the array to build a frame to represent the current notes that are on. It does this using purely combination logic where the height of each column is determined by the velocities of the three notes from the array. The higher the velocity, the taller the column will be; however if all three velocities of the notes associated with a column are zero (indicating that the notes are off) then no LEDs will be on in that column. Therefore, this functions in displaying the frequency spectrum (an estimation of it) of the notes being played.

D. Animated Frame Builder

The **animated_frame_builder** module is the only frame builder module made to display a moving figure. It employs a helper module called **animated_frame_helper** that uses purely combinational logic to construct a colored shape centered around a specific LED. The helper uses a case statement that decides what color this figure will be dependent on what *pitch_class* is passed in and a case statement that decides what shape this figure will be dependent on what *chord_type* is passed in. The **animated_frame_builder** module runs on the system's *clk_100mz* and waits for *ready_next_frame*, passed in from the **rbg_to_led** module, to be high for it to alter the *led_counter* that it passed into the **animated_frame_helper** module. Thus, at a higher level, every time the LED panel is ready to display a new frame (*ready_next_frame* goes high), the **animated_frame_builder** module knows to change the *led_counter* passed into **animated_frame_helper** module which then builds an entirely new *rgb_array*, thus making it appears as though the figure is moving across the LED panel.

E. RGB to LED

When desiring to display a certain behavior to the LED panel, the first factor to consider is which *rgb_array* to display. This selection involves using *rgb_state* which is assigned to sw[15:14] to index into *rgb_arrays* effectively choosing which frame builder module's *rgb_array* to use. Moreover, given that the LED panel only has one pin (one bit) dedicated to data, in order to communicate with the panel, data must be sent one bit of the *rgb_code* for one specific LED at a time. However, to do this there must also be a way to differentiate between a high bit and a low bit in the *rgb_code*. The **rbg_to_led** module makes this distinction by first having an LED counter that indexes into the selected *rgb_array* to get the *rgb_code* for the current LED. Secondly, it grabs one bit of the *rgb_code* (starting with the most significant bit) and depending on whether the bit is high or low it behaves differently. If the current bit is high then it makes *pixel_data* high for 800ns and then low for 450ns. If the current bit is low then it makes *pixel_data* high for 400ns and then low for 850ns. The module repeats this process for every bit of every *rgb_code* for every LED in the array until it gets to the last LED. It then sends a reset signal to the LED panel by holding the *pixel_data* low for over 50µs. At this point, the module outputs a *ready_next_frame* signal that can indicate to the top level that it is prepared to display a new frame, thus preventing it from switching frames before a frame is fully displayed. If no new frame is passed in, it will continue to display the same graphics to the screen.

XII. Design Evaluation

A. Latency and Throughput

Many of the modules are designed to have high throughput and low latency. For example, our **Amplitude** module can take in new data every cycle and outputs a solution after 2 clock cycles. However, our module **Pitch Harmonics** has a latency of 26 clock cycles and must wait this long before feeding it new data. **Pitch Harmonics** has the worst latency and throughput in the entire system, and could potentially be improved to decrease latency.

B. Resource Utilization

Our system utilizes 4 BRAMs. The largest BRAM is 256Kbits (16384 X 16 bit) and is used to store a quarter sine wave table. This would be the easiest to reduce since we can decrease the number of samples stored and instead use linear or quadratic interpolation to generate sine wave data. The next largest BRAM is 22Kbits (1408 X 16 bit) and is used to store the phase of each pitch and its 16 harmonics. This would probably not be easy to remove, since our synthesizer is additive and would need to remember the phases for every harmonic. It could potentially be decreased significantly if we implement a frequency-modulated synthesizer, which could produce our desired harmonics without needing to store 16 16-bit phases for each pitch. The last 2 BRAMs use up 616 bits (88 X 7 bit) and 88 bits (88 X 1 bit). These could potentially just be combined into 1 BRAM which has a width of 8 and a depth of 88, where the most significant bit represents if a note is on or off and the remaining 7 bits represent the last velocity it was assigned. We utilize 3 look-up tables to store pre-computed phase increments, chord types, and harmonic amplitudes. I believe only the chord types LUT can be removed since the calculations to determine a chord type given the 12 pitch classes that are on are not very difficult.

C. Timing Requirements

Our system generates audio at a rate of 44.1KHz. This is roughly equivalent to 2267 cycles of the 100MHz system clock. This means that if our system had all 88 notes turned on, each note would get at most 25 cycles to compute its corresponding audio data. Our *Pitch Harmonics* module is the most worrisome since it is blocking code and takes 26 clock cycles to compute the corresponding harmonic pitch data. Thankfully, it is highly unlikely all 88 pitches would ever be on. We expect no more than 10 pitches to ever be on simultaneously, so our system will never really find itself taking more than 2267 clock cycles to generate audio.

D. Use Cases

Our synthesizer is able to produce 88 different pitches, corresponding to the 88 pitches on a full-size piano. Our synthesizer is also able to output sine waves, square waves, triangle waves, and sawtooth waves. Furthermore, our system can detect all Major and Minor triads and we use this detection to control designs on an LED matrix. The LED matrix can produce a spectrum band corresponding to which pitches are on. The LED Matrix can also display characters indicating which chord is being played (i.e. C # M). Lastly, the LED matrix can animate a ball moving across the screen whenever it detects a chord and changes the size of the ball depending on if the chord is major or minor. These are all deliverables we described in our project checklist that we met. Unfortunately, we were unable to get the MIDI in port to properly communicate with a piano keyboard.

E. Goal Achievement

Regarding the synthesizer, it is able to produce 88 distinct pitches that match the 88 pitches on a piano. It can produce sine, square, triangle, and sawtooth waves. The overall volume of the audio produced can be scaled by 128 values. For any 1 pitch, the synthesizer is able to produce 16 of its harmonics. Unfortunately, the MIDI Decoder module was unable to successfully communicate MIDI data to the FPGA. In regards to the LED portion of the project, all ideal goals were met. The frequency spectrum as well as the chord description of a set of notes being played successfully were displayed on the LED Matrix. A portion of the stretch goals were also met as the project was able to display a moving figure dependent on what combination of notes are played. With very slight modifications, the project could be used in

XIII. Retrospective

Regarding the synthesizer, it could have potentially been easier to produce harmonics using frequency-modulated synthesis instead of adding 16 harmonic sine waves. One issue that could have saved a whole lot of time would have been to have all modules exist in the top-level hierarchy. Initially, the **Synthesizer** module had numerous small and large modules embedded within it. Some of those modules also had other modules embedded within them. As a result, the audio being produced was several layers deep and would need to be passed back up through the modules to be exposed to the top layer. These embedded modules also resulted in many points where other modules were waiting idly for deeper modules to finish so that they can begin to do work. In regards to the LED portion of the project, it was very eye-opening to know that if one wanted to build an entire frame within a single clock cycle using combinational logic, it is highly encouraged to create a new module to implement this combinational behavior so as to not mix it with sequential logic and open oneself up to pesky bugs. It was also immensely helpful to dedicate the time to understanding the LEDs reference sheet in order to familiarize oneself with how to communicate with the LED Matrix.

XIV. Credit and Code

Andrew Sepulveda worked on all modules related to the synthesizer. This includes the Note Controller, the 44.1KHz clock signal, the Velocities and Note-On BRAMs, the Sine WaveTable, the Amplitude Modulator, all submodules of Synthesizer and Pitch Harmonics, the Chord Analyzer, and the PWM Audio output.

Jenzel Freeman contributed to the project by developing the driver that facilitates communication between the FPGA and the LED Matrix. Additionally, he worked on building the arrays to be displayed onto the LED panel and made sure they were dependent on the data that Andrew's contribution provided. He also created all block diagrams used in this report.
https://github.mit.edu/andrews9/FAVS/