FPianoGA Final Report

Anahita Srinivasan
Massachusetts Institute of Technology
anahi183@mit.edu

Elise Wingard
Massachusetts Institute of Technology
elisew@mit.edu

Abstract—**We present the design for FPianoGA, a digital piano implemented on an FPGA with accompanying graphics that allows users to play a series of notes, hear the audio of those notes, and visualize them on a musical staff on a monitor. We use the upper eight switches present on the FPGA board as the keys of the piano. When a switch, representing a particular note on the piano, is flipped, the appropriately pitched note is re-constructed through the superposition of sine wave outputs, each generated at the playing note's harmonic frequencies. The corresponding audio output is then transmitted to the on-board audio jack for consumption by the user. In addition, the corresponding note is drawn on a musical staff using the HDMI protocol on a monitor in one of two modes: free-play and guided-play. We describe the implementation of this design in SystemVerilog, evaluate its performance with quantitative measurements (latency, throughput, etc.) and qualitative measurements (clarity of audio and visual accessibility of graphics), and discuss implementation insights.**

1. Introduction

We propose FPianoGA, a portable, digital piano with both audio and visual components that plays piano key notes through the FPGA's built-in audio jack and displays the notes on a musical staff transmitted through the FPGA's HDMI port to a monitor. We design FPianoGA to prioritize clarity and accuracy of the piano key audio and accessibility of the graphics for those who may be unable to read music or have limited exposure to it.

Section 2 discusses Elise's design of the audio component of FPianoGA, which handles the creation and processing of audio depending on key input. Section 3 discusses Anahita's design of the visual component of FPianoGA, which handles drawing the played notes and musical staff and implements both a free-play mode and a guided-play mode for user interaction. We evaluate our design in section 4.

We implement FPianoGA on the RealDigital Urbana FPGA board. FPianoGA uses the upper eight switches of the sixteen present on the board, `sw[15]` through `sw[8]`, as the keys of the piano. The upper 8 switches mimic the natural ordering of keys on a piano, with `sw[15]` representing the `C4` note and `sw[8]` the `C5` note. `sw[0]` is used to toggle the piano between free-play mode and guided-play mode, and `sw[1]` is used to select the song users wish to play in guided-play mode, the details of which we discuss in section 3. We use the built-in audio jack to transmit FPianoGA's note sounds and the built-in HDMI port to transmit the music staff and associated visual components to an external monitor.

[redacted]

2. Audio Component (Elise)

The main goal for the audio portion of the FPGA is to mimic the sound of a piano note being played with the correct pitch and timbre (i.e, sound quality that specifically mimics the sound of a piano note). To this end, the audio portion of the FPianoGA uses the complex coefficients of the frequency-domain representation of a C4 piano note to reconstruct a time domain-valued sound wave that is representative of any combination of the natural notes from C4 to C5 being played on the digital piano. This sound wave is then retransmitted through the FPGA board's audio port.

The audio portion of the project satisfies our commitment to generating an appropriately pitched note for any combination of notes played through its implementation of a sine-wave generator and a summing module that sums the outputs of multiple sine generators. It tunes these sounds to be more piano-like through the incorporation of multiple harmonic sine waves for each note that are appropriately scaled and offset. Although the sounds are admittedly not as piano-like as initially envisioned, the design does manage to incorporate undertones in the played sounds.

The primary problem to solve in the audio component was how to most accurately recreate the sound of a piano note being played using a digital system. The technical challenges in doing so arose from attempting to recreate a continuous-valued, analog signal despite the constraint of working with binary numbers and calculation methods. In particular, creating a sine wave generator to replicate the continuous-valued sine functions and super-imposing these sine waves proved to be a technical challenge. Because different harmonics were scaled by values that could be up to two orders in magnitude different, selecting the appropriate bits of the scaled sum to look at was crucial, as was making sure that the summed output had a large enough bit width so as to have no clipping and maintain the clarity of the audio.

Moreover, because there exists an inherent trade-off between the range of analog values that can accurately be represented through a given number of bits and state and the space available on the FPGA, we found ourselves needing to choose between increasing the number of bits to represent phase or the number of sinusoidal outputs to increase accuracy and choosing to keep these values lower to reduce the size of any subsequent LUTs. Similarly, while having more harmonics could be used to re-create the analog sound of a piano note more accurately, storing the necessary information for the phase, magnitude, and additional phase increments for every note requires memory. This computational efficiency and space versus reconstructed signal accuracy trade-off required us to maintain a careful balance between optimizing the use of available resources on the FPGA board so there would be enough for the audio and visual portion, all the while still routing the available resources intelligently to the modules to be able to re-create the key notes' sounds the most faithfully. More information on the implications of the technical challenges on the audio portion's design choices are discussed along with the description of the audio system description.

Figure 1 provides an overview of the complete audio processing pipeline.

2.1. Sine Phase and Magnitude Extraction

Pure sine waves alone could be used to generate sounds of the correct pitch for given notes being played. However, to mimic the sound characteristics of a piano note, we use additional information derived from the complex coefficients of a C4 piano note's harmonics in the frequency domain. For this project, a Python script was used to compute the FFT of said C4 piano note and extract these coefficients. For the project we use the first 5 harmonics of each note to reconstruct its sound in the time domain, meaning we generated the coefficients for the C4 note's first 5 harmonics.

Each harmonic coefficient has a 16-bit wide real and imaginary portion which, for convenience when being passed as inputs into in downhill modules, were concatenated into a single 32 bit value. The five Python-generated harmonic coefficients are stored in our `top_level` module. Using these coefficients, for each harmonic we then derive a phase and magnitude that is then applied to the corresponding harmonic of whatever note(s) is being reconstructed further down in the pipeline.

First, we compute the magnitude of each of the five complex coefficients. Because the complex coefficients remain constant in time, they are only computed upon system reset. This design choice is intended to minimize unnecessary power use on the FPGA by avoiding computing values that have yet to change. We also wished to use a single instantiation of the module `magnitude_compute` which calculates the magnitude of a single coefficient at a time in order to save in distributed RAM space.

To both of these ends, a module named `coeff_info_gen_mags` was implemented. The module takes in a vector of the 5 complex coefficients and outputs both a 5-entry wide vector of the corresponding magnitudes as well as a `valid_out` signal when these have been computed. `coeff_info_gen_mags` instantiates the `magnitude_compute` module in which `magnitude_compute` does a split square sum of the imaginary and real coefficients using the method provided by the code in Lecture 11, slide 72, this semester.

Once this split square sum has been calculated, the output is passed into a piece of IP for a square root, a `sqrt_int` module built by Will Green under the MIT license, to compute the square root of this split square sum.[1]

Once the square root is generated, the both the magnitude and a single-cycle high valid_out signal are sent to the `coeff_info_gen_mags` module, which upon detecting the single-cycle high for completion either sends in the next coefficient whose magnitude needs to be computed into `magnitude_compute`, or outputs a valid_out signal if all magnitudes have been computed.

The phase of each complex coefficient was computed in radians off-board and then converted to the appropriate phase encoding for the downstream `sine_wave_sum` module. This phase is then stored for use by `sine_wave_sum` in the `top_level`.

For more information on the same implementation of audio with the audio coefficients coming from an on-board FFT computation rather than being hard-coded, refer to the "Implementation Insights" section as well as the FFT audio folder in the repository.

2.2. Time Domain Sound Signal Reconstruction

FPianoGA reconstructs sound waves in the time domain through the superposition of sine waves. For each note played on the digital keyboard, 5 sine waves - one for each harmonic - are generated. We chose to implement 5 harmonics for sound reconstruction because of the limited bit depth of the output audio. Because the output audio is only 8 bits in depth, only two orders of magnitude of difference in scaling could fully be supported before resolution was since, otherwise, larger-scaled harmonic sine waves would determine the value of the output, with smaller-scaled higher harmonics not contributing to the upper bits of the final sum as much.

We support up to all 8 notes being played at once, with the appropriate-pitched sound being played for any combination of notes. Hence, for x number of notes being played, there will be $5x$ sine waves 'generated,' each with a distinct frequency corresponding to a given note's harmonic's frequency.

The `sine_generator` and `sine_wave_sum` modules generate a sine wave for each harmonic and are scaled, offset, and summed in these modules, respectively. The

1. Project F Library: Square Root (Integer). Will Green, 2021. MIT. https://github.com/projf/projf-explore/blob/main/lib/maths/sqrt_int.sv

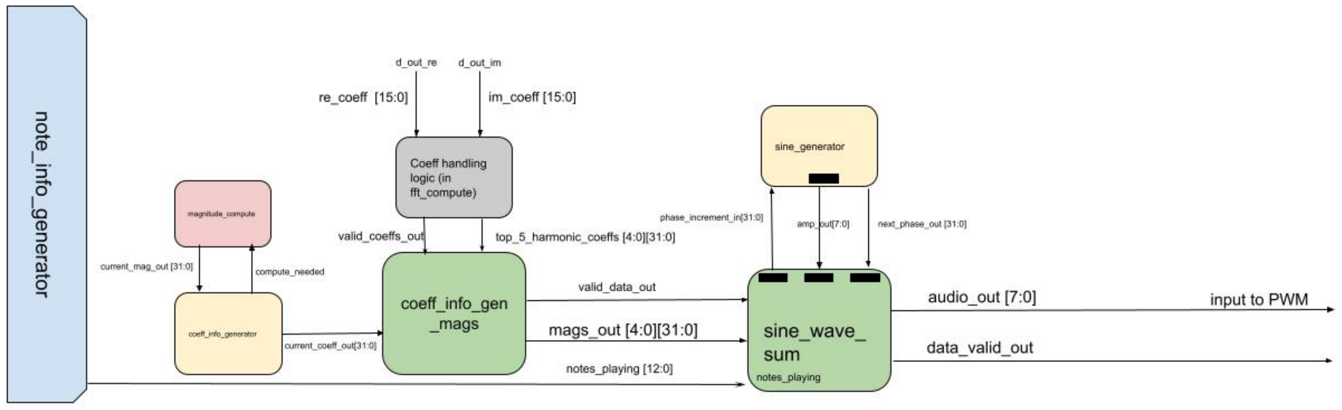

Figure 1. Audio component block diagram. We use five harmonics to reconstruct the time-domain audio signal.

phase offset and scaling factor are determined by the complex coefficients and calculated as discussed in Section 2.1.

2.2.1. Sine Wave Generator. The `sine_generator` module outputs fluctuating amplitudes in a periodic manner to mimic the functions of a continuous sine function. It outputs an amplitude based on the phase (angle in a unit circle), a stateful attribute, that it is on. This phase is 32-bits wide. The top 6 bits of the phase are used to select the output amplitude out of 64 options, and the bottom 24 bits are used to keep track of fractional phase increments to provide more resolution in the frequencies that can be represented by the sine generator.

The `sine_generator` is being run at an 8 KHz rate, meaning a phase increment is triggered every 12500 clock cycles at which point the phase will be incremented by a dictated amount `phase_increment`. We choose this rate because the audio output is generated by the `PWM` module, which samples the sine wave sums at an 8 KHz rate. Using the same rate eliminates unnecessary computations that would need to be done if the sine generator was being run at a faster rate and prevents information loss.

The same instantiation of the `sine_generator` module is used to determine the output of the different notes' sine waves between audio output samplings. Since all sine waves are incremented once per 12500 clock cycles, we account for their different frequencies by choosing a unique phase increment for each of them that reflects how much the phase would have to progress between samples based on their frequency.

This phase increment is passed into the `sine_generator` module as an input rather than being stored as a parameter of the function. The previous phase is then incremented by the `phase_increment`, and the output amplitude of the sine wave corresponding to the new phase is output to the encapsulating module, `sine_wave_sum`.

2.2.2. Sine Wave Sum. The `sine_wave_sum` module is used as a "manager module" that takes in as input the keys being played, phase offset, and scaling factors for the sine waves and outputs audio signal to be sampled by the PWM. It is implemented using a finite state machine that operates at an 8 KHz rate, or cycles through all of its states once every 12500 clock cycles.

This FSM leverages the fact that there are over 10000 clock cycles between audio sample outputs and only performs one operation (whether it be a single addition or multiplication) per clock cycle, minimizing the risk of timing violations from many mathematical operations in a single cycle through lots of pipelining.

The states of the FSM are **IDLE**, **UPDATING**, **SCALING**, **ADDING**, and **OUTPUTTING**. In each of the states, the FSM steps through a lookup table that is 8 deep (for the number of notes) and 5 wide (for the number of harmonics), one value per clock cycle at a time, and performs the appropriate operation. In the **UPDATING** state, the phase of the sine waves for the notes being played are updated in their respective LUT for access in the next iteration of the FSM, and the output of each sine wave is updated as well. Once all $8 \cdot 5$ entries have been stepped through, the FSM moves onto the **SCALING** state, where every output from the sine waves is scaled by the appropriate coefficient (depending on the harmonic the sine wave output corresponded to) one at a time, one clock cycle at a time. The FSM then moves onto the **ADDING** state where these scaled sine outputs are added one clock cycle at a time to `out_sum_total`. Finally, the FSM moves to the **OUTPUTTING** state where it outputs the appropriate amplitude and holds the output steady until it is sampled by the PWM module and output to the FPGA's audio port.

This implementation of the `sine_wave_sum` module allows for multiple sine waves to be run simultaneously on the same instantiation of the `sine_generator`. Note that currently these LUTs are fairly small and implemented using an array, but if we had used more harmonics, it would be beneficial to use a BRAM instead.

3. Visual Component (Anahita)

The visual component of FPianoGA is transmitted via HDMI to a monitor. It consists of a five-line staff, the treble clef sign to contextualize note placement, a metronome indicator, a cursor, and several notes that appear and are oriented on the staff depending on which mode the user is playing in. There are two possible modes for the user to interact with FPianoGA: free-play mode, in which the user can play whatever notes they choose and see them visualized on the staff; and guided-play mode, in which the user can select a pre-loaded song from a library and play along to the notes on the staff.

The visual component is driven by a 74.25 MHz clock, which is requisite for the HDMI protocol. Each pixel on the display is drawn in a raster pattern, with a new pixel being drawn every clock cycle. Figure 2 contains a complete diagram of the visual module.

The primary problem to solve with the visual component was how to produce one pixel every clock cycle at the 74.25 MHz rate, given the complexity of the overlap and color calculations for each pixel on screen. To this end, we needed to carefully partition our system to allow for multiple clock cycles of computation for each `hcount` and `vcount`, while also pipelining each `hcount` and `vcount` through the system to match which pixel was entering the TMDS encoder.

3.1. HDMI Signaling Infrastructure

We use the previously-created video signal generator, TMDS encoder, and TMDS serializer modules for the visual component of FPianoGA. The video signal generator outputs several signals for 1280 by 720 pixel video; of these signals, the visual component chiefly uses the `hcount`, `vcount`, and `new_frame` signals to decide which pixel should be drawn. The `hor_sync`, `vert_sync`, and `active_draw` signals, pipelined appropriately, are in turn passed to the TMDS encoder, which sends its outputs to the TMDS serializer.

3.2. Background Components

The underlying background is the same for both modes. We implement a five-line `staff` module, drawn simply with a combinational module that returns black if the raster index is the same as one of the vertical pixel indices that comprises the staff, and white otherwise.

We also implement a `treble_clef` module that stores a black-and-white image of a treble clef sign in the FPGA's BRAM. The treble clef is 200 pixels wide by 360 pixels tall. We create one BRAM module that, for each pixel, stores a 0 for white and a 1 for black. We then use that bit to combinationally determine if the outputted color should be the 24-bit hex code for white or black.

The metronome indicator is a 50 pixel by 50 pixel square in the upper left of the monitor that switches color from black to white once every 0.5 seconds. The `metronome` module monitors the `new_frame` output from the video signal generator module; the system operates at 60 frames per second, so the metronome indicator waits for 30 `new_frame` triggers to occur before switching color.

We also implement a cursor blinking at the same rate as the metronome that travels horizontally across the monitor as notes are played. We discuss later in this section the implementation of a `counter` variable in both modes that keeps track of which note is being played. Based on the value of `counter`, the cursor's `x_in` value is changed to move it to the appropriate horizontal location on the staff to signal the next note to be played.

Finally, we include user instructions displayed in the top right corner of the screen in the form of a `instructions` module. The instruction text is written in black-and-white in a 300 pixel by 110 pixel image. The `instructions` module, like the `treble_clef` module, uses a BRAM to store a 0 for white and a 1 for black for each image pixel, then determines using that bit which 24-bit hex color should be outputted.

3.3. Free-Play Mode

The free-play mode of FPianoGA allows users to play a variable-length sequence of notes, with each note being played for a different duration, and see the notes visualized on the staff. For optimal user accessibility visually, we accommodate eight notes on the staff before it is cleared automatically for the next batch of notes to be played.

The dynamic components of free-play mode are the `note_sprite` modules, which take in the `hcount` and `vcount` that the raster pattern is currently at and return the note's color if the `hcount` and `vcount` are within the note's current bounds. We instantiate eight of these modules, one for each note that can fit on the staff. Upon system reset, the `color_in` of each module is set to white to make the `note_sprite` appear "invisible" in the background, and the vertical location of each `note_sprite`, denoted by `y_in`, is set to the top of the screen, away from the staff lines.

The `free_play` module handles the location and coloring changes of the `note_sprite` modules as keys are played. It uses the `key_press_counter` module, which returns a `counter` that keeps track of which of the eight notes on the staff is next to be moved, a `key_played` indicator, and a `note_duration` indicator.

Each clock cycle, the `free_play` module checks the value of `counter`. It moves the appropriate `note_sprite` based on the value of `counter` to the appropriate location on the staff by changing its `y_in` value based on the value of `key_played`. Because FPianoGA is designed to be accessible for music beginners who may not yet know how to read a staff or understand what different note durations look like on formal sheet music, we choose to convey that information through color. To make the note "appear," the `note_sprite`'s `color_in` value is changed to that note's unique color. Based on the `note_duration` output

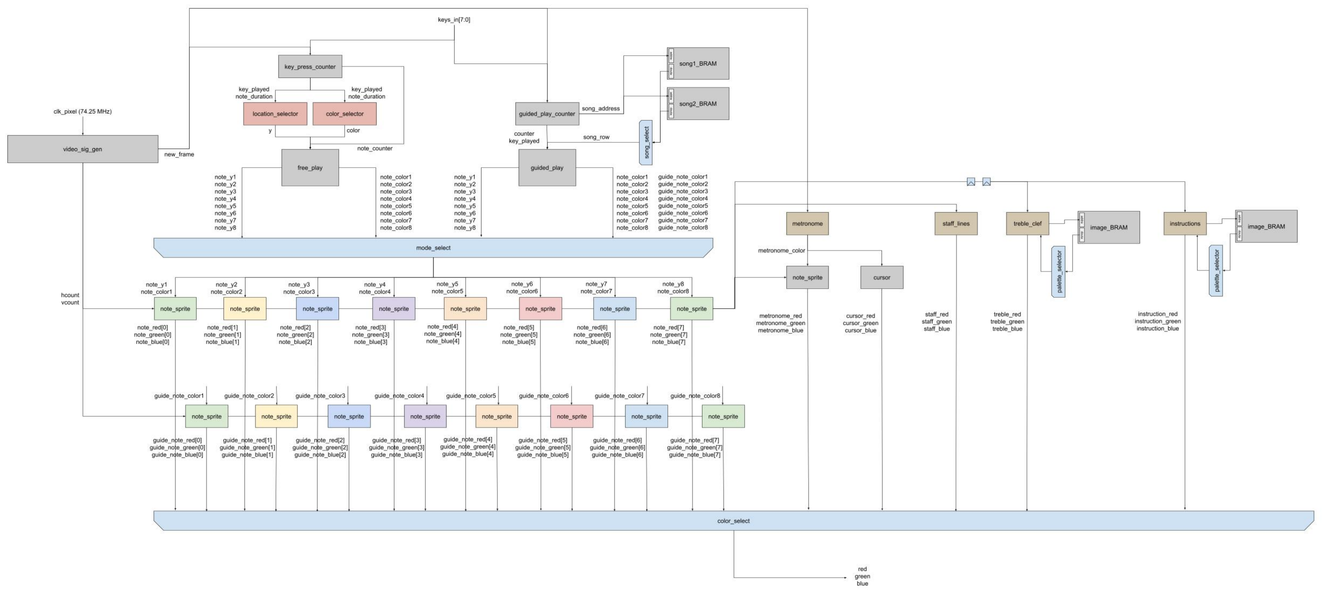

Figure 2. Visual component block diagram. The red, green, and blue signals leaving the visual component are passed through a TMDS encoder and serializer, as well as an OBUFDS, to create the final HDMI signal.

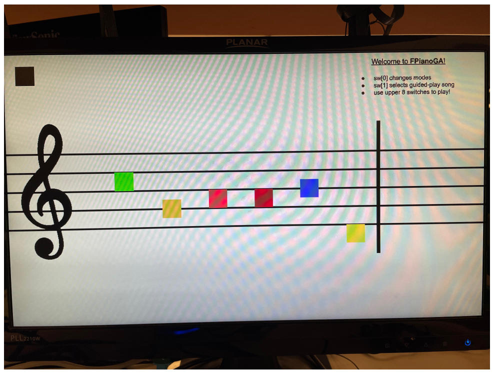

Figure 3. Example of notes being played in free-play mode. Note that the third and fourth notes, both A4, are different colors because the fourth note was played for a longer duration.

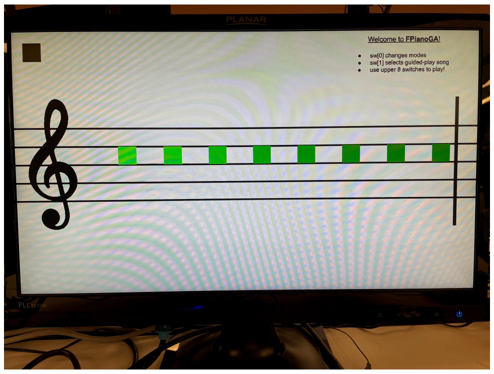

Figure 4. Color gradation of C5 note depending on duration played.

from the `key_press_counter` module, the `color_in` value slowly darkens as a visual indicator of the duration for which the note was played, based off the `metronome`. Figure 4 displays how the C5 note changes color on the staff depending on the value of `note_duration`. We use two combinational helper modules, a `color_selector` and `location_selector`, to determine where each `note_sprite` should be placed and what color it should take.

Once the staff fills up with eight notes, `counter` resets value, clearing the staff up and allowing the `note_sprite` modules to take new values as the user continues playing.

3.4. Guided-Play Mode

The guided-play mode of FPianoGA allows users to follow along with a song whose notes are displayed on the monitor's staff. Again, as in free-play mode, we accommodate eight notes of the song on the staff at a time before the staff clears and displays the next eight notes.

Each song available in the library is stored in a BRAM of width 32 and depth 4. Each note is represented with four bits, so each BRAM row, which we denote as `song_row`, contains eight notes' (or one full staff line's) information.

Like the `key_press_counter` that the `free_play` module uses, `guided_play` implements a `guided_play_counter`. It maintains its own `counter` variable that loops from 0 to 8 and keeps track of which key is currently being played by the user. When the `counter` is at 0, the read address to all of the song BRAMs, `song_address`, increments by 1, fetching a new set

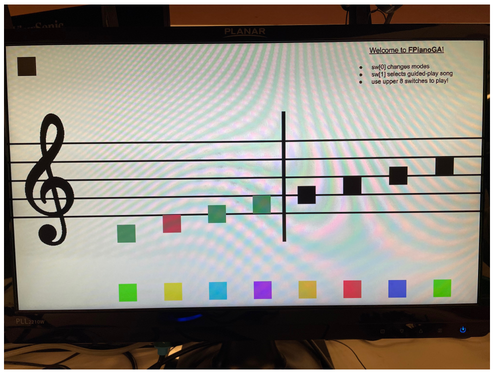

Figure 5. Example of a scale being played in guided-play mode. The second note is red because the user played the incorrect note. All other notes were played correctly and thus are green.

of eight notes from each BRAM. Based on the value of `sw[1]`, the switch used to select which song the user wants to follow along with, we select one of the BRAM rows, assign it to `song_row`, and index the eight notes' values out of `song_row`. Using those values, we move the eight `note_sprites`' locations to the appropriate place on the staff by changing their y_in values. We implement two songs to select from in guided-play mode: the first, which appears when `sw[1]` is set to 0, is the C-major scale as well as a few scale patters; and the second, which appears when `sw[1]` is set to 1, is a (rough) one-octave approximation of "Never Gonna Give You Up" by Rick Astley.

When the user plays a note, the `guided_play` module checks the switch that was played against the correct note indexed out of `song_row`. Based on the value of `counter`, which indicates which note in the row is currently next to be played, the appropriate bits of `song_row` are selected for the comparison. If the correct note was played, the module sets the `color_in` value of the `note_sprite` to green; otherwise, it sets the value to red.

Guided-play mode also includes a set of eight additional `note_sprite` modules situated at the bottom of the monitor that serve as "guide notes." These notes are colored using `song_row` as they would be in free-play mode to guide the user visually as to what note the `note_sprite` drawn on the staff represents. They remain static in position throughout. When a new `song_row` is read, the guide notes' colors change as well.

3.5. Integration

The `free_play` module outputs eight 10-bit `note_y` values and eight 24-bit `note_color` values every clock cycle, one for each `note_sprite` module. Similarly, the `guided_play` module outputs eight 10-bit `note_y` values and eight 24-bit `note_color` values, as well as eight 24-bit `note_guide_color` values for the guide notes along the bottom. Based on the value of `sw[0]`, either the `free_play` output or `guided_play` output for `note_y` and `note_color` is fed into the `y_in` and `color_in` ports for the eight `note_sprite` modules. If `sw[0]` is in free-play mode, the eight guide `note_sprite` modules are set to white and are "invisible"; otherwise, they assume guided-play's `note_guide_color` values.

Once the pixel values for a given `hcount` and `vcount` are calculated for each of the background components, the eight `note_sprite` modules, and the eight guide `note_sprite` modules, the system takes another clock cycle to determine which of these pixel values should be drawn, i.e. which component should be "overlaid" on top of the screen. This combinational logic determines the final 8-bit `red`, `blue`, and `green` values that get passed into the TMDS encoder.

4. Evaluation

4.1. Latency, Throughput, and Timing

The audio and visual pipelines run in parallel, with the audio pipeline clocked at 100 MHz and the visual pipeline clocked at 74.25 MHz. Thus a clock period for the audio calculations is 10.00 nanoseconds and a clock period for the visual calculations is 13.468 nanoseconds. There is no clock domain crossing in our system. We use different clock periods because the visual component requires a 74.25 MHz clock to comply with the HDMI protocol, while a 100 MHz clock makes audio sampling calculations easier.

Our system has a worst negative slack of 0.383 nanoseconds; the path with this slack occurs in the audio pipeline in the `sine_wave_sum` module while creating the weighted sine lookup table. This process takes 9.454 nanoseconds (in the 10.00 nanosecond clock period), of which logic takes 4.963 nanoseconds (52.496% of the time) and routing takes 4.491 nanoseconds (47.504% of the time). We meet timing requirements in both pipelines.

Within the audio component of FPianoGA, all calculations are performed in the 12500 clock cycles between audio sample outputs, which is more than enough time for the pipeline to produce the next `audio_out` sample.

When a note is played, the `sine_wave_sum` module spends 40 cycles in the **UPDATING** state, one cycle to update each LUT based on the output of `sine_wave_generator`, which takes only one cycle to return a sine value given a phase offset and increment. Similarly, it then spends 40 cycles in the **SCALING** state to scale each entry appropriately, and 40 cycles in the **ADDING** state to add each scaled entry to the total sum. Finally, the module spends one cycle in the **OUTPUTTING** state to return the final sum. So the latency of this module in outputting a note's final audio is 121 clock cycles, or 1.21 microseconds.

However, the latency of the pipeline as a whole is 12500 clock cycles, or 12.5 microseconds, because the audio sampling rate of 8 KHz serves as a bottleneck. Even if the sine wave calculations could be completed earlier, we cannot start a new round of calculations until the audio signal is output through the FPGA's audio port. For that reason, we can use the same numbers to calculate the throughput of the audio pipeline as 8 KHz, or 8000 audio samples per second. For the audio portion of the design, this throughput is ideal. Too high or low of a throughput would generate audio samples at a frequency too high or low to be perceived by the human ear. This throughput also allows for more pipelining in computation-heavy modules such as sine-wave-sum, effectively minimizing the amount of logic per clock cycle and helping the audio portion meet timing.

The visual component of FPianoGA is separate from the audio pipeline and is thus not constrained by the audio sample rate. Each `hcount` and `vcount` value is outputted after one clock cycle. The `free_play` and `guided_play` modules update their `note_y` and `note_color` values in the next clock cycle, and the `note_sprite` modules update their locations and colors based on the current mode and the `hcount` and `vcount` from the previous clock cycle as well. However, given that the image components take 2 clock cycles to output pixel values because they must read from the BRAM, the `note_sprite` modules hold for an extra clock cycle so all pixel values are synced. These pixel values, along with the pixel values of the background components, are inputs to a piece of sequential logic that takes another clock cycle to determine the final `red`, `green`, and `blue` pixel values to be sent to the TMDS encoder. So the latency of the visual pipeline, from video signal generator to entering the TMDS encoder, is five clock cycles, or 67.34 nanoseconds.

Every clock cycle, a new `hcount` and `vcount` are generated by the video signal generator to be given a pixel value, so the throughput of the visual pipeline is 1 pixel per 13.468 nanoseconds.

4.2. Memory Requirements

The primary use of memory for the audio component of the project is to store the complex coefficients, the sine-wave lookup table (LUT), and the phase-increment LUT. Each of these is stored not in BRAM but as an array on the FPGA. There are five complex coefficients, each 32-bits wide, necessitating 160 bits of storage. The sine-wave LUT has 64 entries, each 8 bits wide, necessitating 512 bits of storage. Finally, the phase-increment LUT contains 40 entries (5 entries for each of the 8 notes, one entry for the fundamental frequency and four for the harmonics) each 32 bits wide, necessitating 1280 bits of storage.

Thus the total memory of the audio component is $160 + 512 + 1280$ bits, or 1.952 kilobits.

The visual component of the project uses two BRAM modules for the images on-screen, and two BRAM modules to store the correct notes for the guided-play songs, one module for each song. The treble-clef image is 200 pixels by 360 pixels and needs only 1 bit of color resolution, so its BRAM is 72000 entries wide with a bit-depth of 1. Similarly, the 300-pixel-by-110-pixel instructions image uses a BRAM that is 33000 entries wide with a bit-depth of 1. Each song available in guided-play mode uses 4 rows of BRAM storage, with each row storing 32 bits of information.

Thus the total BRAM usage of the visual component is $72000 + 33000 + 128 + 128$ bits, or 105.256 kilobits.

4.3. Use Cases

Our design supports numerous use cases. Users can play FPianoGA in two modes, free-play and guided play, and can even hear multiple notes being played at once. We have clear visual indicators of tempo via the blinking metronome and cursor as well as the darkening of notes based on the duration they have been played for. These use cases support our design goal of accessibility with clear instructions, an easy-to-approach interface, and a clear indication of how to connect the sheet music with the audio. Aurally, we met our minimal goal of having working audio for eight keys that was clear and in-tune. Visually, we achieved not just our minimal goals but also the ideal goal of supporting two 32-note songs in guided-play mode.

Aurally, the first extension to consider would be adding more harmonics to more closely mimic a piano sound. In a similar vein, we could add more instrument noises simply by changing the coefficients, phases, and magnitudes of each note's harmonics, perhaps by switching to different lookup tables using the remaining switches on the FPGA.

Visually, the easiest extension to support would be the addition of more guided-play songs of larger length. This could be accomplished simply by allocating more BRAM and increasing the bit width of the `song_row` variable to accommodate more BRAM rows.

5. Implementation Insights

Aurally, the main task that could have been improved on was beginning to code the project earlier. A lot of time was initially spent planning out each stage of the pipeline on paper, specifying how each module would connect, and trying to compare different possible designs which while helpful at times often needed to be revised anyways during the actual implementation of the code. Had implementation of some of the larger modules begun a week or two earlier, we would have noticed where the planning from earlier fell apart and seen where the limitations of theory were a lot sooner: this would have allowed us time to resolve unexpected problems. For example, the audio quality was initially poor due to clipping issues when summing the sine waves, and the design of the LUT for the sine wave which ultimately needed to be re-designed. Had this been sorted out earlier, it perhaps would have allowed more time to deal with deeper, more complex issues, such as those with the FFT portion of the pipeline that eventually had to get cut out. While we were able to build that portion and get it working, we did not have time to integrate it into our project with full confidence and have left it as stand-alone code in the `FFT_AUDIO` folder. Additionally, `Pipeline_With_Audio.jpg`, which provides an overview of the pipeline with the FFT included, is uploaded in the folder as well. Dealing with more essential modules earlier on, such as the `sine_wave_sum` module, rather than modules that could more easily be substituted, such as the now-removed arc-tangent module, would have allowed other, more crucial additions to the project to be explored, such as implementing note decay features or other instrumental sounds.

Visually, we had almost the opposite problem: we began implementation almost immediately, causing lots of iterative redesigning as new issues came up. For example, when we decided to add a metronome and cursor that blinked in time, the additional logic needed to modulate their pixel colors every second caused us to break timing, which we hadn't accounted for; we needed to introduce another one-clock-cycle stage in the visual pipeline solely for layering components on-screen. Similarly, when we decided to add colored guide notes along the bottom of the screen in guided-play mode, we had to redesign the `free_play` module as well to set those note colors to white because they weren't needed.

6. Contributions

Elise designed and implemented the audio portion of the project, and Anahita designed and implemented the visual portion of the project. We worked together on the report, each writing the sections and creating figures pertaining to our portion of the project, with Anahita performing the majority of the evaluation calculations.

We used Will Green's integer square root module from the open-source Project F repository available on GitHub.

References

1) Project F Library: Square Root (Integer). Will Green, 2021. MIT. https://github.com/projf/projf-explore/blob/main/lib/maths/sqrt_int.sv