FPGA-Based "Sounds of Music" Final Report

1st Sahil Pontula
Departments of Physics, Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
spontula@mit.edu

2nd Benjamin Wu
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA, USA
benwu12@mit.edu

***Abstract*—We demonstrate an Field Programmable Gate Array (FPGA)-based piano and additionally develop an end-to-end key modulation pipeline that receives single-note melodies, transposes them to a different key, and transmits the resulting melody. The FPGA piano is built with added functionality (e.g., different timbres, VGA display, and multiple note selection (chords)) to allow the user flexibility and convenience beyond that of a tone generator. We also develop a "player piano" that modulates the user-specified key and performs stored tunes. Together, these pipelines combine elements of audio processing (e.g., fast Fourier transforms) and digital logic-based music theory to showcase the versatility of FPGAs in audio signal reception and transmission. The design is centered around the Nexys 4 DDR FPGA, with added elements to receive and transmit analog audio signals at the ends of the pipelines.**

***Index Terms*—Audio processing, Field programmable gate array, Pulse width modulation, Fast Fourier transform**

I. Introduction (Sahil)

This project explores audio pipelines and processing using digital logic in a musical setting. We first design the equivalent of an FPGA piano, which takes "notes" as inputs from different sources (computer keyboard/switches) and maps them to their corresponding frequencies via internal digital logic. Afterwards, we drive a speaker with pulse width-modulated output to play these frequencies using different waveforms and implement a VGA display as an added user interface for note visualization. We show how the FPGA piano design (and its variants) implement the following objectives:

1) Basic single frequency tone generation,
2) Different sound qualities via distinct waveforms (i.e. timbres) that have contributions from higher harmonics relative to the fundamental frequency (e.g., sawtooth and square waves),
3) Chords via the ability to play multiple notes simultaneously, and
4) Key modulation/transposition via the extension of a "player piano" which performs tunes predefined as a sequence of scale degrees.

The second part of this project involves transposing music via key modulation.

II. FPGA Piano (Sahil)

A. Basic Piano

In this section, we describe the audio pipeline that takes user input and produces notes of different frequencies via PWM audio output. The overall block diagram is presented in Figure 4. The piano consists of the following elements:

1) Computer keyboard - user input to specify one of 12 notes in given octave
2) Nexys 4 DDR (Digilent) Field Programmable Gate Array (FPGA) - other user inputs and synthesis
3) Computer monitor - VGA display
4) Speakers with audio jack (and volume control)

User input occurs through switches/keyboard for note selection (one of 12 possible notes in a given octave) and buttons `btnu`/`btnl` to control the octave. Table I gives the mapping of piano keys to computer keyboard keys. (This mapping is performed using the PS2 protocol: that is, more precisely, the note index is determined from the PS2 code produced by a PS2 decoder module.) When switches are used, sw[3:0] directly give the note index, with 1 corresponding to C and 12 corresponding to B (last note before next octave). Users can also control the timbre - the setup is currently designed with seven waveforms: square, sawtooth, sine, piano, violin, trumpet, and flute (Figure 2). Apart from the square wave, these waveforms are generated in Python scripts and encoded as look-up tables in .mem files with 256 16-bit values per period (values are in the range $[0, 2^{16})$). In order to generate waveforms for the instruments, publicly-accessible .wav files were downloaded and read using `librosa` [1,2]. The resulting audio samples were downsampled to 256 values (for a single period) and rescaled to the range $[0, 2^{16})$, then saved in a waveform BRAM. The .wav files were for the note C4: though the different waveforms produce the correct frequency, some timbres are harder to achieve than others (e.g., flute).

The note index is used to pull the period of the corresponding note (in clock cycles) from a BRAM (which stores periods for the octave C4-B4). This period is calculated according to

$$\texttt{period} = \frac{\texttt{clock_frequency}}{\texttt{note_frequency}}, \quad (1)$$

where $\texttt{clock_frequency} = 65$ MHz to avoid clock domain crossing with the VGA pipeline and `note_frequency` denotes the note's frequency in Hz (e.g. A440 is 440 Hz). Once this period is extracted, it is bit shifted based on the user-specified octave. Specifically, $\texttt{octave} = 2$ corresponds to the octave C4-B4 ("middle octave"), so that for example the period is reduced by 2 if $\texttt{octave} = 3$ (higher frequency) or increased by 2 if $\texttt{octave} = 1$. The resulting octave-corrected period is

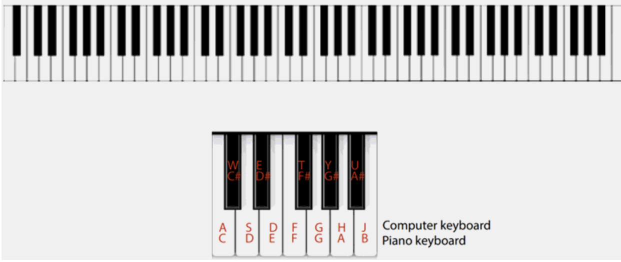

Fig. 1: VGA display consisting of seven-octave piano keyboard and 12-note octave with mapping between computer keys and music note names indicated. One block sprite spans the current octave in the top keyboard and a second sprite is overlaid on the current key.

divided into 256 intervals, corresponding to each value in the waveform BRAM. These analog values were then passed to a pulse width modulator (PWM) that performs a comparison to a sharply varying sawtooth wave to create a digital signal whose duty cycle corresponds to the analog signal. The PWM signal is then used to directly drive the mono audio output on the FPGA.

Separately, the note index and octave are passed to the `piano_drawer` module, which takes in $(\texttt{hcount}, \texttt{vcount})$ from a VGA module and outputs pixel color such that we obtain two block sprites corresponding to the current octave and key overlaid upon a custom-made piano image (stored as .mem file with 256 color look up `palette.mem`) via α blending (Figure 1). The sprites allow users to see which note/octave they are playing in real time. The VGA sequence is pipelined to avoid artifacts at the edges of the screen.

1) Design Evaluation: The FPGA piano pipeline has a worst-case negative slack (WNS) of 1.882 ns, which results in little worry about the stability of operation. Note that the timescales for audio considered here are on the order of milliseconds, which is significantly slower than timescales intrinsic to the FPGA such as the clock cycle. This means timing is generally not an issue (users will also change input on a timescale much longer than the clock cycle, of course). The design is also very low on resource utilization, with 0.51% of LUTs used for logic, 0.16% of LUTs used for memory, and 0.20% of registers used as flip flops. DSP utilization

Note	Keyboard key	sw[3:0]	"One-hot encoding"
C	A	1	000000000001
$C^\sharp$/$D^\flat$	W	2	000000000010
D	S	3	000000000100
$D^\sharp$/$E^\flat$	E	4	000000001000
E	D	5	000000010000
F	F	6	000000100000
$F^\sharp$/$G^\flat$	T	7	000001000000
G	G	8	000010000000
$G^\sharp$/$A^\flat$	Y	9	000100000000
A	H	10	001000000000
$A^\sharp$/$B^\flat$	U	11	010000000000
B	J	12	100000000000

TABLE I: Mapping of piano keys to computer keyboard keys.

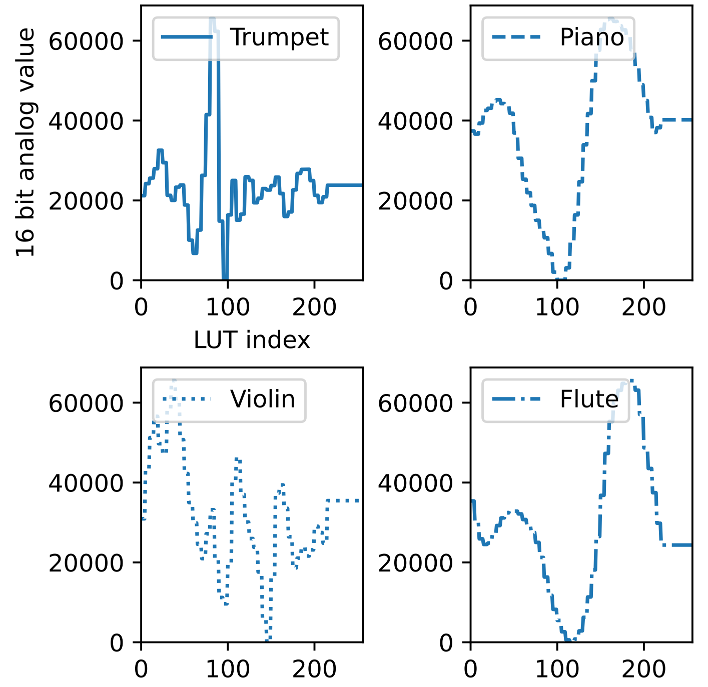

Fig. 2: Single period waveforms for piano, flute, violin, and trumpet. Calculated from C4 audio signal and rescaled to 256 16-bit analog values. The sampling rate can be increased for higher resolution.

is 0.42%. However, 50% of BRAMs are used - memory utilization comes mainly from LUTs storing note periods in clock cycles ($20 \times 12 = 240$ bits) and waveform LUTs ($256 \times 16 \times 5 = 20480$ bits, or 2.56 kilobytes). This usage is quite large and could be reduced by using 8-bit encoding for the waveform LUTs and floating point representation for the `note_count` lookup. More efficient encodings are possible: for example, one could only store a quarter cycle of the sine wave and use combinational logic to encode relationships between different quadrants of the unit circle.

As mentioned above, the VGA pipeline must also be pipelined to avoid small artifacts at the left edge of the piano display: in particular, 4 clock cycle delay was introduced before the blanking logic due to pulling from the BRAMs encoding the piano image sprite.

B. Player Piano

As an extension to the design presented in Section II-A, we next consider the related goal of constructing a "player piano," which plays a stored melody given user inputs. As shown in the second block diagram of Figure 4, the main differences from the FPGA piano pipeline are as follows:

1) User selection of a note now corresponds to a key rather than a note index (this can still occur via computer keyboard or onboard switches). Together with a switch to control major/minor quality, this completely specifies the key (e.g., C major or F minor) that the tune will be performed in.

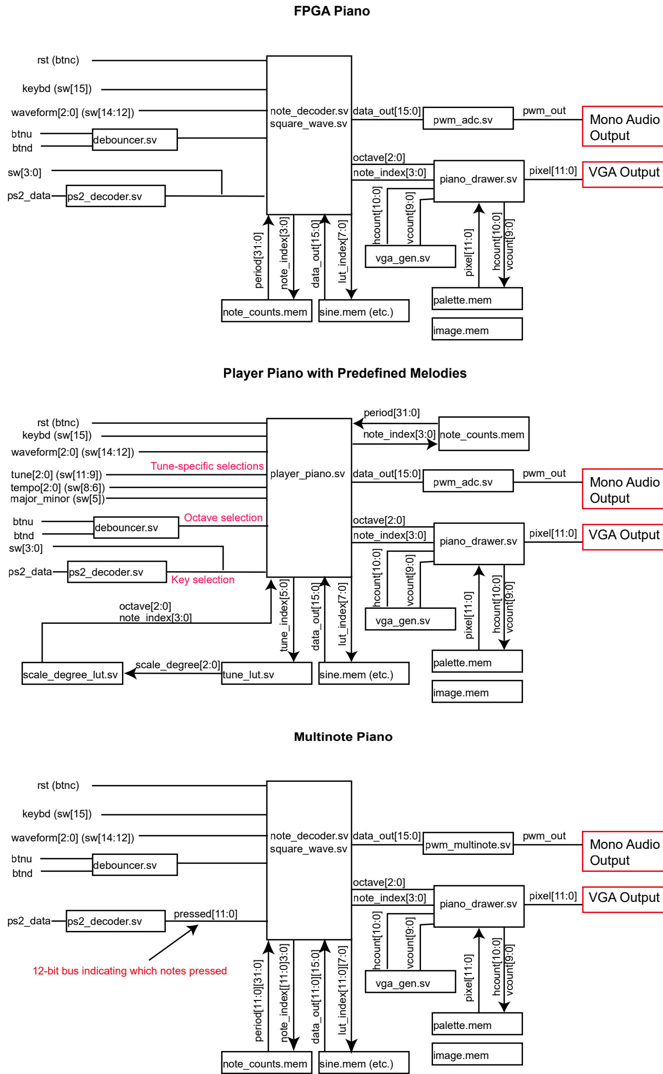

Fig. 3: Top: Block diagram for FPGA-based piano. The user has control over the note, octave, and waveform (e.g., sine, sawtooth, etc.); loudness is just controlled externally via the speakers. Internal digital logic stores the user's choice as a note index and performs look ups to sample the specified waveform at the correct frequency. The resulting analog signal undergoes pulse width modulation and drives an external speaker. For user convenience, a VGA display shows the current note/octave. Middle: Modified design for a "player piano." The user now specifies a key (together with major/minor quality) and can choose a tune, which is encoded as an LUT of scale degrees. This LUT is decoded to give a note index and octave which are then processed according to the FPGA piano pipeline. Bottom: "Multinote piano" design. In contrast to earlier designs, we respresent the note index as a 12-bit bus `pressed` that indicates which notes were pressed (1) and which notes were released/not pressed (0). Computation of the analog signal is essentially performed in parallel for each of the 12 possible notes and PWM is performed over the note-averaged composite analog signal.

2) Users have control over the tune itself (e.g., an ascending scale, arpeggio, or nursery rhyme) and tempo at which the tune is performed.
3) The central part of the pipeline occurs as follows. The tune is written as a series of scale degrees in the `tune_lut` module, comprising a key-independent encoding of the tune. The relationship between the scale degree and note index depends on the key selected and may involve a change in octave if needed (shown in Table II for C major). This is taken care of by a separate module `scale_degree_lut`.

In addition to acting like a musical jukebox, this design gives a proof-of-concept demonstration of key modulation. Specifically, tunes are defined by key-independent scale degrees and can be transposed internally by matching corresponding scale degrees of the original and transposed keys. Mathematically, if a given note is the ith scale degree of a key $f(x)$, the transposed note in another key $g(x)$ is simply $g(i)$ (here, f, g are mappings of scale degrees to notes modulo octaves).

The transmitting stage of this pipeline is analogous to that presented in Section II-A. In particular, the processed note index and octave are used to drive the VGA display and mono audio output pipelines as outlined above. Modules designed here, in particular the `scale_degree_lut` are shared with the key modulator pipeline described below.

1) Design Evaluation: The 12 major/minor key scales comprise a total of 96 bits and are simply defined as packed arrays and stored in DRAM on the FPGA. Similarly, the tunes are also defined as packed arrays, totaling only around 240 bits. The encoding of tunes by scale degrees and implementation of key scales as "lookup tables" significantly reduces the memory requirements of this pipeline. A naive brute force method that encoded lookup tables for each key and tune combination would require nearly 25 times the memory. Thus, using our encoding, the player piano can handle tunes of hundreds of beats without incurring significant memory cost. Currently, `tune_lut` is designed such that a scale degree is assigned to the smallest unit of time in the tune. For example, a tune whose smallest unit of time is a quarter note would be represented as as a packed array of dimension equal to the total number of quarter note beats. This means that for longer beats (e.g., half notes in the above example), there is repetition in the

Note	Scale degree	Note index
C	1	1
D	2	3
E	3	5
F	4	6
G	5	8
A	6	10
B	7	12

TABLE II: Notes and their corresponding scale degrees and internal note index encodings for one octave of a C major scale. A change in note index by 1 indicates a half step; a change by 2 indicates a whole step.

encoding of a scale degree. (Note that rests are also encoded by a "scale degree" that, when read, does not drive the PWM audio output.) Thus, more efficient schemes that only store scale degree changes and the length of time to play each scale degree may be possible.

For concrete numerical values, the WNS is 2.712 ns (i.e. combinational logic does not need to be pipelined with registers). Resource utilization is still very low for LUTs and DSPs, though BRAM usage is around 49% (the same BRAMs as in Section II-A are being used). LUT utilization for memory is larger for this design since the predefined tune sequences are stored in DRAM (there is no reason not to have this on BROM besides the clock cycle delay which is irrelevant for the timescales of this problem).

C. Multinote Piano

As another variant on the FPGA piano design, we demonstrate a piano capable of playing multiple notes simultaneously. As shown in Figure 4, the note encoding is fundamentally different from the previously described schemes. The previously used note index is replaced by a 12-bit bus `pressed` (similar to one-hot encoding), with each bit toggled on/off depending on whether the corresponding computer keyboard key is pressed/released. Computation of the analog signal is done in parallel for each pressed note according to the pipeline described in Section II-A, the resulting analog signals are summed and averaged over the number of pressed notes before pulse width modulation. Averaging is done to keep spectral energy density (i.e. audio volume) roughly constant when single or multiple notes are pressed.

1) Design Evaluation: Without pipelining, the design does not meet timing due to the usage of `divider.sv` in averaging the analog signals (this is relevant when the number of active notes is not a power of 2; when it is, simple bit shifts can be used). Instead of pipelining, we employ another method. Since the divisors d are very small (number of active notes must be less than 12) while the dividend n is much larger (sum of analog signals can exceed 2^{16}), using the division algorithm in `divider.sv` is somewhat efficient. We also do not need a very accurate average: small fluctuations in the spectral energy density will translate into negligible fluctuations in volume. Thus, we express division by d as an approximation of sums of bit shifts which are much faster to compute:

$$\frac{1}{3} = \frac{1}{4} + \frac{1}{16} + ...$$
$$\frac{1}{5} = \frac{1}{8} + \frac{1}{16} + ...$$
$$\frac{1}{6} = \frac{1}{8} + \frac{1}{32} + ...$$
$$\frac{1}{7} = \frac{1}{8} + \frac{1}{64} + ...$$

Furthermore, care must be taken in pipelining the VGA-related modules in this design. In particular, in `piano_drawer.sv`, the key block sprite is essentially being instantiated 12 times combinationally each clock cycle. `pixel_out` is computed as the sum of the pixel values of all of the block sprites and the piano image sprite (possibly with α blending if overlapping). A naive implementation that only pipelines the syncing and blanking logic for the VGA as in the above designs (due to the 4 clock cycle delay incurred by the image sprite) fails here: it eliminates artifacts at the edges of the screen, but a WNS of nearly -3 ns is obtained. To avoid this issue, we introduce 5 registers between the 12 block sprite pixel calculations and the combinational `pixel_out` calculation. The WNS is then 1.331 ns. This turns the VGA pipeline into a 6-stage pipeline including blanking logic, so that the latency is $L = 6 \cdot 1/t_{\text{clk}} \approx 92$ ns and throughput is $T = 1/t_{\text{clk}} = 65$ MHz. The latency might be optimized further, but for our applications it is not important given that the timescales of operation are on the order of ms.

III. Key Modulator (Ben/Sahil)

A. Construction

The key modulator consists of the following elements:

1) PmodMIC3 microphone module
2) Nexys 4 DDR (Digilent) Field Programmable Gate Array (FPGA). Not the same as the one used for the FPGA Piano.
3) Speakers with audio jack

B. Design description (Ben)

There are three main pieces to the key modulation pipeline; in order, they are the extraction of the frequency of a note from audio, the actual transformation (modulation) of this frequency, and the output of this transformed note. The design of the third part is identical to the design of the pulse width modulation scheme for the FPGA piano, so we focus on the first two. The extraction of the frequency is the more complicated of the two, and can be broken up into a series of steps:

1) The PmodMIC3 consists of a microphone which picks up analog audio signals and a 12-bit Analog-to-Digital Converter. It interfaces with the FPGA board through the SPI protocol. The module `pmod_to_audio_spi.sv` facilitates the communication between the PmodMIC3 and the FPGA board. The MISO port passes in an audio sample one bit at a time, and the 12-bit audio sample output is passed downstream for processing. Our clock has a frequency of 104MHz, and it takes 16 clock cycles to produce one 12-bit audio sample (the first four bits are leading zeros), so the value of samples per second produced is 6.5MSPS.
2) We now seek to narrow the range of the frequency spectrum of the incoming audio signal so that there is sufficient frequency resolution in determining notes of interest. This is accomplished via downsampling in `downsampler.sv`. The notes lie in the frequency range of 130Hz to 530Hz, so our Nyquist rate has to be a bit more than a kilohertz. Audio signals have a maximum

frequency of around 20KHz, so running the signals through a LPF with passband range $\frac{1}{16} \cdot 2\pi$ means that the output frequencies all have maximum frequencies less than $20/16 = 1.25$KHz. We generate a low pass 31-tap FIR filter using the Vivado IP (`fir_compiler_1`); the (scaled) coefficients for this FIR filter are fed into the Vivado GUI and can be found using the command $\texttt{round}(\texttt{fir1}(30, 0.0625) * 1024)$ in MATLAB. To finish downsampling, we decimate the resulting output by a factor of 2400, giving us a samples per second rate of $6.5 \cdot 10^6/2400 = 2.708$KSPS, sufficient to avoid aliasing.

3) We now want to perform an FFT on the downsampled audio. We can do so with the `xfft` module in the Vivado IP, and setting the number of bins as 1024. We get a frequency resolution of around $2.708/1024 \approx 2.7$Hz, which is sufficiently small to differentiate notes from each other. After getting the frequency spectrum coefficients, we find their magnitude by summing the squares of their real and imaginary parts and then passing the result through the Vivado IP `CORSIC` square root module.
4) The 1024 magnitudes are written into a BRAM with port B having a clock rate of 65MHz, which we assume for the rest of the downstream modules. The reason for this is it is the appropriate rate for drawing pixels through a VGA connection, and we want to display the relative amplitudes of each frequency component as a bar graph. The width of the BRAM is 32 and the depth is 1024, so the BRAM will store around 4Kb, which is within its storage capacity. This is also the largest amount of storage we will need, as downstream modules will only need a frequency corresponding to one of the components, not all 1024.
5) `freq_extract.sv` reads from the BRAM and finds the frequency with the largest magnitude. The problems of noise and harmonics arises. I was not able to implement it, but a possible solution is to take advantage of the fact that there are a relatively small number of notes. For every note, we could play the same note a couple of times and average the frequency spectrum that results in a lookup table. Finding the note corresponding to a frequency spectrum with the highest similarity (cosine correlation is a criterion) might be more robust and also does away with the need for the next module in the pipeline, `closest_note.sv`. This module uses a lookup table with the frequencies of the twelve notes between C4 and G4, inclusive. It iterates through the twelve notes and possible octaves and returns the note-octave combination giving the smallest difference with the input frequency.
6) The `note_converter.sv` module takes in an old key, a new key, a note index and octave, and outputs the note index and octave in the new key. Applying music theory to note conversion allows for us to only have to use combinational logic (we only have to shift the note index and octave with simple arithmetic) and this module takes very little resources and time.

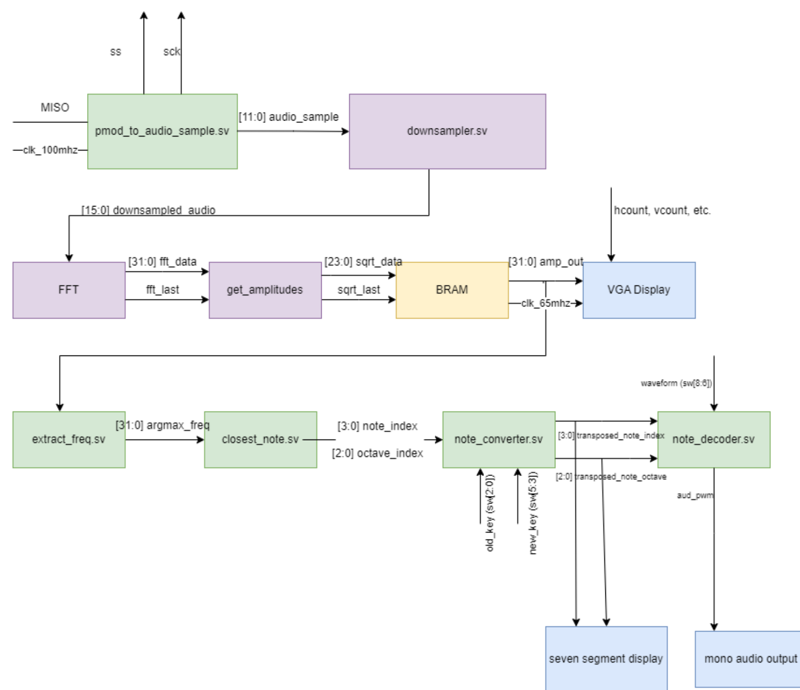

Fig. 4: Block diagram for the key modulator.

Finally, we pass the result of the key modulation into `pwm_adc.sv` to drive the speaker.

IV. Design Evaluation

The dependency chain of modules in this pipeline is pretty linear (one module after the other), so a worry is latency. Another worry is that a small discrepancy upstream can have enormous complications upstream.

V. Retrospection

A. *FPGA Piano and Variants (Sahil)*

This was a good opportunity to combine knowledge from music theory and digital logic to create fairly efficient implementations with increasing complexity. For example, bit shifts gave a simple way to produce different octaves, therefore only necessitating storage of frequencies for a single octave (12 notes). Utilizing scale degrees and converting between scale degrees and notes were central to the key modulation demonstrated in the “player piano.” This project also provided many opportunities to interface with material covered earlier in class, such as the PS2 protocol and VGA display, and practice with more performance-related design considerations (such as pipelining VGA stages) even though these were not as relevant for our objectives.

B. *Key Modulator (Ben)*

The key modulator was a great way to learn about audio signal processing and how it works on the FPGA. I enjoyed designing the modules and thinking about how to apply the skills from the labs to an open-ended project. Unfortunately, I was unable to debug the first half of the pipeline up to the

BRAM, which was the major technical difficulty. I take full responsibility for this (and emphasize that all of these failed deliverables were on my end and not my partner's). The end product was unable to work in hardware. Despite not getting it done in time I plan on getting it to work after the deadline and after that will critically evaluate possible improvements to the project in latency, memory usage, etc.

VI. Code Availability

Code for this project is publicly available at https : //github.mit.edu/spontula/fpga − sounds − of − music.

VII. Author Contributions

S.P. conceived the project idea, designed and implemented the "FPGA piano" and its variants, wrote the corresponding sections in this report, and created the block diagrams in Figure 4. B.W. designed and implemented the "Key Modulator", wrote the corresponding sections, and created Fig. 3.

VIII. Acknowledgments

We gratefully acknowledge useful discussions with Joe Steinmeyer. The code from Lecture 10 was adapted for the FFT, magnitude, and VGA display modules.

References

[1] McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E., & Nieto, O. (2015). librosa: Audio and music signal analysis in python. In Proceedings of the 14th python in science conference (Vol. 8).

[2] Ellis, D. (n.d.). Musical Instruments. Sound examples: Musical Instruments. Retrieved December 14, 2022, from https : //www.ee.columbia.edu/ dpwe/sounds/instruments/.