Project Report: Music Synthesizer

by

Keelin Becker-Wheeler

EECE7368: High Level Design of Hardware-Software Systems

Department of Electrical and Computer Engineering

Fall 2019

Becker-Wheeler, Keelin

Project Report: Music Synthesizer

Abstract – This project creates a system that synthesizes music in response to an input stream of musical notes. The system is composed of software that parses the input stream and a hardware accelerator that synthesizes the audio. Techniques for creating audio waveforms on a PWM channel are described, including combining signals and producing sinusoidal waveforms on the binary output. The system is modeled and simulated in SpecC, then implemented in Python and VHDL to be ran on a CPU and FPGA. The goal is to demonstrate ways in which a more general audio synthesizing application might be implemented on the hardware. Particular focus is made on areas such as voice synthesis and text-to-speech, which have importance in communication, translation, artificial intelligence, and in synthesized music. These systems can be implemented through software, but the waveform generation step for synthesized sound is better suited for hardware acceleration, especially when multiple channels of audio are synthesized. The implemented system shows how the structure and design of an audio synthesizer can be made in order to respond quickly to real-time input.

Contents

Chapter

1 Introduction **1**

1.1 Motivation 1

1.2 Related Works 2

1.3 Background 3

2 Design/Approach **8**

2.1 Simulation 8

2.2 Implementation 12

3 Results **18**

3.1 Experimental Results 18

3.2 Lessons Learned 22

4 Conclusions **23**

4.1 Future Work 23

References **25**

Appendix

A Supplemental Code Samples **28**

Figures

Figure

1.1 MIDI Note Number Chart 4
1.2 PWM Chord Generation 6

2.1 SpecC Simulation Code Structure 9
2.2 Excerpt from Wii Channel Music 10
2.3 Overall System Design 12
2.4 CPU Code Design 13
2.5 FPGA Code Design 15

3.1 Square Wave Simulation, Individual Notes 19
3.2 Square Wave Simulation, Chords 19
3.3 Sine Wave Simulation, Full 20
3.4 Sine Wave Simulation, One Period 21

Chapter 1

Introduction

1.1 Motivation

There are many use-cases where a system must produce sound in response to a stream of input, such as in: **Communication**—where the blind and vision impaired can rely on text-to-speech when navigating their digital devices[15], or where those with speech disabilities can become able to speak again through use of synthesized voices[12]. **Translation**—where systems are being developed that allow recorded speech to be directly translated into audio without any intermediate text representation[17], helping speakers of different languages to understand each other. **Artificial intelligence**—where modern devices already exist that speak with their users to communicate information[24], helping to create human-machine interactions that are more comfortable and understandable to humans. And, **Synthesized music**—where systems can be used that apply effects on human voices in order to create desirable sounds[29] or correct vocal pitch[19].

These applications show use-cases where systems must respond to real-time input. Due to the large range of possible inputs to these systems, storing pre-recorded outputs for every input is not very practical, or even possible. Thus, the process of generating the audio response to the input must be done quickly and efficiently. In such cases, hardware acceleration can be used to greatly benefit the response time of the system. Especially in this field of audio, for which parallelized audio channels[3] and digital signal processing seems so widely applicable to hardware implementation.

1.2 Related Works

One of the simplest applications for generating audio comes from the act of modifying an input voice. The primary examples of this are vocoders, which split an input audio into component parts, applying transformations on those parts in order to achieve a desirable effect[29]. An example project comes from the final project three individuals worked on in their master's course at Cornell University, an FPGA Speech Vocoder[10]. Their project shows an implementation of a vocoder that can apply a number of effects on their speech when input via a microphone. A series of transformations and filters are applied on the input signal in order to achieve this, but new audio is not synthesized. The example shows where an FPGA can be used to successfully process and output audio, and do so quickly in response to real-time input. However, the example does not show how audio can be newly generated.

Many methods which generate audio for text-to-speech applications use a technique known as Concatenative Speech Synthesis. This method relies on taking audio clips taken by voice actors or other sources, splitting the audio into it's individual speech components, and then using these samples as sources when forming new speech—concatenating the samples in the correct order to reach the finished result[23]. One example solution comes from "FPGA-based Implementation of Concatenative Speech Synthesis Algorithm"[6], a thesis written by Praveen Bamini for their Master's study in Computer Engineering at the University of South Florida. In this work, Praveen shows an implementation on FPGA for speech-synthesis using Concatenative Speech Synthesis. Such techniques are generally capable of creating highly-understandable speech results. However, these solutions suffer from memory constraints where a large amount of pre-recorded vocal data is needed in order for the system to operate. We see also the complexity involved in speech synthesis, wherein Praveen spent their whole thesis working towards the implementation.

The common theme between these example projects is that brand new audio is not being synthesized. Rather, the applications take existing audio and manipulate it in order to generate the output. Similar to these referenced projects, I aim to produce audio on an FPGA. Where my goals

differ is that I hope to synthesize the audio directly, showcasing an application that would produce results in more memory-constrained spaces. That is, where pre-recording and use of existing audio samples are not practical.

1.3 Background

Due to the sheer complexity of synthesizing vocal audio—and especially due to my inexperience in the field—my project aims to model an approach that might be used to similarly implement the systems mentioned before, but the example application I will focus on creating is a music synthesizer. A system will be created that can respond to a real-time stream of musical notes and synthesize the audio waveforms that correctly play the associated music. In order to synthesize musical audio, a number of concepts and representations must first be understood.

1.3.1 MIDI Notes

The means of representing notes will be through the MIDI note standard[2]. In this standard, every note is associated with a number, and organized into an octave. In one octave, there are 12 available notes: C, C#, D, D#, E, F, F#, G, G#, A, A#, B. These notes are repeated across a number of octaves, starting at an octave "-1" and ending at an octave "9". The notes are numbered in order, starting at $C_{-1} = 0$ and ending at $G_9 = 127$. Figure 1.1 shows the organization of these notes.

These notes are all further associated with a specific frequency. Sound is of course caused by the movement of air at some frequency[16] and when that frequency is equal to that of any of these individual notes, a musical tone results. This audio frequency doubles on each increase in octave, wherein each successive note increases the frequency by a factor of $\sqrt[12]{2}$. The MIDI standard defines the frequencies relative to "Concert Pitch", an A_4, which has a frequency of 440Hz. From this information, a useful equation can be derived in order to calculate the frequency given any MIDI note value: $freq(\text{note}) = 440\text{Hz} * 2^{\frac{\text{note}-69}{12}}$ [5].

Figure 1.1: MIDI Note Number Chart[11].

Note	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	60	72	84	96	108	120
C#	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
D#	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
F#	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
G#	8	20	32	44	56	68	80	92	104	116	
A	9	21	33	45	57	69	81	93	105	117	
A#	10	22	34	46	58	70	82	94	106	118	
B	11	23	35	47	59	71	83	95	107	119	

1.3.2 PWM Waveform Synthesis

Once a frequency has been determined for a given note, a waveform must be generated. Many basic audio interfaces can only be written to via a single binary value. That is, the audio waveform must be generated with only a "1"(HIGH) or "0"(LOW) output.

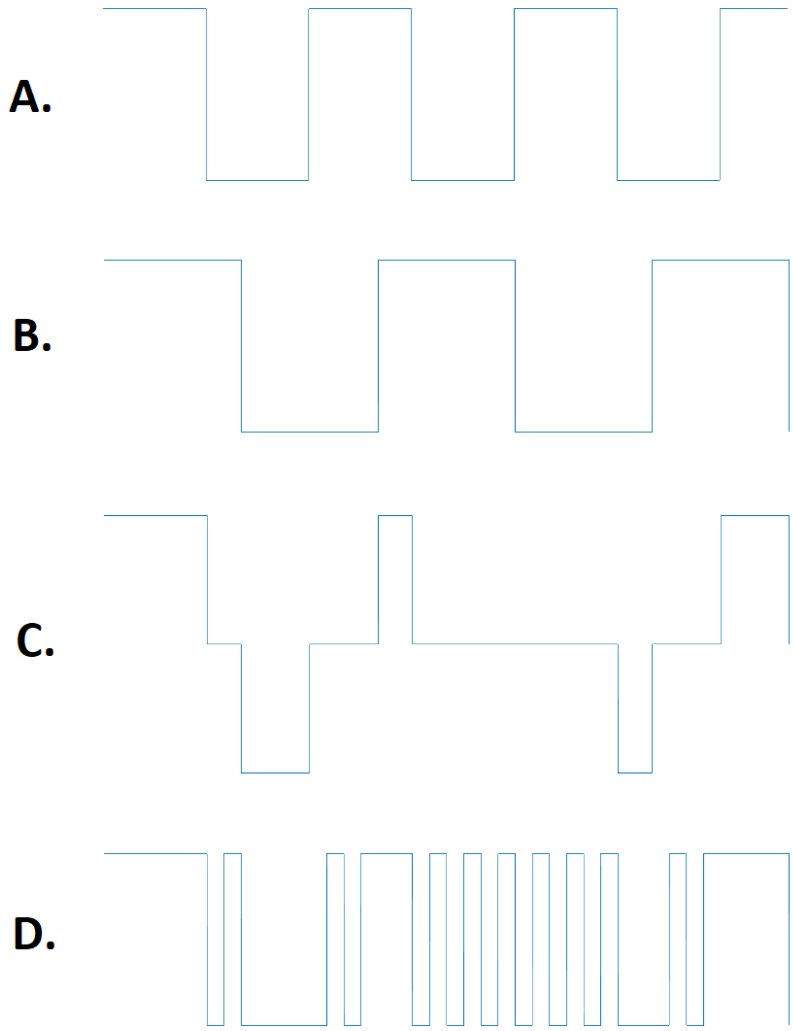
1.3.2.1 Square Wave Synthesis

A simple square wave can be generated at the desired frequency by toggling the output at twice the note frequency. That is, (i) write a HIGH value to the output, (ii) wait half the waveform period, (iii) write a LOW value to the output, then (iv) wait half the waveform period. Repeating this four step sequence will generate a square wave of the desired frequency on the output port.

The process becomes more involved when it becomes necessary to combine multiple signals into one. For instance, when playing multiple notes in a single chord. For these scenarios, the individual waveforms can be generated at their respective frequencies as normal. But before writing to the output, the waveforms are added together to create the new waveform. This process creates more than a binary number of signal levels, which cannot be written out onto the output audio interface. This waveform addition is shown in Figure 1.2 with waveforms (A) and (B) adding together to create (C). In order to produce the output, the intermediate values must be used in order to identify a duty cycle[4], which will be applied to the output. The duty cycle is simply applied by finding the percentage that the intermediate value falls between the HIGH and LOW values of the combined waveform. The output waveform is then written HIGH for this percentage of a sampling period and LOW the remainder of that period. Waveform (D) of Figure 1.2 shows an example of how a duty cycle might be applied.

Care must be taken when applying these duty cycles, as a sampling period that is too large can result in an ambiguous signal, which might be interpreted as containing a higher frequency note. Looking again at Figure 1.2, the higher frequency sections created from the duty cycle process could be indistinguishable from a higher frequency note played during that interval. In order to reduce

Figure 1.2: PWM Chord Generation, through combining multiple simple signals (A) and (B) into a complex waveform (C) then applying a duty cycle for resulting intermediate values in order to produce a binary output (D).



this ambiguity, a sufficiently high sampling period should be used, one at a frequency much higher than the highest note synthesizable in the system. For this system, the highest note, G_9, has a frequency of about 12.5kHz. In general, the highest frequencies humans are capable of hearing are around 20kHz[20].

1.3.2.2 Sine Wave Synthesis

In cases where more complex waveforms need to be generated, a quick method of generation is through the use of a wave table. In wave table synthesis, one period of a waveform is stored in memory and can be scanned at various speeds or trajectories in order to produce a desired frequency or sound[30]. In order to use this technique in a system requiring a binary audio interface, duty cycles can again be used. That is, when a value is read from the wave table, that value will be one of a range of possible values. The percentage that this value falls between the maximum and minimum values in the range will determine the duty cycle to use.

When many of these wave-table-synthesized waveforms need to be combined and played simultaneously, the same method of combining waveforms can be used as described in 1.3.2.1. That is, another duty cycle will be applied over those used to create the individual waveforms. As the construction of the duty cycle was defined based on a percentage within a range of values, such a hierarchy of applied duty cycles will have no substantial affect on the resolution or quality of the waveform even when each is applied at the same sampling frequency. That is, so long as the sampling frequencies used remain much higher than the highest audible note frequencies.

Chapter 2

Design/Approach

The method taken in completing this project was first to complete a model and simulation of the system in SpecC. Then, after verification of the basic functionality, to implement that model in real components to demonstrate a full working system. All source code can be found on the GitHub repository associated with this project[8].

2.1 Simulation

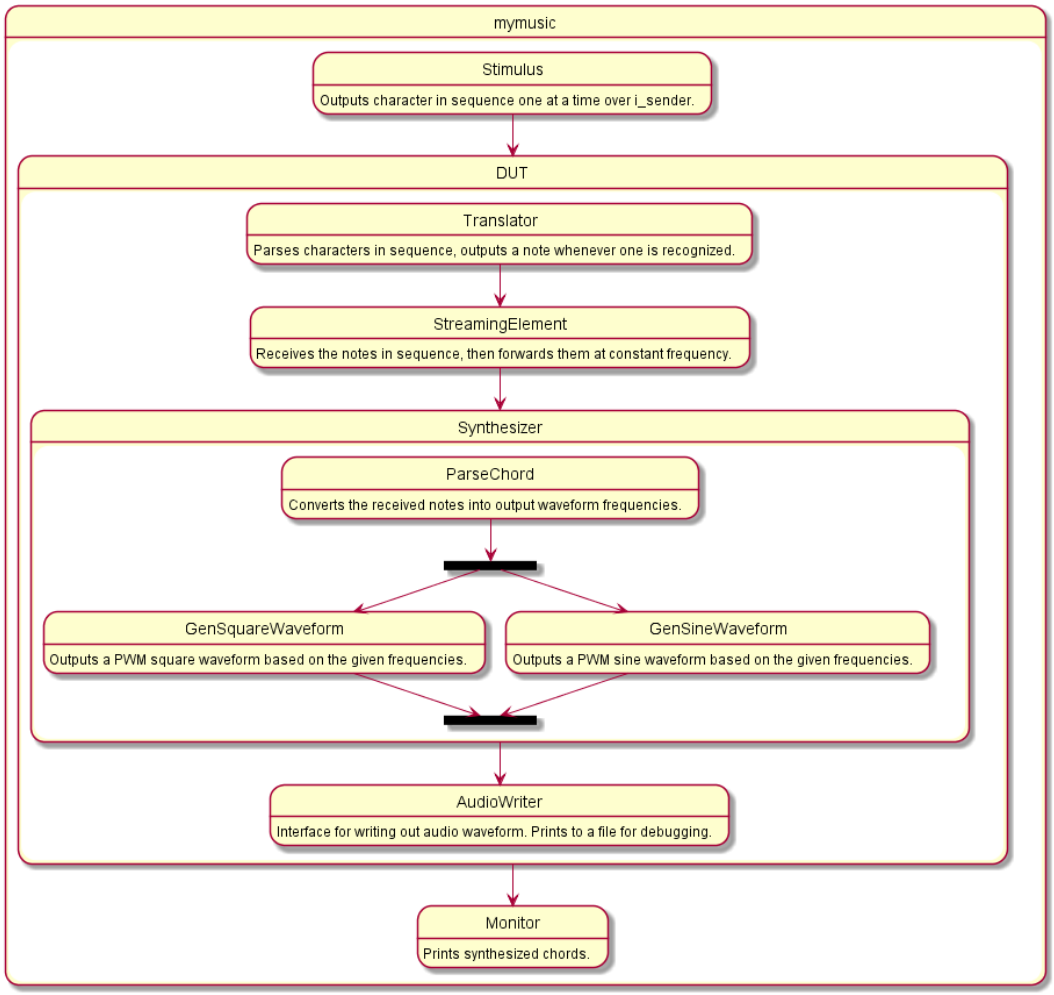
The structure of the SpecC simulation code is shown in Figure 2.1. A "Stimulus" behavior provides the input to the system, behaviors within the DUT operate on this input, and a "Monitor" behavior reports the results to be reviewed. Within the DUT, a sequence of behaviors operate on the data in a pipeline-like process.

2.1.1 Translator

Firstly, the "Translator" behavior receives the data from the Stimulus, one character at a time. It is expected that the input adheres to a specified format, so that the Translator can parse the input for instances of MIDI notes that are to be played. Each character in the input string will correspond to one of the characters shown in Table 2.1. The string will consist of a sequence of chords, i.e. "CHORD?+"[1] . The format of CHORD shall be ".|-|NOTE|(NOTE?)"[2] . The format

[1] Where the construct x?+ means zero or more instances of x.
[2] Where the construct x? means zero or one instances of x. And the construct x|y means either x or y.

Figure 2.1: SpecC Simulation Code Structure.



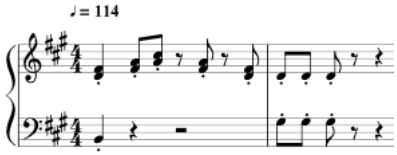
of NOTE shall be "{A-G}{b|#}?{0-9}"[3] . Note that this syntax does not allow entering the -1 octave, despite being valid within the MIDI note standard. If such low frequency notes really need to be played, then they can be reached by applying successive flats to any particular note.

A-G	Value, indicating the base value for the note.
b	Flat, results in decreasing the value of a note by 1.
#	Sharp, results in increasing the value of a note by 1.
0-9	Octave, a single-digit integer indicating the octave for the note.
()	Begin/end chord, notes placed within will be played simultaneously.
-	Repeat, corresponding to the repetition of the previously played beat.
.	Pause, corresponding to a break in the playing of musical data.

Table 2.1: Input Format

An example input sequence is given by "`(B2D4F#4)-(F#4A4)(A4C#5).(F#4A4).(D4F#4)(`
↪ `D4G#3)(D4G#3)(D4G#3)...`", corresponding to the notes shown in Figure 2.2. Given any piece of sheet music, such a sequence can be formed by applying basic knowledge of music notation [25][26].

Figure 2.2: Excerpt from Wii Channel Music, by Kazumi Totaka[28].



Whenever the translator completes parsing of an individual chord, that chord—in other words, a collection of one or more MIDI note values—will be passed to the next behavior of the system. For this system, a pause in the playing of notes is represented by a note value of -1.

[3] Where {} are used to indicate a single character whose value is one of those valid in the internal syntax. A-G and 0-9 represent the inclusive range of values.

2.1.2 Streaming Element

The "Streaming Element" behavior receives the chords sent from the Translator. Each chord is received one at a time and forwarded at a constant rate. This rate is determined by the speed of the given musical sequence. Any given piece of music is assigned a speed at which that music should be played, this *tempo*[1] determines how many beats of the music will be played per minute. For example, the tempo shown in Figure 2.2 is 114 beats per minute. When entering the speed for this song, care needs to be taken as the note sequence can contain chords at half-beat or smaller intervals. For Figure 2.2, a configured speed of 228 beats per minute will correctly play the sequence when represented by half-beats.

2.1.3 Synthesizer

The "Synthesizer" behavior receives the timed output of the Streaming Element. Whenever a new chord is received, the Synthesizer will convert the MIDI values within that chord into a collection of frequencies. This is done within the internal "ParseChord" behavior, which simply maps the MIDI value to a frequency using the methods as described in 1.3.1.

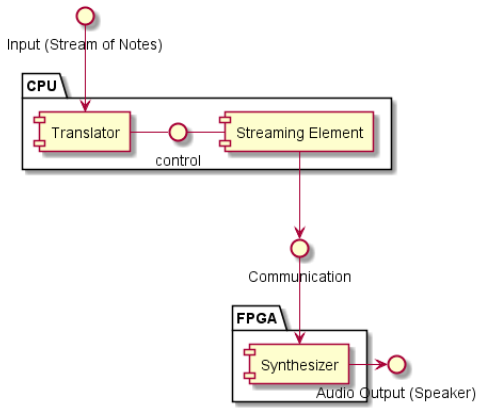
Next, a PWM waveform is generated, through a process as was described in 1.3.2. Either a sinusoidal or square waveform can be generated, depending on the configuration of the system. These waveforms are generated through the "GenSineWaveform" or "GenSquareWaveform" behaviors, respectively.

Finally, the generated waveform is fed into an "AudioWriter" behavior, representing the binary audio port as expected in the physical system. This behavior aids in simulation by writing the values of the final waveform to a file, so that the waveform can be graphed and visually verified later.

2.2 Implementation

Once the system has been modeled and verified through the simulation step, the system is implemented into real physical components. The "Translator" and "Streaming Element" behaviors from the SpecC model are mapped to software (within a CPU), and the "Synthesizer" behavior is mapped to a hardware accelerator (an FPGA). A UART protocol[7] is used to handle the communication between the two components. This system implementation is visualized in Figure 2.3.

Figure 2.3: Overall System Design.



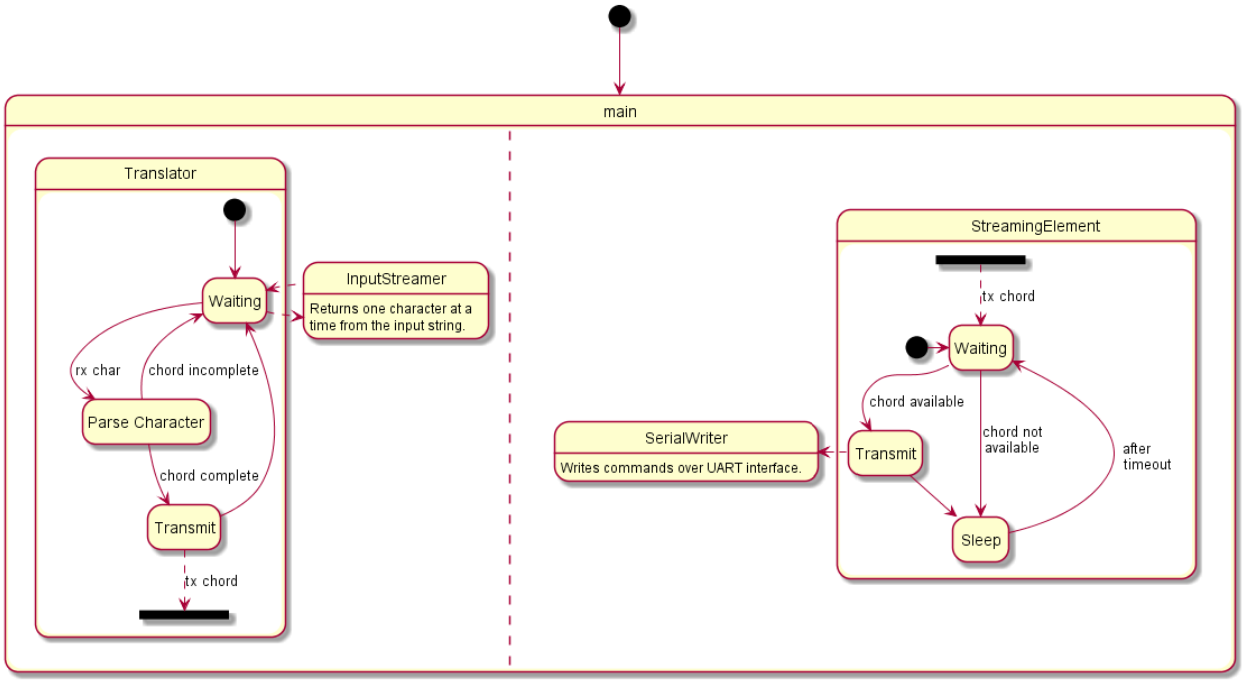
2.2.1 CPU

For the CPU of the implemented system, an Intel Core i7-8550U was used. This is the computer core as present within a personal laptop, used due to ease in availability.

A Python program was written based on the SpecC code for this portion of the system. Python was used due to personal familiarity and greater ease in tasks such as implementing UART communication, as compared to using the C-based SpecC. The design of this CPU component is shown in Figure 2.4.

Two parallel threads are implemented, one each for a "Translator" and "StreamingElement" python behavior. An "InputStreamer" is written which is called by the Translator in order to receive the input sequence one character at a time. The Translator parses the input sequence in

Figure 2.4: CPU Code Design.



the manor described in 2.1.1, transmitting a chord only when one has been parsed from the input.

The StreamingElement receives the chord from the Translator, via a thread-safe Python Queue object[21]. This process of receiving the chord is non-blocking in the StreamingElement, so that when no chord is available, the thread can sleep. Sleeping is used by the StreamingElement thread in order to implement the necessary timing of the music sequence, as described in 2.1.2. This allows the forwarding of received chords to occur in lock-step with the configured tempo.

Forwarded chords are sent through a "SerialWriter" which transmits the notes to the FPGA over a UART interface. When transmitting messages over the UART interface, a custom command format is followed. This command format allows the software to control the volume and playback state of the FPGA, in addition to the currently played notes. This format will be described in the next section.

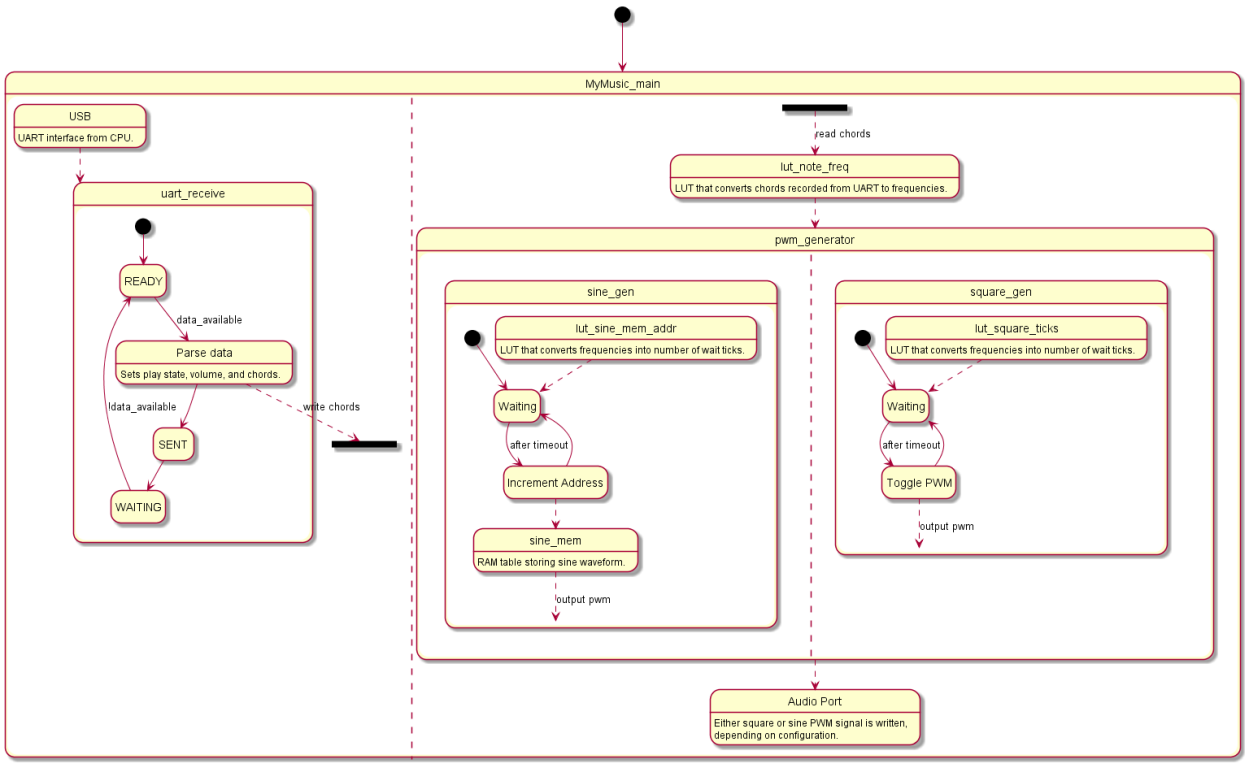
2.2.2 FPGA

For the FPGA of the implemented system, a Xilinx Artix-7 was used. This FPGA came packaged in a Nexys 4 DDR Development Board[13], providing the peripherals needed for the project. This FPGA was used due to ease in availability.

A VHDL program was written based on the SpecC code for this portion of the system. VHDL was used due to familiarity and because a process for synthesizing the FPGA bitstream from SpecC was not known. To aid in the rapid development of this VHDL code, some existing demo projects for the FPGA were used as a starting point[9][14][27]. These samples simply provided example implementation of a UART interface and outlined a structure that could be built-off of to develop the project. The VHDL code was written in a manor to mimic the functionality as was described through the SpecC code. The design of this FPGA component is shown in Figure 2.5.

Firstly, UART-receiving functionality is implemented in order to to receive the messages as sent to the FPGA from the CPU component. These messages are sent over a USB cable, connecting the FPGA to the laptop. The VHDL code implementing the UART state machine is borrowed from an online source[27]. The UART commands are received as single bytes and are interpreted by the

Figure 2.5: FPGA Code Design.



FPGA in the manor described by Table 2.2.

Byte Mask	Meaning
`0b1000XXXp`	When p=1, the system will enable audio playback. When p=0, playback is disabled.
`0b1010vvvv`	The audio volume will scale based on the 4-bit number v.
`0b0nnnnnnn`	The current chord is appended with the 7-bit number n, a MIDI note value.
`all others`	The current chord is appended with an empty note, i.e. a pause.

Table 2.2: Communication Command Format

The FPGA bitstream is compiled with support for a maximum number of simultaneous notes per chord. The FPGA expects to always receive this many notes each time a new chord is transmitted from the CPU. Due to this, the CPU will pad its transmission of chords with empty notes in order to reach the configured chord size. Recall from 2.1.1 that a pause in the playing of notes is represented by a note value of -1, e.g. `0b11111111`.

Once notes have been received, they are translated through a look-up table into an associated frequency. The method for determining the frequency was described in 1.3.1. This method was used to create a small Python script, used in order to generate the VHDL code for the look-up table, see S.1 of Appendix A. It is noteworthy that due to the number of bits used in precision for the frequency (14-bits), some of the lowest MIDI note values map to the same frequency value. This is hardly consequential, as those notes are well below the threshold of frequencies which create solid tones. In fact, the lowest frequency notes create clicks at a clearly audible interval, e.g. 8Hz. Additionally, empty notes aren't ignored, rather they are implemented as waveforms with a frequency of 0Hz.

The frequency output from this look-up table is then used as an input in additional look-up tables. These additional tables determine the number of clock ticks that must be waited during waveform synthesis[4] .

For the square waveform generation, this number of clock ticks corresponds to the half period

[4] In hindsight, these look-up tables could have been written to convert the MIDI note values directly into the number of wait ticks, which would have resulted in smaller tables and would remove the need for the frequency table.

of a wave with the associated frequency. Thus, after each timeout of these wait ticks the PWM signal for the note is toggled, creating the square wave. The look-up table for the square wave ticks is generated from Python, using the method of doubling the note frequency as mentioned in 1.3.2.1. See S.2 of Appendix A for the Python script.

For the sinusoidal waveform generation, this number of clock ticks corresponds to the time needed to wait before incrementing the address within the wave table. The value from the wave table is then used to create the sinusoidal wave, using the duty cycle method as described in 1.3.2.2. As with the other look-up tables, the table that finds the wait ticks between incrementing was generated from a Python script, S.3 of Appendix A. The math necessary for calculating these clock ticks uses the dimensions and increment step chosen for the sine wave table memory. In the sine wave table, a resolution of 4096 samples is used. With 4-bits per sample, a total memory size of 16K-bits is thus used for the wave table. The table is implemented in this manor as an IP was already known about and readily available from Xilinx that conveniently implements 16K-bits of memory with a bus size of 4-bits[31]. Additionally, when incrementing the address in this table, an increment of 9 addresses is used. This was decided as 9 was the minimum increment amount found which guaranteed that all unique frequencies from the 128 MIDI note values had a distinct waveform. This fact was of course determined through use of the table generation scripts in Python. Additionally, Python was used to generate the memory structure for the sine wave table, see S.4 of Appendix A for that script.

These steps of generating a PWM waveform are performed in parallel for each note composing the chord. This fact is also why the system is pre-compiled to support a maximum number of notes per chord, as the waveform generation must be copied for each parallel path of note generation. Each component waveform for the chord is then combined together using the duty cycle technique described in 1.3.2. A single combined PWM signal is created for each of the square and sinusoidal waveform data paths[5] . One of the PWM signals is chosen to be output to the audio interface, depending on the state of a toggleable switch on the FPGA.

[5] Take note that this aggregation of PWM signals is not explicitly diagrammed in Figure 2.5

Chapter 3

Results

3.1 Experimental Results

Running the simulation will create a file containing the PWM waveform as a sequence of 1's and 0's. This file can be graphed using a Python script provided in the codebase[8]. These values are written at an interval of one clock cycle. In order to match the chosen FPGA, the simulated clock ran at 100MHz. Three example sequences were simulated, each lasting 40ms, or 4 million simulated clock ticks. Each 10ms, the next note in the sequence was transmitted. This corresponded to a configured speed of 6000 beats per minute.

The first sequence simulated was used to verify the basic correctness of the square wave step for individual notes. The sequence used was "`G4G5G6.`". Figure 3.1 shows the results. The graphed results verify that the square wave was generated at the correct frequency, that the waveform changed notes correctly and at the appropriate times, and that the method for playing pauses in the musical sequence worked.

The next sequence simulated was used to verify the duty cycle implementation for combining wave tables. The sequence used was "`G4(G4G5)(G4G5G6).`". Figure 3.2 shows the results[1] . The graphed results verify that the duty cycle method worked for combining waveforms, up to at least the three simultaneous waveforms as tested. This process of combining the waveforms also did not break any of the other functionalities in the system.

[1] Note that compared to Figure 3.1, the waveform starts on the opposite value, this is because the code was changed to start at 0 rather than 1. After that change, a new graph for the first case was never bothered to be made.

Figure 3.1: Square Wave Simulation, for individually played notes. Graphed is the audio PWM response to the input sequence of "`G4G5G6.`".

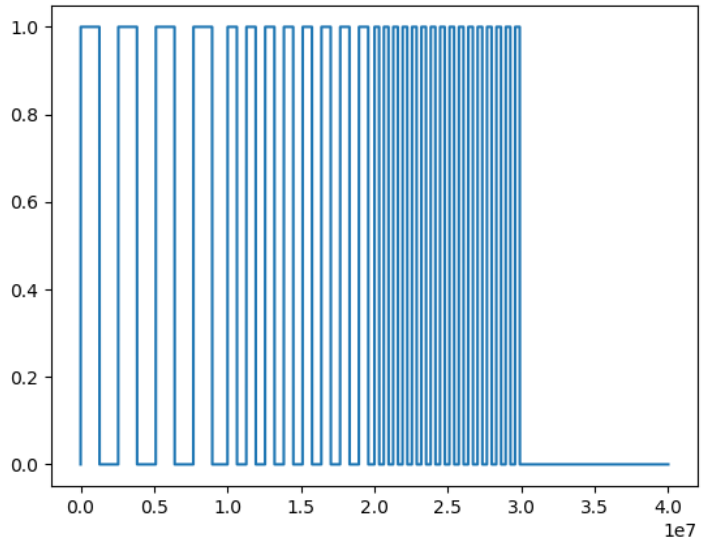
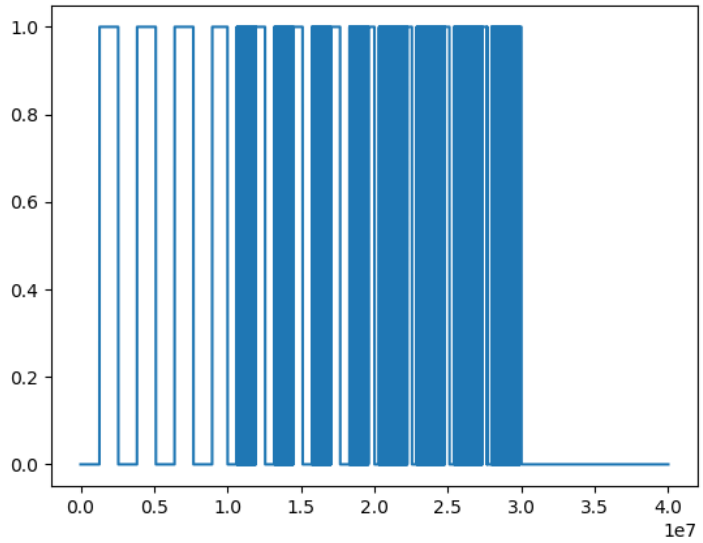
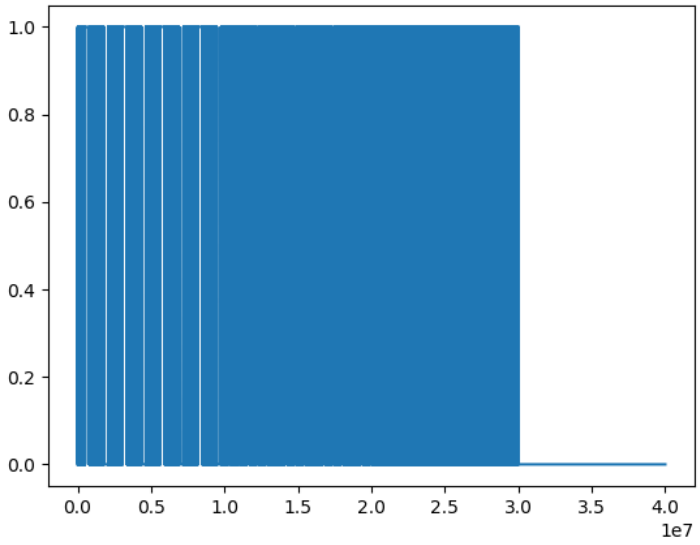


Figure 3.2: Square Wave Simulation, for simultaneously played notes. Graphed is the audio PWM response to the input sequence of "`G4(G4G5)(G4G5G6).`".



The final sequence simulated was used to verify the wave table method for generating sine waves based on a wave table. The sequence used was "`G4G5G6.`". Figure 3.3 shows the results and Figure 3.4 shows the results zoomed in around one period of a sine wave. The graphed results verify that a sine wave could be simulated using the duty cycle method. Additionally, the sine wave was simulated at the correct frequency for each of the notes and no other functionalities in the system broke due to this functional addition.

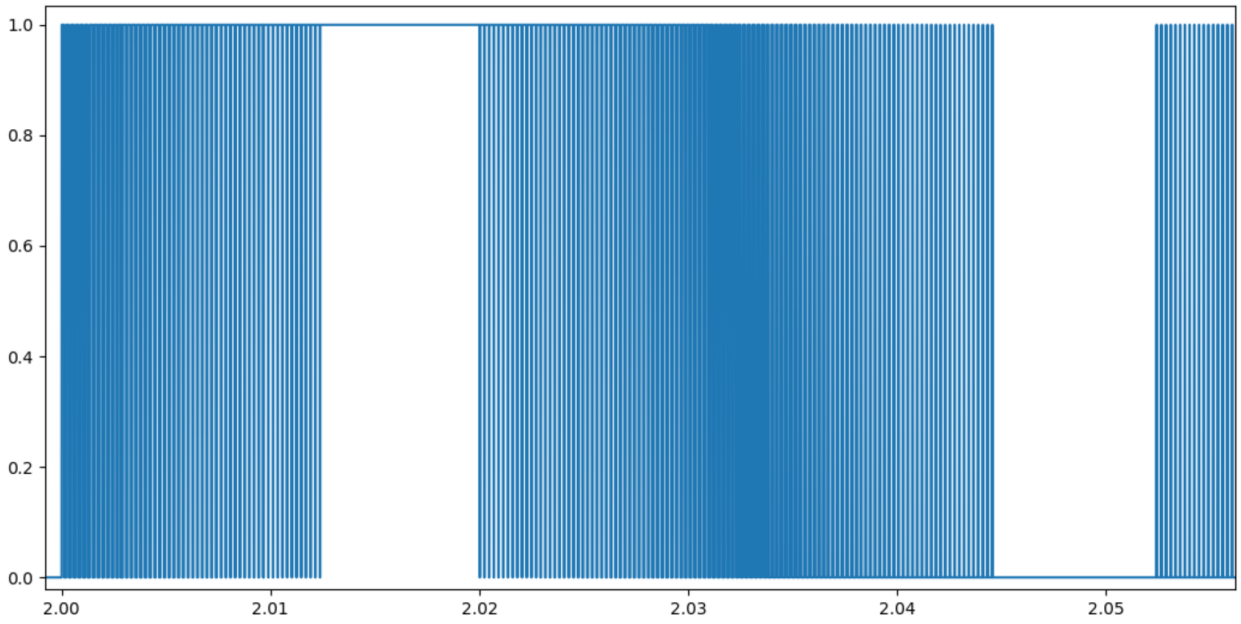
Figure 3.3: Sine Wave Simulation, the full output for the input sequence of "`G4G5G6.`".



Implementation of the system on the CPU and FPGA also succeeded. The system plays a large variety of musical sequences correctly[18][28][22], with appropriate tempos, pitches, and volume. The method of playing simultaneous notes also worked and the use of sinusoidal waveforms made a clear and expected change in the quality of the sound.

In all, the set goal of synthesizing music in response to a real-time stream of musical notation was accomplished.

Figure 3.4: Sine Wave Simulation, zoomed in to a little less than one period of a sine wave.



3.2 Lessons Learned

A number of observations were made while working on this project:

Firstly, using SpecC to simulate the system beforehand was a good choice. It turned out that compilation of the VHDL code took a rather long time (around 11 minutes). Should a very agile methodology be used, iterating changes while testing on the implemented system itself would have taken a while. By comparison, simulation of the 4 million clock cycles in SpecC took only a few seconds, and graphing the results took no more than 10 seconds in addition.

Furthermore, the choice of translating the SpecC code into native implementation languages proved to cause some extra steps in iterating, in order to have the implementation match the simulated functionality. Should direct code synthesis from the SpecC model be used, this extra effort would have been avoided. Additionally, the SpecC code itself was written much more abstractly than how the FPGA code was eventually implemented as. This extra abstraction in the simulation code made the process of verifying functionality quicker, but resulted in somewhat greater effort when translating the model into a hardware implementation.

Finally, I saw that making heavy use of my computer while it was transmitting audio sequences to the FPGA resulted in slight inconsistencies in the musical timing. This was due to contention over the CPU and system resources, plus scheduling from the computer's operating system. If a a dedicated CPU platform was available and used, these issues may have been avoided. This showed that dedicated computing resources are very important for timing critical applications.

Chapter 4

Conclusions

The problem I aimed to complete was that of creating a system that synthesizes new audio in response to real-time streamed input. I first modeled the system, using SpecC to iterate over the model and eventually verifying functionality through simulation. Afterwards, I implemented the modeled system on a real system combining hardware and software elements. Through this implementation, I continued to iterate over the model and simulate further when required. Finally, an implementation in real software and hardware was reached that successfully synthesized recognizable and correct musical audio in response to a streamed input of musical notation. Through this implementation, methods that can be used for constructing complicated sound waveforms on the PWM pin of an FPGA were also demonstrated and proven, namely the concatenation of simpler waveforms and the use of a reference wave table. Additionally, the real-time response to streamed input was integrated. Though, admittedly, the latency requirements for the audio application were not too strict. In all, the project goals were achieved and an interesting and easy-to-use system was put together worthy of showing friends and family.

4.1 Future Work

Implementing this project using devices I already own and have easy access to means I have the freedom to improve on or enhance the project in my own time. I will likely continue looking into the original motivations behind the project, namely text-to-speech and voice synthesis. Now that I have become a little more exposed to the theories behind audio synthesis in hardware—and I

have more time due to a lack of due-date constraints—I would love to look into adapting the system to play some sort of voice. Perhaps at a sooner step before then, looking into instrument-voiced music synthesis would be fun as well.

With the design concepts learned and practiced throughout this course, I feel a little more prepared for organizing and navigating myself through such complex projects. I look forward also to the challenges and practice!

References

[1] Beat and tempo. https://learningmusic.ableton.com/make-beats/beat-and-tempo.html. Accessed: December, 2019.

[2] MIDI Note Numbers for Different Octaves. https://syntheway.com/MIDI_Keyboards_Middle_C_MIDI_Note_Number_60_C4.htm. Accessed: December, 2019.

[3] Sound Channels and Polyphony. https://www.liquisearch.com/sound_card/general_characteristics/sound_channels_and_polyphony. Accessed: December, 2019.

[4] What is Duty Cycle. https://www.fluke.com/en-us/learn/best-practices/measurement-basics/electricity/what-is-duty-cycle. Accessed: December, 2019.

[5] Inspired Acoustics. MIDI Note Numbers and Center Frequencies. https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies. Accessed: December, 2019.

[6] Praveen Kumar Bamini. FPGA-based Implementation of Concatenative Speech Synthesis Algorithm. Master's thesis, University of South Florida, 2003. https://scholarcommons.usf.edu/cgi/viewcontent.cgi?article=2327&context=etd. Accessed: December, 2019.

[7] Circuit Basics. Basics of UART Communication. http://www.circuitbasics.com/basics-uart-communication/, 2016. Accessed: December, 2019.

[8] Keelin Becker-Wheeler. eece7368-project. https://github.com/keelimeguy/eece7368-project, 2019. Accessed: December, 2019.

[9] Keelin Becker-Wheeler. FPGA-Vivado-skeletons. https://github.com/keelimeguy/FPGA-Vivado-skeletons/tree/master/Nexys-4-DDR-__skeleton__, 2019. Accessed: December, 2019.

[10] J. Carvo, J. Joco, and T. Krishnathasan. FPGA Speech Vocoder. Master's project, Cornell University, 2019. http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2019/jc2697_jaj263_tk455/jc2697_jaj263_tk455/jc2697_jaj263_tk455/index.html. Accessed: December, 2019.

[11] Greg Cerveny. MIDI Note Number Chart for iOS Music Apps. https://medium.com/@gmcerveny/midi-note-number-chart-for-ios-music-apps-b3c01df3cb19, 2017. Accessed: December, 2019.

[12] Paulo A. Condado and Fernando G. Lobo. EasyVoice: Breaking Barriers for People with Voice Disabilities. In Computers Helping People with Special Needs, pages 1228–1235. International Conference on Computers for Handicapped Persons, 2008. https://www.researchgate.net/publication/221010467_EasyVoice_Breaking_Barriers_for_People_with_Voice_Disabilities. Accessed: December, 2019.

[13] Digilent. Nexys 4 DDR. https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start. Accessed: December, 2019.

[14] Digilent. Nexys 4 DDR GPIO Demo. https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-gpio-demo/start. Accessed: December, 2019.

[15] American Foundation for the Blind. Screen readers. https://www.afb.org/blindness-and-low-vision/using-technology/assistive-technology-products/screen-readers. Accessed: December, 2019.

[16] Benjamin Hollis. Physics of Sound. https://method-behind-the-music.com/mechanics/physics/, 2017. Accessed: December, 2019.

[17] Ye Jia and Ron Weiss. Introducing Translatotron: An End-to-End Speech-to-Speech Translation Model. https://ai.googleblog.com/2019/05/introducing-translatotron-end-to-end.html, 2019. Accessed: December, 2019.

[18] Koji Kondo. Great Fairy Fountain. https://sheetmusic-free.com/great-fairy-fountain-sheet-music-the-legend-of-zelda/. Accessed: December, 2019.

[19] Physics.org. How does auto-tune pitch correction work? http://www.physics.org/article-questions.asp?id=75. Accessed: December, 2019.

[20] Dr. Ir. Stphane Pigeon. High Frequency Range Test. https://www.audiocheck.net/audiotests_frequencycheckhigh.php, 2018. Accessed: December, 2019.

[21] Python. queue—A synchronized queue class. https://docs.python.org/3/library/queue.html, 2019. Accessed: December, 2019.

[22] Lena Raine. Old Site - Resurrections. https://musescore.com/user/14522131/scores/4984475. Accessed: December, 2019.

[23] Utkarsh Saxena. Speech Synthesis Techniques using Deep Neural Networks. https://medium.com/@saxenauts/speech-synthesis-techniques-using-deep-neural-networks-38699e943861, 2017. Accessed: December, 2019.

[24] Siri Team. Deep Learning for Siris Voice: On-device Deep Mixture Density Networks for Hybrid Unit Selection Synthesis. https://machinelearning.apple.com/2017/08/06/siri-voices.html, 2017. Accessed: December, 2019.

[25] All About Music Theory. Learn to Read Bass Clef Notes. https://www.allaboutmusictheory.com/musical-staff/learn-read-bass-clef-notes/, 2019. Accessed: December, 2019.

[26] All About Music Theory. Learn to Read Treble Clef Notes. https://www.allaboutmusictheory.com/musical-staff/learn-read-treble-clef-notes/, 2019. Accessed: December, 2019.

[27] Warren 'DoctorWkt' Toomey and 'Hamster'. VHDL UART RX for Nexys4 DDR? https://forum.digilentinc.com/topic/766-vhdl-uart-rx-for-nexys4-ddr/, 2015. Accessed: December, 2019.

[28] Kazumi Totaka. Mii Channel Music. https://musescore.com/static/musescore/scoredata/gen/6/1/6/3556616/cdacfa60d7f6b0832f8fcd9afe2c32756c4ea12c/score_0.svg?no-cache=1535929439. Accessed: December, 2019.

[29] Sophie Weiner. Cryptography and Cyborg Speech: The Strange Journey of the Vocoder. https://daily.redbullmusicacademy.com/2017/11/vocoder-instrumental-instruments, 2017. Accessed: December, 2019.

[30] WikiAdmin. Wavetable Synthesis. https://www.wikiaudio.org/wavetable-synthesis/, 2018. Accessed: December, 2019.

[31] Xilinx. RAMB16_S4. In Spartan-3E Libraries Guide for HDL Designs, pages 376–381, 2009. https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/spartan3e_hdl.pdf. Accessed: December, 2019.

Appendix A

Supplemental Code Samples

S.1 Script to generate lut_note_freq.vhd

```
import math

def freq(n):
    return 440 * 2**((n-69)/12)

def note_freq():
    print('        freq_value <= ', end='')
    for n in range(128):
        f = math.floor(freq(n)+0.5)
        print('                      "{:014b}" when "{:08b}",'.format(f, n))

note_freq()
```

S.2 Script to generate lut_square_ticks.vhd

```
import math

def ticks(f):
    return 100000000 / (2 * f) if f else 0

def half_ticks():
    print('        half_period_ticks <= ', end='')
    for f in range(0x4000):
        s = math.floor(ticks(f)+0.5)
        print('                             {:d} when "{:014b}",'.format(s, f))

half_ticks()
```

S.3 Script to generate lut_sine_mem_addr.vhd

```
import math

sine_resolution = 4096

def step(f):
    return 100000000 / (sine_resolution * f) if f else 0

def addr_step(step_amount=9):
    print('  nine_incr_ticks <=  ', end='')
    for f in range(0x4000):
        s = math.floor(step_amount*step(f)+0.5)
        print('                      {:d} when "{:014b}",'.format(s, f))

addr_step()
```

S.4 Script to generate sine_mem.vhd

```
import math

sine_resolution = 4096

def sine_lut():
    arr = []
    for i in range(sine_resolution):
        v = math.sin(i * math.pi/(sine_resolution/2))
        v = (v + 1) * (15/2)
        v = math.floor(v+0.5)
        if i % 64 == 0:
            print('            INIT_{:02x} => X"'.format(int(i/64)).upper(), end='')
        arr.append('{:x}'.format(v).upper())
        if i % 64 == 63:
            arr.reverse()
            print(''.join(arr), end='",\n')
            arr = []

sine_lut()
```