MusiLinx Audio Synthesizer

Final Report

ECE532H1 LEC0101
April 14th, 2023

Alexandre Singer
Arthur Bastos
Hamza Saddour
Fahim Rahman Talukder

1.0 Overview **1**
2.0 Outcomes **2**
3.0 Project Schedule **3**
4.0 Description of the Blocks **5**
4.1 Custom IPs 5
4.1.1 AudioVoice 5
4.1.2 Mixer 9
4.1.3 AudioSampleToAxiStreamAudio 10
4.1.4 Sequencer 12
4.1.5 TempoGenerator 13
4.1.6 AudioPulseGen 14
4.2 Xilinx / Digilent IPs 15
4.2.1 AXI IIC Bus Interface 15
4.2.2 I2S Transmitter 15
4.2.3 MicroBlaze 15
4.2.4 AXI GPIO 15
4.2.5 PS2-AXI Receiver 16
4.2.6 AXI Interrupt Controller 16
5.0 Description of the Design Tree **16**
6.0 Tips and Tricks **16**
6.1 Tips and Tricks for the Technical Aspect of any project 16
6.2 Tips and Tricks for Team Dynamics 17
7.0 User guide to Musilinx **17**
References **20**

1.0 Overview

MusiLinx Audio Synthesizer is an Audio Synthesizer with 32 voices tuned to the Western Music Scale with notes ranging from C3 (130.81Hz) to G5 (783.99Hz). All voices can produce classic Subtractive Synthesis waves such as Triangle, Sawtooth and Square waveforms and can be controlled and triggered via an external PS2 over USB keyboard in conjunction with the MicroBlaze soft-processor. Refer to Figure 1 for an overview of the system Block Diagram.

The motivation behind MusiLinx is the team's intrigue with music and how it is a shared language across all cultures. With this in mind, our team's intention was to create a product which is musically playable and accessible to people regardless of their educational background. Also, the history of technology centered around media and music is fascinating for its direct influence on the way music is created, captured, processed and distributed. With the advent of digital technology, music creation and most of the associated tools have moved toward fully digital CPU-based computers. As a result, where most of the technology was originally analog, and therefore able to process audio signals in real-time and mostly in parallel, the industry's move towards software has presented a challenge. The challenge being that software based solutions are not able to keep up with performance requirements, such as real-time processing, and are deployed on CPUs which are inherently serial. Hence, our team decided to create an instrument on an FPGA, since it will have the benefits of digital technology while exploiting the parallelism and real-time processing capabilities due to its faster processing speeds in comparison to CPUs.

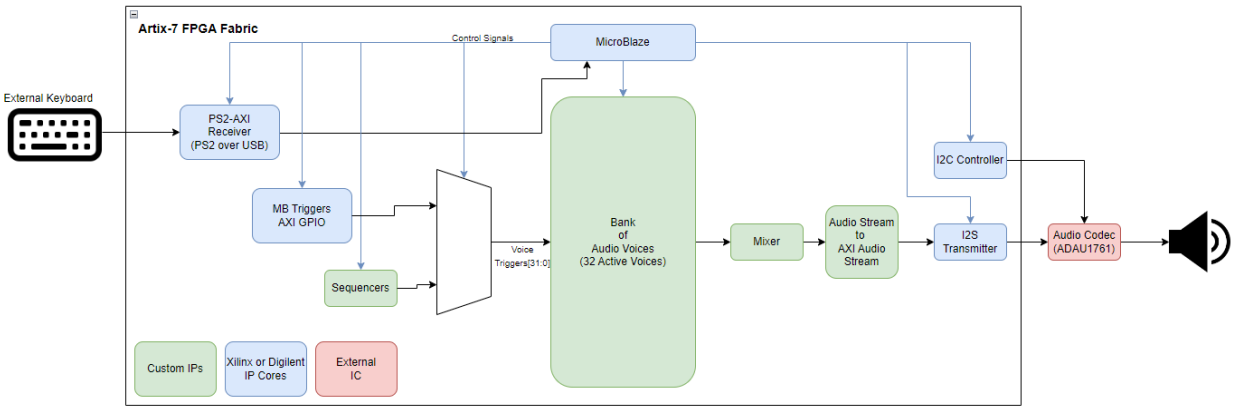

Figure 1: Overview of the system block diagram.

2.0 Outcomes

The outcomes of this project are broken into three categories: Results, Possible Further Improvements, and Future Designer Recommendations.

Results

MusiLinx worked as initially intended, as shown in Table 1, audio was synthesized by the different stages from creation, to propagation, to audio output. The tuning for all the voices is within +/- 5 cents and is playable over PS2 protocol using a keyboard. There are 5 different functional modes, including monophonic, chord, and piano modes. More audio voices can easily be added to the voice bank and more presets can be added for the different songs that can be arranged.

Table 1: Proposed Features and Their Statuses

Features	**Status**
Synthesize audio from sources, sequencers and ADSR.	Completed
Control ADSR and VCO modes.	Completed
Control key options (e.g., notes, chords, presets) with GUI.	Modified + Completed: Moved from GUI control to keyboard
Propagate audio through fabric to the output speaker.	Completed
Microblaze control for switching audio voices.	Completed

Possible Further Improvements

A polyphonic mode and control can be added to diversify the music produced. As well as creating a custom GUI to control input of the user-controlled keys, the GUI could have displayed system state in response to the keyboard input keys that are triggered. Finally, user input controls could be added to expand the options for audio production, such as increasing the range of ADSR values or adjusting the source waveforms frequencies.

Future Designer Recommendations

Working on a board with an easier navigation of an audio codec would be much more preferred as the one for the Digilient Nexys Video Board was not pre-configured to assist with audio propagation. Furthermore, modifications should be applied for the triggering of the audio voices to ensure timing constraints are not violated; as these signals travel long distances and tend to run into timing issues.

3.0 Project Schedule

The following is a detailed timeline of the project, milestones achieved, tasks completed, and any dependencies among tasks. The schedule is presented in the form of a Gantt chart, shown in Figure 2, which provides a visual representation of the project timeline. The team had decided to split the building of MusiLinx project into two stages: Stage 1 (Milestones #1 to #3) was to build a basic version of the Audio Stream and Stage 2 (Milestones #4 to #6) was to integrate multiple Audio Streams together to be controlled by the MicroBlaze. This split was chosen such that the team would have a working demo for the Mid-Project presentation.

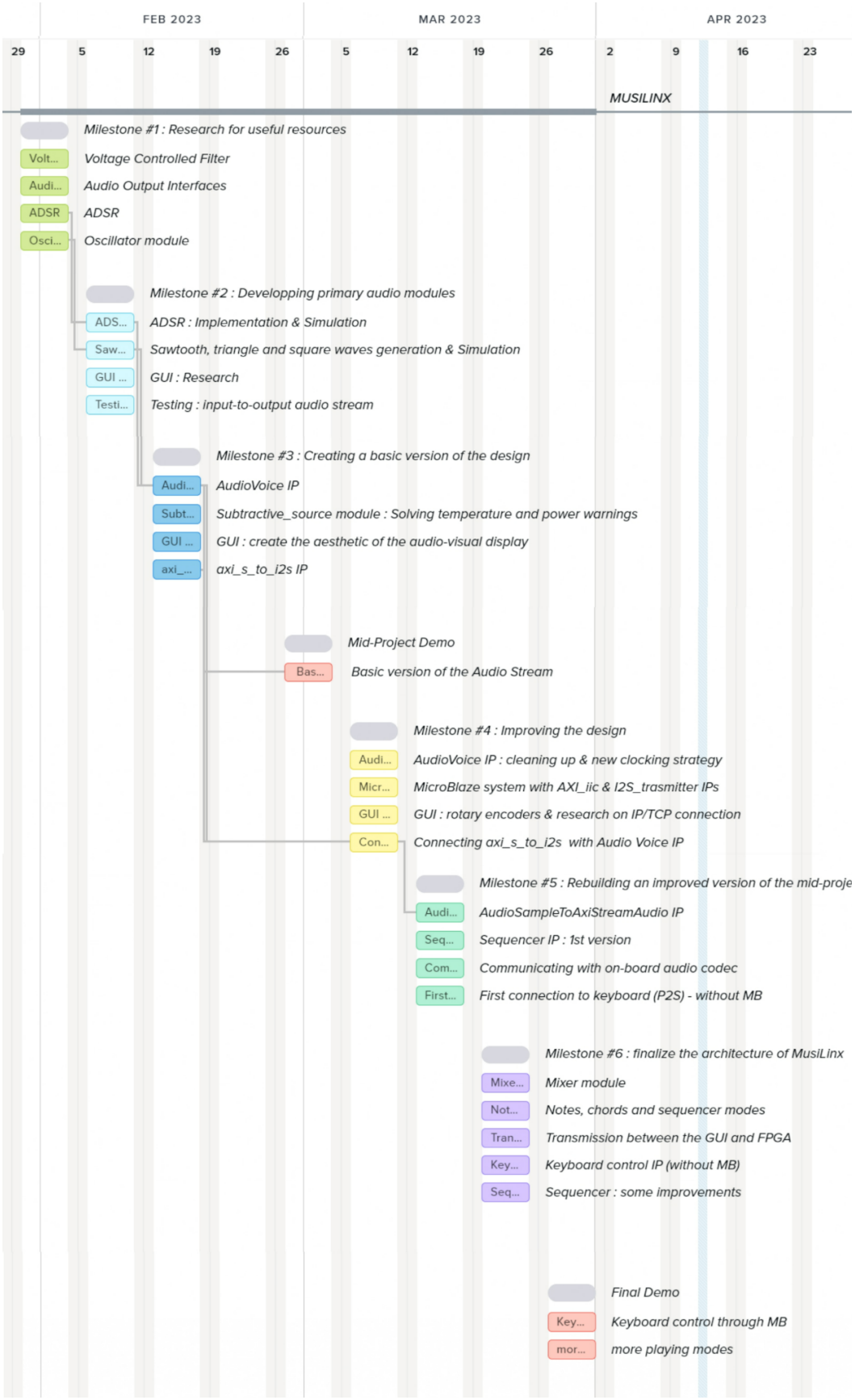

Figure 2: MusiLinx Project Schedule

4.0 Description of the Blocks

As shown in Figure 1, there are two types of IP blocks used within MusiLinx: Custom IPs (IPs created by the team) and Xilinx / Digilent IPs (IPs provided by Xilinx or Digilent).

4.1 Custom IPs

The custom IPs, created by this team, were used to generate and transmit audio signals based on user inputs. Since this is not a problem that has been done before, no existing IPs exist that fit MusiLinx's needs.

4.1.1 AudioVoice

The AudioVoice IP is responsible for generating the audio samples that are played by MusiLinx. As shown in Figure 3, the AudioVoice module is made of three sub-modules: the Oscillator, VCA, and ADSR. Each of these modules require parameters to be set, which is handled in the IP by an AXI-Lite Slave Register Interface. For this project audio samples are represented as 16-bit integers; as such, the output of the AudioVoice is a 16-bit integer representing the amplitude of the sound wave at a given point in time.

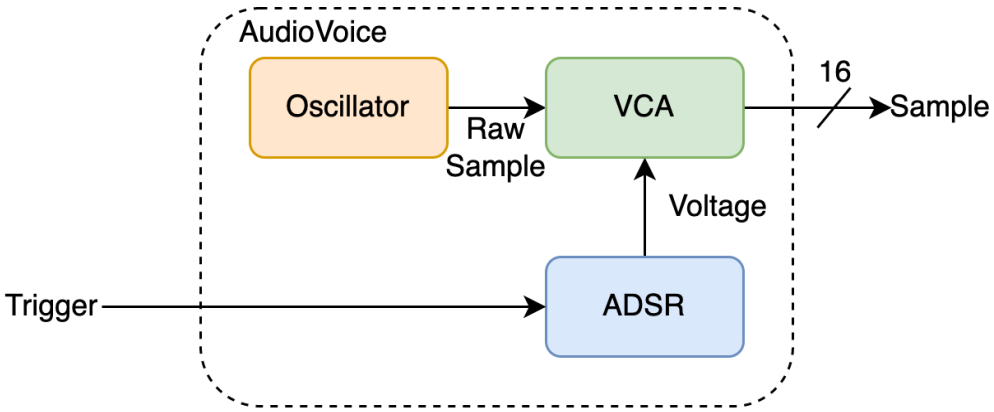

Figure 3: High-Level Diagram of the AudioVoice Module

The Oscillator Module

The Oscillator module is capable of generating three types of waves: triangle, sawtooth, and square. The wave that the Oscillator will generate is configured by the *wave_select* parameter. The period of the generated wave is configured by the *half_period* parameter, which is half of the desired period in audio clock cycles (sample rate). The *half_period* parameter can be derived from the desired audio frequency by the formula shown in Equation 1.

$$half\ period = \frac{f_{sample}}{2 * f_{desired}}$$

Equation 1: Calculating the Half Period From the Desired Frequency

The Oscillator module was built using a Finite State Machine with two states: “rise” and “fall”. During the rise state, the triangle and sawtooth waves will be rising and the square wave will be zero. During the fall state, the triangle wave will be falling, the sawtooth will continue to rise, and the square wave will be 1. Special care is taken during the transition from the fall state to the rise state for the sawtooth, which will be reset to zero. The states transition only if a pulse from the AudioPulseGen is sent to this module (as will be explained in Section 4.1.6).

The ADSR Module

ADSR stands for Attack, Decay, Sustain, and Release. Sound produced by real world instruments tends to have a very specific shape, as shown in Figure 4.

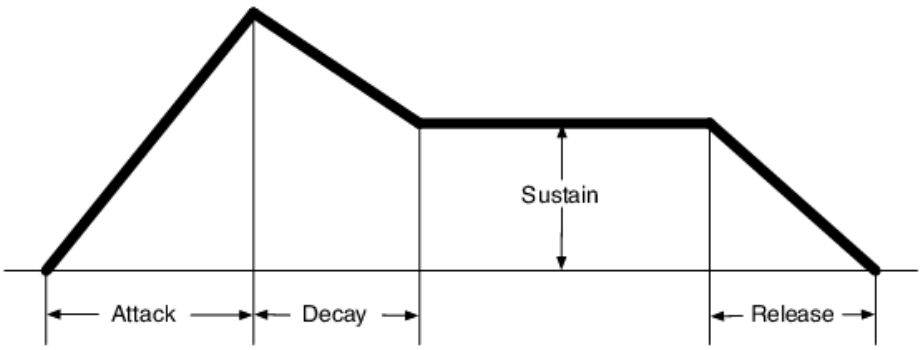

Figure 4: The Shape of the ADSR Curve

The Attack and Decay regions of this curve can be thought of as plucking a string on a guitar, which has a sharp rise to the max point before falling to a sustained region. The Sustain region corresponds to how long the note is heard (at a constant volume) before falling silent (the Release region). This ADSR module will hold the sustain value until the sustain duration is reached and the trigger is brought low. The ADSR module generates a 16 bit signal corresponding to the ADSR curve, with the *attack*, *decay*, *sustain*, *sustain_duration*, and *release* parameters corresponding to the size of these regions. The *attack*, *decay*, and *release* parameters (in seconds) can be found with Equation 2 and the *sustain_duration* parameter (in seconds) can be found with Equation 3. The *sustain* parameter is just the 16 bit signal value the sustain region should be held at.

$$parameter\ value\ = \frac{2^{16}-1}{t_{desired} * f_{sample}}$$

Equation 2: Calculating the *attack*, *decay*, and *release* Parameters From Duration (s)

$$sustain\ duration\ = t_{desired} * f_{sample}$$

Equation 3: Calculating the *sustain_duration* parameter From Desired Time Duration (s)

The ADSR module was built using a Finite State Machine with 5 states: Wait, Attack, Decay, Sustain, and Release. This Finite State Machine is shown in Figure 5, where ‘counter’ is the output of the ADSR.

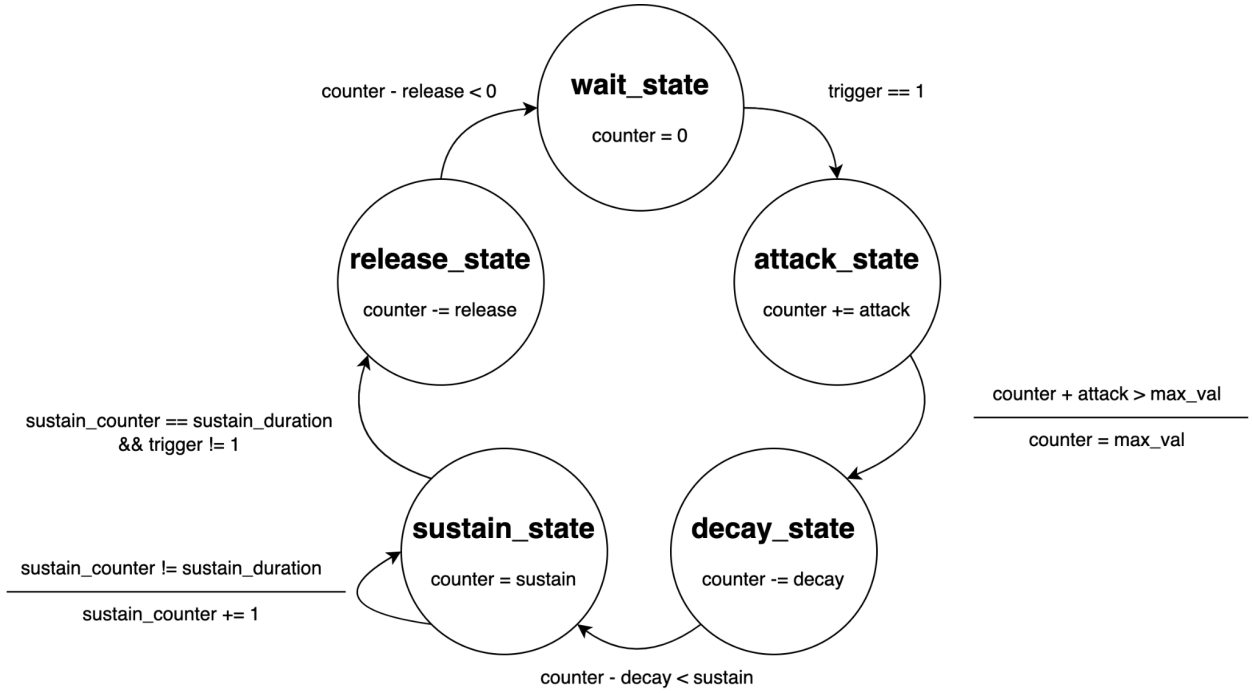

Figure 5: Finite State Machine for the ADSR

The VCA Module

The Voltage Controlled Amplifier (VCA) module is named after its real-world counterpart, even though in this design there is no real “voltage” value. The VCA module takes an input sample and amplifies it based on a “voltage” signal and a volume signal. In this design, the “voltage” is the 16-bit output of the ADSR and the *volume* is a 16-bit, U(0,16) fixed-point parameter (where the decimal point is right in the front); thus the max volume would be all 1s and the min volume would be all 0s.

As shown in Figure 6, the VCA module is composed of two multipliers separated by flip-flops. The flip-flops are used to store the results between clock cycles to allow for a higher max clock frequency. The product of two 16-bit values is a 32-bit value. After each of the multiplier units, the lower 16-bits of the results are discarded. This is done for rounding, as the operands of the multiplier units are U(16,0) and U(0,16) fixed-point, the result would be a U(16,16) fixed-point; thus, the lower 16 bits are the fractional components of the result and can be ignored (since the resulting output sample must be a 16-bit integer).

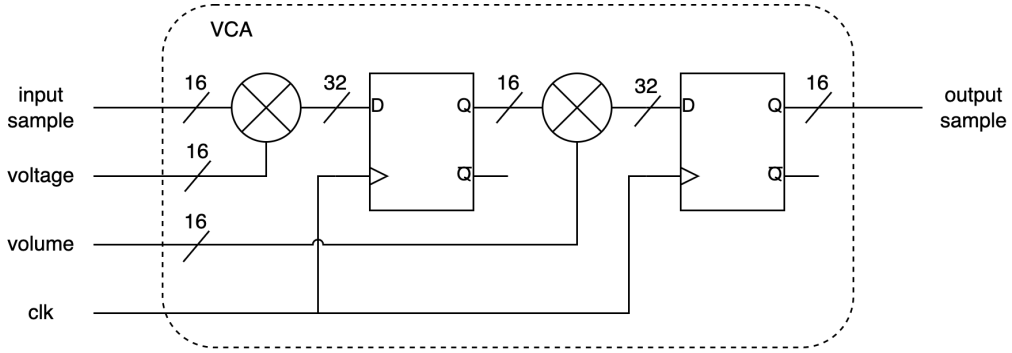

Figure 6: High-Level Diagram of the VCA Module

AudioVoice AXI-Lite Register Interface

An AXI-Lite Slave Register Interface was used to make configuring the different AudioVoice modules easier. It is composed of 8 write-only registers for each of the parameters (explained in the prior sub-sections). These registers, their memory-mapped addresses, and a description of their values are listed in Table 2.

Table 2: AudioVoice AXI-Lite Slave Registers

Offset	Parameter	Description
0x00	*wave_select*	2-bit value selecting the wave of the oscillator. - 2'b00: triangle - 2'b01: sawtooth - 2'b10: square
0x04	*half_period*	16-bit value used to specify the frequency of the output waveform from the oscillator.
0x08	*adsr_attack*	16-bit value used to specify the attack duration of the ADSR.
0x0C	*adsr_decay*	16-bit value used to specify the decay duration of the ADSR.
0x10	*adsr_sustain*	16-bit value used to specify the sustain voltage of the ADSR.
0x14	*adsr_sustain_duration*	16-bit value used to specify the sustain duration of the ADSR.
0x18	*adsr_release*	16-bit value used to specify the release duration of the ADSR.
0x1C	*volume*	16-bit value used to specify the volume of the outputted audio sample.

A MicroBlaze soft processor is expected to be used to set these parameter values during operation of the system.

4.1.2 Mixer

The Mixer module is a simple module that combines two 16-bit audio samples into one sample. This is done by adding the two samples together; however, special care must be taken in the case of an overflow. This can be handled in two ways: Clipping and Averaging. Clipping is when the output is clamped to the max value of all 1s in the case of overflow and Averaging takes the average of the two signals (thus never overflowing). The user can set which mode to set the Mixer in by setting the *mode* parameter to either 0 (for Clipping) or 1 (for Averaging). Figure 7 shows the high level diagram of the Mixer module.

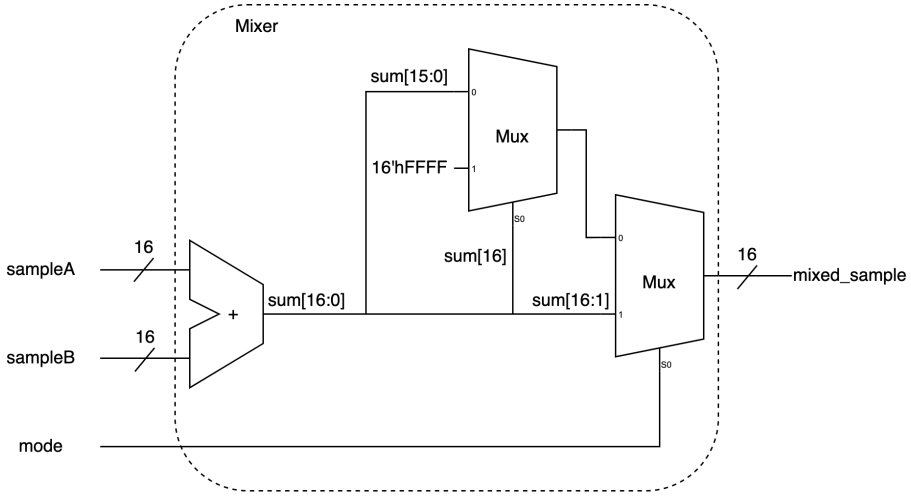

Figure 7: High-Level Diagram of the Mixer Module

To mix more than two samples together, a tree of Mixer units should be used. For the Clipping mode the tree need not be balanced, as shown in Figure 8; however, for the Averaging mode the tree must be balanced as shown in Figure 9. This is because if the mixing tree is unbalanced in the Averaging mode, mixers closer to the final output sample will have a higher priority than mixers farther away; and, as such, all audio sample leaf nodes must be at the same depth.

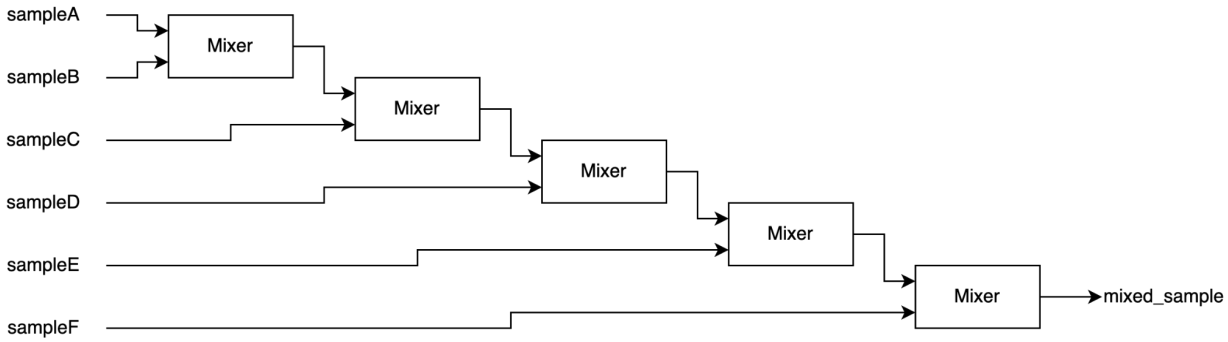

Figure 8: Unbalanced Mixer Tree

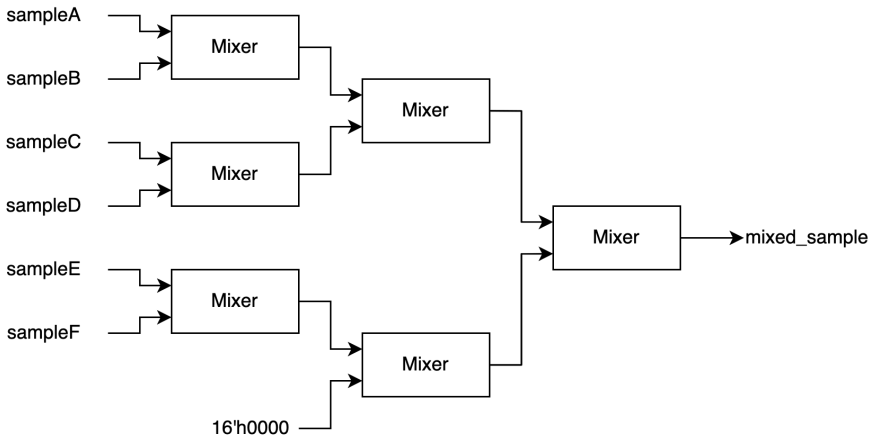

Figure 9: Balanced Mixer Tree

As shown in Figure 9, although the Mixer module closest to the bottom seems redundant, it is effectively cutting the averaging of samples E and F in half; making the overall averaging equal amongst the samples.

4.1.3 AudioSampleToAxiStreamAudio

As further explained in Section 4.2.2, in order to communicate with the Audio Codec the audio samples need to be communicated to the I2S Transmitter IP. As shown in Figure 10, taken directly from the I2S Transmitter documentation, the communication protocol used for this transaction is a special flavor of AXI Stream called AXI Stream Audio or AES3.

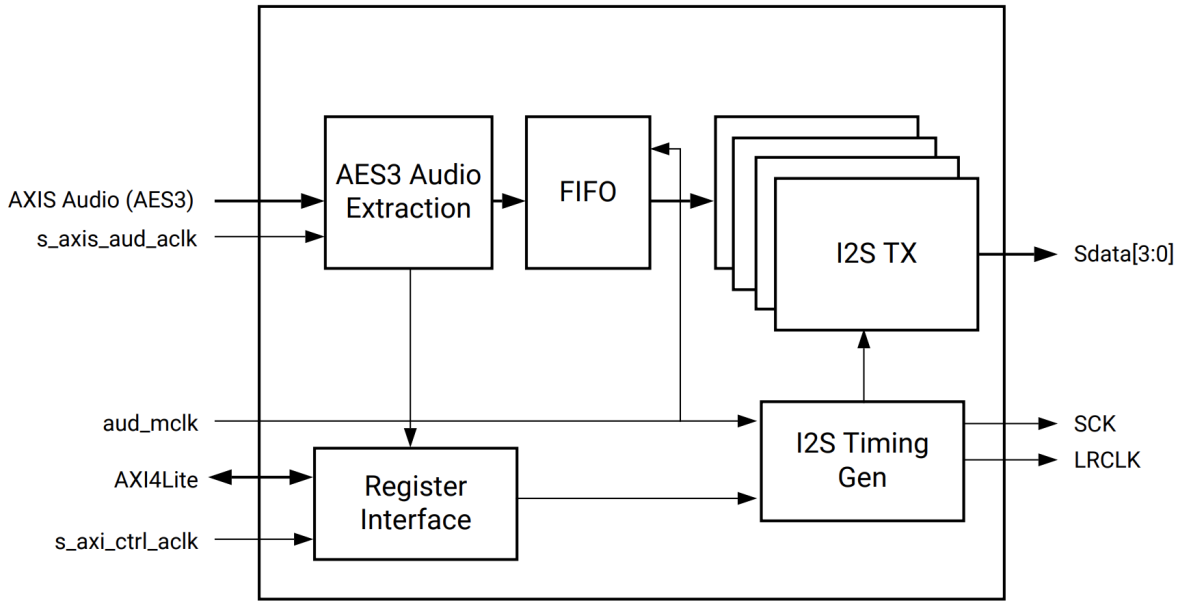

Figure 10: High Level Diagram of the I2S Transmitter [1]

AXI Stream Audio (AES3 Data Format) Protocol

The AXI Stream Audio protocol is based on the AES3 (IEC60958-3) Data Format [1]. Further documentation is provided for another IP which provides more information on this communication protocol [2]. As shown in Figure 11, the protocol uses six signals to transmit different frames of data.

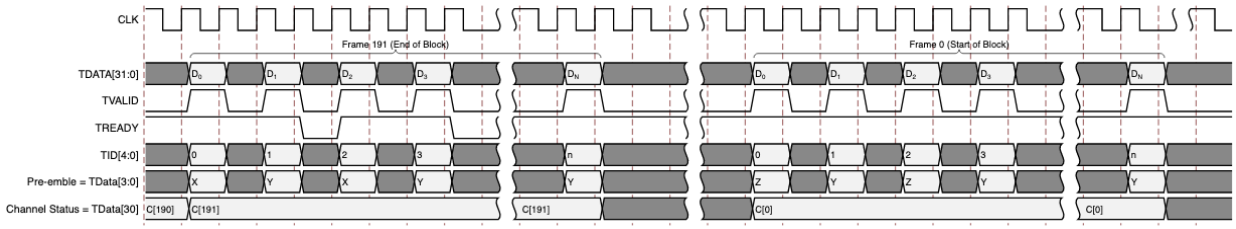

Figure 11: AXI Stream Audio Protocol Timing Diagram [2]

According to the protocol, the sender is expected to send 192 frames before wrapping back to frame ID 0. Each frame contains sub-frames, shown in Figure 12, that contain the audio data for each channel of the audio. For example, since this project only needed two channels, a frame contains two sub-frames which transmit the sample going to channel 1 and the sample going to channel 2.

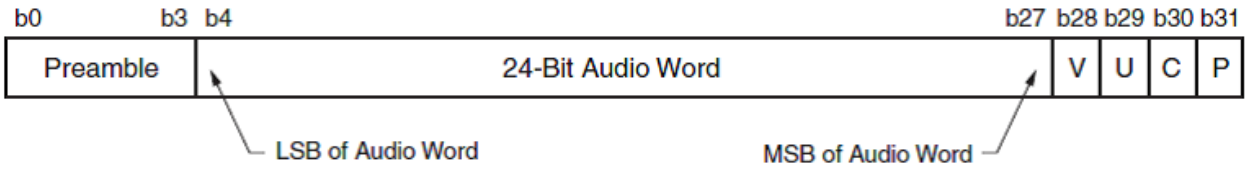

Figure 12: AXI Stream Audio Sub-Frame TDATA Format [2]

The TVALID and TREADY signals are straightforward and match the AXI standard. The TID is also very simple and is just what channel the sub-frame's data is for. As shown in Figure 12, the TDATA signal is where things get complicated. The TDATA signal is 32 bits wide and contains three portions: the Preamble, the Audio Word, and the VUCP.

The Preamble will be one of three values: BSYNC (4'b0001), SF1SYNC (4'b0010), and SF2SYNC (4'b0011). Odd channels always have a SF2SYNC Preamble. Even channels usually have a SF1SYNC unless it is the very first frame (frame 0); in that case, the BSYNC is used. This is shown (somewhat poorly) in Figure 11.

The Audio Word can either be 24 or 16 bits wide. As shown in Figure 12, when the Audio Word is 24-bit it fills the entire space with the least significant bit being located at b4. When the Audio Word is 16 bits (as in this project), the least significant 8 bits (b4 - b11) are padded with zeros and the rest is filled with the 16 bit Audio Word.

VUCP stands for Validity bit, User bit, Channel Status bit, and Parity bit. The Validity and User bits are not explained anywhere in the documentation and, from experimentation, are not used for the I2S Transmitter; thus they are always tied to logic zero. The parity bit is supposed to be even parity over the sub-frame except for the preamble; however, from experimentation, this is also unused by the I2S Transmitter IP and is also tied to logic zero. The Channel Status bit is the most complicated and is the reason why there are 192 frames. The Channel Status is a 192-bit word that is used to communicate information about the audio such as the sample rate, the number of channels, and more [3]. Each bit of this Channel Status word is sent, one at a time, in each sub-frame per frame. This is shown (also somewhat poorly) in Figure 11. For this project, the module uses a preset Channel Status used in an example provided by the I2S Transmitter IP; however, the actual value seems very odd and is unlikely to be transmitting any information; likely this also goes unused for the I2S Transmitter IP.

Building the Module

To build the module to translate the audio samples, generated by the AudioVoice module, into the AXI Stream Audio protocol, the demo code provided by the I2S Transmitter IP was used as a guide [1]. The AXIS Data Generator module was designed to send pre-made samples of audio to the I2S Transmitter. This module tries to send all of its data continuously until it fully fills the FIFO in the I2S Transmitter. This did not work directly for this project as the data is generated just-in-time by the AudioVoice and, to reduce latency, the data cannot be backed up into a FIFO. As such, the demo code was modified such that it would take an *enable* signal, which is a pulse to signify to the module to send a frame to the I2S Transmitter. This *enable* signal was tied to the AudioPulseGen module (explained in Section 4.1.6), which sends a pulse at the sampling frequency.

4.1.4 Sequencer

The Sequencer Module generates a trigger signal based on a given sequence and the tempo (signal from the TempoGenerator module described in Section 4.1.5). Figure 13 shows an example sequence and the resulting trigger signal which can be used in the AudioVoice module to trigger the ADSR module.

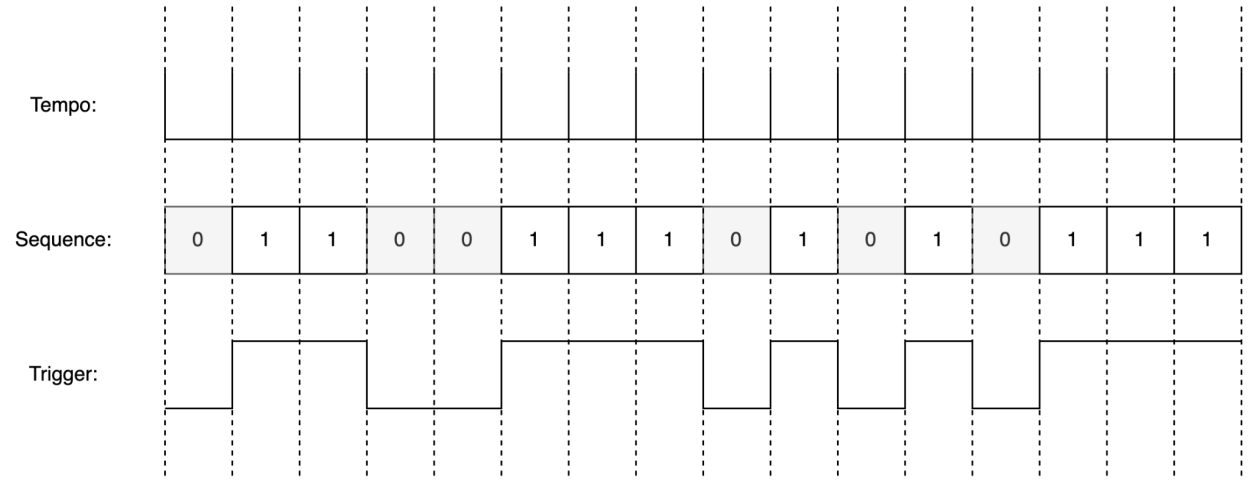

Figure 13: Example Sequence and Trigger Output

The Sequencer module was built using a simple counter that looks-up into the sequence register to produce the trigger. The counter counts on each pulse of the tempo. This allows the sequence to simulate a note being held down by having a sequence of all ones. The length of the sequence can also be set by the user to represent shorter sequences than the number of bits in the register.

The *sequence* and *sequence_length* parameters can be set by the MicroBlaze soft-processor using the AXI-Lite Protocol. The Sequencer IP comes with an AXI-Lite Slave Register Interface with two read-only registers (offset 0x0 for *sequence* and offset 0x4 for *sequence_length*).

4.1.5 TempoGenerator

To produce sound that occurs at regular intervals, a generator is required to keep the tempo. The TempoGenerator module creates a pulse at a regular interval. The rate can be set using the *tempo_rate* parameter. The parameter can be calculated from a desired tempo, in Beats Per Minute (BPM), using Equation 4.

$$tempo\ rate = \frac{60 f_{sample}}{bpm} - 1$$

Equation 4: Calculating the *tempo_rate* From the Desired BPM

The TempoGenerator module is a simple counter that counts up to the *tempo_rate* parameter, and when it is about to roll-over it produces a pulse. Similar to the AudioVoice module, it is interacted with by the MicroBlaze using an AXI-Lite Slave Register interface. It only contains one read-only register for *tempo_rate*, located at the base memory location.

4.1.6 AudioPulseGen

The Audio Clock is a 96 kHz clock generated by the I2S Transmitter IP that is used to send the audio sample rate to the Audio Codec. This clock is much slower than the System Clock, which may run as high as 100 MHz. It is also more efficient and would produce lower latency to do work on the System Clock; however, the generation of each audio sample must be triggered on the rising edge of the Audio Clock. It is necessary to create a pulse train, synchronized to the System Clock, that produces a pulse for exactly one cycle every time the Audio Clock transitions from low to high.

The AudioPulseGen module takes in the Audio Clock and System Clock and generates the necessary pulse train. As shown in Figure 14, the system is composed of two parts: the Synchronizer and the Positive Edge Detector.

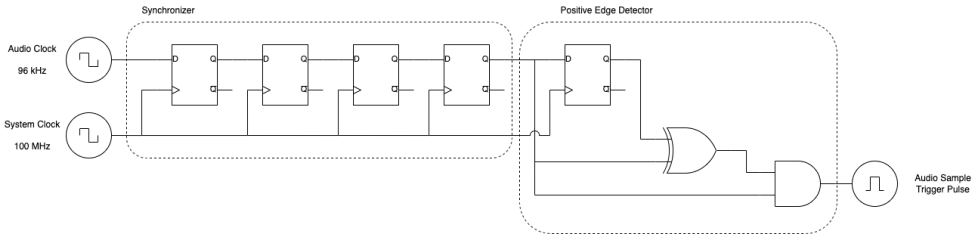

Figure 14: Schematic of the AudioPulseGen Module

The Synchronizer is used to resolve metastability that may occur when capturing the Audio Clock signal using a flip-flop clocked by the System Clock. If the Audio Clock transitions during the hold or setup time of the flip-flop, metastability may occur. The extra flip-flops are used to allow the metastability to stabilize before being used within the system. A Xilinx Parameterized Macro (XPM) was used to build this synchronizer [4]. The 'xpm_cdc_single' macro generates the flip-flops used in the synchronizer and tells the timing analysis within Vivado that synchronization is occurring (to tell the tool that we know there is a timing violation and we are handling it properly). This prevents the tool from flagging that there is a timing violation here, when there is not.

The Positive Edge Detector is a simple circuit that produces a logic 1 if the previous value of Audio Clock was a 0 and the current value is 1. This is done after the synchronizer to ensure that the Audio Clock being sampled is not metastable. If the Audio Clock were to transition from 0 to 1 at a bad time and goes metastable, the value will randomly resolve to either 0 or 1 by the time it exits the synchronizer. If the value resolves to zero, then the positive edge will be detected on the next clock cycle; if the value resolves to one, then the positive edge will be detected and on the next clock cycle the Audio Clock value would still be a one and not produce another pulse. The same is true on the falling edge. As such, it is extraordinarily unlikely that errors or glitches should occur in the case of metastability.

The AudioPulseGen's pulse train is fed into the AudioVoice, Tempo Generator, and the AudioSampleToAxiStreamAudio. This triggers the oscillator to generate the next audio sample, allows the tempo to count time, and sends the previously generated sample to the I2S Transmitter. This means that every sample is generated and then sent to the Audio Codec every sample period.

4.2 Xilinx / Digilent IPs

The following are IPs provided by external sources, not custom designed by this team.

4.2.1 AXI IIC Bus Interface

The AXI IIC Bus Interface IP by Xilinx is used in MusiLinx to allow the MicroBlaze processor to configure the on-board Audio Codec (ADAU1761) at system boot-up. In other words, the MicroBlaze is used to send read and write commands to the Control Registers of the Audio Codec to route and properly gain stage the audio being generated within the FPGA fabric out onto external speakers; however, the MicroBlaze follows the AXI protocol and so the AXI IIC Bus Interface IP was used to convert these AXI messages into the I2C protocol such that the Audio Codec could properly accept the incoming commands. Note, this IP was used simply to send configuration commands for the Audio Codec. The I2S Transmitter, described below, is responsible for the transmission of the audio data.

4.2.2 I2S Transmitter

The I2S Transmitter IP by Xilinx is used in MusiLinx to provide a path for the synthesized audio to be transmitted from the FPGA fabric to the on-board Audio Codec. Specifically, the I2S Transmitter converts AXI Audio Stream compliant data into I2S compliant data.

4.2.3 MicroBlaze

The Embedded MicroBlaze Soft-Processor is used in MusiLinx to configure various parts of the system at boot-up as well as provide system control to the user. Firstly, the MicroBlaze configures the AXI Interrupt Controller, then the I2S Transmitter IP and finally the Audio Codec. The MicroBlaze then enters into a while loop waiting for an interrupt signal from the PS2-AXI Receiver IP which, when received, will cause the MicroBlaze to enter into an Interrupt Service Routine (ISR) which decodes the incoming PS2 messages and routes them to the appropriate parts of the system such as triggering Audio Voices or changing ADSR parameters.

4.2.4 AXI GPIO

The AXI GPIO IP by Xilinx is used in MusiLinx to allow for the MicroBlaze to communicate directly with all 32 Audio Voices inside of the FPGA fabric. Depending on the Mode of the system (monophonic, chord, piano, etc…) the MicroBlaze will write into a 32-bit register inside of the AXI GPIO IP in such a way to trigger the appropriate voices.

4.2.5 PS2-AXI Receiver

The PS2-AXI Receiver IP by Digilent is used in MusiLinx to allow for the keyboard to be used within the system to play the synthesizer and change its modes.

4.2.6 AXI Interrupt Controller

The AXI Interrupt Controller IP by Xilinx is used in MusiLinx to allow the PS2-AXI Receiver IP to interrupt and trigger the appropriate ISR to control the system.

5.0 Description of the Design Tree

The source code for the design is located in a public GitHub repository [5]. The repository includes 5 directories which may be useful to the reader: AudSynth, IP_src, IP_tests, and doc. One will also find a README file including a description of MusiLinx, how to use MusiLinx, the repository structure of the GitHub, the names of the team members, and acknowledgements.

This Final Report as well as a video demonstrating MusiLinx can be found within the doc folder. The reader is highly recommended to watch this video. The IP_src and IP_tests folders contain Verilog code used to make and test the IPs explained in Section 4.1. The AudSynth folder contains the Vivado project files that can be used to run MusiLinx on a Nexys Video Artix-7 FPGA; including the final versions of all the IPs in a folder named ‘repo’.

6.0 Tips and Tricks

The following are Tips and Tricks the team felt would be useful for future projects in ECE532.

6.1 Tips and Tricks for the Technical Aspect of any project

Before attempting to develop a feature from the ground up, do your due diligence and research. Use google scholar to search for academic papers, books and other online resources such as datasheets and reference manuals. There is a very good chance that what you’re looking to develop has already been done and their findings might be useful to you in avoiding common pitfalls and help speed your development cycle. This is also essential if you or any other team member are missing prerequisite knowledge in the specific field of the product you are developing.

Find Demo projects by the manufacturer of the development board. This will clarify how certain features of the board work.

Be humble and ask for help from your team members and from the course staff. Time is limited in this course, it's important to know when and how to ask for help.

Simulate, simulate, simulate (When possible).

6.2 Tips and Tricks for Team Dynamics

Prioritize team building activities early on in the semester in order to make sure that every member feels heard and valued. Remember that four members can accomplish much more than what one or two members can accomplish within the same time frame!

7.0 User guide to Musilinx

Multiple modes were developed for playing Musilinx using a keyboard for a versatile user experience. Mode F1 offers a wide range of notes from C3 to G5, with the placement of each note shown in Figure 15. Mode F2 is designed for chords, making it easier for users to play chords by pressing a single key. This mode provides a variety of chords to be played, as shown in Figure 16. Mode F3 transforms the keyboard into a piano, allowing users to play melodies in a piano-like manner. More details about this mode can be found in Figure 17. Musilinx also includes the option to play pre-registered songs by pressing any key in any order, which can be activated by pressing F4 and F5 to play "Für Elise" and "Gymnopédie" respectively; shown in Figure 18 and Figure 19. Additionally, different keys are designed to control various sound features, such as keys 1, 2, and 3 to adjust the release time, and keys 4, 5, and 6 to adjust the attack time. Lastly, users can change the waveform with keys 7, 8, and 9 to choose between triangle, sawtooth, and square waveform, respectively. A video of MusiLinx, in action, can be found in the GitHub (as mentioned in Section 5.0).

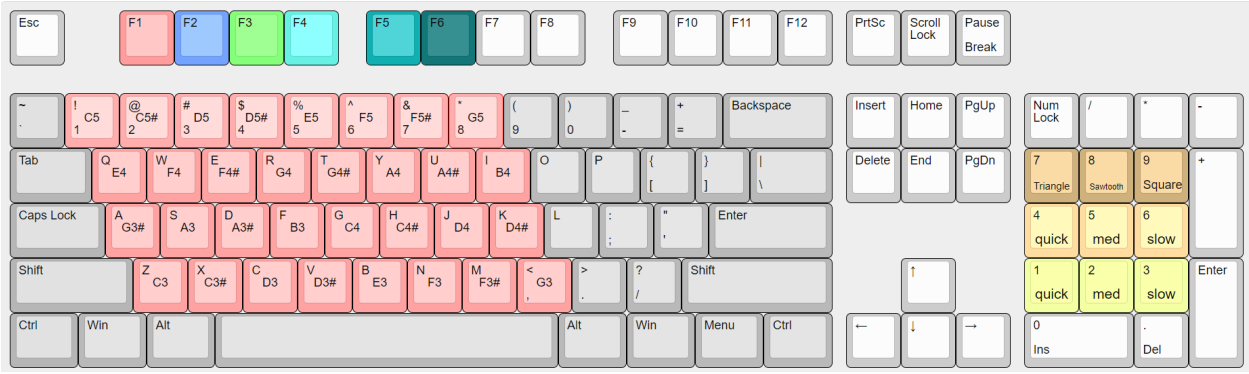

Figure 15: Monophonic Mode (Mode F1)

Figure 16: Chord Mode (Mode F2)

Figure 17: Piano Mode (Mode F3)

Figure 18: “Für Elise” Sequenced Song Mode (Mode F4)

Figure 19: “Gymnopédie” Sequenced Song Mode (Mode F4)

References

[1] "I2S Transmitter and I2S Receiver v1.0." Xilinx, 10-Nov-2021. https://www.xilinx.com/content/dam/xilinx/support/documents/ip_documentation/i2s/v1_0/pg308-i2s.pdf

[2] "UHD SDI Audio LogiCORE IP Product Guide," AMD Adaptive Computing Documentation Portal. [Online]. Available: https://docs.xilinx.com/r/en-US/pg309-v-uhdsdi-audio/AXI4-Stream-Audio-Interface. [Accessed: 11-Apr-2023].

[3] "SPECIFICATION OF THE DIGITAL AUDIO INTERFACE (The AES/EBU interface)." European Broadcasting Union, Geneva, Switzerland, 2004. https://tech.ebu.ch/docs/tech/tech3250.pdf

[4] "Versal Architecture AI Core Series Libraries Guide," AMD Adaptive Computing Documentation Portal. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1353-versal-architecture-ai-libraries/XPM_CDC_PULSE. [Accessed: 11-Apr-2023].

[5] https://github.com/AlexandreSinger/ECE532-Project

*Keyboard layouts were designed using Keyboard Layout Editor website http://www.keyboard-layout-editor.com/#/