conFFTi: An FPGA Music Synthesizer

Authors: Hongrun Zhou, Jiuling Zhou, Michelle Chang
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—The system is a digital music synthesizer that accepts real-time input from a MIDI keyboard, processes the signals via an FPGA, and generates output through the audio CODEC. User input is given through controls on the MIDI keyboard, and the effects of the modulations are able to be produced with a very short latency that is undetectable to the human ear. The system is capable of generating sounds from basic waveforms, modulating its ADSR envelope, producing an arpeggiation, and applying a unison.**

***Index Terms*—Audio, FPGA, MIDI, Music synthesizer**

1 INTRODUCTION

conFFTi is a FPGA-based digital hardware synthesizer that is capable of providing the user with an intuitive musical composition experience. We chose to use FPGA for its low latency, configurability, and portability—all three of which are vital characteristics for realistic and professional uses of synthesizers. Our approach is advantageous compared to other musical synthesizers on the market for its ability to produce results at a low latency of less than 10 milliseconds, its reasonable price point, as well as its portability. The output produced by our product is compliant with the industry standard for audio signal processing, with a frequency of 44.1kHz and formatted as a 24-bit, single channel signal.

Additionally, conFFTi is designed with an emphasis on assisting with the musical composition process, with the implementation of the arpeggiator and the unison effect. With the arpeggiator, the user can input a set of notes to generate a melody with custom tempo, pattern and rhythm. When not operating under the arpeggiator mode, the system supports a 4-note polyphony, giving the user freedom in their choice of input. The unison effect is a simple yet powerful way for the audio output to sound more saturated. Other capabilities of our synthesizer include modulations of the ADSR envelope and waveform oscillators of 4 types: sine, square, sawtooth, and triangle; a set of these effects will be implemented in each of our 4 audio processing pipelines. With the aforementioned features, conFFTi is an effective and intuitive, yet affordable and portable, musical synthesizer.

2 DESIGN REQUIREMENTS

2.1 Output quality: 44.1kHz, 24-bit, single channel output

We will be providing a high fidelity audio output. The industrial standard CD quality audio has a sample frequency of 44.1kHz and a bit depth of 16 bits. The 44.1kHz sample frequency is just over the Nyquist frequency for the uppermost limit of human hearing at 22kHz, which means the 44.1kHz sample frequency is able to capture the full range of frequencies that human can hear without aliasing. As for the bit depth, we decided to provide a higher resolution of 24 bits to give a higher quality sound with a lower quantization error.

2.2 Latency: less than 10ms from MIDI input to audio output

One of our goals is to bring low latency to a configurable and expandable design. Modern hardware synthesizers typically have latency around 3ms while software synthesizers have around 12 - 24ms [8]. We are setting the audio latency to 10ms which is the same as what similar past capstone projects have [5, 6].

However, with our preliminary calculation, the latency of UART for each MIDI message is 3bytes/msg $\times$ (8 + 2)bits/byte / 31250baud = 0.96ms. All the other digital modules will have either combinational or single-cycle design. Therefore, the theoretical latency of out system will be around 1ms.

Human ear is about to distinct sounds that are 30ms apart. [1] Our audio latency requirements way below this physical limit. However, as a music production tool, lower latency is always better.

To measure the latency of our system, we will use a oscilloscope to capture the MIDI input signal and the audio line out signal and compare the time difference of the signal starts for a note press event.

2.3 Frequency distortion: less than 5%

We will measure the frequency distortion by applying FFT to output waveform by FPGA and calculate the percentage of distortion of each frequency, as well as the total distortion. This ensures that we are producing the desired amount of harmonics for each oscillator generated waveform and evaluates the quality of our waveform synthesis technique.

2.4 Pitch deviation: less than 1%

As a music production instrument, it is important to produce accurate sounds. We would like our user to perceive the minimum level of pitch deviation on their end. To measure this quantitatively, we will use a commercial tuner to detect the pitch of the generated sound of each note and achieve a less than 1% deviation.

2.5 Choice of waveform: sine, square, sawtooth and triangle

Each of the audio processing pipelines will be able to generate sine, square, sawtooth and triangle waveforms. These four waveforms are fundamental to digital synthesis. The user is able to choose the waveform with the pads on the keyboard.

2.6 Audio effects: unison detune and ADSR envelope

To support generation of more interesting sounds, we support unison detune to each note. The unison detune effect augments the signal playing with multiple slightly out of phase and detuned version of the signal to produce a fuller sound, like the violin section in an orchestra.

The ADSR envelope specifies the amplitude profile over time of each note. The user is able to configure the attack time, decay time, sustain level and release time to achieve a wide range of sounds.

2.7 Polyphony: 4 notes

We think it is crucial for us to support harmony or chords as they are essential to a practical music production environment. As we are developing a proof-of-concept product, we decided that a 4-note polyphony support is sufficient for demonstration purposes. A 4-note polyphony support already requires a development of a polyphony control unit that can be easily expanded to support more notes if desired. Since the number of audio processing pipelines would need to match the number of notes played simultaneously, we decide to cap this number at 4 so that we will have enough logical elements and RAM on FPGA to support all the functionalities.

In practice, 4-note polyphony is able to support most chords as well.

2.8 Arpeggiator effects

The arpeggiator cycles through a series of notes that the user plays to some tempo, pattern and rhythm.

The user will be able to configure the arpeggiator tempo, mode, rate and rhythm. Tempo specifies the pace the notes will be playing, ranging from 40bpm to 240 bpm.

The mode specifies the order in which the notes are replayed, including Up (rising in pitch), Down (descending in pitch), Up/Down (rising in pitch followed by descending in pitch), Played, Random, and Chord (up to 4 notes).

The rate specifies the speed of the arpeggiated notes using common musical note values: quarter (1/4), eighth (1/8), sixteenth (1/16) and thirty-second (1/32) notes. Additionally, user can turn the arpeggio notes into quarter, eighth, sixteenth and thirty-second note triplets by using the Triplet function.

Lastly, the custom rhythm feature adds musical rests to the arpeggio's pattern, allowing for greater variations in the arpeggios. We support three rhythmic patterns (note only, note - rest - note, note - rest - rest - note) and a random pattern where each step has a 50% chance of being either a note or a rest.

3 ARCHITECTURE OVERVIEW

conFFTi consists of just two pieces of hardware: a MIDI keyboard and an FPGA board. To connect the two components, a MIDI breakout board is used to interface the keyboard to the FPGA board.

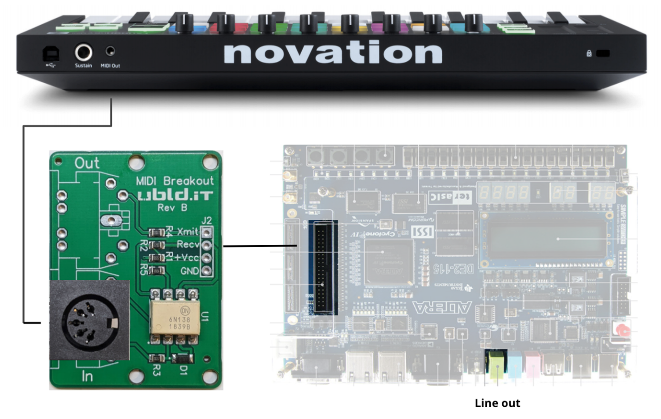


Figure 1: Hardware connections

The MIDI keyboard is in charge of taking in musical note inputs from the users and providing a parameter control user interface. The keyboard of our choice, Launchkey MINI Mk3 MIDI keyboard, provides a piano roll of two octaves with 25 notes in total, which gives the user moderate freedom in creating musical melodies. In addition to the piano roll, it also provides 16 drum pads, 10 buttons and 8 rotary knobs, which could be programmed to control various parameters of the music synthesizer and arpeggiator, e.g. ADSR envelopes and detune.

The DE2-115 Cyclone IV FPGA acts as a hardware platform for digital signals processing. The FPGA is programmed by SystemVerilog scripts which generates synthetic musical sounds based on the music notes played by the user on the MIDI keyboard. In order to perform this synthesis, the workflow on the FPGA is broken down into four stages, as explained in brief in the below paragraphs as well as in detail in section V:

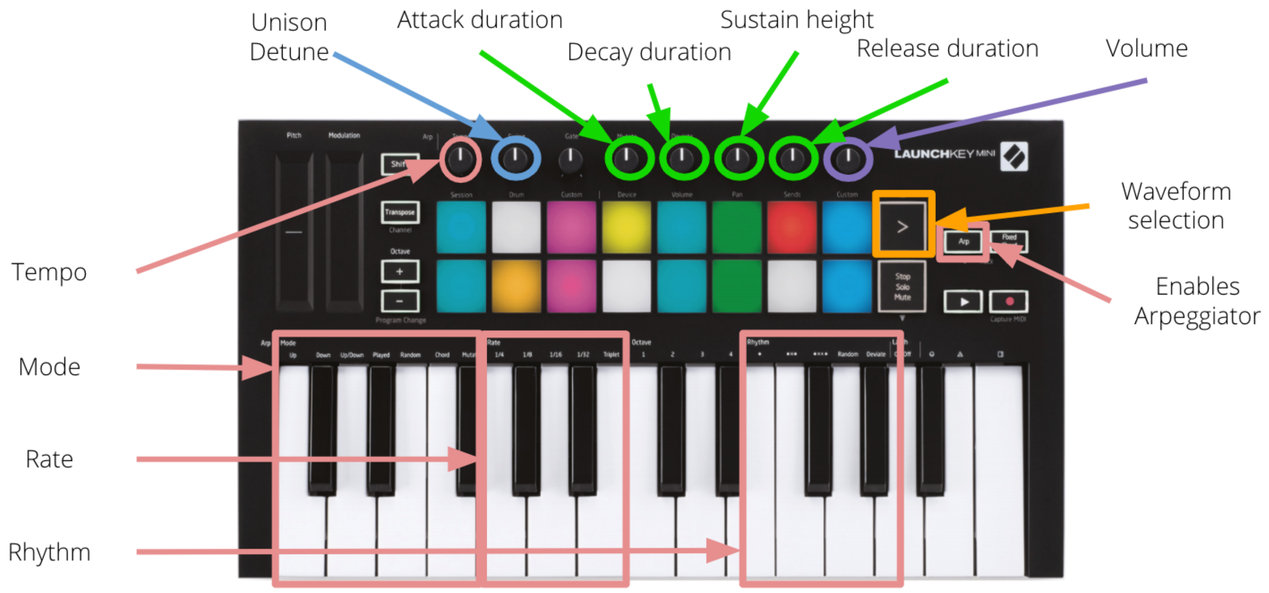


Figure 2: MIDI keyboard user interface design

The first stage is a MIDI decoder. This stage takes in MIDI control signals inputs from the GPIO pin via UART, parse and aggregate that information into a MIDI event object that the subsequent system components are able to interpret. These events could be a changing value of a turning knob, a hit on a particular drum pad, or a key-press on the piano roll. If the MIDI event is a musical note, it will be passed down to the following stages. Otherwise, the MIDI event indicates a change in the state of the system, e.g. a mode change, a parameter adjustment, in which case the global state of this system will be updated in this stage. A datapath and an FSM are designed for this stage to handle all the different possible MIDI events accordingly.

The second stage is the dispatcher stage, where MIDI events are routed to four polyphony pipelines. As illustrated in Figure 3, our polyphony system operates in either polyphony control mode or the arpeggiator mode. The polyphony allows the user to simultaneously play four notes on the piano roll and hear the synthesized sounds of their inputs up to four notes at a time. Whereas in the arpeggiator mode, the system loops over a set of notes given and recorded by the user. In both modes, the musical note inputs are organized in a queue data structure of size 4, which dispatches the stored notes on a first come first serve basis. The dispatcher stage also keeps track of when the notes are released to clear the queue structure in real time. In the case where all 4 pipelines are occupied, a fifth input will not be accepted into the program. In this way, the dispatcher stages handles simultaneous inputs in an organized and predictable manner.

The third stage is the audio processing stage that happens within each of the pipelines. This stage begins with a waveform oscillator, that takes in the frequency and velocity parameters of each musical note input, generates a square, sine, triangular or sawtooth waveform as requested by the user. Next, a feedback loop is setup for supporting the unison feature, which takes in the original waveform, detunes and phase shifts the waveform several times to produce a richer and fatter sound. Then the aggregated waveform gets passed into the ADSR envelope that tweaks how the volume of the sound should unfold in the attack, decay, sustain and release stages respectively throughout the duration of the note. According to how the user sets up the values on the attack, decay, sustain and release buttons on the keyboard, the amplitude of the waveform will be scaled differently. The ADSR envelope will give a more nuanced tone to the sound of the output.

The fourth and final stage is the mixer and the audio CODEC. The mixer takes in all the outputs of the four pipelines, normalizes the final waveform and passes the result to the Audio CODEC of the FPGA board. To hear the result, the user can simply plug a wired headphone or a speaker to the Line Out of the FPGA board.

4 DESIGN TRADE STUDIES

4.1 FPGA Choice

From the experience of similar past capstone projects [5, 6], we learned that to support 4 note-polyphony, the system requires a large number of logical elements. Also, to support arpeggiator and unison detune, the system needs to store multiple notes for an extended period of time, which means the FPGA needs to provide a considerable amount of storage space.

Due to the COVID-19 pandemic, each of the three members is located in different parts of the United States. Therefore, in order for us to develop the system in parallel, each of us require an FPGA. From ECE inventory,

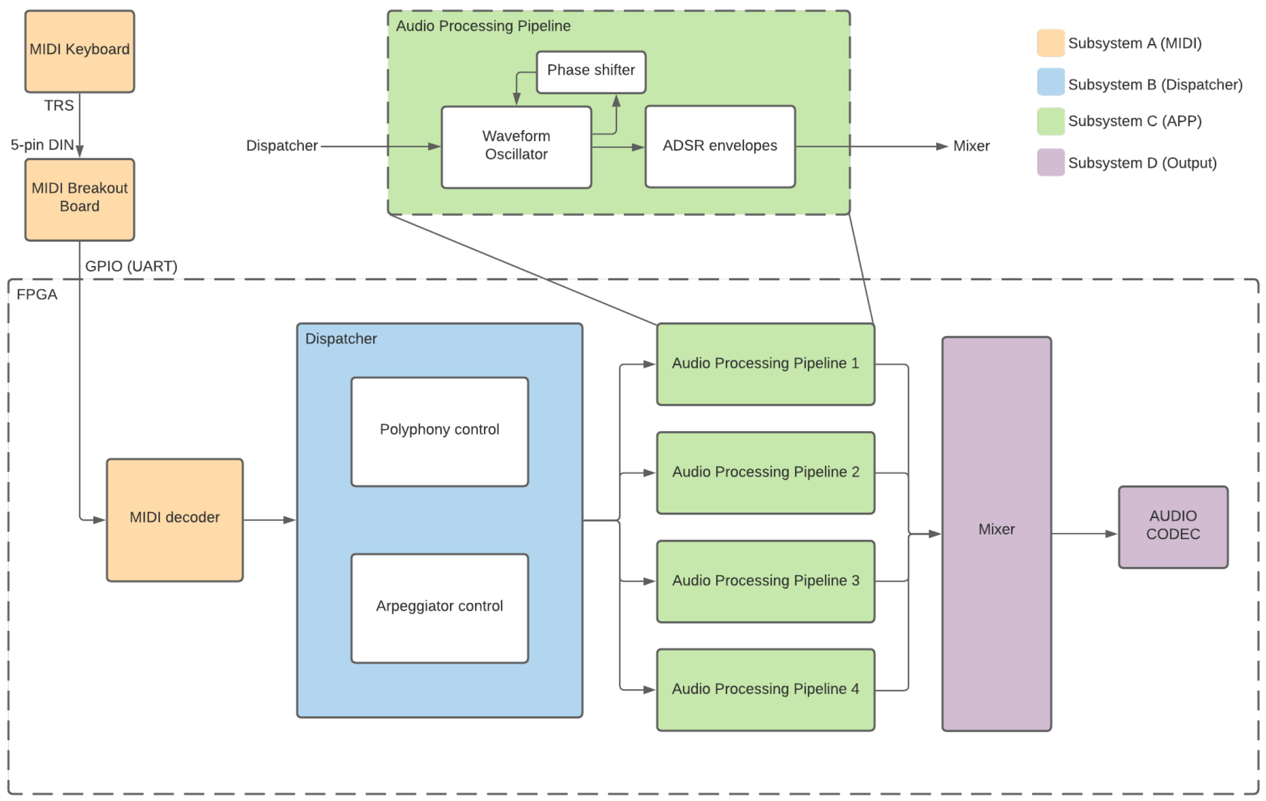


Figure 3: Block diagram

the FPGA models that potentially have enough logical elements and memory, and with large number of units available are limited to DE2-115 (used by 18-240) and DE0-CV (used by 18-341). The DE2-115 board is a larger board with 114,480 logic elements, 3,888kbits embedded memory, and 128MB SDRAM [3]. On the other hand, the DE0-CV has only 40,000 logical elements, 3,080kbits embedded memory, and 64MB SDRAM [4]. To avoid switching board mid-development which will cause a huge delay and shipping hassle, we decided to go with the large DE2-115 board.

Also, since all three members have only taken 18-240, we are most familiar with the DE2-115 board. Having a familiar board saves us some ramp up time and give us the opportunity to reuse the configurations for the board (like pin assignments).

4.2 MIDI Controller Selection

There are many mini MIDI controllers on the market. There are two kinds of interfaces that these controllers offer, the USB MIDI interface or the MIDI 5-pin DIN interface. Most controllers in a low price range comes with the USB output, but by choosing such a controller, we need to either implement a USB controller or buy an external USB device controller to parse the MIDI signals, which is complex and expensive. In avoidance of this extra work, we decided to work with a keyboard that offers the legacy DIN output and buy a MIDI breakout board to interface with the keyboard in the easiest possible fashion.Another requirement for the MIDI controller is that it needs to come with a good number of knobs and faders. This is so that we can use these existing real estates to provide an easy and intuitive user interface for manipulating various parameters on the synthesizer. We did our research and found that the cheapest controller that fits both two requirements is the Launchkey MINI Mk3 MIDI keyboard. Even though this controller comes at a higher price of $109.99, we think the choice is reasonable because it reduces the complexity of development and is affordable within our $600.0 budget.

4.3 Audio output

For audio output, we have two choices – use the 24-bit audio CODEC on the DE2-115 board or build a DAC circuit that will output a 16-bit audio together with an audio op amp. We initially wanted to go with a DAC circuit, but on second thought, we decided using the audio CODEC might be a better choice. Even though the audio CODEC can be more complicated to interface to with SystemVerilog code, it reduces the number of external peripherals of the entire system to the minimum. This can be a great plus for the user experience, as it is a lot easier to just plug in a set of headphones to the FPGA board than having to hook up amplifiers to the external DAC circuit.

4.4 Wavetable synthesis

We initially considered designing conFFTi as a wavetable synthesizer. We decided not to proceed with this design since there were previous groups that designed musical synthesizers of similar synthesizing techniques. Instead, we will invest our time into more unique features such as the arpeggiator and the unison. In order to show a proof of concept, we are focusing on the actual functionality of the synthesizer.

5 SYSTEM DESCRIPTION

5.1 Subsystem A – MIDI Interface And Decoder

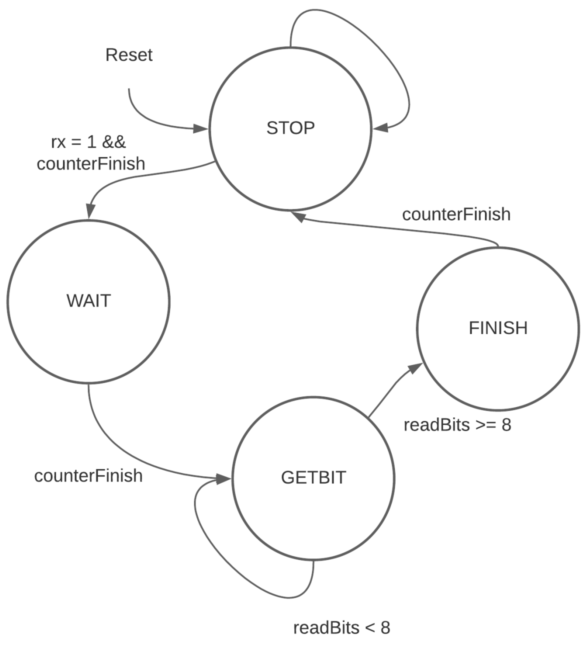


Figure 4: Deserializer FSM

The MIDI signal is received from one of the GPIO pins on the FPGA board at a baud rate of 31,250 on a 50MHz clock. A clock counter based mechanism is in place to synchronize with the lower baud rate and sample the bitstream. At this sample rate, the serial MIDI The *deserializer* module takes in the serial signal via *rx* and collects MIDI bytes according to the UART protocol. It uses an FSM (Fig. 4) to keep track of the position in a MIDI byte and its surrounding start and stop bits so as to aggregate the serial data into 8-bit chunks that will be consumed by the subsequent *decoder* module.

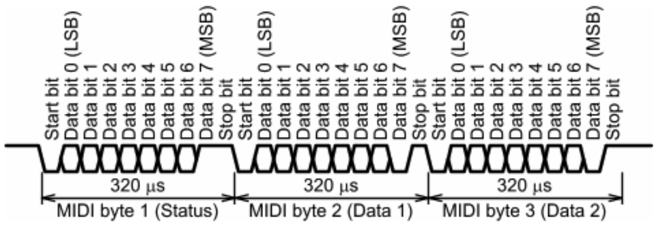


Figure 5: Example MIDI event consisting of three MIDI bytes

A MIDI event is described by three MIDI bytes in a row from the *deserializer* as shown in Fig. 5. The first byte is a status byte that tells whether the event is a NOTE ON, NOTE OFF or a PARAMETER CHANGE. The second byte contains information on the value of MIDI note being played or which knob or fader is being modified. The third and final byte tells the new velocity or the new value of the knob or fader being modified. The *decoder* module implements a FSM (Fig. 6) to collect all three data bytes and aggregate this information into a MIDI event packet to send to the *dispatcher* module. In addition, the *decoder* also converts MIDI note number to its corresponding frequency from a lookup table. This frequency lookup result will also be included in the MIDI event packet that gets sent to the *dispatcher*.

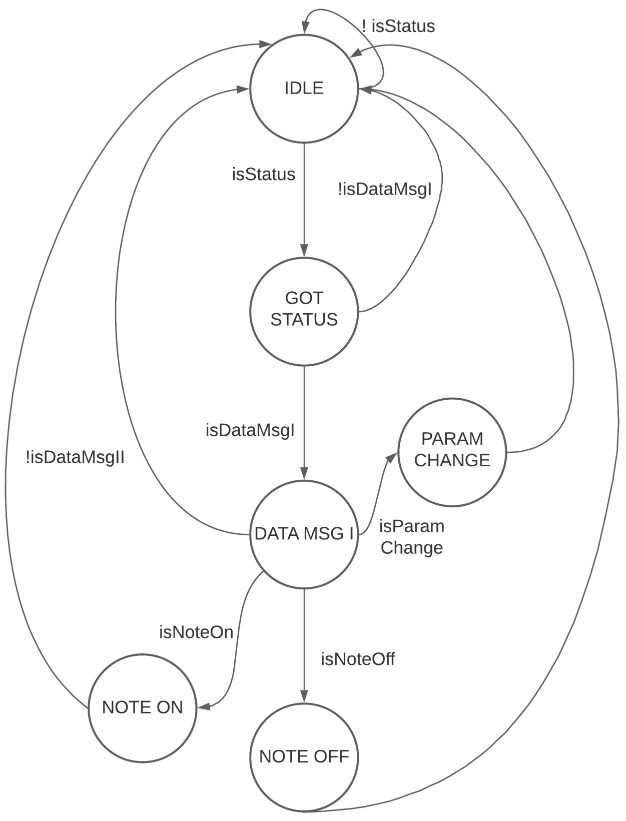


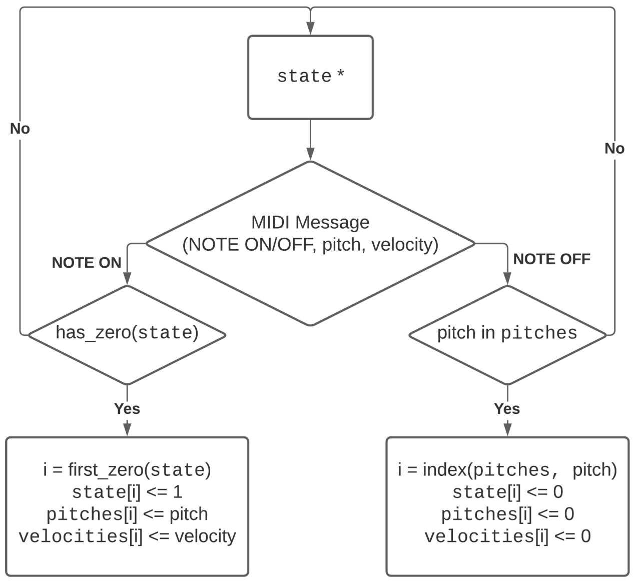
Figure 6: MIDI decoder FSM

5.2 Subsystem B – Dispatcher

This subsystem takes the MIDI events decoded by the Decoder, and dispatches notes to each audio processing pipelines with the correct timing according to the mode of operation.

The dispatcher selects between the polyphony mode and the arpeggiator mode based to the event of arpeggiator button press. If the system is in polyphony mode, the NOTE ON/OFF events are passed in to the polyphony control module. If the system is in arpeggiator mode, in addition to the NOTE ON/OFF events, the arpeggiator parameters are passed in to the arpeggiator control module. For audio effects parameters, the dispatcher passes on the parameter values to the audio processing pipeline.

5.2.1 Polyphony



*`state` is a 4 bit value. The i-th bit of `state` encodes the occupancy status of the i-th pipeline.

Figure 7: Polyphony control ASM

The polyphony control module is described by the algorithmic state machine (ASM) in Fig. 7. It implements a *first note priority* policy for notes that are not released, where the the newly pressed notes are ignored once the maximum polyphony is reached. For notes that are already in released stage (i.e. the key has been release but the volume is slowly fading), a new note would take priority. In other words, once 4 notes has been pressed, any additional notes pressed are ignored until any of the first 4 are released.

On NOTE ON event, the pitch and the velocity of the notes is stored in `pitches` and `velocity`, respectively. Then the polyphony control signals the respective Audio Processing Pipeline to start producing the note.

On NOTE OFF event, the release signal of that note is sent to its Audio Processing Pipeline and the note information is removed.

5.2.2 Arpeggiator

The arpeggiator generates start signal and release signal for the Audio Processing Pipeline on the specified parameters.

For example, if notes C, E, G (do, mi, so) are playing, the *tempo* is set to 60bpm, the *mode* is set to Up, the *rates* are set to 1/4 and the *rhythm* is set to note - rest - note, then the arpeggiator generates the following signal pattern (empty slots are for rests) in Fig. 8.

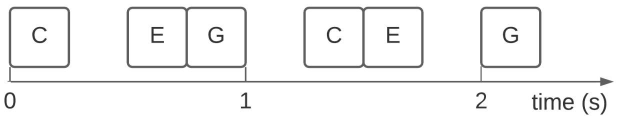


Figure 8: Arpeggiator pattern generation

To account for the release time, the release signal is sent to the Audio Processing Pipeline before the next note start is sent. In other words, if t_0 and t_1 are the time when two consecutive notes start and release time is $t_{\text{release}}(\ll t_1 - t_0)$, then the release signal of the first note is sent at $t_1 - t_{\text{release}}$.

5.3 Subsystem C – Audio Processing Pipelines

The system will be consisting of four Audio Processing Pipelines. Each pipeline is comprised of a waveform oscillator and a module controlling the ADSR envelope.

The waveform oscillator is in charge of generating four types of waveforms: sine, sawtooth, square, and triangle. The generation of these waveforms will be accomplished with either combinational (square, sawtooth, and triangle) or with a lookup table (sine). The operation of the module will be based on incrementing an 8-bit value that represents the phase, so it is able to represent angles at 1.4 degree precision. Based on the accumulated phase value, the module will generate amplitude values of the chosen wave type at the specified location. Because of the nature of sine waves, generation of sine waves can be accomplished with look-up tables designed for quarter-wave look-ups[9]: instead of storing amplitude values for every point along the wave, only store values for a fourth of the waveform. This way, sine wave generations are efficiently implemented with some additional logic for index computations.

In order to implement the unison effect, the waveform oscillators will be implemented with a feed-back loop and a phase offsetting module. With these additions, when the user enables unison, the waveform oscillators will output a compounded waveform resulting from adding several copies of a single waveform, each slightly detuned and applied with a phase offset.

The ADSR module (Fig. 9)will be implemented by scaling the amplitude of the incoming waveform, depending on the Attack, Delay, Sustain, and Release values set by the user. This module will be keeping an incrementing counter to keep track of the current time index into the ADSR process. Thus, the computation of this module is consisted of numerical comparisons and multiplications, and so it would not introduce additional latency to the system.

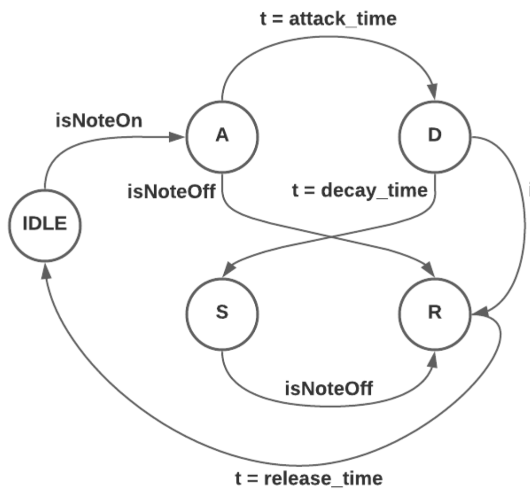


Figure 9: ADSR FSM

5.4 Subsystem D – Audio Mixing and Output

The mixer simply takes in the outputs from the four audio processing pipelines and adds them, normalizes them into a final waveform, which will be passed to the Audio CODEC component on the DE2-115 via SystemVerilog Module. A 24-bit, 44.1kHz sound output will be heard via a wired headphone or a speaker.

6 PROJECT MANAGEMENT

6.1 Schedule

The Gantt chart (Fig. 10) is inserted in appendix B of the report. We have divided our project into 4 main checkpoints. By checkpoint 1, we aim to achieve a minimal viable product that is capable of receiving a single user input, generating a single type of waveform, and producing an audible output. By checkpoint 2, we aim to achieve waveform generation of all 4 types of waves, 4-note polyphony, as well as the final mixer that will combine the signals into the output waveform. By checkpoint 3, we aim to achieve the normal mode effects including the ADSR modulations and the unison, and we will also start with some basic implementation of the arpeggiator. During checkpoint 4, we will collectively focus on implementing the complex features of the arpeggiator. After each checkpoint, we have planned 2 days of integration and verification time. We have also planned 2 weeks of slack time after the checkpoint 4 integration deadline.

6.2 Team Member Responsibilities

Hongrun will be working on the MIDI keyboard interface and the audio CODEC, implementing the ADSR envelope, and verifying the output waveforms by comparing them with our reference MATLAB script results. Jiuling will be working on polyphony control, implementing a random number generator, and setting up the basic functions of the arpeggiation mode. Michelle will be working on setting up the note-to-frequency mappings, the value look-up tables for computing the sine wave, and the unison effect. Collectively, we will work on the advanced arpeggiator modulations, which are the rhythm and the mode of the arpeggiation result. A detailed breakdown of each of our responsibilities over the course of the development process is color-coded in our Gantt chart (Fig. 10).

6.3 Budget

The budget chart is included in the Appendix section of the report (Table 1). Since we are working on our project remotely, we had to each purchase a set of the necessary equipment. The respective quantities for each item are included in the chart. We are borrowing FPGA boards from the lab, so the cost for the FPGA boards are not included in the budget calculation.

6.4 Risk Management

Our current risk mitigation strategy is to have two days dedicated for integration and verification after each checkpoint. Ideally we would be able to make sure everyone is caught up on the latest version of each other's progress, and that we would be able to conduct verification of the code that we would have written. We have also planned two weeks of slack time at the end, before the final presentation date, so that we have time to catch up in case there were any trouble in the implementation process. If we were able to finish our project abiding to the timeline, we also have a few stretch goals that we could work on during the slack time (i.e. frequency modulation and more arpeggiator effects).

7 RELATED WORK

FPGA-based music synthesizer is indeed a popular Capstone project idea, and in order to create something unique from the previous projects, we have studied reports from those projects from previous semesters. In particular, we studied FMPGA[6] from the Fall 2020 semester and Soundcloud[5] from the Spring 2019 semester. These projects gave us valuable insights on the expected risks, a good estimation of workload, and also some tradeoffs that we should consider. Other s that we have studied is the an FPGA Digital Music Synthesizer project by students of other universities [2, 7]. These papers include extensive details on the implementation methods of the synthesizer effects such as frequency modulation and waveform generation.

8 SUMMARY

For our project, we are aiming to implement a FPGA-based music synthesizer with emphasis on effects that can aid musical composition. As conFFTi combines benefits from hardware and software synthesizers, users can enjoy an professional and intuitive experience.

References

[1] *BINAURAL HEARING.* URL: `https://www.sfu.ca/sonic-studio-webdav/handbook/Binaural_Hearing.html`.

[2] Evan Briggs and Sidney Veilleux. *FPGA Digital Music Synthesizer.* Tech. rep. Apr. 2015.

[3] *DE0-CV Board Specification.* URL: `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502&PartNo=2`.

[4] *DE0-CV Board Specification.* URL: `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921&PartNo=2`.

[5] Jens Ertman, Charles Li, and Hailang Liou. "Check Out Our Soundcloud: An FPGA Wavetable Synthesizer". May 2019.

[6] Joseph Finn, Eric Schneider, and Manav Trivedi. "FMPGA: The Frequency Modulating Programmable Gate Array". Dec. 2020.

[7] *Implementing a Sampling Synthesizing Keyboard on an FPGA.* 2007. URL: `http://web.mit.edu/6.111/www/f2005/projects/mmt_Project_Final_Report.pdf`.

[8] Martin Walker. *The Truth About Latency: Part 1.* Sept. 2002. URL: `https://www.soundonsound.com/techniques/truth-about-latency-part-1`.

[9] ZipCPU. *Building a quarter sine-wave lookup table.* Aug. 2017. URL: `https://zipcpu.com/dsp/2017/08/26/quarterwave.html`.

A Budget and Material

Item Name	Quantity	Price
DE2-115 Cyclone IV FPGA	3	N/A
Launchkey MINI Mk3 MIDI keyboard	3	$109.99
MIDI Breakout Board	3	$10.39
SinLoon 5 Pin Din MIDI Plug to 3.5mm TRS Stereo Male Jack Stereo Audio Cable	3	$6.32
Resistor kit (unused)	2	$6.99
DaFuRui Breadboard Jumper Kit	2	$13.99
MAX541 DAC (unused)	3	$19.74
		$481.28

Table 1: Budget chart

Item Name	Quantity	Price
DE2-115 Cyclone IV FPGA	1	N/A
Launchkey MINI Mk3 MIDI keyboard	1	$109.99
MIDI Breakout Board	1	$10.39
SinLoon 5 Pin Din MIDI Plug to 3.5mm TRS Stereo Male Jack Stereo Audio Cable	1	$6.32
Jumper cable	3	N/A
		$127.70

Table 2: Bill of Materials (for one system)

B Gantt Chart

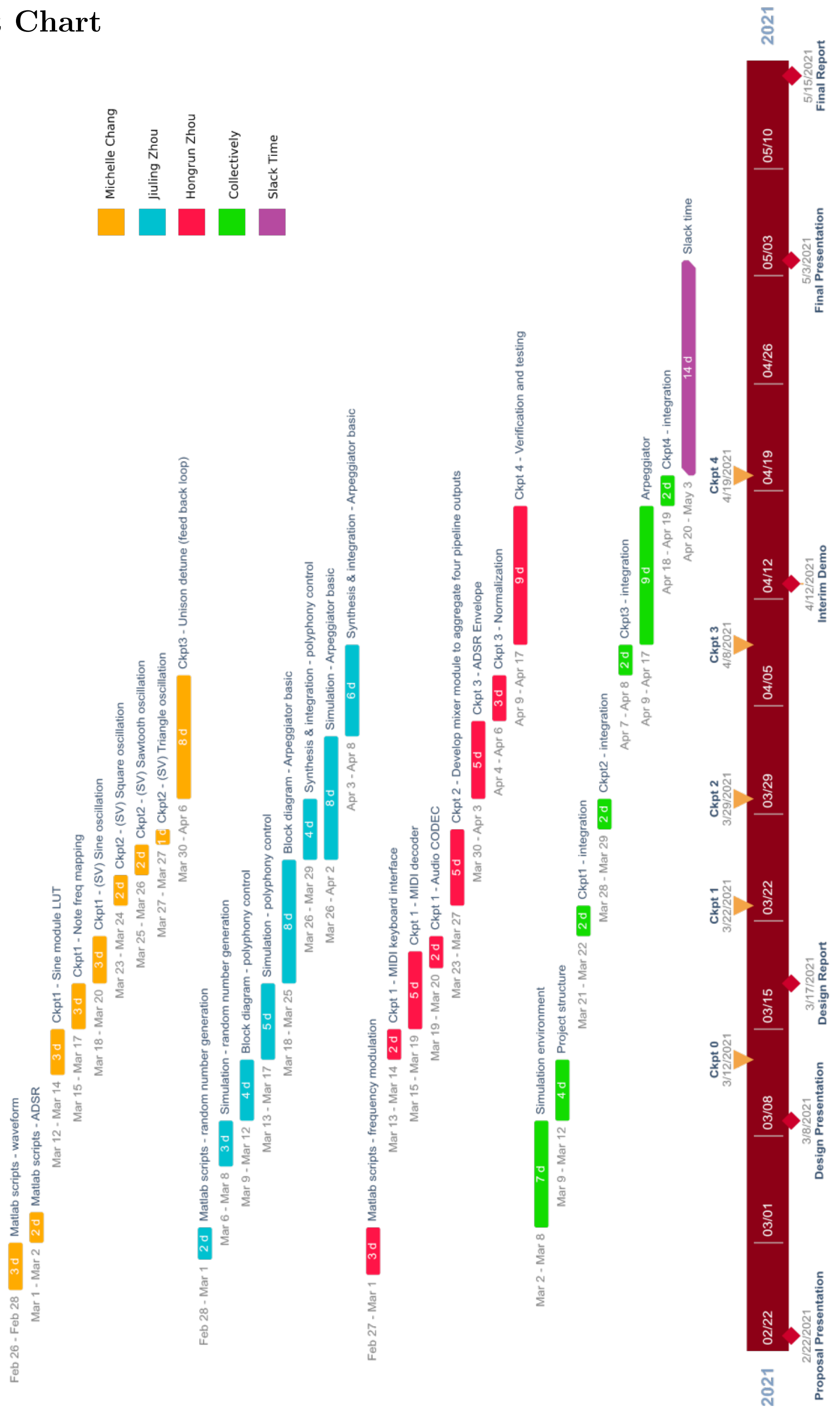


Figure 10: Gantt Chart