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, synthesizes the sound via an FPGA, and outputs the audio through a DAC circuit. User input is given through controls on the MIDI keyboard, and the effects of the modulations are being produced with a very short latency that is undetectable to the human ear. The system is capable of generating sounds with a selection of waveforms, modulating the ADSR envelope, replaying a sequence, and applying unison detune.**

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

1 INTRODUCTION

conFFTi is a FPGA-based digital hardware synthesizer 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 audio produced by our product is compliant with the industry standard for audio signal processing, as a 44.1kHz, 16-bit and dual channel signal.

Additionally, conFFTi is designed with an emphasis on assisting with the musical composition process. The system supports 4-note polyphony, giving the user freedom in their choice of input. The system also provides 8 types of waves with configurable duty cycles that gives an extra flare to the sound. For sound effects, the system provides unison detune and ADSR envelop that alter the sound signature. For sequence generation, the system supports recorder and arpeggiator, both allowing the user to generate a continuous melody with a sequence of notes. 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, 16-bit, dual channel

ConFFTi provides a high fidelity audio output at the industrial standard CD quality with a sample frequency of 44.1kHz and a bit depth of 16 bits. The 44.1kHz sample frequency is over the Nyquist frequency for the uppermost limit of human hearing at 20kHz, which means the 44.1kHz sample frequency is able to capture the full range of frequencies that human can hear without aliasing.

2.2 Latency: <10ms

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 from MIDI input to audio output 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

$$3\text{bytes/msg} \times (8 + 2)\text{bits/byte} \,/\, 31250\text{baud} = 0.96\text{ms}.$$

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 able to detect 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 used an oscilloscope to capture both the MIDI input signal and the DAC output signal, and compared the time difference of the signals for a key press event.

2.3 Signal Shape distortion: <5%

We measured the frequency distortion by comparing the waveforms produced by the synthesizer with the waveforms produced from MATLAB scripts, and calculated the percentage of distortion. This effectively evaluates the quality of our waveform synthesis technique.

2.4 Frequency deviation: <10 cents

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 used a commercial tuner to detect the pitch of the generated sound of each note and achieve a deviation that is less than 10 cents.

2.5 8 choices of waveform with duty cycle

Each of the audio processing pipelines is able to generate three basic waveforms: sine, pulse width, and triangle, along with five waveforms that we sampled from musical instruments. The system allows for duty cycle modulations for all of the waveforms produced. The user can choose the waveform with the switches on the FPGA.

2.6 Effects: unison detune and ADSR

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 versions of the signal to produce a fuller sound, like the violin section in an orchestra. The user can adjust the degree of the detune with a single knob on the keyboard.

The ADSR envelope specifies the amplitude profile over time of each note. ADSR refers to the attack time, decay time, sustain level, and release time for each note played. These quantities can be controlled with four knobs on the keyboard.

2.7 Polyphony: 4 notes

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 on FPGA to support all the functionalities.

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

2.8 Record and Play

With the recorder, the user can press a series of notes at their desired rhythm while holding the record button. Then, when the play button is triggered, the synthesizer will replay the notes at the played rhythm.

2.9 Arpeggiator

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, octave 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, Chord (all notes played on every rhythmic step), and Mutate (configurable with knob to alter the arpeggio in unexpected ways).

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.

The notes can be played across up to 4 octaves. For example, an arpeggio that was C3, E3, and G3 at 1 octave will become C3, E3, G3, C4, E4, and G4 when set to 2 octaves.

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

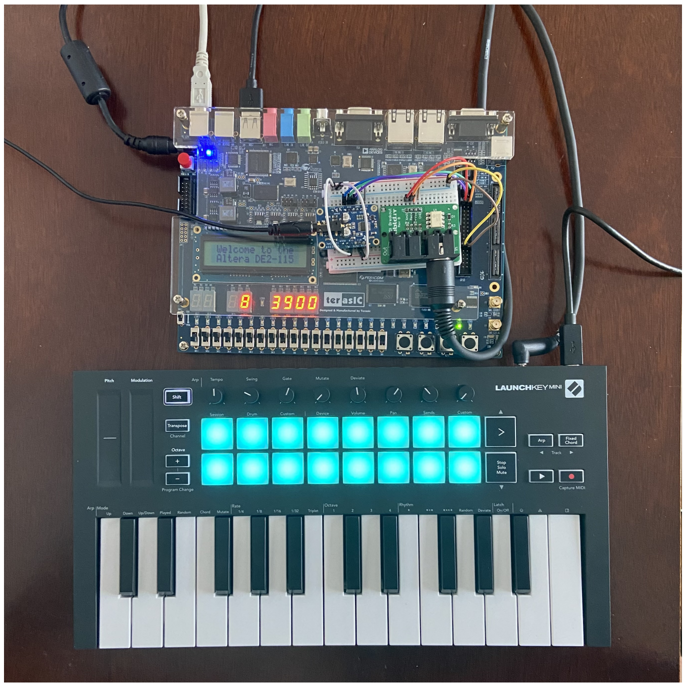


Figure 2: Overall system

conFFTi consists of just two main 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. A DAC circuit is also

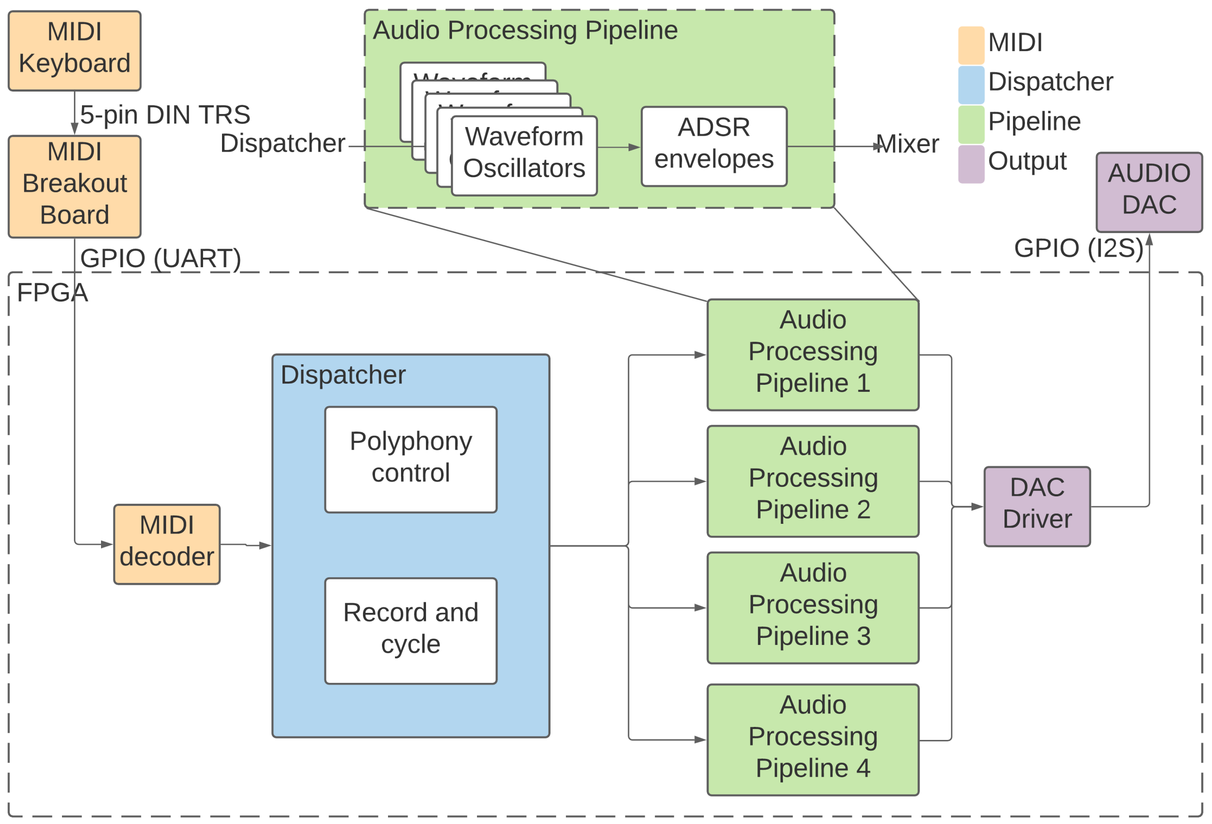


Figure 1: Block diagram

necessary for enabling the connection between our system with an 3.5mm audio jack.

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. See control mappings in Fig. 3.

The DE2-115 Cyclone IV FPGA acts as a hardware platform for digital signals processing. The FPGA is programmed by SystemVerilog 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.

The first stage is a MIDI decoder. This stage takes in MIDI signal inputs from the GPIO pin via UART, parsing and aggregating the 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.

The second stage is the dispatcher stage, where MIDI NOTE ON/OFF events are routed to one of the four audio processing pipelines. As illustrated in Fig. 1, the dispatcher module is consisted of two parts: the polyphony and the recorder. The polyphony allows the user to simultaneously play four notes on the piano roll and hear the synthesized sounds of all four notes at once. The recorder allows the system to loop over a set of notes given by the user. Both features are realized by keeping a short history of the NOTE ON/OFF events.

The third stage is the audio processing stage. This stage begins with waveform oscillators, which generates audio samples over time with the given period and duty cycle. One main oscillator oscillates at the period of the pressed key, and four auxiliary oscillators at slightly longer or shorter periods. The output of the oscillators are combined, and the combined output is scaled with time-varying ADSR envelopes. According to how the user sets up the values on the attack, decay, sustain and release knobs on the keyboard, the amplitude of the waveform will be scaled differently. Lastly, the audio is scaled by the velocity of the key press.

The fourth and final stage is the mixer and the DAC driver. The mixer takes in all the outputs of the four pipelines, normalizes the final waveform and passes the re-

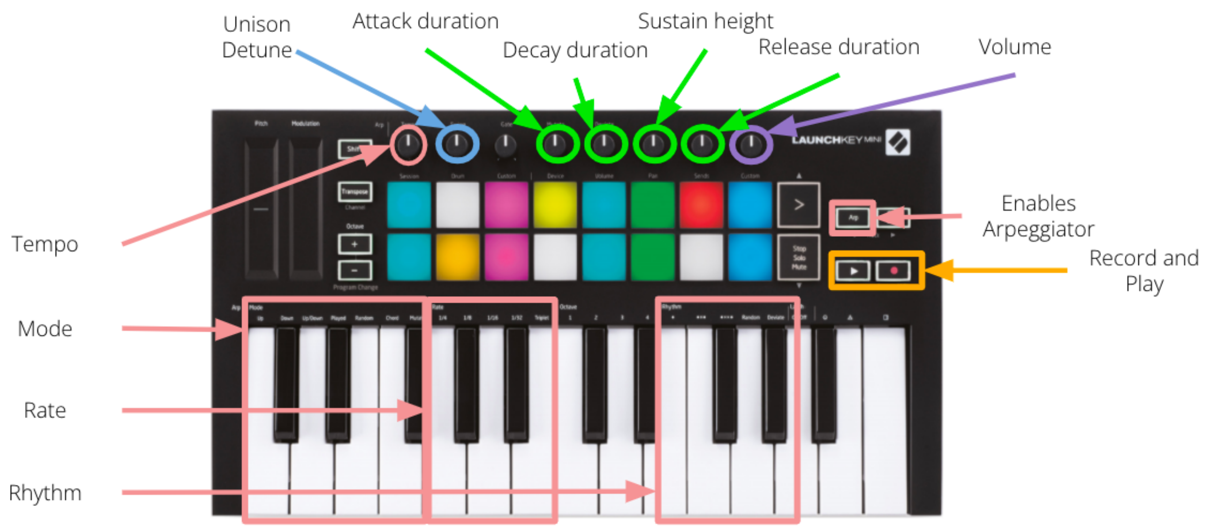


Figure 3: MIDI keyboard user interface design

sult to the DAC driver. The DAC driver encodes the audio in I2S format and outputs to the DAC through GPIO pins. To hear the result, the user can simply plug a wired headphone or a speaker to the audio jack on the DAC circuit.

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. 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 needs an FPGA. From ECE inventory, the FPGA models that potentially have enough logical elements and with large number of units available are limited to DE2-115 and DE0-CV. The DE2-115 board is a larger board with 114,480 logic elements [3], while the DE0-CV has only 40,000 logical elements [4]. To avoid switching board mid-development which will cause a huge delay and shipping hassle, we decided to go with the DE2-115 board.

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 classic DIN output and use 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 - using the audio CODEC on the DE2-115 board or an external a DAC circuit. We initially tried use the on-board CODEC, but the initialization of this device is unintuitive from the FPGA logic side, as oppose to the on-board processor side. It's problematic since our system does not use the on-board processor. Also, by using external circuit, our system is less tied to a specific FPGA board. Therefore, we chose to use an external DAC with audio jack for simplicity and compatibility.

5 SYSTEM DESCRIPTION

5.1 Subsystem A – MIDI Interface And Decoder

A standard MIDI protocol is shown in Fig. 4. The MIDI signal is received from one of the GPIO pins on the FPGA board at a baud rate of 31,250. The bitstream with 8+2 UART format is first deserialized into bytes in the UART driver.

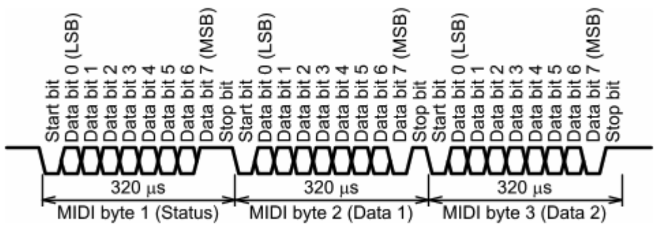


Figure 4: Example 3-byte MIDI event

The supported MIDI events consist of three MIDI bytes. 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 which MIDI note is 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.

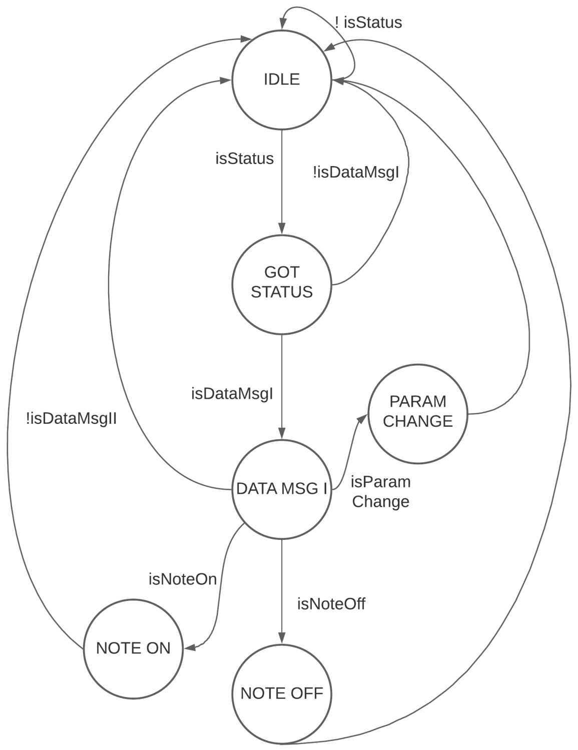


Figure 5: MIDI decoder FSM

5.2 Subsystem B – Dispatcher

This subsystem takes the MIDI events and dispatches notes to each audio processing pipelines. The recorder keeps a history of NOTE ON/OFF events and their timestamps. Upon replay, the recorder sends the event to polyphony module according to the timestamps. The polyphony module them decides how to dispatch the notes to the pipelines.

5.2.1 Recorder

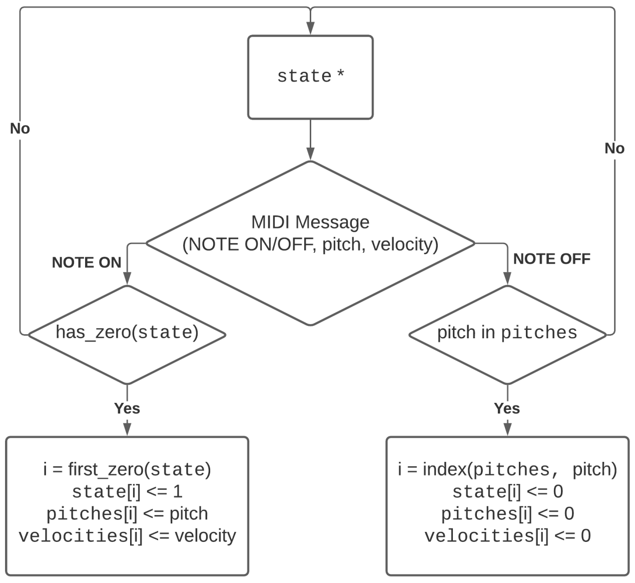
In recording mode, the recorder module uses a counter to keep time. Upon receiving a NOTE ON/OFF event, the event is pushed into a queue, along with the current counter value as its timestamp. The timestamp when the record button is release is also pushed into the queue.

When replaying the events, we keep track of the index of the next event. The counter is incremented at each tick. When the counter value matches the timestamp of the next event, the event is sent to the polyphony module. Then the index of the next event is incremented. The counter resets when it reaches the last timestamp, i.e. the timestamp when the record button is release.

5.2.2 Polyphony

The polyphony control module is described by the algorithmic state machine (ASM) in Fig. 6. 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 pipeline to start producing the note. On NOTE OFF event, the release signal of that note is sent to its pipeline and the note is marked as off.



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

Figure 6: Polyphony control ASM

5.3 Subsystem C – Audio Processing Pipelines

The system has four audio processing pipelines. Each pipeline is comprised of five waveform oscillators and a module controlling the ADSR envelope.

The waveform oscillators are in charge of generating waveforms. The module is given the period and duty cycle as a input. The generation of these waveforms is accomplished by a mix of direct computation and wavetables. The audio is generated at 400kHz. A 16-bit phase counter is incremented at each generation tick. The period for the note being generated in number of generation cycles is looked up in a period table. The number of generation cycles of the FRONT and BACK segments is calculated by multiplying the period with the duty cycle. A percentage is calculated by phase / segment length, and this percentage is used to either calculate the current amplitude, or to index the wavetable for the current segment.

The unison detune effect is accomplished by adding four auxiliary oscillators to each pipeline. We are generating linear detunes, meaning the frequency of the auxiliary oscillators are evenly spaced out on both sides of the main oscillator. To achieve this, we approximate the detune period shifts by period × period × detune parameter value. The auxiliary oscillators then generates sounds at period - 2 × period shift, period - period shift, period + period shift, and period + 2 × period shift.

The ADSR module (Fig. 7) generates a scaling envelope depending on the Attack, Delay, Sustain, and Release values set by the user. This module uses a counter to keep track of the current time into the ADSR process and calculates the scaling percentage. The envelope is multiplied with the detuned audio.

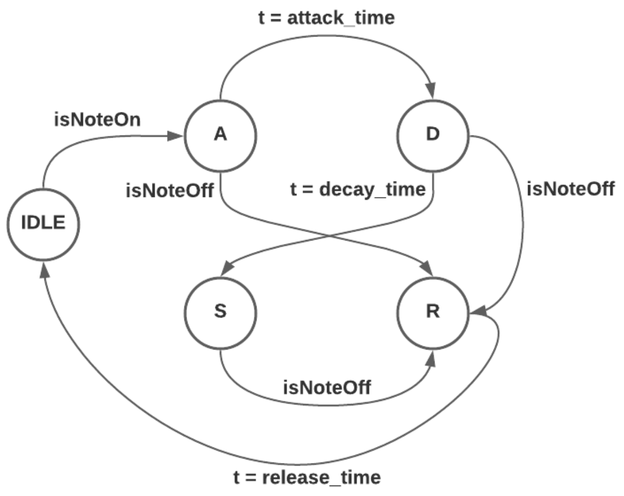


Figure 7: ADSR FSM

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 DAC driver.

The DAC driver samples the 400kHz audio down to 44.1kHz and encode it into I2S format before sending to the DAC through GPIO pins.

Table 1: Pitch deviation across all octaves

Note	C0	C1	C2	C3	C4	C5	C6	C7	C8
wavelength (us)	60935	30625	15300	7631	3843	1903	962	469	232
frequency (Hz)	16.41	32.65	95.36	131.04	260.16	525.37	1039.00	2129.10	4309.80
target frequency (Hz)	16.35	32.70	95.41	130.81	261.63	523.25	1046.50	2093.00	4186.01
deviation (cents)	-6.34	2.65	0.91	-3.04	9.75	-7.00	12.45	-29.61	-50.45

6 TEST AND VALIDATION

6.1 Latency

We tested the end-to-end latency from the MIDI input to DAC output using an oscilloscope. As can be seen from Fig. 8, the yellow signal is the MIDI message sent from the MIDI controller, the green signal is the DAC output. The cursors in the picture is taking the time difference between these two signal which is to be 3.30ms. This is meeting our requirement of 10ms.

Additionally, we also took the latency of the system from the MIDI input to DAC input. This metric is measured to be 940us as shown in Fig. 9. This shows that most of the 3.30ms latency is coming from the DAC component that we purchased, and the latency that came from our system is very minimal.

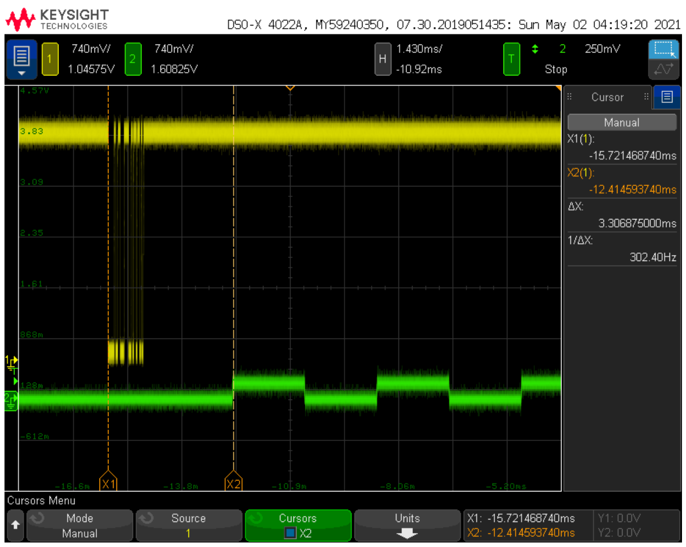


Figure 8: Latency from MIDI input to DAC output

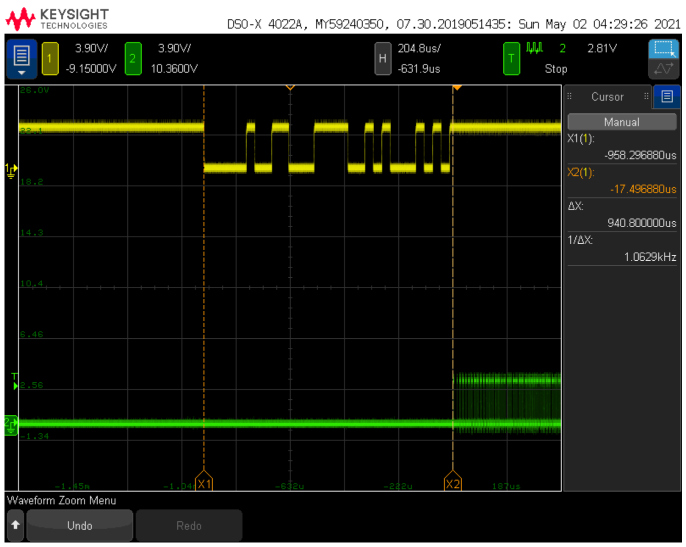


Figure 9: Latency from MIDI input to DAC input

6.2 Correctness

We ensured correctness of our programs via unit tests. For each of the feature we implemented, we wrote a unit test to check that the module produces the output that we expect from particular inputs. We ran these tests in a simulation environment using AlteraSim and VCS to check the correctness of our programs.

6.3 Accuracy

We evaluated two metrics for accuracy – frequency distortion and shape distortion. Frequency distortion is measured using the oscilloscope. A square wave of a certain note is played on the MIDI keyboard, and the audio signal is captured on the oscilloscope. A single cycle of the audio is taken using the cursor function to measure the period, so that the frequency of this waveform can be calculated. After that, this frequency is compared to the frequency of this particular musical note. The deviation is calculated in cents, which is a standard unit for measure pitch deviation. The results are shown in Table. 1 and Fig. 10. The pitch deviation is meeting our less than 10 cents requirement for all notes below C6, which is the majority of notes, but not the notes above C6. This is best we can achieve encoding the period as a 16-bit integer. If we were to use a higher precision, the cost of division in the oscillator will become too large. Therefore we accept the result.

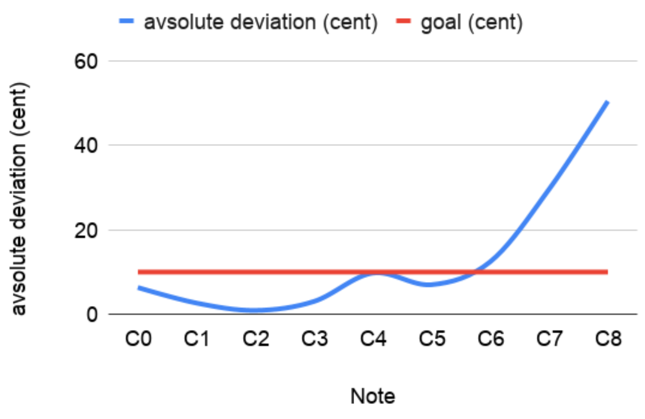


Figure 10: Pitch deviation in cents

Shape distortion is measured by comparing simulation output of SystemVerilog against the software version of the oscillators produced by MATLAB. The results are shown in Table. 2. The shape distortion is kept very minimal across signals of all frequencies, and meets our goal of less than 10% distortion across all notes.

Table 2: Shape distortion

period (# of data points)	50	500	5000
sine	0.20%	0.20%	0.42%
triangle	0.20%	0.20%	0.43%
square	0.00%	0.00%	0.00%

7 PROJECT MANAGEMENT

7.1 Schedule

The Gantt chart is in Appendix. B. We have divided our project into 4 main checkpoints. By checkpoint 1, we aimed to have a that is capable of receiving a single user input, generating a single type of waveform, and producing an audible output. By checkpoint 2, we aimed 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 aimed 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. We found out that the arpeggiator is already implemented by the keyboard so we were able to skip it. By checkpoint 4, we aimed to have a fully integrated system with all functionalities, including additional ones.

7.2 Team Member Responsibilities

Hongrun worked on the MIDI keyboard interface and the audio output, implementing the ADSR envelope, and testing and verification. Jiuling worked on polyphony control, implementing a random number generator, the arpeggiator feature, and the integration efforts after each checkpoint. Michelle worked on setting up the note-to-frequency mappings, the value look-up tables for computing the sine wave, and the unison detune effect.

A detailed breakdown of each of our responsibilities over the course of the development process is color-coded in our Gantt chart (Appendix. B).

7.3 Budget

The budget chart is included in the Appendix. A (Table. 3). 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 borrowed FPGA boards from the lab, so the cost for the FPGA boards are not included in the budget calculation.

7.4 Risk Management

Our risk mitigation strategy was to have early integrated results. This made sure we have a product that is working end to end early on. We were able to improve our product by adding features that are contained and in a manageable size each time. We were also able to see the end to end result of each feature.

We had also planned two weeks of slack time at the end, before the final presentation date, so that we had time to catch up when we had debugging difficulties. These pre-allocated slack time also enabled us to complete a few stretch goals (i.e. duty cycle and wavetable synthesis).

8 ETHICAL ISSUES

Since our project is a gadget for entertainment purpose, there were only minor ethical concerns. There were little to none edge case operations to our product, especially when our synthesizer is taken off of the FPGA and made into PCB boards, which would eliminate the opportunity for the users to abuse the FPGA for other purposes. We also did not identify any victims that would substantially be affected by our product adversely.

9 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. Others that we have studied includes FPGA Digital Music Synthesizer projects by students of other universities [2, 7].

10 SUMMARY

For our project, we aimed 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.

10.1 Lessons Learned

A big challenge we faced in our project emerged while we were implementing the two physical interfaces – the MIDI breakout board/FPGA interface via GPIO pins and the DAC/FPGA interface. At one point, we found that the off-the-shelf TRS cable we purchased that connects the MIDI controller and the MIDI breakout board was not functional. We had a difficult time isolating this issue, and for some time was really worried that we could not get them working in time. How we mitigated this unforeseen situation was that everyone paused on planned tasks and focused on working with alternative interfaces while waiting for a new cable to arrive. The slack time we planned came in handy and we spent a week on this problem. In retrospect, we should have looked into these options before settling on using DAC and the MIDI breakout board. This would have saved us some time and reduced our stress when dealing with this situation.

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.

A Budget and Material

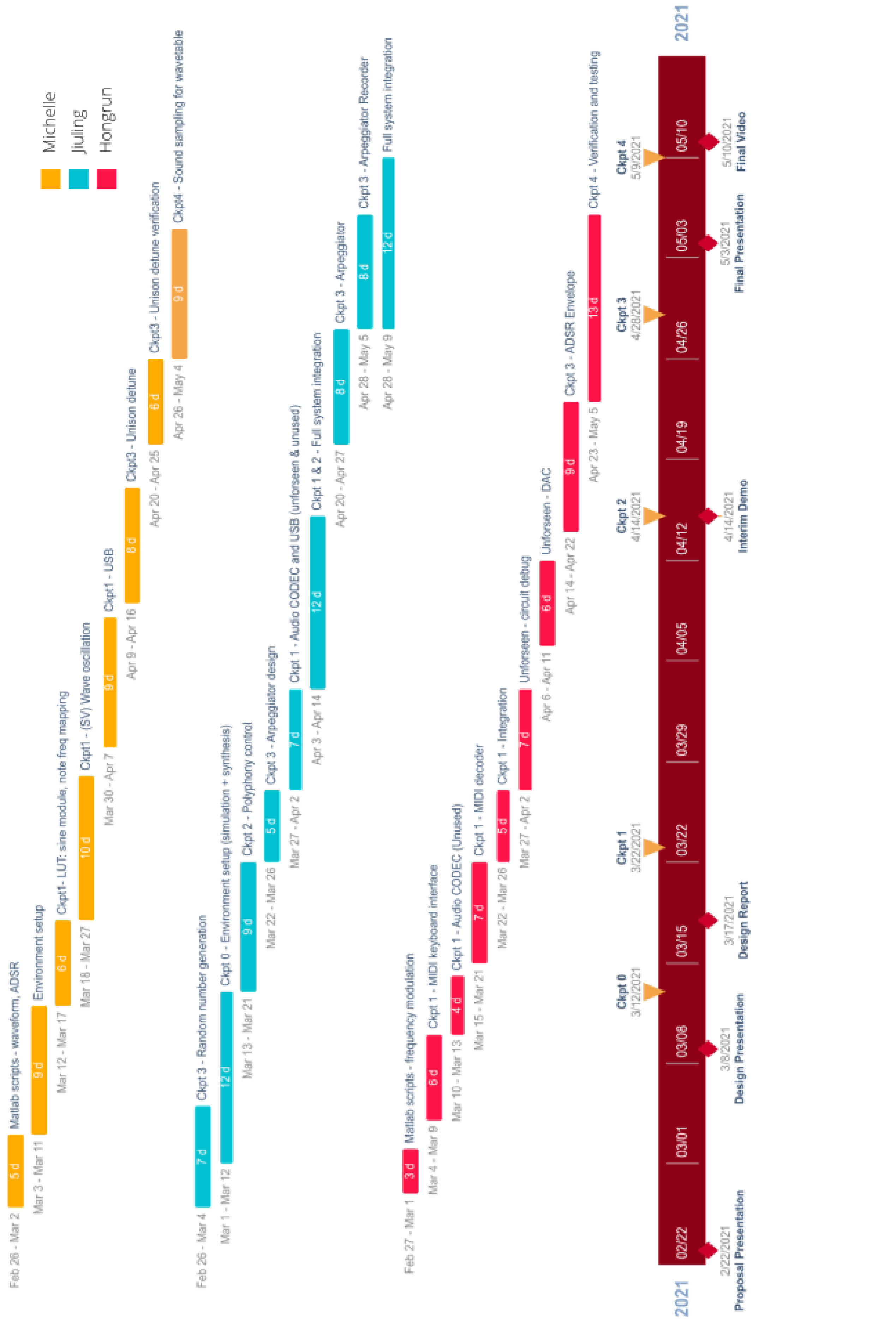
Item Name	Quantity	Price
DE2-115 Cyclone IV FPGA	3	N/A
Launchkey MINI Mk3 MIDI keyboard	3	$109.99
MIDI Breakout Board	3	$10.39
SinLoon 5 Pin Din MIDI Plug to 3.5mm TRS Stereo Male Jack Stereo Audio Cable	3	$6.32
Resistor kit (unused)	2	$6.99
DaFuRui Breadboard Jumper Kit	2	$13.99
MAX541 DAC (unused)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

Table 3: 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
I2S Stereo Decoder - UDA1334A Breakout	1	$6.95
SinLoon 5 Pin Din MIDI Plug to 3.5mm TRS Stereo Male Jack Stereo Audio Cable	1	$6.32
Jumper cable	3	N/A
		$134.65

Table 4: Bill of Materials (for one system)

B Gantt Chart


Michelle
Jiuling
Hongrun
Feb 26 - Mar 2
5 d
Matlab scripts - waveform, ADSR
Mar 3 - Mar 11
9 d
Environment setup
Mar 12 - Mar 17
6 d
Ckpt1 - LUT: sine module, note freq mapping
Mar 18 - Mar 27
10 d
Ckpt1 - (SV) Wave oscillation
Mar 30 - Apr 7
9 d
Ckpt1 - USB
Apr 9 - Apr 16
8 d
Ckpt3 - Unison detune
Apr 20 - Apr 25
6 d
Ckpt3 - Unison detune verification
Apr 26 - May 4
9 d
Ckpt4 - Sound sampling for wavetable
Feb 26 - Mar 4
7 d
Ckpt 3 - Random number generation
Mar 1 - Mar 12
12 d
Ckpt 0 - Environment setup (simulation + synthesis)
Mar 13 - Mar 21
9 d
Ckpt 2 - Polyphony control
Mar 22 - Mar 26
5 d
Ckpt 3 - Arpeggiator design
Mar 27 - Apr 2
7 d
Ckpt 1 - Audio CODEC and USB (unforseen & unused)
Apr 3 - Apr 14
12 d
Ckpt 1 & 2 - Full system integration
Apr 20 - Apr 27
8 d
Ckpt 3 - Arpeggiator
Apr 28 - May 5
8 d
Ckpt 3 - Arpeggiator Recorder
Apr 28 - May 9
12 d
Full system integration
Feb 27 - Mar 1
3 d
Matlab scripts - frequency modulation
Mar 4 - Mar 9
6 d
Ckpt 1 - MIDI keyboard interface
Mar 10 - Mar 13
4 d
Ckpt 1 - Audio CODEC (Unused)
Mar 15 - Mar 21
7 d
Ckpt 1 - MIDI decoder
Mar 22 - Mar 26
5 d
Ckpt 1 - Integration
Mar 27 - Apr 2
7 d
Unforseen - circuit debug
Apr 6 - Apr 11
6 d
Unforseen - DAC
Apr 14 - Apr 22
9 d
Ckpt 3 - ADSR Envelope
Apr 23 - May 5
13 d
Ckpt 4 - Verification and testing
02/22
03/01
03/08
03/15
03/22
03/29
04/05
04/12
04/19
04/26
05/03
05/10
Ckpt 0
3/12/2021
Ckpt 1
3/22/2021
Ckpt 2
4/14/2021
Ckpt 3
4/28/2021
Ckpt 4
5/9/2021
2/22/2021
Proposal Presentation
3/8/2021
Design Presentation
3/17/2021
Design Report
4/14/2021
Interim Demo
5/3/2021
Final Presentation
5/10/2021
Final Video