FMPGA: The Frequency Modulating Programmable Gate Array

Authors: Joseph Finn, Eric Schneider, Manav Trivedi

Electrical and Computer Engineering, Carnegie Mellon University

Abstract—**An FPGA system capable of generating, modulating, applying effects to, and outputting a digital audio stream in real time using a MIDI keyboard as input. We are mixing the benefits of software and hardware solutions to digital synthesis by providing the performance and portability of hardware synthesizers while incorporating the extreme waveform offered by software synthesizers.**

Index Terms—**Audio, Digital Synthesis, FPGA, Modulation**

I. Introduction

THE FMPGA is a hardware digital synthesizer that uses a MIDI keyboard as input. Audio producers and musicians require complete control over the timbre, tone, and quality of their sound. Products that solve this issue are divided into two classes: hardware solutions and software solutions. Software solutions provide nearly limitless options for adjusting the tone of a sound. However, since they run on a host machine, they suffer from two major flaws: (1) it is not easily portable for live performance and (2) high latency is often suboptimal for professional use. Hardware solutions can solve both of these issues by being portable and specially designed for low latency audio processing; however, many hardware solutions employ expensive analog circuits that significantly limit the product's performance at a reasonable price point. By designing a custom digital synthesizer on an FPGA, we are able to solve all the flaws of both hardware and software while retaining their benefits.

As an FPGA, it is able to interface with existing keyboards that a musician would already own. It will also provide less than 10ms of latency between a note-press and audio output. This is on par with current hardware solutions and between 5x-10x faster than software solutions. By designing it for an FPGA, the cost of our hardware is cheap enough to reasonably produce 4 note polyphony, while most commercial solutions are cost-limited to offer only 1 note polyphony. We offer 12 envelopes, 3 low frequency oscillators, 8 wavetable oscillators, frequency filtering, and distortion. Together, these features will allow the FMPGA to fulfill its goal of combining the benefits of hardware and audio synthesizers into an FPGA.

II. Design Requirements

A. Less than 10ms latency between MIDI keyboard press and audio output

A major goal of this project is to bring software-grade sound control to a lower latency platform. We have created this requirement to quantify this goal. Modern hardware synthesizers that are used commercially have between 3 and 15 ms. Our design aims to have a latency of 10 ms to remain competitive with the other entry-level synthesizers that tend to be a lot slower. The reason we opted for this value was because it was significantly less than the human auditory reaction time, which is around 140-160 ms, and within the range of a solution that most audiophiles would consider more than sufficient for audio synthesis.

B. Less than a 1% deviation in frequency from equal temperament tuning

Another thing that is important for the use case of this project is accuracy in terms of intonation. Our design will aim to have a less than 1% deviation in frequency from equal temperament tuning. It is important that the note produced is as accurate the intended sound of the key. We decided on the <1% deviation to stay within the bound of average human pitch tolerance, which ranges from around 10-30 cents (1-3%). This ensures that even those with perfect pitch will find the sounds to be fairly accurate.

C. Achieve 48 KHz, 16-bit, single channel audio output

A high fidelity sound is also crucial to the digital synthesizers, so we plan on ensuring that we can achieve a 48KHz 16-bit audio output. This is important for the purposes of creating a clear, smooth audio output. To those unfamiliar with audio terminology, this is similar to how we find videos with a high pixel density and fast frame rate more appealing to choppy, low quality ones. The goal of 48KHz and 16-bit audio output stems from industry standards of 44.1KHz and 16-bit audio which represent most music these days. To account for the fact that the average person interested in audio synthesis has greater expectations, we decided to design a model slightly beyond these parameters. We have also decided to currently focus on single channel output since it seems outside our time and area limitations to also provide dual channel output.

D. 4 wavetable oscillators capable of generating square, sine, triangle, sawtooth, and noise waves

Since our product is designed to make the actual sound we wish to augment, our synthesizer needs its own set of oscillators. For our purposes, to generate all the sounds we require 4 wavetable oscillators. These must be capable of generating a variety of different waves and sounds to allow for flexibility. The types of oscillators we are currently striving to implement include square, sine, triangle, sawtooth, and noise. Each of these generate a unique sound, which we felt we could not neglect in our design, so we hope to have each of them represented in the final design.

E. The system must be able to modify all the settings using knobs and display on an LCD screen

To allow for user input and an elegant user experience when using the product, we plan on incorporating knobs and an LCD display. The purpose of the knobs are critical since they are the means by which the waves can be modified. Using a rotary encoder instead of a standard potentiometer had the benefit of being a knob that could infinitely be spun in one direction and not need to be reset when cycling through various settings. The inclusion of a display has the benefit of reducing the number of knobs that are in the system, being a lot more appealing to entry-level producers, while still allowing the same degree of audio synthesis that comes with typical hardware synthesizer. The LCD screen also allows for the product to display a graphical representation of the envelopes and waveforms that the user would be modifying, which provides a visual element to digital synthesis that has generally only been available on software synthesizers.

F. Achieve 4-note polyphony

Something we are striving for to differentiate our design from many commercial solutions is polyphony, the ability to play multiple notes at once. Entry-level solutions at a reasonable price are limited to a single note, which would not be enough for users that may want to produce live, synthesized music, often involving multiple notes for harmony or chords. The reason why this isn't an easy task is because of the area limitations on-chip since each note of polyphony requires a duplication of the audio processing units. Thus, our design strives to have 4-note polyphony to strike a balance between hardware limitations and user flexibility.

G. Apply digital filtering and distortion on audio stream

The key component to our design is its ability to modulate the sounds that are passed in real time. Among many such manipulations we could have performed, the ones we found by far to be the most desired was the ability to modulate pitch and amplitude. Fine granular control over these allows for infinite different sounds to be produced so we decided to make those the basis of how we define success for this project. In addition to this, once we have a way to introduce the ability to modulate one aspect, we could continue to quickly and easily do so for any other effects that are supported by this project., and supported audio effects.

H. Use 12 configurable envelopes as modulation sources

In audio synthesis, the modulation source used to apply modifications to aspects of sounds like pitch or volume or other effects is called an envelope. Each envelope in the system applies a modification on a singular aspect, for example pitch for a particular note. Thus, in order to provide this vast versatility, we would need the number of envelopes to be very large. Our design aims to have around 12 envelopes throughout our model to at the very least perform volume and pitch modulation on every note of polyphony, along with 1 envelope reserved for other various effects. This would provide an ample range of possible sounds and modification, while still staying within the multiplier limitations of the FPGA we plan to use.

I. The system must have the ability to fully modulate pitch, amplitude, and other supported audio effects

While effects are not the primary goal of this project, we felt as though there were a couple notable effects we should develop to allow for a greater degree of modularity. Filtering and distortion were modulations that we agreed would expand the scope of sounds that our device could produce by manipulating the frequency using the available envelopes. The final version of this project should be able to apply digital low-pass filtering, high-pass filtering, and distortion to an audio stream. In addition to this, all of these effects can easily utilize the envelopes we plan on developing, which would grant us modulation of these effects for free once we complete the core of this design.

J. Implement 3 global low frequency oscillators for further effects

In addition to the standard wavetable oscillators, we also want to implement low frequency oscillators (LFOs) to provide another dimension of modulation to the audio output. An LFO can augment the output sound by introducing effects like tremolo, vibrato, and phasing. In order to introduce an adequate range of effects, we will strive to have 3 globally-affecting LFOs that interface with every audio processing unit in the system.

K. Save and load at least 10 presets to memory

On top of all of this, we would like this to be a product that musicians and producers continue to use for their synthesis needs. Generally, whenever they stumble upon a sound they like, they wish to use it down the line again. Thus, for a lasting user experience, we thought it was necessary to allow for the ability to save and load presets/configurations in memory. All the memory on FPGAs tends to be volatile, which means upon reset, everything stored in memory is wiped clean. So, it is paramount that we design a way to read and write to external memory to properly utilize this technology. For our purposes, we think it is a reasonable goal to have the ability to save and load 10 presets in RAM. This would provide the flexibility to have multiple sounds available, while not being too encumbering on the limitations of our design.

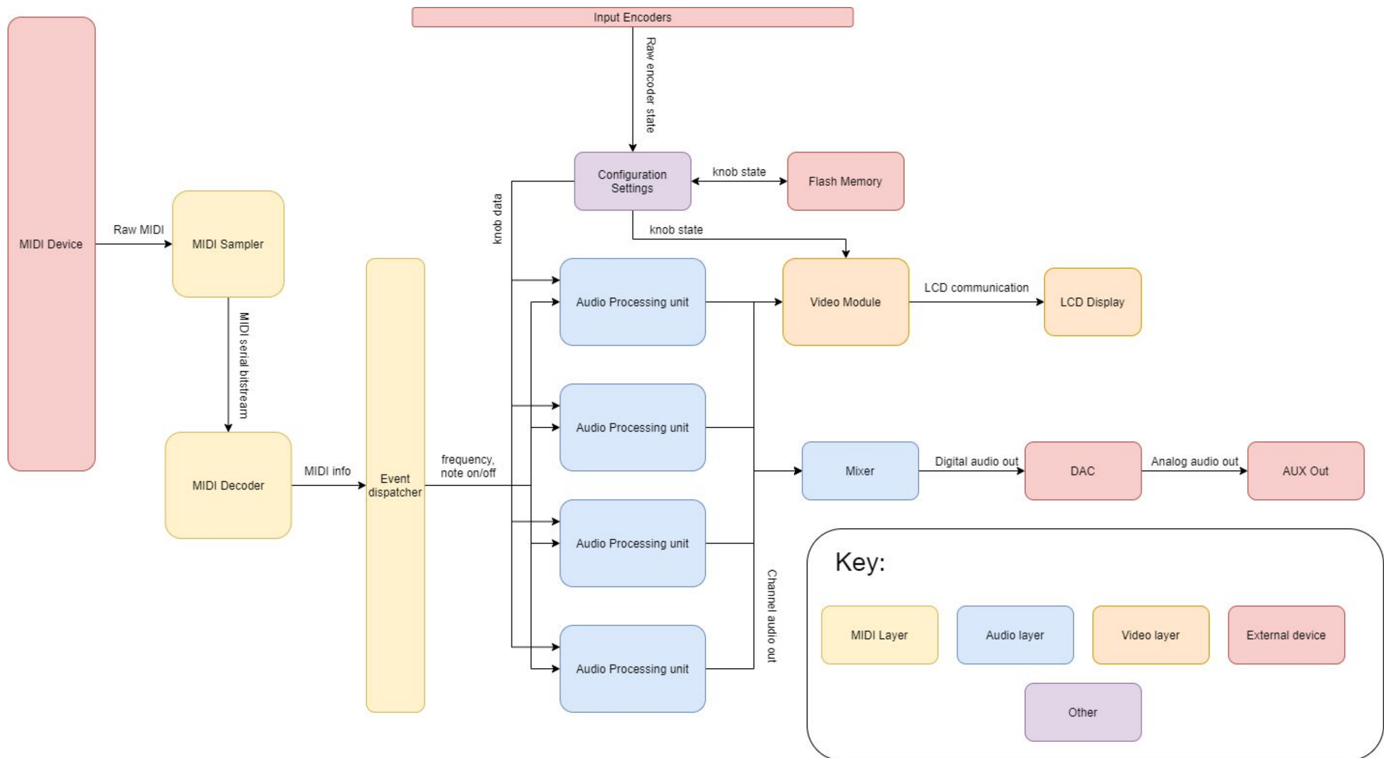

Fig. 1. High level view of the system architecture.

III. Architecture

The design consists of 3 major components: the MIDI, audio, and video layers.

A. *MIDI Layer*

This set of hardware modules functions to convert raw MIDI input into a set of 4 notes, described by the frequency, velocity, and when the note is pressed or released.

MIDI data is transmitted serially at a rate of 31,250 baud. Our FPGA will be running at a clock frequency of 50MHz, leaving us with 1,600 clock cycles per MIDI bit. The MIDI Sampler uses this information to sample the incoming bitstream by detecting an edge and counting the necessary number of cycles in order to sample the bit nearest to the exact middle of its transmission. Through this method, we are able to pass a clocked MIDI serial bitstream down the rest of the MIDI pipeline.

The MIDI Decoder takes in the MIDI bitstream and interprets the information embedded in the message. MIDI protocol dictates that the controller will hold the transmission line low until it is ready to send a packet. Then, to send a packet, the MIDI controller will transmit a "command" byte followed by zero or more bytes of data, where the number of bytes of data is determined by the command byte that prefixes the data. In our case, we only care about a subset of the MIDI messages: “Note On” events, “Note Off” events, and Pitch Bend Changes. From these messages, we can determine when notes are being played, as well as their velocity and pitch. The MIDI Decoder outputs the relevant information to the event dispatcher in the form of a packed struct.

The Event Dispatcher arbitrates which audio processing unit will handle which note press. The method by which it plans to load balance is by treating the inputs of the audio processing units as a FIFO queue. While the APUs are not currently full, the event dispatcher is free to send a new note_on signal to any of the other available units. In the case that we receive a note_off signal for any of the notes previously pressed, we can simply send that signal to corresponding units. Otherwise, once full, if we are to receive a note_on event, we have to decide which of the following APUs is the oldest and replace that particular note. This is due to a limitation in terms of how many notes we can play at once. Upon deciding the recipient unit, the event dispatcher will output the relevant information, which is frequency, velocity, note on, and note off, to the corresponding audio processing unit.

B. *Audio Layer*

The audio layer is a set of four identical audio processing pipelines and an audio mixer. Each audio processing unit (APU), depicted in Fig. 2, produces a 16 bit audio output for a single channel; the mixer combines each of these channels into a single digital audio stream.

Each APU has three envelope generators. They each produce an Attack, Decay, Sustain, Release (ADSR) curve like the example in Fig. 3. The values for each of these four parameters within each envelope generator are configurable by the rotary encoders and delivered to the

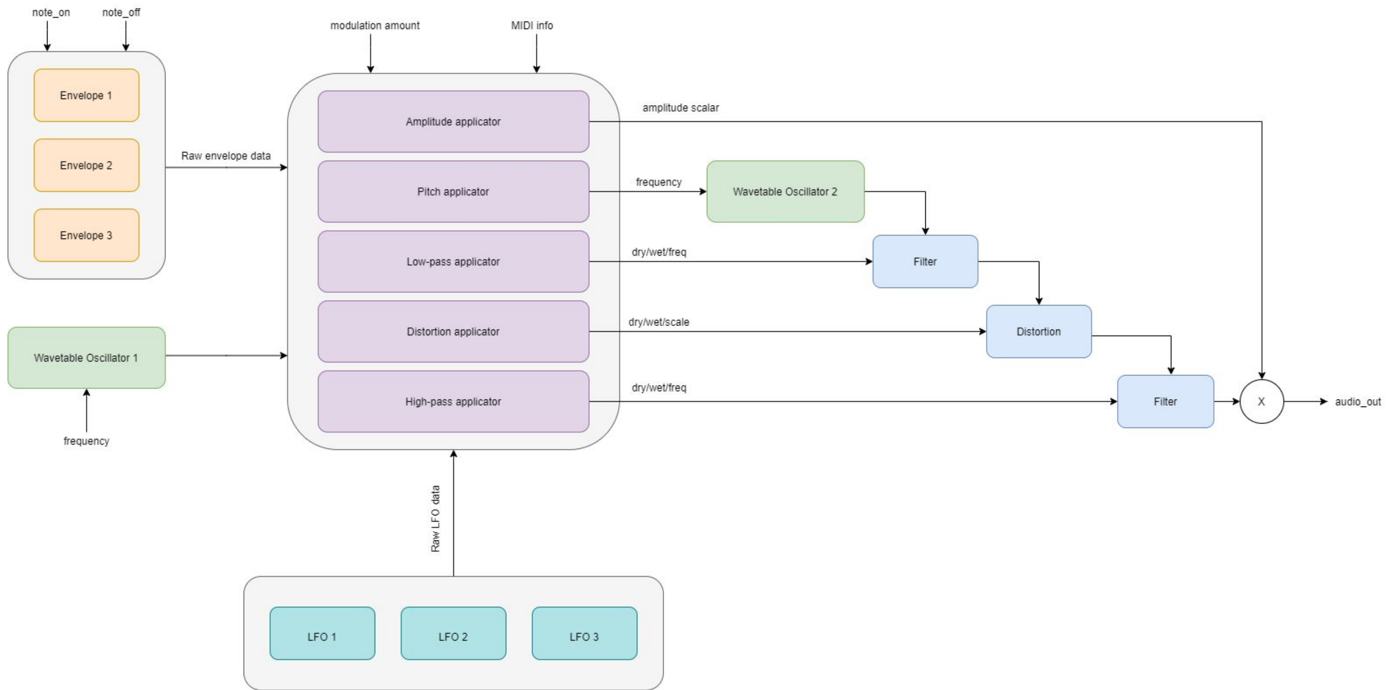

Fig. 2. Block diagram of a single Audio Processing Unit

module from the *Configuration Settings* component. The envelope generators output a 32 bit fixed point value ranging between 0 and 1 which can be multiplied to any parameter to modulate it.

Each unit also has a set of applicators. The general applicator circuit is depicted in Fig. 4. Based on the configuration settings, the applicator will multiply its dedicated parameter by the output of the appropriate envelope. The applicators allow any combination of their modulation sources to be applied to any of their possible modulation parameters (i.e. pitch, amplitude, filters, and distortion). Possible modulation sources are the envelopes, low frequency oscillators, and Wavetable Oscillator 1.

Each unit has two wavetable oscillators, which generate a wave at a given frequency. The shape of the wave can be configured to choose a sine, square, or sawtooth wave, and the oscillators can play frequencies between 20Hz and 4000Hz with <1% error. There will be dedicated flash RAM to store preconfigured wavetables, which contain 2048 samples of a single cycle of a given waveform. The oscillator constantly increments a phase offset by a step size, where

$$step(f) = (N_s \cdot f) / F_s \quad (1)$$

and N_s is the number of samples in the wavetable (2048), F_s is sampling frequency (48KHz), and f is the desired output pitch. In order to index into a discrete table, the phase is incremented by the integral component of the step size on each cycle, while the fractional component is accumulated in an error register. When the error register overflows to greater than 1, it is added to the phase and reset. One wavetable oscillator produces a wave at the output pitch of the current note, while the other oscillator is used as a modulation source.

The APU contains a distortion effect as well as two filters, which can be toggled between low-pass and high-pass.

The high-pass and low-pass filters are implemented as an infinite impulse-response filter (IIR) and a finite impulse response filter (FIR) respectively. We chose these filters because they're very effective filters that can be fully described by only a handful of coefficients. As well, the FIR can be implemented as a special case of the IIR, so the hardware will be identical.

The coefficients for these filters are generated using very complex math that is infeasible to do in real-time on hardware. To compensate for this, we plan to generate these coefficients in software for a variety of frequencies, then linearly interpolate between them in hardware to approximate the coefficients in a computationally efficient way.

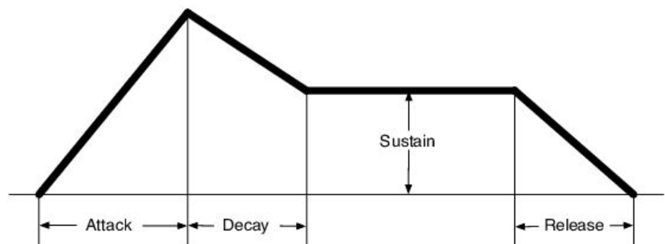

Fig. 3. An example ADSR curve.

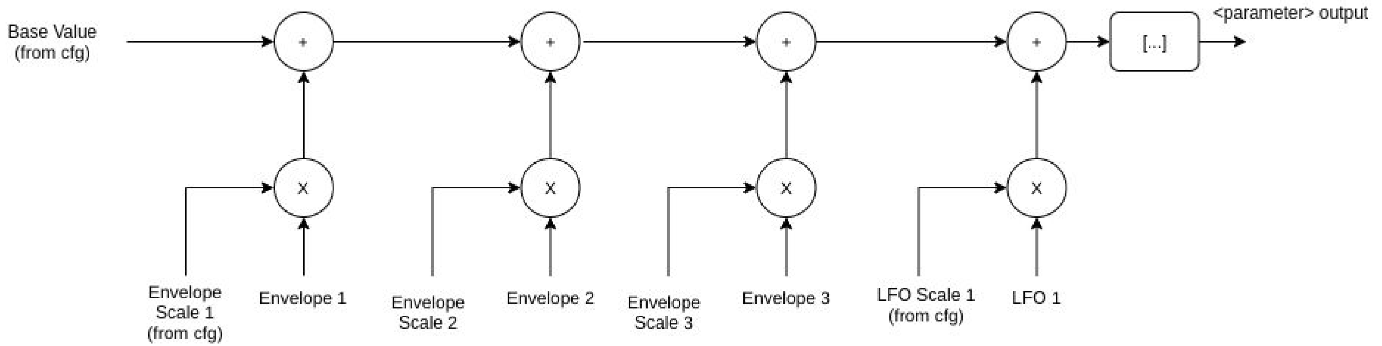

Fig.4 The general multiply-accumulate circuit inside the applicators.

The distortion effect is relatively simple, and, unlike the filters, only operates on the current audio sample, rather than a history of audio samples. To create a "distorted" or "saturated" effect, we take in a normalized sample that exists in the range [-1, 1], and we pass it through a function whose output is also constricted to the range [-1, 1]. To create a "distorted" effect, we use an S-shaped curve for this function, as shown in Fig. 5. The function *distort* is defined as:

$$distort(x,\ k) = 1 - (1 - x)^k \ \text{ if } x \geq 0 \qquad (2)$$
$$distort(x,\ k) = -1 + (1 + x)^k \ \text{ if } x < 0 \qquad (3)$$

In this function, k represents the intensity of the distortion. When $k = 1$, there is no distortion, and as k approaches infinity, this function approaches a square wave. The variable x is the value of the current sample. This function was chosen because it generates an S-like curve without using trigonometric functions, exponentiation, or division. The only complexity introduced with this function is that k may be non-integral—we deal with this by linearly interpolating between integral values of k.

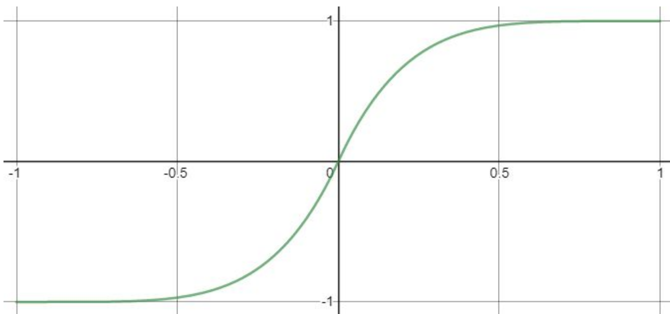

Fig.5 The distortion function with k=5.

The entire audio layer shares three low-frequency oscillators. These oscillators function similarly to the wavetable oscillator, except at much lower frequencies, in the range of [0Hz, 127Hz]. These oscillators are used by the applicators to periodically modulate parameters in addition to the ADSR envelopes.

The four audio processing pipelines are combined in the mixing module. This module simply adds the outputs of the four audio processing units and normalizes it to a 16 bit digital output. This is the final stage of audio processing, and the rest of the audio pipeline consists of an external digital-to-analog converter which takes in the output of the mixer and converts it into playable audio.

C. Video Layer

The video display on the FMPGA serves two important functions. Its most important role is to serve as a visual indication of the settings that the user can affect when designing sounds on the synthesizer. Once the sound has been created, the display can be configured to show the audio waveform as the note plays.

The display we use has a resolution of 128x64 monochrome pixels, and it is driven through an SPI interface. The display can only be driven at a maximum clock speed of 20MHz, and since our clock runs at 50MHz, that means we can drive the display at a speed of 16.67MHz. Because it takes 16 cycles to update eight pixels of the display, and the display contains 8,192 samples, we calculate that updating the screen will take, at minimum, $2N_{pixels}/f_{SPI} = 983\mu s$ to fully update the screen. It will take slightly more time than that, as there are more SPI packets that need to be sent to configure the display, but in total, updating the display shall not take more than 2ms. Because we only need to update the display every 16.7ms to attain 60 frames per second, we have a lot of time to perform rendering calculations.

To render to the display, we have two separate rendering modules that always store their rendered frame in their own memory. Then, to render the displays, we simply mux those RAM outputs with a select line, and draw to the screen using SPI.

The first rendering module renders the waveform. This module records all outgoing samples in a shift register, and plots the audio in terms of its squared magnitude. Because we attempt to render every 16.7ms, we have to fit 735 samples onto a screen that is only 128 pixels wide, which we accomplish with averaging. The second rendering module displays the configuration settings. This gives the viewer a

visual representation of the settings that they modify with the rotary encoder, which is fairly trivial—it simply involves drawing rectangles dynamically on a background image that is stored in memory.

IV. System Implementation and Design Trade Studies

A. Choice of FPGA

Our requirement of four note polyphony necessitates that we use an FPGA which is large enough to hold four copies of our audio processing pipeline. As a result, we had to choose an FPGA such that we maximized the number of logic elements and on-board multipliers. We identified two FPGAs, shown in Table 1, that were suitable for our needs. The DE-10 has a larger area but fewer multipliers, while the 5CEBA5 has less area but more multipliers. Both choices seemed reasonable, so we selected the one which is easiest to work with, the DE-10. A breakout board is readily available and provided by CMU, so it's much easier to obtain than the 5CEBA5, which is hard to find on a breakout board at a reasonable price.

B. Memory type

Due to the volatile nature of on-chip memory, it is necessary to have some form of external memory that allows our design to read and write without the fear of loss of data. This led us to the crossroad of debating between either flash memory or SD cards. Both methods seem equally challenging and the more appealing option as of now is flash memory since we do not need to interface with any other peripherals.

C. Display

In order to retain the ease-of-use of software synthesizers, we need to include visual feedback for the current settings of the synthesizer. The user will be able to cycle through the different modulation sources and see their individual state. They will also be able to view the output waveform in real time. A reasonable display for our use case needs to simply be black and white since we have no particular need for color. The pixel density for this screen should be enough to display a waveform. We reasoned based on the pictures and reviews of the particular 128x64 LCD panel, we decided that it was sufficient enough to display numbers and waves.

TABLE I. FPGA Options

FPGA	Logic Elems	27x27 Multipliers
Terasic DE-10	110k	112
Cyclone 5CEBA5	77k	150

D. Audio Layer Latency Analysis

The audio layer is the critical path of our design, and will determine if we can meet our timing requirement of <10ms latency. We needed to verify this latency is realistic for our proposed design, so we performed the following estimation.

First, we made the following assumptions:

i. Sampling rate $F_s = 48KHz$
ii. Total audio pipeline stages N_A will lie in the conservative range of $50 \leq N_A \leq 400$
iii. FPGA clock frequency F_C will be in the range $10MHz \leq F_C \leq 50MHz$

To meet our latency requirement, it is necessary that:

$$10ms \geq N_A / F_c \quad (4)$$

In the worst case, we maximize N_A and minimize F_C. By combining (4) with assumptions (i) and (ii), we get a latency of 40µs. This leaves us with over 9ms to decode MIDI and pass our output through the DAC, which we estimate is very feasible.

V. Project Management

A. Schedule

Initially, we were very ambitious with creating a very tight schedule for ourselves: we planned to be done with simulations within the first 7 weeks. We quickly realized that the burden of other classes deemed this schedule unrealistic, so we adjusted it to spread out our time more evenly over the course of the semester. We redistributed our workload to spend more time designing and planning in software, which extends the amount of time before we get a working prototype; however, it has the enormous benefit of allowing us to complete hardware implementations much more efficiently. Our schedule is available on the last page of this document as Fig. 7.

B. Team Member Responsibilities

Because of the modular nature of this project, we decided to break up responsibility based on each of the layers described in Section III.

Manav's primary responsibilities are the MIDI layer to convert serial input to a series of discrete events that are arbitrated to the APUs, interfacing with flash memory to write and retrieve configuration settings, and the DAC interface to convert digital signal to audio out.

Joe's responsibility is the audio layer, which consists of the oscillators, which are used to generate a digital wave; envelopes to perform modulation on pitch, volume, and other effects; effects such as low-pass, high-pass, and distortion for a greater range of sounds; LFOs to generate effects like vibrato, tremolo, and phasing.; and a polyphony mixer to generate a combined wave that can be used as output.

Eric's responsibilities are the video layer, which consist of the I2C interface for the LCD to display the waves and configuration settings; the input encoders to gather user input to modify and the modulations and effects applied; and the configuration settings module to serve as the centralized hub for modifying the system.

C. Budget

Though not an official requirement, we have planned the

project since the beginning with a goal of being low cost. Our goal was under roughly $200; since we are not purchasing an FPGA, we did research to find an appropriate FPGA to use should we manufacture our product, to see if our price target was indeed reasonable.

TABLE II. BUDGET

Part Name	*Quantity*	*Price*	*Total*
FPGA - Terasic DE-10 Standard (Provided by CMU)	1	$0.00	$0.00
---------- 5CEBA5F23C8N (price if we were to manufacture it)	1	$88.04	$88.04
Rotary Encoders (EN11-HSM1BF20)	10	$1.08	$10.80
Digital to Analog Converter (DAC101S101CIMK/NOPB)	1	$1.69	$1.69
MIDI DIN connector	2	$1.75	$3.50
LCD Display (NHD-C12864A1Z-FSW-FBW-HTT)	1	$22.69	$22.69
MIDI Keyboard - Novation Launchkey Mini mk3 (for testing)	1	$109.99	$109.99
---------- Estimated cost of PCB and other misc. manufacturing costs	1	$50.00	$50.00
	Cost for Project		$149.62
	Cost to Manufacture		$176.67

D. Risk Management

In terms of time, we attempted to be very realistic about how much of a time investment this project would be and made sure to allocate an adequate amount of slack near the tail end of our project to make up for any unforeseen circumstances.

In order to increase our hardware design efficiency, we have been trying to design each part of each subsystem to be extremely modular such that we avoid large, grotesque modules that are hard to test. This way, once we can ensure that the singular element is working correctly, we can assume down the line, any emergent issues are with other elements in the design. Paired with a detailed verification platform, we can be assured that the final design works as smoothly as possible.

In terms of resources, we have accounted for the fact that many of the smaller pieces like rotary encoders, buttons, and so on are bound to break, so we added slightly more than necessary to our budget as a backup.

On a planning level, we tried to make sure that the FPGA we plan on using had more than enough multipliers and logic elements to make sure we don't find that our board is insufficient for our use case.

VI. VERIFICATION

There was a significant emphasis placed on verification and testing from the very beginning of this project. We understand that large hardware-based projects are often hard to test and in the real world have entire teams dedicated to verification, thus we wanted to incorporate a strict testing platform as a priority. We wanted to make sure that we were testing throughout the span of the project, rather than leaving it all at the end, causing more hassle than necessary. We still have a dedicated time for verification of the whole project near the end of our schedule, but this is meant to represent testing the high level design and fine tuning.

Before we start implementing any hardware for the design, we ensure that we have a working software prototype designed in Python. The purpose of this is two-fold: first, to provide us with a general understanding of how to design the SystemVerilog description of the module, given that software is generally easier to implement; and second, to allow us to systematically compare the inputs and outputs at every clock cycle to determine exactly when unexpected behavior occurs. Developing such a robust software model of our design has allowed us to engage in test-driven development for the entirety of the semester.

The testing environment is derived from the Universal Verification Methodology (UVM). The exact flow can be seen in Fig. 6. We designed a module that monitors all inputs and outputs exchanged between a SystemVerilog testbench and the device-under-test (DUT). It outputs these signals to files in a standardized format. A Python module then reads these files and turns them into cycle-by-cycle input and output streams, respecting the names and bitwidths of each signal. The inputs are fed into a Python test driver, which sends them to a Python "golden" model of the DUT. The outputs of the golden model are then automatically compared to the outputs of the SystemVerilog DUT within the test driver in order to verify the design. All of this functionality is automated through a script, aside from the creation of the DUT, the testbench, and the golden model, which are necessarily custom to each module.

Almost all of our verification can be done using this method, since it allows us to bring up and verify the entire design in simulation. The only further verification that needs to be done post-simulation is (a) verifying the accuracy of intonation through the DAC using a tuner, (b) verifying the latency from keypress to audio-output using high speed audio capture, and (c) verifying that the output of the LCD is to our liking.

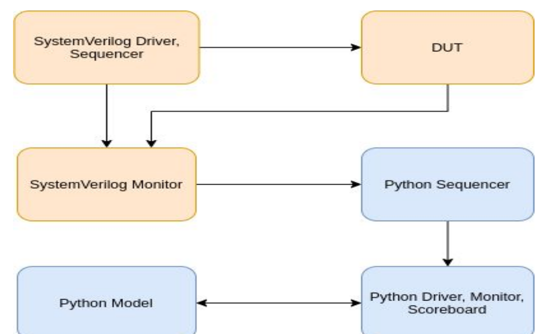

Fig.6 Flow chart of our test environment

VII. References

[1] ADSR diagram, The elements in an ADSR envelope
[2] DE-10 Standard Design Specification, Cyclone V Device Overview
[3] MIDI message table, Summary of MIDI Messages
[4] Digital Filter, Lecture 6 - Design of Digital Filters
[5] IIR Filters, IIR Filters - an overview
[6] Display Specifications, NHD-C12864A1Z-FSW-FBW-HTT
[7] Display Controller Specification, Sitronix

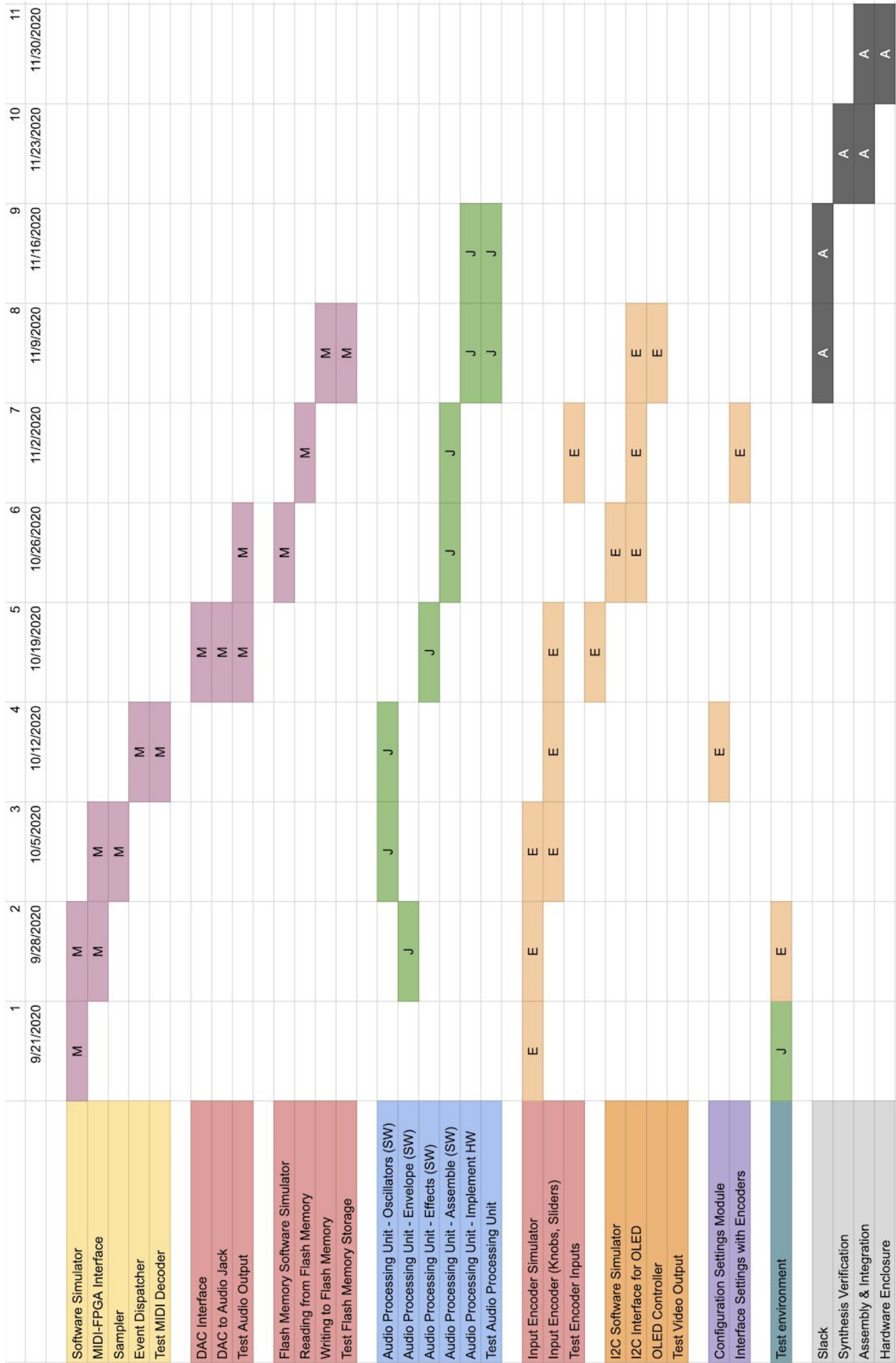

Fig.7 Schedule and division of labor