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 44.1 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 44.1KHz 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 44.1KHz 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, and sawtooth 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, and sawtooth. 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 distortion effect to 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 12 envelopes (3 per note of polyphony) throughout our model to perform volume and pitch modulation, with 1 envelope per note reserved for amplitude. 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

Modulation is the primary purpose of our project. We are able to modulate amplitude, pitch, and distortion using any available modulation source on the device. We have developed an architecture that easily allows more audio parameters or modulation sources to be built into the design, allowing the production of even more complex sounds.

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.

III. Architecture

The design consists of 3 major components: the MIDI, audio, and video layers. The top level block diagram can be found at the end of this document as Fig. 11.

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, a value we will call SAMPLING_RATE. The MIDI Sampler uses this information to sample the incoming bitstream. Every SAMPLING_RATE clock cycles (given no edges are detected), it will sample the current value of the bitstream. Otherwise, if it encountered an edge as it was in the midst of counting the necessary number of cycles, it will restart the count with SAMPLING_RATE/2 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.

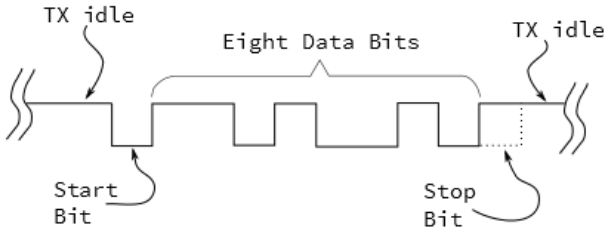

Fig. 1. Diagram of MIDI UART transmission

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 high until it is ready to send a packet. Then, to send a packet, the MIDI controller will transmit messages in accordance to the UART protocol. This predicates that it sends 10-bits: the controller will first transmit a start bit "0" followed by 8-bits of data, and finish with a stop bit "1".

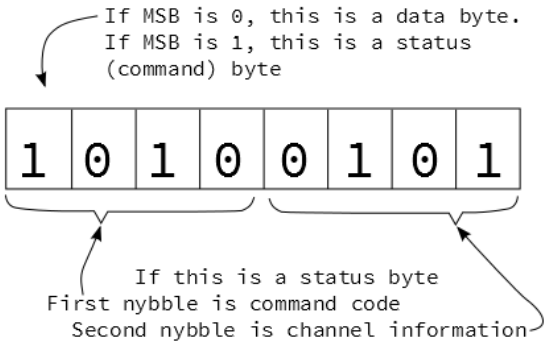

Fig. 2. Deconstruction of MIDI messages

Given that the start and stop bit construction is valid, the decoder will utilize these data bits to construct messages. There are two classes of messages: status and data, represented by the first bit of the message "1" or "0". There are numerous status messages, but for our purposes, we only care about a subset of them: "Note On" events and "Note Off" events. From these messages, we can determine when notes are being played, as well as their velocity and pitch. The pitch and velocity are data values that are transmitted immediately after the status message.

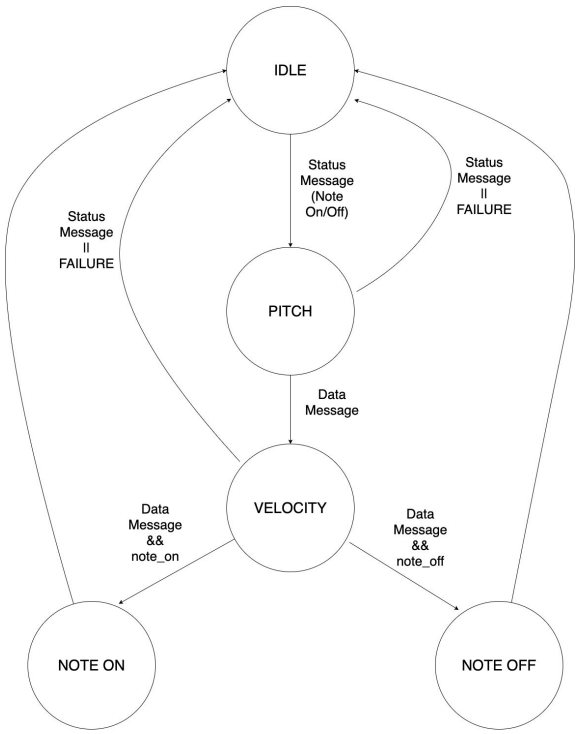

Fig. 3. Simplified FSM of MIDI decoder

A change we had to make was that the MIDI controller we utilized did not actually send note off messages. Instead, in its place, it sent note on messages with a velocity of 0. Since the bulk of the other components were created with pretense that note off's would be received, we translated this version of note off to our expected version as the message was in transit. Secondly, the pitch value outputted by the MIDI controller was not in the form we planned to use directly since it just denoted the corresponding note number. In order to get the pitch in the expected form, we created a module to function as a lookup table to convert MIDI pitch to a 27-bit fixed point frequency we would perform modulation upon. All this information is then coalesced in the form of a packed struct by the event packager to distribute to the APUs.

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 final implementation, we realized it shouldn't be this simple because upon receiving

a note off message, we unallocate an APU, but the note may still be in the release stage of the ADSR curve. In order to handle this, we always allocate to the least recently used APU. 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.

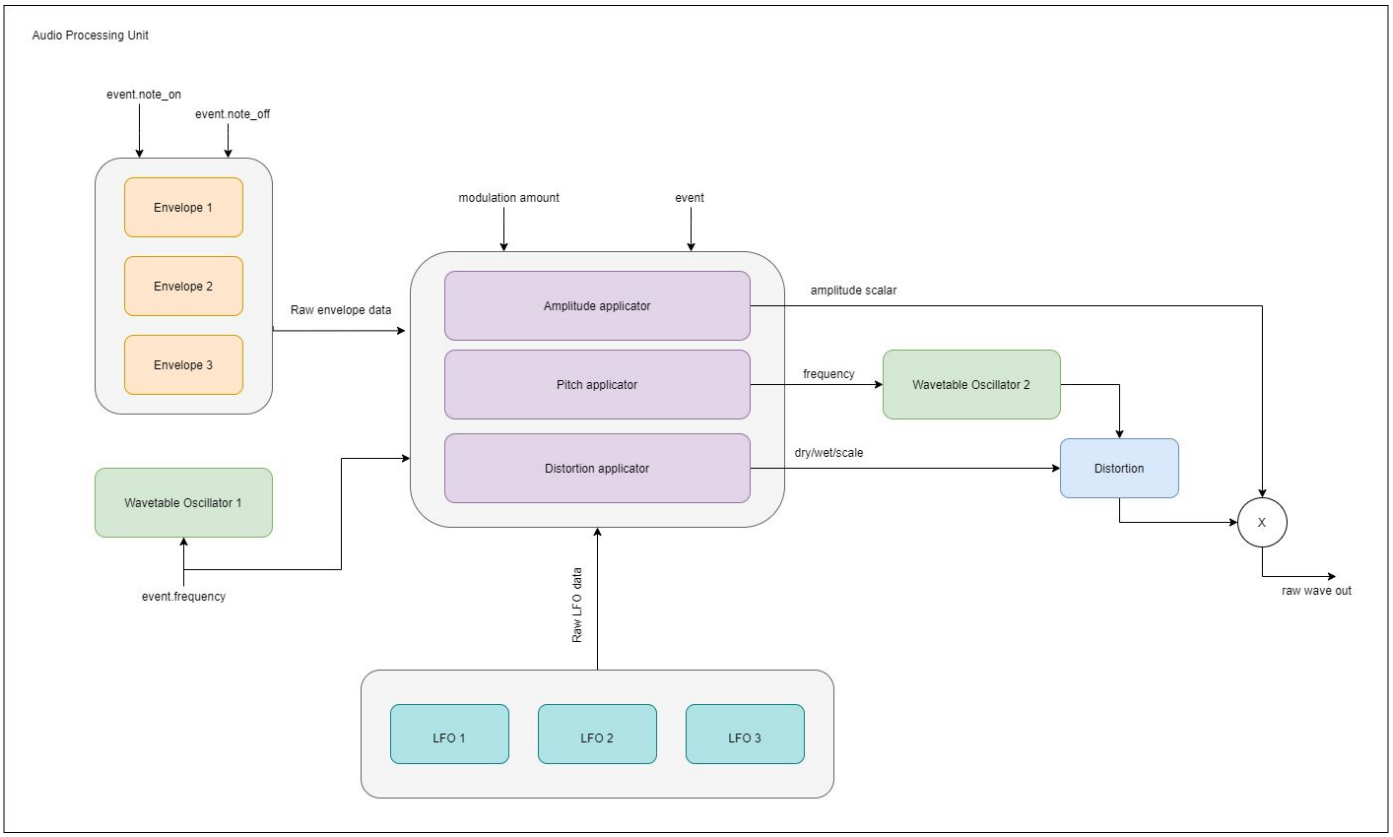

Fig. 4. Block diagram of a single Audio Processing Unit

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. 4, 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. 6. The values for each of these four parameters within each envelope generator are configurable by the rotary encoders and delivered to the module from the *Configuration Settings* component. The envelope generators output a 27 bit fixed point value ranging between 0 and 1 which can be multiplied to any parameter to modulate it. The A, D, and R parameters are time values ranging from 0 to 4 which control how long, in seconds, each stage of the curve occurs for. The S parameter is a scalar between 0 and 1 that describes an amplitude value during the sustain phase. Each of the four stages has a corresponding linear equation to describe it:

$$attack(t) = t / A \quad (1)$$
$$decay(t) = t * (S - 1) / D \quad (2)$$
$$sustain(t) = S \quad (3)$$
$$release(v_i, t) = t * -v_i / R \quad (4)$$

The v_i parameter in *release* refers to an initial value, which is constantly updated as the newest output from the envelope generator. It exists to create a smooth transition to the release state if the note is released during the attack or decay states. The division operations are unavoidable, and generic division is not easily solvable on an FPGA. Our solution is a 32KB ram block that stores a lookup table for the function $1 / x$ for $0 \leq x < 4$, allowing us to compute the division operations in a single cycle.

Each unit also has a set of applicators. The general applicator circuit is depicted in Fig. 5. 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 (envelopes, LFOs, velocity, and modulation

wavetable) to be applied to any of their possible modulation parameters (pitch, amplitude, distortion *k*, and distortion *dry/wet*). To save FPGA area, the applicator circuit uses one multiplier, and continuously refreshes the output value by performing a sequential multiply-accumulate on all of its modulation sources and scalars.

Each unit has two wavetable oscillators, which generate a wave at a given frequency. The shape of the wave can be

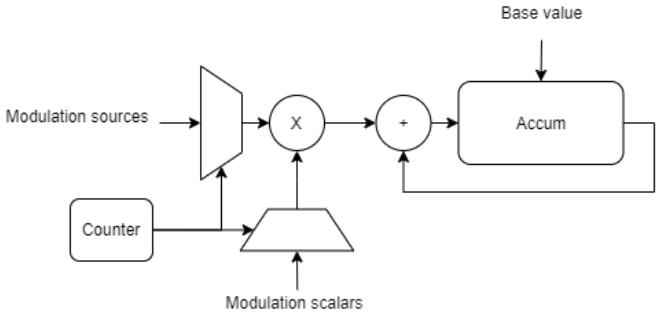

Fig. 5. Applicator multiply-accumulate circuit

configured to choose a sine, square, or sawtooth wave, and the oscillators can play frequencies between 20Hz and 4000Hz with <1% error. There is dedicated 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

$$step(f) = (N_s \cdot f) / F_s \qquad (5)$$

and N_s is the number of samples in the wavetable (2048), F_s is sampling frequency (44.1KHz), and f is the desired output pitch. The inverse of sampling frequency was precomputed to avoid the need for division. 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.

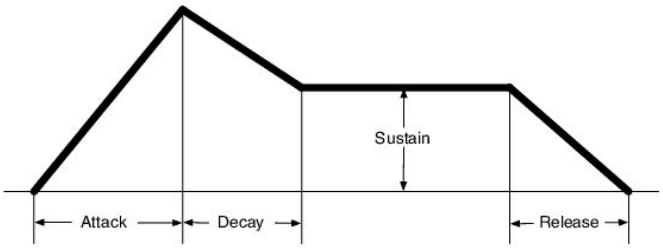

Fig. 6. An example ADSR curve

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]. We use an S-shaped curve for this function, as shown in Fig. 7. The function *distort* is defined as:

$$distort(x, k) = 1 - (1 - x)^k \text{ if } x \geq 0 \qquad (6)$$
$$distort(x, k) = -1 + (1 + x)^k \text{ if } x < 0 \qquad (7)$$

Where 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 or division. A possibly non-integral power of k necessitates a power-computing module. This module repeatedly multiplies the base by itself for $floor(k)$ and $ceil(k)$ number of times, and we can linearly interpolate between these two outputs to determine the true result. The distortion module also takes a *dry/wet* parameter, which determines how much of the signal becomes distorted. The final output of the distortion module is:

$$x_o = dist(x_i, k) * wet + (1 - wet) * x_i \qquad (8)$$

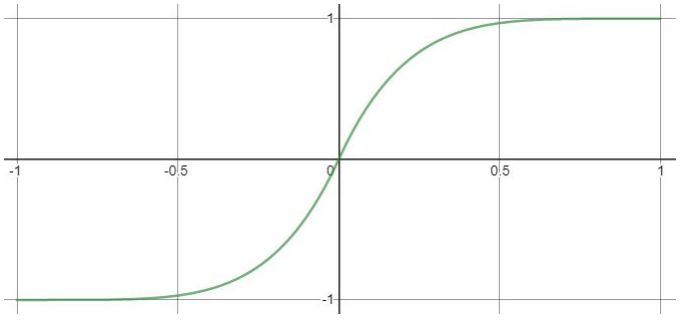

Fig. 7. The distortion function with k=5.

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, 20Hz]. 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 gives the user a visual representation of the modulation settings. Presenting this information to the user in a useful way is key to the usability of the system as a whole; since we have over 50 parameters to modulate, it's crucial that we create an environment that

allows easy access to all parameters, yet doesn't overload the user with information. As such, we decided on a page system, where the parameters are grouped by category into nine pages, each containing up to eight parameters.

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.

Due to concerns with FPGA resource utilization, we decided not to buffer our video output. Instead, we generate the SPI outputs on the fly with combinational logic. To do this, we keep multiple ROMs: one which contains the name of each page, one which contains the name of each parameter, and two which store information about the rectangle graphics we render around each parameter. Then, our output is generated by some clever address lookups and logic gates. In the end, the video driver module was one of our smallest modules in terms of logic utilization, but this came at the cost of high memory usage.

To interface with the display, we use three encoders. The first encoder changes which page the display is showing. The second encoder changes which parameter is selected. The third encoder changes the value of that parameter. The name of the page is shown in the top left, and the name of the parameter is shown in the top right. On the display, a visual indicator of the parameter's value is displayed for each parameter on the page.

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. 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

C. Space/Functionality Tradeoff

Our original design included digital low and high pass filters in each APU. These modules would each require tens of multipliers, up to 400 bytes of register storage, and lots of combinational logic — and we would need eight of them. We used nearly all of our on-board DSP blocks and logic elements as shown in Fig. 9, so there simply was not enough space to include them in the design. This was an unfortunate tradeoff that we had to make in order for our product to work.

We needed to make some tradeoffs in order to fit the completed design on the FPGA, even without the filters. We chose to use 27 bit arithmetic to satisfy our FPGAs limited multiplier space. This does not offer enough precision to represent our sample period, which could cause our wavetable oscillators to output at an inaccurate frequency. Thankfully, the difference in precision seems to have a negligible effect on the output frequency.

We also needed to sacrifice ease of implementation in exchange for less multipliers in a few modules, mainly the applicators. Rather than using one multiplier per modulation source and performing a parallel multiply-accumulate, we decided to change our design to perform sequential multiply-accumulate with a single multiplier. This increases latency by a few cycles and makes implementation less convenient, but saves a lot of multipliers.

D. Implementation time/Functionality Tradeoff

We originally had plans to save knob configurations as presets to flash RAM, and to display the output waveform on the LCD. We ended up needing to focus our time towards more critical parts of our system in order to create a minimum viable product, and so we did not have the time to implement these features. Our design took up an unexpectedly large amount of logic elements, so these features probably wouldn't fit on our FPGA anyways.

E. Audio Layer Latency Analysis

The audio layer is the critical path of our FPGA design, and we need to determine how much time it takes to process a sample in order to leave ample time for MIDI and DAC

latency to fit our 10ms budges.

We achieved the following parameters in our design:

i. Sampling rate $F_s = 44.1KHz$
ii. Total audio pipeline stages N_A is on the order of $10k_{dist}$, with $k_{dist} \leq 10$
iii. FPGA clock frequency F_C is 50MHz

We then get a worst case APU latency of:

$$L_{APU} = N_{A,max} / F_c + 1/F_s = 25\mu s \quad (9)$$

This leaves us with over 9ms to decode MIDI and pass our output through the DAC.

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. One aspect of our schedule that took a lot longer than we originally allocated time for was integration. We found that synthesizing the whole design and integration not only took our whole slack time but also an extra week at the end to ensure all our promised requirements were met. Our schedule is available on the last page of this document as Fig. 10.

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 and implementing a polyphony mixer to generate a combined wave that can be used as output.

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; distortion effect for a greater range of sounds; and LFOs to generate effects like vibrato, tremolo, and phasing.

Eric's responsibilities are the video layer, which consist of the SPI interface for the LCD, to display the configuration settings, and DAC, to output audible sounds; 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, minus the materials necessary for testing.

TABLE II. Budget

Part Name	*Quantity*	*Price*	*Total*
FPGA - Terasic DE-10 Standard (Provided by CMU)	1	$0.00	$0.00
Rotary Encoders (EN11-HSM1BF20)	10	$1.08	$10.80
Digital to Analog Converter (DAC101S101CIMK/NOPB)	1	$1.69	$1.69
Op Amp (LM741)	10	$0.49	$4.90
MIDI DIN breakout board	1	$14.69	$14.69
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
	Cost for Project		$164.76

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 tried 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.

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. 8. 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.

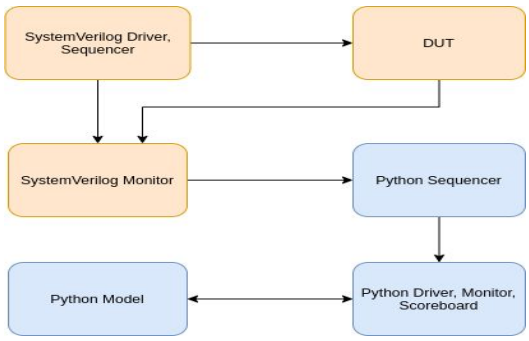

Fig. 8. Flow chart of our test environment

VII. Summary

Overall, we are very satisfied with the outcome of our project. We achieved the modern standard of high quality audio with four note polyphony and full control over amplitude, pitch, and distortion modulation, which is a rare combination of features for hardware synthesizers. We are particularly proud of our ability to meet the audio quality and polyphony requirements, because they are entirely reliant on our ability to efficiently make use of FPGA space and latency. The implementation is also extremely modular, allowing for more polyphony, modulation sources, and audio effects with a simple change of parameters (and a bigger FPGA).

A. Results

We were able to meet and exceed the majority of our requirements, aside from high/low pass filters and preset saving. Thankfully, these two requirements were low-priority relative to the rest of the design.

1. In order to calculate the latency, we recorded the audio and analyzed its waveform in an audio edit. We determined that in the worst cases, we were within 8.5 ± 1ms for audio output from the moment that we pressed the sound. One major difference compared to traditional software solutions was that they had a significant lag between note press and sound out, which is nearly nonexistent in our solution. Based on a qualitative analysis, it sounds instantaneous since our measured latency was well within the human auditory reaction speed.
2. We measured our output frequency over a range of notes using a tuner, and measured within 0.29% of the desired note frequency for all samples. This far exceeds our requirement of <1% deviation in frequency.
3. We achieved both a 44.1KHz output frequency and 16-bit audio.
    a. 44.1KHz output frequency was achieved with ease. We were able to synthesize a 50MHz clock frequency, the maximum allowed by our FPGA, which allows us over 1000 cycles to generate a sample, of which we only used a handful of cycles.
    b. 16-bit audio was achieved, however our limited FPGA area caused this requirement to be quite a challenge. If we had used less bits of precision, we may have been able to fit filters or even an extra note of polyphony.
4. Our final design was well beyond the 4 wavetable oscillators we originally intended to create. We ended up using 8 wavetable oscillators that were each capable of generating square, sine, triangle, and sawtooth waves. Within each APU, the first wavetable oscillator generated the frequency wave and the second wavetable oscillator allowed modulation.
5. Utilizing the encoders and LCD display, we were able to create a fairly clean, easy-to-learn user interface. Using the three encoders, the user will be able to navigate through the 50+ configurable values and modulate them to create their desired sounds.
6. Our final product was able to achieve 4-note polyphony. This was one of the highest-priority requirements that we did not want to sacrifice compared to the other effects. In order to guarantee this, we needed to create a polyphony mixer to merge all the signals into a singular output.
7. We were also able to successfully apply the distortion

effect to the audio stream by modulating this value using the existing ADSR envelopes.

8. Each of the APUs in the system were composed of 3 configurable envelopes each to be used as modulation sources. The total number (12) met the requirements we were hoping to achieve, and were able to fully modulate any of the parameters in our system.
9. The final version of the FMPGA possessed the ability to fully modulate pitch, amplitude, and distortion (dry/wet and k-value) utilizing velocity, a wavetable oscillator, and the LFOs. This superseded our previous expectations of the project and allowed us to generate an astronomical number of sounds.
10. We also were able to generate 3 global low frequency oscillators for further effect pitch, amplitude, and distortion with a tremolo-like effect, meeting this requirement.

B. Lessons Learned

A very unexpected issue we ran into was running out of FPGA area. None of us have ever had experience with such a large design, so had no heuristic to judge how much area we would need. It turns out that we needed a lot, as indicated in Fig. 9. In the future, we can hopefully make a more accurate estimate for the amount of FPGA space we require and how many features we can include.

In a hardware system this large, communication is incredibly important. Our largest bugs ended up being simple miscommunications in the ways we passed values from component to component. Clearer and more accurate parameterization, code readability, communication, and design documentation would have saved us many cycles of flashing our FPGA and trying to figure out why a bunch of simulation-verified modules do not work in conjunction with each other during synthesis.

Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	34,611 / 41,910 (83 %)
Total registers	11781
Total pins	103 / 499 (21 %)
Total virtual pins	0
Total block memory bits	20,480 / 5,662,720 (< 1 %)
Total DSP Blocks	99 / 112 (88 %)

Fig. 9. Quartus DSP and logic utilization area

VIII. 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
[8] MIDI Tutorial, Sparkfun

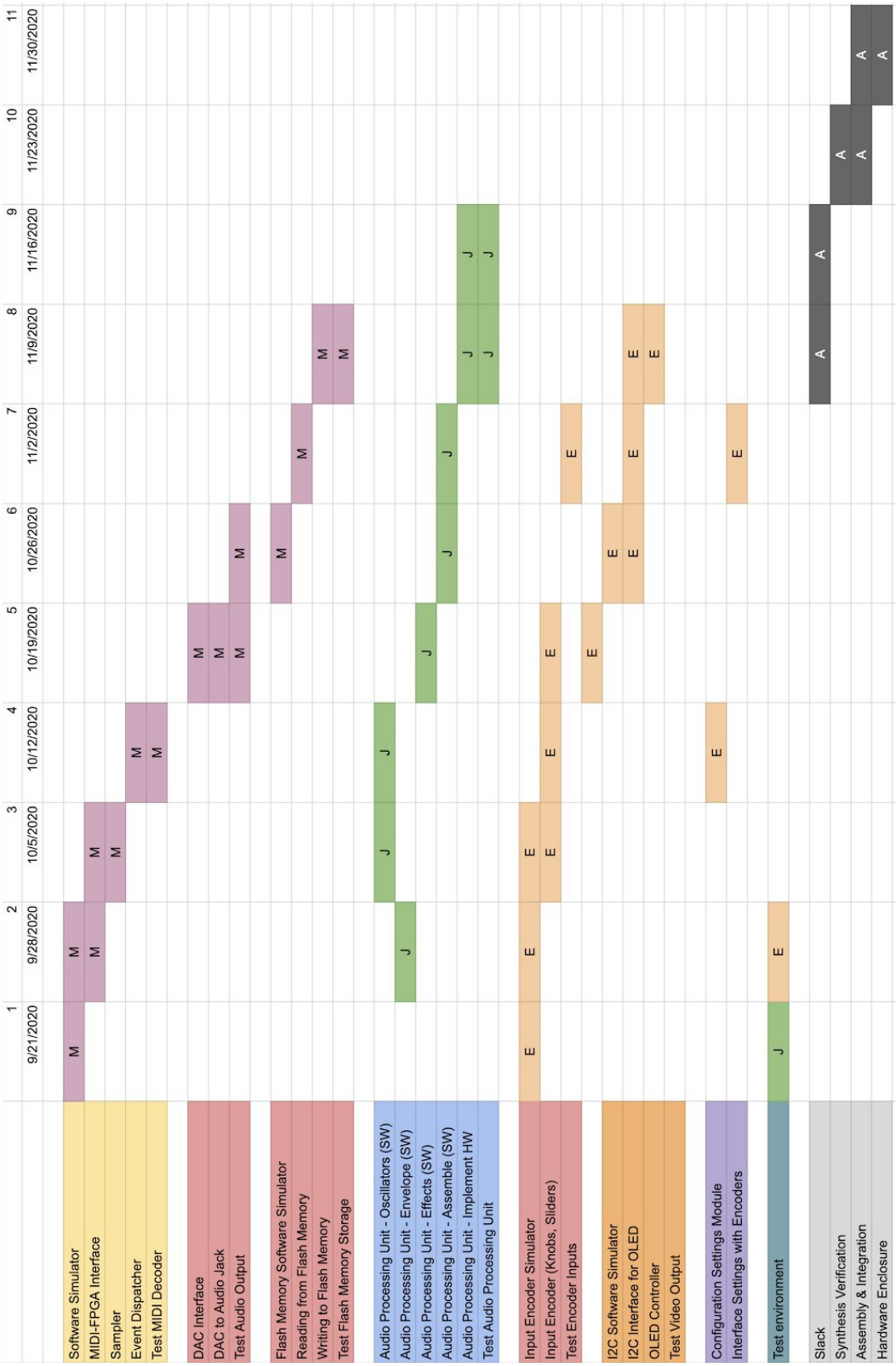

Fig. 10. Schedule and division of labor

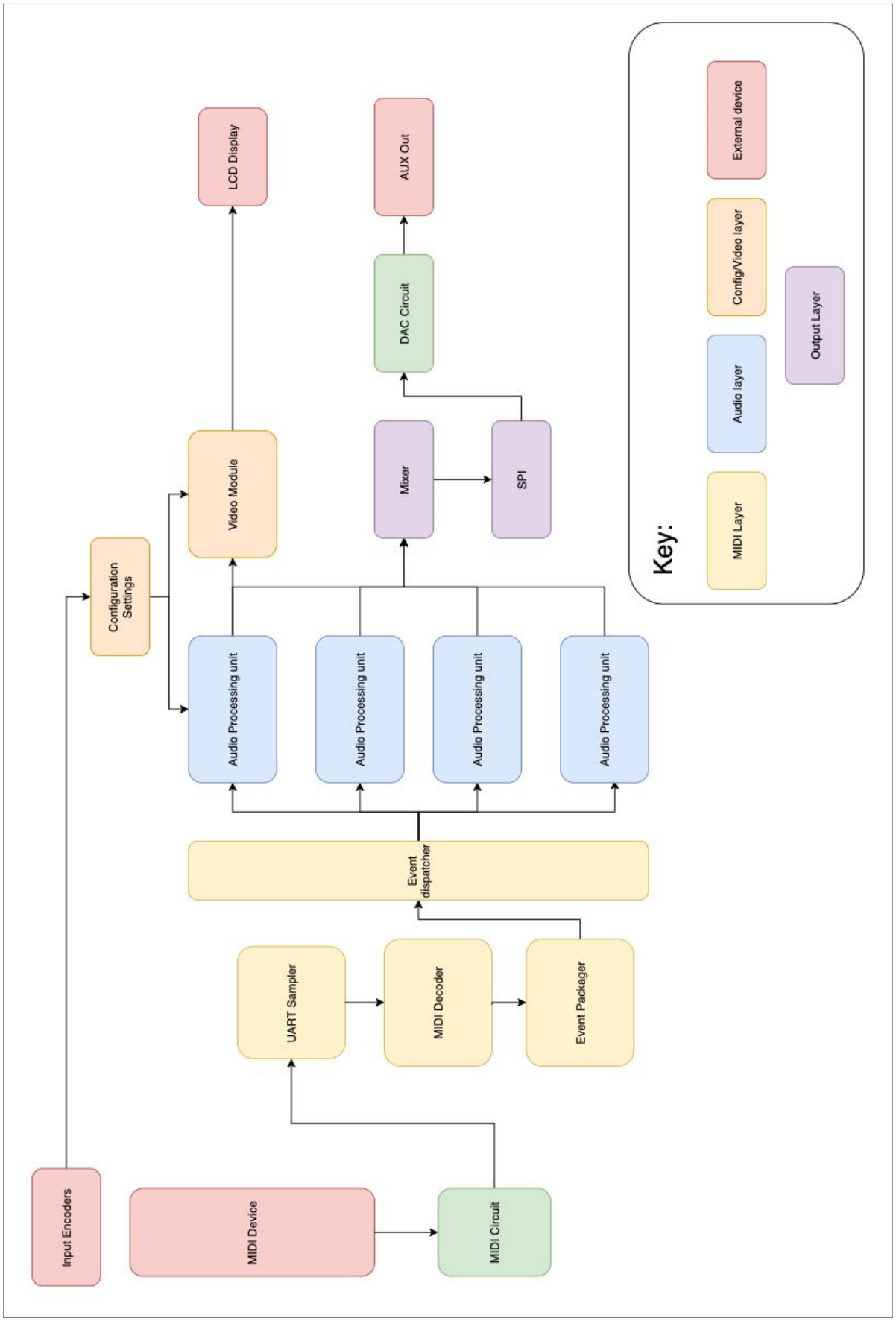

Fig. 11. Top-Level block diagram

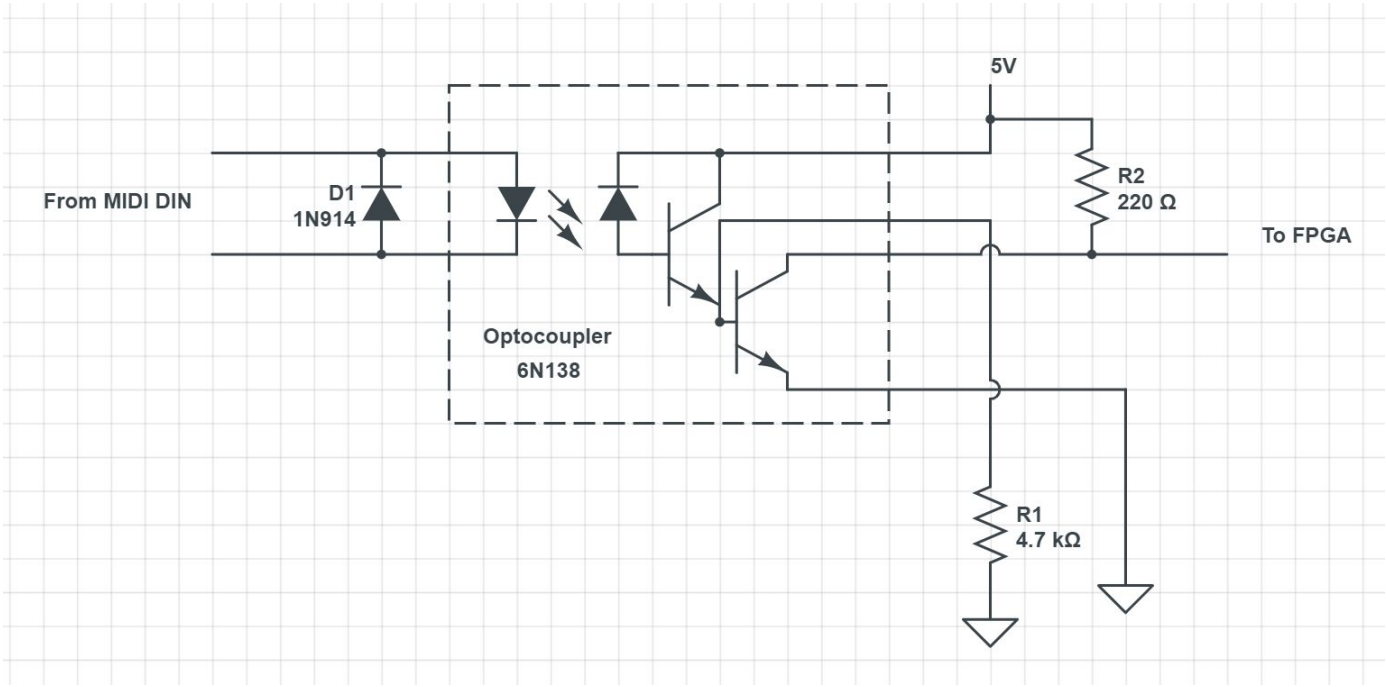

Fig. 12. MIDI Input Circuit

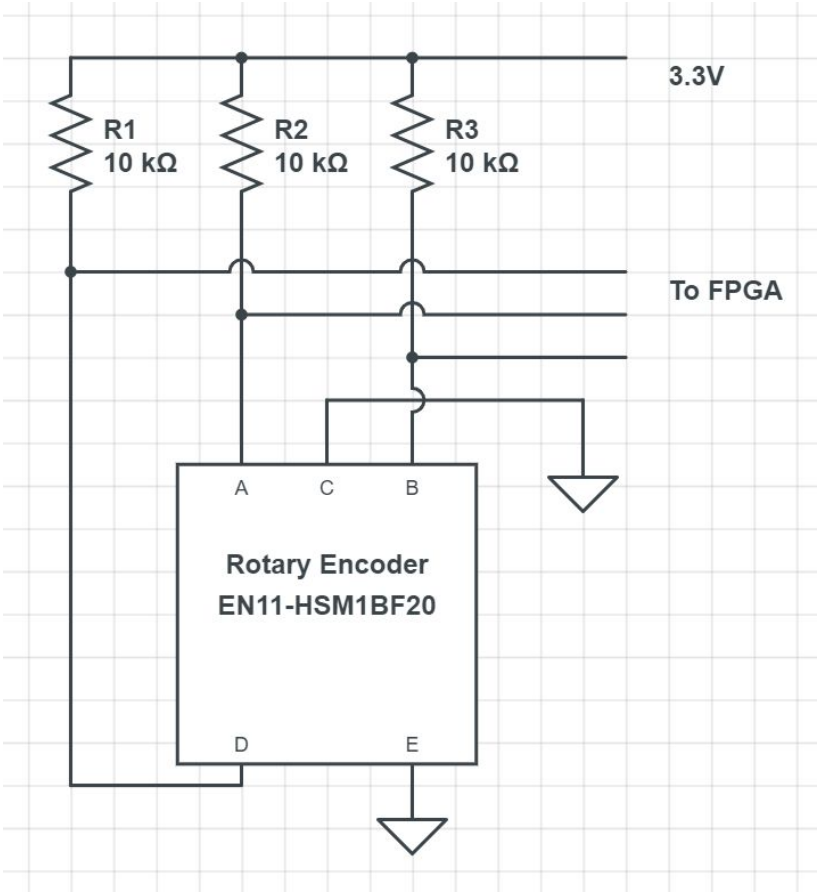

Fig. 13. Encoder Input Circuit