FPGSZR

Authors: José Cestero, Kaloyan Stefanov, Mihail Alexandrov
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Modern day music production is a rich field with many different options for creating and playing. One of these is the hardware synthesizer, a musical instrument capable of generating musical notes through oscillators, filtering, and effects. However, a capable hardware synthesizer with these features and an additional drum machine built in is not cheap. Our group aims to solve this problem by providing a modern, capable digital hardware synthesizer with filtering, effects, wavetable sampling, and a drum track, to help lower the barrier to entry for new users who are interested in music production.**

***Index Terms*—Digital Design, FPGA, Hardware Synthesizer, Signal Processing**

1 INTRODUCTION

One of the most common tools in the production of music is the digital synthesizer. It allows a user to create, shape, and control sound from scratch in an easy to use manner. Such a pivotal item should be readily accessible, yet many mid-range hardware synthesizers carry an average cost of approximately $500 [8]. This creates an economical barrier for many trying to enter the field of music, and limits the amount of people who can express their culture through the medium. With that in mind, FPGSZR aims to provide a lower-cost alternative to a traditional synthesizer, replicating many of the features that are standard for such products and allowing people of all economical and social backgrounds to express their musical desires.

As the name suggests, FPGSZR is a synthesizer with an FPGA (Field Programmable Gate Array) as the core component for computation and interfacing. While we aim to replicate the standard synthesizer experience with a MIDI keyboard for input control, all of the actual processing is meant to be done on the FPGA. Because of its relatively low cost and set of features, this is a device targeted to people from the beginner to intermediate skill levels in music production. It is meant to be easy to configure, requiring no previous hardware or musical experience. Moreover, by having a standard set of filters and effects, a novice could learn and improve their production skills while a more intermediate user could create impressive productions on par with those that come from mid-range synthesizers.

There are two main types of synthesizers available on the market, being software synthesizers and hardware synthesizers. The former tend to be cheaper than dedicated hardware, but require a good computer and at times complex technical setup. Moreover, interfacing needs to be done through keyboard and mouse if the user does not have a keyboard accessory in their possession. Hardware synthesizers, on the other hand, allow for a portable, immediate, and tactile experience but come at a much heftier cost (as previously mentioned). So, while existing solutions do partially address the aforementioned need, they come with their own significant drawbacks. With that in mind, our setup aims to replicate the ease of use and tactile interface of a hardware synthesizer and the cost effectiveness of a software synthesizer.

2 USE-CASE REQUIREMENTS

As the average user of our system ranges from beginners in the field of music to those with more experience, our baseline user requirements can be defined through four main rules. First, the user would want a system that responds quickly to any sort of input. In the case of complex inputs to the MIDI keyboard, slow response time being noticeable can ruin the user's flow and general experience. Second, there needs to be accuracy with the notes being produced. Just like with a piano, the average user would notice if the key being pressed differs to the one being outputted. Not only would it ruin their experience with the product, but it also shows that it is not reliable for producing music. Third, the output generated by the synthesizer should be of high quality. Since FPGSZR natively generates audio output, it is our responsibility to assure that quantization and aliasing is minimized as much as possible. Finally, the system should be as easy to use as possible, with minimal setup time and techincal knowledge necessary to operate the system. By doing this, we ensure that the system has a low barrier to entry, making it accessible to beginners. Using these four rules, then, we can derive the following user requirements:

2.1 End-to-end Latency of Less Than 5 ms

As previously described, the average user of the FPGSZR should expect fast response times once a key is pressed. That is why we're aiming for the time between a key press and the audio output of such key to be less than 5 ms. We had originally planned for a 10 ms latency ceiling, since that generally the lower limit on what an individual can detect in terms of delay [1], but upon further research, discovered that those particularly sensitive may detect latency within the 5 to 10 ms range. Thus, we decided to tighten our goal. This is a stricter timing requirement when compared to devices for other fields, yet this stems from the fact that performers use auditory feedback for timing. If the system is too slow, the user's timing could get disrupted. This means that this number should not be

exceeded to allow for bare-minimum responsiveness. While strict, it is one of the most important requirements for the user.

2.2 SQNR of More Than 40 dB

SQNR (Signal-to-Quantization Noise Ratio) SQNR measures the ratio of the signal power to quantization noise power. If our SQNR value is low, it would result in our system having audible digital artifacts or a noticeable hiss accompanying the generated sound. Usually, this is a result of arithmetic imprecision stemming from accumulating rounding errors due to inadequate signal representation or calculations. A SQNR of more than 40 dB would reflect a system in which these artifacts are inaudible to the human ear, making them appear as if they did not exist.

2.3 Four Voice Polyphony

Since we're striving for noise accuracy with our synthesizer, we would want the case in which more than one key is pressed to be outputted accurately. That means that the output of more than one key should be the sum of such. This can only be done if our system has a high count polyphony like an four voice polyphony, which signifies that four notes can be simultaneously processed and outputted together. In the musical sense, polyphony allows for proper note release behavior. This means that when a key is released, the sound it produces gradually lowers and does not halt immediately. If the synthesizer was lacking voices, it would need to cut off these notes to make space for newer ones (resulting in what is called "voice stealing"). Something like this occurring would cause the system to produce an "unnatural" sound, which would take away from our goal of an accurate user experience.

2.4 5 Cent Frequency Difference

A cent is defined as 1/100 of a semitone, wherein humans can begin detecting pitch errors from 5 cents. If our system contains a frequency deviation of more than 5 cents, the difference will become perceptible to trained listeners and cause chords to sound out of tune. By reducing this difference and focusing on this metric, we're able to achieve the most accurate sound possible. Users for this product can vary greatly, yet even a beginner would notice that something is off with our product if we are careless on this front.

2.5 System Weight Under 7 Lbs

A lightweight interface for transport is ideal given that the system may be moved between studios, venues, and performance venues. Keeping the system transportable reduces the physical stress of carrying it around, allowing individuals to frequently relocate the system. A lighter form also encourages more spontaneous use, by reducing the barrier to grabbing the instrument and taking it with a user, the synthesizer becomes a practical tool that can be used in collaborative sessions and informal settings. Thus, we are aiming to keep the system weight under 7 lbs in order to make use as flexible as possible, catering to the wider customer base.

2.6 Short Setup Time

A setup time of under five minutes is critical for the system to be usable for live performances and collaborative use. An overly long setup process / time impinges on the systems intended use case, increasing the barrier to entry for a product that is expressly meant to be accessible. Keeping the setup time under five minutes means the instrument behaves more like a keyboard / effects unit rather than a development platform, users power on the device, load the qpf file, and play. This threshold also better allows a user to focus on the act of actually playing, rather than having to troubleshoot boot sequences. Hence, we aim to keep setup time as low as possible, with five minutes being our ceiling.

3 ARCHITECTURE

3.1 System Description

The system is encapsulated within a custom 24"×12"×6" laser cut wooden enclosure. For input, the system has six Modulino knobs [ABX00107] labeled Attack, Decay, Sustain, Release, Filter Select, and Filter Value. The first four are used to modulate the ADSR envelope parameters, while the other two are used to traverse the system's LCD menu and adjust parameter values depending on which screen the user is in. The system also has 8 buttons, which are used to select drum samples and program drum patterns. To reflect the current state of the system, we use the previously mentioned LCD display to provide a visual for what mode we are currently operating in. These modes are ADSR control, effects control, wavetable select, filter control, drum sample selection, and drum pattern programming. As the user cycles through the different modes, the LCD display updates accordingly.

These inputs are all handled by an Arduino. We opted for an Arduino (as opposed to directly processing them on the FPGA) due to the ease of use and increased I/O it grants us. Through this, the engineering effort to setup the various input interfaces and create an intuitive UI was drastically reduced. The Arduino handles these inputs in different ways. For the buttons, it uses digital I/O to detect switch signals and output a set voltage for LEDs that show if it is set to high or low values. For the knobs, the Arduino receives data from all of them through I^2C, with each knob having a unique address set by us. Finally, the Arduino receives keyboard data through a SparkFun MIDI board interpreter. When the keyboard sends out data through MIDI DIN, the board outputs it through hardware UART for the Arduino.

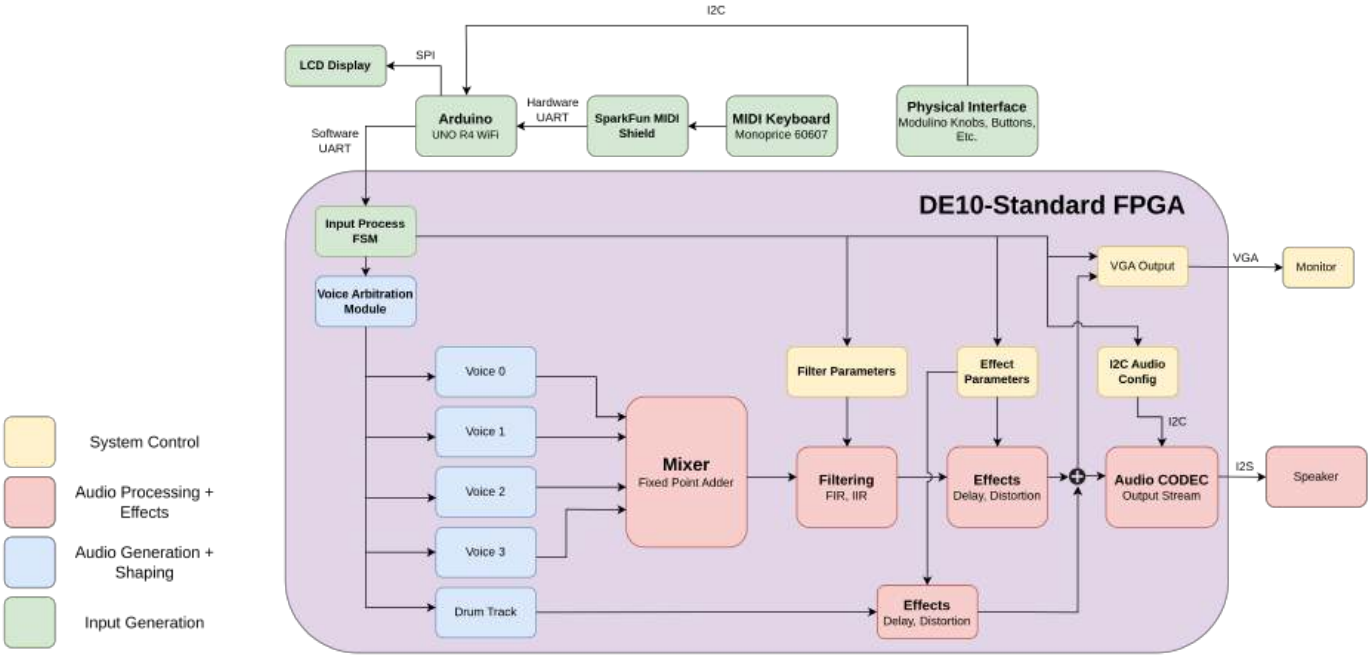

Figure 1: Complete block diagram for our synthesizer.

Once these inputs are processed on the Arduino, it sends them out to the FPGA over software UART. The FPGA receives this stream of input and sends out proper control signals based on an internal FSM. These various control signals include MIDI note values, whether the key is pressed, ADSR values, filter values, effect values, wavetable selection, the drum sample to be selected, and the drum pattern to load in for the currently selected sample.

With these control signals, the FPGA has everything it needs to generate the appropriate audio response. Within the FPGA, we have different modules to properly deal with these control signals. Note values and keypresses get processed through the voice arbitration module, which is a simple round-robin arbiter that assigns note values and keypresses to available voices. Within each of these voice modules, we use the wavetable select to determine which wave to output (sine, square, saw, or triangle). We then use the ADSR values the user has provided to apply an amplitude envelope to the raw wave output (Figure 4). Since our FPGA implementation makes use of Q1.23 fixed point values, combining the output from the 4 voices is simple: it's just a fixed point adder, that then gets shifted back to 24 bits.

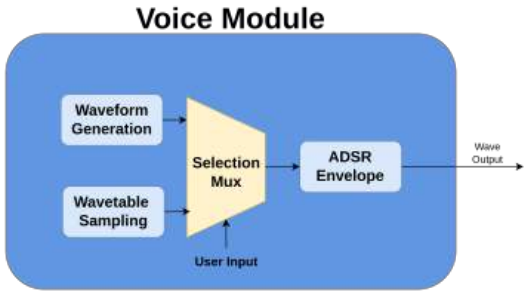

Figure 4: Voice module diagram.

The filter component will take the wave output from the ADSR envelope and uses cutoff and gradient filters supplied by the user to attenuate the inbound signal. Notes within the pass band are allowed through unaltered, while notes further away are progressively attenuated as the distance from the cutoff value increases.

After filtering, the next stage of our audio processing pipeline is effects. Here, we can add delay and distortion to the incoming filtered signal. The delay component will store prior samples, reading them out and adding them to the current wave input based on user-set parameters. Distortion takes in an audio signal as input and then, based on a saturation curve, distorts it to output a grittier sound.

As the audio of our synth is getting processed, our drum track is working in parallel. Through a simple sample select and drum pattern, we can generate complex rhythms to accompany any keyboard playing. We provide 8 stored samples for the user to utilize, which can be loaded into any arbitrary 8-step pattern. Additionally, we offer adjustable BPM control to give the user control over the speed at

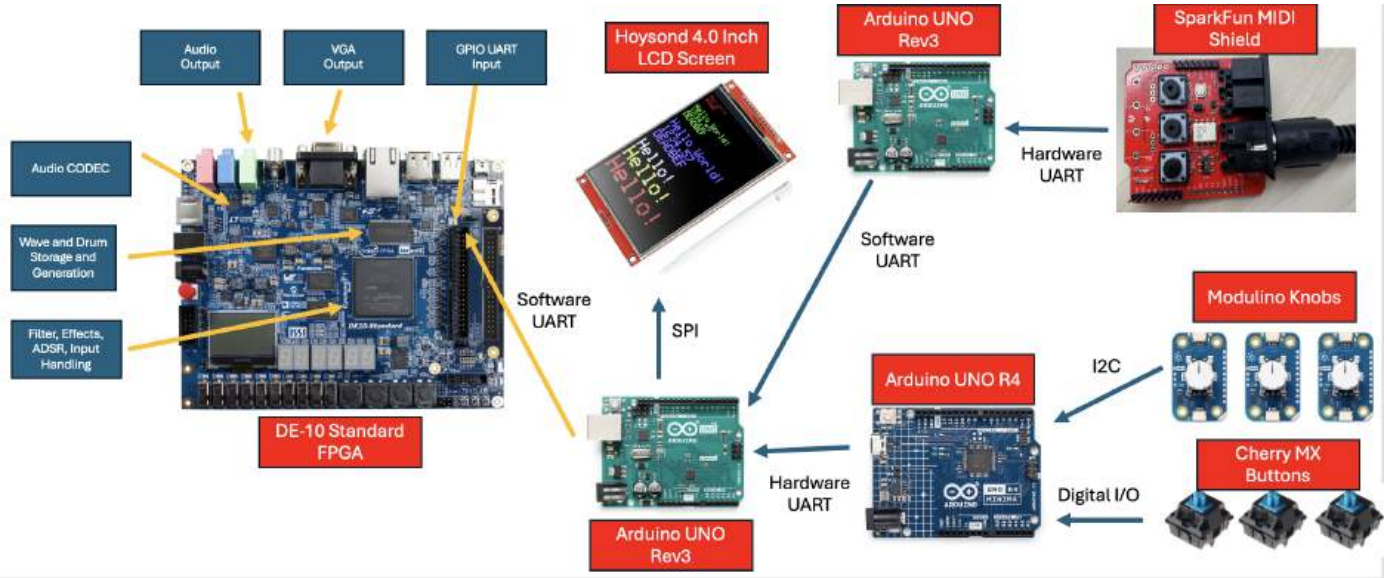

Figure 2: Internal physical system diagram

Figure 3: Exterior physical system diagram

which the pattern plays. We use this to generate the current beat, and then add up the audio output from all the patterns in another fixed point adder (since our drum samples are also in Q1.23 format) for our final drum output.

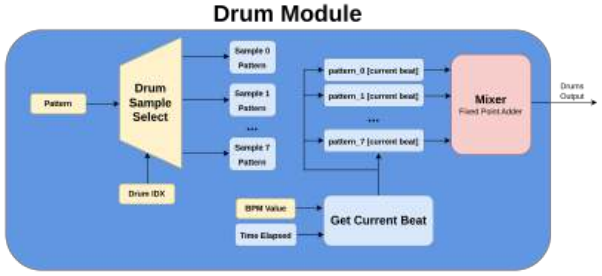

Figure 5: Drum module diagram.

Finally, we add the two audio outputs from our synthesizer and drum track and push it out to the audio CODEC, which handles the digital-to-analog conversion. It outputs this over I2S to a standard 3.5mm audio jack. For additional visuals and debugging purposes, we provide a VGA display that displays the current ADSR envelope. Compared to our initial system described in the design report, we have chosen to instead use an Arduino to offload the input processing that would have gone to the FPGA. Additionally, we decided to build a chassis and provide a more intuitive UI to make the user experience better. To focus more on this user experience, we decided to not implement reverb, multi-band compression, and FM.

3.2 Principles

In our project, we used various principles of engineering. One principle that enabled us to work well together and accomplish our goals is breaking the larger system into smaller subsystems. By assigning things like the wavetable generation, filtering, and MIDI processing to different group members, we were able to develop in parallel. Although there are some blockers that inevitably exist (it is difficult to test the drum pattern when there is no UI to support it), this still allowed us to move much quicker. We also made use of another principle of engineering: simulation and modeling. Simulation and modeling are pivotal in FPGA development (through testbenches and logic simulation), so this principle aligned naturally with our project for testing the output from our various RTL modules.

We also used different principles of science. In particular, we initially attempted to make use of Fourier Transforms to decompose our audio into its various frequencies. We would then use this output to create an interesting visualization on the VGA display for the user. Unfortunately, this involved either using an Intel IP or open-source FFT which added too much complexity to the processing and output for VGA. We also made use of the Nyquist-Shannon theorem to determine what frequency our audio should be outputted at. Since we want to achieve 20KHz audio reconstruction (since this is the basic limit of human hearing [9]), we had to output audio at 48Khz and properly constrain our design to meet this (albeit lax) requirement.

Finally, we also made use of various principles of math. In particular, we used fixed point arithmetic to represent our audio in an efficient and expressive manner. Since our audio CODEC utilized 24-bit, Q1.23 format audio input, we adopted this same format throughout our design. This circumvents the difficulty of implementing floating point operations for a minimal loss in audio accuracy. For our filtering, we also used a mathematical formulation to determine how to filter the input. Our filter operates using difference equations, which smooths out high frequency differences between subsequent samples, allowing us to create a low-pass filter.

4 DESIGN REQUIREMENTS

4.1 24-bit Signal Representation

The way we represent our signals can affect audio quality, processing time, and much more. In order to meet our desired goals, we had to adhere and achieve a specific signal representation. Specifically, we aimed to and implemented a 24-bit fixed-point signal representation. This means that every signal is an array of 24-bit fixed-point values, wherein the most significant bit is a sign bit and the other 23 serve as fractional bits. The representation results in our system operating on rational values from [-1, 1), with steps of 2^{-23}.

4.2 Input and Processing Time Latency

Because we are targeting an overall end-to-end latency of 5 ms (from when a key is pressed to when it is outputted), we naturally wanted to limit any sort of unnecessary delays to allow as much slack as possible for the signal processing time. Therefore, we worked to achieve a maximum end-to-end latency of 2 ms for the MIDI keyboard input. This would allow, in the worst case, 3 ms for the FPGA to conduct any and all digital signal processing work. The following relationship can be derived based on the specification as such:

$$\Delta t_{\text{total}} = \Delta t_{\text{keyboard}} + \Delta t_{\text{processing}}$$

4.3 Audio Output

Our use case requirements specify that audio output needs to be of high-quality (apart from being correct). With that in mind, we aimed to achieve a 24-bit audio output stream which should operate at a 48 kHz sampling rate. This requires the FPGA to produce a hardware-generated clock that operates consistently, since timing variation could drastically affect the quality of the 24-bit output and possibly cause loss in resolution.

4.4 Polyphony

As mentioned in our use case requirements, we were trying to achieve a four voice polyphony with our system. Yet, that would require simultaneous processing of four separate voices that could create delay within our system and affect latency if not designed carefully. We did not want playing multiple notes to feel or sound different than playing one note, thus we decided to maintain the 5 ms latency window with polyphony.

5 DESIGN TRADE STUDIES

5.1 Phase Accumulator Resolution vs. Pitch Accuracy

To determine the frequency and pitch resolution that our phase accumulator can achieve, we must inspect the relationship between its resolution and the finest output frequency we can produce. Our output frequency is determined by $f = \frac{\Delta\phi \cdot f_{clk}}{2^N}$, where ϕ is our phase increment (which is changed based on the inputted MIDI note) and N is the bit resolution of our phase accumulator. Intuitively, this equation is calculating the outputted frequency by finding how long it takes for a given phase increment at a particular clock frequency to complete one full cycle of the waveform, or wrap around from the 2^N value. To find the smallest step change in frequency, we just set $\phi = 1$. We then get $f = \frac{1 \cdot f_{clk}}{2^N}$. Inputting the values from our synthesizer, we see that $f = \frac{50 \cdot 10^6}{2^{32}} \approx 0.0116$ Hz. To then determine the theoretical pitch error, we use a formula for the difference in pitch (cents) between two frequencies.

The lowest note we support is C-1 (8.176 Hz). Since our maximum possible error is when we fall exactly between two of the discrete frequency steps, our worst case error is $0.0116/2 = 0.0058$ Hz, so the corresponding pitch difference in cents is $1200(\log_2(\frac{8.176_0.058}{8.176})) \approx 0.00123$ cents. This gives the theoretical bound for what our worst case pitch error is, which is far below the 1.5 cent error that we initially set. This makes it clear that a higher bit resolution is not necessary and that we have properly sized our phase accumulator. Although our actual implementation error has an average error of 0.33 cents, this is likely due to further quantization and noise artifacts introduced from our audio pipeline and in the ADC.

5.2 MIDI 5-Pin DIN vs. MIDI USB Latency

The MIDI keyboard has two different options for connection that had to be considered: 5-pin DIN MIDI and USB. The former is much slower than USB, being dominated by UART serialization at 31.25kbps [10]. Meanwhile, USB times consist of both transmission times and scheduling times, making it such that:

$$T\_usb = T\_trans + T\_sched \qquad T\_midi = \frac{10B}{(31.25)(10^3)}s$$

For every byte that needs to be transmitted over MIDI, we need to send 10 bits. This is because, apart from the 8 data bits, a start and stop bit needs to be sent out as well [10]. For a standard 3 byte MIDI "Note On" or "Note Off" message on 5-pin DIN MIDI, we'd see that:

$$T\_midi = \frac{10(3)}{(31.25)(10^3)}s = 0.96ms$$

This method of communication is deterministic due to its use of the UART protocol, so we can confidently say that this is the amount of time it would take to transmit the 3 bytes. USB, on the other hand, is non-deterministic because of its reliance on scheduling and polling [7]. While T_trans will always be in the realm of microseconds, T_sched can and will likely be more than 1 ms, creating an overall transmission time of:

$$T\_usb > 1ms$$

This leads to 5-pin DIN MIDI having a much more reliable transmission time over USB. Coupled with its much simpler communication protocol and setup (since USB would require dealing with drivers and scheduling), this method of communication was chosen as the more desirable one for our project.

5.3 FPGA Choice

One of the most important decisions we had to make when beginning to design our project was which FPGA we were going to use as the center of all our computation. With that in mind, we ended up going with the DE-10 Standard FPGA. It provided a substantial amount of DSP blocks which allow us to perform quick computations with our 24-bit fixed-point numbers. Because its DSP adders and multipliers allow for single cycle operations, we were able to minimize our processing latency as much as possible. Moreover, it includes a built-in WM8731 audio CODEC that allowed for our desired 24-bit audio output. Apart from meeting our specifications, it being built-in allows for us to have a centralized system that minimizes set up complications for the average user. Moreover, its high amount of block ram units and look up tables would let us store all of the necessary components for our desired wavetable samples. Finally, this FPGA and the toolchain it uses is the one that all of us have the most experience with from our digital design courses. This means that we were able to avoid the need for any onboarding and begin working on the different portions of the project as quick as possible.

6 SYSTEM IMPLEMENTATION

6.1 User Interface

The user interface for our system was designed with all the different filters, effects, and general options that the user would have available to them when using our product. It has eight buttons for handling drum tracks, six knobs for ADSR, and an LCD display for menu handling and being able to see what values are being used for particular subsystems. To house all of this, we created a chassis using the laser cutting machines available at TechSpark, taking careful measurements of each component to make sure it would all fit properly.

The eight buttons were Cherry MX keyboard buttons, and we used the SparkFun Cherry MX breakout board to interface with them as traditional buttons. These would get wired into one of the Arduino's digital I/O ports, where we would then keep track of the state of the buttons with an 8 bit vector (where each bit represents one of the buttons). Knob communication was handled differently, as our choice (Modulino knobs) required communication through I2C. This meant that each knob had to be assigned a unique address, which would then be daisy chained to every other knob. These would be connected to the same Arduino that was handling the drum track buttons and then be forwarded to the FPGA through UART.

MIDI keyboard input was handled in a similar matter, with the SparkFun MIDI Breakout Board– which converts the keyboard's 5-pin DIN MIDI output into a logic-level asynchronous signal–being connected to another (separate) Arduino. Said device was then programmed to use a dedicated MIDI library to read in input at 31250 baud and forward it to the FPGA (at the same time as the buttons and knobs) through UART.

6.2 FPGA Input Processing

The FPGA would receive input from all parts of the user interface through UART. While MIDI input would be processed through an Arduino, we set it up so that it and all types of inputs (knobs, buttons, etc.) would operate under a fixed communication protocol. We would first send out a status byte (from the Arduino connected to the FPGA), with either one or two extra bytes of data. These are outlined below as follows:

Message Type	Status Byte	Data Byte 1	Data Byte 2
MIDI NoteOn	0x90	Note Number	Note Velocity
MIDI NoteOff	0x80	Note Number	Note Velocity
MIDI KnobVal	0xB0	Knob Value	Knob Channel
Modulino KnobVal	0xC0	Knob Value	Knob Channel
Button Status	0xE0	Button Bitvector	Beat Set or Sample Set

Table 1: Interface Communication Protocol

Whenever a modulino or MIDI knob value would get set, we would then have the FPGA case on the specific channel of the knob (which is how they would each be differentiated), and then depending on the channel we would have a specific value changed. For example, if the knob value message sent was in the channel assigned to a wave's attack value, the value attached to it would be assigned to the eight bit attack register.

6.3 Voice Arbitration

Voice arbitration was achieved using a simple round-robin arbiter. Once the appropriate note value and note on MIDI signal were sent, the arbiter would assign a free voice with the correct note value and register a keypress. Once the note off signal was sent, it would disable the keypress (preserving the ADSR functionality). If more than 4 keys were pressed at once, the least recently used voice would be overwritten.

6.4 Voice Module

The voice module utilizes wavetable sampling and generation to generate the audio. Each voice module uses a phase accumulator, an incoming note value, and a keypress signal, along with the current values for attack, decay, sustain, and release. When a keypress is registered, the phase accumulator uses the current note value to determine its new phase increment. This phase increment is then added whenever a 48 KHz strobe goes high, to match the sample rate of our I2S audio output module. This generates the appropriate frequency of the inputted note, allowing us to play a variety of pitches with different waveforms. The sine wave is generated using a LUT, since generating it using the appropriate equation would've used up too many DSP blocks with little benefit. The saw, triangle, and square wave are generated using the phase accumulator values. For example, the square wave can be generated by monitoring the MSB of the phase accumulator value. Then, this raw output is fed through the ASDR module. This module generates an envelope based on an attack, decay, sustain, and release. Through these values we can achieve 5 distinct phases, which we reach either when a dedicated phase has finished its allotted time, or when a keypress is disabled and we enter the release state. The raw output is multiplied by this envelope every 49 KHz tick.

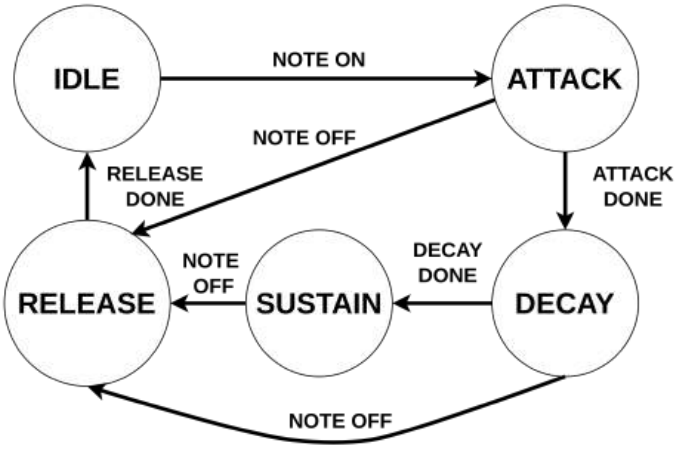

Figure 6: FSM for the ADSR envelope

6.5 Drum Module

The drum module (see Figure 5) utilizes an incoming pattern, drum index, and BPM value to generate rhythm. The user can use any of the 8 pre-loaded samples, in any arbitrary 8-step pattern, to create drums to accompany their playing. The current 8-step pattern (done on the physical buttons) is loaded in for the currently selected drum sample. A BPM calculatoin determines whether or not to advance to the next beat. If a saved pattern has a trigger on the current beat, the sample is read out of BRAM and outputted alongside any audio output from the synth. Difficulties with the SystemVerilog language plagued us here: we tried using more advanced language features (like named structs for parameters) which were synthesized incorrectly, causing a difficult bug.

6.6 Filters

The filter module takes five parameters, those being the current note being played, a low cutoff, a low gradient, a high cutoff, and a high gradient. Based on these values and the current input wave, the module then computes whether the note is in the pass band, and if not, the distance between the current note and the nearest cutoff Δ. All but the first one of these parameters is controlled by the user, which they may set through the knob interface available on the enclosure. Once Δ is computed, the corresponding gradient is selected, with a low gradient being used if the current note is less than the low cutoff, and a high gradient being used if the note is above the high cutoff. From there, the input wave is divided by $g \times \Delta$, where g is the corresponding gradient, to generate an output waveform.

6.7 Saturator

The saturator is implemented using a simple saturator LUT with a pre-drive, configurable gain. We use the current audio amplitude to index into a saturator curve, which pushes the amplitude into a louder region. This can create more grittiness and distortion, producing interesting sounds. The pre-drive gain is used to "tune' the saturator, allowing the user to push more or less of the audio into a higher amplitude range.

6.8 Delay

The delay component implements the following equation:

$$y[n] = x[n] + g \cdot y[n - D]$$

Where D is the number of sample ticks to wait, and g is a feedback coefficient. The module implemented on the FPGA takes in the values of D and g, which are managed by the top level MIDI FSM that processes inbound IO. The FIFO itself is capped by chip memory, with the design having a depth of 1024 samples. This determines the maximum value of D. The module maintains an incrementing pointer to the FIFO, which is not pictured in the block diagram for simplicity. The pointer, along with D, determines the read and write addresses. On each sample tick, a value is read from the buffer, multiplied by the feedback coefficient g, and the 24 most significant digits are added to the inbound wave. The outbound signal is then written to the buffer.

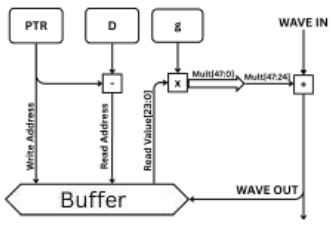

Figure 7: Block Diagram for the Delay Component

6.9 LCD Display

The LCD display is connected to and driven by a secondary Arduino that communicates with the primary Arduino responsible for interactions between the FPGA and user interface. The primary Arduino informs the second about any changes to filter values, while the secondary maintains internal counters identical to those within the FPGA. At any point, the secondary Arduino tracks the current effect that is being altered and pending changes to parameters, writing them out every 150 milliseconds to avoid rewriting every time a value is updated. Internally, every character the Arduino can write to the driver is represented as a 6×7 2 dimensional bit mask, with pixels updated in order. On startup, the system is programmed to show the ADSR values and then changes according to inbound IO.

6.10 Audio CODEC

The FPGA communicates with the audio CODEC through two independent digital interfaces: an I^2C interface for control and an I^2S (Inter-IC Sound) interface for audio output. The former is used only in system initialization, where a serial clock line (SCL) and bidirectional data line (SDL) send out a register address and its corresponding data to be written. Through this, the interface is able to reset the CODEC, set up the desired sampling rate, left and right volumes, communication format, audio format, and activate the device. Once the CODEC is initialized properly, audio transfer begins to occur exclusively through the I^2S interface, wherein our finalized audio samples will begin to stream with the use of a bit clock (BCLK), a word selection signal (LRCLK), and a serial data line (SDATA). Afterward, the CODEC will natively convert the digital samples into analog audio outputs.

6.11 VGA

The VGA output displays a live visual of the ADSR envelope. Through using static lines (which can be evaluated at compile time, thus saving on DSP blocks) and dynamic lines (dedicated to the ADSR envelope), we were able to create a resource efficient implementation to provide visual feedback to the user. We also had plans of adding an oscilloscope (tracking the output of the synthesizer), but integration proved to take too long and we never managed to fully test it and ensure it worked properly.

7 TEST & VALIDATION

A three-level testing strategy was employed to verify the correctness and performance of the FPGA-based synthesizer system. We performed tests at the module, subsystem, and system levels to ensure that individual design components function correctly, that interfaces between components operate as intended, and that the complete system meets use-case requirements.

At the lowest level, we verified individual RTL modules via a combination of RTL simulation and comparison to results from Matlab and python scripts that we wrote. Certain sub-components, such as the FSM defining the ADSR envelope, have defined behavior that were easily tested within SV testbenches, with assertions being of key use. On the other hand, the output of blocks, especially those describing audio signals, were better tested against external scripts. In particular, we tended to use Matlab for comparison against ADSR and effects, while filters were

tested against a python model which modeled both floating point arithmetic and the fixed point arithmetic used on the FPGA.

On the subsystem level, we tested the MIDI keyboard via a combination of FPGA hex display and Sparkfun's Arduino MIDI Software. The amount of inputs is relatively limited, allowing a team member to visually verify the correctness of the system. Once audio output was configured, individual RTL subsystems such as as the voice modules and ADSR envelope were instantiated on a FPGA. From there, we either recorded audio output or collected output data through a dedicated cable and processing it with a spectrum analyzer before comparing against MATLAB scripts. The choice of which testing process to use was dependent on the subsystem in question and team member performing the test. The audio output itself was tested by writing trivial RTL modules to drive the system, synthesizing, and audibly ensuring that the output was what we expected.

Finally, system wide testing was performed by sampling audio output. The data was imported into MATLAB and processed, ensuring compliance with our use case requirements and allowing for us to ensure the system matched specifications.

7.1 Test for use case requirement 1

System latency was broken up into two components, MIDI end-to-end latency and on board FPGA latency. The former value, MIDI end-to-end latency, was tested by disconnecting the cable connected to GPIO on the FPGA and using an oscilloscope to measure the time between an initial MIDI bit being sent and the final bit being sent through the cable.

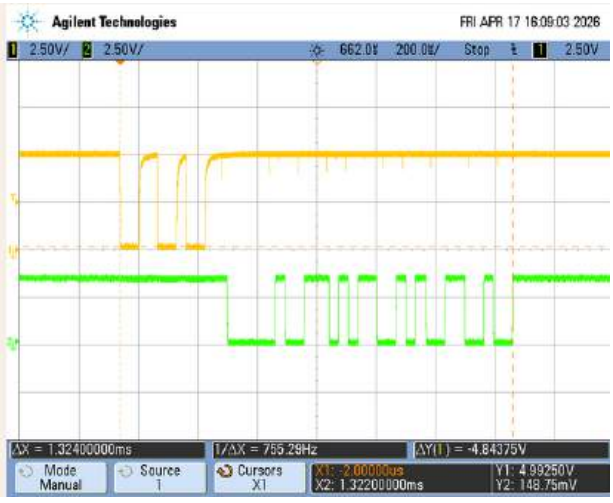

Figure 8: Oscilliscope Output for MIDI Latency Test

FPGA on board latency was measured by inserting a cycle counter into the FPGA code, and measuring the amount of cycles that pass between an input being registered and output data arriving onto the audio CODEC. Since the clock speed is known, the on board delay could be calculated as $\Delta t_{\text{FPGA}} = f_{\text{clk}} \cdot n_{\text{cycles}}$.

We measured a MIDI end-to-end latency of 1.324 milliseconds, a total latency of 2.426 milliseconds without polyphony, and a total system latency of 2.741 milliseconds with polyphony.

7.2 Test for use case requirement 2

The signal to quantized noise ratio (SQNR) of the system was tested by capturing the output of a voice module in a systemVerilog testbench. The data was written to a .txt file, and we extracted the metrics of interest using built in Matlab functions. Specifically, we were interested in a SQNR of at least +40 dB for all wave types, a Total Harmonic Distortion (THD) below -50 dB for sine waves specifically, and a Spurious Free Dynamic Range (SFDR) above +50 dB for sine waves specifically. THD measures the ratio between the amplitude of a pure signal and unwanted harmonics generated, while SFDR measures the ratio of the power of a harmonic signal with the power of the largest unwanted signal, or spur, generated. Since all wave types other than sine waves naturally contain harmonics, we decided that THD and SFDR made sense only in the context of sine waves. We measured the following results:

Wave Type	SQNR (dB)	THD (dB)	SFDR (dB)
Sine	+44.99	-55.29	+59.32
Square	+30.10	-6.31	+9.54
Saw	+25.33	-1.90	+6.02
Triangle	+108.37	-18.33	+19.08

Table 2: Signal Quality Metrics by Wave Type

7.3 Test for use case requirement 3

The target metric for audio accuracy was < 5 cents difference for supported notes, on average. In order to test the accuracy of this, we captured the audio output to the CODEC through a dedicated cable before preprocessing the data with a spectrum analyzer. The metric of interest, pitch error, could then be extracted in MATLAB. All notes measured were within 1.5 cents of the target value, and the average error was just 0.33 cents.

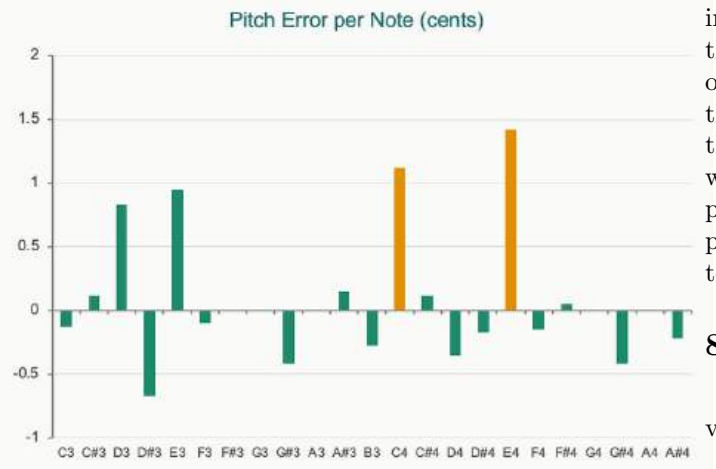

Figure 9: Test Results for Audio Accuracy

7.4 Test for use case requirement 4

Polyphony testing proved to be fairly straightforward, with our metrics of interest being functional correctness and impact on latency. In order to determine correctness, we outputted the current note being played by each voice line to the FPGA hex displays. Visual inspection of these values helped us ensure that the arbiter was routing incoming key presses correctly. From there, audible inspection was used to test that any four notes being played at a time could be heard. In order to determine impact on overall system latency, the test for use case requirement was performed again, with the results stated in section 7.1, but overall impact coming out to be negligible.

7.5 Tests for use case requirements 5 & 6

These tests were performed on the fully integrated system, assessing the usability of the system. To determine the weight, we used a scale, ensuring the system was under 7 pounds. We measured a total weight of 6.72 lbs. To check for setup time, we recruited volunteers individually from TechSpark, provided them with a basic set of instructions, and timed how long it took them to complete the setup on their computers. All participants finished in under five minutes, with the longest time recorded at approximately four minutes and thirty seconds, indicating that the system met our usability goal. The average setup time was three minutes and 16 seconds.

8 PROJECT MANAGEMENT

8.1 Schedule

Our schedule is outlined in section B of the appendix. It was structured such that the first tasks would focus on the creation of a system with basic signal audio output. And, by the interim demo, there would have been a basic set of features and interface functionality with the keyboard. Once that would be done, we would then able to move on into more complicated features, filters, and effects. After the interim demo, it was modified to add more emphasis on the chassis being built and setting up the interface for the integrated system (while we did have basic interfacing through the FPGA, it was not enough for everything we wanted to add to our system). All of this was meticulously planned to also allow time for the necessary status reports, presentations, and communication time necessary between team members.

8.2 Team Member Responsibilities

The responsibilities between team members were divided as follows:

Jose:

- FPGA Audio Output
- Keyboard MIDI Interface
- Chassis Building
- Arduino interfacing with buttons and knobs

Kaloyan:

- Waveform Generation
- ADSR Implementation
- 4-Voice Polyphony
- Distortion Effect
- Drum Track

Mihail:

- Fixed point arithmetic implementation and testing
- Delay Effect
- FIR & IIR Filters
- LCD Interfacing

Since Jose has the most experience with FPGA interfacing, most of the work he was assigned had to do with such. Mihail and Kaloyan, on the other hand, have more in depth knowledge of audio processing, which made them more fit to work on the digital signal processing side of the project. Once the features were all set, the plan was to have the team work together on connecting everything together for the two weeks before the final demo. It is important to note that while Mihail's parts all worked individually, difficulty with integrating everything with the UI made it such that his filters were not be able to be demoed properly. Nevertheless, everyone was able to get their individual work done on time.

8.3 Bill of Materials and Budget

Originally, we planned for our budget to not exceed above $500. Because the project was very FPGA centric, there was not much of a need for purchasing external components. However, the addition of a chassis for an external interface created the need for Arduino controllers, buttons,

knobs, and displays. Even then, though, we did not go over half of our budget estimate. The bill of materials can be seen in section A of the appendix.

8.4 TechSpark Usage

TechSpark was used for both sourcing plywood for our chassis and also laser cutting it with one of the CNC router machines present by the entrance. Additionally, gluing all the cut components together was done in the wood shop as José had access to it.

8.5 Risk Management

Our risk management strategy lied with the general methodology we used for our design. Apart from the pivotal interfacing and audio output, most of the effects and filters were modular components that would be attached to the system. This meant that, in the worst case that something did not work, we could take it out of the pipeline and modify it as needed without affecting the overall design's functionality. Moreover, by dedicating up to a week for testing components, we had enough time to stamp out possible issues and avoid general delays. With that in mind, all of the pivotal features did bring about the greatest risks. Jose had never dealt with audio output, so getting the CODEC to interface properly was the biggest risk. Without output, there would have been no FPGSZR to utilize. Moreover, keyboard interfacing was also an extremely important part of replicating the hardware synthesizer experience. This is why an apt amount of time was given for these tasks in order to ensure completely correct functionality. Our division of work, coupled with the amount of time given for testing, was meant to reduce as much overhead learning time as possible and allow for a proper implementation to be produced.

9 ETHICAL ISSUES

While an FPGA synthesizer does not seem to raise many ethical considerations at first, our team found several important factors to consider in terms of safety and broader societal implications.

From a safety perspective, the synthesizer's mostly digital implementation eliminates the majority of analog circuitry, and entirely eliminates any high-voltage circuitry. This significantly reduces the risk of electrical shock to users. Operating at the low voltages typical of FPGA platforms inherently improves user safety and makes the device suitable for both educators and hobbyists. Additionally, the deterministic nature of a digital system reduces the likelihood of unpredictable behavior that could pose safety concerns. One remaining health consideration is prolonged exposure to high levels of sound pressure; after discussing, however, we came to the conclusion that this must be addressed on a user level, as we cannot determine what audio system the output will go to.

Beyond safety, the project has implications for user welfare and accessibility. By separating the core audio processing from the input device, the synthesizer can interface with a wide range of external controllers, including customized or assistive MIDI devices. This flexibility allows individuals with physical impairments to adapt input methods to their needs without requiring changes to the FPGA internals, making music creation easier for the layman. As a result, the project supports inclusive artistic expression, making musical communities larger and more diverse.

The synthesizer is also designed with affordability and reproducibility in mind, which carries broader societal and economic implications. Traditional hardware synthesizers chains can be prohibitively expensive, limiting access to electronic music production. By contrast, an FPGA-based platform offers a low-cost, reconfigurable alternative that can be improved through firmware updates rather than hardware replacement. This not only extends the lifetime of the device but also empowers users to modify the system for different musical or their own applications and use cases. This also allows more individuals to engage with music production, further democratizing the space.

10 RELATED WORK

The idea of an FPGA based hardware synthesizer has been done several times as an 18-500 project. We have been using two of those projects as references, being "conFFTi" from Spring 2021 and "FMPGA" from Fall 2020 [2], [3]. The former has been extremely useful in determining our 5-pin DIN MIDI input approach and wavetable generation since both were accomplished successfully while the latter has served as inspiration for implementing specific filters and effects.

When looking at commercial implementations, René Garcia's DIY FPGA Synthesizers come to mind. They have a wide variety of features such as (at minimum) 32 voice polyphony, variable modulation algorithms, and monophonic/polyphonic portamento [4]. It is quite cost-effective, with an estimated BOM price of $15-$30. Even though it is not easy to set up for someone who is not experienced with FPGAs, it served as inspiration for what features we wanted to implement (if it was implemented by René, then it should be generally possible to do on an FPGA).

11 SUMMARY

FPGSZR is a hardware synthesizer with an FPGA as the center of its computation. It is designed to have accurate and high quality audio output mixed with fast response times. This is meant for those with beginner to intermediate experience with music such that anyone can start the system up and begin producing music. It receives note data from the 49-key keyboard through UART, processes multiple voices as necessary, filters and applies effects, and

outputs them through a built-in audio CODEC.

11.1 Future work

Future work on this project will most likely continue as a hobby, given that all group members share an interest in music production. Potential targets for improvement include increasing output audio quality, adding multiple oscillators, increasing the number of voices in polyphony, and incorporating additional effects. There is also potential to leverage FPGA GPIO to implement analog features as well, should we want to implement effects that work better in that domain. Overall, given that future work would be as a personal endeavor, we would likely focus less on minimizing resource usage.

11.2 Lessons Learned

Throughout the course of this project, several valuable lessons were learned. First, system integration consistently takes far longer than expected, and should be accounted for when planning a project timeline. It is also important to have a clear vision of a project going in, while still maintaining flexibility as the project is realized and new challenges arise. This is something we had to do several times as details of the system came into full view. Finally, it is sometimes necessary to think around a problem rather than attempting to brute force through it, as creative solutions can often prove more effective than sheer persistence.

Glossary of Acronyms

- ADC – Analog-to-Digital Converter
- ADSR – Attack, Decay, Sustain, Release (envelope generator)
- BLEP – Band-Limited Step
- BCLK – Bit Clock (I^2S audio clock)
- DAC – Digital-to-Analog Converter
- DSP – Digital Signal Processing
- FPGA – Field-Programmable Gate Array
- FIR – Finite Impulse Response (filter)
- FM – Frequency Modulation
- FSM – Finite State Machine
- I^2C – Inter-Integrated Circuit (serial control interface)
- I^2S – Inter-IC Sound (digital audio interface)
- IIR – Infinite Impulse Response (filter)
- LRCLK – Left/Right Clock (Word Select)
- MIDI – Musical Instrument Digital Interface
- SPI - Serial Peripheral Interface
- SQNR – Signal-to-Quantization Noise Ratio
- UART – Universal Asynchronous Receiver-Transmitter
- VGA - Video Graphics Array

References

[1] R. H. Jack, T. Stockman, and A. McPherson, "Effect of latency on performer interaction and subjective quality assessment of a digital musical instrument," in *Proceedings of the Audio Mostly 2016 Conference (AM '16)*, Norrköping, Sweden, 2016.

[2] H. Zhang, J. Zhang, and M. Chen, "conFFTi: FPGA-based music synthesizer," Carnegie Mellon University ECE Capstone Project, Spring 2021. [Online]. Available: `https://course.ece.cmu.edu/~ece500/projects/s21-teamd8/`

[3] E. Schneider, M. Trivedi, and J. Finn, "Fmpga: Digital audio synthesizer on an FPGA," Carnegie Mellon University ECE Capstone Project, Fall 2020. [Online]. Available: `https://course.ece.cmu.edu/~ece500/projects/f20-teama4/`

[4] R. Ceballos, "Futur3Soundz: Pro sound synthesis on FPGAs," futur3soundz, 2019–2026. [Online]. Available: `https://www.futur3soundz.com`

[5] J. Brown, "An Introduction to I2S Audio," Futur3 Electronics, 2018. [Online]. Available: `https://www.phippselectronics.com/an-introduction-to-i2s-audio/?srsltid=AfmBOorjHUqtwtlA2CFeV7xGEqQZORHMyT857Ux-UPz_49tcwGAxP35B`.

[6] Analog Devices, "I2C Primer — What is I2C? Part 1," Analog Devices Technical Article. [Online]. Available: `https://www.analog.com/en/resources/technical-articles/i2c-primer-what-is-i2c-part-1.html`.

[7] USB Implementers Forum, "Universal Serial Bus Device Class Definition for MIDI Devices, Version 2.0," May 5, 2020. [Online]. Available: `https://www.usb.org/sites/default/files/USB%20MIDI%20v2_0.pdf`

[8] Reverb, "Gear," *Reverb.com*, 2026. [Online]. Available: `https://reverb.com/marketplace?query=hardware%20synthesizer`

[9] D. Purves, G. J. Augustine, D. Fitzpatrick, L. C. Katz, A. S. LaMantia, J. O. McNamara, and S. M. Williams, "The Audible Spectrum," in *Neuroscience*, 2nd ed. Sunderland, MA: Sinauer Associates, 2001. [Online]. Available: `https://www.ncbi.nlm.nih.gov/books/NBK10924/`

[10] MIDI Manufacturers Association, "MIDI 1.0 Detailed Specification," MIDI Manufacturers Association, 1996. [Online]. Available: `https://midi.org/midi-1-0-detailed-specification`

Appendix A: Bill Of Materials

Table 3: Bill of materials

Description	Model #	Manufacturer	Quantity	Cost @	Total
MIDI Shield	DEV-12898	SparkFun	2	$24.50	$49.00
1/8 inch Plywood Planks	0022	TechSpark	4	$3.00	$12.00
Modulino Knobs	ABX00107	Arduino	6	$8.00	$48.00
Cherry MX Switch Breakout Board	BOB-13773	SparkFun	9	$2.50	$22.50
MIDI DIN 5-pin Cable	B093SW8ZNX	Mellbree	2	$4.00	$8.00
Arduino UNO R4 WiFi	ABX00087	Arduino	1	$27.50	$27.50
Arduino UNO Rev3	A000066	Arduino	2	$27.60	$55.20
Hosyond 4.0 Inches SPI Display	B053	Hosyond	1	$20.00	$20.00
					$242.20

Appendix B: Project Schedule

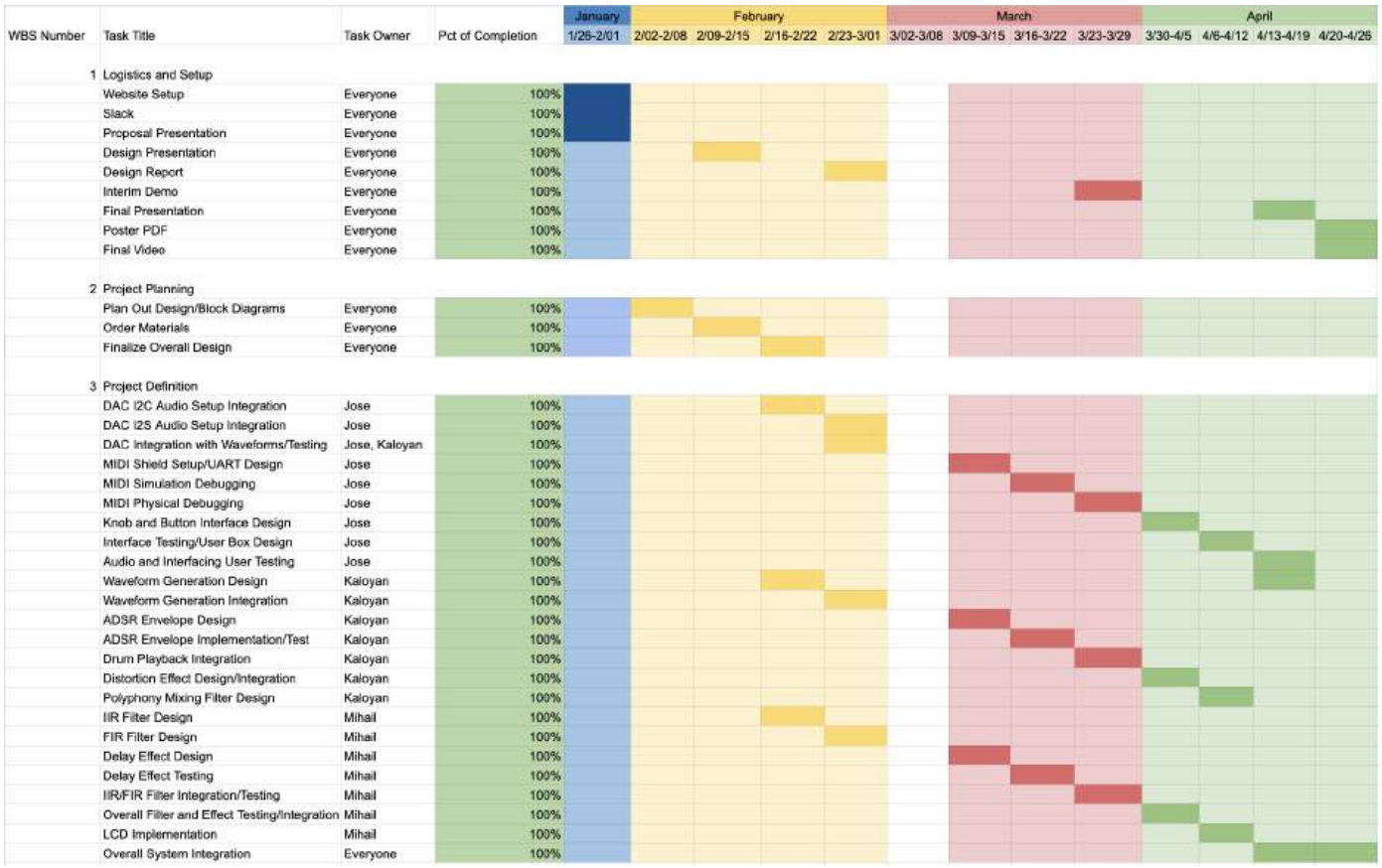

Figure 10: Project Gantt Chart