Check Out Our Soundcloud: An FPGA Wavetable Synthesizer

Jens Ertman, Charles Li, Hailang Liou

{jertman, cli4, hliou}@andrew.cmu.edu

Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of being a low-cost, FPGA-based wavetable synthesizer with digital wave-blending effects and digital effects chain including multi-voice unison, distortions, and reverb. While there are many inexpensive wavetable synthesizers that include basic digital effects, wave-blending effects are only found on expensive full-featured hardware synthesizers or software synthesizers. Our goal is to create a system targeting a platform competitive with other low-cost synthesizers on the market while capable of unique and interesting wave-blending effects not found in this price segment.**

***Index Terms*— FPGA, Music, Synthesizers, Wavetable Synthesis**

I. Introduction

IN this project, our team aims to develop an FPGA-based wavetable musical synthesizer that targets wave-blending and wave-shaping effects along with other standard digital effects found in synthesizers at comparable price points. Although we are developing the system on an FPGA, we envision this project as a prototype for evaluation on the track to create a dedicated chip for musical synthesis. From a market perspective, we are targeting the prosumer audio/music market, where people are interested in unique effects and sounds but are not willing to spend upwards of $4000-5000 for a professional synthesizer kit. Other devices in the market range typically either have simple oscillators combined with effects or wavetables that focus on instrument sound reproduction. While software solutions do exist, they are not self-contained and require a digital audio workstation, a MIDI controller, and some other software for the system to work in, meaning that although the wavetable synthesizer may not be extremely expensive, the combined workflow can be both high cost and limiting. Our goals for our synthesizer focus on having a polyphonic synthesizer with unique wavetable synthesis effects in addition to standard effects found on low-cost synthesizers such as distortion, delay, unison, and delay. On a more technical side, we want to have pitch accuracy within 5 cents of standard tunings, minimal harmonic distortion, and even frequency response. Finally, an important goal is to keep the total parts cost of the project as low as possible, to show that these effects can be had at a low cost.

II. Design Requirements

The requirements of the project will be split into two parts: features and audio fidelity. Other metrics we will be measuring, but without a hard requirement, are FPGA usage (chip area) and price. Power consumption is of a lesser concern because the system will be designed to plug into a wall outlet.

For features, we will first discuss traits of the synthesizer. The synthesizer will support four note polyphony, where four notes can be played through the synthesizer simultaneously. Polyphony is a feature found on some, but not all low-cost synthesizers, but we believe it to be a valuable feature to have in order to support being able to play chords. The synthesizer should have at least four different wave shapes stored in the wavetables; we anticipate the shapes to be two simple ones, such as a sine and a sawtooth, and two complex waveforms which can make interesting sounds when blended with others. The synthesizer can support user-controlled blending any two wave shapes together. For user-controlled synthesizer effects, we plan on implementing distortion (sample reduction and bit-depth reduction), delay, and reverb. the analog side, we will have an analog equalizer doing a final filtering step. For the digital synthesis and effects components, we will test them using Verilog testbenches, and for the analog filtering, we will generate frequency response plots by passing in noise.

For audio fidelity, we have determined several technical requirements that the project should meet. As a musical instrument, we want our synthesizer to be in tune, so we want our synthesizer to produce sounds within five cents of standard tunings. This will be tested with a regular instrument tuner and the synthesizer outputting an undistorted sine. We want the total harmonic distortion to be less than 5%, a property we will test by generating sine waves of different frequencies, running an FFT on the output, and calculating how much distortion exists at higher harmonics. Similarly, we want the frequency response to be even across all levels (<5%), and make sure no effect or filter behaves differently at different frequencies. This will be tested by observing output response over a range of frequencies and looking for deviations.

Our soft metrics, area and power, will be measured using the FPGA synthesis tools and measured for the analog components. Price will be measured by the total cost of components that we purchase, as well as the cost of the FPGA board, which is provided by the university. We aim to minimize these metrics overall but are not working with a hard requirement.

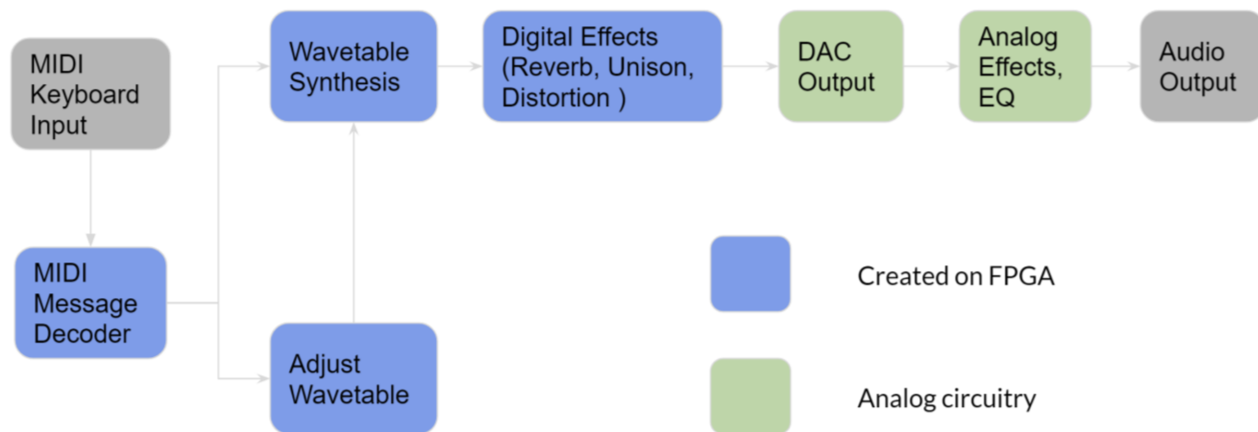


Figure 1. System Block Diagram

III. Architecture and/or Principle of Operation

The way that the overall architecture is designed is that first the player interacts with the MIDI controller that we have purchased. This controller includes the keyboard for playing notes and each of the control knobs. The control knobs each have two functions controlled by a function key on the keyboard. Additionally, active wavetable selection will be controlled by the switches on the FPGA board.

From the MIDI controller, a MIDI control message is sent to the FPGA via serial UART and decoded into a format that the rest of our architecture can read. The MIDI control messages can either be a control knob, which controls the strength of different effects, or a keypress message, which signals that a key was pressed or released, as well as how fast the key was pressed.

The first stage is the wavetable synthesis stage which takes the MIDI control signals for the notes and converts them into samples to feed through the rest of the pipeline. It also takes in the controls to choose which two active wavetables to fetch samples from. The last input control that this stage takes in is the unison control knob which controls the distance from in-tune each of the unison voices is. The samples are generated from traversing the wavetable, containing multiple different wave shapes, and the stride of the traversal determines the frequency or pitch of the sound. These samples are mixed together according to the desired amount of wave blending and then fed through into the effects chain. The number of samples mixed together is dependent on the amount of wave-blending, the number of voices from the polyphony, as well as the degree of unison effect.

Next is the stage of digital effects applied to the samples. These begin with the distortion module. The distortion comes in two flavors, one that reduces bit depth (bit crushing) and another that reduces the sample rate. The bit crushing effect uses a control knob value that adjusts between a value from 0 to 15 that determine how many of the bits will be truncated to zeros from the sample. The sample rate reduction module is similarly controlled by a control knob and discards some number of samples to reduce the effective sample rate. Following the distortion effect is the delay effect. This operates by taking a control knob value to determine the length of the delay and another control knob value to determine the loudness of the repeated sound and then plays back all samples a second time that delay length later. Lastly in the effects chain is the reverb effect. This effect is very similar in operation to delay however, the control knob for reverb controls how strong the reverb is, adjusting the amount of attenuation between “wall reflections”. Once the digital effects have been applied the sample is adjusted to be compatible with the DAC interface and then converted to an analog signal.

After the sample has been converted to analog through the DAC support circuitry, it is sent to a bank of filters that act as an 8-band equalizer with high pass and low pass filters as the outside bands. Each of these bands can be mixed at a different level using potentiometers and the overall amplification level can also be controlled. Finally, this signal goes through one final stage of amplification in preparation for being fed into a speaker.

The wavetable synthesis itself will all be done on-board the FPGA, with the analog filtering done only as a final step. This is quite different than most comparable synthesizers, which do very little in the digital space and do most of the effects processing using analog components. While many music-makers place a lot of value on the idea of the “analog sound,” we seek to give our synthesizer output some of these qualities by having the final output stage be fully analog.

The FPGA board that we have chosen for the project is a Terasic DE0-CV board with an Altera Cyclone V FPGA chip on-board. The full details of why this board was chosen will be expanded on in the later sections of this paper, but the key benefits of this platform were its relatively low cost as well as the large amount of on-board block ram to facilitate the easy storage and retrieval of the wavetables themselves, as well as making the design of the delay and reverb effects much simpler. While we ultimately see this project as a potential prototype for a production model using a dedicated chip, the benefits of using a cheap FPGA platform allow the option of going into production with the FPGA platform in the future while staying cost-competitive.

IV. Design Trade Studies

In order for us to meet our system specifications, we had to make several key tradeoffs throughout our design. Our goal in creating this synthesizer was to create a low-cost musical synthesis platform that could incorporate features from many high-end synthesizers, but in the pursuit of keeping costs low, we could not design everything the way we had wanted.

A. Memory and Block RAM

One of the biggest constraints that we had to work around was the amount of memory available in our system. Memory turned out to be a large bottleneck throughout our design, especially in the effects chain, given the size of each sample and the number of samples per second that had to be stored in the reverb and delay effects. Each wavetable requires several kilobits of block RAM as well. While we investigated other memory options, such as using DRAM, those options turned out to be infeasible both because of the amount of time and resources that would be required to develop and test a memory controller that suited our needs, as well as our need to be able to access many words in the memory at once. Block RAM on-board the FPGA suited this purpose well, because each block RAM unit is small at 10 kilobits per bank but could easily be combined together to form arbitrarily large memory banks. Memory is used all throughout our design independently; for example, each of the wavetables needs to be accessed in parallel, as well as the memories used in the many FIFOs throughout the effects chain. The amount of parallelism inherent to these memory accesses made on-board block RAM the only reasonable choice.

Our original FPGA board selection, the Terasic DE0 with an Altera Cyclone III, was chosen because of its very low cost and ease of development. However the block RAM issue forced us to reevaluate and we settled on the Terasic DE0-CV board, with an Altera Cyclone V instead. With this board, we had 3 megabits of total block RAM available, which according to basic calculations using the estimated block RAM usage of each module would be sufficient. The overall pipeline ended up coming in at just under 50% of the total block RAM usage, which was important to us because we needed the area to add a second, identical pipeline to handle a recording and looping feature. While it would have made things easier with an even larger board, given our priority in keeping the overall cost of the synthesizer low, we did not want to have to use an even larger board which would be significantly more expensive and contain more logic cells than were necessary for the project.

The block RAM limitation did force us to make some compromises in certain effects. The two major users of block RAM were delay and reverb: the delay module needed enough block RAM to store all the samples it needed to delay and the reverb module needed block RAM to simulate reflections and delay from different reflections around the room. In order to have a more natural and fuller sounding reverb, we needed to simulate many reflections happening at many different times. This was accomplished by our early reflection network with the tapped shift register and our late reflection network using the four parallel comb filters. The compromise was made to use only four comb filters because of the memory demands for each filter, as well as using a uniform-tapped shift-register. The uniform tap spacing that we used would cause a downgrade in the quality of the reverb effect, since it is preferable for the taps to be spaced in very different intervals, but doing so would stop the synthesis tools from using a minimal amount of memory and would force us to have a lot of block RAM units sitting partly unused. We tested this tradeoff with a software simulation of the reverb network and compared the outputs of the different configurations we experimented with subjectively to see how much of a difference the more natural reverb designs we could actually hear. This tradeoff did allow us to have a longer possible delay, since we originally had only intended for the delay FIFO to store 32,768 samples, using 60 block RAM units and allowing for a .75 second maximum delay, but saving block RAM in the reverb allowed us to double the delay FIFO to 64556 samples, using 120 block RAM units and allowing for a 1.5 second delay. Since we thought the longer delay effect would be a much more impactful change for the end user compared to the marginally better-sounding reverb, we decided to opt for more delay and slightly more artificial reverb.

B. Multipliers and Dividers

Multipliers and dividers synthesize to very large blobs of logic, and it was important for us to minimize the amount we used to keep the total logic usage within reason. While we had 66 DSP units on-board the FPGA that we had chosen to use, our synthesis and effects pipeline used a significantly larger number of multipliers than were available. We tried mitigating the potentially large amount of logic that could be generated by optimizing for the sizes of the values that needed to be multiplied. For example, while the math for a lot of the mixer operations and effects chain required a lot of fractions and floating-point math, we tried to do as much as possible with a simplified fixed-point system to scale the sample values. One way we optimized was to try to reduce the granularity of the scaling factors. Since the scaling factors were largely a product of the control knob configuration, which by the MIDI standard has a range from 0-127, we experimented with different granularities to see how few bits we could get away with actually using in our calculations. In the end, optimizing the mathematical operations within the system also helped with a lot of timing and critical path issues that we ran into.

C. Analog Filter Design

One tradeoff we had to make was in the decision on the amplifier and filter gains. The analog stage was originally designed to run entirely off the 5V output of the FPGA GPIO pins, and the gains for all the op-amps were set accordingly. Unfortunately, because the op-amps take 2V from the top and bottom of the rails, it meant that our effective rails for amplifications was only 1V peak-to-peak. This greatly limited the volume we could output at without causing clipping issues, and was a tradeoff made in the design to keep the overall physical end-product clean and compact. While we ended up running out of time to fix issues related to the FPGA voltage output being too unstable to use, we did not have the bandwidth to redesign the system to be used with arbitrary rails, and stuck with the 5V rails. However, this design does allow us to utilize the FPGA board's on-board power supply in the future if we were to find a good way to stabilize the output.

V. System Description

A. *MIDI controller to FPGA interface*

The first stage of the synthesizer system is the MIDI keyboard to FPGA interface. This stage consists of the MIDI keyboard, the support circuitry to convert the MIDI serial out to UART, the UART receiver module on the FPGA, and finally a UART decoder to convert the raw bytes into control signals sent into the main synthesis and effects pipeline.

The MIDI keyboard we decided to use for the project, the Stage Right by Monoprice 49-Key MIDI controller, was chosen for several reasons, the three most important reasons being the availability of rotary encoders, MIDI serial output, and low price. First was the number of rotary encoder knobs on the keyboard. The effects that we have on the synthesizer are all adjustable, for example, like the degree of blending between two wave shapes, the decay of the reverb, or the length of the delay, and we decided that leveraging on-keyboard rotary dials to control the effects would be the simplest method of control. Second was the availability of a MIDI out port. While the official MIDI standard specifies a 5-pin DIN connector as the primary connector, almost all modern MIDI controllers use a USB output instead, allowing the keyboard to connect easily to the desktop computer-based digital audio workstations instead. We wanted the MIDI controller we used to have the MIDI DIN output, since the signal sent out can easily converted into UART with some simple support circuitry, and we wanted to avoid falling into a trap of either designing a hardware USB controller from scratch or having to use a pre-made USB controller IP block, which could bring its own set of compatibility and integration issues. Unfortunately, this constraint severely restricted the pool of MIDI controllers we could use, since lower-end controllers only supported USB output and eschewed the legacy DIN connector. Finally, price was a major factor as well. Obviously, we need to stay under the $600 project budget, and one of the goals with our project is the keep the total parts cost as low as possible to stay in the same price/cost range as similar synthesizers. Many higher-priced MIDI controllers come with lots of unnecessary bells and whistles as well, and for our synthesizer, we only needed the most basic features along with the other requirements above. In the end, the Monoprice MIDI controller was the one of the only keyboards under $100 that satisfied the other requirements and did not have atrocious reviews on Amazon.

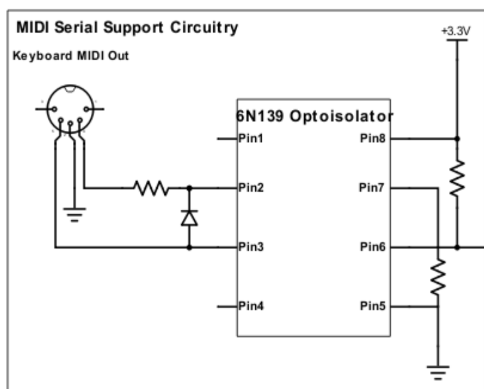


Figure 2. MIDI Support Circuitry

The next component of the chain is the MIDI support circuitry. The DIN connector outputs a serial data stream using current on and off to represent zeros and ones, so support circuitry is required to convert this into a voltage-based signal. Luckily, specific circuitry for this is detailed in the MIDI specification, requiring a specific opto-isolator chip with some other passive components. Although the exact opto-isolator chip used in the specification is no longer produced, equivalent circuitry was easily available online, using an alternative model of opto-isolator chip (6N139) that was mentioned in the MIDI specification. The details for the circuitry are shown in the block diagram. The output of this support circuitry is a UART input line that is connected to the GPIO pins on the FPGA board.

B. *MIDI Message Decoder*

From here on out, the “components” of the synthesis and effects pipeline will be Verilog modules synthesized on board the FPGA until the DAC interface with the final analog filtering component of the pipeline.

The first module of the pipeline is the UART receiver module. This module is fairly straightforward: it takes in the raw 1-bit UART serial signal as input and outputs a 8-bit data byte along with a 1-bit byte data ready signal. The baud rate of the MIDI transmission is specified in the MIDI specification as 31.25 kbaud, very slow compared to the system clock frequency we will be running the FPGA on. This module consists of a state machine to detect when the UART line drops low, signaling the beginning of the start bit, waiting half of a UART bit to start sampling in the middle of each bit, and then proceeding to sample again every UART bit. We anticipate to be running the system clock at 44.1 MHz in order to simplify the interface with the DAC later in the pipeline, so each UART bit will be slightly more than 1411 clock cycles. There is no worry of drifting out of sync due to the very large number of clock cycles per bit and that there will only be 10 bit per message before resynchronizing. The state machine will then proceed to read the 8 data bits, the one stop bit, and then either return to an idle state or detect a new incoming byte. The bits are pushed through an 8-bit serial in, parallel out shift register with the output connected to the output of the module. The format specified in the MIDI specification does not contain any parity bits, just one start and one stop bit per 8-bit data byte. Although MIDI messages are never just a single byte, this module is format agnostic and will only process a single byte through UART, asserting a ready signal after every byte. We will see in the next module how the full message is constructed.

The next module of the pipeline assembles the MIDI message from the individual bytes received by the first module. MIDI messages that we will care about for this project will come in two flavors, a two-byte message for sending rotary encoder information and a three-byte message for sending keypress data. Because the goal of this module is to assemble the whole MIDI message to pass on to the full MIDI message decoder, this module will need to decode the first byte of the message, the header byte, to determine which type of message is being received and how many bytes the message is. Each byte coming

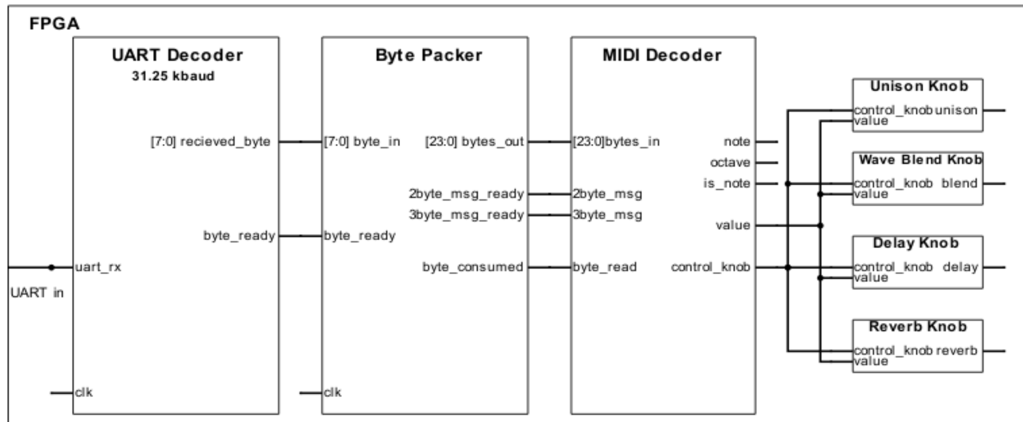


Figure 3. MIDI Message Decoding Diagram

into the assembly module from the receiver is pushed through another serial in, parallel out 32-bit shift register which shifts in 8-bits at a time. The inputs to this module are the byte output and byte ready of the previous module, while the outputs of this assembly module are a 32-bit MIDI message bit-vector as well as a two-bit one-hot signal that signals both that the message is ready and how many bytes the message contains. A value of 2'b00 will mean that the output is invalid, 2'b01 means the output is a valid two-byte message, and 2'b10 means the output is a valid three-byte message, with 2'b11 being an unused, illegal output. The 32-bit MIDI message bit-vector will have the bottom 8 bits be zeroes if the message is only two bytes long. This assembly module will also take an acknowledgement signal from downstream modules signaling that the MIDI message on the output has been consumed.

The next phase of MIDI message decoding is translating this 3- or 2-byte signal into a more readable format for the purpose of digital synthesis. This decoder module ingests a MIDI message and translates it out into note name, octave, note control, control knob value, and velocity. Note name is the name of the note desired using the standard musical note naming scheme of A to G. All half-steps between notes will be named as sharps. The octave output is the octave that the note falls under ranging from 0 to 7. If the MIDI message that is being decoded is not a note message the default values for these two signals are C, and 0 respectively. Note control is a two-bit signal where the high bit represents if the message is a note control message, and the low bit is a 0 if the message is a note off message and is 1 if the message is a note on message. Control knob value is an enum encoding of the names of the various control knobs that control effects on the synthesizer. There is a bank of registers that represent the control knobs if the control knob value matched the name of the control knob velocity is stored in that register and it represents the level of that knob. For a note message velocity represents the volume of the note.

C. Digital Synthesis System

The second major subsystem of the synthesizer pipeline is the wavetable digital synthesis subsystem. This is the portion of the pipeline which takes in the control data sent by the MIDI messages and outputs a stream of 16-bit samples that are to be run through the effects chain. This MIDI control data comes into this portion of the pipeline in a format which has taken the 3-byte or 2-byte MIDI signal and translated it into the more readable format generated at the end of the MIDI control subsystem.

The beginning of this chain receives the decoded MIDI messages. These decoded MIDI messages are then fed into the polyphony control module which manages the storage of the values for up to four simultaneous notes. The polyphony module requires a certain amount of cooperation with the ADSR module because it adjusts the way in which polyphony removes its stored notes. In order for ADSR to properly implement the release portion of its envelope polyphony needs to maintain the stored values for a note even after that note is released, only evicting a note if there is a new incoming note and no more non playing slots available. Additionally because ADSR is applied individually for each note support logic to keep track of the age of each slot of the polyphony module for the purpose of evicting the oldest note is also required. Therefor the polyphony module behaves as such. Until there have been four notes pressed it fills in the four slots from slot one to slot four with each note. As the notes are added they are also given a priority to determine in which order they should be replaced should more than 4 notes be played. They are also given a flag that says whether or not the note is currently being held so that releasing the same note twice will not cause the system to only remove one of the notes from the four slots. When notes are released they are not removed from the slots as the ADSR module requires the note values to remain buffered to play the release. So when a note is released it changes its flag and becomes the highest priority to be replaced by a new incoming note. When a note is replaced the new values are buffered into the slot and the slot is given the lowest priority for replacement. Logic checks the priority of all of the other notes against the priority of the note that was replaced. If the note that was replaced was of higher priority than another note then that notes priority increases by one and if not that notes priority does not change. This implementation of the polyphony module allows for the use of four simultaneous notes to be played with the user not needing to lift any of the currently held notes in order to play more notes. This results in the smoothest possible four voice playing experience. The only drawback to this

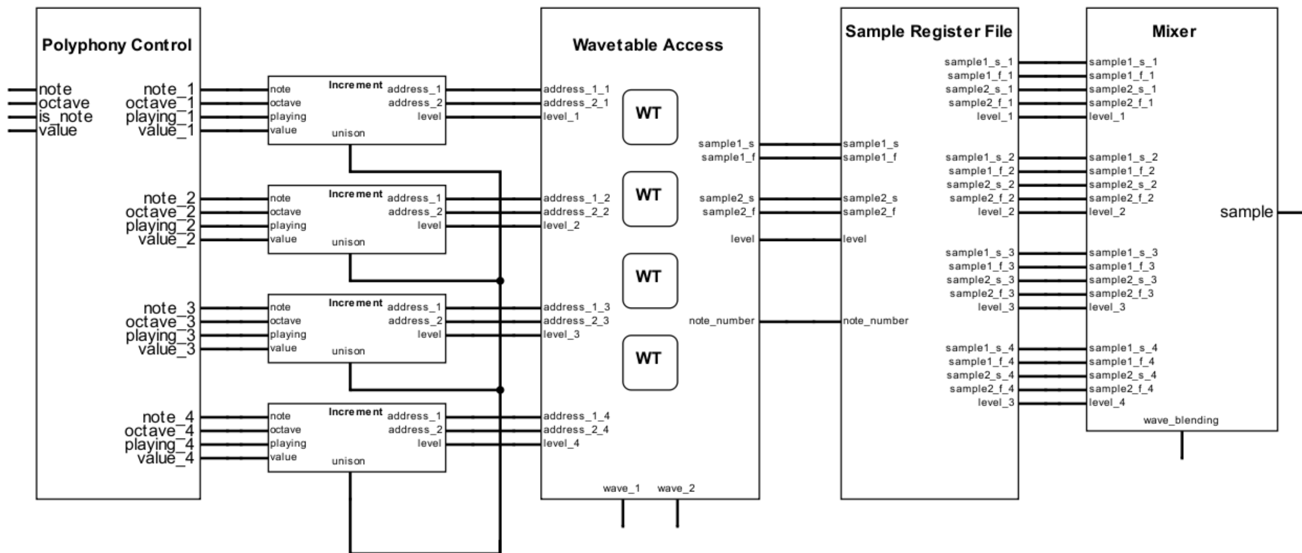


Figure 4. Digital Synthesis Diagram

implementation is that it requires an ADSR module to control when held notes that have been released should stop sounding. From the polyphony module comes the names of the notes that are going to determine the incrementor values. Each of the four note voices gets its own incrementor module which outputs the wavetable addresses for each of the two unison voices for that note. The incrementor module operates by taking in the value of the name of the note on the twelve-tone scale being played and the value of the octave that it is being played in and the velocity with which the note has been pressed. It first takes the value for the name of the note being played and calculates the incrementation value of the lowest version of this note in the MIDI specification. For example, if the note being played was C8 the first step would be calculating the incrementation value for C-1 the lowest C in the MIDI specification. This is done by computing the fundamental frequency of the wavetable running at the full 50 mHz system frequency. From this the value which the wavetable increments by is determined and expressed as a 30 bit number where the bottom 20 bits are treated as a decimal and the top 10 bits are treated as the address to the wavetable for the desired sample. Each clock cycle this incrementation value is added to the current address, and when a new note is pressed the current address resets to address 0 and the incrementation value is recomputed for the new note. Once the base increment value of the note is found it is then multiplied by 2 the correct number of times to represent the number of octaves higher than the base note the note being played is. Additionally, at the end of this module the unison effect is applied. This works by taking the computed incrementation value and multiplying the value by slightly more or slightly less than 1. This results in two frequencies that are slightly higher and lower than the fundamental frequency of the note.

These addresses are then sent to the wavetable access module. This module takes each of these addresses and sends them to M10k block-RAM containing each waveform. Each note has a copy of each of the four waveforms to access samples from. The M10k block-RAM is configured as a 2-ported ROM so that each module can have a port dedicated to each of the two unison addresses. All samples in the wavetables are 12-bit sample being read as a 16-bit sample. The remaining 4-bit are reserved for overhead involved in mixing the samples and applying effects. There are four possible waveforms available, a saw wave, a sine wave, and two more complex shapes generated for the user. For any given configuration of the synthesizer two wavetables will be active at a time. These active waveforms are selected by the player using the switched on the FPGA itself. Because of this every clock cycle 16 samples are fetched from the wavetables. This breaks down as 8 samples per active waveform and 2 samples for each of the 4 notes being held.

Once the 16 samples are fetched from the wavetable memory, the sample and the velocity of note hit are sent to the ADSR envelope generator module. ADSR stands for “Attack”, “Decay”, “Sustain”, and “Release” and are the four standard components of the loudness envelope generated. The idea behind adding ADSR to our design was to allow for better shaping of our notes and to allow a more mellow and smooth tone. In this module, we manipulate the magnitude of the note being played using some set values as well as knob value inputs to determine the attack length, decay length, sustain amplitude, and release length. By allowing the user to control these values, we can more easily shape the note, but a more complex ADSR module could be created with time.

The attack stage allows for the volume of the note to ramp up from 0 to a maximum over the attack length specified by the knobs. The calculation for the magnitude of the note is computed by first determining the slope of the line connecting the 0 to maximum volume, in our case 16, by dividing 16 by the attack length specified by the user. This value is then multiplied by a counter, which allows us to determine where on the line we should fall. This value is then multiplied by the velocity specified by the input to compute the correct scaled magnitude of our value. Decay works in a similar fashion. In the decay mode, we fall from 16 to our sustain amplitude over the decay length specified by the user. This is done in the same stages as attack, by first computing a sloped line, then computing where

we are on the line, and finally multiplying by the velocity we were given. Sustain simply multiplies the input velocity by a set scaled value.

Coming out of the ADSR envelope generator, the final scaled samples are sent to the mixer module. This module takes in each sample and the velocity value for its respective note. It then weights each notes sample by its velocity level and adds together all the samples for each of the two waveforms. Then taking the value provided by the blending control knob adds the resultant samples for each of the waveforms. The blending control knob determines how much of each of the two active waveforms is desired. At a value of 0 only active waveform one is played, at a value of 127 only active waveform two is played. The output of the mixer is buffered and updated on the sample clock as opposed to the system clock. This is done so that the effects chain which involves a delay effect and a reverb effect that both need to use memory must store the minimum number of samples to achieve their effects.

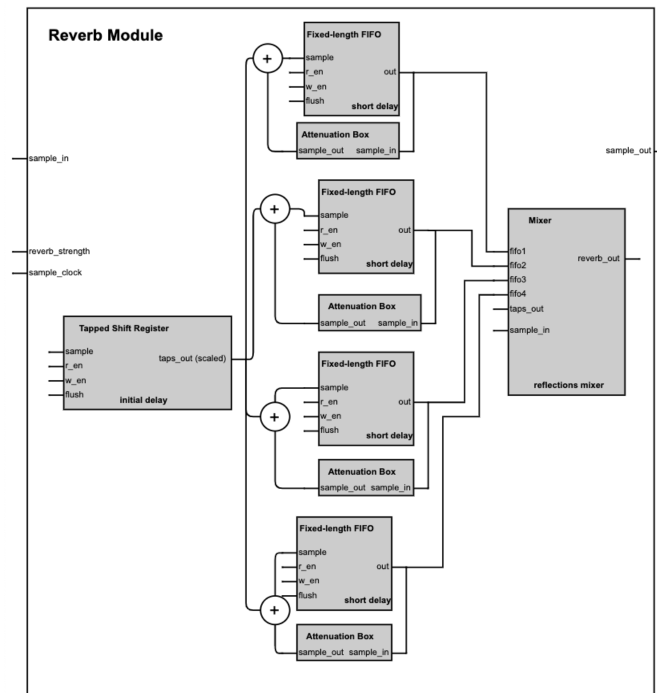


Figure 5. Reverb Module Design

D. Digital Effects Subsystem

After the wavetable synthesis pipeline, the sample is then piped into the effects pipeline. The effects pipeline consists of three parts, the distortion effect, the delay effect, and the reverb effect. These effects are all chained together in serially one after another. All three of these effects are independently adjustable using the rotary encoders on the MIDI controller.

The distortion module is first in the effects chain. The module takes in the 16-bit sample and the values of two rotary encoders as input and outputs a distorted 16-bit sample. The sample is distorted by both a sample-rate reduction effect and a bit-depth reduction effect, both independently controlled by different rotary encoders. The implementation of both effects is fairly straightforward, using a counter and only passing every *n* samples downstream and tossing the other samples for sample rate reduction, or zeroing out some number of the lower bits of the sample for bit-depth reduction. The number of samples discarded and the number of bits zeroed are the two parameters controlled by the rotary encoders.

The delay module follows the distortion module. The delay effect will replay the given sample at the same distortion with some variable loudness after some amount of time, again controlled with an onboard rotary encoder. The module itself will take a sample from upstream effects and the value of the rotary encoders as input, and outputs a sample that is the original input sample mixed with the output of the delay queue. The design of the delay was somewhat difficult at first, given that at 44,100 samples per second and 16-bits per sample, we would need more than half a megabit of memory for this queue. This is far more than can be stored within the LUTs of the FPGA and we decided between placing the queue in SDRAM versus the block ram of the FPGA. Ultimately, we decided the while there is a lot more SDRAM available on the FPGA, the time and effort spent dealing with potential memory controller issues would make it not worth it. Although the original FPGA board we wanted to target, the Terasic DE0 board with an Altera Cyclone III, does not have enough block ram to support the queues for the delay and the reverb effect, we decided that switching to the very similar Terasic DE0-CV board with a Cyclone V would be sufficient. Although this board is slightly more expensive than the low cost DE0 board, the cost increase is not particularly worrying, given the low cost of the rest of the system. We found that Quartus's Megafunction wizard could create fixed-length FIFOs using the M10K block ram and will be leveraging this tool to create the core of the delay queue. We will target a maximum delay of one second for the delay effect, adjustable by the rotary encoders. Ideally, since for every sample we enqueue we also dequeue a sample on the same clock cycle, the FIFO will be a fixed length for each delay length. When the user changes the length of the delay, we will either enqueue without dequeuing to adjust to a longer delay or dequeue without enqueueing to adjust to a shorter delay. While this does mean that there will be some transient distortion, given the short timescales within the queue itself, the transient should not negatively impact the user experience. We will design support hardware around the FIFO to both control the FIFO and adjust the length of the delays, as well as hardware that mixes the input sample and the output of the delay queue. The output of the delay queue will be adjusted in volume based on the user input and then mixed with the input sample and outputted.

The reverb module is similar to the delay module in principle, operating on a delay queue as well. The primary difference, however, is that the queue in the reverb module will incorporate feedback and attenuation to create the echoing effect simu lating the sound bouncing off the walls in a room. The input to the reverb module will be the sample outputted by the delay module as well as the value of the rotary encoder that will control the amount of attenuation in the feedback loop. Again, we will utilize the Quartus Megafunction wizard to create fixed length FIFOs using the M10K block rams aboard the Cyclone V. In the design of the reverb module, we have two stages. In the first stage, we use a tapped shift register to simulate an initial network of early sound reflections. The values out of these taps are then scaled and added to become the inputs to the second stage. The second stage of reflections uses four comb

filters made of feedback FIFOs feeding into an all-pass filter made with a FIFO with both feedback and feedforward loops. The values for the tapped shift register scaling, tap width, FIFO length, and feedback gains were all determined experimentally using software simulations of the effect. Compromises were ultimately made on the quality of the reverb effect to save on the large amount of block ram and multiplier usage. Again as before, the output of the reverb queue will be mixed with the input signal coming into the module; however we do not anticipate this mixing to be adjustable at this time. Currently, since that the reverb module is at the end of the digital effects chain, the mixed sample outputted from the reverb module will be piped into the DAC support module.

E. Drum Pads

Parallel to the digital synthesis and effects chain is the drum synthesis pipeline. While we originally considered using drum samples to provide the best sounding drums, an investigation into drum synthesis techniques used in early video game consoles yielded simple drum sounds that sounded percussive enough yet would not have the hefty memory requirements that a sample-based drum sound would need. There are four different drum sounds that our synthesizer can create: bass drums, snare drums, tom-toms, and closed hi-hats. All four of these sounds are created with only a triangle wave and white noise generator. The three drum sounds are all made by pitch shifting a triangle wave down quickly over the course of a tenth of a second to create a thick percussive sound. For the snare drum, white noise generated by a linear feedback shift register is mixed in with the triangle wave. Finally, for the hi-hats, a short burst of white noise is played that quickly fades away. All four of these drum sounds are controlled using the push buttons on the Terasic DE0-CV development board.

F. Digital to Analog Conversion

The digital signal now must be converted into an analog signal so that it can go through our equalizer filters and become an actual sound through our speakers. In order to do this, we use a MAX841 DAC, shown in figure 5. Our DAC is powered through a typical laboratory power supply with 5V. Our system then outputs an SCLK, generated from our 50MHz output clock. It shows the DAC 16 positive edges to allow the shift register within the DAC to shift in our 16-bit sample. Our data is offset by around 40ns to ensure that we do not violate any set-up conditions for the shift register. Our sample clock is also generated from our 50MHz clock by counting clocks before needing to change.

Verification for the DAC modules was done through testbenches and measuring the outputs of the FPGA GPIO pins through an oscilloscope.

G. Filters and Equalizer

After we have converted our digital signal into an analog one, we wanted to create a simple equalizer to give the user greater control over the sound output. This equalizer also has a volume control tacked onto the end. As a baseline target, we wanted to be able for the user to control sounds from a note referred to as A0, to a note referred to as C8. These notes are denoted as the tone followed by an octave, and the range that we chose gives us a range equivalent to that of a piano. The equalizer has eight different ranges to mirror the number of equalizers in a normal synthesizer. Shown in Fig. 7. below are the ranges of each filter. Regions 1 and 8 are the final low and high pass filters that we wanted to use. Currently, they are fixed filters, but the design allows for us to potentially control where their cutoffs are and make them variable cutoff filters, which is a feature that many on-the-market synthesizers have.

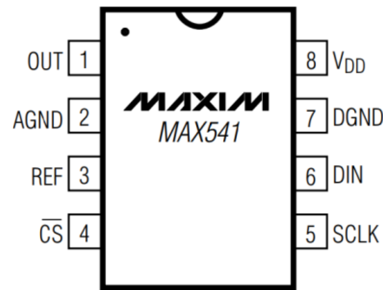


Figure 6. DAC DIP chip pinout

In order to create these, our circuitry implements two-stage filters for each frequency range. The reason for this is that it is simpler to create a low-pass combined with a high-pass that have sharp cut-off frequencies than to create a bandpass with less sharp cutoffs. Furthermore, it allows us to separately design each filter, which makes testing and fixing any errors that have been made much easier. The overall circuitry is shown in Figure Fig. 8., but with only two band-pass filters shown rather than the 6 that we intend to have when our design is completed.

Region Cutoffs	Low (Hz)	High (Hz)	Low (rad/s)	High (rad/s)	Lowest Note in Range	Highest Note in Range
Region 1	0	51.53959	0	323.8328324	-	G1
Region 2	51.53959	96.59381	323.83283	606.9168495	G#1	F#2
Region 3	96.59381	181.0329	606.91684	1137.463609	G2	F3
Region 4	181.0329	339.2860	1137.4636	2131.796905	F#3	E4
Region 5	339.2860	635.8786	2131.7969	3995.343682	F4	D#5
Region 6	635.8786	1191.742	3995.3436	7487.941794	E5	D6
Region 7	1191.742	2233.525	7487.9417	14033.65437	D#6	C#7
Region 8	2233.525	4186	14033.654	26301.4137	D7	-

Figure 7. Filter ranges

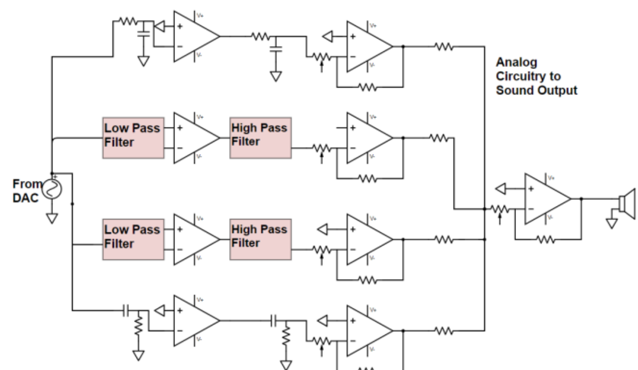


Figure 8. Equalizer design

The low-pass and high-pass filters are in Butterworth topologies and have a second-order pole at the given cutoff frequency. This pole allows us to achieve 40dB per decade roll-off, which is important in ensuring that sound frequencies that we do not want to pass through are not passing through. The topologies for the third-order low-pass and high-pass filters are relatively simple in terms of design and implementation. Shown in figures Fig. 9. and Fig. 10. are the third-order filters.

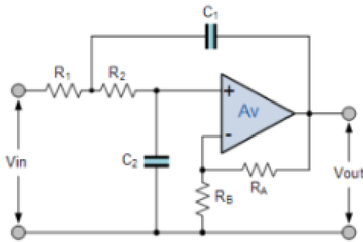


Figure 9. Low pass filter

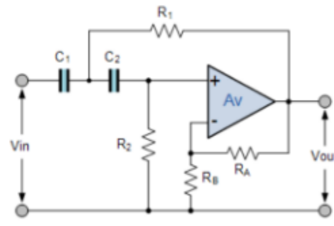


Figure 10. High pass filter

In terms of overall design, we wanted to add as many layers of buffers as possible in order to ensure that there will be minimal loading between all the stages of filters that we have designed. This also allows us to test each filter independently and then combined in order to help smooth integration. After the two stages of filters that we have, we have an effectively variable gain op-amp whose gain is controlled by the potentiometers. These allow us to create an equalizer which can be used to boost or attenuate certain frequencies and allow the user to create unique sounds. Finally, we combine all our signals via a summing amplifier, which also has variable gain, and send the output to a speaker. The speaker is yet to be determined, but the chosen speaker will determine what amount of gain is needed at minimum on the summing amplifier at the end.

VI. Project Management

A. Schedule

Our breakdown of work is relatively simple. We wanted to ensure that everyone was scheduled in their comfort zone where they would be able to do their best work. Furthermore, we wanted to make sure that everyone had enough slack available so that they would have some extra time to do their work if necessary. Our schedule can be seen in the figure below. It may be a bit difficult to read, but in yellow are tasks that require everyone to pitch in to complete, in green are tasks for Hailang, blue tasks for Jens, and red tasks for Charles. Additionally, a larger version of the schedule can be seen after the references section. Each person should have roughly two weeks of slack for themselves, while the overall project where everyone might need slack has around another added week of slack.

These two weeks were consumed with solving unforeseen issues with the ways that certain modules worked. Then at the end of the project the overall slack that had been put in place for the group was consumed in large part by the task of tracking down the analog bug that was creating all of the noise in the circuit. This took a considerable amount of time and was found to be mainly caused by a timing bug in the DAC section.

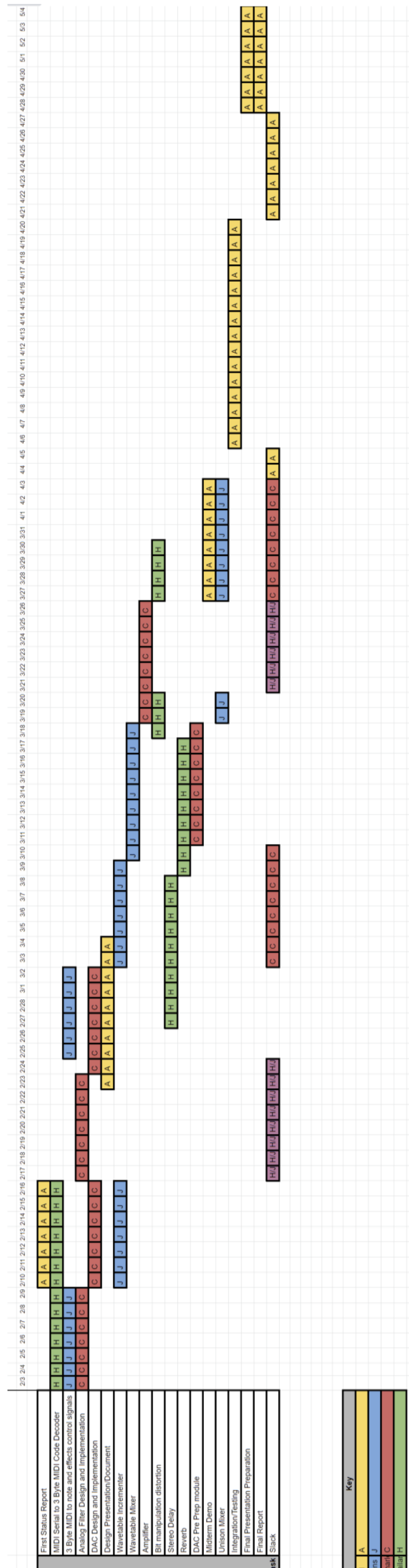


Figure 11. Schedule

However, other issues with the implementation such as the solderless breadboards were found and solved in this time. Additionally, a lot of the time went into adding some bonus features to the project that originally were not budgeted for. These features included the ADSR envelope and the button-based drum system.

B. *Team Member Responsibilities*

In terms of work breakdown, Jens has a work focus on converting signals into digital signals. His focus is on determining what needs to be read from the wavetables which store digital sample values in the distributed block ram.

Jens	Charles	Hailang
MIDI message decoder	Analog filter design	MIDI to FPGA interface
Wavetable Incrementor	DAC support circuitry	Delay effect
Wavetable blending and mixing	FPGA to DAC interface	Reverb effect
Unison mixing	Amplifier design	Bit distortion effect

Table 1. Work Distribution Table

Hailang's focus is on digital logic manipulation, taking a sample value and applying different effects to that to get a final digital value. Charles' work focuses on the analog side of the synthesizer. His work involves converting the digital value through a DAC to get to an analog signal and filtering that signal to reach the speaker and an end sound.

As such, we broke down our project into several parts, which are distributed as described above. The distribution of work can be seen in Table 2.

C. *Budget*

The total cost of the synthesizer lie mostly in the MIDI controller and the FPGA board. We tried to buy analog parts specifically designed for audio applications as those tended to be less noisy, but as a result, the cost was therefore higher as well. We had smoked out several DACs during testing so we made sure to buy lots of extra ICs over the course of the project so we had spares on hand. For a complete breakdown of parts, see Table 2.

D. *Risk Management*

When initially planning out the organization for completing this project we knew that the largest potential issue that we would face would be in the integration of the entire system. Specifically, when it came to the transition between digital signals and analog. Because of this fact there were a few specific actions that we took to safeguard against one of these issues bringing down the project entirely. The first of which was the built-in slack time at the end of the semester that was meant to pick up any extra time that we would need to complete integration. This ended up being extremely useful as there were a few bugs at the end that took a very large amount of time to finish. This bug had to do with noise on the analog side which took a long time to determine both the cause of and the solution for.

Another stride we took to try and prepare for risk at the beginning of the project was to stress having a complete end to end solution complete by the midpoint demo. This would make sure that at that point we had at least worked out some of the issues that would have been created from the transition from digital to analog. When it came to polishing the end sound that the synth created there was a large issue with the ground noise that had been created through our implementation of the equalizer on a solderless breadboard. Moving this design onto soldered protoboards solved the ground noise issues from the filters. This change helped to stabilize the connections into the filters and from the potentiometers. However, it was not known for sure whether this would fully solve the issue so the system had been designed with the potential fallback of bypassing the filter bank all together to still get a usable sound out at the end. This kind of modular design was also very important to the way that we designed the whole system. The digital synthesis pipeline was designed in such a way that almost any module could be removed from the chain if it ended up being broken and a valid sample would still be output at the end of the

Item	Unit Price	Quantity Used	Quantity Bought
Monoprice MIDI Keyboard	$50	1	1
Terasic DE0 CV FPGA	$150	1	1
MAX 541 DAC	$17	1	4
RC4580 Op-amp	$0.99	9	14
6N138 Opto-isolator	$1.80	1	3
LM386 Audio Amplifier	$5	1	3
Various Passives	$30	N/A	N/A
Total	$262.71		

Table 2. Bill of Materials

pipeline. This design mentality came about from trying to make sure that as features were added to the system as a whole they would not break any of the currently functioning components. Any parts that ended up being either broken or not working exactly as intended could be cut out of the rest of the system without any negative side effects.

When purchasing physical components for the system one of the potential risks that we faced was that some of the components are very sensitive to the currents and voltages that are run through them. Specifically, the DAC was a potential area for risk and on one occasion when a error was made with fast modifications being made to the circuit to try and diagnose the issue of noise. This error resulted in driving far more current through the DAC than we had intended and fried the component. To mitigate this risk when ordering parts that were sensitive to being damaged multiples were ordered where we had the budget. The DAC was a prime example of where having the spare parts ready to go and replace prevented this issue from setting the project back while waiting for a new part to be ordered.

VII. Related Work

In researching similar work to the synthesizer that we wished to create we used three specific works for reference. Those were: Serum, the Waldorf Blofeld, and the Waldorf Quantum. These three synthesizers were chosen as they all use wavetable synthesis to produce sound but are very different in the markets they target and the features they provide.

Serum is a software wavetable synthesizer plugin made by Xfer Records and is the industry standard wavetable synthesizer for electronic music production. It is built on the core feature of being able to blend and manipulate waveforms. Additionally, it comes in at a relatively price point of 300 dollars. However, it is locked into a software environment that requires a significant amount of additional software and compute power to run to its full potential.

The first of the hardware synthesizers that we compared to was the Waldorf Blofeld. This synthesizer was chosen for its low price for a hardware synthesizer of 400 dollars. However, the core feature that our synthesizer targets of waveform blending is completely absent from this synthesizer. This lead us to attempt to find an example of a hardware synthesizer that did include this core feature.

Moving up Waldorf's product stack we arrived at the Waldorf Quantum, a wavetable synthesizer with an over 4000-dollar price tag. This synthesizer did include wavetable blending effects but also included copious other effects to justify the extreme price. This confirmed that there was a vacancy in the market for a hardware wavetable synthesizer that focused on wave manipulation effects while achieving a lower budget price target.

VIII. Summary

Our system overall met almost all of our design specifications at the end of the day. Our effects were just as we expected and our notes played at the correct volume and frequency that we defined. Our system is of course tied to a power supply at the moment, which is less than ideal since we want to be an operational system outside of the limit of a power supply. We think that the best way to deal with this would be to introduce some DC-DC voltage convertors into our system that can run from the FPGA to the rest of the circuitry. We were also limited by the voltage that we gave our filters and final amplifying stage since we could never get a swing larger than the 5V rails that we provided that chip. Again, this problem would be solved by introducing new DC-DC convertors to give us a larger voltage rail. The only issue we ran into with our specifications was the consistency in frequency response. Unfortunately, the response of synthesizer deviates by more than 5% at the lowest octave and a half of the MIDI specification. This is likely because of the number of DC offset filters built into out analog stage also attenuating some of the lowest frequencies as well. However, the frequency response was very consistent, and did not deviate at all for the rest of the MIDI range, which we are satisfied with, especially since many notes at the bottom of the MIDI range are only barely within the limits of human hearing.

A. *Future Work*

We intend to continue to flush out some of the features that are available on our board by adding frequency modulation (FM) synthesis and a looping module that allows for a sequence of notes to be played once and then looped continuously in the background as other notes are played. We also would like to eventually move our filters and analog components onto a PCB to set in stone our design in a way that we could not before finishing the project since we wanted to have a higher ability to tune our circuitry.

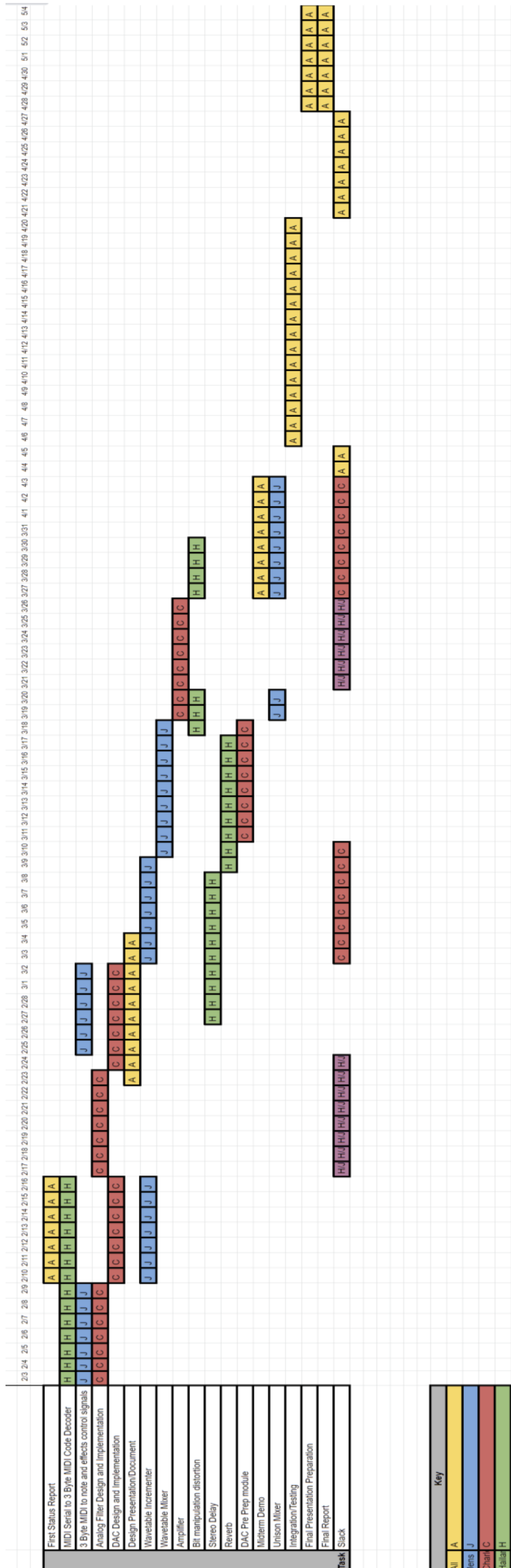
B. *Lessons Learned*

In terms of lessons learned, the biggest issue we had was making sure that our digital to analog interface came through clean. The high frequency digital circuitry has a lot more tolerance on it than the audio analog circuitry can take. As such, at the interface, any source of noise could create a lot of problems. This was one of the biggest issues we had with our system. Another issue on the analog side of the system was that there were a lot of loose wires when we used a typical breadboard to build our circuit. These wires caused us to have a lot more noise on our output than we wanted. In order to fix this issue, we decided to solder our filters on to protoboards. This helped our presentation and also provided us with a way to isolate the problem away from the analog circuitry since it was less noisy than before. On the digital side of the project, it is important to make sure that everyone understands the full front-to-back behavior of the system. This will help the team when integration time comes. We had defined our interfaces well, but experienced problems when it came to building the entire system since although any two parts worked fine together, the overall architecture of the system did not work initially because everyone had been focused on their small part as opposed to the system as a whole.

References

[1] J. Moorer, “About This Reverberation Business”, *Computer Music Journal*, vol. 3, no. 2, 1979, pp. 13–28. JSTOR, www.jstor.org/stable/3680280.
[2] Xfer Records xferrecords.com.
[3] Waldorf www.waldorfmusic.com/en/
[4] MIDI Spec www.midi.org/specifications

APPENDIX A. WORK SCHEDULE



APPENDIX B. MODULE DIAGRAMS