Digital Synthesizer Built with Virtex 2 FPGA

Sarah Pohorecky

December 13, 2018

1 Overview

Digital synthesizers are electronic musical instruments that generate their sounds using digital processes instead of simply playing back recorded instruments. This synthesizer was implemented on the 6.111 labkit's Xilinx XC2V6000 Virtex 2 series FPGA and used direct digital synthesis to generate clear, in-tune tones. Recording tracks was planned to be implemented using ZBT RAM, but it was unsuccessfully integrated. A method of ADSR enveloping was designed, but not implemented in the prototype version. I wish more had worked for the final demo, but I plan to continue working on it once I've acquired my own FPGA board.

The synthesizer compromises of five main components: the tone generator, the note input interface, the AC97 interface, the memory ("recorder") interface, and the control module (which provides certain synchronization and command signals to other modules). Fig.1 shows a high-level block diagram.

2 Note Input

The synthesizer receives signals from pushbutton "keys" connected to the GPIO pins of the FPGA. Each button is connected to 5V power and FPGA ground through a pull-down resistor. The note input module takes these signals and debounces them. Then for each of the `N_NOTES` notes it sets the necessary tuning word for the tone generation. These tuning word values are included in the module as parameters, and correspond to the value required for the desired frequencies of each note. Based on the value of `octave_select` and whether each key is pressed, the module outputs a concatenation of all the tuning word values as a `N_NOTES`$\times$11 (the length of each tuning word) wire.

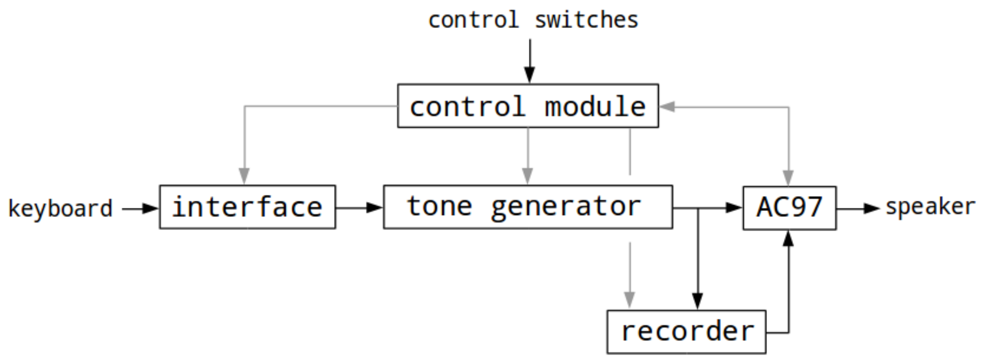

Figure 1: General System Diagram

3 Tone Generation

Since the goal of this project was to produce a working synthesizer, the tone generation module is the most important component. The tone generator is responsible for supplying sine wave samples at desired frequencies to the rest of the system. The method used to create these wave samples is known as "direct digital synthesis." This method can create a sine wave in three steps: first, at each time step, the phase accumulator module generates the current angle for the wave. This angle is then passed to a phase-to-amplitude converter, which gives the value of the wave at the desired angle. Finally, a DAC smooths the samples into a sine wave[1]. A diagram of the tone generator is shown in fig.3

[1]Analog Devices. *All About Direct Digital Synthesis* `https://www.analog.com/en/analog-dialogue/articles/all-about-direct-digital-synthesis.html`

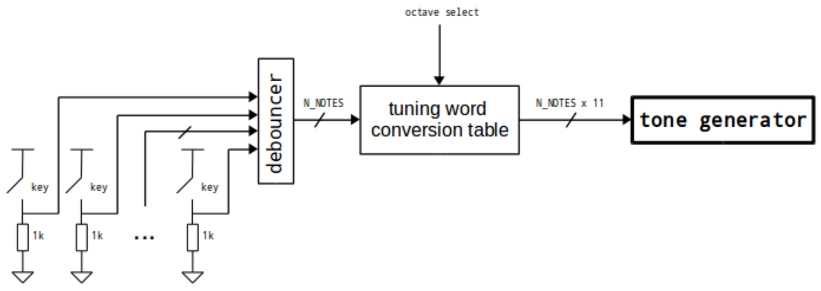

Figure 2: Note Input Module

3.1 The Phase Accumulator

Each phase accumulator is essentially a wrap-around counter. This counter is also called a "phase wheel" and a visual representation of it is shown in fig.4. Each value in the accumulator represents an angle that can be used to lookup an amplitude value of the sine wave. Changing the number of steps by which the counter increments (the tuning word) can be used to generate a series of angles that will gives amplitudes that represent waves of different frequencies. This is derived from how signals can be interpolated from samples. In fig.5, if the tuning word (called M) is one, then every angle is selected. If $M = 2$ then every second angle is selected, and if $M = 4$ then every fourth angle is selected. Once these angles are converted to amplitude values, we get extrapolated waves of higher frequencies as the step size increases.

A step is taken on each clock cycle, so the minimum frequency (when $M = 1$) that can be output is $f_o = \frac{f_{\text{clk}}}{2^N}$ where N is the size of the counter in bits. Higher frequencies can be created by increasing M, and generally, the output frequency for a given M is $f_o = \frac{f_{\text{clk}} M}{2^N}$. Since we want a new angle for each audio

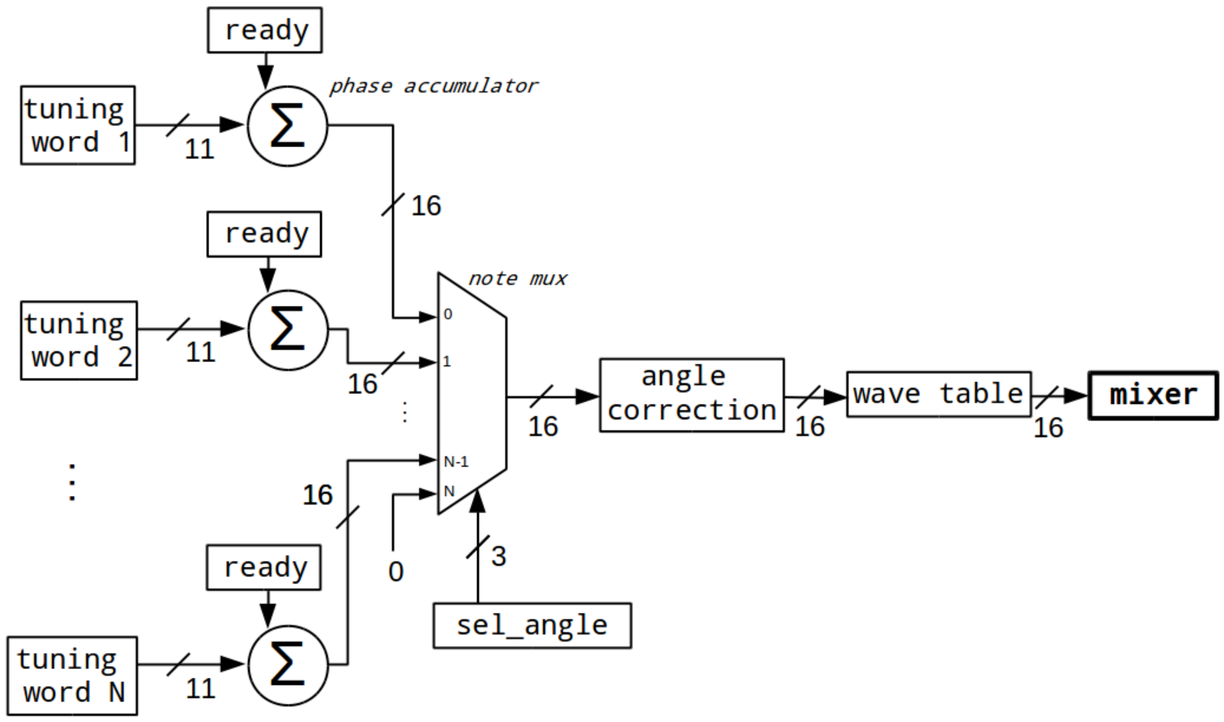

Figure 3: Tone Generator Module

sample, we step the accumulators on the AC97 `ready` signal, which gives us $f_{\text{clk}} \approx 48\text{kHz}$. We can choose N and calculate M to fit our design needs. The larger N is, the more memory we require for our amplitude lookup table, since there will be more angles that can be selected. However, larger N also allows for more precise frequency selection (it is clear that for any given N, arbitrary frequencies cannot be selected, since M must be a whole number). Since we know which frequencies we want to accurately output (the frequencies of all the notes that the synthesizer plays), we can solve for M given an N as $\frac{f_o}{f_{\text{clk}}} * 2^N = M$. This M will not necessarily be an integer, so it is rounded to the nearest whole number. The error in the actual output frequency can be calculated for all note frequencies. A value for N can then be chosen that has acceptable error. As expected, the error goes down with higher N. With $N = 16$, the average error across all the notes is 0.19Hz and the maximum error on any note is 0.36Hz.

The tone generator contains `N_NOTES` phase accumulators, one for each key. On each `ready` signal, they all increment by their individually supplied M.

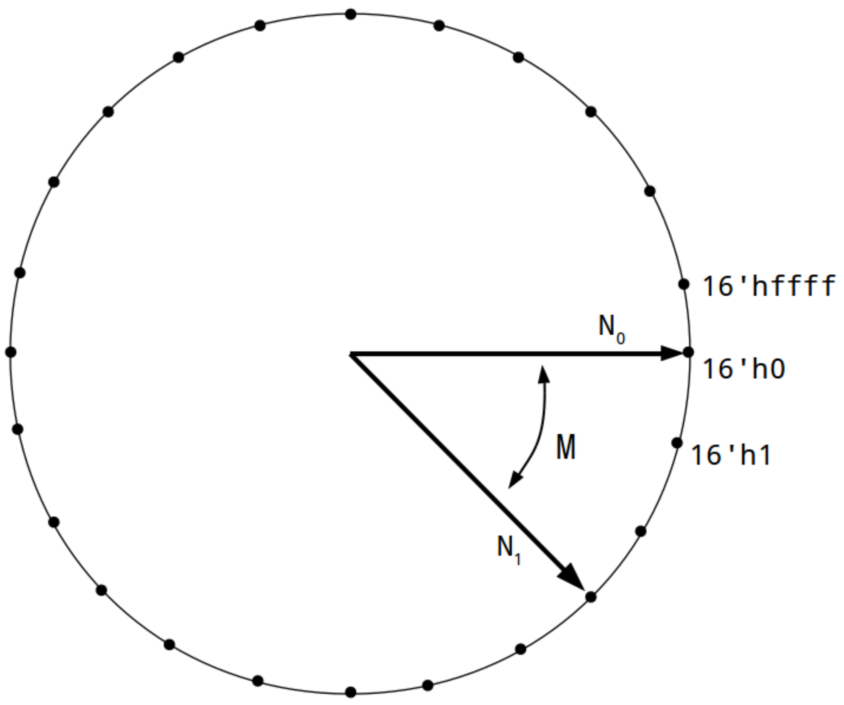

Figure 4: Phase Accumulator/Phase Wheel Representation

3.2 Angle-to-Amplitude Conversion

After the phase accumulators step their angles on the `ready` signal, those angles must be converted to amplitude values. This is done in a serial fashion, where one angle is taken in per step and passed through to a lookup table which chooses the correct value for the given angle. These conversions were designed as serial accesses since the lookup table is the most memory-intensive component of the system and parallelizing these lookups would require multiple copies of this large table, which would dominate the memory of the Vertex. The serial access works since there is ample time for processing running at clock speed (27 MHz) to finish between each ready pulse (48 kHz).

The serialization is controlled by a multiplexer which switches between phase accumulators depending on the value of `step_angle`. `step_angle` is set by the control module. After the `ready` signal is received by the control module, it delays for several clock cycles in order to let the phase accumulators calculate the new angle. After this delay, it resets `step_angle` to the first phase accumulator and holds at that accumulator

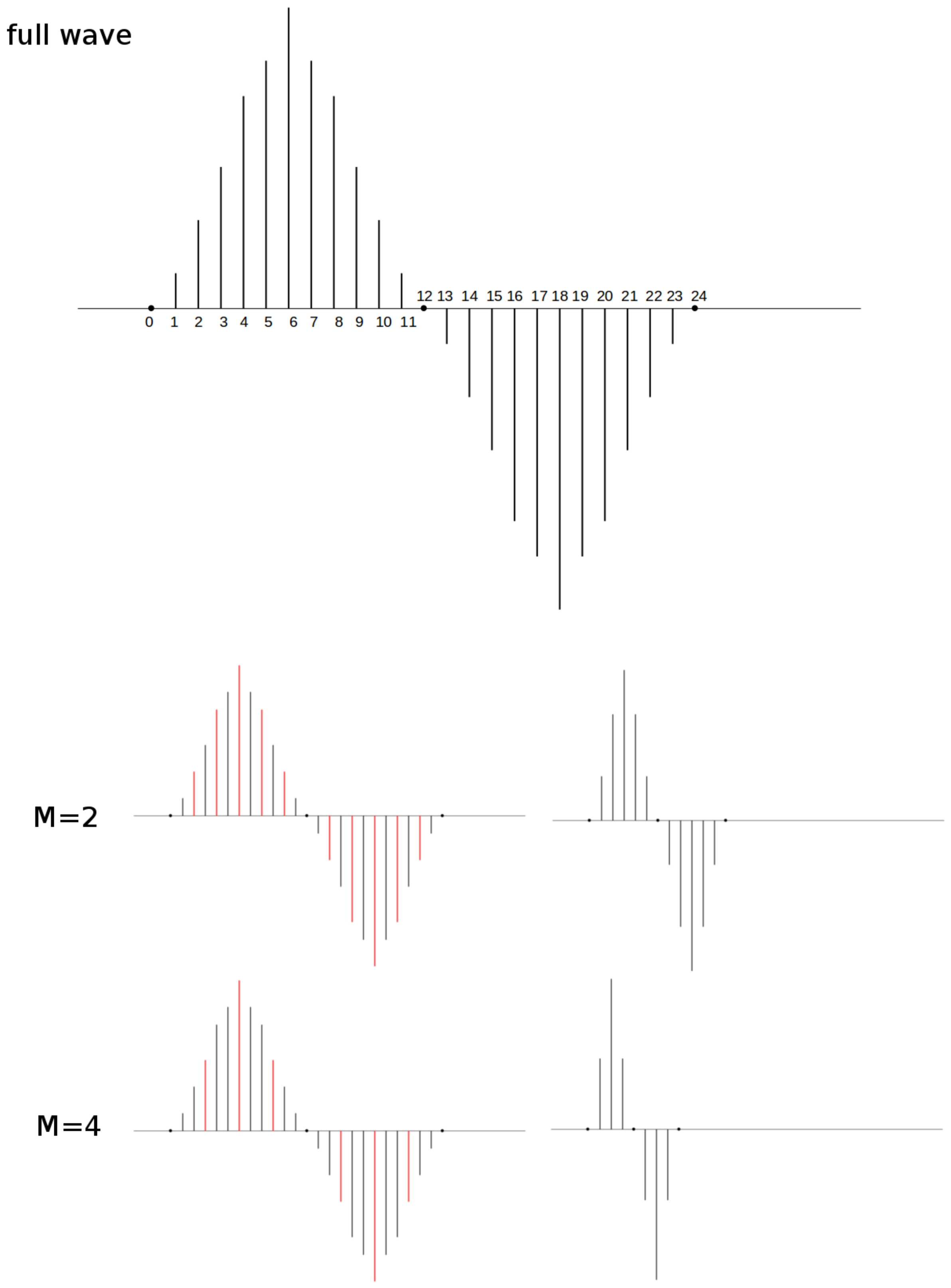

Figure 5: Effect of Changing M on Wave Frequency

until the lookup is complete and the new amplitude value has been piped to the mixer. After that time has elapsed, it increments to the next accumulator. It continues this process until it reaches the last input, which is statically held at 0 so that there are no more effective inputs into the mixer.

The wave table is a ROM loaded with amplitude values corresponding to each input angle. This was originally hard-coded into the module as a case-statement lookup, but this was incredibly inefficient with regard to synthesis time, so a ROM module made much more sense. The wave table size is set by the size of the phase accumulators. Since the accumulators are 16 bits wide, the full table would need 65536 entries of 16 bits each, resulting in 1048576 bits of data. This can be reduced by 75% to 262144 bits by making a quarter-wave table instead of a full-wave table. The quarter wave table utilizes the symmetry of the sine wave by only storing the first quarter of the values and changing the order in which they are accessed and negating them as necessary to calculate the correct value for angles higher than the first quarter. This memory reduction comes at the expense of some latency in angle correction (calculating the index into the table from the given angle) and negating the memory values.

In order to better understand the quarter table, say that A is the angle corresponding to the peak of the wave (at the end of the first quarter wavelength). For angles θ less than or equal to A, the amplitude value with simply be $\sin(\theta)$. For values $A \leq \theta \leq 2A$ (still in the positive half-cycle, but above the first quarter), the angle must be adjusted to essentially "index backwards" through the values in the first quarter (since the second quarter is a mirrored copy of the first). This is accomplished by assigning the amplitude value as $\sin(2A - \theta)$. Then for the third quarter, the index should be run upwards again (since the magnitude grows), but the value must be negated. This region's amplitude value is given by $-\sin(\theta - 2A)$. Finally, the last quarter must have a negated value and its indexing run backwards, so the amplitude value is given by $-\sin(4A - \theta)$. In the context of the FPGA implementation, this is accomplished by preprocessing the output angles from the phase accumulators to get the index angle a, according to the rules above, before reading the correct value from the ROM. After the value is read, it is negated if the original angle was in the negative half-wave. Since the AC97 doesn't require explicitly signed values, nothing in the system is declared signed, but since all the wave values are formatted as twos-complement numbers, all of the arithmetic in the system works out, as detailed in section 4. Thus, the negating is simply done by bit-flipping the amplitude and adding one.

4 Mixer

The mixer simply adds amplitude values together for each sample in order to make multi-note tones. The mixer resets its sum to zero after each `ready` signal. For recording and simple play mode, it adds each value it receives from the tone generator module. This addition is controlled by the `read_value` signal from the control module, which ensures that each value is only added once. While in playback mode, it accepts the memory output values from the recorder module and pipes them through to the AC97 interface.

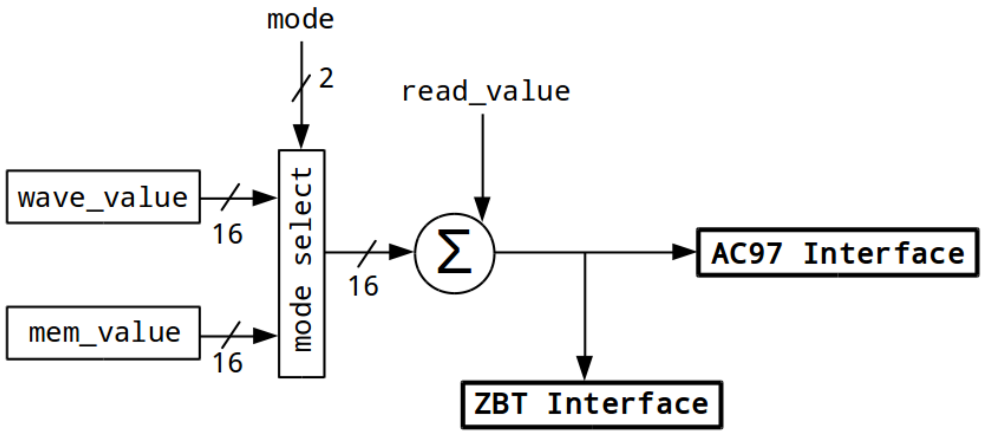

Figure 6: Mixer Module

An issue that had to be resolved with this module was correctly adding pseudo-signed values while also avoiding overflow in the sum. The samples acquired from the tone generator represent signed wave amplitude values in twos-complement form, but it was decided to not explicitly declare them as signed in order to avoid potential sign bugs. So instead, the mixer has to account for whether the new value being added to the sum is positive or negative. If the value or the current sum is negative (ie. greater than 0x7fff), then the mixer always adds the new value, since overflowing is required to make signed addition work. However, if both the sum and the value are positive, the mixer checks whether adding the value will “overflow” the sum past 0x7fff, which would be interpreted as a negative value. If this is so, the mixer sets the sum to 0x7fff (the maximum positive value), otherwise it adds the new value as normal.

When the mixer receives the `ready` signal, it sends the final sum to the AC97 interface and resets to zero.

5 Audio Output

Audio output is handled by the 6.111 labkit's LM4550 and AC97 codec. The AC97 interface module accepts samples from the mixer and is responsible for building the serial frames to be sent to the chip, as well as setting the internal state registers. Since our samples are 16 bits and the AC97 codec requires 20 bit samples, the interface also pads the samples with zeros in the least significant bits.

6 Control Module

The control module ensures that timing requirements are met and delays are accounted for when actions are taken in the system.

One of these controls is the `sel_angle` signal, which multiplexes the phase accumulators. Since there is a one clock-cycle delay from the `ready` signal to when the accumulators have stepped to their new values, the control module must also delay before resetting `sel_angle` to 0 and stepping through the accumulators. The control module selects each accumulator for two clock cycles, in order to give more leeway in the timing of the following modules and make it easier to coordinate. Once `sel_angle` reaches `N_NOTES`, it stops incrementing and the tone generator defaults to an output of 0, leading to no further relevant additions to the final signal.

The `read_value` signal makes sure that the mixer module only reads each tone value once and while it is not transitioning. It flips its value every clock cycle after the ready signal and is aligned with the changes in tone value to satisfy the above requirements. It continues to change after all the phase accumulators have been piped through, but since the mux is default-tied to zero, the mixer will just add zero to the sample, making no difference to the sample's value.

7 Recording

Unfortunately, the recorder module was unsuccessfully integrated, and suffered from bugs rendering it unusable. The bugs seemed to be in the driver, because the interface module appeared to work to specification. The ZBT interface module was the high-level module responsible for recording and playing back tracks. It abstracted the timing requirements of the ZBT memories, as well as their size, index location and the fact there were two devices being recorded to. Internal state is kept in order let the rest of the synthesizer system ignore the memory details. The states responsible for keeping track of the current memory location are r_index, w_index, and ram_select which indicate which addresses the memory will read from or write to, and on which device. The ZBT memories were used in series, such that when one RAM filled up, the system immediately continued recording to or playing from to the second memory.

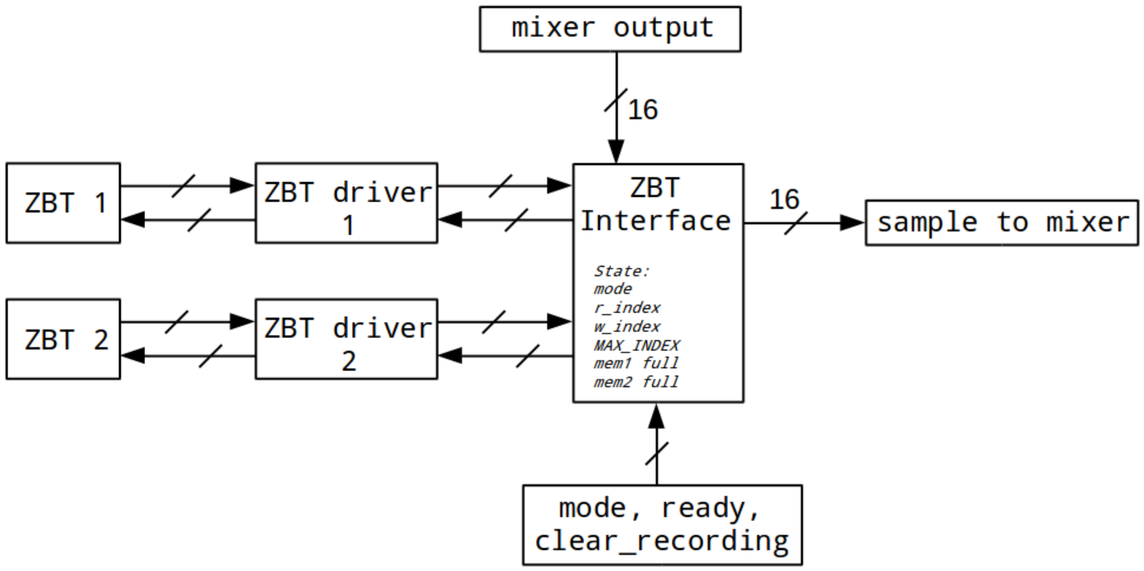

Figure 7: ZBT Interface Module

While in record mode, each time `ready` asserted (indicating a new sample was just provided to the DAC), the ZBT interface provided the sample to the appropriate ZBT driver (based on the value of `ram_select`), incremented `w_address`, and set the write-enable/clock-enable bits for the RAMs as necessary. If `w_address` reaches `MAX_RAM_ADDRESS` then either `ram_select` is swapped to the second ZBT (if the current RAM is the first ZBT) or recording stops because the memory has been filled. `w_index` is not reset between recordings. If the user deselects record mode and then reselects it, the recording will continue from where it left off. The user can "delete" the recording by pressing the `clear_recording` button which resets `w_index` to 0 and allows the user to record over their previous recording the next time they enter record mode.

Playback mode is similar except that instead of sending a sample to the ZBTs on the `ready` signal, the interface reads a value from the appropriate RAM and sends it to the mixer before incrementing `r_index`. If playback mode is deselected and then reselected, `r_index` is reset to 0 to play the track back from the beginning every time.

The ZBT drivers interface with each of the two ZBT RAMs and set up the correct timing delay between address and data latching. A deskewed clock was created with the ISE Core Generator, and directly passed to both drivers in order to have correct timing between the memory and the rest of the system.

8 ADSR Enveloping

ADSR (Attack-Delay-Sustain-Release) enveloping is a method used for synthesized instruments to make them sound more traditional. It creates variation in the magnitude of the sound over time so that it gradually increases and falls off as the key is held, rather than simply switching the tone on and off whenever the key is pressed or released.

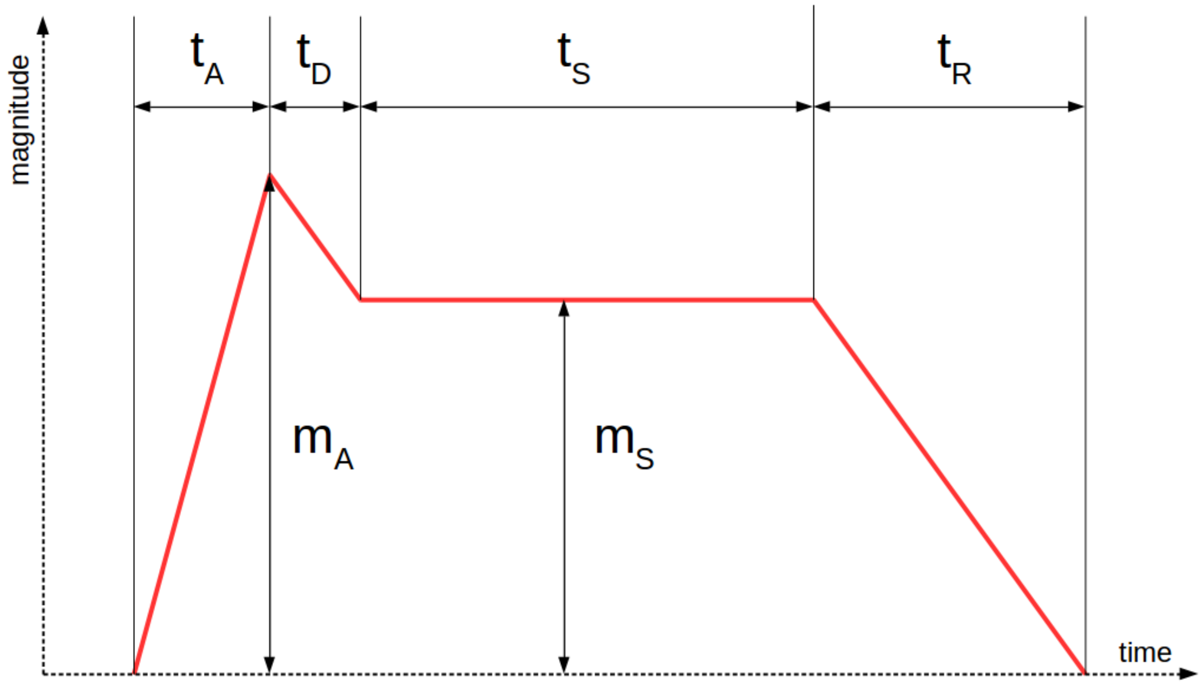

Figure 8: Attack-Delay-Sustain-Release Time and Magnitude Parameters

While ADSR Enveloping was not integrated into the project, adding it would be straightforward. First, state would be kept of which keys are currently active, so that the enveloping will start whenever a new note is played, but will proceed for keys that are still being pressed. An `envelope_reset` signal would be send to the ADSR module whenever a key is re-pressed. This could be tracked by seeing when M becomes 0, since whenever the user releases a key, there will be a non-zero number of clock cycles where there is no input since the clock is significantly faster than human reaction time. The module controlling the enveloping would have to keep a cycle count for each key in order to control where in the envelope the tone should be. To actually adjust the magnitude, the tones from the tone generator would be scaled to the desired magnitude. This could easily be done by arithmetically right-shifting the tone to get smaller magnitudes. The tone generator would then receive a value to shift by from the module implementing the ADSR control and shift the value before piping it to the mixer.

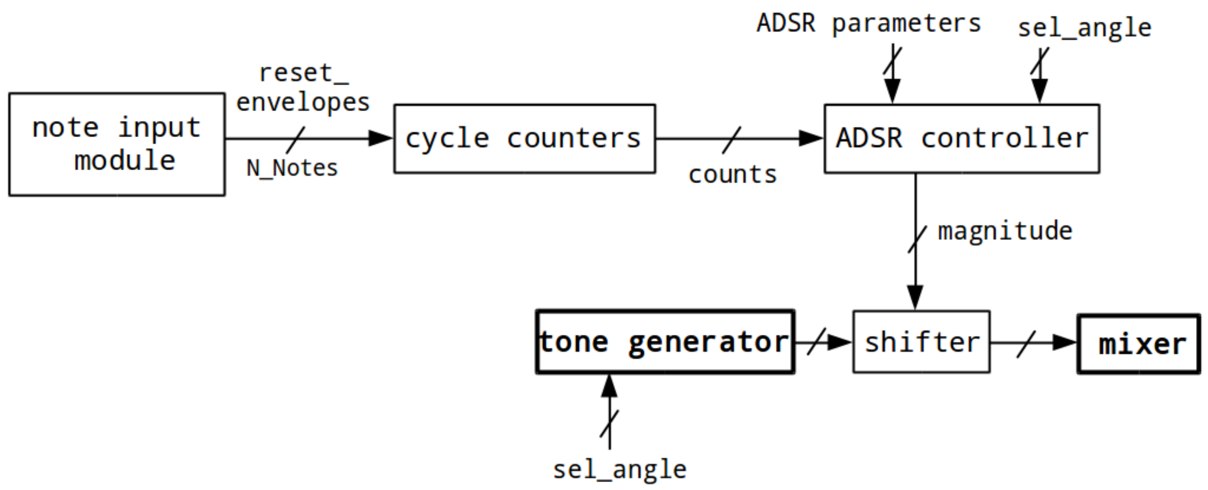

Figure 9: ADSR Module

The times for each envelope segment would be set by the user in quarter-second increments that could then be converted to clock cycles. The magnitudes will vary between 0 and 16, since that is the maximum number of bits by which the tone can be shifted (since it is 16 bits long). For simplicity, 16 will be considered the maximum and the shifting will be done by first subtracting the desired magnitude by 16 (so no shifting occurs when maximum magnitude is desired). The attack magnitude, m_A, will be automatically set to the maximum value of 16. Once the key press begins, every $t_A >> 4$ cycles, the magnitude of the tone will increment by one until it reaches the maximum.

The sustain magnitude, m_S will be adjustable by the user. Legal values of m_S will be 16, 14, 12, and 8. The reasoning behind this is so $(m_A - m_S)$ is always a power of two, allowing t_D to be divided into equal segments using a right shift by $\log_2(m_A - m_S)$ and avoiding a divider here. Once m_A has been reached, every $t_D >> \log_2(m_A - m_S)$ cycles, the magnitude is decremented by one until m_S is reached. The magnitude then holds at m_S for t_S cycles. Unfortunately, to calculate the steps for the release phase, t_R must be divided by m_S, so every t_R/m_S cycles, the magnitude is decremented by one until the magnitude reaches zero.

In short, the ADSR module would receive the `reset_envelope` signals from the note input module whenever a key is re-pressed. This signal would reset a cycle counter for each key repressed that will track where in the envelope the note should be. This count will be passed to the ADSR controller which, using the user-set parameters (the times and magnitudes), will determine what the current magnitude of the note should be. This value will be passed to the shifter added to the tone generator. Just like in the tone generator, this signal will be muxed to switch between each key by the `sel_angle` signal.

9 Reflection

The most headache-inducing part of the project were timing issues between the different modules. Solving them required scoping the intermediate outputs to confirm expected delays within the logic and designing the upstream modules to account for this delay. In addition, the overflow addition problem was a significant detractor from audio quality until I found the issue. Despite not getting memory to work and running out of time to implement ADSR Enveloping after getting the major bugs worked out of the tone generator, I am happy that I got the tone generator to work reasonably well and generate pure-sounding notes. I plan to improve my synthesizer in the future and make it work better.