EE108_Final_Report_PYNQ_Board.docx

Parthiv Krishna, STANFORD *Department of Electrical Engineering,* `parthiv@stanford.edu`
Sydney Marler, STANFORD *Department of Electrical Engineering,* `smarler@stanford.edu`
Rinni Bhansali, STANFORD *Department of Electrical Engineering,* `rinbha@stanford.edu`
Sathya Edamadaka, STANFORD *Department of Electrical Engineering,* `sath@stanford.edu`

I. High Level Design

Our system is an audio visual synthesizer that specializes in high quality, multi-harmonic, extremely realistic pieces. It supports writing *chords* with 4, concurrent notes at the same time, each of which can be played by a separate instrument. Instruments are abstractions for our implementation of *harmonics* and *ADSR enveloping (dynamics)*. In addition, our extremely detailed transcribed pieces include parameterized metadata to inform the system's panning and phaser effects, to round out the system's audio capabilities with *stereo capabilities*. By parameterizing each of the above with said metadata, we are able to accomplish *multiple voicing*, allowing multiple instruments to play different melodic lines at the same time like chords, each with individual ADSR envelopes, panning and phaser effects and harmonics. To better visualize what is going on in our synthesizer, a text-based *note display* reads out the chords currently playing to the screen below the waveform. Also displayed are the chords just played and the next chords to be played. These chords change in real-time along with the music.

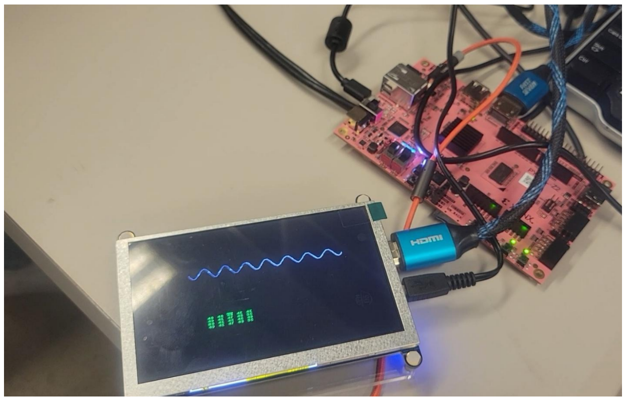

Fig. 1. Demo of the functionality of the synthesizer

On the visual front, we've also significantly extended our previous functionality by implementing a *chord display*, in which we publish the letter and number indicating each of the fundamental frequencies played by the instruments in each chord. In total, the display shows the current notes/chords, 2 notes/chords from the past, and 2 notes/chords from the future! All together, this is a total of 21 points.

Acknowledgements: The authors would like to thank our parents for their support, in addition to the wonderful EE108 teaching team: Jack, Sam, Chris, Luis, and Professor Mitra! 😊

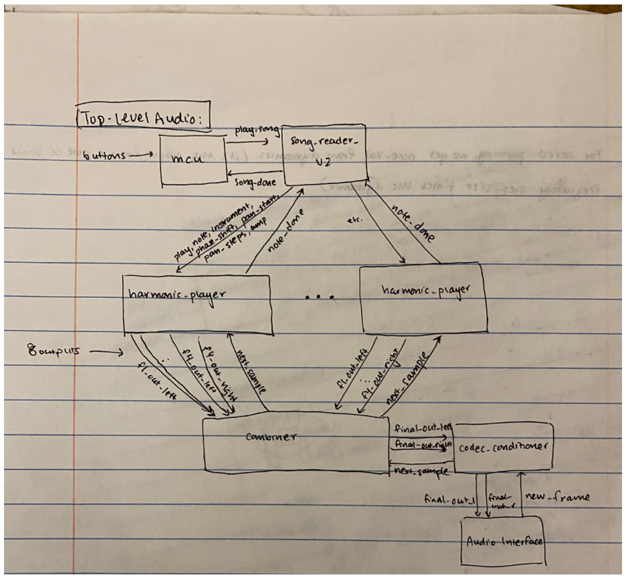

Fig. 2. Top-level block diagram for our entire system.

The audio interface, at the highest level, is described in Fig. 2. It works similarly to that of the interface given in Lab 5, but with `harmonic_player` replacing what was `note_player` to allow for our additional functionality.

Specifically, the buttons on our FPGA serve as inputs to the (unchanged) `mcu` module. This sends the signal `play_song` to our new `song_reader_v2`. Our song reader reads from an adjusted `song_rom` to get the notes to play (and their duration) for each of 4 voices, along with the metadata parameters (given to each note, for each voice) which specify: `instrument` (to determine the harmonic content of each note and its ADSR envelope), `phase_shift` (the speed of phase shifting), `pan_stage` (the initial panning), and `pan_steps_shift` (the amount of panning that is to occur in the note).

These parameters, the note, and its duration all go into `harmonic_player`. There are four copies of `harmonic_player` (one for each voice in our songs); each copy will output 8 samples from a sine wave. The 8 samples can be divided into 4 harmonic pairs, which get combined into 2 samples–one for the left channel of the codec, and one for the right. These samples enter the `codec_conditioner`, which will condition and send them to the audio interface. The interface will request a new sample from the conditioner when ready, which then asks harmonic_player for the next_sample when it is ready. Then harmonic_player sends note_done (based on its duration) to song_reader_v2 to request a new note. Eventually, song_reader_v2 will tell mcu when the song is done based on these signals.

II. Specification

A. Chords

For chords, we followed the specification laid out in the final project handout. We created a new module, song_reader_v2, that reads the updated song ROM format. It loads entries with associated notes, duration, and metadata and sends this information to a note_arbiter module, which helps distribute the note to an available harmonic_player. Once song_reader_v2 reaches an entry where the first bit is 1, it then advances time for the specified number of beats. See Fig. 3 for an FSM diagram explaining the states.

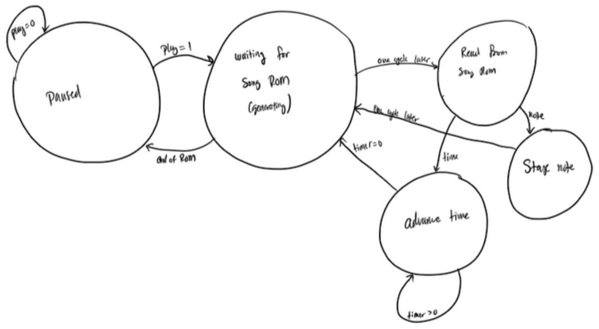

Fig. 3. FSM diagram for song_reader_v2.

The note_arbiter keeps track of which harmonic_players are playing notes, and when it receives a pulse on load_new_note, it passes this along to the first available harmonic_player (priority goes to 0, then 1, then 2, then 3). It stores the current note for each harmonic_player in a dedicated flip flop and updates the value stored in a given flip flop if that player is free. Finally, it adds the output signals from the four harmonic_players to get the merged sample.

With a successful implementation of chords that supports four concurrent notes, we believe we should receive 4 points.

B. Note Display

We implemented one of the suggested features in the final lab document under the "Visual" category: "Note Display" for 4 points. Our note display is able to display the two previous notes (if they exist), the current note, and the future two notes (if they exist). In the case of chords, the display shows the previous two chords, current chord, and future chords all at once.

Note display is an extended version of wave_display.v from lab 5. The chords are displayed in the empty screen space around the output waveform. The key implementation detail for the note display feature lies in ROMs. We wrote our own ROM (wd_rom.v) and then utilized one of the ROMs provided in previous labs (tcgrom.v). wd_rom is charged with returning the addresses that will be used in tcgrom and tcgrom contains information in the form of bitmaps of how to make letters and numbers. Note display relies on the logic established in note_player.v and song_reader.v to know when to change notes/chords. Note display is passed addresses whenever a note/chord is changed. To handle the case of when there are no more previous or future notes to be played, we check the address bits for overflow or underflow.

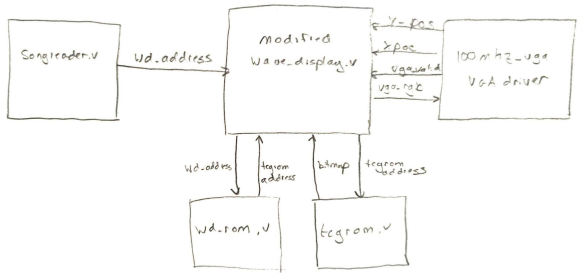

Fig. 4. Block diagram for note display implementation

We believe we should receive 4 points for this extension as we successfully displayed the past two notes/chords, the current note/chords and the next two notes/chords.

C. Harmonics

One of the core parts of our implementation was the ability to generate several harmonics from a single sample. We wanted to parameterize how loud each harmonic was when compared to each other, so we created an abstraction called an "instrument". An instrument is a specification that can be made for each note. We planned for four instruments: flute, oboe, clarinet, and guitar (sharp, purely-sine wave). For each of those instruments, we added a set of parameters to harmonics which detailed how to weight each harmonic frequency multiple compared to the initial fundamental frequency sample. We outputted four harmonics for each sample inputted: the fundamental frequency, f0, and 2x, 3x, and 4x the fundamental frequency (f1, f2, and f3 respectively).

First, let's overview how harmonics fits into place in the system. Harmonics, dynamics, and panning (described in stereo effects) are all invoked together in a harmonic player. Specifically, harmonics takes in a single note and produces several, scaled outputs representing the fundamental frequency and its multiples. Then, each of these scaled outputs and the duration of the original sample are inputted into dynamics get an ADSR-enveloped sample and duration. Lastly, this signal is panned between the left and right ears based on additional metadata for the particular note.

Let's first dive into harmonics. The block diagram for harmonics is shown below. Specifically, the module takes in a sine wave sample. It then uses a small array (or ROM) that we've defined that provides the amplitude multipliers for harmonic multiples of the original note given the instrument. For instance, if given instrument $2'b1$, the ROM would return

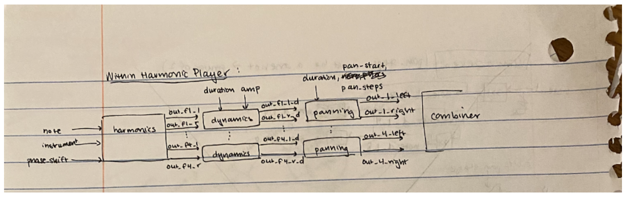

Fig. 5. Block diagram for harmonic player, which implements harmonics, dynamics, phasing, and panning.

the amplitudes of the second, third, and fourth harmonic notes for the 2nd instrument (clarinet). It then calls `note_player`, which we have custom defined, and will discuss more in stereo effects. The final output of the harmonics module is a series of 8 samples, 4 of which correspond to the four harmonic frequencies to be emitted to the left audio output channel, and the other 4 for the right channel.

We believe this qualifies for the 3 points because of how we've deeply parameterized our harmonics capabilities, as we've developed a method of not only outputting 4 different harmonics for each instrument, but also for each of the left and right headphone channels.

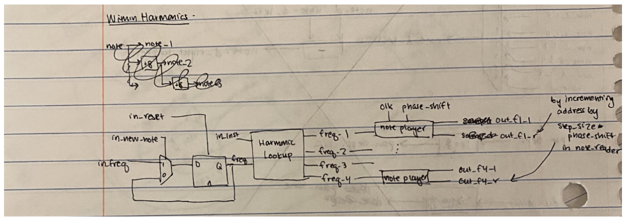

Fig. 6. Block diagram for harmonics.

D. ADSR Dynamics

We implemented ADSR enveloping (4 points as per the handout) for each individual sample being played (down to each harmonic)! We did so by creating a module, `dynamics`, that took in a single sample, the total duration of that sample, a `curr_timer` that represented how many beats we were into the note (generated by note_player, and at max equal to the total duration). In real time (with no flip flops), this module evaluated how to scale the amplitude of the note as to create an attack, decay, sustain, and release portion of a note. This was done by again creating a instrument-parameterized ROM that contained lengths of attack and decay sections for notes played by each instrument. This module works by constantly evaluating which section of the signal is being played by checking if `curr_timer` has surpassed the length of the attack section (in the decay state), if it's less than that (in the attack state), or greater than attack plus decay section. For the latter case, the remaining duration of the note minus the attack and decay phase lengths is halved, and each half is the duration of the sustain and release phases. When the timer surpasses the sum of the lengths of the attack and decay section, it drops into the sustain state, and after that, the release state.

The rate of change of the amplitude in the attack, decay, and release phases is exponential (accomplished by bit-shifting the

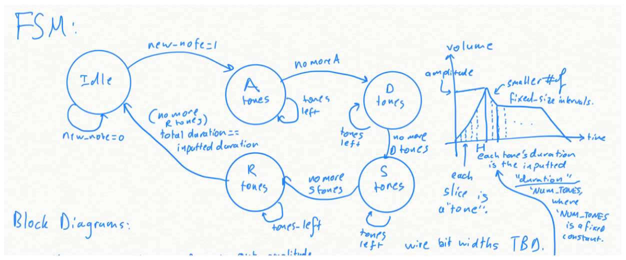

Fig. 7. FSM diagram for ADSR dynamics

note by 1 either left or right, appropriately), which translates well to human hearing, since humans hear in decibels (also on a log scale).

This relates to the diagrams shown below in that the "discretization" of our signal is accomplished by `curr_timer`.

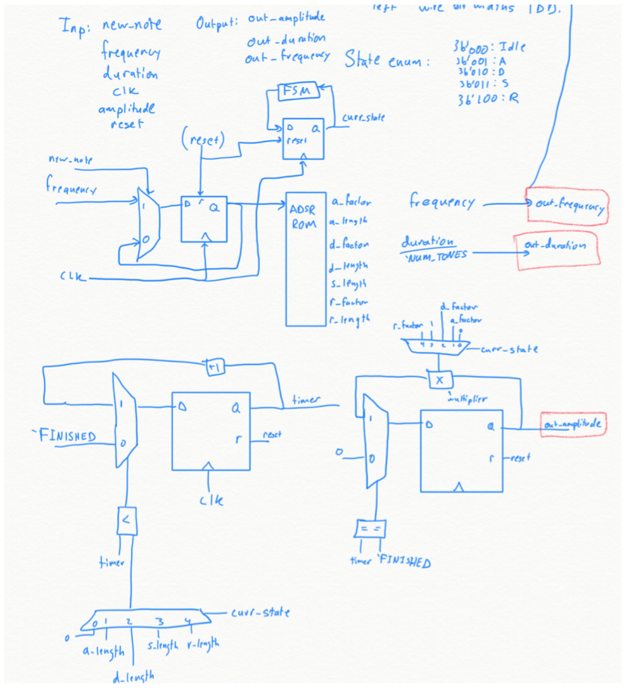

Fig. 8. Block diagram for ADSR dynamics

We believe that we've earned 4 points for this extension due to our successful implementation of a parametrizable ADSR envelope for dynamics.

E. Multiple Voices

By implementing chords, dynamics, and harmonics, allowing each resonant frequency to be played with a different timbre, we implemented the ability to play pieces with multiple voices. We used formant analysis to weight harmonics according to their strength in sound samples (e.g. implementing the primary formants in each instrument). In other words, we aimed to reproduce the sound of the chosen instruments via a combination of configuring harmonics parameters and ADSR values. Combining this with our existing chord functionality, and allowing the instrument of any given note to be configured via metadata in the song ROM, we successfully implemented multiple voices. We believe this qualifies us for an additional 3 points due to tuning the parameters to sound like a flute, clarinet, oboe, or guitar.

F. *Stereo Effects*

We enabled two types of stereo effects: panning and a phaser effect, for 3 points total.

1) Phaser: The first stereo effect, phasing, primarily consisted of adjusting the sine reader module, and propagating the changes up to the higher level blocks in our implementation. A phaser effect is created when the signals in the left and right channels are shifted in phase, where the shift varies in time. To implement this, we use the original `sine_reader` implementation to generate the addresses the `sine_reader` should use to read from the `sine_rom` on each clock cycle, according to the parameter `step_size`. This output goes to the output `sample_left`, which will ultimately be used to generate the left channel signal.

At the same time, we use a flip flop to generate the signal `delta_lr` in the shape of a triangle wave. This signal is used to define the phase shift (measured in samples) by which we shift the right channel output (essentially by incrementing the address, but using the quadrant-logic used in the original `sine_reader`); because the shift changes in time, the listener hears a "wah"-like phase effect.

Then, because `sine_reader` outputs two samples, `note_player` (which lies above `sine_reader`, from previous labs) will also output two samples, for the left and right channels. These outputs are used in `harmonics`, to generate the 2 left and right signals for each of the 4 harmonics of a note.

2) Panning: Once `harmonics` outputs 8 signals (4 left and 4 right), they all go through dynamics, where the ADSR envelope is applied to them. From there, each pair of signals (left and right) enters `panning`. Here, an additional volume envelope is applied to both signals. Specifically, given the input parameters `pan_start` and `pan_steps_shift` (in our implementation, these are stored in the `song_rom` as metadata), the module applies a pan between the left and right channels. The parameter `pan_start` describes which panning stage a given note should begin in. There are 5 possible panning stages (each stage defines the split between the left and right channels in total volume), described here:

1) 1 left, 0 right (so the left channel is at maximum volume, and the right channel has no volume).
2) $\frac{3}{4}$ left, $\frac{1}{4}$ right (so the left channel is at $\frac{3}{4}$ of its volume, and the right is at $\frac{1}{4}$ of its volume–the remaining stages follow this pattern).
3) $\frac{1}{2}$ left, $\frac{1}{2}$ right.
4) $\frac{1}{4}$ left, $\frac{1}{2}$ right.
5) 0 left, 1 right.

Then, the note will traverse through $2^{\texttt{pan\_steps\_shift}}$ panning stages in its duration (as we split the duration up into intervals of length `duration >>> pan_steps_shift`, and calculate which panning stage the signals must be in based on the current_time, which comes from note_player). This is described in the block diagram and FSM below.

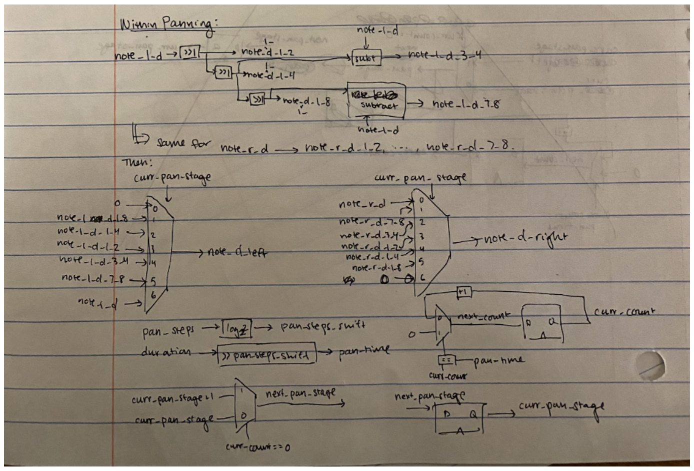

Fig. 9. Block diagram for panning

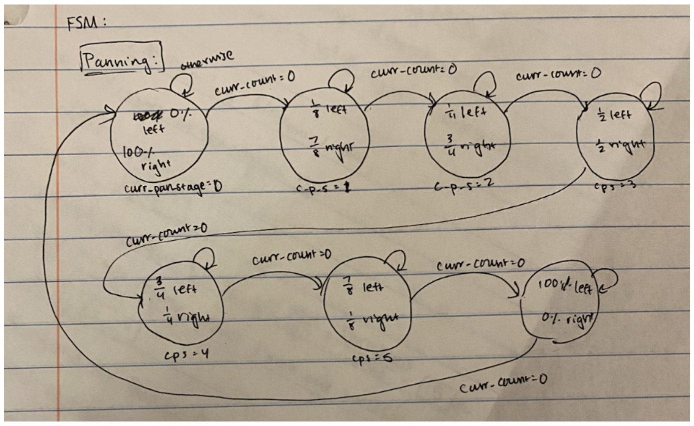

Fig. 10. FSM for panning

With a combination of both the stereo phaser effect and the stereo panning effect, we believe we qualify for an additional 3 points of extensions.

III. Usage

The board usage is the same as lab 5; no additional controls were added to the PYNQ board. The micro USB port on the left edge of the board provides power and the data used to program the device. The 3.5mm audio output and HDMI video output along the top of the board are used to hear and see the results of the synthesizer. Additionally, RGB LEDs on the bottom half of the board change colors as the samples change, another form of visual output. Along the bottom of the board, Reset, Play, and Next buttons allow for the user to interact with the synthesizer. Reset resets the state of the synthesizer, Play plays or pauses the current playback, and Next causes the synthesizer to jump to the next song (or to the first song if the current song is the last song). See Fig. 11 for a diagram of the various I/O on the PYNQ board.

IV. Key Implementation Details

A. *Chords*

One key implementation detail for Chords was ensuring the timing of the `new_sample_ready` signal. Af-

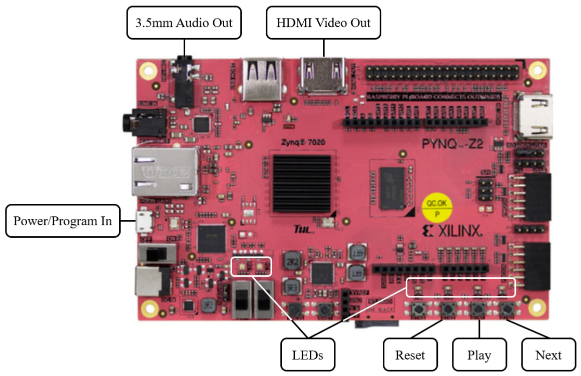

Fig. 11. Usage diagram

ter the `codec_conditioner` queries for a new sample by pulsing the `generate_next_sample` signal, it waits for the `new_sample_ready` signal to be pulsed before latching the current value of `new_sample_in`. This worked well when there was a single note_player that could set the `new_sample_ready` signal when it was done computing; however, our synthesizer now has four `harmonic_players` which in turn each have four `note_players`. We needed some way to combine the sixteen associated `new_sample_ready` signals so that a sample was not latched until all sixteen updates had occurred. Since there could be small timing differences in the amount of time it took for each one of the sixteen component samples to be computed, a simple AND or OR would not suffice.

Instead, we wrote a new `one_pulser` module. This module takes in N input signals (in this case, 4) that pulse. It outputs a single pulse when all signals have pulsed at least once since the last output pulse. This is useful to ensure that all `note_players` in a given `harmonic_player` are ready with a sample, and then that all `harmonic_players` in `music_player` are ready with a sample before telling the `codec_conditioner` that the sample is ready to be latched. See Fig. for two sample input/output sequences.

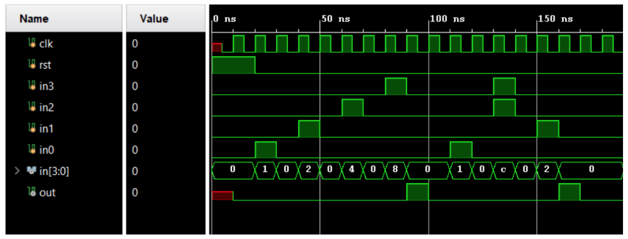

Fig. 12. One pulser testbench

B. Note Display

The note display feature builds off of the wave display testbench below. Edge case addresses are passed in as `wd_address[0:3]` because each chord can have up to 4 notes played at a time. One cycle later the ROM then returns the 18 bit addresses (9 bits per letter or number) that will be looked up in `tcgrom`.

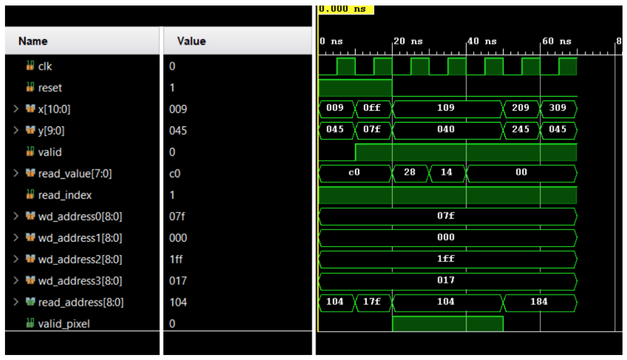

Fig. 13. The wires wd_address[0:3] are passed in from song_reader.v.

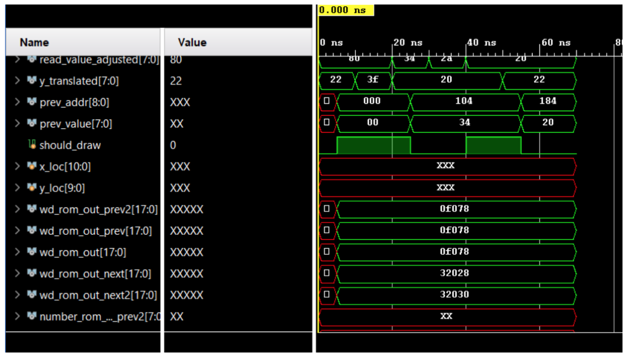

Fig. 14. One cycle later, wd_rom.v is updated with the address to lookup in tcgrom.v.

C. Harmonics

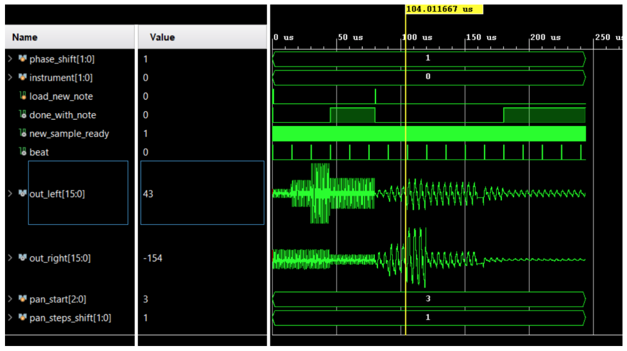

Fig. 15. Harmonic player testbench

Our testbench for harmonics player includes several different modules working together. We will only discuss harmonics here and a brief bit of interplay with stereo effects, since we discuss dynamics below and note player later on. It's clear that the waveform included in Fig 14 is not a uniform sine wave—it has several different frequencies overlayed on each other, all contributing to an overall left and overall right waveform. This shows that harmonics is working as intended! It's also important to note the phase shift in these waveforms—this changing phase shift is what provides the phaser effect when listening to the waveforms in different ears!

D. ADSR Dynamics

We can clearly see above (Fig. 15) how the different amplitudes of the sine waves of a certain note change according to the ADSR components! The first period of the attack phase can be seen in either of the waveforms. The length of the attack period is 3 for the flute (0th instrument), so we can clearly see how the amplitude of the note increases exponentially for 3 cycles (after each beat). Next, the decay period is 1 long, and thus we see the amplitude decayed once. Then, the amplitude was sustained for half of the remaining time, and it releases for the remaining time, as expected!

E. Stereo Effects

In Fig 16, the test bench for `note_player`, we can clearly see the variable phase shifting between the left and right outputs. One key implementation detail is that this phase shift goes from 0 to $\frac{1}{4}$ of the wave length. This triangular waveform is shown in Fig. 17 as `delta_lr`, in the test bench for for `sine_reader`.

In Fig. 15, we see panning for both notes. In particular, each note begins in stage 3, and goes to stage 4 midway through (giving the auditory effect of it moving from the left to the right ear). For `panning`, it is important to note that the actual number of stages a note can go through must be a power of 2 (as we cannot divide, so we must split `duration` into intervals with left-shifting). Also, in order to keep a smooth panning effect, the panning stages initially increment, then decrement once they reach the top, and the cycle continues (so a single note will first move right, then left, and so on).

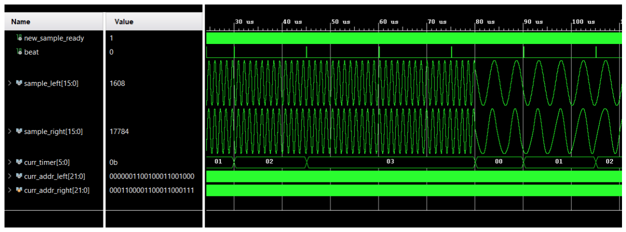

Fig. 16. Note player testbench

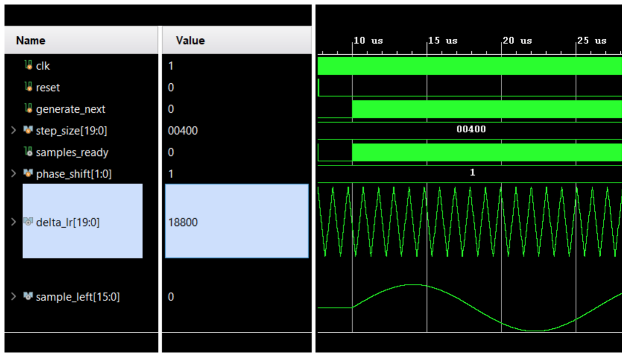

Fig. 17. Sine reader testbench

V. Problems

We ran into issues of overflow on several occassions. In general, because we are adding many 16 bit signed integer audio samples and creating a single 16 bit signed integer audio sample, we run the risk of overflowing the available space. The most straightforward approach we had to fixing this was to left-shift the values before adding. For example, if we were adding four samples to create one sample, we would first left-shift each sample by 2 (effectively a division by 4), then add the left-shifted samples together. By doing this, we ensured that we would not overflow after the addition. However, one thing that we had to correct was using `wire signed` and `>>>` to ensure that the first bit of the sample (sign bit) was not shifted.

Another problem we had was with properly handling the phase shift inside `sine_reader`. In particular, when we were attempting to convert from the current left address to the current right, phase-shifted address, we were simply subtracting `1024 - curr_left_addr[19:0]`. This led to some instances where we would have some zero-samples when our right phase-shifted sample reached the end of the ROM (see Fig. 18). At first, we thought we just had an off-by-one error where we just needed to subtract from 1023. However, we soon realized that this was still incorrect – we needed to also account for the fixed-point fractional bits in `curr_left_addr`. To solve this, we instead subtracted `{20{1'b1}} - curr_left_addr[19:0]` to get the correct right, phase-shifted address. We eventually ended up revamping the entire phase-shifting algorithm to support better dynamic phase-shift modulation, so this ended up not being part of our final code.

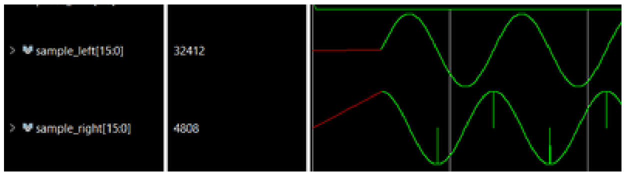

Fig. 18. Issue with `sine_reader`: zero samples on phase-shifted version

We encountered one particularly pernicious problem that caused significant distortions and delays. We realized that there were setup time violations that occurred between the beats flip flop and the one that assigned where in the sine rom to get notes from. This caused the `sine_reader` to miss every other beat, and thus cause notes to (non-deterministically) slow down, both in speed and frequency, by a factor of 0.5. After doing some analysis, we realized that if we pipelined our dataflow by adding another flip flop to store the quadrant of the sine wave that we'd want to read from, we could resolve this setup time violation! We thought this was a particularly subtle timing violation resolution, especially since it bears so much resemblance to our homework problems!