Such Super Awesome Final Report! wow

Andrei Terentiev
Alanna Tempest

Dynamics

We implemented a linear attack and exponential decay envelope in our dynamics module. This was accomplished by writing a python script which generates exponential roms. In our actual module we read from the rom and multiply our sample_outs amplitude by some number and then divide by 1024. So we are essentially multiplying by some fractional value e.g multiplying by 512 and dividing by 1024 would give us half our amplitude. The python script generation gives us the ability to make a wide variety of envelopes by tweaking the parameters.

Chords:

Chords was one of the most elaborate portions of our design. Initially we wanted to create 64 note players each responsible for a signal note. However, this proved to be quite ineffective. So we trimmed the number of note players down to 10, so we can play up to 10 notes at once. Each of these note players emits an in_use signal when it has a note loaded up. Just outside of the note player is an arbiter which finds a note player that is not in_use and sends a note and duration to that note player when the song_reader has read a new note. Each note player will emit a sample which we we send to our aggregator to generate the overall sample. The aggregator adds up each sample and then divides by the number of note players that are in use.

Enhanced Display

Enhanced display required routing the sample_out signals from each individual note player to the wave display module. From there, all that needed to be done was make multiple instantiations of preexisting code to generate multiple waves. Choosing the colors was fun.

Smooth waveform

Our implementation of waveform smoothing was fairly simple; basically, each vertical portion of each wave has a border on top and bottom of a color halfway between the wave and the black background. This helps the wave blend and not appear so jagged. Logic was also implemented to try to make the boundary pixels of taller waves (the ones that generally look worse) a little brighter for better blending, but this does not make that much of a difference.

Stereo:

For stereo we implemented a panning effect for the music. Essentially one speaker starts at full volume and the other is silent. After a certain amount of clock cycles one eighth of the sample shifts from one speaker to another. This goes back and forth to produce an interesting panning effect. Initially the stereo module was placed in our top module. However, because these samples were not sent through the codec conditioner, the output was quite staticy. This was fixed by inserting the stereo module inside our aggregator, so that three samples are sent to the codec conditioner and we end up with a steady signal. This change in the codec conditioner could allow various types of stereo effects to be implemented in the future.

Echo

The implementation of echo is probably the most interesting. We reuse the 1w2r RAM module provided for wave capture in Lab 5 to store wave samples, and once the RAM is full (prior to which the original sample is outputted), the stored samples are added to the incoming samples. Incoming samples overwrite old samples; the read address for stored samples is always one ahead of the write address (and serves as the next_address input to the address tracker flip flop). The samples stored in the RAM are scaled exponentially by 4 times duration, so they decay 4 times more quickly than played notes. Reverb is visible in the wave display, which is super cool.

Triangle/Square Wave (multiple voices):

Since harmonics never came through (see below), the triangle and square waves serve as multiple voices (in addition to the default sine wave). We use two bits in the song ROM metadata to indicate which type of wave to use. The meta signals are routed through and we case on them in the sine reader (which should probably be called the wave reader, since it reads the triangle and square waves too), which reads from the sine, triangle, and square ROMs.

Harmonics

For harmonics, we instantiated four sine readers whose step size inputs (frequency) are double, triple, quadruple, and quintuple the original step size (fundamental frequency). These are the first, second, third, and fourth harmonics. We scale and join these samples, then scale and join the resulting sample with the current input sample. While harmonics works in all testbenches, in synthesis the waves are jagged and the sound is unclean. Harmonics was one of the first modules we implemented, so we spent a lot of time trying to debug this module, with no luck.

High Level Design: We started off by building off our lab4 code. We altered the song reader to be able to read the format that was specified in the hand out. Then we had to alter the link between the song reader and the codec conditioner. We created a new module called the sample generator. This contains an arbiter, several note players, and an aggregator. The note player has all the effects built into is such as harmonics, dynamics, and echo. The arbiter says the notes into various note players which then send their samples off to the aggregator which combines all the samples, and then sends these off to the conditioner. We altered the codec conditioner to take in a left, right and full sample for stereo effects. At a high level the transition between our visual extensions and lab 5 were minimal.

Key Implementation Details: There were three places where we believed we had a pretty cool implementation.

Stereo: Initially we had tried to generate the stereo signals in the top module. However this led to a staticy signal that was not very signal. In order to combat this we changed the location of the stereo signal to inside of the music player, inside of the aggregator. This allows the codec conditioner to condition the samples which makes the signal significantly less staticy and we were able to generate a pretty good panning effect.

Sample_Generator: The way that we distributed our notes allowed for a lot of flexibility in the amount of note players and also makes it quite easy to have separate note players generate different effects. It ensures that notes are always being properly routed.

Echo:
We used the provided 1w2r RAM module from lab5. Instead of doing a read-index, we repurposed the RAM so that we were doing simultaneous reads and writes. Once the RAM is full for reading, the write address follows one address behind the read address, overwriting the old samples right after they are read. Close following saves space in our RAM. Also, because of the write enable signal of the RAM, we needed to update our exponential ROM to send out a sample_ready signal. Echo takes a few cycles but it outputs a sample_ready signal with its final signal.

Problems:
One pretty big problem that we ran into was in a combinational loop in our logic inside of the sample_generator. Essentially there were a lot of dependencies on the in_use signal, and after endless debugging we realised that the in_use signal actually depended on itself in a very indirect way. However, we managed to break up the logic to get rid of this combinational loop.

Another area we were having issues with was timing constraints. One source of timing constraints were when we were doing large multiplications such as in our aggregator and in our harmonics module. In order to combat this we decided that we would have to give up some of the accuracy that we had when generating these samples, in order to meet the timing constraints. We noticed that using only powers of two, even though we were technically losing accuracy, could not be detected by the human year.

A smaller problem in regards to timing constraints was simply due to the fact that our critical path was too long. Inserting flip flops in key areas - while keeping an eye on our overall logic timing - made it possible to break up the critical path and meet timing constraints.