Team Bayarmochin: Nomin-Erdene **Bayar**saikhan, Naomi **Mo**, Sylvia **Chin**
12/13/2023

PROJECT OVERVIEW

We implemented a music synthesizer with a comprehensive set of switch-toggled audio effects; additional effects implemented include echo, harmonics (tuned to sound like an organ), dynamics, and stereo changes (switching between left and right audio channels). The music synthesizer is also capable of playing up to three notes at a given instance of time.

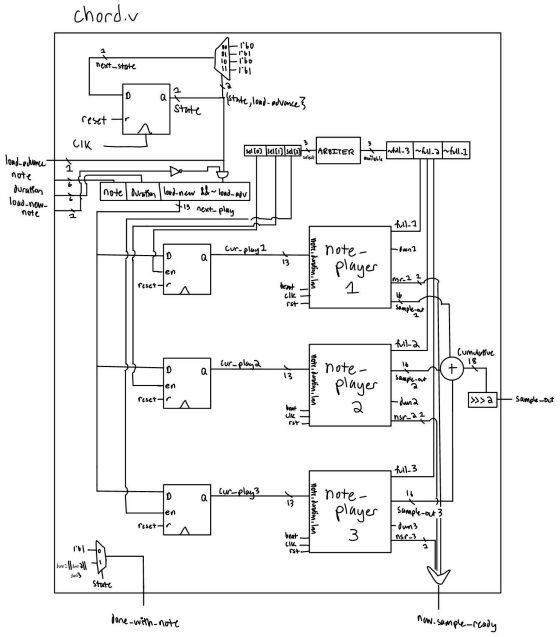
Users can toggle various audio effects using the add-on switch board; effects can be enabled simultaneously. The following list maps switches to their corresponding audio effect:

- ➢ SWA - Echo
- ➢ SWB - Harmonics
- ➢ SWC - Dynamics (decay)
- ➢ SWD - Stereo Effects

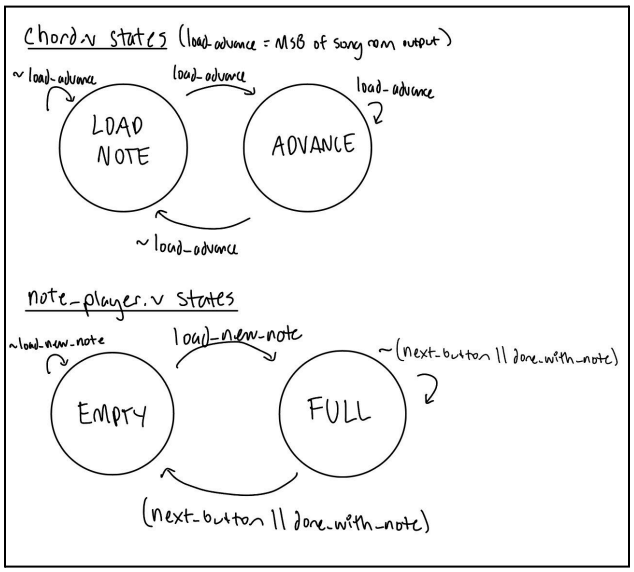
FEATURES

CHORDS | *4 points, implemented by Naomi*
Specification: Module that loads up to 3 note_players with loaded notes from the song_rom, then advances the note_players when the song_rom is "advancing" (MSB = 1).

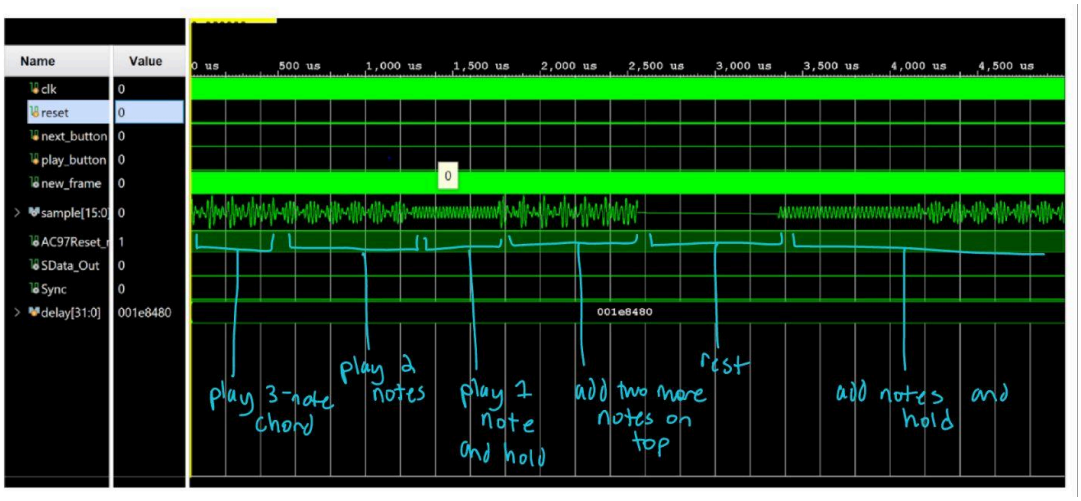
Block Diagram:



State Diagrams:



Simulation Results:



We tested multiple areas of functionality of chord.v using music_player_tb.v. We tried loading 1, 2, and 3 notes, as well as loading rests. We also tested holding notes over multiple loading periods. We verified functionality by visually inspecting the output waveform.

Usage: Constant feature, not toggled by the user.

High-level Design: To implement chord functionality, we created a new module chord.v that combines the outputs of three different note_players (nominally 1, 2, and 3) by roughly averaging their samples (summing and then dividing by 4) into a combined chord waveform. This sample_out is what goes to the codec. To do this, we had to modify the note_player module by adding a flip-flop to keep track of whether it is "empty" or "full" (whether or not it has been loaded by a note from the song_rom). Based on which note_players are full, we are able to designate song_rom notes to a specific note_player module. Inputs to the note_players are flip-flops enabled by the empty/full state of that note_player, and the note_players are only advanced when the chords module is in the "advance" stage.

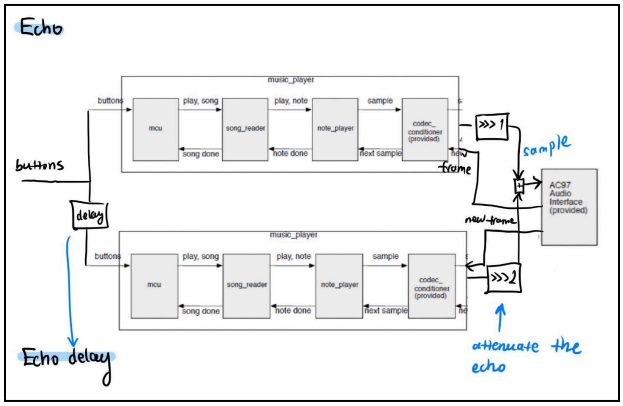
Key Implementation Details: Using an arbiter to determine which note_player to load was a clever solution for streamlining assignment priority. The arbiter is loaded with a 3-bit value; each bit indicates whether a note_player is full (1) or empty (0). The output value of the arbiter is a one-hot value indicating the first available note_player to load.

Challenges: I had frequent timing errors on the communication between note_player and chord to make sure that they were signaling each other properly. To solve this, I experimented with modifying some of the outputs of note_player such as note_done.

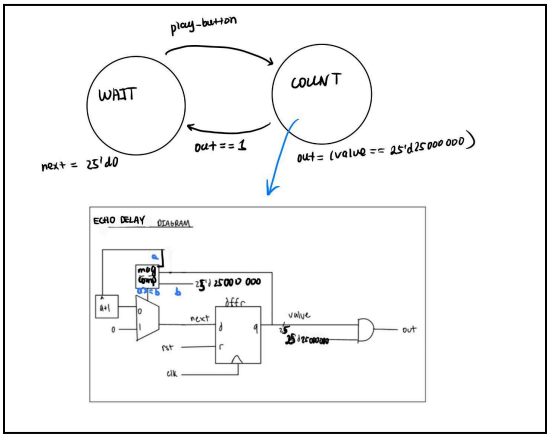
ECHO | *2 points, implemented by Nomin*

Specification: Module that causes an attenuated, echoed note to be played after an initial note.

Block Diagram:

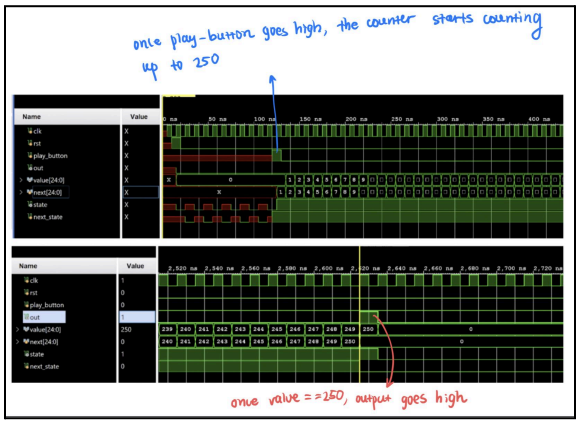


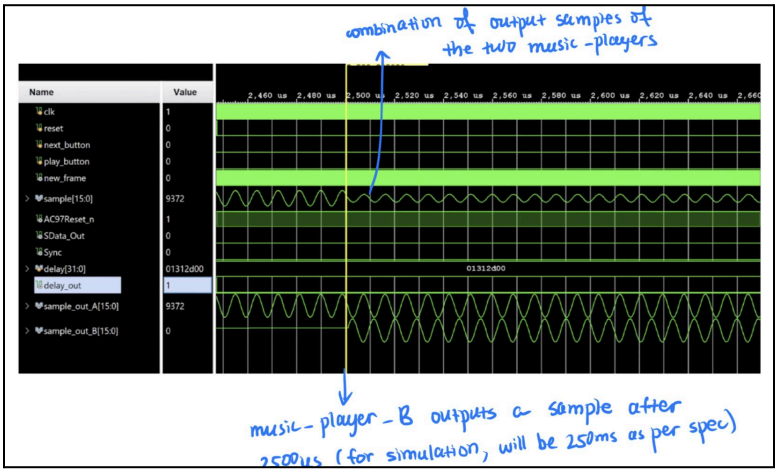
State Diagrams:



Simulation Results: echo_delay and echo

Here, we count up to 250 instead of 25,000,000 for simulation purposes. We only start the counter when the play_button is pressed (want to delay the effects of play_button), which is reflected in the state diagram of Echo delay.





Usage: Toggle the the switch SWA on the add-on switch board on and off

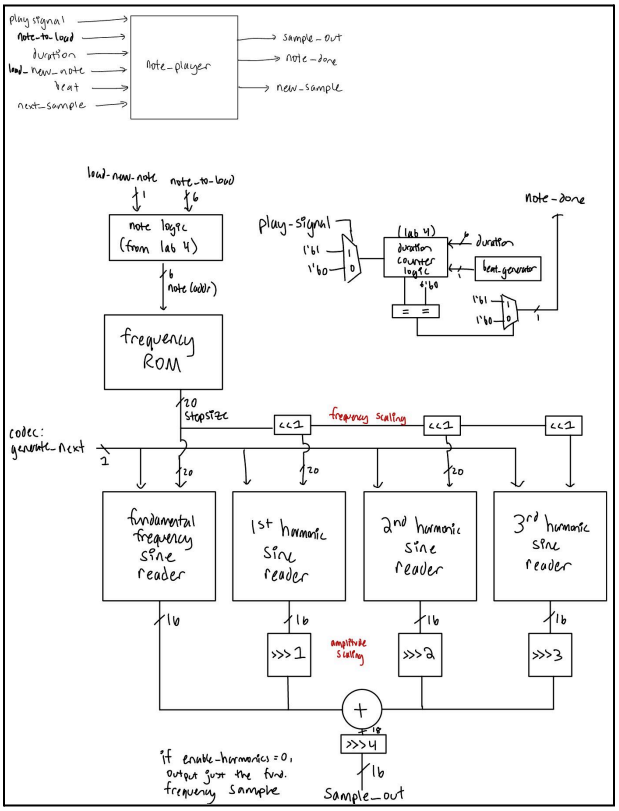
High-level Design: We implemented echo by instantiating two music_player modules. We use an additional module, echo_delay, that delays the play_button signal to the second music_player by a certain amount. In our implementation, we chose our delay to be 250ms (which is within the specified 100-500ms range).

Key Implementation Details: In echo implementation, I wrote a separate module echo_delay.v to handle the counter logic, which makes the main echo.v module simpler.

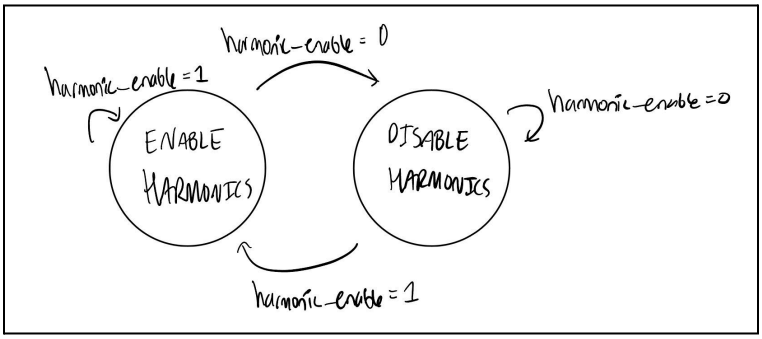
Challenges: In the initial design, I had it so that the counter starts counting as soon the system starts–goes high after 250ms. However, I had to modify echo_delay so that the counter has kind of a "beat" feature, similar to beat32, so that the output of echo_delay goes high *every time* play_button is pressed.

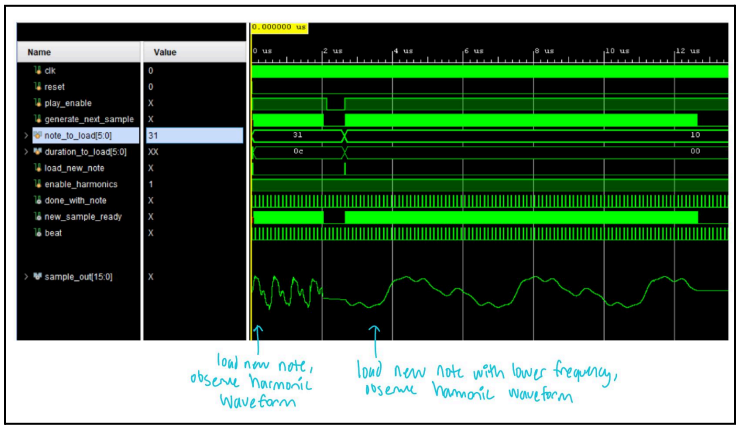
HARMONICS (Organ) | *3 points, implemented by Naomi*
Specification: Enables an organ-like sound quality to a given note by adding attenuated harmonic frequencies to the note's fundamental frequency.

Block Diagram:



State Diagram and Simulation:





Usage: Switch SWB toggles the layering of the harmonic frequencies.

High-level Design: For our implementation, we added three harmonic frequencies to a given fundamental frequency (the 2nd, 4th, and 8th); we then took the harmonic frequencies, scaled them down in amplitude (by factors of 1/2, 1/4, and 1/8 respectively), summed each of the harmonic frequencies together, took an approximate of the average (divided by 4), and then outputted this value as sample_out. The result is a harmonic waveform that produces a sound

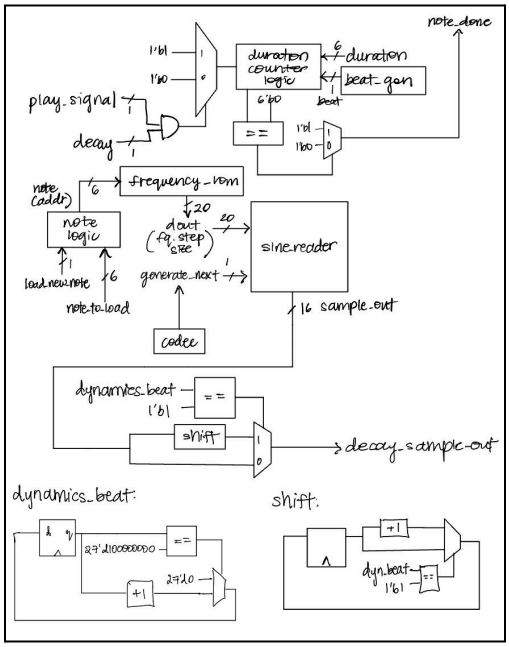
similar to that of a church organ. We also introduce new states to the note_player module that allows us to hard code when we want to introduce harmonics to a given note. When the 1-bit input wire harmonic_enable is driven high (in other words, when the corresponding feature switch is high), harmonics for the note will be enabled. When it is low, the regular note without harmonics will be output.

Key Implementation Details: Using https://www.falstad.com/fourier/ was a great way to get a qualitative understanding of harmonic outputs before having to synthesize to hardware; I used it to determine how I wanted to shift + attenuate the harmonic frequencies.

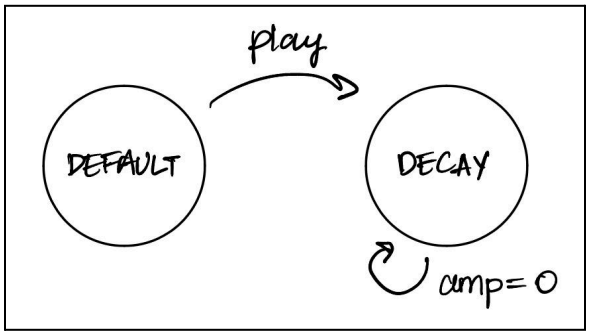
Challenges: One challenge was determining a compromise between attenuating signals to avoid clipping when summing vs. preserving integrity of the output volume. My final approach instantiates longer-length wires to store the sum of frequencies, and then subsequently attenuates by a factor of 2. I found this to be an optimal compromise.

DYNAMICS | *3 points, implemented by Sylvia*
Specification: Module that exponentially decays the amplitude (volume) of an output wave.

Block Diagram:



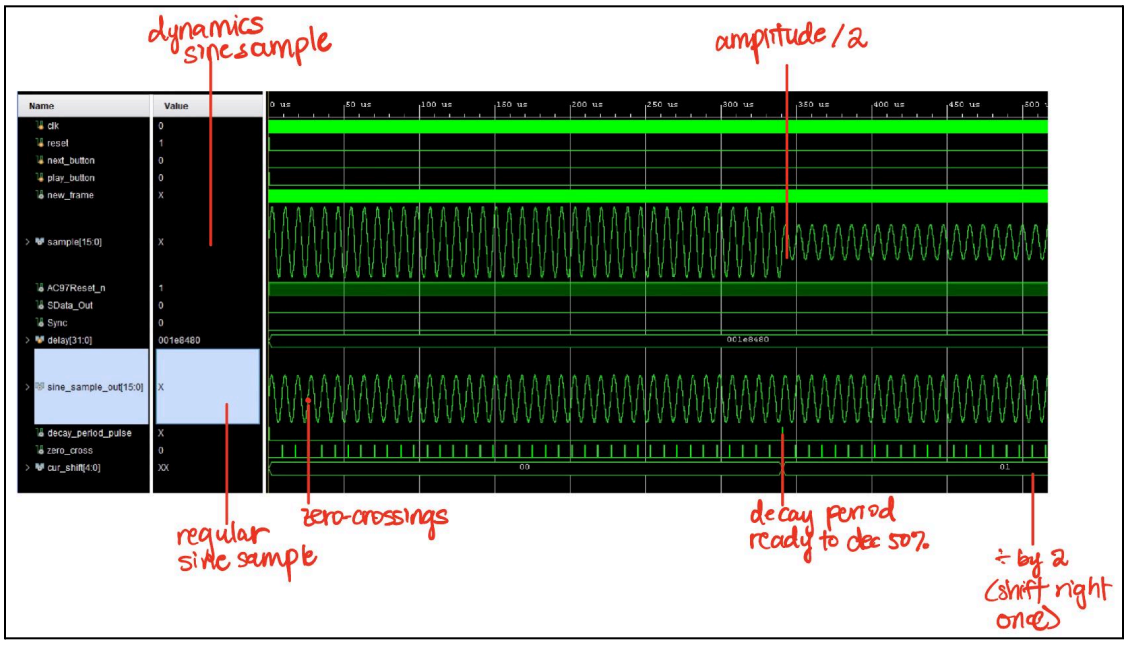
State Diagram:



Timing Diagram:

Cycle	play_enable	dynamics_en	attack_en	note_to_load[5:0]	duration_t	load_new_note	beat (/48)	dynamics_beat	shift	done_with_note	sample_out[15:0]	decay_sample_out
1	1	1		d'49	48/48 (1 wi	1	1	...		0	sinusoid	
2	1	1		d'49	...	0	2	100M	1	0	sinusoid	sample_out >>> shif
3	1	1		d'49		0	3	...		0	sinusoid	sample_out >>> shif
4	1	1		d'49		0	4			0	sinusoid	sample_out >>> shif
5	1	1		d'49		0	5			0	sinusoid	sample_out >>> shif
6	1	1		d'49		0	6			0	sinusoid	sample_out >>> shif
7	1	1		d'49		0	...			0	sinusoid	sample_out >>> shif
8	1	1		d'49		0	75	200M	2	0	sinusoid	sample_out >>> shif
9	1	1		1 d'49		0		...		0	sinusoid	sample_out >>> shif
10 ...		1		1 d'49		0				0	sinusoid	sample_out >>> shif
11		1		1 d'49		0				1	sinusoid	sample_out >>> shif
12		...	...	...		...	...			...	...	sample_out >>> shif
13												sample_out >>> shif
14									3			sample_out <<< shif
15												...

Simulation Results:



Usage: Toggle the the switch SWC (decay) on the add-on switch board on and off

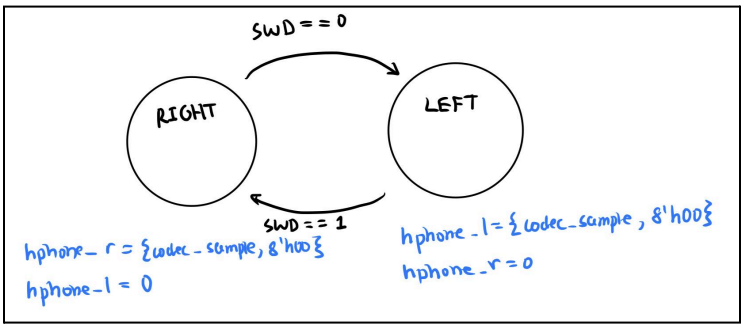
High-level Design: Signed shift the sample output by a factor of 1 (equivalent to dividing by 2) for each dynamics_echo "beat" when the switch is toggled. Beats for decay were determined by testing what was best audible.

Key Implementation Details: We can adjust note_player to accommodate the exponential decay of the sinusoid over the course of the sinusoid it receives from the sine_reader. Since this system is open-ended, we can declare the decay factor to be 50%. For example, we approximately expect the note to go from its given 100% → 50% → 25% …→ 0% using similar counting logic from Lab4 and Lab5. In order for the decay to be audible, we arbitrarily slowed it down to decay every 31 dynamic beats.

Challenges: Integrating with the other features was a challenge because we needed to include additional logic that chose which sine wave to output. However, it ended up working out quite well since we used switches to choose whether to decay or attack or not.

STEREO EFFECTS | *1 point, implemented by Nomin and Sylvia*
Specification: Module that selects the output channel of the audio; when the switch is turned on, the audio is output on the right side of the headphones/earbuds.

State Diagram:



Usage: Toggle the the switch SWD on the add-on switch board on and off

High-level Design: For this implementation, we simply had to introduce a logic that decides where the output from the echo/high level module goes. If the switch SWD is high, the output sample goes to the right side of the headphone, and if it's low, it goes to the left side.

Key Implementation Details: We implemented this feature in the top file, so it nicely did not have any effect on low level implementations. Implementation of stereo effects also did not introduce any additional timing constraints that would affect our other functionality, so we did not have to worry about timing errors.

Challenges: One challenge we had was ensuring that we updated the logic for both headphone right and left.