FPGA Music Synthesizer

Luke Qiao
Built with: Aarav Wattal, Joe Li

Project Overview

This project presents the design and implementation of a music synthesizer fully realized on a Field Programmable Gate Array (FPGA). Unlike traditional software-driven approaches, this system leverages application-specific hardware to efficiently deliver audio synthesis. Check out the project demo here: FPGA Music Synthesizer Demo.mov

The main objectives of this project are:

- Develop a high-performance music synthesizer with preloaded song sequences implemented purely in hardware.
- Integrate machine learning (ML) for emotion prediction into the hardware pipeline.
- Demonstrate advanced digital signal processing (DSP) and sound synthesis techniques with hardware efficiency.
- Provide real-time audio-visual feedback through an external display controlled using a VGA driver feeding into a HDMI/DVI encoder.

The main features are:

- **Preloaded Songs:** The music synthesizer has the ability to play preprogrammed songs stored in ROM, controlled by the play and pause buttons.
- **Emotion Prediction:** Uses a feedforward neural network (FNN) implemented in hardware to predict the emotion of the current song playing. The FNN architecture is lightweight, making it suitable for processing one-dimensional data streams in this application, ensuring efficient inference within FPGA resource constraints. The predicted emotion is displayed as the color of the waveform on the external display.
- **Chords:** The ability to play multiple notes simultaneously, enabling the synthesizer to output harmonies. The duration of each note in the chord can be individually controlled with a custom protocol in ROM to achieve the illusion of multiple lines in the music.
- **DSP & Harmonic Synthesis:** The sound generation of the synthesizer leverages hardware DSP blocks for precise waveform shaping and additive harmonic synthesis. Supports multiple harmonics to generate various characteristic waveform timbres. Each note also follows an Attack, Decay, Sustain, and Release (ADSR) envelope. The envelope is implemented as hardware-controlled modules, which provides fine-grained control of note dynamics, simulating acoustic instrument behavior. The notes also can be programmed to have the desired dynamics.
- **Read-Only Memory (ROM)**: Preloaded waveform samples for sine waves and lookup tables are stored in on-chip ROMs. This facilitates efficient retrieval of the waveforms without significant memory overhead, which ensures rapid access for high-frequency audio synthesis.
- **Random-Access Memory (RAM):** This project implements a dual-port RAM for the module displaying the waveform to the external screen. This allows the modules to write data into an address and allows multiple modules to simultaneously access the same

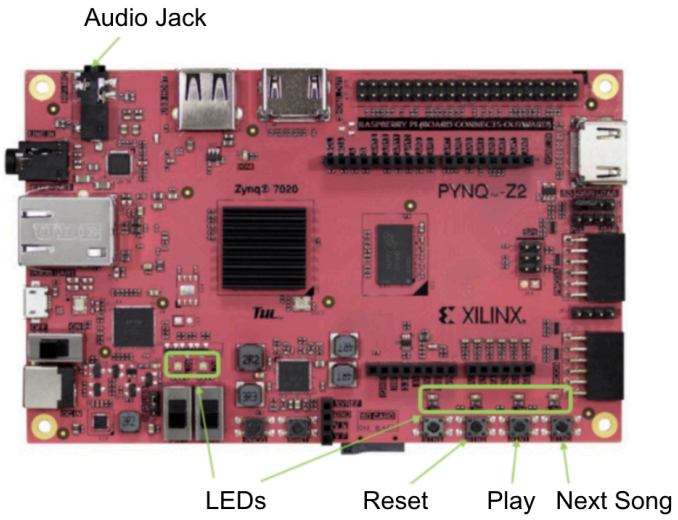
memory independently. To prevent issues that arise from writing and reading to the same address at the same time, we use double buffering. This involves splitting the addresses in RAM in half, and keeping track of a read index bit that determines which half of the RAM to read to/write from such that no two modules read and write from the same half in a single clock cycle.

- **VGA Driver for External Screen:** An integrated video controller outputs the synthesized waveform and emotion states via a VGA driver feeding into a HDMI/DVI encoder. Provides an intuitive interface for both performance and debugging.

This project demonstrates the capability of FPGA platforms to serve not only as real-time audio synthesizers but also as hardware accelerators for embedded machine learning tasks. By uniting DSP, ML, and interactive visualization within a single FPGA system, it highlights the potential for scalable, low-latency, and adaptive digital instruments.

Hardware Overview

The project uses the built-in XILINX FPGA buttons and audio jack and an external screen. The LEDs light up when the synthesizer is playing notes.

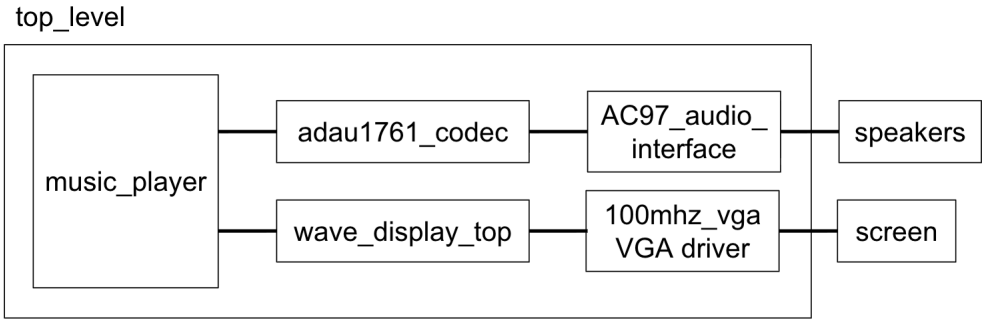


High-Level Design

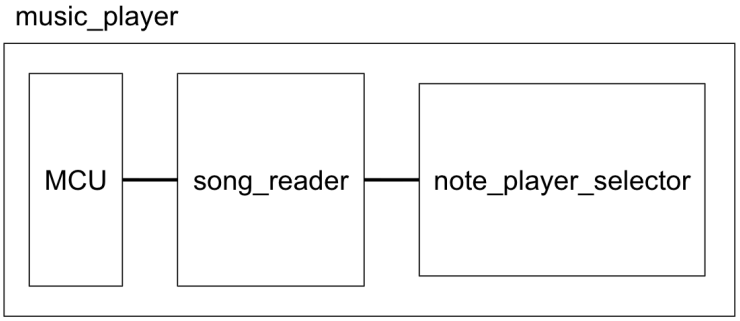
The **top_level** module contains the **music_player** module, which is responsible for handling the logic for the current song and determining which notes to play. For the high-level design, the output of the **music_player** module is fed into the **wave_display_top** module to the screen, and the audio output is fed into the **adau1761_codec** conditioner to the speakers.

The pipelining of the **music_player** module is: **song_reader** → **note_player_selector** → **adau1761_codec**. The pipelining of the **note_player_selector** module is: **note_player** (3) → **frequency_rom** → **sine_reader** (4) → **sine_rom** and **dynamic_rom**.

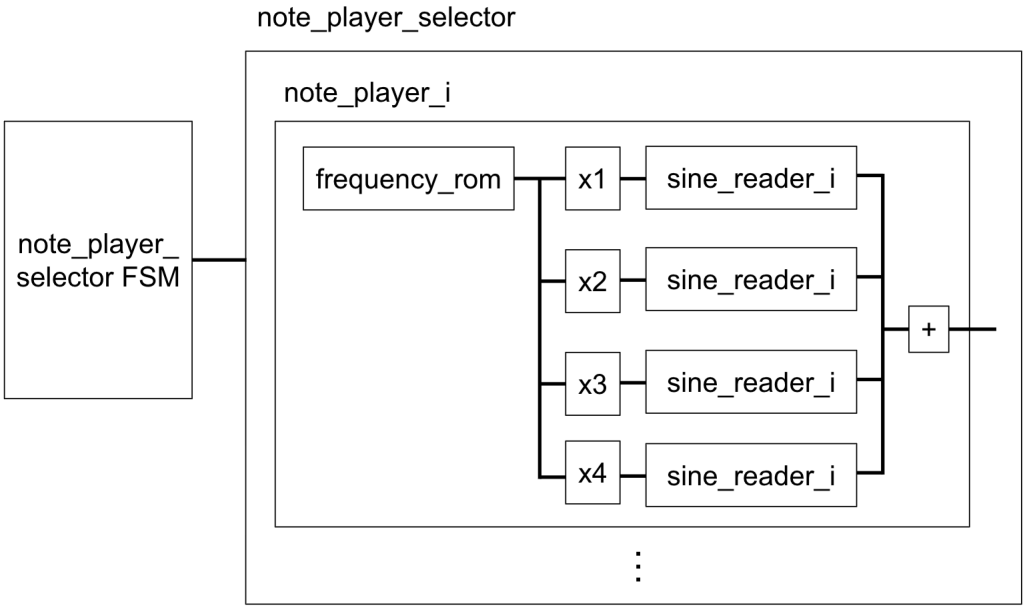
Top-Level Block Diagram



Music Player Block Diagram



Note Player Selector Block Diagram



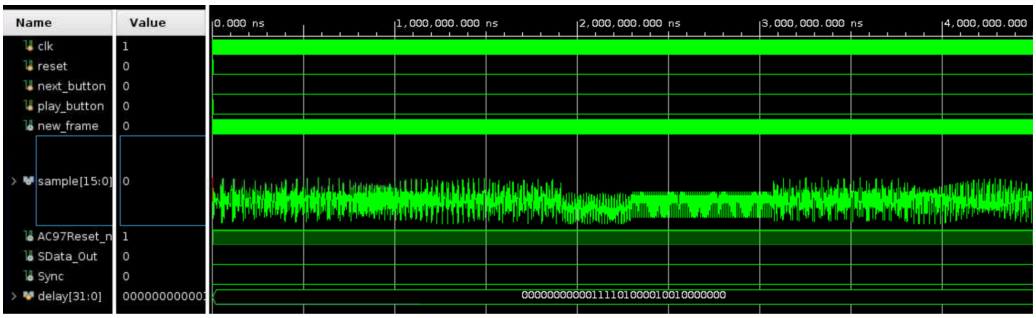
Note that we have $i \in [1, 3]$.

Core System Modules

Music Player

The **music-player** module is the highest level module relating to synthesizing the proper notes to be displayed on the screen and played on the speakers. Its block diagram is shown above in the high-level design.

Music Player Testbench

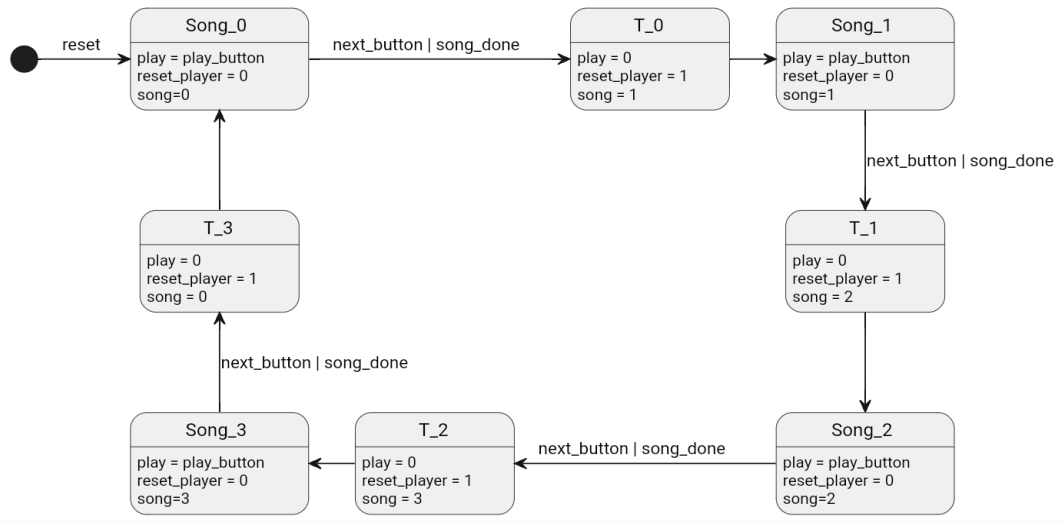


The test strategy was letting the music player module run and recording the output waveform. We show the first 4 million nanoseconds of the simulation for the sake of space. We see that after the play button is pressed, the output samples start being generated to create the audio signal, as expected. The signal looks more complicated than a pure sine wave, as 3 different note frequencies in the chord are added together, in addition to 4 harmonics per note.

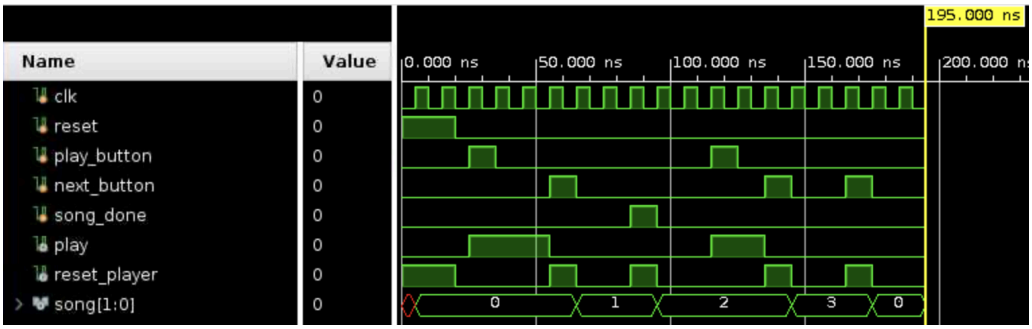
MCU

The **MCU** is a sub-module of **music_player**. It initially starts in its reset state. When the play button is pressed, the **MCU** will signal to begin playing the current song. When the song finishes, the **MCU** should wait in the paused state at the beginning of the next song. If the next button, the **MCU** instructs the **song_reader** to go to the beginning of the next song and pause.

MCU FSM



MCU Testbench

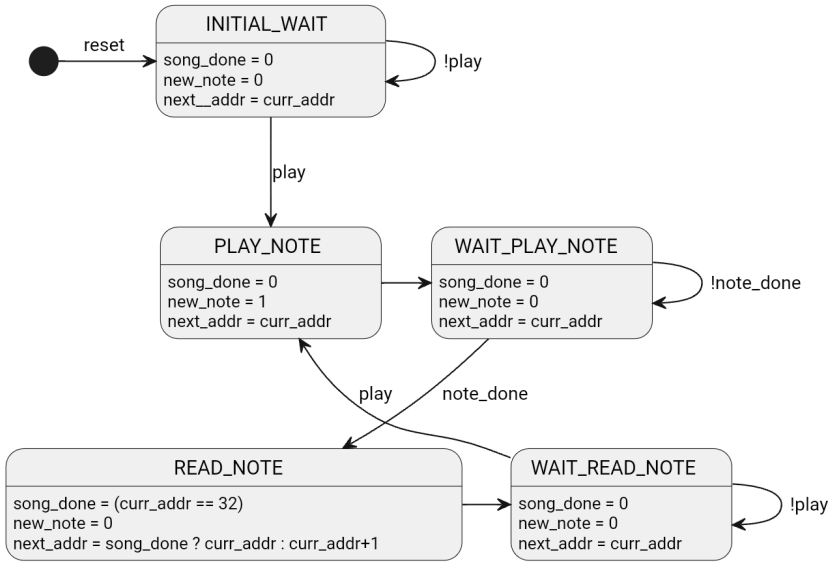


We see that when the play button is pressed, the play output goes high until the song_done signal goes high, at which point play goes low and reset_player goes high, as expected because the MCU should reset the player and start out paused at the next song. We also see that the song is correctly incremented every time song_done goes high.

Song Reader

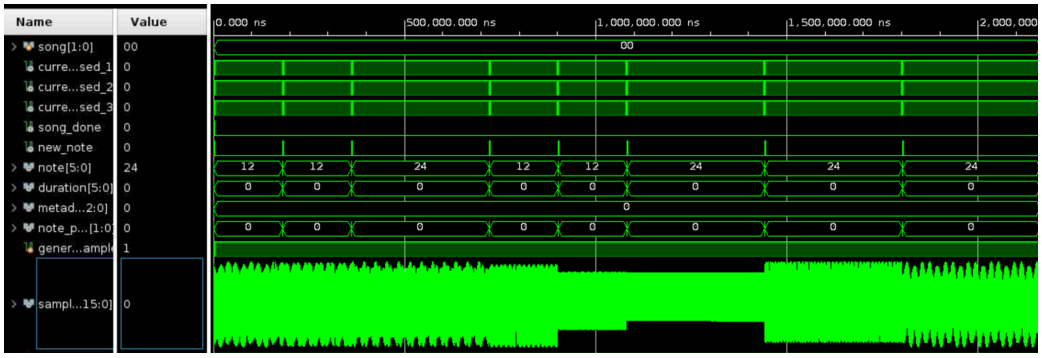
The **song_reader** is responsible for determining which note to play given the provided **song_rom**, generated using a custom script. The **song_reader** takes in the current song to read from the MCU and reads from the **song_rom** (note that this reading operation takes one clock cycle). It then sends the new note and the duration on to the **note_player**. The **song_reader** then waits for the **note_player** to tell it that it has finished the note. When the **note_player** finishes playing the current note, the **song_reader** looks up the next note and sends it to the **note_player**. This repeats until the **song_reader** has finished the current song, at which point it tells the MCU that the song is done.

Song Reader FSM



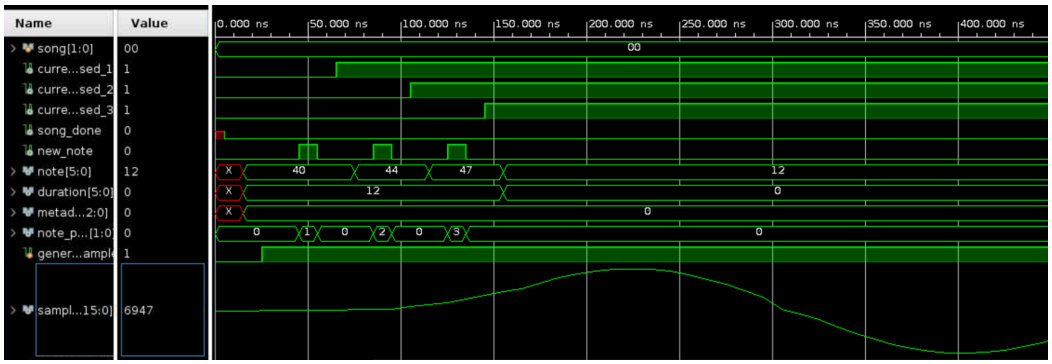
For all states: {note, duration} = ROM[{song, curr_addr}]

Song Reader Testbench



The testing strategy was letting the song reader module run and recording the output waveform. We show the first 2 million nanoseconds of the simulation for the sake of space. We can verify that the note, duration, and metadata that is shown in the waveform matches the entries of the song rom. Notice that the long periods of time where the note and duration are constant correspond to the "wait" entries in the song rom to wait for the note players to finish playing their current notes. As expected, the note changes every time new_note goes high for one clock cycle. Also, we see that the currently_used output of each of the note players correctly reflects when each note player is playing a note of the chord. The output samples are generated to create the audio signal when generate_next_sample is high.

Then, the zoomed in version of the output waveform is shown below for more detail.



We see that the shorter periods of time where note and duration are constant correspond to the individual notes that are being read in from the **song_rom** and fed into the three note players. From the **note_player_to_use** waveform, we see that **note_player_1** takes the first note, **note_player_2** takes the second note, and **note_player_3** takes the third note, as expected. The currently_used waveforms for each note player also reflect this expected behavior.

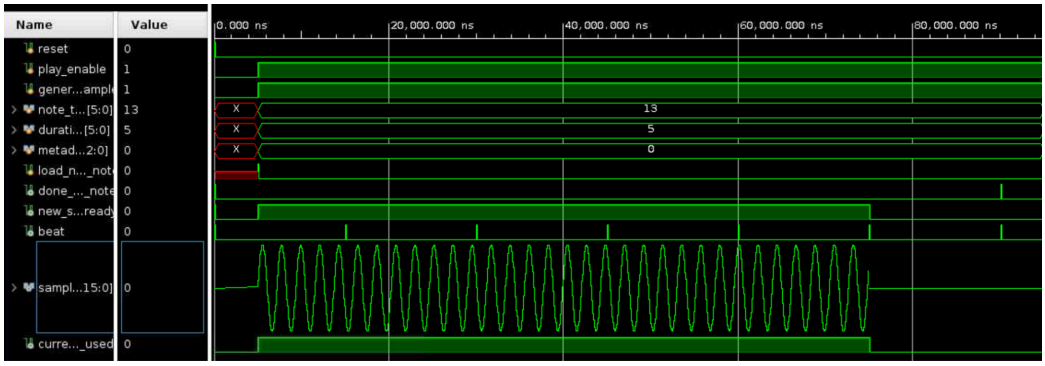
Note Player

This is the module that is responsible for playing a single note with a given duration and pitch. This is done by looking up the step size required to generate the correct frequency for that note, and then synthesizing a sine wave at that frequency for as long as the note is playing. The

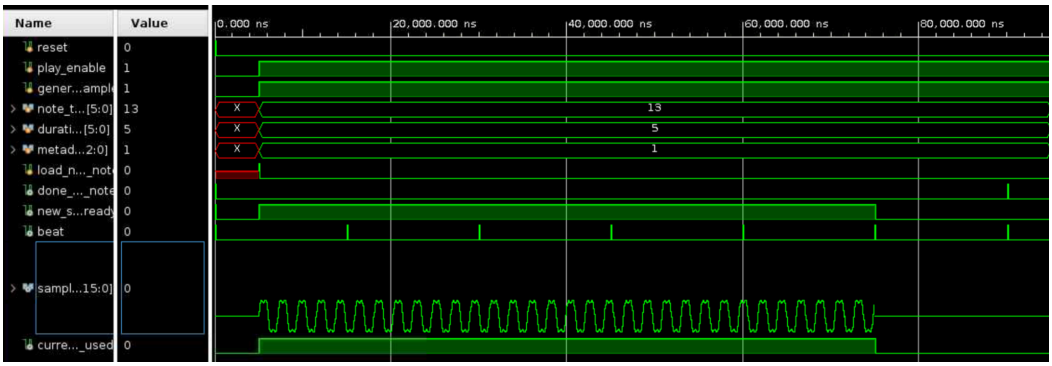
note_player also needs a counter to keep track of the duration of the current note. This counter ticks every 48th of a second while the note is playing. If we pause the song, the counter pauses as well. The 48th beats are generated by a **beat_generator** module.

Note Player Testbench

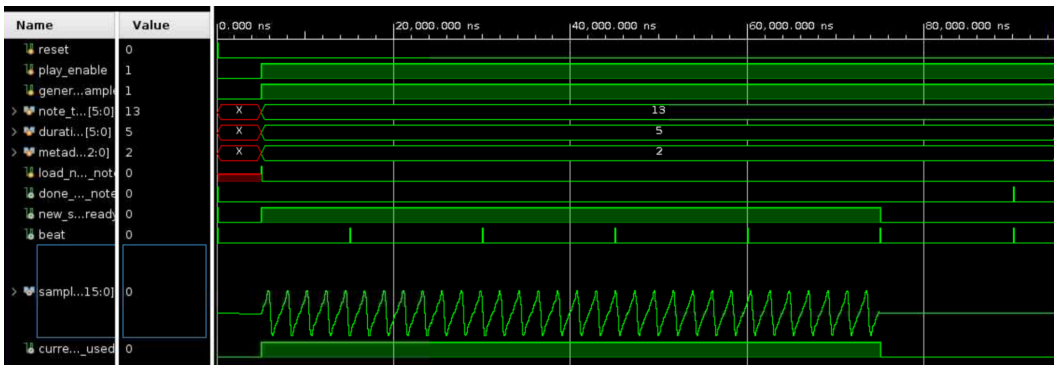
Our overall testing strategy is to let the note player module run for multiple different values of metadata (encoding different frequency spectra) and verifying that the output sample has the correct audio signal shape. Also, we see that in all waveforms, the signal lasts for 5 beats and becomes 0 afterwards, as expected.



Metadata = 0 represents a pure sine wave, which we see.



Metadata = 1 represents a square wave, which we see is approximated.

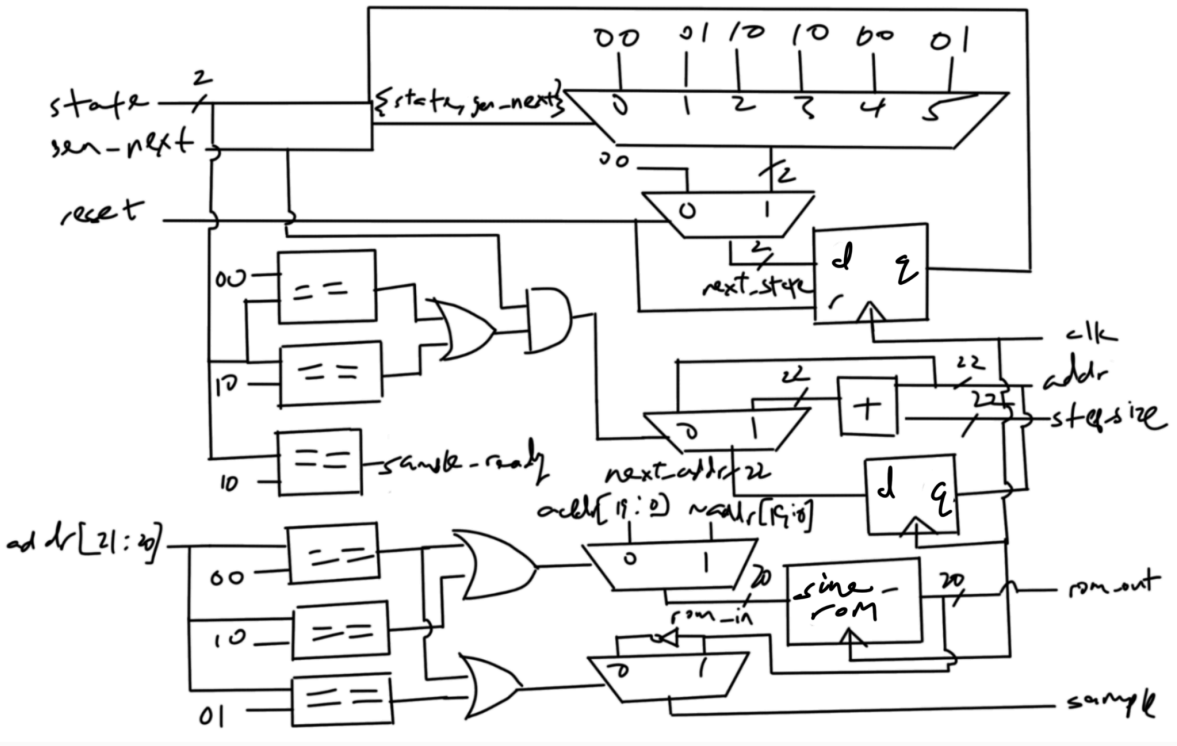


Metadata = 2 represents a sawtooth wave, which we see is approximated.

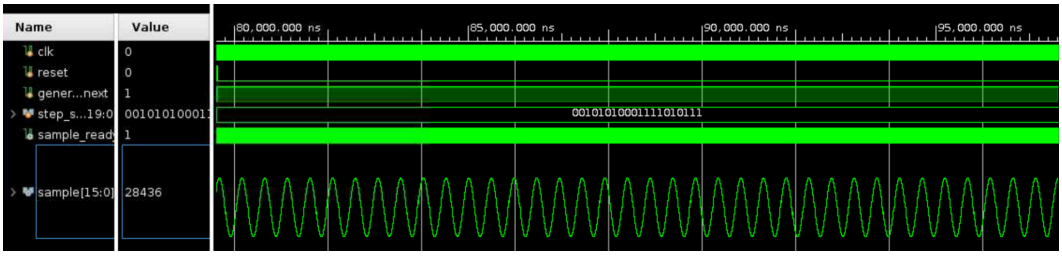
Sine Reader

The **sine_reader** is a sub-module of the **note_player** which takes in a step size (determined by the frequency of the note) and repeatedly reads from the **sine_rom**, generating a sine wave with frequency determined by the step size. The **sine_rom** takes in an address from **sine_reader** and produces a sample from the sine wave. If we address the sine ROM correctly, we can generate sine waves of arbitrary frequency. For every next sample, we increment the address pointer into our **sine_rom** by a certain step size that represents frequency. If we increment the address pointer by a larger step size, we traverse the sine wave faster, generating a higher frequency, and vice versa. The module output is generated from the **sine_rom**.

Sine Reader Block Diagram



Sine Reader Testbench

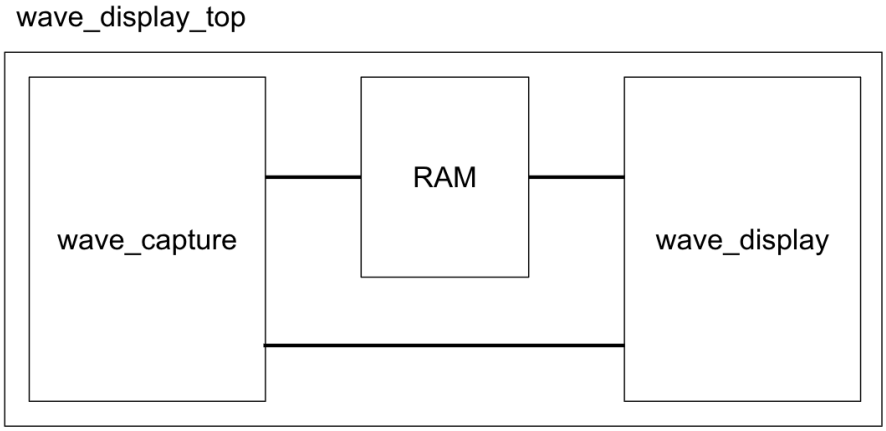


Here, we test the output of **sine_reader** and verify that it is a sine wave. A step size of {10'd168, 10'd983} was used to control the frequency of the output sample.

Wave Display Top

The top-level **wave_display_top** module is responsible for displaying the waveform to the external screen. The **wave_display_top** module is made up of two sub-modules: 1) The **wave_capture** module, which collects the individual samples being outputted by the **music_player** and reassembles them in a RAM, and 2) The **wave_display** module, which is given screen coordinates by the VGA driver and determines what color the pixel at that coordinate should be by reading from the **RAM**.

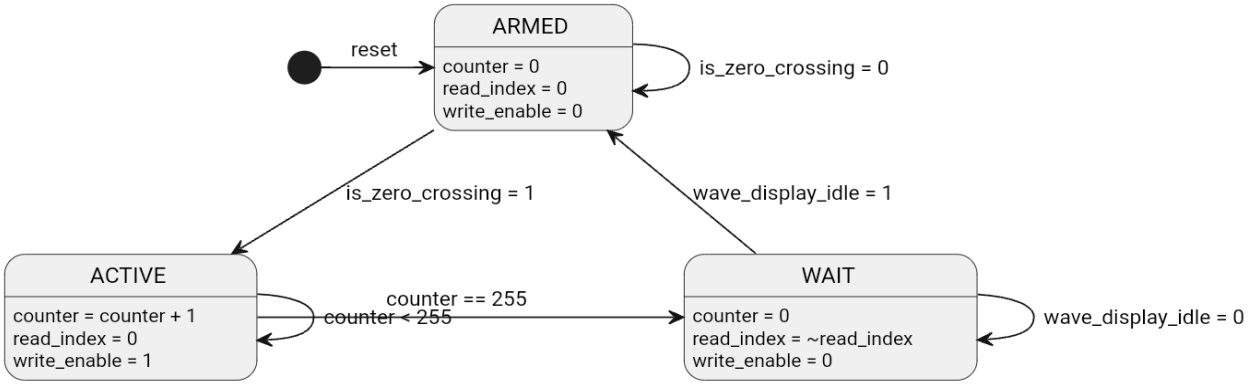
Wave Display Top-Level Block Diagram



Wave Capture

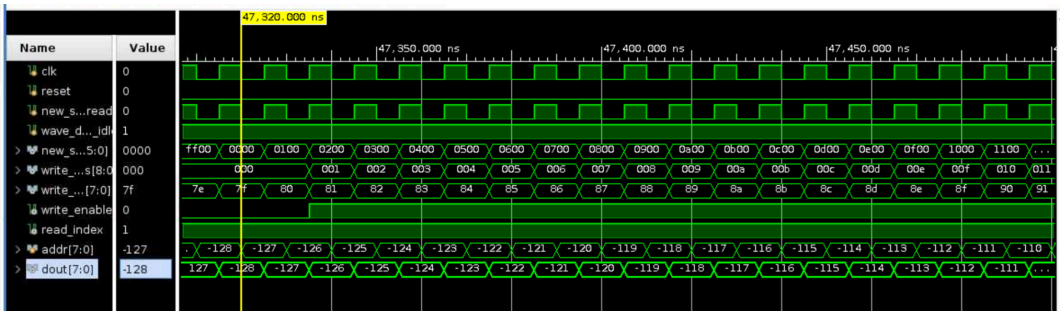
The **wave_capture** module implements a FSM that handles the audio samples from the **music_player** module. The FSM is initially in the armed state. After a zero crossing, the FSM switches to the active state, where it stores the 8 most significant bits of the next 256 audio samples into the RAM. Once it writes the final sample, it moves into the wait state, which ensures that we do not flip the read_index bit for double buffering.

Wave Capture FSM



For all states: write_sample = sample[15:8] ^ 8'd128, write_addr = {read_index, counter}

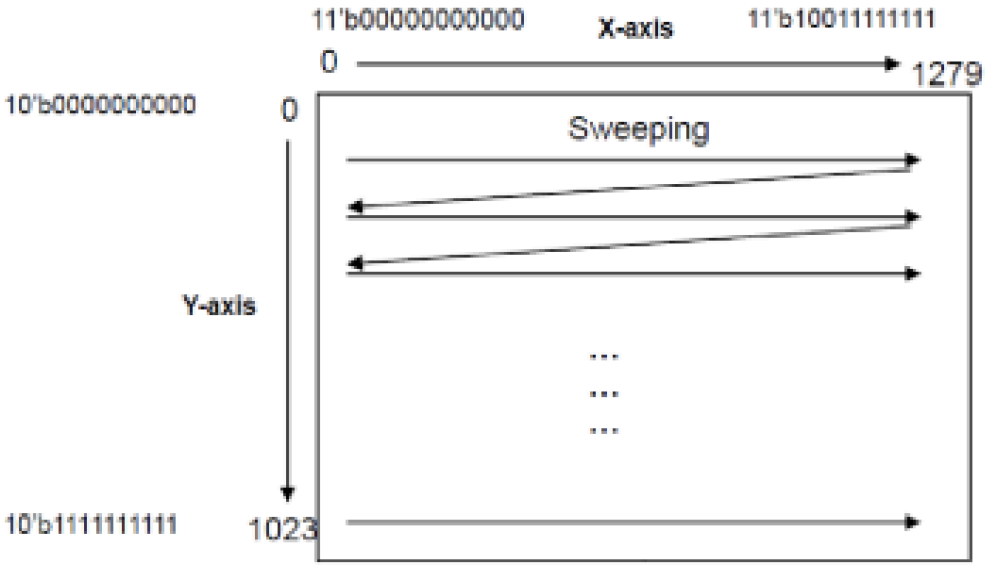
Wave Capture Testbench



Here, we can see that one clock cycle after dout shifts signs, write enable turns on. Here, dout shifts from positive to negative (but the module reverses the first bit of dout, so it actually goes from negative to positive. We can see that write enable turns on as soon as this sign change happens. Moreover, we can see that the write_sample is also correct (it starts counting from zero as soon as this sign change happens).

Wave Display

The **wave_display** module is responsible for displaying the waveform onto the raster display. On every clock cycle, the **wave_display** module is given the X (11 bits) and Y (10 bits) coordinates of a screen pixel from the VGA driver (see figure below), and on the next cycle, it outputs what color that pixel should be. At any given coordinate (X, Y), the pixel should be colored if the Y value is between RAM[X-1] and RAM[X].



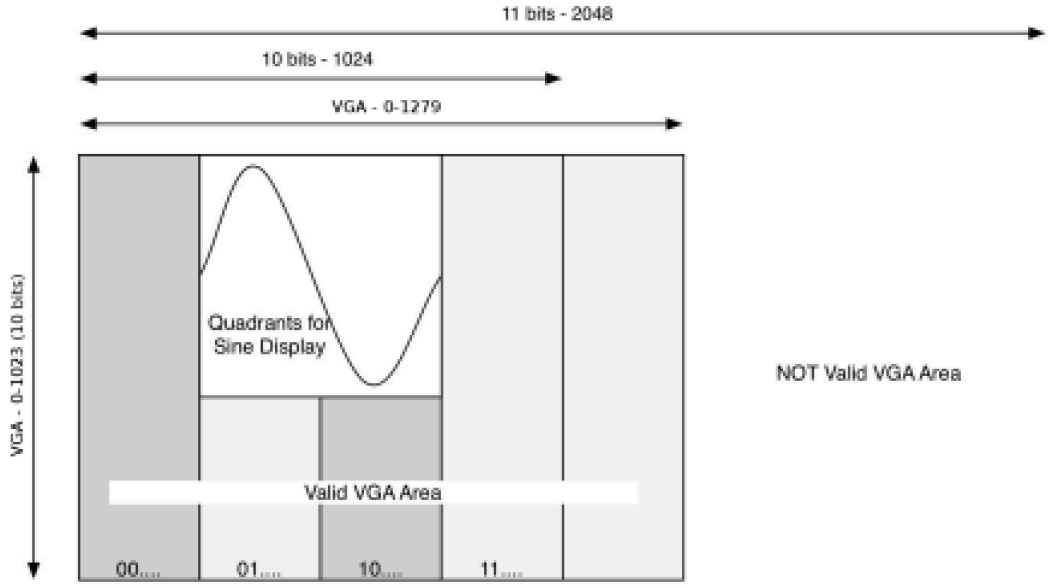
We must convert the given X and Y coordinates to the proper 9 bit RAM addresses to read from to correctly render the waveform on the display. For the waveform to be more visible, we drop

the least significant bits of X and Y, shown in red. Then, in order to draw the waveform in the correct quadrants of the display (see figure below), we follow the mapping:

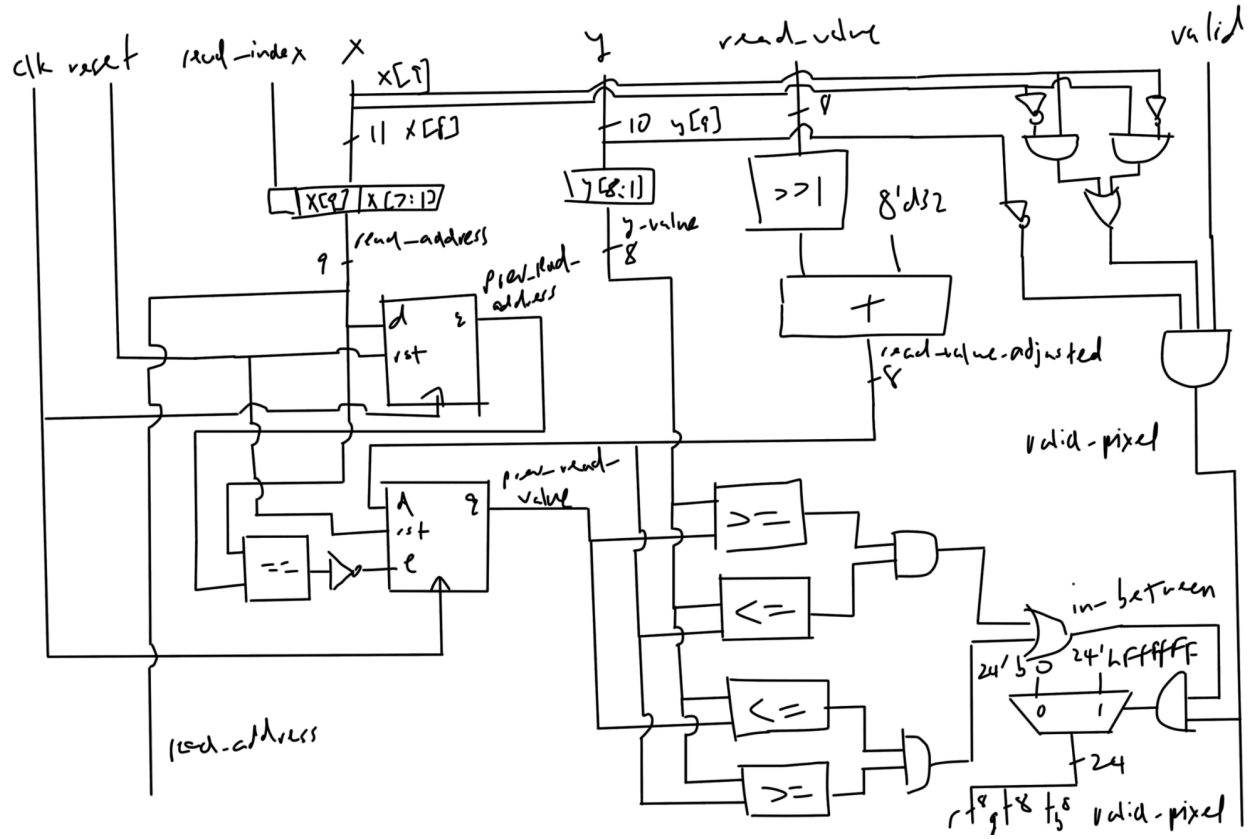
X[10:0]	read_addr
11'b000xxxxxxx	DNC
11'b001xxxxxxx	{read_index, 8'b0xxxxxxx}
11'b010xxxxxxx	{read_index, 8'b1xxxxxxx}
11'b011xxxxxxx	DNC

Y[10:0]	read_addr
10'b0xxxxxxxx	{read_index, 8'bxxxxxxxx}
10'b1xxxxxxxx	DNC

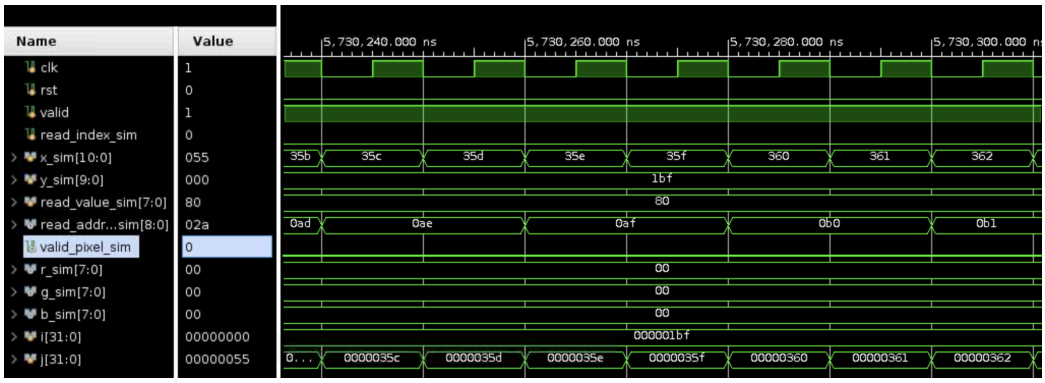
Note that we do not care when the X coordinate is in the 1st or 4th quadrants, and we do not care when the Y coordinate is on the bottom half of the screen, denoted by "DNC" in the mapping.



Wave Display Block Diagram



Wave Display Testbench



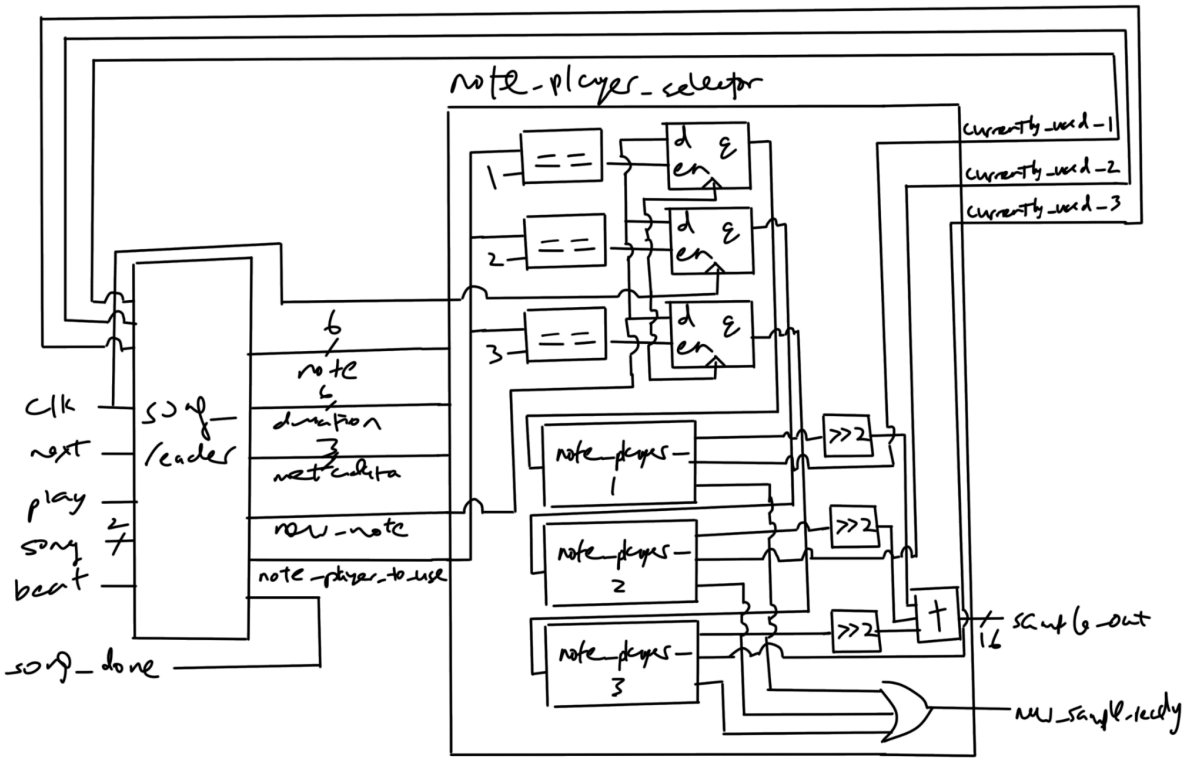
In the test bench, we see x_sim and y_sim continually changing because of the nested for loops. A new combination of x_sim and y_sim happens once per clock cycle as shown. Also, the RAM address changes every two x increments, which we can see on the waveform diagram. In this specific example, the rgb values have not changed because the simulation window is not long enough, but when the simulation was extended those values change over time. Additionally, valid_pixel_sim only shows 1 when valid is high and when the coordinate is in the desired quadrants (X and Y are both in valid ranges).

Chords

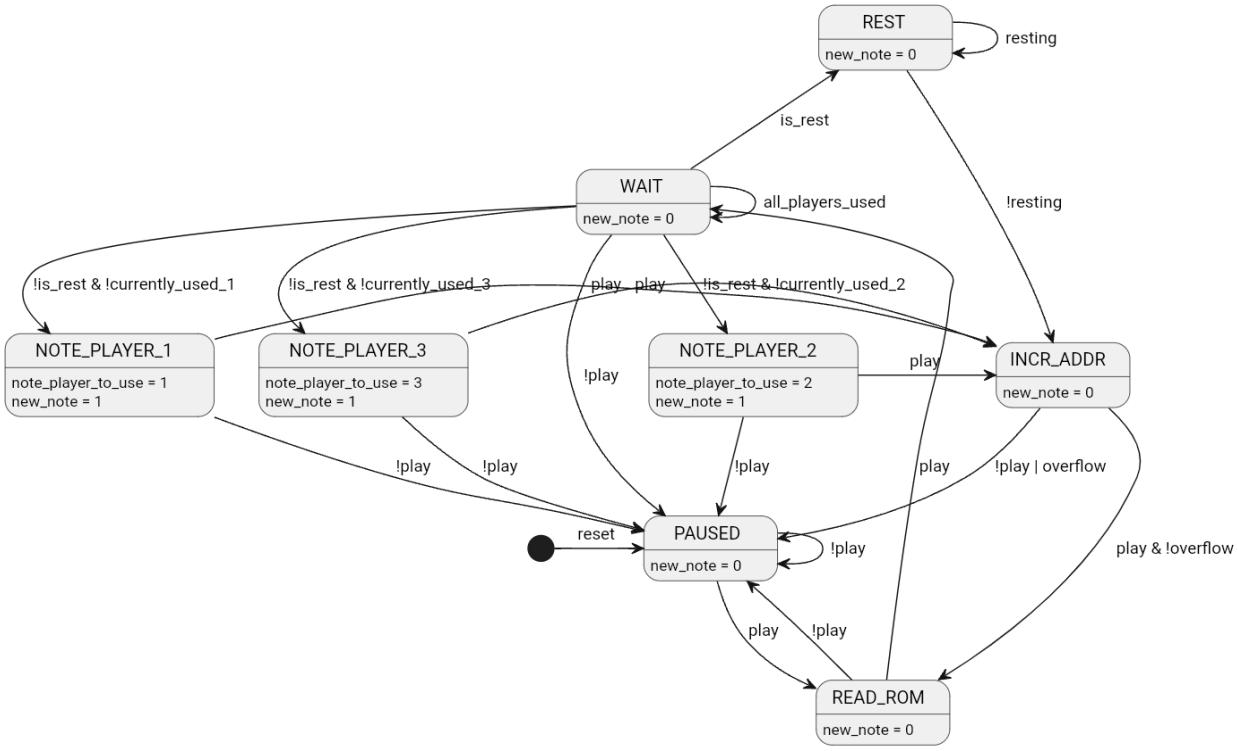
Chords involve playing multiple notes at the same time. The design is capable of playing up to 3 notes simultaneously by instantiating 3 **note_player** modules in **note_player_selector**. We use the following format for entries in the **song_rom** to support chords:

ROM format	Meaning
0 xxxxxx yyyyyy zzz	Schedule note xxxxxx (6 bits) to immediately start playing for a duration of yyyyyy (6 bits) 48th notes. The 3 zzz bits are metadata which encode timbre and the harmonics of the note. We use 48th notes as our time basis to allow both 16th notes and triplets to be encoded. After scheduling this note, move to the next item in the **song_rom**.
1 yyyyyy	Wait yyyyyy (6 bits) 48th notes before reading the next entry in the **song_rom**.

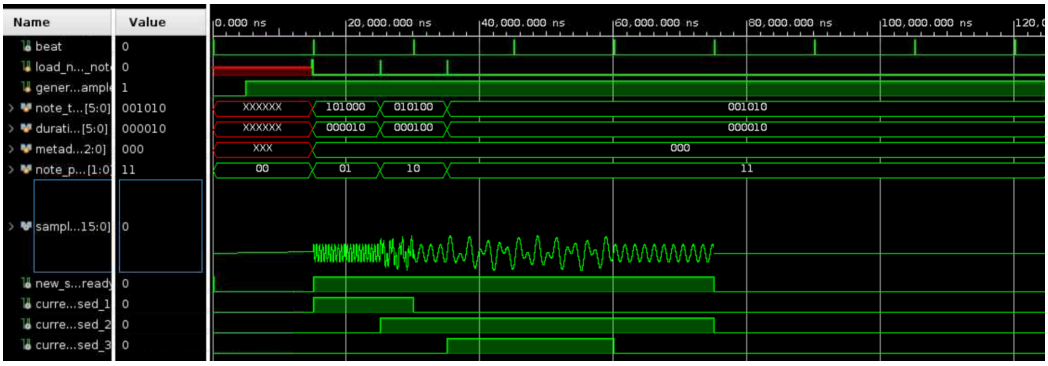
Chords Block Diagram



Note Player Selector FSM



Note Player Selector Testbench

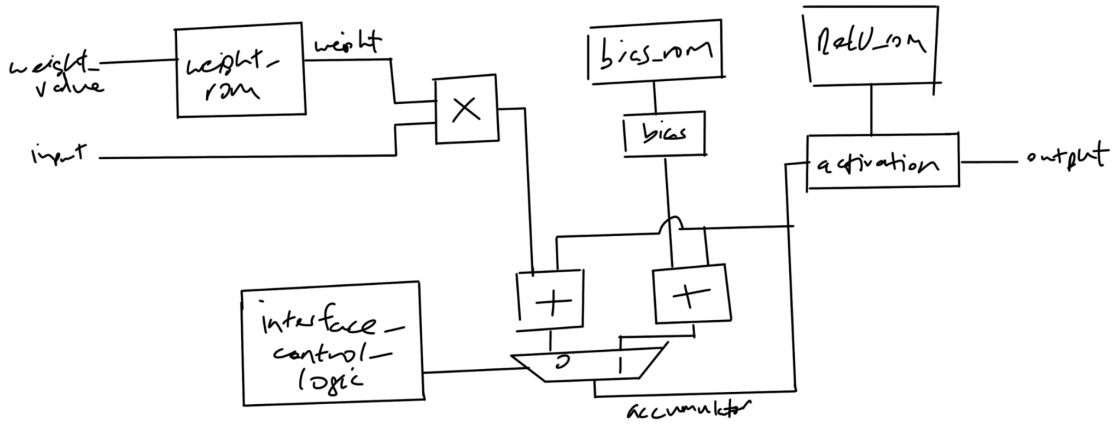


We test this module by manually feeding in 3 different notes with different durations, and also manually specifying which note player to use using the note_player_to_use input. Then, we see that the first note player plays for 2 beats, the second note player plays for 4 beats, and the third note player plays for 2 beats, as expected. We can verify this in the currently_used outputs of each note player. Then, the output sample is the superposition of all three output audio signals from the note players. Notice that in the regions where multiple note players are playing notes at the same time, the output signal appears more complicated, and in the regions where exactly one note player is playing, the output signal is a pure sine wave at the correct frequency.

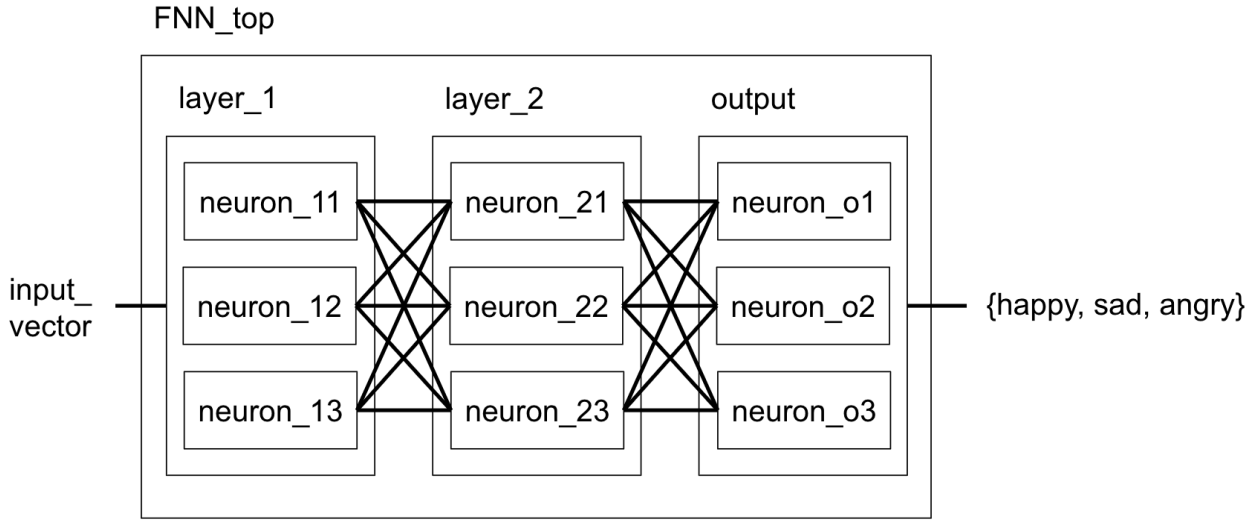
Music Emotion Classifier

The **music_emotion_classifier** module implements a feedforward neural network (FNN) that uses the last 2 chords played to predict the sentiment of the song (happy, sad, angry) at that point in time. To do this, we will integrate a FNN from scratch, where the training will be done off-FPGA and the inference will be computed on-FPGA. We first receive an input vector of the notes played, which is loaded into the processing pipeline. The input is fed into the first layer of the network. Each layer contains multiple **neuron** modules, and each **neuron** multiplies the input by weights, adds a bias, and applies an activation function for sigmoid. Weight and bias values are preloaded into memory blocks to make sure each **neuron** has the proper parameters for computation. Finally, the output from the last layer is converted into a final result that can be used for classification. We represent happy with green, sad with blue, and angry with red. We visualize the output as a weighted sum of these colors on the output wave.

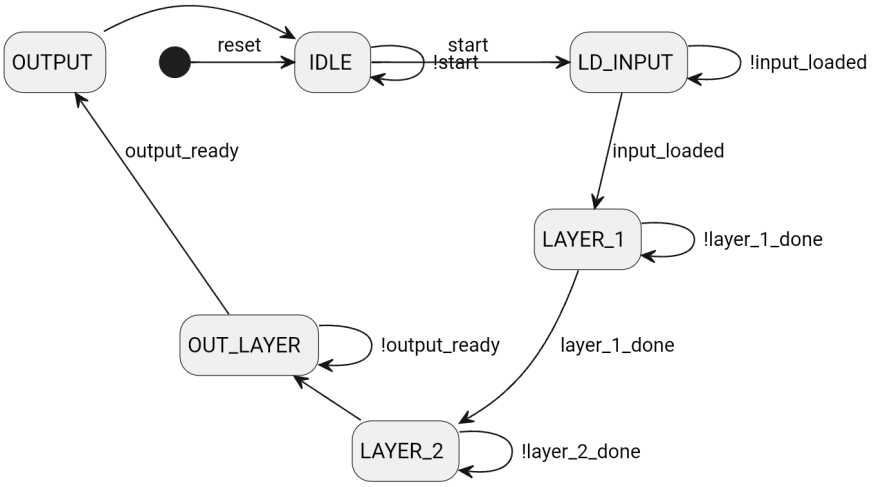
Neuron Block Diagram



FNN High-Level Block Diagram



FNN FSM



Neuron Testbench

```
# run 1000ns
t=0 state=00 i=0 act=0 w=1 bias=1 acc=0 done=0 out=0
t=10000 state=00 i=0 act=2 w=1 bias=1 acc=0 done=0 out=0
t=25000 state=01 i=0 act=2 w=1 bias=1 acc=0 done=0 out=0
t=35000 state=01 i=1 act=3 w=2 bias=1 acc=2 done=0 out=2
t=45000 state=01 i=2 act=3 w=-1 bias=1 acc=8 done=0 out=8
t=55000 state=01 i=3 act=0 w=3 bias=1 acc=5 done=0 out=5
t=65000 state=10 i=3 act=0 w=3 bias=1 acc=6 done=0 out=6
t=75000 state=00 i=3 act=0 w=3 bias=1 acc=6 done=1 out=6
Final Neuron output:    6
```

In this testbench, the **neuron** multiplies activation index 0 by weight 0 to get 2, activation index 1 by weight 1 to get 6, and activation index 2 by weight 2 to get -3, and sums these products to get 5, as expected. Then, we add the bias of 1 to get a total weighted sum of 6, so the ReLU activation function outputs 6 because it is non-negative, as expected. If the value fed into the ReLU was negative, the **neuron** outputs 0.

Music Emotion Classifier Testbench

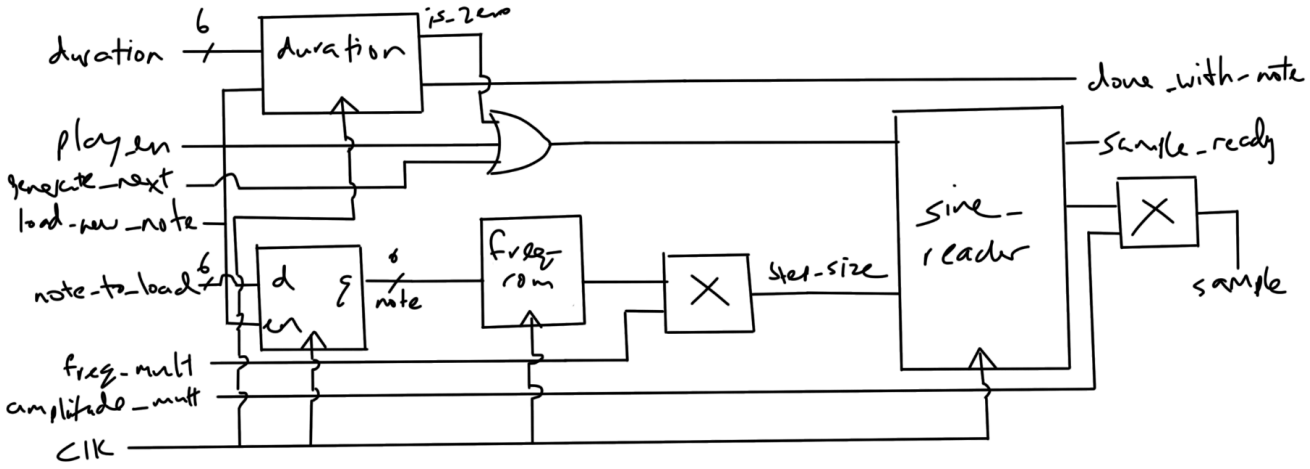
```
Classification complete for C4 to E4:
Output: 3600ef
Output values: [ 54,   0, 239]
Hidden Layer 1 outputs: [  99,   16,    0]
Hidden Layer 2 outputs: [ 118, -107,  -12]
Output Layer outputs: [ 54,   0, 239]

Test Case 2: G4 to B4
Classification complete for G4 to B4:
Output: 8b00b4
Output values: [139,   0, 180]
Hidden Layer 1 outputs: [   0,   11,    0]
Hidden Layer 2 outputs: [  44,    4,  -42]
Output Layer outputs: [139,   0, 180]
```

In this testbench, the full **music_emotion_classifier** is tested after being integrated into the music synthesizer system. It is clear that the expected output matches the output layer outputs, as the hex values of the expected output convert exactly to the output layer outputs. All of the computations within the hidden layers are also being computed properly as shown.

Harmonics

Harmonics involves playing multiple overtones over the fundamental frequency for each note. In the design, we play the first 4 terms in the harmonic series (first 4 integer multiples of the fundamental frequency), and we tune the weights using Fourier analysis to approximate a square and sawtooth wave for different timbres in the sound. These weights are stored in **harmonic_rom**. The output wave is a weighted sum of each of the terms in the harmonic series.

Harmonics Block Diagram



Dynamics

Dynamics involves creating an ADSR envelope for each note that is outputted, and we implemented this feature by modifying the **sine_reader** module. In the design, we opted for a very fast rise time (essentially instant), a gradual exponential decay, and a short sustain period. The exponential decay was implemented using an **exponential_rom** module that contains 4096 sampled values from a decaying exponential function with tuned parameters. The exponential envelope has a delay of 1.5 s and decay rate of 1 s^{-1}. Note that for the sake of space and clarity, the testbench waveforms throughout this project document are zoomed, making the time scale too short to visibly see a large effect from the ADSR envelope.

Key Implementation Details

Some important implementation details include the use of flip flops to reduce the length of the critical path to resolve timing errors. Without these flip flops, the critical path (from the **sine_reader** output to the output of the **song_reader** module) would be too long for the 100

MHz clock. We also tried to keep the neural network separate from the music player, as we implemented both separately and integrated them together at the end. Minimizing the interaction between the neural network code and music player code made the codebase easier to maintain and debug, and ensured that introducing the neural network did not create any strange behaviors. We also decided to implement dynamics at the **sine_reader** level, which is the lowest level in the high-level design and will therefore apply to every note that we play. Finally, it is worth noting that we drew inspiration from how we implemented chords to implement harmonics.

Problems Encountered

Some problems encountered were mainly due to timing errors. As previously mentioned, we resolved the setup time error after synthesis by adding flip flops to break up the critical path. We also encountered errors in values changing at the wrong time (i.e. changing too early). We were able to diagnose these issues by carefully looking at the waveforms from testbench simulation. Finally, at a system design level, we ran into lots of integration errors, as the entire system would not work when run at the top level. The way we debugged these errors was to start at lower levels of the design (often **note_player** or **sine_reader**), confirm that they work via testbench outputs, and move up the hierarchy to higher level modules.