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 implemented purely in hardware.
- Integrate machine learning (ML) for emotion prediction into the hardware pipeline.
- Demonstrate digital signal processing (DSP) and sound synthesis techniques with hardware efficiency.
- Provide real-time audio-visual feedback through an external display.

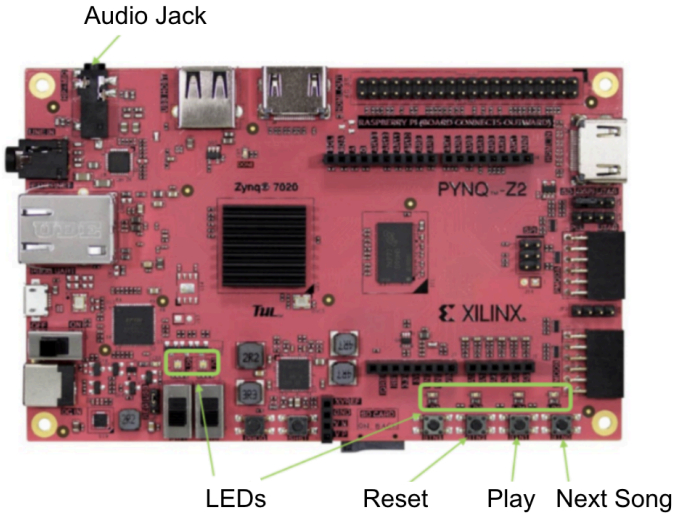
The main features are:

- **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.
- **Chords:** The ability to play multiple notes simultaneously, enabling the synthesizer to output harmonies.
- **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.
- **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.
- **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.
- **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.

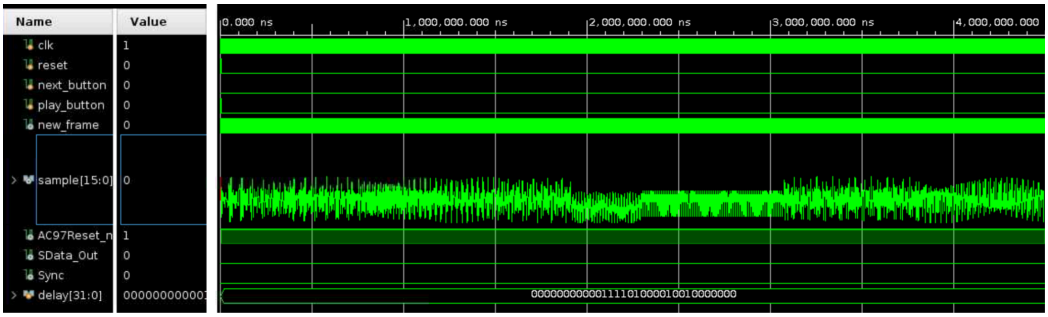


Overview of Core System Modules

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.

Music Player Testbench



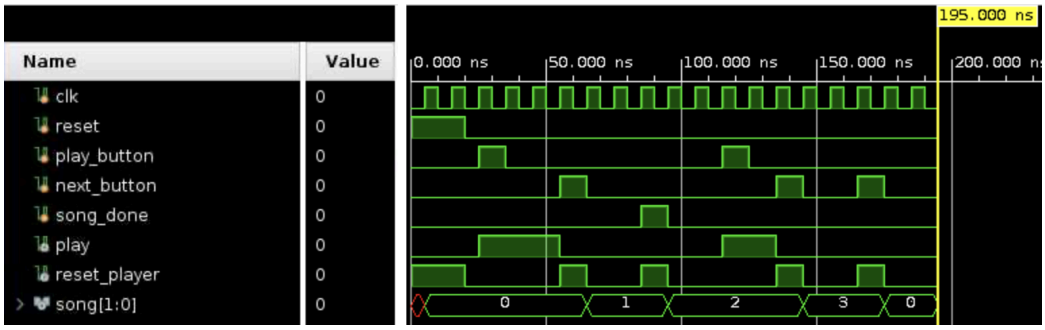
The test strategy was letting the music player module run and recording the output waveform. We show the first 4 million nanoseconds of the simulation for the sake of space. We see that after the play button is pressed, the output samples start being generated to create the audio signal, as expected. 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 Testbench

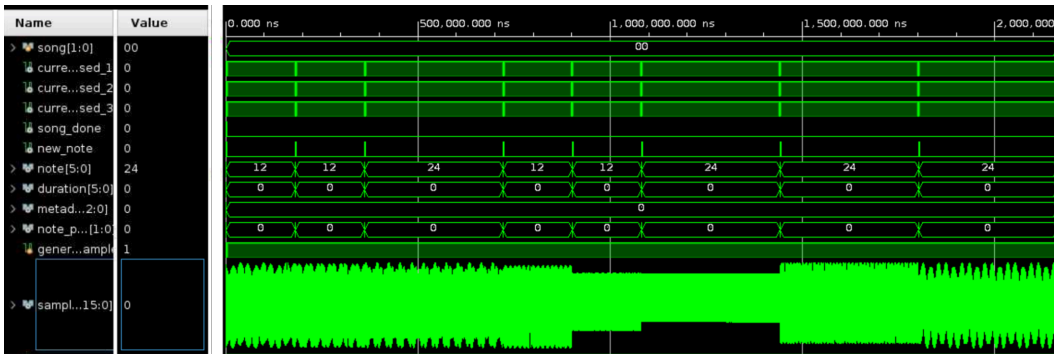


We see that when the play button is pressed, the play output goes high until the song_done signal goes high, at which point play goes low and reset_player goes high, as expected because the MCU should reset the player and start out paused at the next song. We also see that the song is correctly incremented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

Song Reader

The **song_reader** is responsible for determining which note to play. The **song_reader** takes in the current song to read from the MCU and reads from the song ROM. It then sends the new note to be played, and when the current note is finished, the **song_reader** looks up the next note and sends it to be played. This repeats until the **song_reader** has finished the current song, at which point it tells the MCU that the song is done.

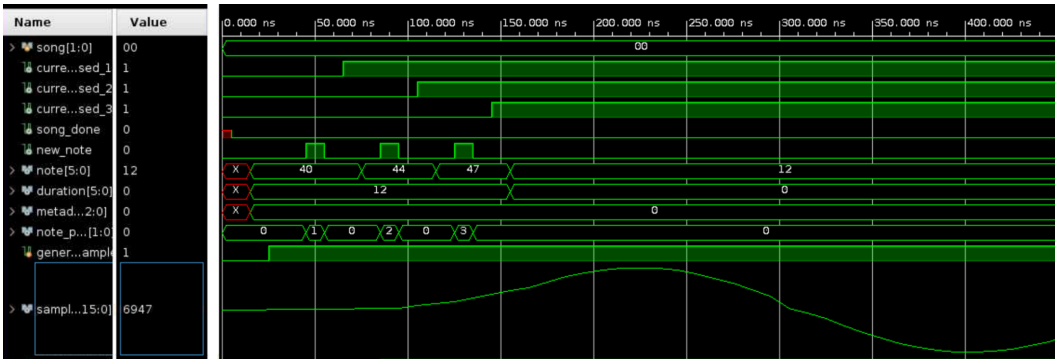
Song Reader Testbench



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 and duration 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.



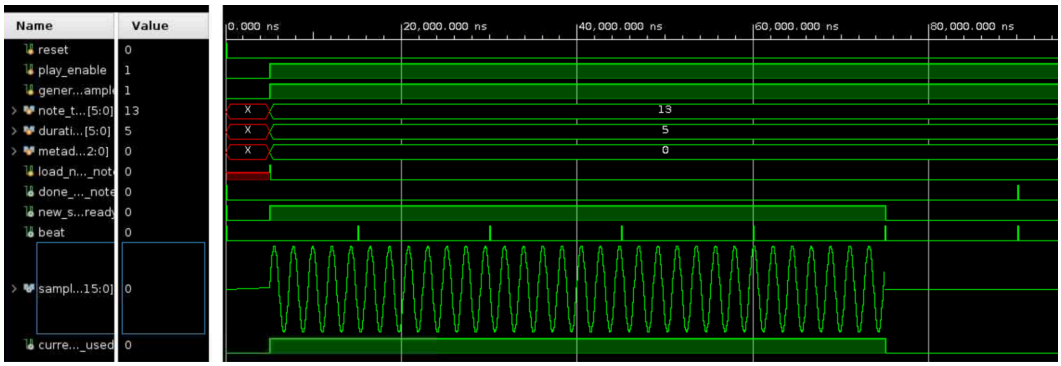
We see that the shorter periods of time where note and duration are constant correspond to the individual notes that are being read in from the song ROM and fed into the three note players. From the **note_player_to_use** waveform, we see that **note_player_1** takes the first note, **note_player_2** 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

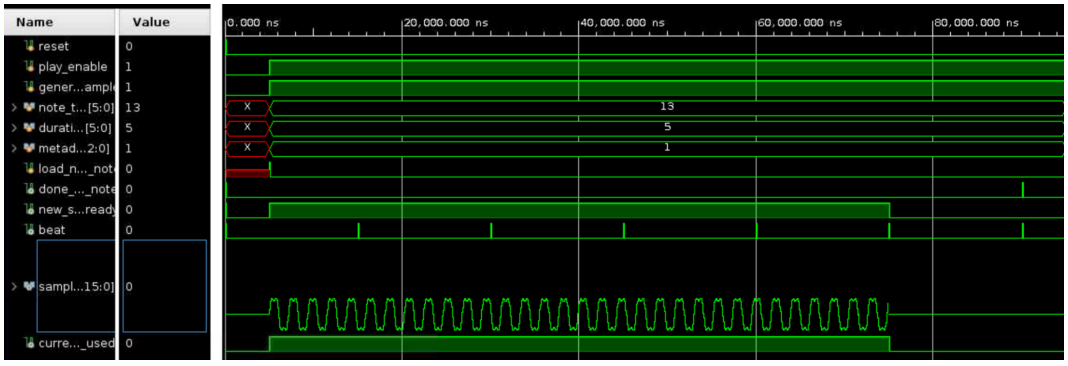
This is the module that is responsible for playing a single note with a given duration and pitch. This is done by looking up the step size required to generate the correct frequency for that note, and then synthesizing a sine wave at that frequency for as long as the note is playing. The **note_player** also needs a counter to keep track of the duration of the current note, which ticks every 48th of a second. If we pause the song, the counter pauses as well.

Note Player Testbench

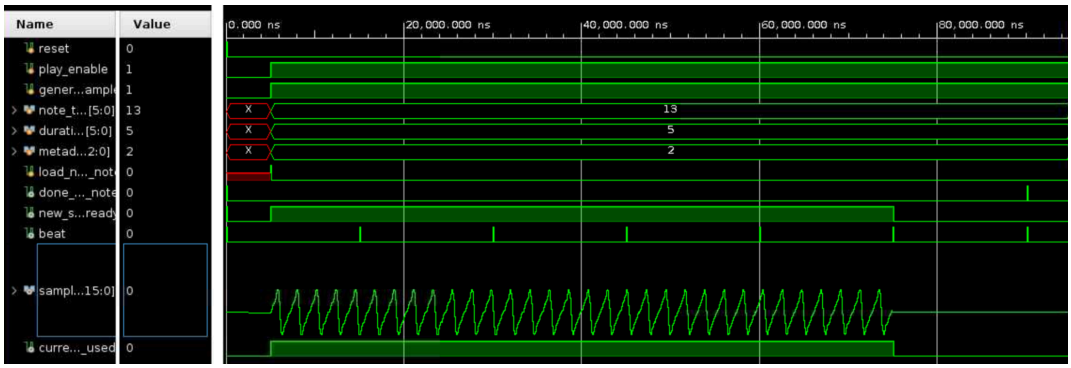
Our overall testing strategy is to let the note player module run to synthesize different types of waves and verifying that the output sample has the correct audio signal shape. Also, we see that in all waveforms, the signal lasts for 5 beats and becomes 0 afterwards, as expected.



A pure sine wave



Approximating a square wave

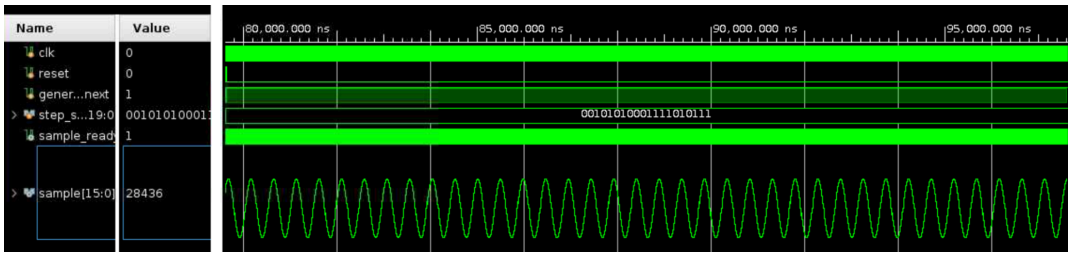


Approximating a sawtooth wave

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. If we increment the address pointer by a larger step size, we traverse the sine ROM faster, generating a higher frequency, and vice versa.

Sine Reader Testbench



Here, we test the output of **sine_reader** and verify that it is a sine wave.

Wave Display Top

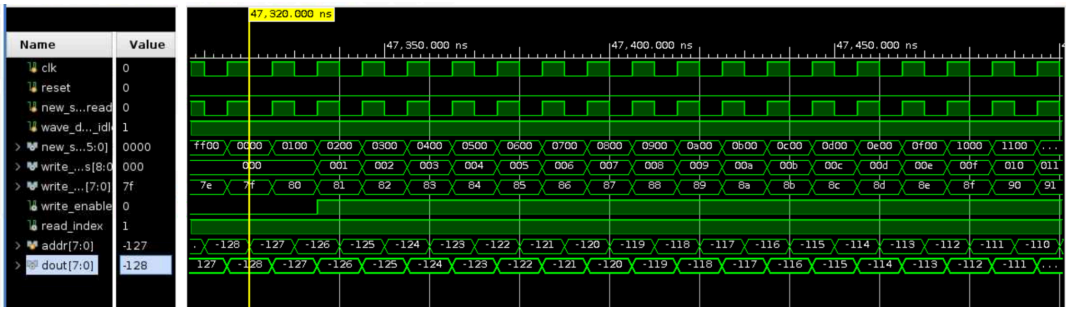
The top-level **wave_display_top** module is responsible for displaying the waveform to the external screen. The **wave_display_top** module is made up of two sub-modules: 1) The

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 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 audio samples it receives into 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 Testbench

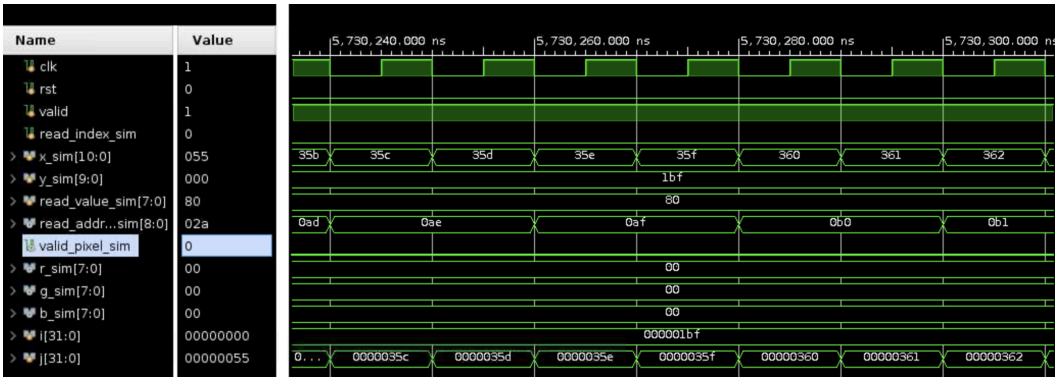


Here, we can see that one clock cycle after dout shifts signs, write enable turns on. We can see that write enable turns on as soon as this sign change happens. Moreover, we can see that the write_sample starts counting from zero as soon as this sign change happens.

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 and Y coordinates of a screen pixel, and outputs what color that pixel should be.

Wave Display Testbench

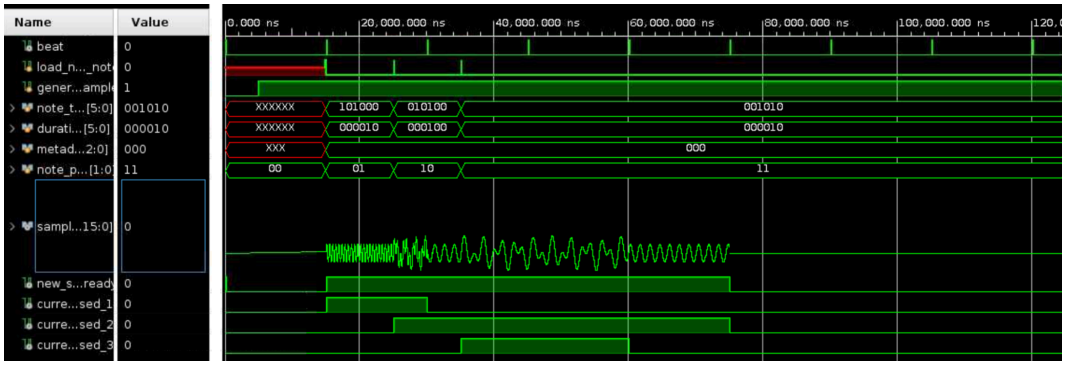


In the test bench, we see x_sim and y_sim continually changing because of the nested for loops. A new combination of x_sim and y_sim happens once per clock cycle as shown. Also, the RAM address changes every two x increments, which we can see on the waveform diagram. In this specific example, the rgb 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**.

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 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 a ROM. The output wave is a weighted sum of each of the terms in the harmonic series.

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 a ROM that contains sampled values from a decaying exponential function with tuned parameters. 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.