6.111 Final Project: Freeform Production of Gorgeous Audio

Rahul Yesantharao and Jacob Pritzker

Table of Contents

I. Overview / Motivation:

For our final project, we planned to make a musical system with multiple modes.

In our baseline goal, we aimed to have a keyboard mode, in which a user could play many octaves' worth of notes using switches on the Nexys4, with audio coming from the Nexys4's audio output port. We also aimed to have a simple game mode, similar to Guitar Hero, in which a user could select a song, and then LEDs on the Nexys4 would light up, corresponding to notes in a song that a user would try to match using the switches.

In our expected goal, we wanted to replace the switches on the Nexys4 and audio generation from the Nexys4 with a MIDI keyboard. This required proper interfacing with the MIDI keyboard. In addition, we wanted to provide a display system on a lab monitor that would allow a user to navigate a menu to select between different modes and different songs. Furthermore, we wanted to include a falling notes display, like in Guitar Hero. Lastly, we wanted to allow for menu navigation either with buttons on the Nexys4 or with whistling a high vs. low note into a microphone.

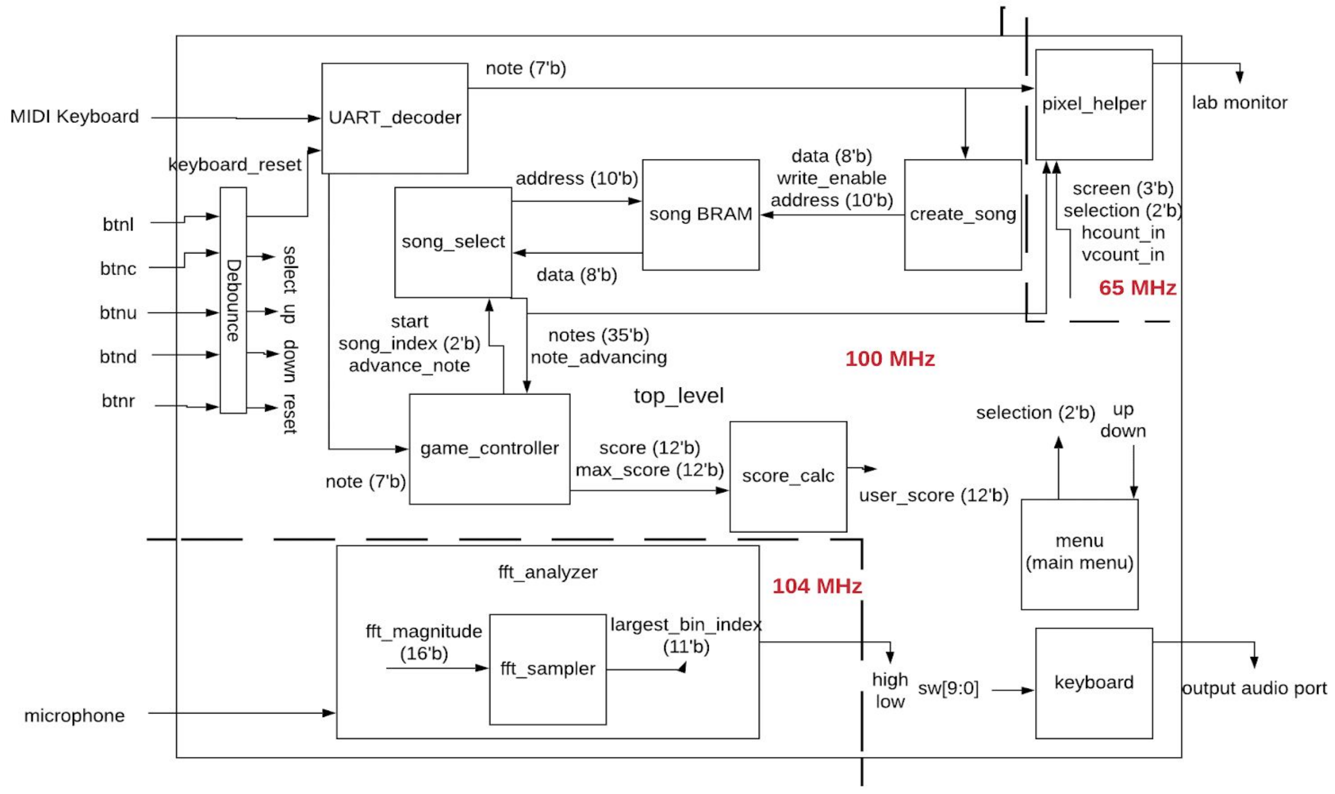
In our stretch goal, we wanted to allow for a singing input mode, where a user would try to sing a song accurately, rather than play it on a keyboard. In addition, we wanted to implement a learning mode, in which a user would be walked through a song in order to learn it before being scored for it. Lastly, we wanted to allow a user to define a song of their choosing for the game mode, rather than being restricted to our preset options.

We ultimately accomplished all of these goals except the singing mode stretch goal.

To accomplish this, we wrote many major modules for our system, which we describe in depth below. At a high level, the major modules performed MIDI decoding, output display logic, game logic, interfacing with memory, and FFT calculations for microphone menu navigation. Due to FFT requirements and VGA display requirements, our system used three different clock domains.

The overall system diagram, and then discussion of each piece, can be found below, in Sections II and III.

II. Total System Diagram



III. Major Modules

A. Keyboard (Jacob)

For our baseline goal, we used the bottom 7 switches on the Nexys4 to choose a note, and then output a PWM signal at the corresponding frequency from the audio output of the Nexys4.
The keyboard module works similarly to the starting code we used in Lab 5A. However, I wanted to extend this to work for an entire piano's worth of notes. In order to do so, I first made a look up table to convert from 7-bit note indices to frequencies (in Hz). I then used the formula shown in Lab 5A to convert from a frequency to a phase increment:

$$\frac{\left(440\frac{\text{cycles}}{s}\right)\cdot\left(2\times10^{32}-1\frac{\text{phase}}{\text{cycle}}\right)}{48\times10^{3}\,\frac{\text{samples}}{s}}\approx 39370534\frac{\text{phase}}{\text{sample}}$$

(This example showed how to calculate this for a 440 Hz desired frequency. It is from http://web.mit.edu/6.111/volume2/www/f2019/index.html.)
By making the phase increment variable, according to this formula, it allowed me to at any time switch which note was being played by stepping through the sine wave look up table (from Lab 5A) at different speeds. I used switches 7-9 for volume control.
I used the same code for generating the PWM wave as was used in Lab 5A.
In our final version of the project, we included an option to still use the switches on the Nexys4 instead of the MIDI keyboard. If the top switch (sw[15]) is high, then the input mechanism is the switches. If it is low, then the input mechanism is the MIDI keyboard.

B. MIDI Keyboard Interface (Jacob)
We wanted to allow for user input via a MIDI keyboard. The keyboard we used has a MIDI output port, but the Nexys4 does not have a MIDI input port. Therefore, in order to connect them properly, I needed to build the following circuit to replicate the front-end of a MIDI input port:

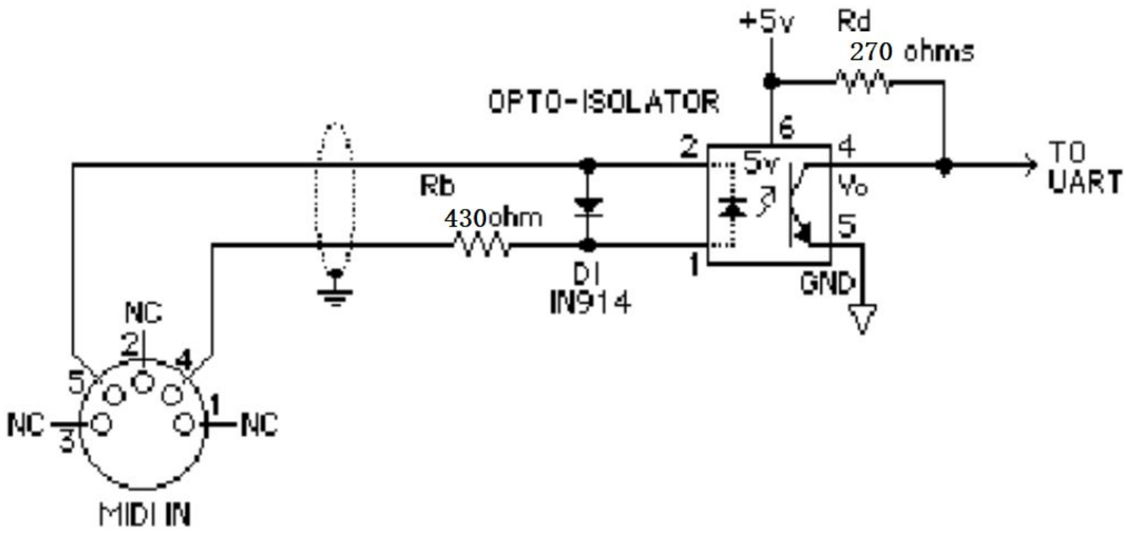
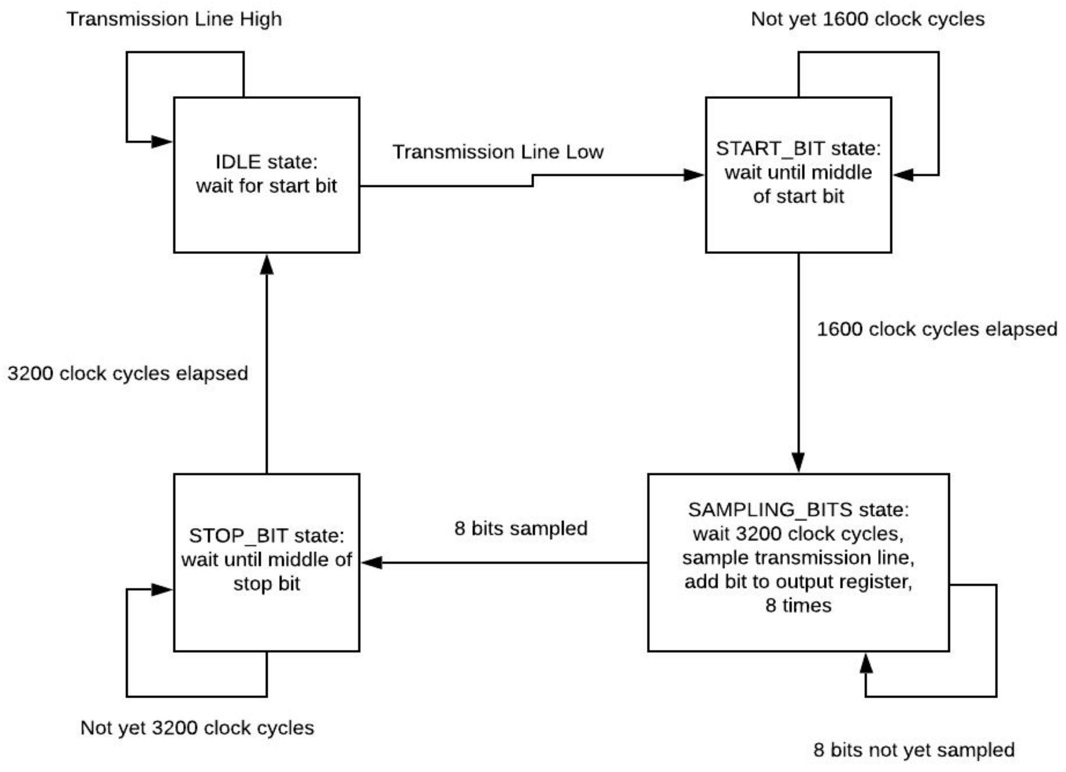


diagram from https://learn.sparkfun.com/tutorials/midi-tutorial/all
This circuit uses an opto-isolater, meaning the keyboard and the Nexys4 are not electrically connected. This made voltage regulation simple, even though the keyboard outputs 5V and the Nexys4 works at 3.3V. I powered the opto-isolater at 3.3V, meaning the output would be in the range 0V - 3.3V, but because the input side of the opto-isolater is not electrically connected to the power rails, it was okay to put the 5V output from the keyboard into it. The opto-isolater I used was the CPC5002.
Once this circuit was in place, the output from it was essentially a UART transmit line. I could then connect this to an input pin on the Nexys4, and then sample it and interpret it properly.

Before I did this, however, I needed to establish what messages the keyboard would output. While it was similar to standard MIDI messages, the keyboard we used slightly modified some of the MIDI protocols. The only information we cared about was which note was being pressed at any given time.
By scoping the output from the keyboard, I learned that whenever a note is pressed, the keyboard sends two bytes: first, a byte indicating a note index, second, the byte 8’h40. Similarly, whenever a note is released, the keyboard sends two bytes: first, a byte indicating a note index, second, the byte 8’h00. By contrast, the standard MIDI protocol sends the status (on/off) byte first, then the note byte, as well as other information. In addition, the actual on/off bytes are different in the standard MIDI protocol.

Now that I understood the messages that would be sent, I wrote three modules to convert the UART transmit line signal into the information we needed.

First, I wrote the UART_decoder module, which samples the transmit line and determines the bytes sent by the MIDI keyboard. This module is a 4-state finite state machine with the following state diagram:


Transmission Line High
Not yet 1600 clock cycles
IDLE state: wait for start bit
Transmission Line Low
START_BIT state: wait until middle of start bit
1600 clock cycles elapsed
3200 clock cycles elapsed
8 bits sampled
STOP_BIT state: wait until middle of stop bit
SAMPLING_BITS state: wait 3200 clock cycles, sample transmission line, add bit to output register, 8 times
Not yet 3200 clock cycles
8 bits not yet sampled

The MIDI protocol sends 31,250 bits per second. It sends a byte as a 10-bit packet, where the first bit is a low start bit, then it sends the 8 bits for the byte (least significant bit first), then it sends a high stop bit.

Running the UART_decoder module with a 100 MHz clock, there are 3,200 clock cycles per bit. Since the UART transmission line sits high when idle, my FSM has an IDLE state, in which it samples the input transmission line once every 200 clock cycles, trying to catch the falling edge, indicating a start bit. Once it sees the line go low, it switches to the START_BIT state, in which it waits 1,600 clock cycles, so that it is then near the middle of that start bit. It then switches to the SAMPLING_BITS state, in which it repeatedly waits 3,200 clock cycles and samples the transmission line 8 times, thus grabbing the 8 bits for the byte and filling up the output register. Then, after doing this 8 times, it switches to the STOP_BIT state, in which it waits another 3,200 clock cycles to get to the middle of the high stop bit, at which point it switches back to the IDLE state, waiting for the next low start bit.

The UART_decoder module instantiates the MIDI_decoder module, and passes in the output bytes that it got from the transmission line as input to the MIDI_decoder.

Essentially, the MIDI_decoder's job is to take the bytes from the UART_decoder module and output an 88-bit register, with a 1 at every index corresponding to a note currently being played, and a 0 everywhere else. In order to do so, it checks whether each incoming byte is a status byte (either the note on or note off byte). If it is not, then it stores the note index contained in that byte in a register called last_note. If it is a status byte, then it checks the last_note register to see what note that status of on or off corresponds to, and updates the 88-bit register accordingly.

The UART_decoder module also instantiates the key_press module, which takes the 88-bit output from the MIDI_decoder and outputs a single 7-bit index corresponding to the note currently being played. It works by first adding up all 88 bits in the input register. If this sum is 0, it means no note is being played, so it outputs 7'h7F, which is our index for a blank. If this sum is greater than 1, it outputs 7'h7E, which is our index for an invalid note (since we only allowed for one note being played at a time). If this sum is 1, then it has a case statement based on the value of the 88-bit register, outputting the appropriate index of the single note being played. This output is then passed up to the UART_decoder module, which outputs this 7-bit index.

C. VGA Display (Jacob)

For the VGA display, I wrote a module called pixel_helper that output the appropriate pixel for given hcount and vcount values, depending on what mode our system is in.

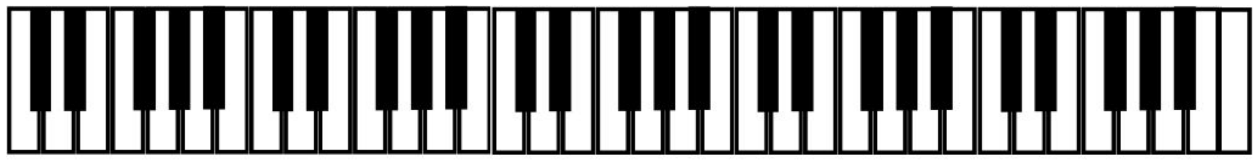
There are 7 different possible types of screens to display, so pixel_helper has a 3-bit input register called screen, indicating which type of screen to display.

There is:

1) Main Menu
2) Keyboard mode
3) Song Creation mode
4) Basic song menu
5) Custom song menu
6) Learning mode
7) Game mode

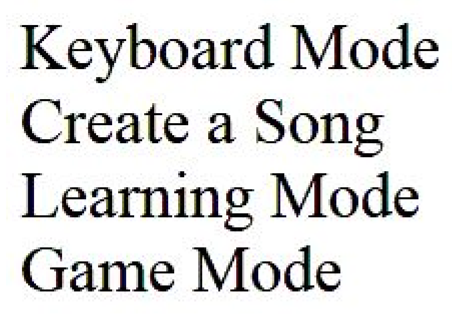
Because the display is always from among this set (with a few squares or rectangles on top), I made a COE file for each of the special images I required, and then generated ROMs to store these images and their colors in memory.

I needed a keyboard image for keyboard mode, learning mode, and game mode:



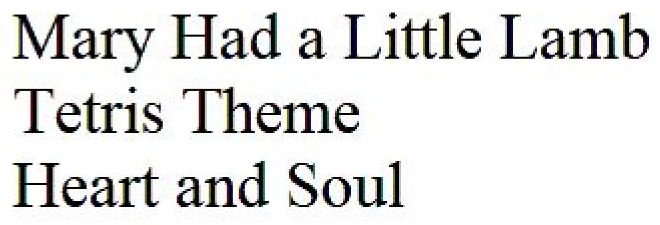
(I constructed this image in Paint so that it would have the precise correct number of octaves to match the MIDI keyboard we used.)

I needed an image for the main menu display:



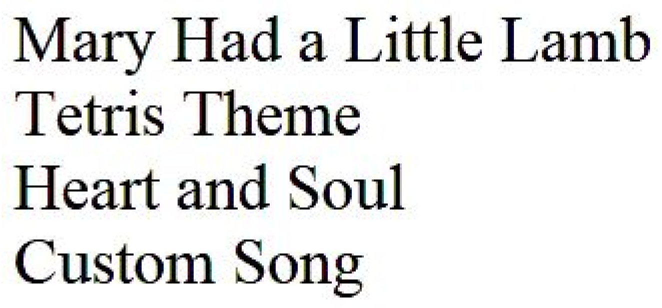
(This text and text in all other images uses Times New Roman font.)

I needed an image for the basic song menu:



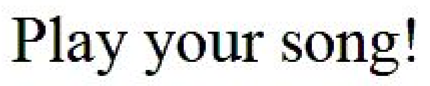
Mary Had a Little Lamb
Tetris Theme
Heart and Soul

I needed an image for the custom song menu:



Mary Had a Little Lamb
Tetris Theme
Heart and Soul
Custom Song

Lastly, I needed an image for instructions for song creation mode:



Play your song!

I made 5 specialized versions of the blob_image module from Lab 3, one for each of these images, and instantiated a copy of each in pixel_helper.

In addition to these images, I also needed a selector for menu navigation. For this, I instantiated the blob module from Lab 3, making a 35 by 35 red blob. Another input to pixel_helper is the 2-bit selection register, which tells it where a user currently is within a menu (all of our menus have at most 4 options, so 2 bits is enough to specify the location). I then made a look up table module to convert a position (0, 1, 2, or 3) within a menu into a y-position. By setting the x-location of this blob to 650 and the y-location determined by the output of the lookup table, users can navigate the menus and see the red selector move between options.

Furthermore, pixel_helper has a 6-bit register user_note as input, indicating what note is currently being played by the user. I wanted there to be a corresponding display for this during keyboard, learning, and game modes. I therefore made another instantiation of the blob module, this time a 10 by 60 green blob. The y-position is always 580 (to be just above the image of the keyboard, which I placed at y-location 640 to be at the bottom of the screen). For the x-position, I made a look up table that converted note indices into x-positions, based on the exact location where each note started and ended in the image display. For a blank note or an invalid note (multiple notes played at once), the x-location is set to 11'h7FF, thus not showing up on the screen.

During learning mode, I wanted to display the single note that the user is up to in the song they are learning. I therefore gave pixel_helper a 6-bit register learning_note as input, and made yet another instantiation of the blob module, this time a 10 by 160 blue blob. The y position is always 480 (to be just above the image of the keyboard), and the x-position is set by the output of the look up table that converts note indices to x-positions.
Lastly, during game mode, I wanted to make a falling notes display, showing 5 notes at once. I therefore gave pixel_helper a 35 bit register *notes* as input, which is a shift register. It contains the note index of the note that should currently be played in notes[34:28], the note after that in notes[27:21], then the next note in notes[20:14], then the next note in notes[13:7], and the next note after that in notes[6:0]. I therefore made 5 more instantiations of the blob module, all 10 by 160 blue blobs.

In order to synchronize the falling notes display with the underlying game, I knew that each note should last 0.25 seconds. Therefore, I needed each note blob to drop 160 pixels in 0.25 seconds. Working at a 65 MHz clock, I had each blob increase its y-location by 2 every 203,125 clock cycles.

I also gave pixel_helper an input called new_note, which goes high when a new note is shifted into the notes register, and is low otherwise. When this signal is high, the 5 note blobs' y-locations are reset to 864 (which is -160 since the height of the screen is 1024), 0, 160, 320, and 480. Then (on the next clock cycle, to make sure the notes register has already updated properly), the x-locations of these 5 blobs is updated to the 5 locations corresponding to the notes in the shift register. Since the new_note signal goes high every 0.25 seconds, each note blob falls at the right speed so that when a new note shifts in, the transfer looks seamless.

Lastly, in order to give the user a sense of the speed of the notes, I included horizontal, single-pixel thick black lines at y-locations 0, 160, 320, and 480. This was done by just checking if vcount was one of those values, and if so, outputting 12'h000.

In total, pixel_helper has a case statement depending on what screen display it should have.

1) In main menu mode, it outputs the logical AND of the pixels from the main menu image, and the selector blob.
2) In keyboard mode, it outputs the logical AND of the pixels from the keyboard image, and the user note blob.
3) In song creation mode, it outputs the pixels from the song creation image.
4) In the basic song menu, it outputs the logical AND of the pixels from the basic song menu image, and the selector blob.
5) In the custom song menu, it outputs the logical AND of the pixels from the custom song menu image, and the selector blob.
6) In learning mode, it outputs the logical AND of the pixels from the keyboard image, the user note blob, and the learning note blob.
7) In game mode, it outputs the pixels from the keyboard image if vcount is at least 640. Otherwise, it outputs 0 if vcount is 0, 160, 320, or 480 (for the horizontal lines). Otherwise, it outputs the logical AND of the pixels from the 5 note blobs, as well as the user note blob.

D. Song Creation (Jacob)

In song creation mode, I needed to sample the user's input from the keyboard every 0.25 seconds, and then send the appropriate signals to store these input notes in our song BRAM.

The create_song module has an enable input and a 7-bit input value (based on what note the user is playing). It outputs an 8-bit data value to write to the BRAM, a write_enable signal, and an address to write to in the BRAM.

Because we want to sample the user's playing every 0.25 seconds, we wait 25,000,000 clock cycles in between samplings (due to the 100 MHz clock).

When enable is first asserted, the address value is initialized to 753. This is because address is always incremented by 1 just before writing to the BRAM, meaning the first location written to is 754. This ensures that the first 4 notes of a custom song will all be blanks (giving the user time to get to the correct starting note in game mode), since we initialize the song BRAM to have blanks at addresses 750, 751, 752, and 753.

Then, when enable is high, a counter is incremented every clock cycle. When it has reached 24,999,999, the user input value is written to the next location in the BRAM (so write_enable is set high for one clock cycle), and the counter goes back to 0. This continues as long as enable is high and there is still room left in the BRAM.

We allow for 242 notes to be input by the user (and at one note every 0.25 seconds, this corresponds to 1 minute and a half second for a custom song). When the user has reached the maximum allowable memory location in the BRAM, create_song never sets write_enable high again. Since the BRAM is initialized with an END_NOTE special value (indicating to the game logic that the song is over) in the final address, create_song does not need to add this value.

However, if enable stops being high (a user has left song creation mode), then create_song immediately (on the next clock cycle) writes the END_NOTE special value to the next address in the BRAM (so that the game logic will know that the song ends there). After this, as long as enable is low, write_enable is set low so that the BRAM is not changed.

Due to this setup, if a user again enters song creation mode, they will simply overwrite their previous custom song.

All of the outputs from the create_song module get passed up to the top_level module, which instantiates the BRAM. The BRAM is dual port, so the output values from create_song are the input values for the write port on our song BRAM.

E. Game Logic (Rahul)

The game controller is a large finite state machine that tracks every state of the game, as well as the user’s score and progress. As input, it takes a pulse indicating whether or not the game is currently active, the note that user is currently playing (represented by a MIDI index), the current game type (learn or play), button inputs (for menu navigation), and a 1-bit indicator of whether a custom song has been created.

The FSM has 6 states, each of which are only used when the game is on. When the game is off, as indicated by a 0 on the game_on input, the module simply zeros out all the internal state and waits to be turned on once again. When the game is powered on, it enters the idle state, STATE_IDLE, and latches all of the relevant input values, such as the current game type (play/learn). It then progresses to the song selection state (STATE_SONG_SELECT), which enters into a menu (see menu module below) that allows the user to select which song they wish to play or learn. Once this has been chosen, the FSM saves the song selection as state, and feeds a start signal into the song select module (described below). This allows the song select module to begin feeding a shift register of 5 notes to the game controller, which are used for the actual gameplay.

The game controller progresses to one of two states, either the play (STATE_PLAY) or learn (STATE_LEARN) mode. In both of these states, the FSM uses almost the same logic. In the play state, It uses a counter to sample the user’s input note every tenth of a second, and it compares this to the expected note from the song selection module to determine whether the user is correctly playing the note. If they are indeed playing the correct note, it increments their score. In either case, it increments a second score counter which represents the maximum possible score for the current song. In playing mode, this is the only logic needed, as the song select module automatically advances the notes every quarter of a second, as described in the song select module section below. However, in the learn state, the module samples the user's note continuously, in order to determine whether or not the user has held the correct note for the correct length of a quarter of a second. Once they have, it asserts an advance_note signal to signal to the song select module to shift in the next note. In this way, the user is able to correctly progress through the song, and thus learn it. In both states, the module has special logic for when it encounters a sentinel index that represents the end of a song (7'b111_1100). Once it has encountered this note index, it advances to the next state, SONG_FINISH, which represents the end of the game.

In STATE_FINISH state, the game gets ready to reset. It zeros out all the values except for the user's score and the maximum possible score of the song. These two values are set as outputs back up to the top level module, where they are used to calculate and display the user's final percentage score. It then moves back to STATE_IDLE, to wait for the next game to be played. This extra state mainly serves to signal to the top level that the game has finished, and to transfer control of the VGA display back to this upper level. When the module is in STATE_FINISH, it sends this to the top level through an output register that allows the upper level to revert its internal state back to its top level menu, and thus restart the system.

The sixth state is STATE_MODE_SELECT, which is not currently used. It is intended for our third stretch goal, allowing the user to input notes by singing, rather than playing the keyboard. This mode takes the user to an extra menu that allows for the option to choose whether they will play keyboard or sing into a mic, but we did not complete implementing this stretch goal.

F. Song Select Module (Rahul)

The Song Select Module represents the interface between the Game Controller and the BRAM that holds the songs. It takes as input the desired song index as well as a start pulse that indicates to it when it should begin outputting that song, and whether it is operating in play or learn mode. In turn, it outputs a shift register that represents the next 5 notes in the song.

In addition to these control signals, the song select module also outputs the current address it wishes to read from and takes as input the note from that address. We could not directly put the BRAM into this module, because the song creation module (discussed above) also needs to interact with it. Thus, we had to place the BRAM itself at a higher level, and route the signals up to it. However, for the intents of the abstraction presented to the Game Controller (a module that takes the song index and a start signal and outputs the top 5 notes), this does not make a difference.

When the start signal is asserted it latches the current song index and the game type (play or learn). It then resets the current address to the beginning of the desired song. We placed every song in a fixed 250 byte segment of the BRAM, so these addresses were easy to calculate. It then enters a “startup” state, which is used to quickly load the first five notes of the song in 10 clock cycles (because the BRAM has a 2 cycle read latency). After performing this load, it moves to the regular state, which has two different behaviors, depending on whether it is operating in play or learn mode. In play mode, the module simply uses a counter to advance the address being read from the BRAM every quarter of a second, as each note represents a quarter of a second of song length. In learn mode, the module uses an input advance pulse to determine when to advance to the next note. In both modes, when the module reads a note index that represents the end of the song, it stops reading from the BRAM and begins shifting in rests. This is important, as the display may otherwise show random values falling on the screen after the end of the song, depending on whatever values may come after the song in memory.

One final important output of this module is a pulse that goes high every time a new note is shifted into the notes shift register. This is used by the pixel_helper module to correctly update the display with the notes shifting in. The fact that the signal is used by the VGA logic is important, because VGA is clocked at 65MHz, which is slower than the game’s 100MHz clock. Thus, although this signal is really a pulse, it goes high for four clock cycles every time it should represent a new song. This duplication is required so that the slower clock domain is able to still correctly sample the pulse.

In order to quickly and correctly generate the coe files for the song BRAM, I wrote a Python script that followed our internal format. I have included this at the end of the writeup in Appendix B.

G. Learning Mode (Rahul)

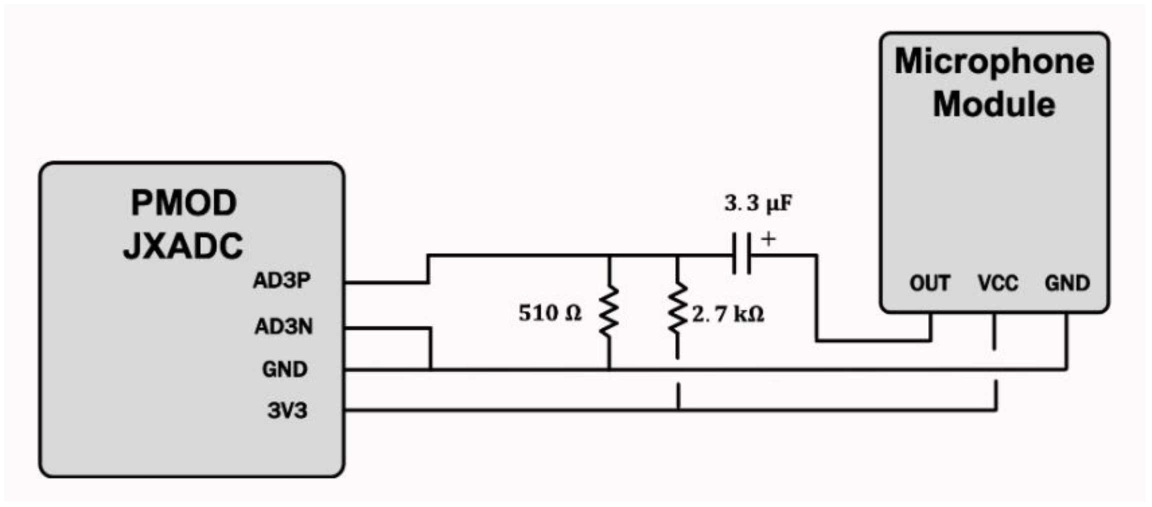
The learning mode of the game is part of the game controller and song select modules, as described above. It requires the addition of extra state, which is described in more detail here. The learning mode is similar to the playing mode, in that it reads notes from the BRAM and sends them out to the VGA display modules. However, the mechanism for advancing notes is different, as it must now wait for user input and advance depending on whether or not the user is correctly playing notes, rather than keeping an internal timer to continuously advance notes. To this end, the game controller was augmented with a special state that works similarly to a debouncer; it tracks the most recently played note, and for how long it has been played. When the most recently played note is the current note of the song, and its length is equal to the correct note length (a quarter of a second), it advances the song forward to the next note. In order to allow for this external control on the note advancement, an advance input signal had to be added to the Song Select module, which in turn had to maintain an internal state of whether it was operating in play mode (and thus should advance notes on its own), or whether it was operating in learn mode (and should only advance notes when told to).

This extra functionality was possible to add in because of the clean abstractions chosen for other aspects of the game. By splitting the game control logic out into a separate FSM, adding the main learning mode logic was clean, as it represents an extra state in the game controller FSM. Furthermore, the separation of the song select module to act as an interface to the BRAM made it relatively clean to account for the fact that learning mode only advances notes when the user plays the correct note. The game controller took care of tracking all of the user input, and then only had to present a pulse on note advancement to the song select module, which only had to use this signal to determine when to advance notes. The modularization allows for clean coding and separation of functionality.

H. Microphone Input (Rahul)

The microphone input component of this project allows the user to navigate menus by whistling high or low notes. In order to accomplish the microphone input, I wrote two main modules, the fft_analyzer and fft_sampler modules. I also used some ancillary modules, such as the timer described below.

To perform the microphone input, I first had to actually set up an external microphone that provided an analog audio signal to the Nexys4. I used a small microphone breadboard module from lab and set up a quick circuit that used a capacitor to couple the signal and overlay it on a 0.5V DC offset, set up by a voltage divider between power (3.3V from the Nexys4) and ground. The microphone module outputs a 1V peak to peak signal, and the Nexys4 analog input reads on the range 0 to 1V, so this circuit allowed for maximum sensitivity to the analog waveform. The circuit is the same as is used in Lab 5A, and I have included an image from the lab below.



Microphone Input Circuit, image from Lab 5A
(http://web.mit.edu/6.111/volume2/www/f2019/handouts/labs/lab5_19a/index.html)

The fft_analyzer runs on a 104MHz clock, because the Nexys4's internal ADC module uses 26 clock cycles to take samples. Thus, by clocking at a multiple of 26, I was able to configure the ADC IP block to exactly output a 1MHz sampled digital signal from the microphone input. The module consists of two major pieces.

First, it contains an entire pipelined FFT implementation, inspired heavily by the FFT Demo provided in class. This implementation performs a 4096 sample FFT of the current input audio at a rate of 60Hz, providing more than enough resolution to correctly identify the user's whistling and respond quickly. Second, it contains an FSM that performs all of the functions necessary to analyze the FFTs taken and determine whether a low or a high note is being whistled.

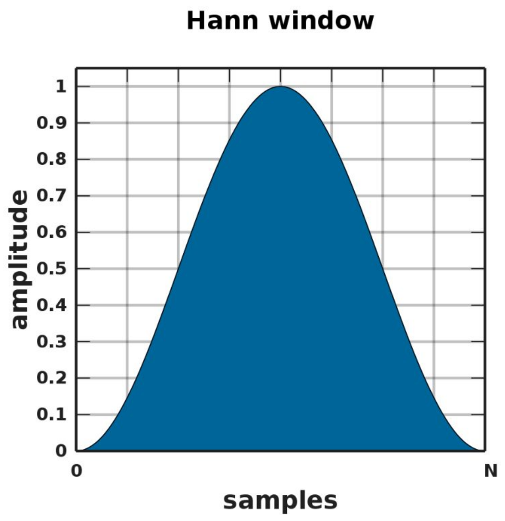
H.i. FFT Calculation

The first stage of the FFT calculation is an ADC IP block, which takes as input the analog signal from the vauxp3 and vauxn3 ports from the Nexys4 and outputs a 12-bit 1MHz sampling of the analog audio.

The second state of the pipeline is an oversampling step, which oversamples the digital audio sample by 16. In particular, it performs a sliding window summation of every 16 samples, and divides the resulting waveform by 4, so that it adds 2 bits to the waveform, resulting in 14-bit samples. This oversample allows the waveform to be downsampled to 62.5KHz, which is much better for the FFT we are interested in taking. We are differentiating whistling by a human, which ranges in the low 1000s and high 100s of Hz, so the FFT needs to be granular enough to clearly distinguish these frequencies. Thus, using a 1MHz sampling rate is useless, as it represents far too many high frequencies that are not relevant. By downsampling to 62.5 KHz, we only consider frequencies up to 31.25 KHz, which is a tighter range around our relevant frequencies. This smaller range also allows the FFT to fit within a relatively small window, so that it does not use up too many resources on the chip (especially when they are not needed). The average oversampling is used rather than a simple sample in order to introduce less noise through the sampling process.

The next stage in the FFT pipeline is a BRAM that stores a frame of the most recent 4096 data samples from the oversampling stage. This allows the stages further down the line to read the input data when it is convenient, rather than having to read and buffer values when they are inputted. This BRAM simply has a looping counter that keeps writing in data from the oversampling stage. Because it is writing in data from the oversampling stage, it has to correctly synchronize with the oversampling stage and only write in a sample every 16 cycles, when a new one is ready.

The fourth stage in the FFT processing pipeline is a module to move data from the BRAM to the FFT IP block. Because the FFT is calculated only once every 60th of a second, it reads from the data BRAM much less frequently than data is written to it, so this module acts as an interface between the two. This module is triggered on the 60Hz pulse to send the current 4096-sample data frame to the FFT. It also uses the frame_tready signal from the FFT module, which indicates that the FFT IP is ready to receive the next data value, to advance the address it is currently reading from the BRAM. This module is a simple counter FSM, and simply maintains its current position in the BRAM and advances this position every time the FFT IP indicates over frame_tready that the next sample should be sent. One extra piece that I added is a Hann windowing filter over this BRAM data. I created a 4096 element lookup table that holds Hann filter coefficients (rounded to 24 bits). I then read the correct element from the LUT and multiply it against the data value read from the BRAM to correctly filter the outputted data. Because the FFT takes in only a single finite frame of data, it implicitly treats it as though it is repeated infinitely. Thus, if the edges do not smoothly match up, the resulting FFT is very noisy. I used the Hann windowing filter to create a clean and more easily analyzable output FFT (see the image below for the filter). I have also included the Python script I used to generate this lookup table in Appendix B.



Hann Window of length N, image from Wikipedia (https://en.wikipedia.org/wiki/Hann_function)

The fifth stage in the FFT pipeline is the actual FFT IP block. This is a specially designed FFT logical block that performs a fully-pipelined 4096 element FFT. I used the IP module from the FFT Demo provided in class without modification, as a length of 4096 provided enough granularity in the resulting FFT to distinguish the frequency range I was interested in.

The sixth stage of the FFT is another BRAM that holds the results of the FFT that is calculated 60 times a second. Because the FFT is calculated so infrequently, it is desirable to cache and hold the results in a BRAM, from which they can be read when needed, rather than having to immediately begin processing the FFT samples as they arrive. Thus, the final stage of the FFT is a BRAM that simply takes as input the output of the FFT IP and writes to the BRAM in sequential order. The BRAM actually only holds the magnitude of the first 1024 of the 4096 total samples, because this is block represents the range of frequencies that the whistle could range in, and is enough for the following analysis.

H.ii. FFT Analysis

The FFT analysis runs 60 times a second, processing each FFT after it has completely been written to the output BRAM. As soon as this happens, the first stage of the analysis is the fft_sampler module. This module performs a sweep over a subset of the 1024-sample FFT and determines the index of the largest bucket. It only considers samples between indices 8 and 300, both of which are parameters that I tuned so that my system was responsive only to the frequency range of whistling. Within this range, it performs a sweeping window-sum of the samples, and saves the index of the window that has the largest magnitude. This window sum, rather than simply choosing the largest sample, helps to filter out any extraneous noise that randomly shows up, because I noticed after spending time analyzing FFT outputs that the true frequency content generally has ancillary content in neighboring frequency buckets, whereas noise is more isolated to a single frequency range. The sample performs this scan using an FSM that is generally in an idle state, but then moves to a sweep state whenever a start input (based on the completion of a new FFT in the FFT calculation stage above) pulse is asserted. In the sweep state, it keeps an accumulating sum and a shift register of values currently in the window, calculates the sum at each index, and saves the index with the largest sum, as well as the actual sum. If the actual sum is larger than a threshold value, which I tuned to make sure that the system was only responsive to intended input and not environmental noise, it returns the index; otherwise, it returns a value to indicate that there is no current input.

The three tunable parameters of the fft_sampler module are very important for tuning the system to work for a given input frequency range. Because this project dealt with whistling, I tuned the set of considered FFT indices to the lower end of the spectrum, 8 to 300. Because the underlying audio samples are at 62.5KHz and this is a 4096 sample FFT, that represents frequencies from 122Hz to 4.5KHz, which is a reasonable range for human whistling. The third parameter, the threshold magnitude for the windowed sum, is important to rule out any environmental noise. The environment always has a reasonable frequency component, and may randomly be large enough to dominate the input, especially when no noise is currently intended to be inputted to the system. By increasing the minimum threshold for how large an FFT window must be before it is considered actual input into the system, I could make it reject more noise and be more sensitive only to close inputs.

After this, there is a final FSM that keeps track of whether the user is inputting a low or a high whistle into the microphone. It cannot simply check at each cycle whether the user is inputting low or high and return this value, because the microphone was noisy enough that this was not a consistently high value as the user whistled. Thus, even with rising edge detection, the inputted note would look more like several inputted notes and the user would shoot up or down the menus without having any fine grained control. Thus, I used a more sophisticated FSM to track user input.

The high/low detection works as follows. When the system is in the tracking state, if it detects either a high or a low note for long enough (both the cutoff frequency between high/low and the minimum required length are tunable parameters), it outputs the correct value as a detected note, and enters the waiting state. In the waiting state, the system starts a 0.5 second timer, and holds the previously detected note on the output until this timer expires. By using a timer, the system makes sure that it represents the users intended input with a clean output signal for a sufficiently long time and does not provide the user such fine grained control (e.g. 104MHz control) that they would quickly shoot up or down the menu even with a short whistle. To implement this behavior, the FSM has two states: STATE_TRACKING and STATE_WAITING. It uses a 2-bit state variable to represent the current state of the detected audio, either high, low, or none. This state variable is updated as described. Further, it uses a counter variable to count how long the current note has been held, in order to determine when it has been held long enough to be considered a valid input.

There are three important tunable parameters in this case. The cutoff frequency between high and low notes is important to make sure that the system is usable. If the cutoff is too low or too high, the user will not be able to input both high and low notes and will not be able to navigate the menus with sound. Thus this threshold must be tuned so that it is easy to whistle both below and above it. The minimum length of a note to be considered a valid input is important to screen out noise. Many notes in the valid range may be generated for a cycle or two due to environmental noise. By making this parameter large enough, it is possible to screen out such noise. Making it too large, however, makes the system feel unresponsive, as the user will have to hold the note for a long time. Thus, there is a tradeoff between noise sensitivity and usability with this parameter. Finally, the third parameter is the length of the timer, which again provides a tradeoff between usability and sensitivity. If the timer is too short, the system will be too sensitive, and will not be able to accurately navigate menus. However, if the timer is too short, the system will be too slow to respond, and will provide a bad experience. Thus, it is important to choose good values for these parameters.

The overall output of the entire fft_analysis module are two 1-bit signals, hi and lo, indicating whether a high or low note (respectively) are being inputted into the system. This interface provides a very clean integration with the top level, as it looks exactly equivalent to the btnu and btnd buttons on the Nexys4, which provide an alternative means of navigating menus.

I. Top Level

The top level module represents an integration of all of the major components described above. It essentially just consists of instantiations of all of the major modules, such as the UART decoder, VGA display, Game Controller, and FFT analyzer. In addition, it also maintains a small FSM to allow for the top level module to choose between the four main modes of the system: keyboard mode, song creation mode, learning mode, and playing mode. The FSM has two states, to represent whether it is currently in the top level menu, or whether it is operating in one of the chosen modes. There are two other main pieces of logic in the top level module. The first is a 1-bit signal that indicates whether or not a custom song has been created. It starts as 0, and goes to 1 once a new song is created. This signal is routed to the game_controller module for its user. In addition, it also contains logic to switch the input mode of the system. Our baseline goal was to implement the system with switches as input, so we maintained this input option in the final version of the project. It can be activated by flipping on switch 15, at which point the system will operate through the switches on the Nexys4, rather than the keyboard. This was purely for demonstration of the baseline goals, and it is expected that most users will prefer to play the game on the keyboard.

The other main purpose of the top level module is to correctly debounce and synchronize all of the signals. All of the user inputted signals must be debounced (buttons) and synchronized (switches) to the main 100MHz clock domain. Furthermore, the system uses 3 clock domains, 65MHz, 100MHz, and 104MHz, so any signals that are passed between these different boundaries must be synchronized to their new clocks. All of this synchronization logic is provided in the top level.

Finally, because the song BRAM is written to by the song creation mode and read from by the Song Select module, we place the actual BRAM itself in the top level because it is the lowest common ancestor of these two modules in the hierarchy of the project. We route all of the relevant signals (read and write addresses, write data, write enable) up to the top level to interface with the song BRAM.

J. Miscellaneous Modules (Menu, Timer, Score Calculation) (Rahul)
Many of the major modules used several helper modules that performed common subtasks in the larger components of the project. We briefly list a few of these modules here to describe some of the required functionality.

The Menu Module serves as an interface for all of the menus in the system. It provides an FSM that tracks the user's current menu choice based on an input up and down signal. It outputs this menu choice. It has a programmable upper bound, so that any number of menu items can be represented and dynamically changed. This module is used for all the menus in the system.

The Timer module provides a countdown timer for the FFT analysis. This countdown timer starts whenever the start input pulse is asserted, and counts down for a variable length, depending on the length inputted to it. It latches this length and counts down from it until it expires, and then is ready to start again with a new length. It perform this countdown with an FSM that has two states: counting down and idle.

The Score Calculation module provides an approximate decimal division functionality. It takes as input the user's score and the maximum possible score they could achieve, and returns their percentage score, rounded to the nearest integer. It accomplishes this by using two lookup tables for powers of 10 and 16, and by starting from the largest possible power of 10 (10^2) and subtracting off ever decreasing multiples of until the score hits zero. In other words, simulates division through repeated subtraction. It begins this computation every time a start signal is asserted, at which point it latches the current score and maximum score, and performs the percentage calculation.

IV. Trickiest Pieces of the Project

Jacob

One of the hardest pieces was getting the external circuit connecting the MIDI keyboard to the Nexys4 working. Originally, I used resistor values that were too low, which caused me to inadvertently blow out the LED inside the opto-isolator I was using. I spent hours trying to figure out why the circuit was not working. Gim helped me debug this and helped me choose resistor values in order to prevent the internal LED from blowing out.

Another difficulty I found was that occasionally, if a key was hit or released very quickly on the MIDI keyboard, no message would be sent. Therefore, the UART_decoder module would not register the change in what note(s) were being played. This initially led to some issues, because if the UART_decoder module thought some note was continually being played, then any other note played would then register as an invalid note, since then multiple notes were on. In order to fix this issue, I allowed for a reset of just the 88-bit register that keeps track of which notes are currently being played, using button btnl on the Nexys4. That way, if this ever happened, I could press btnl to indicate that no note is being pressed at the moment.

Perhaps the trickiest bug I had was that when we first integrated the game logic with the falling notes display for game mode, it worked most of the time, but every so often, the display would glitch, showing notes jumping all over the screen. It was interesting that it was always the case that a song glitched throughout, from beginning to end, or not at all. This was very difficult to debug because despite my best efforts, I could not find a pattern regarding when it would glitch and when it would not. I tried updating the x-locations of the 5 different notes on the display at different times from one another in order to see if perhaps the issue was with only some of them, but I could not find anything from that.

Then, I realized that the issue was occurring because of the slower clock speed of pixel_helper than the game logic. The game logic runs at 100 MHz, while pixel_helper runs at 65 MHz. The display logic for the falling notes, as explained above, relies crucially on the new_note signal, which is an input to pixel_helper that goes high when a new note has just been shifted into the shift register. Even though we were synchronizing this signal for the 65 MHz clock, I realized that because the game logic clock is faster, a single clock cycle of the 100 MHz clock might be too fast for the 65 MHz clock to see, depending on the shift between the clocks at the start of gameplay. This was the root of the bug, and also explains why there seemed to be no pattern to when there was or was not glitching; glitching was determined by the shift between the clocks at the moment the game was started.

In order to remedy the situation, since the 100 MHz clock is less than twice as fast as the 65 MHz clock, we made the new_note signal go high for 4 clock cycles. As soon as we did this, the display worked smoothly.

Rahul

One of the trickiest pieces of this project for me was the note representations of the songs, and how to correctly detect and deal with songs ending. Early on, I played around with the idea of storing the length of the song in the BRAM, before the notes of the song. This was a pretty simple idea, and it allowed for the songs to be variably sized. However, the implementation would be fairly complicated, as the lengths of songs would have to be read into register and held as state throughout the duration of the song, and passed on to interested modules, like the game controller and scoring modules. Further, I knew that we would try to complete the song creation mode, and this would complicate that logic as it would have to track and write in song lengths. Thus, I came up with a different way to represent the ends of songs. In particular, I assigned a special sentinel value that was unused by the note indices to represent the ending of a song. This representation allows for the songs to be played and the end states to be calculated on the fly, without needing any prior knowledge about the songs. This made it much simpler to track song state and to transition between songs when they ended. It also made it much easier later on to keep score, as all I had to do was add an extra counter that always incremented score, rather than only when the user gets the correct note, and I had the correct maximum score value.

The FFT analysis segment of this project also had many tricky parts. First off, it was difficult to get the FFT itself correctly functioning

After I had a clean FFT output that could correctly detect the input note, it was even more challenging to devise a system that would detect the user input in a responsive and smooth manner. The first issue to tackle was detecting the dominant note being inputted into the microphone at any given time. The microphone picks up a lot of environmental noise, so it was not enough to simply sweep through the frequency range and choose the largest bucket. Because of random noise input, this would sometimes be an invalid bucket index. Instead, after analyzing many FFT frames, I noticed that any valid input note produced relatively large frequency components in the immediately neighboring buckets. Thus, I decided to use an averaging window of size 2 to find the largest bucket, rather than a simple maximum. This choice provided robustness to noise.

After fixing this frequency detection issue, there were still two main issues with user high/low differentiation. The first is that the detection is happening at 60Hz (the number of times per second an FFT is taken). Thus, if I naively outputted the detected hi/lo bucket at every sample, the system would be far too fast for a human to operate it, and any attempt to input a note would result in the menu pointer quickly advancing through the entire menu. The second issue is that the FFT itself was still fairly noisy, and so I could not necessarily assume that a user holding a steady input note would result in a steady high/low detection (if this was the case, I could have used an edge detector to fix the first issue). Thus, in order to remedy these issues, I used a timeout based technique. First, I ensured that the user's note was held for a sufficient number of cycles. At a fraction of a second, this value was short enough to expect to happen without intervening noise, and yet long enough to not accidentally misconstrue environmental noise as user input. Second, I used a timeout clock to insert a 0.5 second delay in between consecutive user inputs. In this manner, I was able to avoid the issue of updating the detected note too fast and making the system unusable. Tuning these two lengths was crucial to making the system responsive to the user and yet robust to noise.

V. Lessons Learned / Advice for Future Projects

Jacob

One lesson I learned that I would like to pass on to future project groups was regarding project management and scope. Initially, our goals were far too ambitious for a 2 person group, given the time constraints of the project. Essentially, we initially hoped to have keyboard mode and game mode, but also allow for user input via singing or via the MIDI keyboard. On top that, to help with the singing and playback of the singing, we were planning on doing some form of noise cancellation, in which our system would learn the characteristics of the noise in its environment and strip those from future recordings for noise-attenuated playback.

Because we saw how ambitious these goals were, we decided very early on to dive into the more challenging-seeming aspects, so that in case they were not feasible, we had time to focus our time elsewhere. I went to lab very early on in the project and began playing around with the FFT demo provided, and saw how much noise there seemed to be, spread throughout the frequency spectrum. I therefore recognized that picking out features of the noise in that environment may be very difficult.

I then did some research online to see if people had ever tried to do noise cancellation on an FPGA. Most of what I found had to do with real-time, active noise cancellation using beam-forming, which was not our intended goal. I soon recognized that noise cancellation would have been an entire project on its own, so we ended up removing that from our project, instead supplementing keyboard and game mode with learning and song creation mode.

The advice I would give future groups is to begin exploring some of the more difficult-seeming yet fundamental pieces of your project early on. This will give you time to assess how difficult and time-consuming that aspect of your project will be, which will give you a much better overall sense of how feasible your project is.

One other piece of advice I have for future groups is to integrate earlier than you think you need to. It always seems that integration should be simple, in that if each piece works, they should just work together. Due to different clock domains, as well as generating IP sources for slightly different boards, Rahul and I ended up spending many days just integrating pieces that already worked individually. While it might be best to integrate as you go as opposed to leaving all integration for the end, certainly make sure to integrate more than a few days before the project is due.

Lastly, I strongly recommend trying to get hardware working as early as possible. Hardware can be deceptively difficult to get working and integrated with the Nexys4, as I learned in trying to interface properly with the MIDI keyboard. Even something as simple as realizing I needed to order a cable, which delayed my working on the keyboard interface for a week, could have been detrimental had I not been working on it early in the project. Therefore, just trying to get some of these more unknown pieces operational early on can be a huge help.

Rahul
One lesson that I think will be very valuable to future groups is to think carefully about the abstractions and formats they wish to use in their project before beginning to implement anything. When I was working on the song selection and game controller logic, I had to create a good way to represent songs in memory and, in particular, how to deal with variable length songs. A naive idea is to store the lengths of the song in the BRAM, before the song data itself, but this implementation turns out to be much more involved and higher latency than the solution I used, as described in the section above on tricky parts of the project. Thinking about this abstraction before implementing it was very useful, and made it much more feasible for me to implement my stretch goal of learning mode afterwards, as this representation was much more amenable to different modes of song iteration. Similarly, by thinking through the operation of the system, I was able to write an abstract menu wrapper that we used throughout the system for all of the menus.

Another general piece of advice that I have for future groups is to begin early and implement the large components as soon as possible. As the semester progresses, all classes inevitably get much busier, and it is very easy to overestimate the amount of time you have left. I think the best way to counter this issue is to find the most complex component of the project and to implement and test it first, so that a large piece of the project is complete early on, rather than tackling smaller, easier pieces first. This way, you will be able to make significant progress later on in smaller chunks, rather than leaving a large and complex piece for the end. Along this line of thinking, I implemented the entire first version of the game controller in the first week of working on the project, and was able to iteratively refine this implementation in smaller chunks throughout the following weeks. Similar to this advice, I would also generally suggest that future teams think through their projects and determine what the most difficult parts will be, so that they are able to appropriately order their work on it. I think that breaking down the work to be done into a clear and directly actionable set of objectives ordered by projected difficulty makes the project much more tractable.

A final piece of advice I have for future teams is to ensure that they begin working with any new concepts as early as possible. Before this project, I had never worked with the FFT module, and I had only very limited experience with audio input to the Nexys4 in general. Thus, understanding and getting these aspects of the system to work was disproportionately difficult. I did not originally anticipate this, and it took many late nights in the lab to figure out how to get the FFT to produce valid results that I could use to begin differentiating note inputs. When calibrating for how long various parts of the projects will take, I would suggest to teams to keep in mind that new concepts take much longer to learn than expected, and that starting on these parts early is a good idea.

Appendix A: Verilog Code

See our GitHub repo (https://github.com/rahulyesantharao/FPGA) to view this code in a more convenient code viewer. We have also included all of the project source code below, for completeness.

top_level.sv

```
///////////////////////////////////////
// Top level module: Integrates all of the components.
module top_level(
    input clk_100mhz,
    input [15:0] sw,
    input btnc, btnu, btnd, btnr, btnl,
    input vauxp3,
    input vauxn3,
    input vn_in,
    input vp_in,
    input [7:0] jb,
    output logic [15:0] led,
    output logic ca, cb, cc, cd, ce, cf, cg, dp,
    output logic [7:0] an,
    output logic aud_pwm,
    output logic aud_sd,
    output[3:0] vga_r,
    output[3:0] vga_b,
    output[3:0] vga_g,
    output vga_hs,
    output vga_vs
);

// setup clocks
wire clk_104mhz, clk_65mhz;
clk_wiz_0 clockgen(
    .clk_in1(clk_100mhz),
    .clk_out1(clk_104mhz),
    .clk_out2(clk_65mhz));

// debounce reset
```

```
logic reset;
debounce btnr_debounce(.clk_in(clk_100mhz), .noisy_in(btnr), .clean_out(reset));

// synchronize switches
logic [15:0] sync_sw;
synchronize sw0_sync(.clk_in(clk_100mhz), .unsync_in(sw[0]),
.sync_out(sync_sw[0]));
synchronize sw1_sync(.clk_in(clk_100mhz), .unsync_in(sw[1]),
.sync_out(sync_sw[1]));
synchronize sw2_sync(.clk_in(clk_100mhz), .unsync_in(sw[2]),
.sync_out(sync_sw[2]));
synchronize sw3_sync(.clk_in(clk_100mhz), .unsync_in(sw[3]),
.sync_out(sync_sw[3]));
synchronize sw4_sync(.clk_in(clk_100mhz), .unsync_in(sw[4]),
.sync_out(sync_sw[4]));
synchronize sw5_sync(.clk_in(clk_100mhz), .unsync_in(sw[5]),
.sync_out(sync_sw[5]));
synchronize sw6_sync(.clk_in(clk_100mhz), .unsync_in(sw[6]),
.sync_out(sync_sw[6]));
synchronize sw7_sync(.clk_in(clk_100mhz), .unsync_in(sw[7]),
.sync_out(sync_sw[7]));
synchronize sw8_sync(.clk_in(clk_100mhz), .unsync_in(sw[8]),
.sync_out(sync_sw[8]));
synchronize sw9_sync(.clk_in(clk_100mhz), .unsync_in(sw[9]),
.sync_out(sync_sw[9]));
synchronize sw15_sync(.clk_in(clk_100mhz), .unsync_in(sw[15]),
.sync_out(sync_sw[15]));
// debounce buttons
logic db_btnc, db_btnu, db_btnd, db_btnl;
debounce btnc_debounce(.rst_in(reset), .clk_in(clk_100mhz), .noisy_in(btnc),
.clean_out(db_btnc));
debounce btnu_debounce(.rst_in(reset), .clk_in(clk_100mhz), .noisy_in(btnu),
.clean_out(db_btnu));
debounce btnd_debounce(.rst_in(reset), .clk_in(clk_100mhz), .noisy_in(btnd),
.clean_out(db_btnd));
debounce btnl_debounce(.rst_in(reset), .clk_in(clk_100mhz), .noisy_in(btnl),
.clean_out(db_btnl));
```

```
// Instantiate the keyboard module, to allow for system input through the Nexys4
switches.
keyboard my_keyboard(.clk_100mhz(clk_100mhz), .sw(sync_sw[9:0]), .vauxp3(vauxp3),
    .vauxn3(vauxn3), .vn_in(vn_in), .vp_in(vp_in), .reset(reset),
.enable(sync_sw[15]),
    .aud_pwm(aud_pwm), .aud_sd(aud_sd));

// 7-segment display
wire [31:0] seg_data;
wire [6:0] segments;
assign {cg, cf, ce, cd, cb, cc, ca} = segments[6:0];
display_8hex seven_seg_display(.clk_in(clk_100mhz), .data_in(seg_data),
.seg_out(segments),
    .strobe_out(an));
assign dp = 1'b1;

// game controller
localparam VGA_IDLE = 3'd0;
localparam VGA_MODE_SELECT = 3'd0;
localparam VGA_SONG_SELECT = 3'd1;
localparam VGA_GAME_PLAY = 3'd2;
localparam VGA_GAME_FINISH = 3'd3;

// edge detectors of up/down buttons
logic old_db_btnd;
logic rising_btnd;
logic old_db_btnu;
logic rising_btnu;
logic old_db_btnc;
logic rising_btnc;

assign rising_btnd = db_btnd & !old_db_btnd;
assign rising_btnu = db_btnu & !old_db_btnu;
assign rising_btnc = db_btnc & !old_db_btnc;

always_ff @(posedge clk_100mhz)begin
```

```
    if (reset) begin
        old_db_btnd <= 1'b0;
        old_db_btnu <= 1'b0;
        old_db_btnc <= 1'b0;
    end else begin
        old_db_btnd <= db_btnd;
        old_db_btnu <= db_btnu;
        old_db_btnc <= db_btnc;
    end
end

////// TOP LEVEL FSM ////////////////////////////////////////
// forward declaration of FFT signals and game state
logic rising_lo;
logic rising_hi;
logic [3:0] game_state;

// top level menu
localparam TYPE_KEYBOARD = 2'd0;
localparam TYPE_4 = 2'd1;
localparam TYPE_LEARN = 2'd2;
localparam TYPE_PLAY = 2'd3;
logic [1:0] current_type_choice;
menu #(.BOTTOM_CHOICE(TYPE_KEYBOARD))
    mode_menu(.clk_in(clk_100mhz), .rst_in(reset), .btn_up(rising_btnu |
rising_hi), .btn_down(rising_btnd | rising_lo), .choice(current_type_choice),
.top_choice(TYPE_PLAY));

// PARAMETERS
// the state of the game when it is completed
localparam GAME_STATE_FINISH = 4'd5;
// state parameter values
localparam STATE_MENU = 1'd0;
localparam STATE_TYPE = 1'd1; // absorbing, for now

// STATE
```

```
logic [1:0] current_type; // the current mode of the system (top level or game
controller)
logic state; // the state the system is in (main menu or subchoice)
always_ff @(posedge clk_100mhz) begin
    if(reset) begin
        state <= STATE_MENU;
        current_type <= TYPE_PLAY;
    end else begin
        case(state)
            STATE_MENU: begin
               state <= (rising_btnc) ? STATE_TYPE : STATE_MENU; // when btnc is
pressed, transition
               current_type <= current_type_choice; // just keep tracking
            end
            STATE_TYPE: begin
                // only go back to the top level when the game finishes
                state <= (game_state == GAME_STATE_FINISH) ? STATE_MENU :
STATE_TYPE;
                current_type <= current_type;
            end
        endcase
    end
end
////////////////////////////////////////////////////

// Determine whether a custom song has been created.
logic enable; // whether or not the system is currently in song creation
assign enable = (state == STATE_TYPE && current_type == TYPE_4);
logic song_created = 1'b0; // whether or not a song has been created
always_ff @(posedge clk_100mhz) song_created <= enable | song_created;

// game controller signals
logic [2:0] game_vga_mode;
logic [1:0] game_menu_pos;
logic [34:0] game_current_notes;
logic [11:0] game_current_score;
logic [11:0] game_current_max_score;
logic [1:0] mode_choice;
logic [1:0] song_choice;
logic new_note_shifting_in;
logic is_game_on; // calculated based on states from the top level FSM
assign is_game_on = (state == STATE_TYPE &&
    (current_type == TYPE_PLAY || current_type == TYPE_LEARN)) ? 1'b1 : 1'b0;

// UART controller - retrieves the note played by the user
logic [6:0] user_note_out_keyboard;
UART_decoder my_note(.jb(jb), .clk_100mhz(clk_100mhz), .reset(db_btnl),
.led(user_note_out_keyboard));

// The user's input note. When switch 15 is on, it uses the switches, and
otherwise, it uses the keyboard.
logic [6:0] user_note_out;
assign user_note_out = sync_sw[15] ? sync_sw[6:0] : user_note_out_keyboard;

// The song creation module
logic ram_wea;
logic [9:0] ram_address;
logic [7:0] ram_write_data;
create_song my_song_creator (.clk_100mhz(clk_100mhz), .enable(enable),
.note_in(user_note_out), .value(ram_write_data), .write_enable(ram_wea),
.address_out(ram_address));

// The song BRAM, which interfaces with both create_song and song_select
(indirectly)
logic [7:0] song_read_note;
logic [9:0] song_read_current_addr;
song_rom my_songs(.clka(clk_100mhz), .addra(ram_address), .dina(ram_write_data),
.wea(ram_wea), .clkb(clk_100mhz), .addrb(song_read_current_addr),
.doutb(song_read_note));

// The game controller.
game_controller #(
    .VGA_IDLE(VGA_IDLE),
    .VGA_MODE_SELECT(VGA_MODE_SELECT),
    .VGA_SONG_SELECT(VGA_SONG_SELECT),
    .VGA_GAME_PLAY(VGA_GAME_PLAY),
    .VGA_GAME_FINISH(VGA_GAME_FINISH))
my_game (
    .clk_in(clk_100mhz),
    .rst_in(reset),
    .game_on(is_game_on),
    .btnu(rising_hi | rising_btnu),
    .btnd(rising_lo | rising_btnd),
    .btnc(rising_btnc),
    .keyboard_note(user_note_out),
    .mic_note(7'b0),
    .game_type_in(current_type),
    .vga_mode(game_vga_mode),
    .menu_select(game_menu_pos),
    .current_notes(game_current_notes),
    .current_score(game_current_score),
    .current_max_score(game_current_max_score),
    .game_state_out(game_state),
    .mode_choice_out(mode_choice),
    .song_choice_out(song_choice),
    .shifting_out(new_note_shifting_in),
    .song_select_read_note(song_read_note),
    .song_select_current_addr(song_read_current_addr),
    .custom_song_activated(song_created)
);

// FFT Analyzer
logic fft_hi, fft_lo;
fft_analyzer fft_in(
    .clk_104mhz(clk_104mhz),
    .vauxp3(vauxp3),
    .vauxn3(vauxn3),
    .hi(fft_hi),
    .lo(fft_lo)
);
```

```
// synchronize fft lo/hi back to main clock
logic fft_sync_hi, fft_sync_lo;
synchronize sync_fft_hi(
    .clk_in(clk_100mhz),
    .unsync_in(fft_hi),
    .sync_out(fft_sync_hi)
);
synchronize sync_fft_lo(
    .clk_in(clk_100mhz),
    .unsync_in(fft_lo),
    .sync_out(fft_sync_lo)
);

// edge detectors of hi/lo buttons
logic old_sync_lo;
logic old_sync_hi;

assign rising_lo = fft_sync_lo & !old_sync_lo;
assign rising_hi = fft_sync_hi & !old_sync_hi;

always_ff @(posedge clk_100mhz)begin
    if (reset) begin
        old_sync_lo <= 1'b0;
        old_sync_hi  <= 1'b0;
    end else begin
        old_sync_lo <= fft_sync_lo;
        old_sync_hi <= fft_sync_hi;
    end
end

// VGA mode calculation - combines the game controller VGA mode and top level
state.
// vga states used by pixel_helper
localparam MAIN_MENU = 3'b000;
localparam KEYBOARD_INSTRUCTIONS = 3'b001;
localparam SONG_INSTRUCTIONS = 3'b010;
localparam BASIC_SONG_MENU = 3'b011;
localparam GAME_MODE = 3'b110;
localparam LEARN_MODE = 3'b101;

logic [2:0] full_vga_mode; // the final VGA mode.
assign full_vga_mode = (is_game_on) ?
  game_vga_mode : ((state == STATE_MENU) ?
    MAIN_MENU : ((current_type == TYPE_KEYBOARD) ?
      KEYBOARD_INSTRUCTIONS : SONG_INSTRUCTIONS));

// VGA signals - sync all of the relevant signals above to the 65MHz clock.
logic [6:0] sync65_user_note;
logic [2:0] sync65_full_vga_mode;
logic [1:0] sync65_game_menu_pos;
logic [34:0] sync65_game_current_notes;
logic sync65_new_note_shifting_in;
logic [1:0] sync65_current_type_choice;
synchronize3 sync_full_vga_mode(
    .clk_in(clk_65mhz),
    .unsync_in(full_vga_mode),
    .sync_out(sync65_full_vga_mode)
);
synchronize2 sync_game_menu_pos(
    .clk_in(clk_65mhz),
    .unsync_in(game_menu_pos),
    .sync_out(sync65_game_menu_pos)
);
synchronize35 sync_game_current_notes(
    .clk_in(clk_65mhz),
    .unsync_in(game_current_notes),
    .sync_out(sync65_game_current_notes)
);
synchronize sync_new_note_shifting_in(
    .clk_in(clk_65mhz),
    .unsync_in(new_note_shifting_in),
    .sync_out(sync65_new_note_shifting_in)
);
synchronize2 sync_current_type_choice(
    .clk_in(clk_65mhz),
    .unsync_in(current_type_choice),
    .sync_out(sync65_current_type_choice)
);
synchronize7 sync_user_note(
    .clk_in(clk_65mhz),
    .unsync_in(user_note_out),
    .sync_out(sync65_user_note)
);

// VGA output
wire [10:0] hcount;    // pixel on current line
wire [9:0] vcount;     // line number
wire hsync, vsync;
wire [11:0] pixel;
reg [11:0] rgb;
wire blank_ignore;
xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
  .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank_ignore));

wire phsync,pvsync,pblank;
//pixel_helper module to get pixel values for VGA display
pixel_helper ph(.clk_65mhz(clk_65mhz), .screen(sync65_full_vga_mode),
.selection((sync65_full_vga_mode == BASIC_SONG_MENU) ? sync65_game_menu_pos :
sync65_current_type_choice),
            .notes(game_current_notes), .new_note(sync65_new_note_shifting_in),
.learning_note(sync65_game_current_notes[34:28]), .user_note(sync65_user_note),
            .hcount_in(hcount),.vcount_in(vcount), .reset(reset),
            .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank_ignore),

.phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(pixel));

reg b,hs,vs;
always_ff @(posedge clk_65mhz) begin
        hs <= phsync;
        vs <= pvsync;
        b <= pblank;
        rgb <= pixel;
    end

// the following lines are required for the Nexys4 VGA circuit - do not change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;
//////

// Performs the percentage calculation of the score and holds this in disp_score.
logic [11:0] disp_score;
score_calc my_score_calc(
.score(game_current_score),
  .max_score(game_current_max_score),
  .start(game_state == GAME_STATE_FINISH),
  .clk(clk_100mhz),
  .disp_score(disp_score)
);

// segment display - show the current note on the top two digits, and the final
score on the bottom 3 digits.
assign seg_data[31:24] = {1'b0, game_current_notes[34:28]};
assign seg_data[23:12] = 0;
assign seg_data[11:0] = disp_score;

// leds - shows the user input note on the top 7 leds and the desired note index
on the bottom 7
assign led[15:0] = (game_vga_mode == GAME_MODE || game_vga_mode == LEARN_MODE) ?
{user_note_out, 2'b0, game_current_notes[34:28]} : 16'b0;
endmodule

// Synchronization Modules
//  - Modules to synchronize values of varying bitwidths.
// 1-bit Synchronizer
module synchronize #(parameter NSYNC=3) (
    input clk_in,
    input unsync_in,
    output reg sync_out
);
reg [NSYNC-2:0] sync;
always_ff @(posedge clk_in) begin
    {sync_out, sync} <= {sync[NSYNC-2:0], unsync_in};
end
endmodule

// 2-bit Synchronizer
module synchronize2 (
    input clk_in,
    input [1:0] unsync_in,
    output reg [1:0] sync_out
);
reg [5:0] sync;
always_ff @(posedge clk_in) begin
    {sync_out[1:0], sync[5:0]} <= {sync[5:0], unsync_in[1:0]};
end
endmodule

// 3-bit Synchronizer
module synchronize3 (
    input clk_in,
    input [2:0] unsync_in,
    output reg [2:0] sync_out
);
reg [8:0] sync;
always_ff @(posedge clk_in) begin
    {sync_out[2:0], sync[8:0]} <= {sync[8:0], unsync_in[2:0]};
end
endmodule

// 7-bit Synchronizer
```

```
module synchronize7 (
    input clk_in,
    input [6:0] unsync_in,
    output reg [6:0] sync_out
);
reg [20:0] sync;
always_ff @(posedge clk_in) begin
    {sync_out[6:0], sync[20:0]} <= {sync[20:0], unsync_in[6:0]};
end
endmodule

// 35-bit Synchronizer
module synchronize35 (
    input clk_in,
    input [34:0] unsync_in,
    output reg [34:0] sync_out
);
reg [104:0] sync;
always_ff @(posedge clk_in) begin
    {sync_out[34:0], sync[104:0]} <= {sync[104:0], unsync_in[34:0]};
end
endmodule

// Debounce Module - debounces an input signal, includes synchronization
module debounce (
    input rst_in,
    input clk_in,
    input noisy_in,
    output reg clean_out
);

reg [19:0] count;
reg new_input;

always_ff @(posedge clk_in) begin
    if(rst_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 20'd0;
    end else if(noisy_in != new_input) begin
        new_input <= noisy_in;
        count <= 20'd0;
    end else if(count == 20'd1_000_000) begin
        clean_out <= new_input;
    end else begin
        count <= count + 20'd1;
    end
end
endmodule

// display module for 7-segment display
module display_8hex(
input clk_in,                  // system clock
input [31:0] data_in,          // 8 hex numbers, msb first
output reg [6:0] seg_out,      // seven segment display output
output reg [7:0] strobe_out    // digit strobe
);
localparam bits = 13;

reg [bits:0] counter = 0;  // clear on power up
wire [6:0] segments[15:0]; // 16 7 bit memorys
assign segments[0]  = 7'b100_0000;  // inverted logic
assign segments[1]  = 7'b111_1001;  // gfedcba
assign segments[2]  = 7'b010_0010;
assign segments[3]  = 7'b011_0000;
assign segments[4]  = 7'b001_1001;
assign segments[5]  = 7'b001_0100;
assign segments[6]  = 7'b000_0100;
assign segments[7]  = 7'b111_1000;
assign segments[8]  = 7'b000_0000;
assign segments[9]  = 7'b001_1000;
assign segments[10] = 7'b000_1000;
assign segments[11] = 7'b000_0101;
assign segments[12] = 7'b100_0110;
assign segments[13] = 7'b010_0001;
assign segments[14] = 7'b000_0110;
assign segments[15] = 7'b000_1110;

always_ff @(posedge clk_in) begin
    // Here I am using a counter and select 3 bits which provides
    // a reasonable refresh rate starting the left most digit
    // and moving left.
    counter <= counter + 1;
    case (counter[bits:bits-2])
        3'b000: begin  // use the MSB 4 bits
            seg_out <= segments[data_in[31:28]];
            strobe_out <= 8'b0111_1111;
        end
        3'b001: begin
            seg_out <= segments[data_in[27:24]];
            strobe_out <= 8'b1011_1111;
        end
        3'b010: begin
            seg_out <= segments[data_in[23:20]];
            strobe_out <= 8'b1101_1111;
        end
        3'b011: begin
            seg_out <= segments[data_in[19:16]];
            strobe_out <= 8'b1110_1111;
        end
        3'b100: begin
            seg_out <= segments[data_in[15:12]];
            strobe_out <= 8'b1111_0111;
        end
        3'b101: begin
            seg_out <= segments[data_in[11:8]];
            strobe_out <= 8'b1111_1011;
        end
        3'b110: begin
            seg_out <= segments[data_in[7:4]];
            strobe_out <= 8'b1111_1101;
        end
        3'b111: begin
            seg_out <= segments[data_in[3:0]];
            strobe_out <= 8'b1111_1110;
        end
    endcase
end
endmodule


//////////////////////////////////////////
// Score Calculation Module - Calculates the score as a percentage to the
// nearest integer. Performs this calculation in an iterative fashion,
// subtracting of multiples of decreasing powers of 10.
module score_calc(
  input logic[11:0] score, // the current score
  input logic[11:0] max_score, // the maximum score possible
  input logic start, // a start pulse to begin the calculation
  input logic clk, // the input clock
  output logic[11:0] disp_score // a 3-digit decimal output (4 bits per digit)
);
// The scaling factor to get to the nearest integer.
localparam ACCURACY = 7'd100;

// state
logic [18:0] dividend = 19'd0; // the score that is currently being divided
logic [11:0] divisor = 12'd0; // the max-score that is dividing
logic [3:0] pow_index = 4'd0; // the index of the current place value
logic [11:0] calculated_score = 12'd0; // the current decimal score

// state transitions
logic [18:0] next_dividend;
logic [11:0] next_divisor;
logic [3:0] next_pow_index;
logic [11:0] next_calculated_score;

// constants (the current place value)
```

```
logic [9:0] multiplier;
logic [9:0] hex_multiplier;
powers_of_ten m(.ind(pow_index), .pow(multiplier));
powers_of_sixteen n(.ind(pow_index), .pow(hex_multiplier));

always_comb begin
  if(divisor > 12'd0 && dividend >= multiplier * divisor) begin
    // we are able to add one more to the current place value
    next_dividend = dividend - multiplier * divisor;
    next_divisor = divisor;
    next_pow_index = pow_index;
    next_calculated_score = calculated_score + hex_multiplier;
  end else begin
    next_dividend = dividend;
    next_calculated_score = calculated_score;
    // we must step down to the next place value
    // stop once the pow index is done with zeros
    next_pow_index = (pow_index > 4'd0) ? pow_index - 4'd1 : 4'd0;
    next_divisor = (pow_index > 4'd0) ? divisor : 12'd0;
  end
end

// state update and outputs
assign disp_score = calculated_score;
always_ff @(posedge clk) begin
  if(start) begin
    // latch the current input values and start at highest place value
    dividend <= ACCURACY * score;
    divisor <= max_score;
    pow_index <= 4'd2;
    calculated_score <= 12'd0;
  end else begin
    dividend <= next_dividend;
    divisor <= next_divisor;
    pow_index <= next_pow_index;
    calculated_score <= next_calculated_score;
  end
```

```
end
endmodule

// A lookup table for powers of 10.
module powers_of_ten(input logic[3:0] ind, output logic [9:0] pow);
always_comb begin
  case(ind)
    4'd0: pow = 10'd1;
    4'd1: pow = 10'd10;
    4'd2: pow = 10'd100;
    default: pow = 4'd0;
  endcase
end
endmodule

// A lookup table for powers of 16.
module powers_of_sixteen(input logic[3:0] ind, output logic [9:0] pow);
always_comb begin
  case(ind)
    4'd0: pow = 10'd1;
    4'd1: pow = 10'd16;
    4'd2: pow = 10'd256;
    default: pow = 4'd0;
  endcase
end
endmodule
```

game_controller.sv

```
`timescale 1ns / 1ps
///////////////////////////////////////////////////////////////
// Game Controller Module - Runs the entire game, including Learn and Play
// modes.
//  - Takes in inputs from the outer level, including button presses
//    to interact with menus and the notes played by the user (on keyboard
//    and microphone).
//  - Outputs the state of the game, including score, current notes, and
//    VGA mode, to be displayed by the VGA helper.
```

```
module game_controller
(
    input logic clk_in, // input clock
    input logic rst_in, // reset signal
    input logic game_on, // indicates whether the game is currently active
    input logic btnu, // up button (menu navigation)
    input logic btnd, // down button (menu navigation)
    input logic btnc, // center button (select)
    input logic [6:0] keyboard_note, // the note being played on the keyboard
    input logic [6:0] mic_note, // the note being sung into the mic
    input logic [1:0] game_type_in, // the current game type (play or learn)
    output logic [2:0] vga_mode, // the current VGA mode, to be fed to
[pixel_helper]
    output logic [1:0] menu_select, // the current menu choice, for
[pixel_helper]
    output logic [34:0] current_notes, // the current notes being played, for
[pixel_helper]
    output logic [11:0] current_score, // the current score
    output logic [11:0] current_max_score, // the maximum possible score
    output logic shifting_out, // a pulse to indicate when a new note is shifted
in
    // debug
    output logic [3:0] game_state_out, // the internal game state
    output logic [1:0] mode_choice_out, // the current mode choice
    output logic [1:0] song_choice_out, // the current song choice
    // song select
    input logic [7:0] song_select_read_note, // the note being read from the song
BRAM
    output logic [9:0] song_select_current_addr, // the address to read from the
song BRAM
    // custom song bit
    input logic custom_song_activated // whether or not a custom song exists, for
the song menu
);
// Game FSM states
localparam STATE_IDLE = 4'd0;
localparam STATE_MODE_SELECT = 4'd1;
localparam STATE_SONG_SELECT = 4'd2;
localparam STATE_PLAY = 4'd3;
localparam STATE_LEARN = 4'd4;
localparam STATE_FINISH = 4'd5; // can reset back to IDLE with btnc

// Game types
localparam TYPE_PLAY = 2'd3;
localparam TYPE_LEARN = 2'd2;

// Game modes
localparam MODE_BITS = 2;
localparam MODE_KEYBOARD = 2'd0;
localparam MODE_MIC = 2'd1;

// Song choice
localparam SONG_BITS = 2;
localparam SONG_1 = 2'd0;
localparam SONG_2 = 2'd1;
localparam SONG_3 = 2'd2;
localparam SONG_4 = 2'd3;
localparam SONG_FINISH = 7'b111_1100;

// Scoring
localparam SCORE_INTERVAL = 24'd10_000_000;
localparam NOTE_LENGTH = 26'd25_000_000; // switch notes every quarter second

// State
logic [3:0] state = STATE_IDLE;
logic [11:0] score = 12'd0; // player's current score
logic [11:0] max_score = 12'd0; // maximum possible score at the moment
logic [23:0] score_counter = 24'd0; // counter to track scoring intervals
logic [1:0] game_type; // learn/play
logic [MODE_BITS - 1:0] mode_choice = MODE_KEYBOARD; // WIP: for the singing
stretch goal
logic [SONG_BITS - 1:0] song_choice = 2'd0; // menu choice of which song to
play/learn
logic advance_note; // indicates that a new note is shifting in
```

```
// only used in learning mode
logic [6:0] old_input_note; // used to track length of current note being played
logic [25:0] old_input_note_counter;

// HELPER MODULES -----------
// mode menu
logic [MODE_BITS - 1:0] current_mode_choice;
menu #(.NUM_BITS(MODE_BITS), .BOTTOM_CHOICE(MODE_KEYBOARD))
    mode_menu(.clk_in(clk_in), .rst_in(rst_in), .btn_up(btnu),
      .btn_down(btnd), .choice(current_mode_choice), .top_choice(MODE_MIC));

// song selection menu
logic [SONG_BITS - 1:0] current_song_choice;
menu #(.NUM_BITS(SONG_BITS), .BOTTOM_CHOICE(SONG_1))
    song_menu(.clk_in(clk_in), .rst_in(rst_in), .btn_up(btnu),
      .btn_down(btnd), .choice(current_song_choice),
      .top_choice((custom_song_activated ? SONG_4 : SONG_3)));

// song select module
logic song_start;
logic [34:0] song_notes;

song_select song_selector(.clk_in(clk_in), .rst_in(rst_in), .start(song_start),
  .game_type_in(game_type), .song_choice(song_choice),
.advance_note(advance_note),
  .notes(song_notes),
.shifting_out(shifting_out),.read_note_in(song_select_read_note),
  .current_addr_out(song_select_current_addr));

// STATE TRANSITIONS ------------------------------------------
logic [3:0] next_state;
logic [11:0] next_score;
logic [11:0] next_max_score;
logic [23:0] next_score_counter;
logic [MODE_BITS - 1:0] next_mode_choice;
logic [SONG_BITS - 1:0] next_song_choice;
logic next_song_start;
logic [1:0] next_game_type;
logic next_advance_note;

// only used in learning mode
logic [6:0] next_old_input_note;
logic [25:0] next_old_input_note_counter;

logic [6:0] input_note;
assign input_note = (mode_choice == MODE_KEYBOARD) ? keyboard_note : mic_note;
always_comb begin
    if(game_on) begin // only track states if currently in play/learn mode
        next_game_type = game_type; // once the game is on, type is fixed
        next_old_input_note = input_note;
        case(state)
            STATE_IDLE: begin
                next_state = STATE_SONG_SELECT;
                // hold these values until the next scoring period
                next_score = score;
                next_max_score = max_score;
                // just zero these out until they are needed
                next_score_counter = 24'd0;
                next_mode_choice = mode_choice;
                next_song_choice = 2'd0;
                next_song_start = 1'b0;
                next_advance_note = 1'b0;
                next_old_input_note_counter = 26'd0;
            end
            STATE_MODE_SELECT: begin // WIP: used for singing stretch goal
                next_state = (btnc) ? STATE_SONG_SELECT : STATE_MODE_SELECT;
                // hold until the next scoring period
                next_score = score;
                next_max_score = max_score;
                next_mode_choice = current_mode_choice; // keep changing with
current choice
                next_song_choice = song_choice; // don't change
                // just zero these out until they are needed
                next_score_counter = 24'd0;
                next_song_start = 1'b0;
                next_advance_note = 1'b0;
                next_old_input_note_counter = 26'd0;
            end
            STATE_SONG_SELECT: begin
                next_state = (btnc) ?
                  ((game_type == TYPE_PLAY) ? STATE_PLAY : STATE_LEARN) :
STATE_SONG_SELECT;
                // reset these values when the game/learning is started
                next_score = (btnc) ? 12'd0 : score;
                next_max_score = (btnc) ? 12'd0 : max_score;
                next_mode_choice = mode_choice; // don't change
                next_song_choice = current_song_choice; // keep changing with
current choice
                // signal when the song is about to start
                next_song_start = (btnc) ? 1'b1 : 1'b0;
                // just zero these out until they are needed
                next_score_counter = 24'd0;
                next_advance_note = 1'b0;
                next_old_input_note_counter = 26'd0;
            end
            STATE_PLAY: begin
                if(song_notes[34:28] == SONG_FINISH) begin
                    // we have reached the end of the song, transition to the
                    // FINISH state
                    next_state = STATE_FINISH;
                    next_score = score;
                    next_max_score = max_score;
                    next_score_counter = 24'd0;
                end else begin
                    next_state = STATE_PLAY;
                    // update the score every SCORE_INTERVAL/10M seconds
                    if(score_counter < SCORE_INTERVAL - 24'd1) begin
                        next_score = score;
                        next_max_score = max_score;
                        next_score_counter = score_counter + 24'd1;
                    end else begin
                        // sample and check whether the input note is correct
                        next_score = score +
                          ((input_note == song_notes[34:28]) ? 12'd1 : 12'd0);
                        next_max_score = max_score + 12'd1;
                        next_score_counter = 24'd0;
                    end
                end
                // maintain these values
                next_mode_choice = mode_choice;
                next_song_choice = song_choice;
                next_song_start = 1'b0; // don't restart the song once we have
entered game mode
                next_advance_note = 1'b0;
                next_old_input_note_counter = 26'd0;
            end
            STATE_LEARN: begin
                if(song_notes[34:28] == SONG_FINISH) begin
                    // we have reached the end of the song, transition to the
                    // FINISH state
                    next_state = STATE_FINISH;
                    next_advance_note = 1'b0;
                    next_old_input_note_counter = 26'd0;
                end else begin
                    next_state = STATE_LEARN;
                    // advance to the next note if the played note is correct
                    // and has been held for the appropriate length
                    if((input_note == song_notes[34:28]) && (input_note ==
old_input_note)) begin
                        next_advance_note = (old_input_note_counter ==
NOTE_LENGTH - 26'd1) ?
                          1'b1 : 1'b0;
                        next_old_input_note_counter =
                          (old_input_note_counter == NOTE_LENGTH - 26'd1) ?
                            26'd0 : old_input_note_counter + 26'd1;
                    end else begin
                        next_advance_note = 1'b0;
                        next_old_input_note_counter = 26'd0;
                    end
                end
                // maintain these values
                next_score = score; // no scoring in learning mode
                next_max_score = max_score;
                next_score_counter = score_counter;
                next_mode_choice = mode_choice;
                next_song_choice = song_choice;
                next_song_start = 1'b0; // don't restart the song once we have
entered game mode
            end
            STATE_FINISH: begin
                // this state serves to reset state and signal the end to
                // top_level
                next_state = STATE_IDLE;
                next_score = score;
                next_max_score = max_score;
                next_score_counter = 24'd0;
                next_mode_choice = mode_choice;
                next_song_choice = song_choice;
                next_song_start = 1'b0;
                next_advance_note = 1'b0;
                next_old_input_note_counter = 26'd0;
            end
        endcase
    end else begin
        // game is turned off, switch back to initial state
        next_state = STATE_IDLE;
        next_score = score;
        next_max_score = max_score;
        next_score_counter = 24'd0;
        next_mode_choice = MODE_KEYBOARD;
        next_song_choice = 2'd0;
        next_song_start = 1'b0;
        next_game_type = game_type_in; // latch the game type
        next_advance_note = 1'b0;
        next_old_input_note = 7'd0;
        next_old_input_note_counter = 26'd0;
    end
end


// OUTPUT & STATE UPDATE -------------------------------------
// VGA helper signals
localparam MAIN_MENU = 3'b000;
localparam KEYBOARD_INSTRUCTIONS = 3'b001;
localparam SONG_INSTRUCTIONS = 3'b010;
localparam BASIC_SONG_MENU = 3'b011;
localparam CUSTOM_SONG_MENU = 3'b100;
localparam LEARN_MODE = 3'b101;
localparam GAME_MODE = 3'b110;

// a large multiplexer to decide what to display on the screen
assign vga_mode = (state == STATE_MODE_SELECT) ? MAIN_MENU :
                    ((state == STATE_SONG_SELECT) ? (custom_song_activated ?
CUSTOM_SONG_MENU : BASIC_SONG_MENU) :
                        ((state == STATE_PLAY) ? GAME_MODE :
                            ((state == STATE_LEARN) ? LEARN_MODE : MAIN_MENU)));

// make most of the state available to top_level
assign menu_select = (state == STATE_MODE_SELECT) ? current_mode_choice :
current_song_choice;
assign current_notes = song_notes;
assign current_score = score;
assign current_max_score = max_score;
assign game_state_out = state;
assign mode_choice_out = mode_choice;
assign song_choice_out = song_choice;
always_ff @(posedge clk_in) begin
    if(rst_in) begin // reset to zeroed out values
        state <= STATE_IDLE;
        score <= 12'd0;
        max_score <= 12'd0;
        score_counter <= 24'd0;
        mode_choice <= MODE_KEYBOARD;
        song_choice <= 2'd0;
        song_start <= 1'b0;
        game_type <= TYPE_PLAY;
        advance_note <= 1'b0;
        old_input_note <= 7'd0;
        old_input_note_counter <= 26'd0;
    end else begin
        state <= next_state;
        score <= next_score;
        max_score <= next_max_score;
        score_counter <= next_score_counter;
        mode_choice <= next_mode_choice;
        song_choice <= next_song_choice;
        song_start <= next_song_start;
        game_type <= next_game_type;
        advance_note <= next_advance_note;
        old_input_note <= next_old_input_note;
        old_input_note_counter <= next_old_input_note_counter;
    end
end

endmodule

///////////////////////////////////////////////////////////////
// Song Select Module - Serves as an interface between the Game Controller
// Module and the song BRAM.
//    - Takes as input the song choice and signals to start the song
//      and advance notes, as well as the note read from the BRAM
//    - Outputs the shift register of active notes and the address to
//      read from the BRAM
module song_select (
    input logic clk_in, // input clock
    input logic rst_in, // system reset
    input logic start,  // pulse to start outputting song addresses
    input logic [1:0] game_type_in, // current game type (play/learn)
    input logic [1:0] song_choice, // song index
    input logic advance_note, // pulse to indicate a note advance (only for learn
mode)
    input logic [7:0] read_note_in, // the note read from the song BRAM (in
top_level)
    output logic [34:0] notes, // the 5-note shift register, [34:28] is the
current note
    output logic shifting_out, // a pulse to indicate a new note shifting in
    output logic [9:0] current_addr_out // the address to read from the song BRAM
);
// Constants ----------------------------------
localparam NOTE_LENGTH = 26'd25_000_000; // switch notes every half second

// Initial values for outputs
localparam INIT_NOTES = 35'd0;
localparam INIT_ADDR = 10'd0;

// Game types
localparam TYPE_PLAY = 2'd3;
localparam TYPE_LEARN = 2'd2;

// Special note indices
localparam END_NOTE = 7'b111_1100;
localparam REST = 7'b111_1111;

// STATE ----------------------------------
logic [34:0] current_notes = INIT_NOTES;
logic [25:0] counter = NOTE_LENGTH - 26'd1;
logic [9:0] current_addr = INIT_ADDR; // current address in the song BRAM
logic [3:0] start_counter = 4'd10; // a special register to deal with the startup
routine
logic [1:0] game_type; // holds the type for the current song
logic shifting; // a pulse to indicate a note shifting in
// a shift register to hold the shifting signal (so it can be properly
synchronized to 65MHz).
logic [3:0] shifting_regs = 4'd0;
logic end_notes = 1'b1; // a special state to fill the register with RESTs at the
end

// BRAM Values ----------------------------------
logic [7:0] read_note;
// if the end of the song has been reached, only shift in rests.
assign read_note = end_notes ? REST : read_note_in;

// STATE TRANSITIONS ----------------------
logic [34:0] next_notes;
logic [25:0] next_counter;
logic [9:0] next_addr;
logic [3:0] next_start_counter;
logic next_shifting;
logic next_end_notes;

always_comb begin
    // Startup routine: Quickly shift the first 5 notes into the register in
    // 10 cycles (2-cycle latency in song BRAM)
    if(start_counter < 4'd10) begin
      // Every other cycle, shift in a new note
      next_notes = (start_counter & 4'b1 == 4'b1) ?
        {current_notes[27:0], read_note[6:0]} : current_notes;
      next_end_notes = (start_counter & 4'b1 == 4'b1) ?
        ((read_note[6:0] == END_NOTE) ? 1'b1 : 1'b0) : 1'b0;
      next_addr = (start_counter & 4'b1 == 4'b1) ? current_addr + 10'd1 :
current_addr;
      next_shifting = (start_counter & 4'b1 == 4'b1)? 1'b1 : 1'b0;
      // hold the regular counter and increment the startup counter
      next_counter = 26'd0;
      next_start_counter = start_counter + 4'd1;
    end
    else begin
        // Regular Operation
        case(game_type)
            TYPE_PLAY: begin // Play mode, automatically shift a new note every
NOTE_LENGTH/100M seconds
```

```
                if(counter < NOTE_LENGTH - 26'd1) begin
                    next_notes = current_notes; // stay on same notes
                    next_end_notes = 1'b0;
                    next_counter = counter + 26'd1;
                    next_addr = current_addr;
                    next_shifting = 1'b0;
                end else begin
                    next_notes = {current_notes[27:0], read_note[6:0]}; // shift
in new note
                    next_end_notes = (read_note[6:0] == END_NOTE) ? 1'b1 : 1'b0;
                    next_counter = 26'd0;
                    next_addr = current_addr + 10'd1;
                    next_shifting = 1'b1;
                end
            end
            TYPE_LEARN: begin // Learn mode, only shift in new notes when
[advance_note] is high
                if(advance_note | current_notes[34:28] == END_NOTE) begin
                    next_notes = {current_notes[27:0], read_note[6:0]}; // shift
in new note
                    next_end_notes = (read_note[6:0] == END_NOTE) ? 1'b1 : 1'b0;
                    next_addr = current_addr + 10'd1;
                    next_shifting = 1'b1;
                end else begin
                    next_notes = current_notes; // stay on same notes
                    next_end_notes = 1'b0;
                    next_addr = current_addr;
                    next_shifting = 1'b0;
                end
                next_counter = 26'd0; // don't use counter
            end
        endcase
        next_start_counter = (start_counter == 4'd11) ? 4'd0 : 4'd10; // 4'd11 is
a sentinel to goto startup
    end
end
```

```
// OUTPUT & STATE UPDATE ------------------
assign notes = current_notes;
assign shifting_out = shifting;
assign current_addr_out = current_addr;
always_ff @(posedge clk_in) begin
    if(rst_in) begin // reset to initial values
        current_notes <= INIT_NOTES;
        counter <= NOTE_LENGTH - 26'd1;
        current_addr <= INIT_ADDR;
        start_counter <= 4'd10;
        game_type <= TYPE_PLAY;
        shifting <= 1'b0;
        shifting_regs <= 4'd0;
        end_notes <= 1'b1;
    end else if(start) begin
        current_notes <= INIT_NOTES;
        counter <= 26'd0;
        current_addr <= 250 * song_choice;
        start_counter <= 4'd11; // start sentinel value
        game_type <= game_type_in; // latch game type
        shifting <= 1'b0;
        shifting_regs <= 4'd0;
        end_notes <= 1'b0;
    end else begin
        current_notes <= next_notes;
        counter <= next_counter;
        current_addr <= next_addr;
        start_counter <= next_start_counter;
        game_type <= game_type; // only reset on start
        shifting <= |shifting_regs[3:0]; // shift high for 4 cycles, so it can
sync to 65MHz
        end_notes <= next_end_notes | end_notes;
        shifting_regs[3:0] <= {shifting_regs[2:0], next_shifting};
    end
end
endmodule
```

menu.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////
// Menu Module - Provides the state machine for all menus in the system
//   In general, the menu deals with 2-bit indices, and continuously
//   outputs the current choice.
module menu
#(
    parameter NUM_BITS = 2, // the number of bits in the choices
    parameter BOTTOM_CHOICE = 2'd0 // the lowest choice number in the menu
)
(
    input logic clk_in, // input clock
    input logic rst_in, // system reset
    input logic btn_up, // up control
    input logic btn_down, // down control
    input logic [1:0] top_choice, // the highest possible choice in the menu
    output logic [NUM_BITS - 1:0] choice // the current choice in the menu
);
    // current selection
    logic [NUM_BITS - 1:0] current_selection = BOTTOM_CHOICE;

    // state transition
    logic [NUM_BITS - 1:0] next_selection;
    always_comb begin
        case({btn_up, btn_down})
            2'b10: next_selection = (current_selection > BOTTOM_CHOICE) ?
current_selection - 1 : top_choice;
            2'b01: next_selection = (current_selection < top_choice) ?
current_selection + 1 : BOTTOM_CHOICE;
            default: next_selection = current_selection; // if they hit both
buttons or neither, don't change
        endcase
    end

    // output/state updates
    assign choice = current_selection;
    always_ff @(posedge clk_in) begin
        if(rst_in) begin
            current_selection <= BOTTOM_CHOICE;
        end else begin
            current_selection <= next_selection;
        end
    end
endmodule
```

create_song.sv

```
`timescale 1ns / 1ps

//create_song module outputs values in order to write to the song BRAM
//it takes in a 100MHz clock, an enable signal, a note_in from the keyboard,
//an 8-bit value to write to the BRAM, a write_enable signal to send to the BRAM,
and an address_out to use in writing to the BRAM
module create_song(
    input clk_100mhz,
    input enable,
    input [6:0] note_in,
    output logic [7:0] value,
    output logic write_enable,
    output logic [9:0] address_out
    );

    //this is the byte that indicates a song has just ended in the song BRAM
    parameter END_SIGNAL = 8'b01111100;

    //record a note every 0.25 seconds
    parameter CYCLES_PER_NOTE = 25000000;

    logic [24:0] counter = 25'b0;
    logic prev_enable;

    //first address available in BRAM, last address available
    parameter START_ADDRESS = 753;
    parameter MAX_ADDRESS = 997;
    //value to indicate if we should write now
    logic write_now;
    assign write_now = (counter == CYCLES_PER_NOTE - 1);

    //value to indicate if we have maxed out on memory
    logic maxed;

    //address to write to
    logic [9:0] address;
    assign maxed = (address == MAX_ADDRESS - 1);

    assign address_out = address;

    //check if user is currently playing multiple notes, which is not allowed and
will turn into a blank note in the recording
    logic bad_note;
    assign bad_note = (note_in == 7'h7E);

    always_ff @(posedge clk_100mhz) begin

        //if we have just enabled song-creation, go to START_ADDRESS, reset
counter, and set write_enable to 0
        if (enable && ~prev_enable) begin

            address <= START_ADDRESS;
            counter <= 25'b0;
            write_enable <= 0;
            value <= {1'b0, note_in};

        //if song-creation is enabled, then set write_enable high when counter
has reached CYCLES_PER_NOTE - 1;
        //increment counter, or reset to 0 when it has reached CYCLES_PER_NOTE -
1;
        //increment address when we should write a new value and we have not
maxed out on memory;
```

```
        //write the value of the note passed in, or a blank (8'h7F) when an
invalid note is passed in
        end else if (enable) begin

            write_enable <= maxed ? 0 : write_now;
            counter <= write_now ? 25'b0 : counter + 1;
            address <= (write_now & ~maxed) ? address + 1 : address;
            value <= bad_note ? {8'h7F} : {1'b0, note_in};

        //if we have just disabled song-creation, write END_SIGNAL to the next
address
        end else if (prev_enable && ~enable) begin

            write_enable <= 1;
            address <= address + 1;
            value <= END_SIGNAL;

        //otherwise, if song-creation is not enabled, set write_enable to 0
        end else begin

            write_enable <= 0;

        end

        prev_enable <= enable;

    end

endmodule
```

keyboard.sv

```
//Sound Generation for Baseline Goal

//keyboard module outputs values for PWM wave based on note and volume selection
from input switches
module keyboard(   input clk_100mhz,
                    input [9:0] sw,
                    input vauxp3,
                    input vauxn3,
                    input vn_in,
                    input vp_in,
                    input reset,
                    input enable,
                    output logic aud_pwm,
                    output logic aud_sd
    );
    parameter SAMPLE_COUNT = 763;//(will generate audio at approx 131,072 Hz
sample rate).

    logic [15:0] sample_counter;
    logic [11:0] adc_data;
    logic [11:0] sampled_adc_data;
    logic sample_trigger;
    logic adc_ready;
    logic [7:0] recorder_data;
    logic [7:0] vol_out;
    logic pwm_val; //pwm signal (HI/LO)

    assign aud_sd = 1;
    assign sample_trigger = (sample_counter == SAMPLE_COUNT);

    always_ff @(posedge clk_100mhz)begin
        if (sample_counter == SAMPLE_COUNT)begin
            sample_counter <= 16'b0;
        end else begin
            sample_counter <= sample_counter + 16'b1;
        end
        if (sample_trigger) begin
            sampled_adc_data <= {~adc_data[11],adc_data[10:0]}; //convert to
signed. incoming data is offset binary
        end
    end

    //get output data
    recorder myrec( .clk_in(clk_100mhz),.rst_in(0),
                    .ready_in(sample_trigger),
                    .data_out(recorder_data), .sw(sw));

    //volume control from switches 7-9
    volume_control vc (.vol_in(sw[9:7]),
                       .signal_in(recorder_data), .signal_out(vol_out));

    //pwm output
    pwm (.clk_in(clk_100mhz), .rst_in(reset),
.level_in({~vol_out[7],vol_out[6:0]}), .pwm_out(pwm_val));

    //only send nonzero output if keyboard is enabled
    assign aud_pwm = (pwm_val & enable) ? 1'bZ : 1'b0;

endmodule





////////////////////////////////////////////////////////////////////////////////
//
// Record/playback
//
////////////////////////////////////////////////////////////////////////////////


module recorder(
  input logic clk_in,              // 100MHz system clock
  input logic rst_in,               // 1 to reset to initial state
  input logic ready_in,             // 1 when data is available
  input logic [9:0] sw,
  output logic signed [7:0] data_out       // 8-bit PCM data to headphone
);
    logic [7:0] tone;
    logic [31:0] freq;
    //look up table for frequency corresponding to desired note
    freq_lut find_freq (.note_index(sw[6:0] - 7'd9), .clk_in(clk_in),
.freq(freq));

    //generate output sine wave at desired frequency
    sine_generator  out_tone (   .clk_in(clk_in), .rst_in(rst_in),
                                 .step_in(ready_in), .freq(freq), .sine(1'b1),
.amp_out(tone));

    always_ff @(posedge clk_in)begin
        data_out <= tone; //send tone immediately to output
    end
endmodule



//Volume Control
module volume_control (input [2:0] vol_in, input signed [7:0] signal_in, output
logic signed[7:0] signal_out);
    logic [2:0] shift;
    assign shift = 3'd7 - vol_in;
    assign signal_out = signal_in>>>shift;
endmodule

//PWM generator for audio generation!
module pwm (input clk_in, input rst_in, input [7:0] level_in, output logic
pwm_out);
    logic [7:0] count;
    assign pwm_out = count<level_in;
    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            count <= 8'b0;
        end else begin
            count <= count+8'b1;
        end
    end
endmodule
```

```
//Sine Wave Generator
module sine_generator ( input clk_in, input rst_in, //clock and reset
                        input step_in, //trigger a phase step (rate at which you
run sine generator)
                        input [31:0] freq, //desired frequency in Hz
                        input logic sine,
                        output logic [31:0] phase_stepping,
                        output logic [7:0] amp_out); //output phase

    parameter table_num = 64;
    parameter log_pulse_rate = 17;

    logic [31:0] phase_incr;

    //calculate phase increment based on desired frequency
    assign phase_incr = ((freq << (32 - log_pulse_rate)) - (freq >>
log_pulse_rate));
    assign phase_stepping = phase_incr;

    logic [7:0] divider;
    logic [31:0] phase;
    logic [7:0] amp_sine;
    logic [7:0] amp;

    assign amp = amp_sine;
    assign amp_out = {~amp[7],amp[6:0]};

    //sine lookup table to get next value
    sine_lut lut_1(.clk_in(clk_in), .phase_in(phase[31:26]), .amp_out(amp_sine));

    //increment phase, thus stepping through sine table
    always_ff @(posedge clk_in)begin
        if (rst_in)begin
```

```
            divider <= 8'b0;
            phase <= 32'b0;
        end else if (step_in)begin
            phase <= phase+phase_incr;
        end
    end
endmodule

//frequency look up table, converting note_index to frequency value
module freq_lut(input [6:0] note_index, input clk_in, output logic[31:0] freq);
    always_ff @(posedge clk_in) begin
        case(note_index)
            7'd0: freq <= 32'd27;
            7'd1: freq <= 32'd29;
            7'd2: freq <= 32'd31;
            7'd3: freq <= 32'd33;
            7'd4: freq <= 32'd35;
            7'd5: freq <= 32'd37;
            7'd6: freq <= 32'd39;
            7'd7: freq <= 32'd41;
            7'd8: freq <= 32'd44;
            7'd9: freq <= 32'd46;
            7'd10: freq <= 32'd49;
            7'd11: freq <= 32'd52;
            7'd12: freq <= 32'd55;
            7'd13: freq <= 32'd58;
            7'd14: freq <= 32'd62;
            7'd15: freq <= 32'd65;
            7'd16: freq <= 32'd69;
            7'd17: freq <= 32'd73;
            7'd18: freq <= 32'd78;
            7'd19: freq <= 32'd82;
            7'd20: freq <= 32'd87;
            7'd21: freq <= 32'd92;
            7'd22: freq <= 32'd98;
            7'd23: freq <= 32'd104;
            7'd24: freq <= 32'd110;
            7'd25: freq <= 32'd117;
            7'd26: freq <= 32'd123;
            7'd27: freq <= 32'd131;
            7'd28: freq <= 32'd139;
            7'd29: freq <= 32'd147;
            7'd30: freq <= 32'd156;
            7'd31: freq <= 32'd165;
            7'd32: freq <= 32'd175;
            7'd33: freq <= 32'd185;
            7'd34: freq <= 32'd196;
            7'd35: freq <= 32'd208;
            7'd36: freq <= 32'd220;
            7'd37: freq <= 32'd233;
            7'd38: freq <= 32'd247;
            7'd39: freq <= 32'd262;
            7'd40: freq <= 32'd277;
            7'd41: freq <= 32'd294;
            7'd42: freq <= 32'd311;
            7'd43: freq <= 32'd330;
            7'd44: freq <= 32'd349;
            7'd45: freq <= 32'd370;
            7'd46: freq <= 32'd392;
            7'd47: freq <= 32'd415;
            7'd48: freq <= 32'd440;
            7'd49: freq <= 32'd466;
            7'd50: freq <= 32'd494;
            7'd51: freq <= 32'd523;
            7'd52: freq <= 32'd554;
            7'd53: freq <= 32'd587;
            7'd54: freq <= 32'd622;
            7'd55: freq <= 32'd659;
            7'd56: freq <= 32'd698;
            7'd57: freq <= 32'd740;
            7'd58: freq <= 32'd784;
            7'd59: freq <= 32'd831;
            7'd60: freq <= 32'd880;
            7'd61: freq <= 32'd932;
            7'd62: freq <= 32'd988;
            7'd63: freq <= 32'd1046;
            7'd64: freq <= 32'd1109;
            7'd65: freq <= 32'd1175;
            7'd66: freq <= 32'd1245;
            7'd67: freq <= 32'd1319;
            7'd68: freq <= 32'd1397;
            7'd69: freq <= 32'd1480;
            7'd70: freq <= 32'd1568;
            7'd71: freq <= 32'd1661;
            7'd72: freq <= 32'd1760;
            7'd73: freq <= 32'd1865;
            7'd74: freq <= 32'd1976;
            7'd75: freq <= 32'd2093;
            7'd76: freq <= 32'd2217;
            7'd77: freq <= 32'd2349;
            7'd78: freq <= 32'd2489;
            7'd79: freq <= 32'd2637;
            7'd80: freq <= 32'd2794;
            7'd81: freq <= 32'd2960;
            7'd82: freq <= 32'd3136;
            7'd83: freq <= 32'd3322;
            7'd84: freq <= 32'd3520;
            7'd85: freq <= 32'd3729;
            7'd86: freq <= 32'd3951;
            7'd87: freq <= 32'd4186;
            7'd118: freq <= 32'd0;
            default: freq <= 32'd0;
        endcase
    end
endmodule

//6bit sine lookup, 8bit depth
module sine_lut(input[5:0] phase_in, input clk_in, output logic[7:0] amp_out);
  always_ff @(posedge clk_in)begin
    case(phase_in)
      6'd0: amp_out<=8'd128;
      6'd1: amp_out<=8'd140;
      6'd2: amp_out<=8'd152;
      6'd3: amp_out<=8'd165;
      6'd4: amp_out<=8'd176;
      6'd5: amp_out<=8'd188;
      6'd6: amp_out<=8'd198;
      6'd7: amp_out<=8'd208;
      6'd8: amp_out<=8'd218;
      6'd9: amp_out<=8'd226;
      6'd10: amp_out<=8'd234;
      6'd11: amp_out<=8'd240;
      6'd12: amp_out<=8'd245;
      6'd13: amp_out<=8'd250;
      6'd14: amp_out<=8'd253;
      6'd15: amp_out<=8'd254;
      6'd16: amp_out<=8'd255;
      6'd17: amp_out<=8'd254;
      6'd18: amp_out<=8'd253;
      6'd19: amp_out<=8'd250;
      6'd20: amp_out<=8'd245;
      6'd21: amp_out<=8'd240;
      6'd22: amp_out<=8'd234;
      6'd23: amp_out<=8'd226;
      6'd24: amp_out<=8'd218;
      6'd25: amp_out<=8'd208;
      6'd26: amp_out<=8'd198;
      6'd27: amp_out<=8'd188;
      6'd28: amp_out<=8'd176;
      6'd29: amp_out<=8'd165;
      6'd30: amp_out<=8'd152;
      6'd31: amp_out<=8'd140;
      6'd32: amp_out<=8'd128;
      6'd33: amp_out<=8'd115;
      6'd34: amp_out<=8'd103;
      6'd35: amp_out<=8'd90;
      6'd36: amp_out<=8'd79;
      6'd37: amp_out<=8'd67;
      6'd38: amp_out<=8'd57;
      6'd39: amp_out<=8'd47;
      6'd40: amp_out<=8'd37;
      6'd41: amp_out<=8'd29;
      6'd42: amp_out<=8'd21;
      6'd43: amp_out<=8'd15;
      6'd44: amp_out<=8'd10;
      6'd45: amp_out<=8'd5;
      6'd46: amp_out<=8'd2;
      6'd47: amp_out<=8'd1;
      6'd48: amp_out<=8'd0;
      6'd49: amp_out<=8'd1;
      6'd50: amp_out<=8'd2;
      6'd51: amp_out<=8'd5;
      6'd52: amp_out<=8'd10;
      6'd53: amp_out<=8'd15;
      6'd54: amp_out<=8'd21;
      6'd55: amp_out<=8'd29;
      6'd56: amp_out<=8'd37;
      6'd57: amp_out<=8'd47;
      6'd58: amp_out<=8'd57;
      6'd59: amp_out<=8'd67;
      6'd60: amp_out<=8'd79;
      6'd61: amp_out<=8'd90;
      6'd62: amp_out<=8'd103;
      6'd63: amp_out<=8'd115;
    endcase
  end
endmodule
```

UART_decoder.sv

```
`timescale 1ns / 1ps

//Module to take values from UART transmission line, output note that user is
currently playing
module UART_decoder(
    input [7:0] jb,         //input transmission line on jb[0]
    input clk_100mhz,       //100MHz clock
    input reset,            //reset
    output logic [6:0] led  //output note index, displayed on LEDs
    );

    logic [7:0] val_out;
    logic valid;
    logic [6:0] note;
    logic [87:0] notes_played;      //88 bit register, holding a 1 at every index
corresponding to a note being played, 0 otherwise

    //instantiation of MIDI_decoder, which takes in bytes from transmission line,
correctly outputs 88-bit notes_played register
    MIDI_decoder my_MIDI (.byte_in(val_out), .valid_byte(valid),
                            .clk_100mhz(clk_100mhz), .reset(reset),
                            .note_out(note), .notes_played(notes_played));

    //instantiation of key_press module, which takes in 88-bit notes_played
register, outputs a single note index
    key_press my_key_press (.clk_100mhz(clk_100mhz), .np(notes_played),
                            .note_index(led));

    logic UART_in;
    assign UART_in = jb[0];

    parameter CLK_CYCLES_PER_SECOND = 100000000;
    parameter CLK_CYCLES_PER_UART_BIT = 3200;
    parameter IDLE_SAMPLE = 200;
    parameter DELAY = 1600;

    //possible states
    parameter IDLE = 2'b00;
    parameter START_BIT = 2'b01;
    parameter SAMPLING_BITS = 2'b11;
    parameter STOP_BIT = 2'b10;

    logic [1:0] state = IDLE;
    logic [11:0] counter = 12'b0;
    logic [2:0] index;

    always_ff @(posedge clk_100mhz) begin

        if (reset) begin

            val_out <= 8'hFF;
            state <= IDLE;
            counter <= 12'b0;

        end else begin

            case (state)

                //in IDLE state, sample transmission line every 200 clock cycles,
looking for falling edge of a start bit,
                //meaning that a new message is coming in;
                //if the transmission line has gone low, switch to START_BIT
state
                IDLE: begin

                    val_out <= val_out;
                    counter <= (counter == IDLE_SAMPLE - 1) ? 12'b0 : counter +
1;
                    state <= ((counter == IDLE_SAMPLE - 1) && ~UART_in) ?
START_BIT : IDLE;
                    index <= 3'b0;
                    valid <= 0;

                end

                //in START_BIT state, wait 1600 clock cycles to get to middle of
start bit,
                //then switch to SAMPLING_BITS state
                START_BIT: begin
```

```
                    val_out <= val_out;
                    counter <= (counter == DELAY - 1) ? 12'b0 : counter + 1;
                    state <= (counter == DELAY - 1) ? SAMPLING_BITS : START_BIT;
                    index <= 3'b0;
                    valid <= 0;

                end

                //in SAMPLING_BITS state, increment index from 0 to 7 in order to
fill up the output register with the byte transmitted;
                //wait 3200 clock cycles between samplings of the transmission
line, thus grabbing each bit of the byte in the middle;
                //after getting all 8 bits, switch to STOP_BIT state
                SAMPLING_BITS: begin

                    val_out[index] = (counter == CLK_CYCLES_PER_UART_BIT - 1) ?
UART_in : val_out[index];
                    counter <= (counter == CLK_CYCLES_PER_UART_BIT - 1) ? 12'b0 :
counter + 1;
                    state <= ((counter == CLK_CYCLES_PER_UART_BIT - 1) && index
== 7) ? STOP_BIT : SAMPLING_BITS;
                    index <= (counter == CLK_CYCLES_PER_UART_BIT - 1) ? index + 1
: index;
                    valid <= 0;

                end

                //in STOP_BIT state, wait 3200 clock cycles to get to the middle
of the stop bit, then return to IDLE state,
                //waiting for the next start bit
                STOP_BIT : begin

                    val_out <= val_out;
                    counter <= (counter == CLK_CYCLES_PER_UART_BIT - 1) ? 12'b0 :
counter + 1;
                    state <= (counter == CLK_CYCLES_PER_UART_BIT - 1) ? IDLE :
STOP_BIT;
```

```
                    index <= 3'b0;
                    valid <= (counter == CLK_CYCLES_PER_UART_BIT - 1) ? 1 : 0;

                end

            endcase

        end

    end

endmodule

//MIDI_decoder takes in bytes from UART_decoder, outputes 88-bit register
notes_played filled with 1's at indices corresponding to notes
//currently being played, 0's at indices corresponding to notes not currently
being played
module MIDI_decoder(
    input [7:0] byte_in,
    input valid_byte,
    input clk_100mhz,
    input reset,
    output logic [6:0] note_out,
    output logic valid,
    output logic [87:0] notes_played
    );

    //bytes sent by keyboard indicating a note on or note off
    parameter NOTE_ON = 8'h40;
    parameter NOTE_OFF = 8'h00;

    logic [6:0] last_note;
    logic status;
    logic [6:0] last_note_out;
    assign status = (byte_in == NOTE_ON) || (byte_in == NOTE_OFF);

    always_ff @(posedge clk_100mhz) begin
```

```
        //upon reset, set all 88 bits to 0
        if (reset) begin

            last_note <= 7'h7F;
            note_out <= 7'h7F;
            valid <= 0;
            notes_played <= 88'b0;

        end else begin

            //when we've gotten a valid byte, if it's a status byte (note
on/off), then check what note is in the last_note register,
            //and set that note's bit in the 88-bit register according to whether
we just got a note on or note off signal;
            //set last_note to the note index output by the keyboard on bytes
that are not status bytes
            if (valid_byte) begin

                last_note <= status ? last_note : byte_in[6:0];
                note_out <= (byte_in == NOTE_ON) ? last_note : last_note_out;
                valid <= status;
                notes_played[last_note] <= status ? (byte_in == NOTE_ON) :
notes_played[last_note];

            //waiting for a valid byte
            end else begin

                last_note <= last_note;
                note_out <= note_out;
                valid <= 0;
                notes_played <= notes_played;

            end

        end
        last_note_out <= note_out;

    end

endmodule

//key_press module converts 88-bit register into single 7-bit note index;
//this note_index is 7'h7F if no note is played, 7'h7E if multiple notes are
being played,
//or a specific number 0 through 87 if a single note is being played
module key_press(
    input clk_100mhz,
    input [87:0] np,
    output logic [6:0] note_index
    );

    //sum is the number of notes being played
    int sum;
    assign sum = np[0] + np[1] + np[2] + np[3] + np[4] + np[5] + np[6] + np[7] +
np[8] + np[9] + np[10]
                + np[11] + np[12] + np[13] + np[14] + np[15] + np[16] + np[17] +
np[18] + np[19] + np[20] + np[21]
                + np[22] + np[23] + np[24] + np[25] + np[26] + np[27] + np[28] +
np[29] + np[30] + np[31] + np[32]
                + np[33] + np[34] + np[35] + np[36] + np[37] + np[38] + np[39] +
np[40] + np[41] + np[42] + np[43]
                + np[44] + np[45] + np[46] + np[47] + np[48] + np[49] + np[50] +
np[51] + np[52] + np[53] + np[54]
                + np[55] + np[56] + np[57] + np[58] + np[59] + np[60] + np[61] +
np[62] + np[63] + np[64] + np[65]
                + np[66] + np[67] + np[68] + np[69] + np[70] + np[71] + np[72] +
np[73] + np[74] + np[75] + np[76]
                + np[77] + np[78] + np[79] + np[80] + np[81] + np[82] + np[83] +
np[84] + np[85] + np[86] + np[87];

    always @(posedge clk_100mhz) begin
```

```
        //no note played: output 7'h7F
        if (sum == 0) begin

            note_index <= 7'h7F;

        //multiple notes being played: output 7'h7E
        end else if (sum > 1) begin

            note_index <= 7'h7E;

        //one note being played: case statement to determine which note is being
played, output note_index accordingly
        end else begin

            case (np)

                88'h0000000000000000000001 : note_index <= 7'd0;
                88'h0000000000000000000002 : note_index <= 7'd1;
                88'h0000000000000000000004 : note_index <= 7'd2;
                88'h0000000000000000000008 : note_index <= 7'd3;
                88'h0000000000000000000010 : note_index <= 7'd4;
                88'h0000000000000000000020 : note_index <= 7'd5;
                88'h0000000000000000000040 : note_index <= 7'd6;
                88'h0000000000000000000080 : note_index <= 7'd7;
                88'h0000000000000000000100 : note_index <= 7'd8;
                88'h0000000000000000000200 : note_index <= 7'd9;
                88'h0000000000000000000400 : note_index <= 7'd10;
                88'h0000000000000000000800 : note_index <= 7'd11;
                88'h0000000000000000001000 : note_index <= 7'd12;
                88'h0000000000000000002000 : note_index <= 7'd13;
                88'h0000000000000000004000 : note_index <= 7'd14;
                88'h0000000000000000008000 : note_index <= 7'd15;
                88'h0000000000000000010000 : note_index <= 7'd16;
                88'h0000000000000000020000 : note_index <= 7'd17;
                88'h0000000000000000040000 : note_index <= 7'd18;
                88'h0000000000000000080000 : note_index <= 7'd19;
                88'h0000000000000000100000 : note_index <= 7'd20;
                88'h0000000000000000200000 : note_index <= 7'd21;
                88'h0000000000000000400000 : note_index <= 7'd22;
                88'h0000000000000000800000 : note_index <= 7'd23;
                88'h0000000000000001000000 : note_index <= 7'd24;
                88'h0000000000000002000000 : note_index <= 7'd25;
                88'h0000000000000004000000 : note_index <= 7'd26;
                88'h0000000000000008000000 : note_index <= 7'd27;
                88'h0000000000000010000000 : note_index <= 7'd28;
                88'h0000000000000020000000 : note_index <= 7'd29;
                88'h0000000000000040000000 : note_index <= 7'd30;
                88'h0000000000000080000000 : note_index <= 7'd31;
                88'h0000000000000100000000 : note_index <= 7'd32;
                88'h0000000000000200000000 : note_index <= 7'd33;
                88'h0000000000000400000000 : note_index <= 7'd34;
                88'h0000000000000800000000 : note_index <= 7'd35;
                88'h0000000000001000000000 : note_index <= 7'd36;
                88'h0000000000002000000000 : note_index <= 7'd37;
                88'h0000000000004000000000 : note_index <= 7'd38;
                88'h0000000000008000000000 : note_index <= 7'd39;
                88'h0000000000010000000000 : note_index <= 7'd40;
                88'h0000000000020000000000 : note_index <= 7'd41;
                88'h0000000000040000000000 : note_index <= 7'd42;
                88'h0000000000080000000000 : note_index <= 7'd43;
                88'h0000000000100000000000 : note_index <= 7'd44;
                88'h0000000000200000000000 : note_index <= 7'd45;
                88'h0000000000400000000000 : note_index <= 7'd46;
                88'h0000000000800000000000 : note_index <= 7'd47;
                88'h0000000001000000000000 : note_index <= 7'd48;
                88'h0000000002000000000000 : note_index <= 7'd49;
                88'h0000000004000000000000 : note_index <= 7'd50;
                88'h0000000008000000000000 : note_index <= 7'd51;
                88'h0000000010000000000000 : note_index <= 7'd52;
                88'h0000000020000000000000 : note_index <= 7'd53;
                88'h0000000040000000000000 : note_index <= 7'd54;
                88'h0000000080000000000000 : note_index <= 7'd55;
                88'h0000000100000000000000 : note_index <= 7'd56;
                88'h0000000200000000000000 : note_index <= 7'd57;
```

```
            88'h0000000400000000000000 : note_index <= 7'd58;
            88'h0000000800000000000000 : note_index <= 7'd59;
            88'h0000001000000000000000 : note_index <= 7'd60;
            88'h0000002000000000000000 : note_index <= 7'd61;
            88'h0000004000000000000000 : note_index <= 7'd62;
            88'h0000008000000000000000 : note_index <= 7'd63;
            88'h0000010000000000000000 : note_index <= 7'd64;
            88'h0000020000000000000000 : note_index <= 7'd65;
            88'h0000040000000000000000 : note_index <= 7'd66;
            88'h0000080000000000000000 : note_index <= 7'd67;
            88'h0000100000000000000000 : note_index <= 7'd68;
            88'h0000200000000000000000 : note_index <= 7'd69;
            88'h0000400000000000000000 : note_index <= 7'd70;
            88'h0000800000000000000000 : note_index <= 7'd71;
            88'h0001000000000000000000 : note_index <= 7'd72;
            88'h0002000000000000000000 : note_index <= 7'd73;
            88'h0004000000000000000000 : note_index <= 7'd74;
            88'h0008000000000000000000 : note_index <= 7'd75;
            88'h0010000000000000000000 : note_index <= 7'd76;
            88'h0020000000000000000000 : note_index <= 7'd77;
            88'h0040000000000000000000 : note_index <= 7'd78;
            88'h0080000000000000000000 : note_index <= 7'd79;
            88'h0100000000000000000000 : note_index <= 7'd80;
            88'h0200000000000000000000 : note_index <= 7'd81;
            88'h0400000000000000000000 : note_index <= 7'd82;
            88'h0800000000000000000000 : note_index <= 7'd83;
            88'h1000000000000000000000 : note_index <= 7'd84;
            88'h2000000000000000000000 : note_index <= 7'd85;
            88'h4000000000000000000000 : note_index <= 7'd86;
            88'h8000000000000000000000 : note_index <= 7'd87;
            default : note_index <= 7'hFd;

        endcase

    end

end
```

```
endmodule
```

VGA_helper.sv

```
`timescale 1ns / 1ps

//pixel_helper takes as input the type of screen we are on, the current location
of a menu selector,
//a 65MHz clock, the current 5 notes for the falling display in game mode,
//a new_note signal to indicate that a new note has been shifted into the shift
register,
//a reset signal, the current note for learning mode, the current note being
played by the user,
//and then the standard VGA signals (hcount, vcount, hsync, vsync, and blank);
//it uses this information to generate the proper pixel values, outputting
phsync_out, pvsync_out, pblank_out, and pixel_out

module pixel_helper(
    input [2:0] screen,         //type of screen
    input [1:0] selection,      //menu selector location
    input clk_65mhz,            //clock
    input [34:0] notes,         //5 notes to display in game mode
    input new_note,             //pulse to indicate a new note shifted into notes
register
    input reset,                //reset
    input [6:0] learning_note,  //current learning mode note
    input [6:0] user_note,      //current note user is playing
    input [10:0] hcount_in,     // horizontal index of current pixel (0..1023)
    input [9:0]  vcount_in,     // vertical index of current pixel (0..767)
    input hsync_in,             // XVGA horizontal sync signal (active low)
    input vsync_in,             // XVGA vertical sync signal (active low)
    input blank_in,             // XVGA blanking (1 means output black pixel)

    output phsync_out,          // output horizontal sync
    output pvsync_out,          // output vertical sync
    output pblank_out,          // output blanking
    output logic [11:0] pixel_out  // output pixel  // r=11:8, g=7:4, b=3:0
    );

    //different possible screens
    parameter MAIN_MENU = 3'b000;
    parameter KEYBOARD_INSTRUCTIONS = 3'b001;
    parameter SONG_INSTRUCTIONS = 3'b010;
    parameter BASIC_SONG_MENU = 3'b011;
    parameter CUSTOM_SONG_MENU = 3'b100;
    parameter LEARN_MODE = 3'b101;
    parameter GAME_MODE = 3'b110;

    assign phsync_out = hsync_in;
    assign pvsync_out = vsync_in;
    assign pblank_out = blank_in;

    //the main menu
    wire [11:0] main_menu_pixel;
    picture_blob_main_menu
        main_menu(.pixel_clk_in(clk_65mhz),
.x_in(250),.y_in(250),.hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(main_menu_pixel));

    //the keyboard instructions
    wire [11:0] keyboard_inst_pixel;
    picture_blob_keyboard_inst
        keyboard_instructions(.pixel_clk_in(clk_65mhz),
.x_in(250),.y_in(250),.hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(keyboard_inst_pixel));

    //the song creation instructions
    wire [11:0] song_inst_pixel;
    picture_blob_song_inst
        song_instructions(.pixel_clk_in(clk_65mhz),
.x_in(250),.y_in(250),.hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(song_inst_pixel));

    //the basic song menu
    wire [11:0] song_menu_pixel;
    picture_blob_song_menu_basic
        song_menu(.pixel_clk_in(clk_65mhz),
.x_in(250),.y_in(250),.hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(song_menu_pixel));

    //the custom song menu
    wire [11:0] song_menu_custom_pixel;
    picture_blob_song_menu_custom
        song_menu_custom(.pixel_clk_in(clk_65mhz),
.x_in(250),.y_in(250),.hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(song_menu_custom_pixel));

    //the keyboard
    wire [11:0] keyboard_pixel;
    picture_blob_keyboard
        keyboard(.pixel_clk_in(clk_65mhz),
.x_in(0),.y_in(640),.hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(keyboard_pixel));

    //the menu-navigation selector
    logic [9:0] selector_y;
    selector_lut my_sel_lut(.clk_65mhz(clk_65mhz), .selector(selection),
.y_loc(selector_y));

    wire [11:0] selector_pixel;
    blob #(.WIDTH(35),.HEIGHT(35),.COLOR(12'hF00))

selector(.x_in(650),.y_in(selector_y),.hcount_in(hcount_in),.vcount_in(vcount_in)
,
            .pixel_out(selector_pixel));

    //the learning mode note indicator
    logic [10:0] learning_note_x;
    keyboard_lut my_learn_lut(.clk_65mhz(clk_65mhz), .note_index(learning_note),
.x_loc(learning_note_x));
```

```
    wire [11:0] learning_note_pixel;
    blob #(.WIDTH(10),.HEIGHT(160),.COLOR(12'h00F))

learning_note_blob(.x_in(learning_note_x),.y_in(480),.hcount_in(hcount_in),.vcoun
t_in(vcount_in),
            .pixel_out(learning_note_pixel));

    //the user note indicator
    logic [10:0] user_note_x;
    keyboard_lut my_user_lut(.clk_65mhz(clk_65mhz), .note_index(user_note),
.x_loc(user_note_x));

    wire [11:0] user_note_pixel;
    blob #(.WIDTH(10),.HEIGHT(60),.COLOR(12'h0F0))

user_note_blob(.x_in(user_note_x),.y_in(580),.hcount_in(hcount_in),.vcount_in(vco
unt_in),
            .pixel_out(user_note_pixel));

    //5 indicator rectangles for notes in falling display during game mode
    wire [11:0] note_1_pixel, note_2_pixel, note_3_pixel, note_4_pixel,
note_5_pixel;

    //use look up table of x-positions to find x-locations for 5 notes
    logic [10:0] note_1_x, note_2_x, note_3_x, note_4_x, note_5_x;
    keyboard_lut note_1_lut(.clk_65mhz(clk_65mhz), .note_index(notes[34:28]),
.x_loc(note_1_x));
    keyboard_lut note_2_lut(.clk_65mhz(clk_65mhz), .note_index(notes[27:21]),
.x_loc(note_2_x));
    keyboard_lut note_3_lut(.clk_65mhz(clk_65mhz), .note_index(notes[20:14]),
.x_loc(note_3_x));
    keyboard_lut note_4_lut(.clk_65mhz(clk_65mhz), .note_index(notes[13:7]),
.x_loc(note_4_x));
    keyboard_lut note_5_lut(.clk_65mhz(clk_65mhz), .note_index(notes[6:0]),
.x_loc(note_5_x));
```

```
    //locations for actual blob objects, only updated when a new note is shifted
in
    logic [10:0] note_1_new_x, note_2_new_x, note_3_new_x, note_4_new_x,
note_5_new_x;

    //notes' y-positions
    logic [9:0] note_1_y, note_2_y, note_3_y, note_4_y, note_5_y;

    //blob objects for notes in falling display
    blob #(.WIDTH(10),.HEIGHT(160),.COLOR(12'h00F))

note_1_blob(.x_in(note_1_x),.y_in(note_1_y),.hcount_in(hcount_in),.vcount_in(vcou
nt_in),
            .pixel_out(note_1_pixel));

    blob #(.WIDTH(10),.HEIGHT(160),.COLOR(12'h00F))

note_2_blob(.x_in(note_2_x),.y_in(note_2_y),.hcount_in(hcount_in),.vcount_in(vcou
nt_in),
            .pixel_out(note_2_pixel));

    blob #(.WIDTH(10),.HEIGHT(160),.COLOR(12'h00F))

note_3_blob(.x_in(note_3_x),.y_in(note_3_y),.hcount_in(hcount_in),.vcount_in(vcou
nt_in),
            .pixel_out(note_3_pixel));

    blob #(.WIDTH(10),.HEIGHT(160),.COLOR(12'h00F))

note_4_blob(.x_in(note_4_x),.y_in(note_4_y),.hcount_in(hcount_in),.vcount_in(vcou
nt_in),
            .pixel_out(note_4_pixel));

    blob #(.WIDTH(10),.HEIGHT(160),.COLOR(12'h00F))

note_5_blob(.x_in(note_5_x),.y_in(note_5_y),.hcount_in(hcount_in),.vcount_in(vcou
nt_in),
            .pixel_out(note_5_pixel));

    parameter CYCLES_PER_MOVEMENT = 203125;     //number of clock cycles between
downward movement of each note for falling display during game mode
    logic [17:0] counter = 18'b0;
    logic prev_new_note;

    always_ff @(posedge clk_65mhz) begin

        //determine which objects to display depending on screen type
        case(screen)

            MAIN_MENU:                pixel_out <= main_menu_pixel &
selector_pixel;
            KEYBOARD_INSTRUCTIONS:  pixel_out <= keyboard_pixel &
user_note_pixel;
            SONG_INSTRUCTIONS:      pixel_out <= song_inst_pixel;
            BASIC_SONG_MENU:        pixel_out <= song_menu_pixel &
selector_pixel;
            CUSTOM_SONG_MENU:       pixel_out <= song_menu_custom_pixel &
selector_pixel;
            LEARN_MODE:             pixel_out <= keyboard_pixel &
learning_note_pixel & user_note_pixel;
            GAME_MODE:              pixel_out <= (vcount_in >= 640) ?
keyboard_pixel :
                                        (vcount_in == 0 || vcount_in == 160 ||
vcount_in == 320 || vcount_in == 480) ?
                                        12'h000: (note_1_pixel & note_2_pixel &
note_3_pixel & note_4_pixel & note_5_pixel & user_note_pixel);

        endcase

        if (reset) begin

            counter <= 18'b0;

        end else begin
```

```
            counter <= (!(screen == GAME_MODE) || counter == CYCLES_PER_MOVEMENT
|| new_note) ? 18'b0 : counter + 1;

            //update x-positions of notes when a new note is shifted in
            note_1_new_x <= (prev_new_note) ? note_1_x : note_1_new_x;
            note_2_new_x <= (prev_new_note) ? note_2_x : note_2_new_x;
            note_3_new_x <= (prev_new_note) ? note_3_x : note_3_new_x;
            note_4_new_x <= (prev_new_note) ? note_4_x : note_4_new_x;
            note_5_new_x <= (prev_new_note) ? note_5_x : note_5_new_x;

            //set y-positions to (0, 160, 320, 480, 864=-160) when a new note is
shifted in,
            //then increment each by 2 whenever counter reaches
CYCLES_PER_MOVEMENT;
            //with a new note shifted in every quarter of a second, this produces
a smooth, synchronized falling display
            note_1_y <= (new_note) ? 480 : ((counter == CYCLES_PER_MOVEMENT) ?
note_1_y + 2 : note_1_y);
            note_2_y <= (new_note) ? 320 : ((counter == CYCLES_PER_MOVEMENT) ?
note_2_y + 2 : note_2_y);
            note_3_y <= (new_note) ? 160 : ((counter == CYCLES_PER_MOVEMENT) ?
note_3_y + 2 : note_3_y);
            note_4_y <= (new_note) ? 0 : ((counter == CYCLES_PER_MOVEMENT) ?
note_4_y + 2 : note_4_y);
            note_5_y <= (new_note) ? 864 : ((counter == CYCLES_PER_MOVEMENT) ?
note_5_y + 2 : note_5_y);

            prev_new_note <= new_note;
        end

    end


endmodule
```

```
//Look up table for menu navigation; converts current choice within a menu to the
appropriate y-location for the display
module selector_lut(
    input clk_65mhz,
    input [1:0] selector,
    output logic [9:0] y_loc);

    always_ff @(posedge clk_65mhz) begin
        case(selector)

            2'b00: y_loc <= 260;
            2'b01: y_loc <= 300;
            2'b10: y_loc <= 340;
            2'b11: y_loc <= 380;
            default: y_loc <= 260;

        endcase
    end

endmodule

//Look up table converting any note index into the appropriate x-location on the
display to match up with the keyboard image
module keyboard_lut(
    input clk_65mhz,
    input [6:0] note_index,
    output logic [10:0] x_loc);

    always_ff @(posedge clk_65mhz) begin
        case(note_index)

            36: x_loc <= 11'hA;
            37: x_loc <= 11'h1A;
            38: x_loc <= 11'h2A;
            39: x_loc <= 11'h38;
            40: x_loc <= 11'h49;
            41: x_loc <= 11'h5E;
            42: x_loc <= 11'h6F;
            43: x_loc <= 11'h7E;
            44: x_loc <= 11'h8D;
            45: x_loc <= 11'h9C;
            46: x_loc <= 11'hAA;
            47: x_loc <= 11'hBA;
            48: x_loc <= 11'hCF;
            49: x_loc <= 11'hE0;
            50: x_loc <= 11'hEF;
            51: x_loc <= 11'hFE;
            52: x_loc <= 11'h10F;
            53: x_loc <= 11'h124;
            54: x_loc <= 11'h135;
            55: x_loc <= 11'h144;
            56: x_loc <= 11'h153;
            57: x_loc <= 11'h161;
            58: x_loc <= 11'h170;
            59: x_loc <= 11'h180;
            60: x_loc <= 11'h196;
            61: x_loc <= 11'h1A7;
            62: x_loc <= 11'h1B6;
            63: x_loc <= 11'h1C5;
            64: x_loc <= 11'h1D6;
            65: x_loc <= 11'h1EB;
            66: x_loc <= 11'h1FD;
            67: x_loc <= 11'h20D;
            68: x_loc <= 11'h21A;
            69: x_loc <= 11'h22A;
            70: x_loc <= 11'h238;
            71: x_loc <= 11'h24C;
            72: x_loc <= 11'h262;
            73: x_loc <= 11'h274;
            74: x_loc <= 11'h27F;
            75: x_loc <= 11'h28D;
            76: x_loc <= 11'h29E;
            77: x_loc <= 11'h2B7;
            78: x_loc <= 11'h2C8;
            79: x_loc <= 11'h2D7;
            80: x_loc <= 11'h2E6;
            81: x_loc <= 11'h2F3;
            82: x_loc <= 11'h300;
            83: x_loc <= 11'h312;
            84: x_loc <= 11'h32A;
            85: x_loc <= 11'h33A;
            86: x_loc <= 11'h349;
            87: x_loc <= 11'h358;
            88: x_loc <= 11'h368;
            89: x_loc <= 11'h37A;
            90: x_loc <= 11'h38C;
            91: x_loc <= 11'h39B;
            92: x_loc <= 11'h3AA;
            93: x_loc <= 11'h3B9;
            94: x_loc <= 11'h3C8;
            95: x_loc <= 11'h3D8;
            96: x_loc <= 11'h3EB;
            default: x_loc <= 11'h7FF;      //default off-screen for invalid note
indices

        endcase
    end

endmodule


////////////////////////////////////////////////////////////////////////
//
// blob: generate rectangle on screen
//
////////////////////////////////////////////////////////////////////////
module blob
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               COLOR = 12'hFFF)  // default color: white
   (input [11:0] x_in,hcount_in,
    input [10:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   always_comb begin
      if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
     (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
    pixel_out = COLOR;
      else pixel_out = 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////
//
// picture_blob: display a picture for the main menu
//
//////////////////////////////////////////////////
module picture_blob_main_menu
   #(parameter WIDTH = 226,     // default picture width
               HEIGHT = 158)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   main_menu_rom  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   red_main_menu_rom rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
```

```
   green_main_menu_rom gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
   blue_main_menu_rom bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; //
greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////
//
// picture_blob: display a picture for the keyboard-playing instructions
//
//////////////////////////////////////////////////
module picture_blob_keyboard_inst
   #(parameter WIDTH = 173,     // default picture width
               HEIGHT = 50)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [13:0] image_addr;
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   keyboard_inst_rom  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));
```

```
   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   red_key_inst_rom rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
   green_key_inst_rom gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
   blue_key_inst_rom bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; //
greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////
//
// picture_blob: display a picture for the song-creation instructions
//
//////////////////////////////////////////////////
module picture_blob_song_inst
   #(parameter WIDTH = 217,     // default picture width
               HEIGHT = 56)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [13:0] image_addr;
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;
   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   song_inst_rom  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   red_song_inst_rom rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
   green_song_inst_rom gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
   blue_song_inst_rom bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; //
greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////
//
// picture_blob: display a picture for the keyboard
//
//////////////////////////////////////////////////
module picture_blob_keyboard
   #(parameter WIDTH = 1026,     // default picture width
               HEIGHT = 128)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [17:0] image_addr;
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   keyboard_rom1  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   red_keyboard_rom rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
   green_keyboard_rom gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
   blue_keyboard_rom bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; //
greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////
//
// picture_blob: display a picture for the basic song menu
//
//////////////////////////////////////////////////
module picture_blob_song_menu_basic
```

```
   #(parameter WIDTH = 330,     // default picture width
               HEIGHT = 123)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   basic_song_menu_rom  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   red_basic_menu_rom rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
   green_basic_menu_rom gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
   blue_basic_menu_rom bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; //
greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////
```

```
//
// picture_blob: display a picture for the custom song menu
//
//////////////////////////////////////////////////
module picture_blob_song_menu_custom
   #(parameter WIDTH = 329,     // default picture width
               HEIGHT = 157)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   custom_song_menu_rom  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   red_custom_menu_rom rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
   green_custom_menu_rom gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
   blue_custom_menu_rom bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; //
greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= 12'hFFF;
   end
endmodule

////////////////////////////////////////////////////////////////////////////////
/
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                                ---- HORIZONTAL -----     ------VERTICAL -----
//                                Active                    Active
//                      Freq      Video   FP  Sync   BP      Video   FP  Sync   BP
//   640x480, 60Hz    25.175    640     16    96   48        480    11   2     31
//   800x600, 60Hz    40.000    800     40   128   88        600     1   4     23
//   1024x768, 60Hz   65.000    1024    24   136  160        768     3   6     29
//   1280x1024, 60Hz  108.00    1280    48   112  248        768     1   3     38
//   1280x720p 60Hz   75.25     1280    72    80  216        720     3   5     30
//   1920x1080 60Hz   148.5     1920    88    44  148       1080     4   5     36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock_in,
            output reg [10:0] hcount_out,    // pixel number on current line
            output reg [9:0] vcount_out,     // line number
            output reg vsync_out, hsync_out,
            output reg blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));
// 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP -
1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE
+ V_BP - 1)); // 805

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end

endmodule
```

fft_analyzer.sv

```
`timescale 1ns / 1ps
///////////////////////////////////////////////////////////////
// FFT Sampler Module - Performs an averaging sweep over the FFT input
// and outputs the largest index from the resulting signal.
module fft_analyzer(
    input logic clk_104mhz, // input clock signal, at 104 MHz
    input logic vauxp3, // inputs from Nexys4 for ADC
    input logic vauxn3,
    output logic hi, // whether or not the FFT represents a high input note
    output logic lo  // whether or not the FFT represents a low input note
);

// How often to take the FFT, equates to 60Hz = 104Mhz/PULSE_COUNT
localparam PULSE_COUNT = 21'd1_733_333;

// Generate a 60Hz pulse (fft_pulse)
logic [20:0] pulse_counter = 21'd0;
logic fft_pulse; // 60Hz pulse
assign fft_pulse = (pulse_counter == PULSE_COUNT - 21'd1) ? 1'b1 : 1'b0;
always_ff @(posedge clk_104mhz) begin
    pulse_counter <= (pulse_counter == PULSE_COUNT - 21'd1) ? 21'd0 :
pulse_counter + 21'd1;
end

// ADC Conversion of Microphone
wire [15:0] sample_reg;
wire eoc, xadc_reset;
mic_adc mic_adc_0 (
    .dclk_in(clk_104mhz),  // Master clock for DRP and XADC.
    .di_in(0),             // DRP input info (0 becuase we don't need to write)
    .daddr_in(6'h13),      // The DRP register address for the third analog input
register
    .den_in(1),            // DRP enable line high (we want to read)
    .dwe_in(0),            // DRP write enable low (never write)
    .drdy_out(),           // DRP ready signal (unused)
    .do_out(sample_reg),   // DRP output from register (the ADC data)
    .reset_in(xadc_reset), // reset line
    .vp_in(0),             // dedicated/built in analog channel on bank 0
    .vn_in(0),             // can't use this analog channel b/c of nexys 4 setup
    .vauxp3(vauxp3),         // The third analog auxiliary input channel
    .vauxn3(vauxn3),         // Choose this one b/c it's on JXADC header 1
    .channel_out(),        // Not useful in sngle channel mode
    .eoc_out(eoc),         // Pulses high on end of ADC conversion
    .alarm_out(),          // Not useful
    .eos_out(),            // End of sequence pulse, not useful
    .busy_out()            // High when conversion is in progress. unused.
);
assign xadc_reset = 1'b0;

// Oversample this data, sums 16 input samples at a time and outputs them divided
by 4
wire [13:0] osample16;
wire done_osample16;
oversample16 oversampler(
    .clk(clk_104mhz),
    .sample(sample_reg[15:4]),
    .eoc(eoc),
    .oversample(osample16),
    .done(done_osample16));


// Create the frame BRAM that frames of digital audio are written to
wire fwe;
reg [11:0] fhead = 0; // Frame head - a pointer to the write point, works as
circular buffer
```

```
wire [15:0] fsample;  // The sample data from the XADC, oversampled 15x
wire [11:0] faddr;    // Frame address - The read address, controlled by
bram_to_fft
wire [15:0] fdata;    // Frame data - The read data, input into bram_to_fft
frame_bram bram1 (
    .clka(clk_104mhz),
    .wea(fwe),
    .addra(fhead),
    .dina(fsample),
    .clkb(clk_104mhz),
    .addrb(faddr),
    .doutb(fdata));
always @(posedge clk_104mhz) if (done_osample16) fhead <= fhead + 1; // Move the
pointer every oversample
assign fsample = {osample16, 2'b0}; // Pad the oversample with zeros to pretend
it's 16 bits
assign fwe = done_osample16; // Write only when we finish an oversample (every
104*16 clock cycles)

// Read frame data at a chosen rate and send it to FFT
wire last_missing; // All these are control lines to the FFT block design
wire [31:0] frame_tdata;
wire frame_tlast, frame_tready, frame_tvalid;
bram_to_fft bram_to_fft_0 (
    .clk(clk_104mhz),
    .head(fhead),
    .addr(faddr),
    .data(fdata),
    .start(fft_pulse),
    .last_missing(last_missing),
    .frame_tdata(frame_tdata),
    .frame_tlast(frame_tlast),
    .frame_tready(frame_tready),
    .frame_tvalid(frame_tvalid)
);

// Perform the FFT!
```

```
wire [23:0] magnitude_tdata; // This output bus has the FFT magnitude for the
current index
wire [11:0] magnitude_tuser; // This represents the current index being output,
from 0 to 4096
wire magnitude_tlast, magnitude_tvalid;
fft_mag fft_mag_i(
    .clk(clk_104mhz),
    .event_tlast_missing(last_missing),
    .frame_tdata(frame_tdata),
    .frame_tlast(frame_tlast),
    .frame_tready(frame_tready),
    .frame_tvalid(frame_tvalid),
    .scaling(12'b1100_0001_1101),
    .magnitude_tdata(magnitude_tdata),
    .magnitude_tlast(magnitude_tlast),
    .magnitude_tuser(magnitude_tuser),
    .magnitude_tvalid(magnitude_tvalid));

// Write the FFT to BRAM
wire in_range = ~|magnitude_tuser[11:10]; // When 13 and 12 are 0, we're on
indexes 0 to 1023
wire [9:0] haddr; // The read port address
wire [15:0] hdata; // The read port data
fft_bram bram2 (
    .clka(clk_104mhz),
    .wea(in_range & magnitude_tvalid),  // Only save FFT output if in range and
output is valid
    .addra(magnitude_tuser[9:0]),       // The FFT output index, 0 to 1023
    .dina(magnitude_tdata[15:0]),       // The actual FFT magnitude
    .clkb(clk_104mhz),  // input wire clkb used to be clk_65mhz
    .addrb(haddr),     // input wire [9 : 0] addrb
    .doutb(hdata)      // output wire [15 : 0] doutb
);

// Read from the FFT BRAM (2) and find the largest bucket. We can use this to
// determine whether a high or low note is being inputted.
wire [10:0] cur_big_index;
wire index_calc;
wire ignore;
fft_sampler sampler(
    .clk(clk_104mhz),
    .start((magnitude_tuser[9:0] == 10'd1023)),
    .cur_fft(hdata),
    .read_addr(haddr),
    .read_enable(ignore),
    .largest_bucket(cur_big_index),
    .done(index_calc));

// FFT sensitivity timer - the speed at which we sense input notes is
// rate-limited by the timer. The timer serves as a time-out between input
// notes.
// This design stops the input data from being processed at 100Mhz, much
// faster than humans can respond.
logic timer_start, timer_done;
timer fft_timer (
    .clk_104mhz(clk_104mhz),
    .start_timer(timer_start),
    .value(4'd1),
    .expired(timer_done)
);

// State machine to process and analyze input notes
------------------------------------

// PARAMETERS ------------------
// States
localparam STATE_TRACKING = 1'b1;
localparam STATE_WAITING = 1'b0;
// Parameters to tune FFT sensitivity
localparam HILO_THRESHOLD = 11'h45; // The bucket that demarcates a high vs. low
note
localparam HILO_CYCLE_COUNT = 7'd45; // How long a note has to be held before
being acknowledged, in 60Hz
// Noise state (hi, lo, or none)
```

```
localparam HI_NOISE = 2'b10;
localparam LO_NOISE = 2'b01;
localparam NO_NOISE = 2'b00;

// STATE --------------------
logic state = STATE_TRACKING;
logic [1:0] noise_state = NO_NOISE;
logic [6:0] hilo_cycles=7'd0;

// STATE TRANSITIONS ------------------
logic next_state;
logic [1:0] next_noise_state;
logic [6:0] next_hilo_cycles;
always_comb begin
    case(state)
    STATE_WAITING: begin // time-out, waiting for timer to expire
        next_state = timer_done ? STATE_TRACKING : STATE_WAITING;
        next_noise_state = noise_state; // just hold the values steady
        next_hilo_cycles = 7'd0;
    end
    STATE_TRACKING: begin
        if(hilo_cycles == HILO_CYCLE_COUNT) begin // found a new note, set timer
and output
            next_state = STATE_WAITING;
            next_noise_state = noise_state;
            next_hilo_cycles = 7'd0;
        end else begin
            if(index_calc) begin // only update each time the fft is recalculated
                next_state = STATE_TRACKING;
                next_noise_state = (cur_big_index == 11'd0) ? NO_NOISE :
                    ((cur_big_index > HILO_THRESHOLD)? HI_NOISE : LO_NOISE);
                next_hilo_cycles = (noise_state == next_noise_state) ?
hilo_cycles + 7'd1 : 7'd0;
            end else begin // wait for the next FFT input (60Hz)
                next_state = state;
                next_noise_state = noise_state;
                next_hilo_cycles = hilo_cycles;
            end
        end
    end
    endcase
end

// STATE UPDATES --------------------
assign timer_start = (hilo_cycles == HILO_CYCLE_COUNT) ? 1'b1 : 1'b0; // start
timeout if we have a new note
assign hi = (state == STATE_WAITING) ? noise_state[1] : 1'b0; // only output when
counting
assign lo = (state == STATE_WAITING) ? noise_state[0] : 1'b0;
always_ff @(posedge clk_104mhz) begin
    state <= next_state;
    noise_state <= next_noise_state;
    hilo_cycles <= next_hilo_cycles;
end
endmodule

///////////////////////////////////////////////////////////////
// FFT Oversampler - Oversamples the FFT at 16x (sliding window sum with width
// 16), and then divides by 4 to keep small bit sizes.
module oversample16(
    input wire clk, // input clock
    input wire [11:0] sample, // current sample of the FFT
    input wire eoc, // indicates whether a valid value is on the input
    output reg [13:0] oversample, // the output oversampled window value
    output reg done // whether the current accumulated value is calculated.
);
// STATE
reg [3:0] counter = 0;
reg [15:0] accumulator = 0;

// STATE TRANSITIONS
always @(posedge clk) begin
    done <= 0;
    if (eoc) begin
```

```
        // Conversion has ended and we can read a new sample
        if (&counter) begin // If counter is full (16 accumulated)
            // Get final total, divide by 4 with (very limited) rounding.
            oversample <= (accumulator + sample + 2'b10) >> 2;
            done <= 1;
            // Reset accumulator
            accumulator <= 0;
        end
        else begin
            // Else add to accumulator as usual
            accumulator <= accumulator + sample;
            done <= 0;
        end
        counter <= counter + 1;
    end
end
endmodule

////////////////////////////////////////////////////////////////
// BRAM to FFT - Handles the transfer of the data fram from the BRAM
//   to the FFT, at the appropriate speed and addresses. Performs a Hann
//   filter on the frame being outputted in order to clean up the
//   resulting FFT.
module bram_to_fft(
    input wire clk, // input clock
    input wire [11:0] head, // the head of the new frame
    output reg [11:0] addr, // the address to be read from the BRAM
    input wire [15:0] data, // the data being read
    input wire start, // a pulse to begin sending data through
    input wire last_missing, // a signal from the FFT module to align properly
    output reg [31:0] frame_tdata, // the output data for the FFT
    output reg frame_tlast, // signal to indicate the final note of the frame
    input wire frame_tready, // whether the FFT module is ready for the next
sample
    output reg frame_tvalid  // a signal that the next sample is ready
);
```

```
// Get a signed version of the sample by subtracting half the max
wire signed [15:0] data_signed = {1'b0, data} - (1 << 15);

// SENDING LOGIC
// Once our oversampling is done,
// Start at the frame bram head and send all 4096 buckets of bram.
// Hopefully every time this happens, the FFT core is ready
reg sending = 0;
reg [11:0] send_count = 0;

// windowing coefficient
wire [23:0] hann_coeff; // Hann coefficient, read from the LUT
hann my_hann(.n(send_count), .coeff(hann_coeff));

wire signed [40:0] windowed_data; // A scaled version of the data using the Hann
window
assign windowed_data = hann_coeff * data_signed;

always @(posedge clk) begin
    frame_tvalid <= 0; // Normally do not send
    frame_tlast <= 0; // Normally not the end of a frame
    if (!sending) begin
        if (start) begin // When a new sample shifts in
            addr <= head; // Start reading at the new head
            send_count <= 0; // Reset send_count
            sending <= 1; // Advance to next state
        end
    end
    else begin
        if (last_missing) begin
            // If core thought the frame ended
            sending <= 0; // reset to state 0
        end
        else begin
            frame_tdata <= {16'b0, (windowed_data >> 24)}; // outputs the scaled
data value
            frame_tvalid <= 1; // Signal to fft a sample is ready
            if (frame_tready) begin // If the fft module was ready
                addr <= addr + 1; // Switch to read next sample
                send_count <= send_count + 1; // increment send_count
            end
            if (&send_count) begin
                // We're at last sample
                frame_tlast <= 1; // Tell the core
                if (frame_tready) sending <= 0; // Reset to state 0
            end
        end
    end
end
endmodule
```

fft_sampler.sv

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
// FFT Sampler Module - Performs an averaging sweep over the FFT input
// and outputs the largest index from the resulting signal.
//  - Takes in the FFT coefficient as input
//  - Outputs the largest bucket (and a signal to indicate completion of the
//  calculation)
module fft_sampler(
  input logic clk, // input clock signal
  input logic start, // start system pulse
  input logic [15:0] cur_fft, // current FFT magnitude
  output logic [9:0] read_addr, // address to read FFT magnitude from (in FFT
BRAM)
  output logic read_enable, // read enable signal to FFT BRAM
  output logic [10:0] largest_bucket, // the index of the largest FFT coefficient
  output logic done // pulse to indicate that processing is complete
);

// PARAMETERS ----------------------------
localparam IDLE = 2'd0;
localparam SWEEPING = 2'd1;
```

```
// Sweep Indices (audible range in input FFT)
localparam SWEEP_START = 11'd8;
localparam SWEEP_END = 11'd300; // used to be 400

// FFT Tuning Parameters
localparam NUM_BUCKETS = 11'd1024; // The total number of buckets in the FFT
localparam WINDOW_LEN = 2; // size of window for averaging filter
localparam SUM_LOW_THRESHOLD = 21'd100; // Minimum FFT sum magnitude

// STATE --------------------------------
logic [1:0] state = IDLE;
logic [10:0] counter = 11'd0;
logic [15:0] window [WINDOW_LEN-1:0];
logic [20:0] sum = 21'd0;
logic [20:0] largest_sum = 21'd0;
logic [10:0] largest_index = 11'd0;
logic done_state = 1'b0;

// STATE TRANSITIONS --------------------------
logic [1:0] next_state;
logic [10:0] next_counter;
logic [20:0] next_sum;
logic [20:0] next_largest_sum;
logic [10:0] next_largest_index;
logic next_done_state;

always_comb begin
    case(state)
        IDLE: begin // stay in IDLE state
            next_state = IDLE;
            next_counter = SWEEP_START;
            next_sum = 21'd0;
            next_largest_sum = largest_sum;
            next_largest_index = largest_index; // just hold values
            next_done_state = 1'b0;
        end
        SWEEPING: begin
```

```
            // if the sweep is complete, switch back to IDLE
            if(counter == SWEEP_END) begin
                next_state = IDLE;
                next_counter = SWEEP_START;
                next_done_state = 1'b1;
            end else begin
                next_state = SWEEPING;
                next_counter = counter + 11'd1;
                next_done_state = 1'b0;
            end

            // sum and largest sum update
            next_sum = sum - window[WINDOW_LEN-1] + cur_fft;
            // next_sum holds the sum of the window ending on counter:
            //   - (counter- WINDOW_LEN, counter]
            next_largest_sum = (next_sum > largest_sum) ? next_sum : largest_sum;
            next_largest_index = (next_sum > largest_sum) ? counter :
largest_index;
        end
        default: begin // just go to IDLE
            next_state = IDLE;
            next_counter = SWEEP_START;
            next_sum = 21'd0;
            next_largest_sum = largest_sum;
            next_largest_index = largest_index;
            next_done_state = 1'b0;
        end
    endcase
end

// OUTPUT/STATE UPDATES --------------------------------
assign read_addr = counter[9:0]; // the current FFT index we are considering
assign read_enable = (state == SWEEPING); // whether or not to read from the BRAM
assign largest_bucket = (largest_sum > SUM_LOW_THRESHOLD) ? largest_index :
11'd0;
assign done = done_state;
always_ff @(posedge clk) begin
```

```
    if(start) begin // reset to initial values and enter SWEEPING mode
        state <= SWEEPING;
        // zero the starting points
        counter <= SWEEP_START;
        sum <= 21'd0;
        largest_sum <= 21'd0;
        largest_index <= 11'd0; // not a valid index
        // zero the window
        window[1] <= 16'd0;
        window[0] <= 16'd0;
        done_state <= 1'b0;
    end else begin
        state <= next_state;
        counter <= next_counter;
        sum <= next_sum;
        largest_sum <= next_largest_sum;
        largest_index <= next_largest_index;
        // slide the window over
        window[1] <= window[0];
        window[0] <= cur_fft;
        done_state <= next_done_state;
    end
end
endmodule
```

timer.sv

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
/
// Timer Module - Starts on a pulse, and counts down an input length
module timer #(parameter COUNTER_HI=26'd52_000_000) (
    input clk_104mhz, // input clock, at 104 MHz
    input start_timer, // a pulse to start the countdown
    input [3:0] value, // how long to countdown for
    output logic expired, // a pulse indicating whether the timer has expired
    output [3:0] countdown_out // the current countdown position
);
```

```
// fields to convert the input clock to a 2 hz signal
localparam COUNTER_LO = 26'd0;
logic one_hz_enable;
logic [25:0] counter;

// fields perform the countdown
localparam IDLE = 1'b0;
localparam COUNTING = 1'b1;
logic [3:0] countdown;
logic state;

assign countdown_out = countdown;
always_ff @(posedge clk_104mhz) begin
    if(start_timer) begin
        // reset
        counter <= COUNTER_LO;
        one_hz_enable <= 1'b0;
        expired <= 1'b0;
        // will latch onto value when [start_timer] is asserted,
        // and only reread this value when [start_timer] is next asserted
        countdown <= value;
        state <= COUNTING;
    end else begin
        // one hz signal state updates
        if(counter < COUNTER_HI - 26'd1) one_hz_enable <= 1'b0;
        else one_hz_enable <= 1'b1;

        if(counter == COUNTER_HI - 26'd1) counter <= COUNTER_LO;
        else counter <= counter + 26'd1;

        // countdown logic
        case(state)
          IDLE: begin
              expired <= 1'b0;
              countdown <= 4'b0;
              state <= IDLE;
          end
```

```
              COUNTING: begin
                  if(countdown == 4'b0) begin // handle the 0 value edge case
                      countdown <= 4'b0;
                      expired <= 1'b1;
                      state <= IDLE;
                  end
                  else begin
                    countdown <= (one_hz_enable) ? countdown - 4'b1 : countdown;
                    // check here to avoid a one cycle delay between countdown
and output
                    expired <= (one_hz_enable) ? ((countdown == 4'b1) ? 1'b1 :
1'b0) : expired;
                    state <= (one_hz_enable) ? ((countdown == 4'b1) ? IDLE :
COUNTING) : state;
                  end
              end
              default: state <= IDLE; // revert from invalid state to IDLE
            endcase
        end
    end
endmodule
```

hann.sv

See the GitHub repo (https://github.com/rahulyesantharao/FPGA/blob/master/hann.sv) for this document, as it is extremely large (4096 element lookup table). To see how it is generated, see hann_gen.py in Appendix B.

nexys4ddr_audio.xdc

```
## all inputs/outputs changed to lowercase; arrays start with zero. 2019-08-25
## system clock renamed to clk_100mhz
## ja, jb, jc, jd renamed to 0-7
## xa port renamed 0-3

## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top
level signal names in the project

## Clock signal
```

```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports {
clk_100mhz }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{clk_100mhz}];


##Switches

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { sw[0]
}]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { sw[1]
}]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { sw[2]
}]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { sw[3]
}]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { sw[4]
}]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { sw[5]
}]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { sw[6]
}]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { sw[7]
}]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { sw[8]
}]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { sw[9]
}]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { sw[10]
}]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { sw[11]
}]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { sw[12]
}]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { sw[13]
}]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { sw[14]
}]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { sw[15]
}]; #IO_L21P_T3_DQS_14 Sch=sw[15]


## LEDs
```

```
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { led[0]
}]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { led[1]
}]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { led[2]
}]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { led[3]
}]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { led[4]
}]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { led[5]
}]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { led[6]
}]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { led[7]
}]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { led[8]
}]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { led[9]
}]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { led[10]
}]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { led[11]
}]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { led[12]
}]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { led[13]
}]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { led[14]
}]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { led[15]
}]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 } [get_ports { led16_b
}]; #IO_L5P_T0_D06_14 Sch=led16_b
set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports { led16_g
}]; #IO_L10P_T1_D14_14 Sch=led16_g
set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { led16_r
}]; #IO_L11P_T1_SRCC_14 Sch=led16_r
set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led17_b
}]; #IO_L15N_T2_DQS_ADV_B_15 Sch=led17_b
set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { led17_g
}]; #IO_0_14 Sch=led17_g
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { led17_r
}]; #IO_L11N_T1_SRCC_14 Sch=led17_r
```

```
##7 segment display

set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { ca }];
#IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { cb }];
#IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { cc }];
#IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { cd }];
#IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { ce }];
#IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { cf }];
#IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { cg }];
#IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { dp }];
#IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { an[0]
}]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { an[1]
}]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { an[2]
}]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { an[3]
}]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { an[4]
}]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { an[5]
}]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { an[6]
}]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { an[7]
}]; #IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
cpu_resetn }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
```

```
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { btnc
}]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { btnu
}]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { btnl
}]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { btnr
}]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { btnd
}]; #IO_L9N_T1_DQS_D13_14 Sch=btnd


##Pmod Headers


##Pmod Header JA

#set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports { ja[0]
}]; #IO_L20N_T3_A19_15 Sch=ja[1]
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { ja[1]
}]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
#set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports { ja[2]
}]; #IO_L21P_T3_DQS_15 Sch=ja[3]
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { ja[3]
}]; #IO_L18N_T2_A23_15 Sch=ja[4]
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { ja[4]
}]; #IO_L16N_T2_A27_15 Sch=ja[7]
#set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { ja[5]
}]; #IO_L16P_T2_A28_15 Sch=ja[8]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports { ja[6]
}]; #IO_L22N_T3_A16_15 Sch=ja[9]
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports { ja[7]
}]; #IO_L22P_T3_A17_15 Sch=ja[10]


##Pmod Header JB

set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports { jb[0]
}]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { jb[1]
}]; #IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports { jb[2]
}]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports { jbclk
}]; #IO_L15P_T2_DQS_15 Sch=jb[4]
```

```
set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports { jb[4]
}]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports { jb[5]
}]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports { jb[6]
}]; #IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { jb[7]
}]; #IO_L13P_T2_MRCC_15 Sch=jb[10]


##Pmod Header JC

#set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 } [get_ports { jc[0]
}]; #IO_L23N_T3_35 Sch=jc[1]
#set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 } [get_ports { jc[1]
}]; #IO_L19N_T3_VREF_35 Sch=jc[2]
#set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 } [get_ports { jc[2]
}]; #IO_L22N_T3_35 Sch=jc[3]
#set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 } [get_ports { jc[3]
}]; #IO_L19P_T3_35 Sch=jc[4]
#set_property -dict { PACKAGE_PIN E7    IOSTANDARD LVCMOS33 } [get_ports { jc[4]
}]; #IO_L6P_T0_35 Sch=jc[7]
#set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { jc[5]
}]; #IO_L22P_T3_35 Sch=jc[8]
#set_property -dict { PACKAGE_PIN J4    IOSTANDARD LVCMOS33 } [get_ports { jc[6]
}]; #IO_L21P_T3_DQS_35 Sch=jc[9]
#set_property -dict { PACKAGE_PIN E6    IOSTANDARD LVCMOS33 } [get_ports { jc[7]
}]; #IO_L5P_T0_AD13P_35 Sch=jc[10]


##Pmod Header JD

#set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 } [get_ports { jd[0]
}]; #IO_L21N_T3_DQS_35 Sch=jd[1]
#set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 } [get_ports { jd[1]
}]; #IO_L17P_T2_35 Sch=jd[2]
#set_property -dict { PACKAGE_PIN G1    IOSTANDARD LVCMOS33 } [get_ports { jd[2]
}]; #IO_L17N_T2_35 Sch=jd[3]
#set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 } [get_ports { jd[3]
}]; #IO_L20N_T3_35 Sch=jd[4]
#set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 } [get_ports { jd[4]
}]; #IO_L15P_T2_DQS_35 Sch=jd[7]
#set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 } [get_ports { jd[5]
}]; #IO_L20P_T3_35 Sch=jd[8]
```

```
#set_property -dict { PACKAGE_PIN G2    IOSTANDARD LVCMOS33 } [get_ports { jd[6]
}]; #IO_L15N_T2_DQS_35 Sch=jd[9]
#set_property -dict { PACKAGE_PIN F3    IOSTANDARD LVCMOS33 } [get_ports { jd[7]
}]; #IO_L13N_T2_MRCC_35 Sch=jd[10]


#Pmod Header JXADC
#Bank = 15, Pin name = IO_L9P_T1_DQS_AD3P_15,                        Sch name =
XADC1_P -> XA1_P
#set_property PACKAGE_PIN A13 [get_ports {vauxp3}]                   set_property
IOSTANDARD LVCMOS33 [get_ports {vauxp3}];
#Bank = 15, Pin name = IO_L8P_T1_AD10P_15,                           Sch name =
XADC2_P -> XA2_P
#set_property PACKAGE_PIN A15 [get_ports {vauxp10}]                  set_property
IOSTANDARD LVCMOS33 [get_ports {vauxp10}]
#Bank = 15, Pin name = IO_L7P_T1_AD2P_15,                            Sch name =
XADC3_P -> XA3_P
#set_property PACKAGE_PIN B16 [get_ports {vauxp2}]                   set_property
IOSTANDARD LVCMOS33 [get_ports {vauxp2}]
#Bank = 15, Pin name = IO_L10P_T1_AD11P_15,                          Sch name =
XADC4_P -> XA4_P
#set_property PACKAGE_PIN B18 [get_ports {vauxp11}]                set_property
IOSTANDARD LVCMOS33 [get_ports {vauxp11}]
#Bank = 15, Pin name = IO_L9N_T1_DQS_AD3N_15,                        Sch name =
XADC1_N -> XA1_N
#set_property PACKAGE_PIN A14 [get_ports {vauxn3}]                   set_property
IOSTANDARD LVCMOS33 [get_ports {vauxn3}];
#Bank = 15, Pin name = IO_L8N_T1_AD10N_15,                           Sch name =
XADC2_N -> XA2_N
#set_property PACKAGE_PIN A16 [get_ports {vauxn10}]                  set_property
IOSTANDARD LVCMOS33 [get_ports {vauxn10}]
#Bank = 15, Pin name = IO_L7N_T1_AD2N_15,                            Sch name =
XADC3_N -> XA3_N
#set_property PACKAGE_PIN B17 [get_ports {vauxn2}]                   set_property
IOSTANDARD LVCMOS33 [get_ports {vauxn2}]
#Bank = 15, Pin name = IO_L10N_T1_AD11N_15,                          Sch name =
XADC4_N -> XA4_N
#set_property PACKAGE_PIN A18 [get_ports {vauxn11}]                  set_property
IOSTANDARD LVCMOS33 [get_ports {vauxn11}]

##Pmod Header JXADC

set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33     } [get_ports {
vauxn3 }]; #IO_L9N_T1_DQS_AD3N_15 Sch=xa_n[1]
```

```
set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVCMOS33     } [get_ports {
vauxp3 }]; #IO_L9P_T1_DQS_AD3P_15 Sch=xa_p[1]
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVDS     } [get_ports {
xa_n[1] }]; #IO_L8N_T1_AD10N_15 Sch=xa_n[2]
#set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVDS     } [get_ports {
xa_p[1] }]; #IO_L8P_T1_AD10P_15 Sch=xa_p[2]
#set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVDS     } [get_ports {
xa_n[2] }]; #IO_L7N_T1_AD2N_15 Sch=xa_n[3]
#set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVDS     } [get_ports {
xa_p[2] }]; #IO_L7P_T1_AD2P_15 Sch=xa_p[3]
#set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVDS     } [get_ports {
xa_n[3] }]; #IO_L10N_T1_AD11N_15 Sch=xa_n[4]
#set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVDS     } [get_ports {
xa_p[3] }]; #IO_L10P_T1_AD11P_15 Sch=xa_p[4]


##VGA Connector

set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports {
vga_r[0] }]; #IO_L8N_T1_AD14N_35 Sch=vga_r[0]
set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 } [get_ports {
vga_r[1] }]; #IO_L7N_T1_AD6N_35 Sch=vga_r[1]
set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 } [get_ports {
vga_r[2] }]; #IO_L1N_T0_AD4N_35 Sch=vga_r[2]
set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 } [get_ports {
vga_r[3] }]; #IO_L8P_T1_AD14P_35 Sch=vga_r[3]

set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 } [get_ports {
vga_g[0] }]; #IO_L1P_T0_AD4P_35 Sch=vga_g[0]
set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 } [get_ports {
vga_g[1] }]; #IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 } [get_ports {
vga_g[2] }]; #IO_L2N_T0_AD12N_35 Sch=vga_g[2]
set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 } [get_ports {
vga_g[3] }]; #IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]

set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 } [get_ports {
vga_b[0] }]; #IO_L2P_T0_AD12P_35 Sch=vga_b[0]
set_property -dict { PACKAGE_PIN C7    IOSTANDARD LVCMOS33 } [get_ports {
vga_b[1] }]; #IO_L4N_T0_35 Sch=vga_b[1]
set_property -dict { PACKAGE_PIN D7    IOSTANDARD LVCMOS33 } [get_ports {
vga_b[2] }]; #IO_L6N_T0_VREF_35 Sch=vga_b[2]
set_property -dict { PACKAGE_PIN D8    IOSTANDARD LVCMOS33 } [get_ports {
vga_b[3] }]; #IO_L4P_T0_35 Sch=vga_b[3]
```

```
set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 } [get_ports { vga_hs
}]; #IO_L4P_T0_15 Sch=vga_hs
set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports { vga_vs
}]; #IO_L3N_T0_DQS_AD1N_15 Sch=vga_vs

##Micro SD Connector

#set_property -dict { PACKAGE_PIN E2    IOSTANDARD LVCMOS33 } [get_ports {
sd_reset }]; #IO_L14P_T2_SRCC_35 Sch=sd_reset
#set_property -dict { PACKAGE_PIN A1    IOSTANDARD LVCMOS33 } [get_ports { sd_cd
}]; #IO_L9N_T1_DQS_AD7N_35 Sch=sd_cd
#set_property -dict { PACKAGE_PIN B1    IOSTANDARD LVCMOS33 } [get_ports { sd_sck
}]; #IO_L9P_T1_DQS_AD7P_35 Sch=sd_sck
#set_property -dict { PACKAGE_PIN C1    IOSTANDARD LVCMOS33 } [get_ports { sd_cmd
}]; #IO_L16N_T2_35 Sch=sd_cmd
#set_property -dict { PACKAGE_PIN C2    IOSTANDARD LVCMOS33 } [get_ports {
sd_dat[0] }]; #IO_L16P_T2_35 Sch=sd_dat[0]
#set_property -dict { PACKAGE_PIN E1    IOSTANDARD LVCMOS33 } [get_ports {
sd_dat[1] }]; #IO_L18N_T2_35 Sch=sd_dat[1]
#set_property -dict { PACKAGE_PIN F1    IOSTANDARD LVCMOS33 } [get_ports {
sd_dat[2] }]; #IO_L18P_T2_35 Sch=sd_dat[2]
#set_property -dict { PACKAGE_PIN D2    IOSTANDARD LVCMOS33 } [get_ports {
sd_dat[3] }]; #IO_L14N_T2_SRCC_35 Sch=sd_dat[3]


##Accelerometer

#set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 } [get_ports {
acl_miso }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
#set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports {
acl_mosi }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
#set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 } [get_ports {
acl_sclk }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
#set_property -dict { PACKAGE_PIN D15   IOSTANDARD LVCMOS33 } [get_ports {
acl_csn }]; #IO_L12P_T1_MRCC_15 Sch=acl_csn
#set_property -dict { PACKAGE_PIN B13   IOSTANDARD LVCMOS33 } [get_ports {
acl_int[1] }]; #IO_L2P_T0_AD8P_15 Sch=acl_int[1]
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 } [get_ports {
acl_int[2] }]; #IO_L20P_T3_A20_15 Sch=acl_int[2]


##Temperature Sensor

#set_property -dict { PACKAGE_PIN C14   IOSTANDARD LVCMOS33 } [get_ports {
tmp_scl }]; #IO_L1N_T0_AD0N_15 Sch=tmp_scl
```

```
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports {
tmp_sda }]; #IO_L12N_T1_MRCC_15 Sch=tmp_sda
#set_property -dict { PACKAGE_PIN D13   IOSTANDARD LVCMOS33 } [get_ports {
tmp_int }]; #IO_L6N_T0_VREF_15 Sch=tmp_int
#set_property -dict { PACKAGE_PIN B14   IOSTANDARD LVCMOS33 } [get_ports { tmp_ct
}]; #IO_L2N_T0_AD8N_15 Sch=tmp_ct

##Omnidirectional Microphone

#set_property -dict { PACKAGE_PIN J5    IOSTANDARD LVCMOS33 } [get_ports { m_clk
}]; #IO_25_35 Sch=m_clk
#set_property -dict { PACKAGE_PIN H5    IOSTANDARD LVCMOS33 } [get_ports { m_data
}]; #IO_L24N_T3_35 Sch=m_data
#set_property -dict { PACKAGE_PIN F5    IOSTANDARD LVCMOS33 } [get_ports {
m_lrsel }]; #IO_0_35 Sch=m_lrsel


##PWM Audio Amplifier

set_property -dict { PACKAGE_PIN A11   IOSTANDARD LVCMOS33 } [get_ports { aud_pwm
}]; #IO_L4N_T0_15 Sch=aud_pwm
set_property -dict { PACKAGE_PIN D12   IOSTANDARD LVCMOS33 } [get_ports { aud_sd
}]; #IO_L6P_T0_15 Sch=aud_sd


##USB-RS232 Interface

#set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 } [get_ports {
uart_txd_in }]; #IO_L7P_T1_AD6P_35 Sch=uart_txd_in
#set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 } [get_ports {
uart_rxd_ouT }]; #IO_L11N_T1_SRCC_35 Sch=uart_rxd_out
#set_property -dict { PACKAGE_PIN D3    IOSTANDARD LVCMOS33 } [get_ports {
uart_cts }]; #IO_L12N_T1_MRCC_35 Sch=uart_cts
#set_property -dict { PACKAGE_PIN E5    IOSTANDARD LVCMOS33 } [get_ports {
uart_rts }]; #IO_L5N_T0_AD13N_35 Sch=uart_rts

##USB HID (PS/2)

#set_property -dict { PACKAGE_PIN F4    IOSTANDARD LVCMOS33 } [get_ports {
ps2_clk }]; #IO_L13P_T2_MRCC_35 Sch=ps2_clk
#set_property -dict { PACKAGE_PIN B2    IOSTANDARD LVCMOS33 } [get_ports {
ps2_data }]; #IO_L10N_T1_AD15N_35 Sch=ps2_data


##SMSC Ethernet PHY
```

```
#set_property -dict { PACKAGE_PIN C9    IOSTANDARD LVCMOS33 } [get_ports {
eth_mdc }]; #IO_L11P_T1_SRCC_16 Sch=eth_mdc
#set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 } [get_ports {
eth_mdio }]; #IO_L14N_T2_SRCC_16 Sch=eth_mdio
#set_property -dict { PACKAGE_PIN B3    IOSTANDARD LVCMOS33 } [get_ports {
eth_rstn }]; #IO_L10P_T1_AD15P_35 Sch=eth_rstn
#set_property -dict { PACKAGE_PIN D9    IOSTANDARD LVCMOS33 } [get_ports {
eth_crsdv }]; #IO_L6N_T0_VREF_16 Sch=eth_crsdv
#set_property -dict { PACKAGE_PIN C10   IOSTANDARD LVCMOS33 } [get_ports {
eth_rxerr }]; #IO_L13N_T2_MRCC_16 Sch=eth_rxerr
#set_property -dict { PACKAGE_PIN C11   IOSTANDARD LVCMOS33 } [get_ports {
eth_rxd[0] }]; #IO_L13P_T2_MRCC_16 Sch=eth_rxd[0]
#set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 } [get_ports {
eth_rxd[1] }]; #IO_L19N_T3_VREF_16 Sch=eth_rxd[1]
#set_property -dict { PACKAGE_PIN B9    IOSTANDARD LVCMOS33 } [get_ports {
eth_txen }]; #IO_L11N_T1_SRCC_16 Sch=eth_txen
#set_property -dict { PACKAGE_PIN A10   IOSTANDARD LVCMOS33 } [get_ports {
eth_txd[0] }]; #IO_L14P_T2_SRCC_16 Sch=eth_txd[0]
#set_property -dict { PACKAGE_PIN A8    IOSTANDARD LVCMOS33 } [get_ports {
eth_txd[1] }]; #IO_L12N_T1_MRCC_16 Sch=eth_txd[1]
#set_property -dict { PACKAGE_PIN D5    IOSTANDARD LVCMOS33 } [get_ports {
eth_refclk }]; #IO_L11P_T1_SRCC_35 Sch=eth_refclk
#set_property -dict { PACKAGE_PIN B8    IOSTANDARD LVCMOS33 } [get_ports {
eth_intn }]; #IO_L12P_T1_MRCC_16 Sch=eth_intn


##Quad SPI Flash

#set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports {
qspi_dq[0] }]; #IO_L1P_T0_D00_MOSI_14 Sch=qspi_dq[0]
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports {
qspi_dq[1] }]; #IO_L1N_T0_D01_DIN_14 Sch=qspi_dq[1]
#set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports {
qspi_dq[2] }]; #IO_L2P_T0_D02_14 Sch=qspi_dq[2]
#set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports {
qspi_dq[3] }]; #IO_L2N_T0_D03_14 Sch=qspi_dq[3]
#set_property -dict { PACKAGE_PIN L13   IOSTANDARD LVCMOS33 } [get_ports {
qspi_csn }]; #IO_L6P_T0_FCS_B_14 Sch=qspi_csn
```

Appendix B: Python Scripts

create_song_coe.py

```python
import random

songs = [
    [127, 127, 127, 127, 76, 76, 74, 74, 72, 72, 74, 74, 76, 127, 76, 127, 76,
76, 127, 127, 74, 127, 74, 127, 74, 74, 127, 127, 76, 127, 79, 127, 79, 79, 127,
127, 76, 76, 74, 74, 72, 72, 74, 74, 76, 127, 76, 127, 76, 127, 76, 127, 74, 127,
74, 127, 76, 76, 74, 74, 72, 72, 127, 127],
    [127, 127, 127, 127, 76, 76, 71, 72, 74, 74, 72, 71, 69, 127, 69, 72, 76,
127, 74, 72, 71, 71, 127, 72, 74, 74, 76, 76, 72, 72, 69, 127, 69, 69, 127, 127,
127, 74, 74, 77, 81, 81, 79, 77, 76, 76, 127, 72, 76, 76, 74, 72, 71, 71, 71, 72,
74, 74, 76, 76, 72, 72, 69, 127, 69, 69, 127, 127],
    [127, 127, 127, 127, 72, 72, 127, 72, 72, 127, 72, 72, 72, 127, 127, 127,
127, 127, 72, 71, 71, 69, 71, 71, 72, 74, 74, 74, 76, 76, 127, 76, 76, 127, 76,
76, 76, 127, 127, 127, 127, 127, 76, 74, 74, 72, 74, 74, 76, 77, 77, 127, 79, 79,
79, 127, 127, 127, 72, 72, 72, 127, 127, 127, 127, 127, 81, 79, 79, 77, 76, 76,
76, 74, 74, 74, 72, 72, 72, 127, 127, 127]
]

MAX_LEN = 250
END_SONG = 0b1111100
PADDING = 127

with open('../6.111-fp/final_songs.coe', 'w') as f:
    f.write('memory_initialization_radix=2;\n')
    f.write('memory_initialization_vector=\n')
    for song in songs:
        songlen = len(song)
        padding = MAX_LEN - songlen - 1
        assert(padding > 0)
        # write song
        for i in range(songlen):
            f.write('{:08b},\n'.format(song[i]))
        # write song end signal
        f.write('{:08b},\n'.format(END_SONG))
        # pad
        for i in range(padding):
            f.write('{:08b},\n'.format(PADDING))
    # empty space for custom
    for i in range(MAX_LEN-1):
        f.write('{:08b},\n'.format(PADDING))
    f.write('{:08b};'.format(END_SONG))
```

hann_gen.py

```
import math

n_bits = 12
out_bits = 24

L = (2**n_bits)
N = L-1 # n in [0,N]

def hann(n):
    assert(0 <= n <= N)
    c = math.sin(math.pi * n / N)
    print((2**out_bits) * c**2)
    return int((2 ** out_bits) * c**2)

tab="    "
if __name__ == '__main__':
    with open('hann.sv', 'w') as f:
        f.write('{}module hann(input logic [{}:0] n, output logic [{}:0]
coeff);\n'.format(0*tab, n_bits-1, out_bits-1))
        f.write('{}always_comb begin\n'.format(1*tab))
        f.write('{}case(n)\n'.format(2*tab))
        for n in range(L):
            f.write("{}{}'d{}: coeff = {}'d{};\n".format(3*tab, n_bits, n,
out_bits, hann(n)))
        f.write('{}endcase\n'.format(2*tab))
        f.write('{}end\n'.format(1*tab))
        f.write('{}endmodule\n'.format(0*tab))
```