Holiday Hotkeys

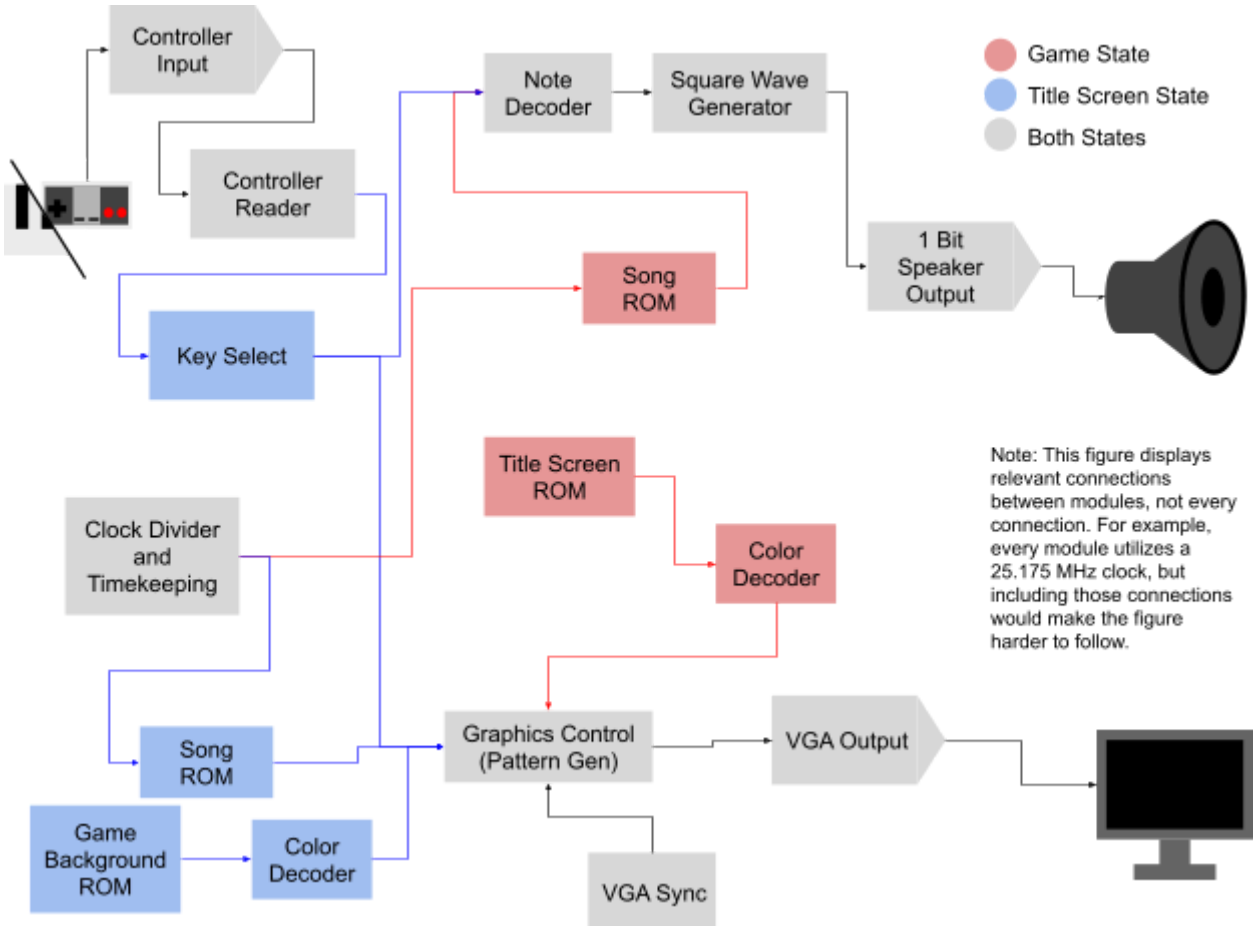
Andre Chabra, Paul Hickey, Ryan McLean, Juan Garcia

Github: https://github.com/McLeanRyan/MIDI-Master

(Note: final code is listed under finished_v3, NOT main)

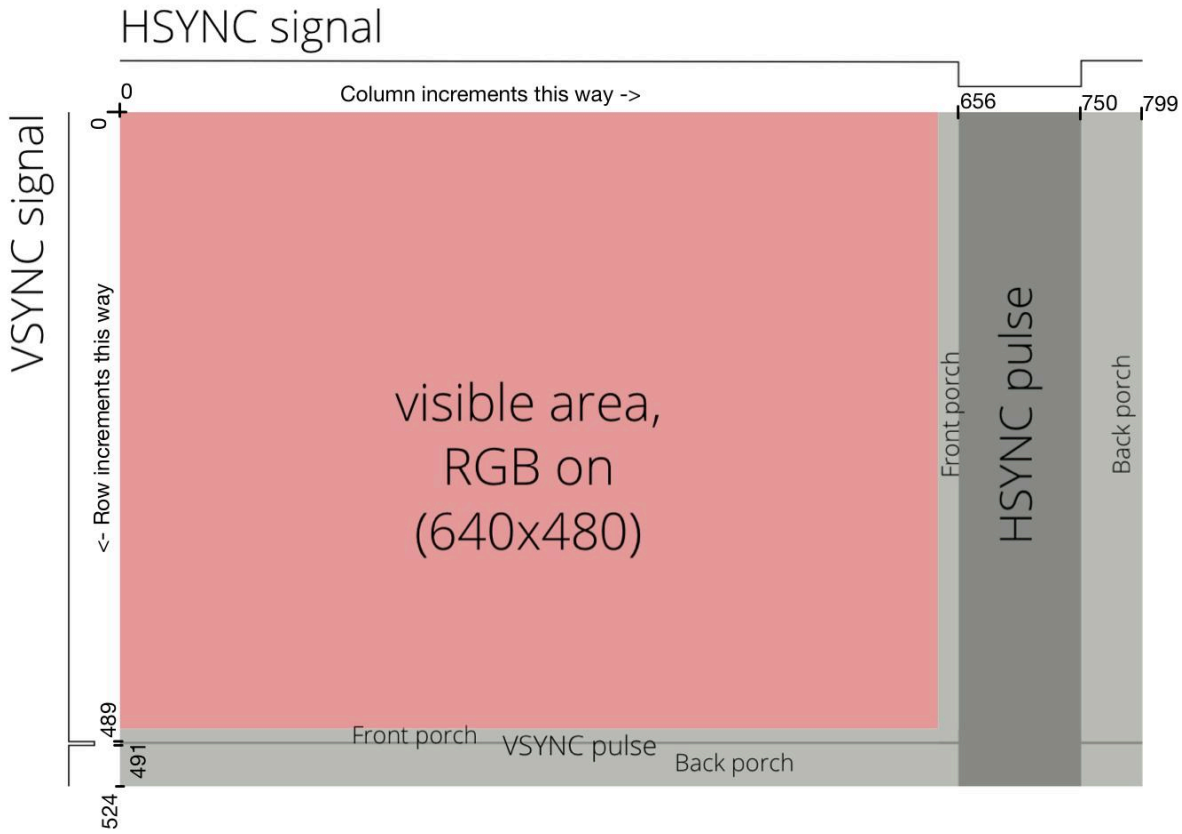
Overview

Holiday Hotkeys (previously named MIDI Master as can be seen in the GitHub repository), is a single-player rhythm game where the player is tasked with playing a slowed down version of Sleigh Ride by Leroy Anderson. Players use an NES Controller to navigate an on-screen keyboard. The left and right inputs on the D-pad move left and right on the keyboard, the A button plays the currently selected note (players can press and hold to play a note as long as they like), and the START button moves between the game and the start screen. As a fun twist, the game's start screen is horizontal while the game itself displays on the screen vertically (requiring the player to rotate the screen 90 degrees before playing!). The game itself has no scoring or winning state. The player gets to play Sleigh Ride on loop as long as they please, and they can return to the start screen at any time. As a final note, the piano itself is fully functional. Players have the option to fully ignore the falling notes and play whatever they would like on the piano.



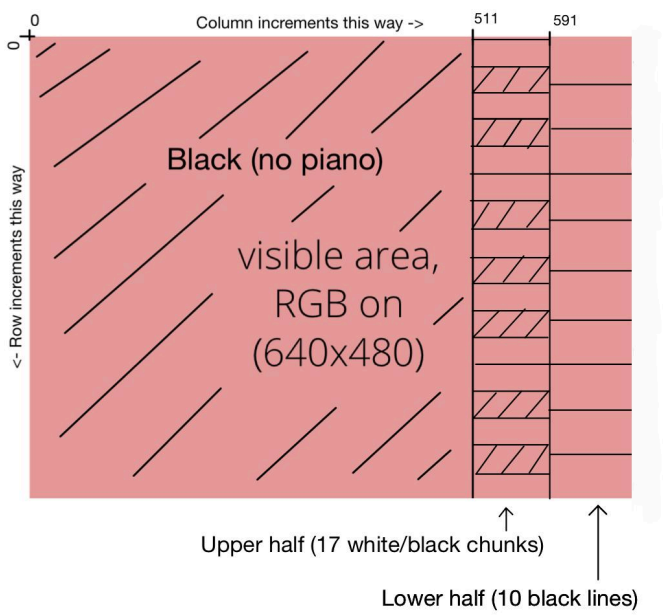
Graphics

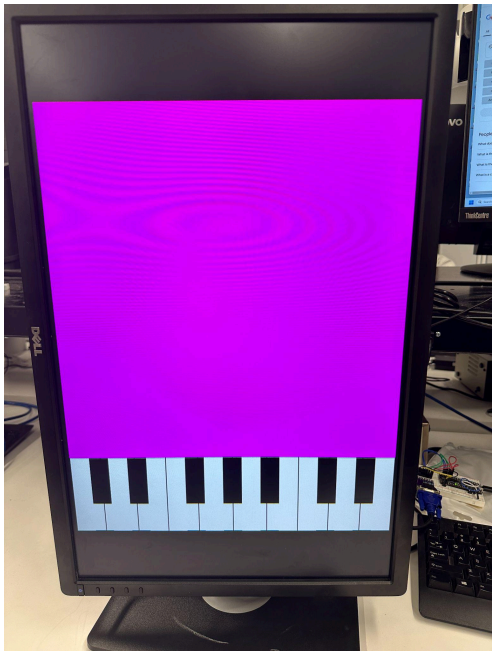
VGA



The VGA module is the backbone for displaying visuals on the screen. It displays 640 x 480 pixel images through its six main outputs (R1, R2, G1, G2, B1, B2, HSYNC, VSYNC). It takes a 25MHz clock as an input from the FPGA's PLL module. The VGA module is also in charge of incrementing two 10-bit unsigned numbers, row and column, which are used heavily to generate the graphics for the project (more on that in other modules). The row and column are used to describe a position on the screen, but it should be noted that they actually increment beyond the 640 x 480 pixel display size due to the nature of how VGA is displayed. The counters increment until 524 and 799, using the counts beyond the screen resolution to calculate when the HSYNC and VSYNC signals should be high/low. This is all depicted in the annotated diagram above.

Piano



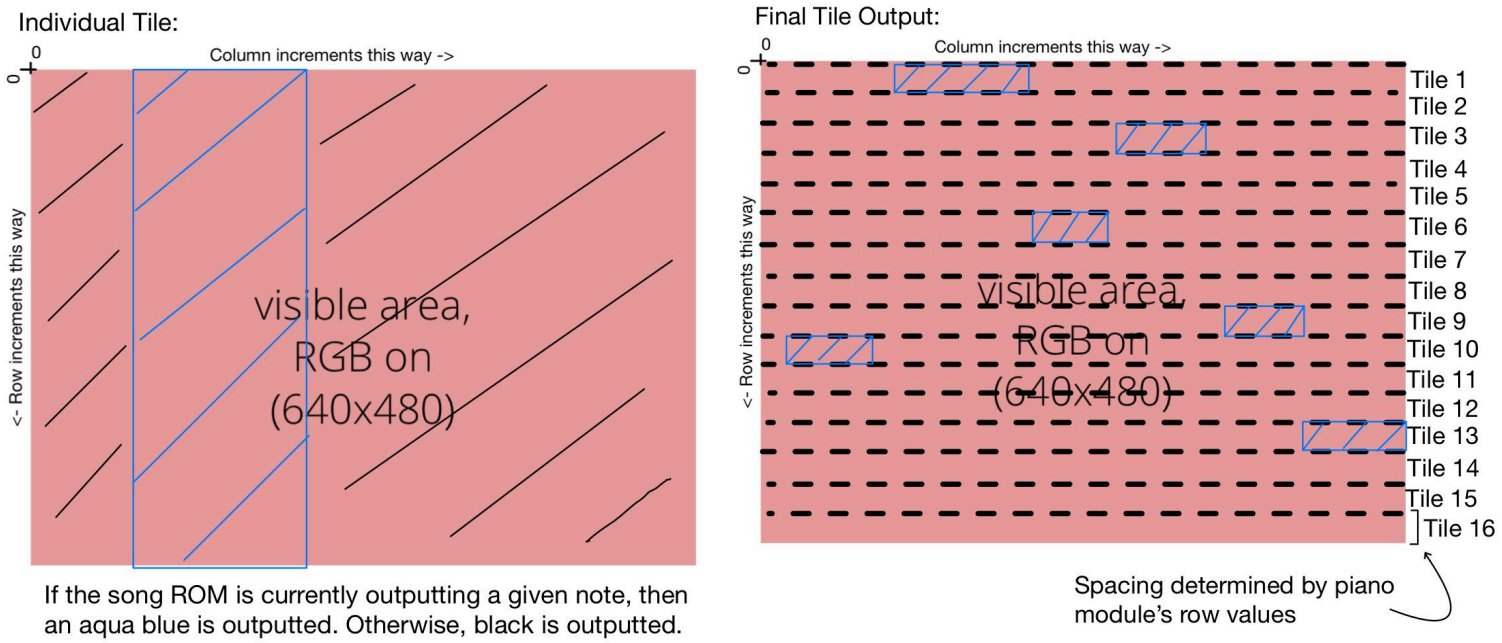


(in the picture above, the non-piano section was changed to purple for visual clarity)

The on-screen piano is generated by using the VGA module's row and column outputs. At any given column or row, a six-bit RRGGBB value outputs either white or black onto the screen. The top and bottom halves of the piano (top being where the black keys are, bottom being where they are not present) are generated separately using hard-coded values. The piano as a whole takes up exactly one fifth of the total display size, meaning that all columns beyond 511 have an RGB value associated with the piano. The widths of each tile are also explicitly written out in the piano.vhd module, which made generating correctly sized tiles very easy.

The yellow bar that sits on top of the player's currently selected piano key is controlled by a large Moore Finite State Machine. There are 17 possible states, one for each key, and the FPGA is always checking to see if either the left or right buttons on the D-pad of the NES controller have been pressed so it can switch to the next key state. This approach allowed us to easily implement wrap-around, where pressing right when you are on the rightmost key will loop you around to the left most key and vice versa. By knowing which state the keyboard is currently in, it knows when to send out the yellow RGB value that gets layered on top of the piano module the player sees on screen.

Tiles



The aqua blue tiles are generated by a module called tile2 which is the second iteration of our work to get the tiles to fall on cue with the music. The tiles are printed by looking ahead in the song ROM. If the row where falling tiles meet the piano is considered to be the present time, then higher rows represent the notes to be played in the future. We decided that the highest row should represent 2 measures (32 rom addresses) in the future of the current note being played. This means that the 512 pixels between the piano and top of the screen need to display 32 16th note tiles at a time, so each tile should be 8 pixels tall.

The simplest way to address this issue is to assign each 8 pixel region on the screen to a number of notes in the future from the current note being played, but that would cause the blocks to fall in 16th note increments, which would give an FPS of **4** (16th notes per beat) * **100** (bpm) / **60** (seconds/minute) = **~15 Hz**. In order to create smoother falling blocks, we effectively slide the 16th note boundaries between notes down at a speed of 8 pixels per note. To achieve this speed, we extended the music counter 3 bits to count 8ths of 16th notes instead of 16th notes.

We can now see the screen in 8ths of tiles instead of whole tiles, where the distance in pixels from the piano to a tile represents how many 8ths of a note that tile is in the future. For any given pixel, we can add that pixel's distance from the piano (our lookahead time) to the song counter's current time (in 8ths of notes) to find the time represented by that pixel.

Put simply, current time (in 8ths of notes) + pixel distance to piano = time at that pixel. Since the piano is at pixel 512, the expression becomes: current time + 512 - row # = time at pixel. Because the tiles are 1 note/8 pixels long, the 3 fractional bits of this number can be ignored (they represent 8ths of a tile), and so the resulting number is given to the song ROM.

Adding 512 to a 9 bit number does nothing, so the final expression is: time at pixel = (current time (8ths of notes) - row (pixels)) >> 3.

From here, generating graphics is simple. For each pixel above the piano, that math is done and the resulting time is used as an address for the song ROM. If the ROM returns a note matched by the piano key underneath that pixel's column, the color blue is printed, otherwise black is printed. Using the row and column counts from VGA, we go through the whole screen and color each pixel depending on whether the song ROM says the note in that column needs to be played.

ROMs

Our Holiday Hotkeys title screen and snowy night game background images are stored in Read-Only Memory (ROM) on the FPGA. They were drawn by hand, pixel by pixel on an iPad and then exported as PNGs from which we could generate the ROM files. A ROM in VHDL is made by using case statements, where each case is represented by an address that corresponds to a specific pixel and what color is stored at that pixel. In our ROM files, we take in a column (x) position and a row (y) position on the screen and then using the PLL clock we concatenate those values to get a unique address and then we look up the RGB value at that address which is displayed on screen through our pattern_gen module.

When displayed on the monitor, our title screen image takes up the full 640 x 480 pixels on the display and our background image takes up 512 x 480 pixels. However, the FPGA only has 122,880 bits of memory. That memory comes from 30 Embedded Block RAMs (EBRs) built into the FPGA and each EBR can only hold up to 4096 bits. This meant we could not store either image on the chip without compressing them. We decided it was best to store the images at ¼ of their intended size and then scale the images when they are displayed on screen. Also, the way ROMs work in VHDL requires that every possible address has a value. The number of possible addresses depends on the bit length of each address, so as part of our scaling process each image has the 2 least significant bits of its row and column values removed. By removing those bits we are dividing by 4 so when a ROM image is displayed on screen it treats blocks of 4 pixels as one and colors them the same. This creates a lower resolution image which takes up the full screen as intended. The title screen ROM requires a 15-bit address using this scaling process. The row value is 9 bits long because the highest y-position is 479 and the column value is 10 bits long because the highest x-position is 639. By removing 2 bits from each and concatenating them, we get a 15-bit address. The background ROM only requires a 14-bit address because the highest x-position we need to store is 511 which can be stored in 9 bits. So removing 2 bits from each

and concatenating them gets us a 14-bit address. Finally, the number of bits in an RGB value also needs to be accounted for because each every possible address in the ROM file stores an RGB value. So the total number of bits required for any image is given by the formula below:

$$total\ bits\ =\ 2^{address\ bit\ length}\ \times\ RGB\ bit\ length$$

To account for the 6-bit RGB values, we decided to encode each RGB value with fewer bits. Our title screen only uses 4 colors so we encoded each 6-bit value as a 2-bit value. Since our background image is smaller, we had more flexibility with colors and decided to use 3-bit values for a total of 8 colors. The table below shows the final memory usage for each image as well as their uncompressed counterparts for reference:

	Row Bit Length	Column Bit Length	Address Bit Length	RGB Bit Length	Total Bits	Total EBRs
Title	7	8	15	2	65,536	16
Background	7	7	14	3	49,152	12
Title Uncompressed	10	10	20	6	6,291,456	N/A
Background Uncompressed	10	10	20	6	6,291,456	N/A

Since even our compressed ROM images are still quite large and every pixel in each image needs its own case in the VHDL file, we decided to write a Python script to generate those ROM files for us. The script takes the PNGs we drew and parses each pixel to get its RGB value, and scales that 8-bit RGB value down to a 6-bit RGB value. Then, it takes each 6-bit RGB value and assigns it to a final encoding (either 2 bits or 3 bits depending on the image), and finally writes out a VHDL ROM file using a case statement which encodes every pixel with its address and encoded color value.

Audio/Controls

Clock Divider

Different parts of the game require counters and clocks running at different rates. We consolidated these into one module in order to make it simpler when we needed to edit things down the line. This module is driven by the 25.175 MHz clock generated by PLL, and outputs slower clocks needed to run the NES controller (83.3 kHz) and for the audio sampling (48 kHz). This file also includes two counters that increments at different tempos of the song, which determines the speed of the song on the start screen as well as the speed at which the blocks fall.

Audio

The audio system is composed of multiple modules that operate together. The entire song with all of its notes is stored in a ROM called song_rom where each note is stored as a 4 bit value since we are only encoding for 16 possible states, the 15 notes used in the song, and the rest note when no music is played. This ROM file was generated by transposing the original music into a new format that fit the notes we were using on our keyboard. This was then transcribed into a text file in a process that took the musical notation and turned it into letters and symbols that represent the notes and timing of the song. Then we ran a custom Python script on this text file which generated a ROM containing the entire song. This ROM only uses $2^9 \cdot 4 = 2048$ bits of memory so it can fit in one EBR. Then we use the clock divider which has the counters which correspond to the tempos of the song. For the title screen, we feed the faster tempo to the song_rom, which outputs a 4-bit note that then goes into the keysDecoder module. This module is basically one giant mux that takes in a 4 bit value and then outputs the corresponding note as a 16 bit std_logic_vector which corresponds to the waveform for a specific musical note. This 16-bit value gets routed to square.vhd which is the final step before the speaker. The square module takes the value and then using the PLL clock, cycles through the bits in the value and pulses out a series of std_logic values to the speaker in the form of a square wave corresponding to the frequency of the desired note. When this process is driven at the correct tempo using the title screen note counter, it sounds like music and the song is clearly distinguishable.

A second instance of song_rom exists in our graphics modules as well for drawing the tiles on screen correctly. It gets the slower note counter so the actual game is more playable than the much faster version of the song which plays on the title screen. This is discussed more in our Graphics section.

Actual sound is generated as a square wave. Though we originally planned to make a polyphonic synth (able to play multiple notes), as we began implementing the musical ROM module and planning out sound we decided it would be easier to use a monophonic synth (one note).

NES Controller

The input to this game is driven by an NES controller. The controller has 8 inputs, which are read in and stored in an 8-bit shift register. The FPGA outputs 3.3V, ground, clock, and latch signals to the NES controller and takes in the data from the controller as an input. As mentioned previously in the clock divider section, the clock that has to be sent out to the controller needs to be at 83.3 kHz. The output clock is only outputted on the eight cycles after the latch goes high, reading one bit of data every cycle. The latch signal goes high for one clock cycle after ~256 NES clock cycles pass and then falls back to low after one cycle. This is counting the internal 83.3 kHz clock rather than the output clock that is running at the same speed but is off for periods of time.

When the clock is outputting to the controller, the incoming data is pushed into an 8-bit shift register on the rising edge of the clock signal. Once eight pieces of data are being read in, the contents of the shift register are copied to a second 8-bit shift register. We then use this to output to the rest of the game, with one bit of the register mapping to an individual std_logic signal for each button. These eight signals are the outputs of the NES module. This allows for the output signal to not fluctuate weirdly as data is being read in, and makes sure that the current button signal is in the correct location in the register.

The NES controller also required debouncing. This is a way to digitally smooth the physical errors in the buttons on the controller. We found a lot of very unstable behavior when trying to get a single response from a button press, where the game was instead registering a lot of inputs as the button was actually moving slightly up and down incredibly quickly. Debouncing removes this small bouncing of the button, which we do by simply placing a “cooldown” on the button after it is placed. When the button is pressed and the signal goes high, the button enters the cooldown state. While in this cooldown state, the button output is set low until a counter reaches a certain value. We used VSYNC as the clock for the debouncing module and made the counter increment once, effectively making it so that the player has one input per frame. This solved our issue of the NES controller giving several false inputs for each real input.

Results

One thing that we did not have time to implement was using MIDI input for our game. Our original plan was to use a MIDI keyboard as input for our game. However, we ran into hardware issues and were not able to properly connect up the keyboard in the way that the MIDI protocol requires (as shown here). We were able to successfully pivot from the MIDI keyboard to an NES controller and developed a solution that we were all very happy with. The final product was bug free and exceeded all of our expectations. Players had a lot of fun trying to play Sleigh Ride, and some of them ignored the tiles and used the piano to play other songs (one player even managed to play the theme from the hit game Among Us on the piano!).

Reflection

Looking back, we probably should have pivoted from the MIDI keyboard to the NES controller sooner. We switched over the weekend, but had we made the change a day earlier, we could have utilized more hours that were spent waiting before we could start integrating discrete components of the system. If we had changed sooner, we could have added more functionality to the NES controller. A lot of players noticed how it was extremely difficult to get to the correct key on time because of how fast the song was and how long it takes to travel up and down the keyboard. If we had more time we could have implemented a feature where the player could double tap the left or right keys on the D-pad to skip some notes and move to a different key state more quickly. Also, if we knew we were doing the NES controller earlier, we could have spent more time on the music and possibly even added another song considering we definitely had enough lookup tables and EBRs to do so.

Work Division

Andre: background_rom, my_rom, key_select, pattern_gen, top, and misc. edits to most other modules
Paul: Audio Synthesis (square, song_rom, keysDecoder), python scripting for song memory, and math for printing tiles
Ryan: nes_controller, clock_divider, debounce, music transcription/transposition, misc. integration and bug fixing
Juan: piano, tile2, key_select, pattern_gen, top, nes_controller, debounce, vga