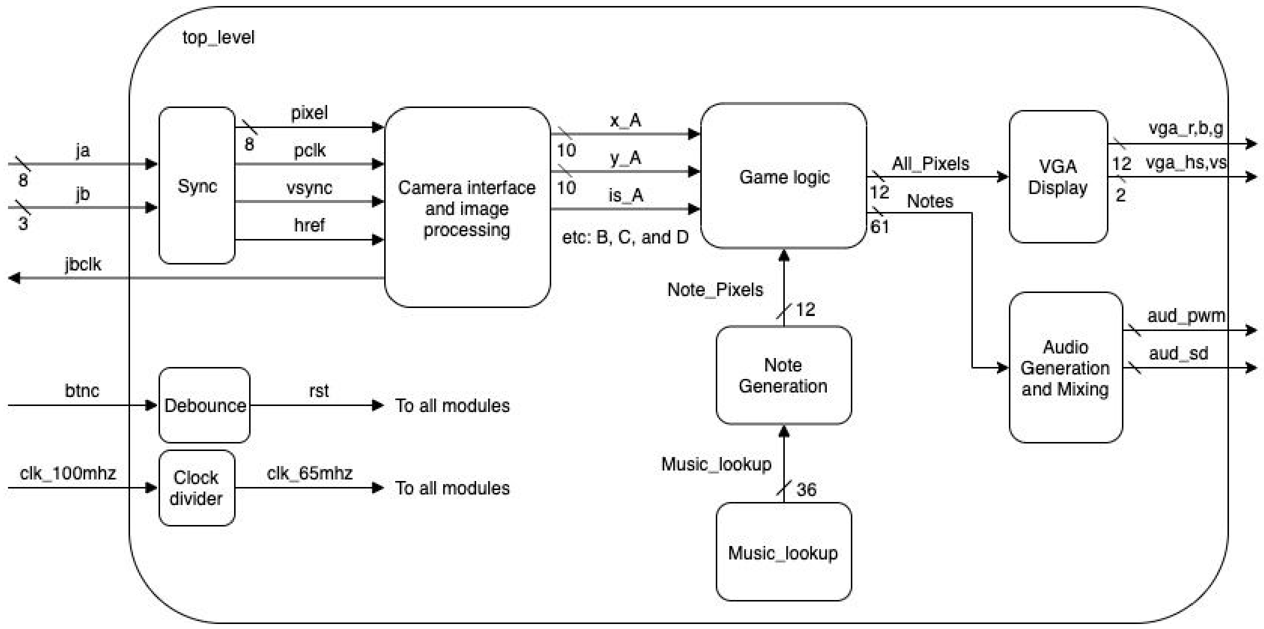
FPGuitAr Hero

Sarah Spector and Alejandro Diaz

6.111 Fall 2019

Table of Contents

Overview and System Diagram



FPGuitAr Hero combines camera-based motion tracking with a graphical interface and audio output to create a game that gets people moving through music. In *FPGuitAr Hero*, players hold up to four colored lights, which are tracked separately by a camera using chroma-keying and mapped onto the graphical interface as paddles. On this interface, users must “hit” the notes of a song as they stream down the screen. Notes are represented by rectangles, whose pitch is encoded by the x position along the screen (61 possible pitches), every octave has a different color (5 possible octaves + A6), and length is encoded by height of the rectangles. The player earns more points the longer they are able to “hold” the note during its potential duration, and holding a note produces the corresponding audio output via speakers.

External hardware

Camera and microcontroller (Sarah)

We use an OmniVision OV07670 CMOS VGA camera chip for LED tracking. The camera connects to the FPGA via the ja and jb ports. To conserve FPGA resources, we initialize registers on the camera using a pre-programmed microcontroller (WeMos D1 Mini) that communicates with camera over I2C. The microcontroller is programmed via the Arduino IDE on a laptop.

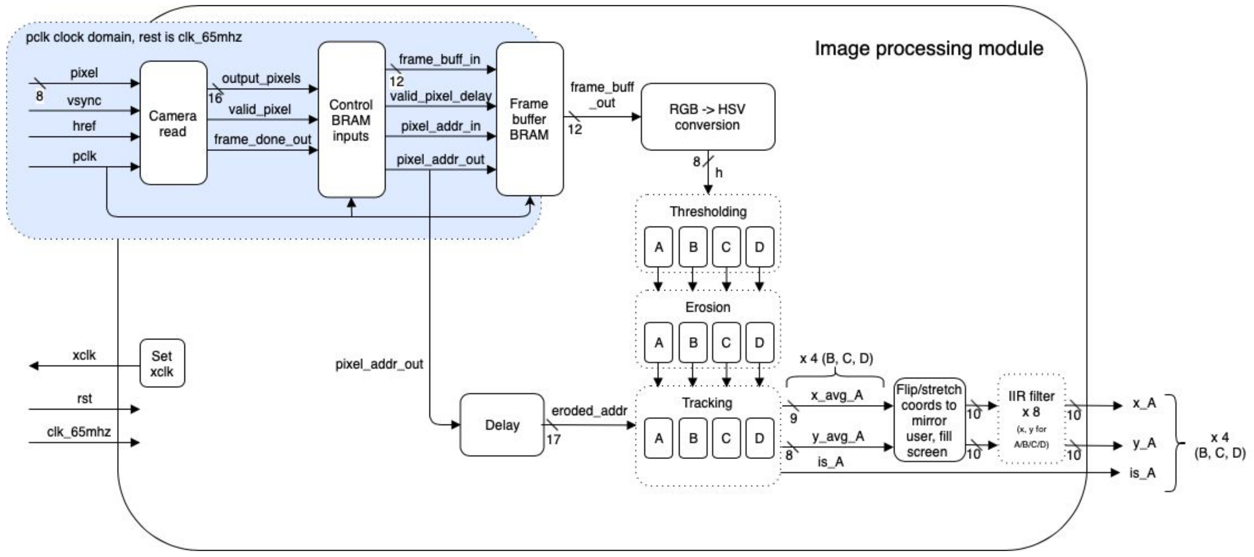
The key parameters we adjust in the camera registers are the resolution, set to the lower option of 320x240 to conserve FPGA resources and memory, and R, G and B gain, which we set to 50 for each channel. After a lot of experimentation, we found that setting the gain lower than the default (80) was ideal for separating out LEDs, which require less gain to be visible, from the scene. We also made the gain in each channel the same since we track LEDs in all parts of the hue spectrum.

Display and Speakers (Alejandro)

The project involves both graphical and audio output; a standard VGA display acts as a GUI for our game. A set of speakers connected to the FPGA via the audio jack plays the music generated by *FPGuitAr Hero* using a sine generator with a 128 sine lookup table (f(x) = sin(x) + sin(2x) + sin(3x) flipped about a vertical axis).

Subsystems

Camera interface and image processing (Sarah)



Camera interface and frame buffer (Sarah)

After synchronization, the signals from the camera on ja and jb which represent the image pixel (8 bits), vsync, href, and pclk are fed into the camera_read module written by Joe Steinmeyer. Pclk is a clock signal generated by the camera which is used to process pixel data at the appropriate rate; consequently, the camera_read module and input side of the frame buffer operate in the pclk clock domain, while the output of the frame buffer and the rest of the system can operate in the 65MHz system clock domain. The frequency of pclk is set by the interface module using a simple divider to be 65MHz/4 = 16.25MHz but the signal is delayed such that we can read the camera output on the rising edge of pclk.

After processing, the camera_read module spits out a 16-bit pixel (5 bits R, 6 bits G, 5 bits B) as well as signals encoding a valid pixel and the end of a frame. These signals are used to write 12-bit versions of the pixels (4 bits per R/G/B) into a frame buffer, the only significant use of memory in FPGuitAr Hero. The BRAM is simple dual port, 16x76800 (76800 is 320*240 for the number of pixels in a frame) and uses 4 18Kb memory blocks and 34 36Kb memory blocks for a total memory usage of 1296Kb. The frame buffer is necessary due to the crossing of clock domains between the camera's pclk and the 65MHz system clock; we could theoretically avoid it if we used pclk as the system clock, but this would increase the system's latency since pclk is 4x slower than 65MHz, and the memory-speed tradeoff makes sense for us since our project is not memory-intensive.

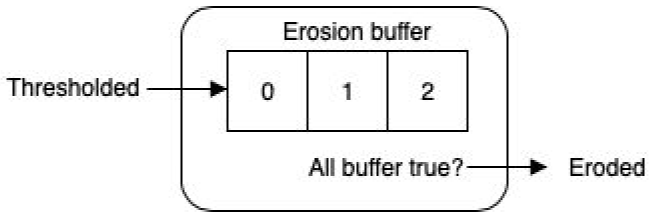
The frame buffer spits out one 12-bit pixel per cycle (with a latency of 2 cycles) on the 65MHz clock, with the pixel address incrementing on each clock cycle up to 76800 when it is reset to 0. Pixels are read from the frame buffer and processed faster than they are written due to the change in clock speed, which does not affect the system's throughput but reduces its latency.

Image processing (Sarah)

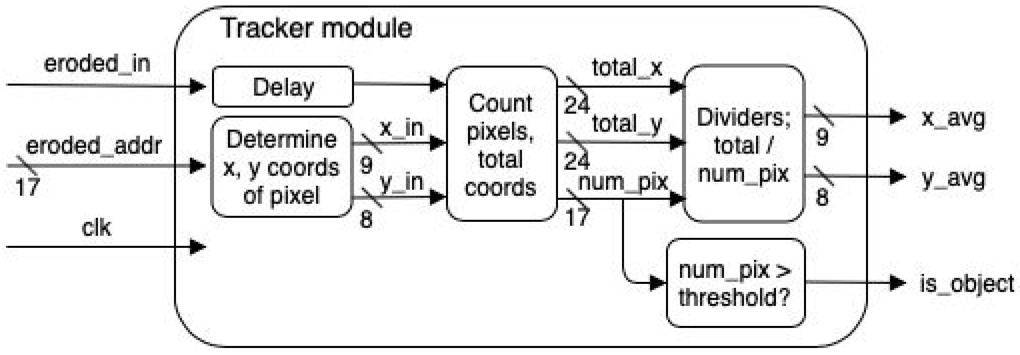
12-bit pixels coming out of the frame buffer first undergo conversion from the RGB to the HSV color space. Chroma-keying based on HSV is superior to using RGB values because when differentiating colored objects or lights, what you care about is differences in hue rather than saturation or value, which can change based on lighting conditions; the HSV space isolates useful data from noise.

The 8-bit hue value of the pixel is then passed to four different instances of the thresholding module in parallel, each set up to threshold for a narrow range of the hue spectrum. Upper and lower hue bounds were set based on experimentation with each of four different colored lights (blue, green, purple, and red). Each thresholding instance passes out a binary 1 or 0 depending on if that particular pixel was within the hue boundaries.

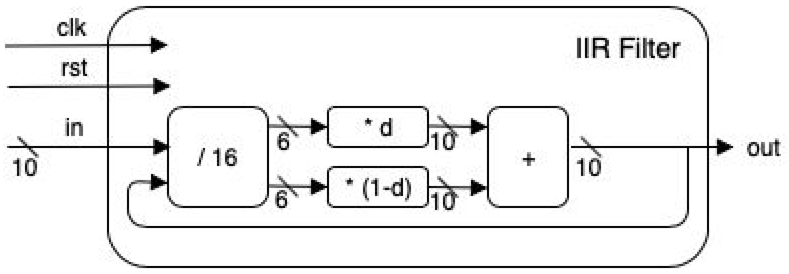
The thresholding instances each pass their output to a separate, parallel instance of the erosion module. To maintain low latency and high throughput, and because the application is not too noise-sensitive, we implemented a linear n=3 kernel instead of a 2-dimensional one; basically, it is a buffer of length 3 which shifts in a thresholded bit on each cycle and constantly outputs the "and" of all 3 bits in the buffer. This allows data to flow smoothly through the system without being held up to wait for later bits to come through for a multi-dimensional buffer. The latency of the module is 2 cycles and the throughput is 1 output bit per cycle. The eroder does not account for pixels at the edge of the screen only being surrounded by 1 pixel instead of 2, but this has a negligible impact on the final system and is safe to ignore.



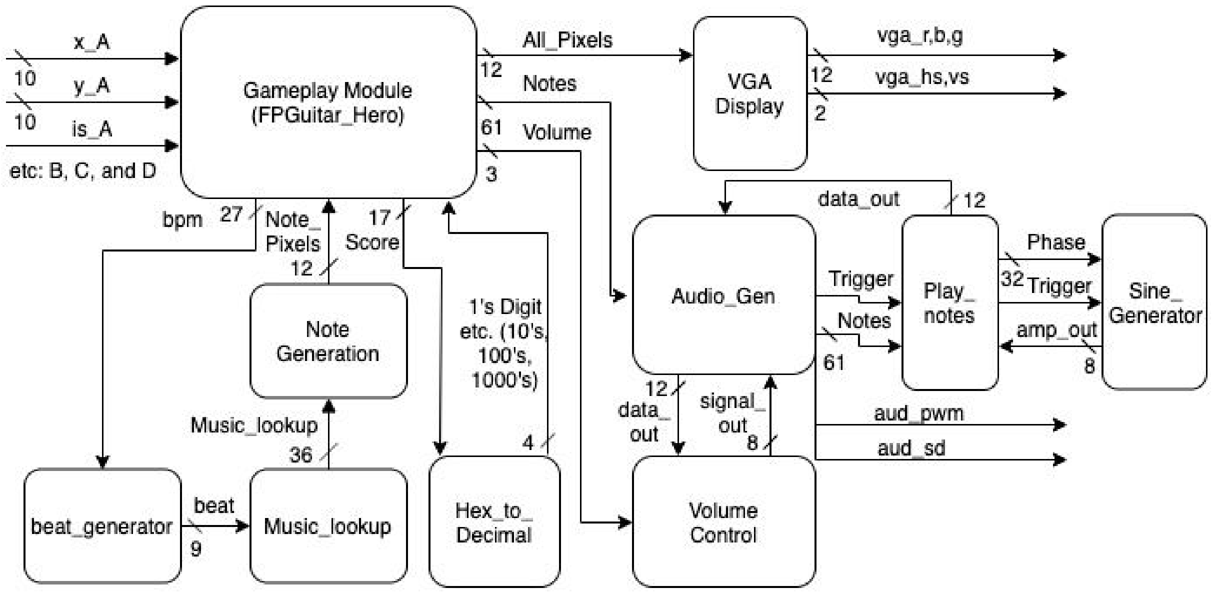
The output of the erosion module instances pipe into separate, parallel instances of the tracking module. This module is a state machine which determines the x, y coordinates of the current pixel based on delaying the address of the frame buffer output the correct number of cycles to account for the latency of the image processing between the frame buffer and tracker. During each frame, the state machine accumulates the number of 1's coming in from the erosion module as well as the corresponding x and y coordinates for pixels that pass erosion. IP core divider modules output the average x and y coordinates (total_x/num_pixels and total_y/num_pixels), which are sent out of the tracker module only at the beginning of a new frame. Also at the beginning of a new frame, the tracker module outputs a binary value representing whether the number of pixels that passed erosion in the previous frame was greater than some threshold; this is used to detect the presence or absence of an object of a particular color.



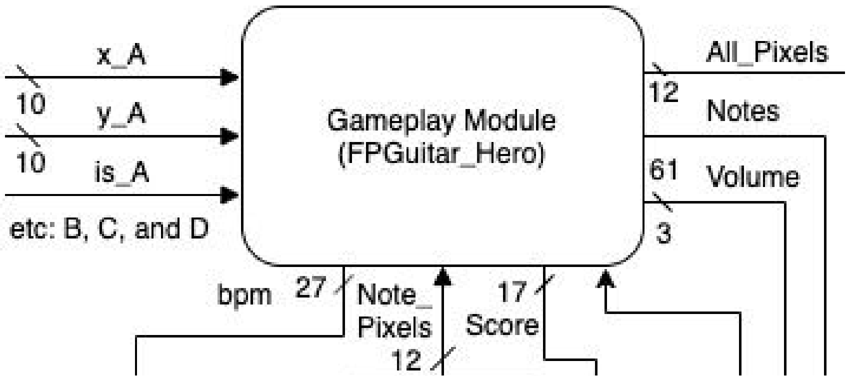
We flip the x-values over the center of the screen by subtracting them from 320 so that the onscreen sprites mirror the player's motions, and multiply all coordinates by 3 to scale from the camera-sized screen to the VGA-monitor size screen. Then, the calculated average x and y coordinates for each of the four objects pass through a separate, parallel infinite impulse response (IIR) filter, for a total of 8 instances of the IIR filter module. The IIR filter decay factor is set by a parameter within the module, and experimentation has shown that 0.5 works well for gameplay. The outputs of the 8 IIR filters combined with the 4 object detection signals together constitute the output of the entire image processing module. The average coordinates are delayed several cycles behind the object detection signals, but it makes no practical difference.



Overall Gameplay and Audio Generation (Alejandro)



Gameplay Module (Alejandro)



Gameplay module is an FSM that has five main states: opening screen, song selection, speed selection, actual gameplay, game over with score.

1. **Opening Screen**: begins with "FPGUITAR HERO" which stays on screen for about 4 seconds using a simple timer. The letters are displayed using a COE file and is described in more detail in the VGA display section. The only output to the screen is the letters.
2. **Song Selection**: screen displays "SONGS", the first player paddle (blue) and 4 numbers (0, 1, 2, 3). In order to select the song the blue x, y values are used to move the hand paddle. The user then hovers over one of the numbers for about 2 seconds. This intersection is detected by checking every frame (when vsync == 1 and hsync == 1) whether or not pixel_values from the hand and one of the digits are non_zero on the same pixel. If this intersection occurs, a flag is thrown every frame. Then a timer begins and once the 2 second mark is reached the song is selected (if the user briefly goes over a number, but does not choose it, the timer resets). The song that is selected affects the music that is looked up in music lookup and decides what is the final beat of the song (when to move on from reading music and go to the game over screen).
3. **Speed Selection**: is very similar to Song selection. The exceptions are "SPEED" is displayed and when the numbers are selected they set the bpm in the game. The FSM then proceeds to gameplay.
4. **Play the Game!**: takes the scaled x, y positions and boolean values (4 boolean values for each color denoting their presence) of up to any combination of 4 hands with blue, red, green and purple colors. The x positions are used to generate hand analogues on the screen during gameplay using the note_blob module. Blue is treated as the first player. The y value of blue is used to control volume by taking the top bits of the y coordinate and feeding that into volume for the audio generation modules. The volume bar moves with the y coordinate of blue. The words "SCORE" and "VOLUME" are displayed

scaled down to better fit the screen (scaling process is described thoroughly in the VGA display section).

The score was initially displayed on the hex display and is now displayed at the top of the screen. The score is calculated by incrementing the score by the number of intersections of hands with notes every frame. The process of throwing flags for each note and not each pixel overlap enables accurate scorekeeping. The scoring is consistent between runs on the same speed and song; however, the scoring is not consistent between speeds. I tried to mitigate this by only incrementing score every time the notes shift down a note, but because this pixel_step happens at a faster frequency than the frame rate for the VGA display the slower speeds allow the user to get a higher score. Although this could be a desirable feature (It just wasn't necessarily intended). The hex_to_decimal conversion for display on screen is detailed in its own section.

Once gameplay has begun, music lookup commences and notes stream down (changing x, y coordinates) from the top of the screen with a specific BPM, rectangular shape and color (corresponding to the 5 octaves + A6). This note generation is detailed separately as well.

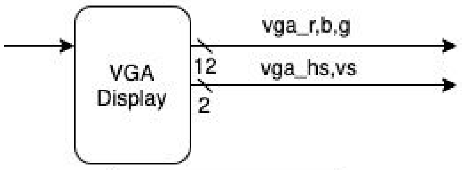
The hand analogues (paddles) that are being tracked (based on the boolean values) are superimposed on a straight horizontal line on the screen in order to standardize timing in the game. When the notes and hands intersect the corresponding note is played based on the x position. The note and hand intersections are detected in a similar manner to song/speed select detailed above, but just on a grander scale because there are up to 61 possible notes. If a note is detected in a frame, then the notes array is updated to have a one in the corresponding location. This array is sent to audio generation to actually generate the corresponding note and audio. The game finishes when the last beat is reached (depends on final beat corresponding to song) and the game goes to the Game Over state.

5. **Game Over**: Displays the score in the center of the screen and the words "SCORE." There is nothing new here. A timer runs for about 4 seconds and then the game restarts back to the first state.

Hex_to_Decimal Module (Alejandro)

This module is fairly straightforward. If score is more than 1000 then 1000 is subtracted until the number is less than 1000. At the point a similar process occurs for 100, 10 and 1. At each point the number of subtractions is recorded to determine what number corresponds to which digit for the decimal conversion. I am careful and intentionally make sure that I do not constantly overwrite the digits as the hardware runs, but rather I wait until all numbers have been determined before updating the score. All four digits are then outputted back to the gameplay module for display.

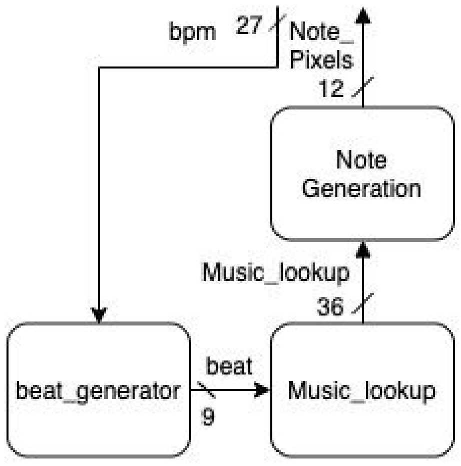
VGA Display (Alejandro)



The VGA Display module relies heavily on the already implemented module that was used in the Pong Lab. This provides 16 different individual RGB values that can be displayed on the monitor. Each pixel is displayed one at a time on a 65MHz clock. A v_sync and a h_sync signal specify to the monitor where each pixel should be displayed.

However for Note and Alphabet display there were alterations from the picture_blob module. For example the x index is x + 2 to account for latency and there is a number available for either a specific digit or a letter to be able to essentially index into the COE file (for example to get a 3 digit send in a 3 to the picture_blob_digit module). For the alphabet and digit COEs this index is multiplied by a distinct number corresponding to pixels in the COE. Also there is the ability to scale the image up in multiples of 2 which is specified with another offset and right shifts the x and y (the height and width coming into the image must also be scaled up).

Note_Generator Module (Alejandro)



Music is looked up from the music_lookup module in 1/16th chunks based on the beat provided from beat_generator. Music is stored in 36 bit chunks composed of up to 4 notes. Each note is composed of 9 bits. 6 bits specify which note within the 5 octaves (60 + 1 notes including the high A6) and the remaining 3 bits specify the length of the note (0: 1/16, 1: ⅛, 2: ¼, 3: ½, 4: 1, 5: ⅛ +1/16). Each chunk corresponds to a 1/16 note length of time. The music is stored in a look up table corresponding to the current song and the current beat.

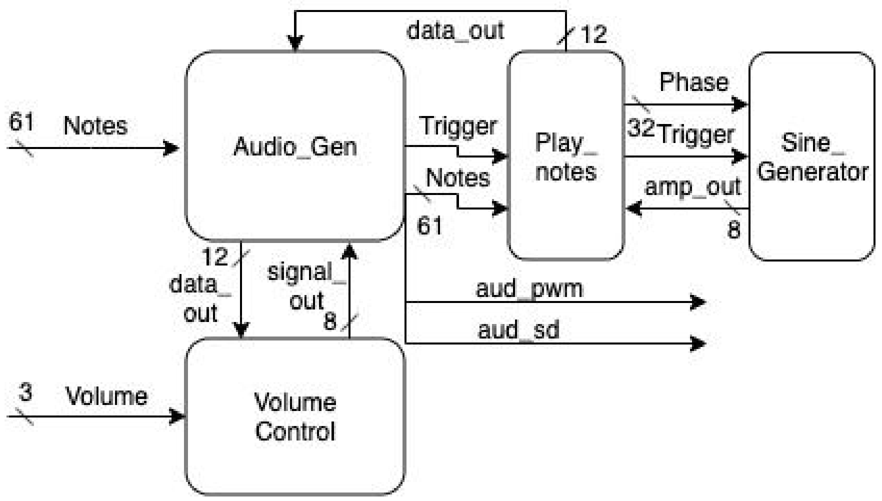
1	2	3	4	5	6	7	8	9	10	11	12
A1	**Bb1**	**B1**	**C2**	**C#2**	**D2**	**Eb2**	**E2**	**F2**	**F#2**	**G2**	**Ab2**
13	14	15	16	17	18	19	20	21	22	23	24
A2	**Bb2**	**B2**	**C3**	**C#3**	**D3**	**Eb3**	**E3**	**F3**	**F#3**	**G3**	**Ab3**
25	26	27	28	29	30	31	32	33	34	35	36
A3	**Bb3**	**B3**	**C4**	**C#4**	**D4**	**Eb4**	**E4**	**F4**	**F#4**	**G4**	**Ab4**
37	38	39	40	41	42	43	44	45	46	47	48
A4	**Bb4**	**B4**	**C5**	**C#5**	**D5**	**Eb5**	**E5**	**F5**	**F#5**	**G5**	**Ab5**
49	50	51	52	53	54	55	56	57	58	59	60
A5	**Bb5**	**B5**	**C6**	**C#6**	**D6**	**Eb6**	**E6**	**F6**	**F#6**	**G6**	**Ab6**

The octaves are overlapped when they stream down the screen to increase game playing complexity. The notes stream down with a BPM (beats per minute, although the number does not strictly translate to a number of beats per minute) depending on the user selected speed and is generated by beat_module that simply counts up to a counter and then advances the beat. Each song has up to 512 beats. The length of the notes correspond to the relative amount of time the note will be played within the music. For example a 50 pixel long note will be played for twice as long as a 25 pixel long note. The location of the notes on the x axis of the screen correspond to the relative pitch of the notes. For one octave (denoted by a singular color) notes on the left side of the monitor will be at a lower pitch and notes on the right side will be at a higher pitch with a gradient in between.

Once a new 36 bit chunk of music is received because of a new beat, note_generator checks if there is a note from [35:30] in the chunk. If so then the note_buffer index is incremented, which selects the note_blob. Based on the note_buffer index only one specific note_blob has its x, y, length and color updated. There is a couple cycle delay built in to accommodate looking up the correct parameters. X position is decided through the note_positions module which maps all notes 1 to 61 to x positions that are interleaved. Y position is always updated to 0. Length for the first note is [29:27] and the note_blob assigns the correct pixel length given the 3 bit length number. Color is assigned based on the octave ([1:12] white, [13:24] red, [25:36] green, [37:48] blue, [49:60] cyan, [61] yellow). Once the values have been assigned to one note_blob the next 9 bits are checked, the note_buffer index is incremented (given there is a note) and the similar note_blob assignments are made. In this way each new note is assigned to a new note_blob. There is some small possible latency between note generation in a chunk, but it is on the order ~10 cycles (which is imperceptible to the eyes and ears).

Notes move from the start of the screen down to the bottom of the screen based on the bpm. A counter counts up to bpm and when the bpm number is reached all note_blobs that have y positions that are less than 770 increment their y position by one. If 770 is exceeded the notes are held outside the screen at 780. The main output of this module is then all the note_pixels or'd together.

Audio Converter & Mixer Modules (Alejandro)



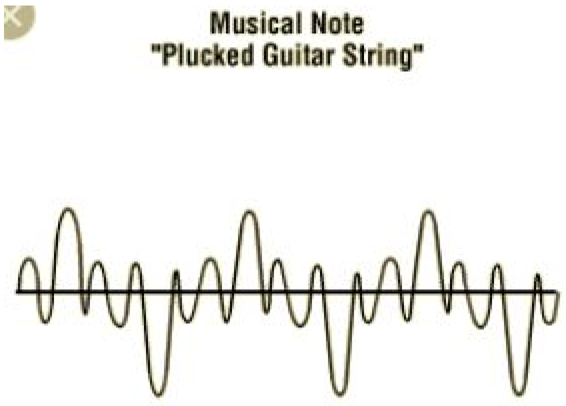
Relies on the modules from the audio lab as a base. The sine generator is used, but has a modified lookup table that contains a wave that is similar to a guitar wave. This could be changed to add different instruments such as the trumpet.

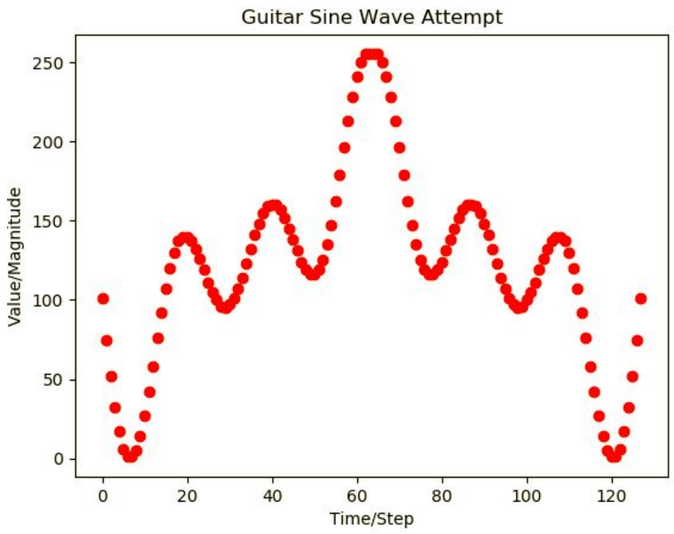
Notes are received in Audio_gen as a 61 bit array from the gameplay module that allow for 5 octaves + A6 of possible notes being played simultaneously. Each bit corresponds to a different note that should be playing. Each note has a specific phase shift that encodes the specific frequency. These notes will only be played if the specific bit in the notes array is a 1.

$$\frac{(440\ cycles/sec) * (2^{32} - 1\ phase/cycle)}{48 * 10^3 sample/sec} = 39370534\ phase/sample$$

Audio_gen passes on the notes array and a trigger for the specific 48KHz frequency (counts up to 677–that's actually a 96KHz trigger, but with the doubled sine table of 128 it works). Play_notes creates all the sine_generators. The sine generators go through the 128 value sine table at the frequency specified by the trigger with the step specified by the phase. The 8 bit from the sine generators is then sent back to the play_notes module. Play_notes loads into 8 bit variables corresponding to each note, either the signal from the sine generators or a 0. All these variables are then added together and stored in an 11 bit variable data_out which goes back to audio_gen. This signal is then scaled by the volume_control module as specified by the blue user's y position and sent out via pwm to the speakers.

An additional note is the sine wave table was updated from 64 to 128 values and uses a function that approximates a guitar audio wave (f(x) = sin(x) + sin(2x) + sin(3x) flipped about a vertical axis). There is definitely room for improvement to create a better sounding guitar-like audio wave. In retrospect I realize the signal should not go up again at both edges and should end low to more closely resemble the "Plucked Guitar String" audio shape.





Implementation Process

The object tracking and the game were developed separately. During development, Sarah displayed the output of her tracking module on a VGA display for visual feedback for testing, while Alejandro used the FPGA buttons to control the positions of the paddles (which would ultimately be controlled by the tracked lights). The only communication between the two parts of the project are the tracked coordinates as well as signals representing whether each of the 4 colored lights are present in the scene. We integrated for testing purposes two weeks before the project was due to make sure we didn't encounter any major surprises, then continued working separately until a few days before the project was due, when we did our final integration.

Object tracking (Sarah)

For object tracking, development happened in stages according to the chronological flow of bits through the system, and visual feedback was used to fine-tune at each stage. I implemented the entire pipeline first for one color, then a second, and then all four. The first step was utilizing the starter code provided by Joe to get a live camera feed to appear on the VGA display. Next was RGB to HSV conversion, again provided by staff, but with minor edits made to align with the current Vivado divider IP. At first I attempted to rewrite this module to increase efficiency because I didn't understand that the dividers had a fixed latency, and my faster implementation was wrong; I discovered this using a test bench, which was more necessary for RGB->HSV than for other modules. To select hue thresholds for the colored objects we had, I made the lower and upper threshold values correspond to the 16 switches on the FPGA and displayed only the pixels that passed the hue thresholding on the VGA display. In this way, I was able to tune the hue thresholds using visual feedback. I did not discover until adding multiple channels how narrow the hue ranges needed to be for it to work well; they were generally about 10 apart on a scale of 0-255, and small changes in hue value made a huge difference. This was also the stage during which I experimented with and selected RGB gain values for the camera via the microcontroller; I reduced them from the default to reduce how much I picked up non-light objects, but could not reduce them too much or the camera stopped seeing color.

Once hue thresholds were selected (the first color channel implemented was blue), I implemented erosion, opting for a linear kernel both to decrease the complexity of the module and to reduce latency, and visually determined that a 3x3 linear kernel was sufficient to reduce noise for our purposes. Next implemented was centroid tracking, the output of which I displayed as crosshairs over the thresholded/eroded pixels on the VGA display. Finally, I wrote the IIR filter, which originally had a constant decay factor, but I found later that it was useful to be able to independently set the decay factors for each color channel. After the first color channel worked well, I implemented the second, and then the final two at once. The "is" signal representing the presence of an object was added on later based on conversations with Alejandro.

Along the way, I had a few "a-ha" moments with regards to pipelining mistakes and timing issues, which were difficult to spot visually on the VGA display but which became apparent as I drew out detailed block diagrams of my modules. I found these block diagrams to be the best way for me to catch signal paths with unequal numbers of registers.

Game logic, GUI, and audio (Alejandro)

Overall the game started with just some structures from the Death Star lab and slowly evolved adding on the skeleton of lab 5A (the lab I didn't do, but wish I had done considering it ended up being so relevant).

I initially had an implementation that had paddles that moved using the buttons, but it was pretty cumbersome so I soon opted for just making the paddles large enough to fit the entire screen to speed up testing of the game. I used buttons for all the features that Sarah's tracking now controls. It was initially difficult to understand how to change the 5A lab to a 65MHz clock, but that ended up being a simple parameter change. Then intersections between notes became a challenge and I spent a few hours with Diana trying to figure out what was wrong with my code (it ended up being that instead of checking intersections each frame I was checking intersections for the entirety of the 1st vsync and all the hsyncs on that line). All of the testing for different songs and speeds was done with the switches. Scoring beta testing was done on the hex display.

The next hard part was having a scrolling index for the note pixel generation and that just included some clever decisions like using 16 pixels and just continually adding to the index of 4 bits which would roll around automatically. With the note generation, pixel intersection and audio working I had to move on to making the game something playable. Once the foundations were laid out it became a more tedious process of scaling everything up to allow for 13 notes and then 61 notes and from 16 notes on screen to 32 notes. This ended up being a lot of quick python scripts to generate the code. Although tedious it was fairly straightforward. Then I began to look at other stretch goals and rework my gameplay module to have different states and get the coe files for numbers and letters. This was actually a long process because my first attempts looked terrible due to scaling issues with conversion which were eventually fixed by some manual editing of bmp files with a square brush. I also ran into errors with having too large of files. This led me to scale everything down and then scale it back up in hardware.

The final tedious process was creating more complicated music that a chromatic scale. It was difficult and tiresome to convert sheet music to the music storage I defined. I go into this in more detail in the review section as it shapes what a possible future implementation of mine would work to include.

Review and Reflections

Sarah

For object tracking, at first I attempted to track colored gloves rather than LEDs; however, I was quickly discouraged by a lack of success in this and had a moment of inspiration where I held up my phone screen to the camera and could suddenly see clearly thresholded colors. It was at this point that I decided to switch to colored lights; it is a far easier solution than attempting to use lamps to evenly light an object like a glove, even if the color is highly saturated. I definitely recommend that future students use colored LEDs instead of unlit objects.

I think the biggest learning experience for me while working on this project was thinking like a hardware engineer instead of a software engineer; i.e., thinking about pipelining a system so that bits flow through smoothly and are processed one by one. It took drawing lots of block diagrams to realize all the delicate timing issues of a long pipeline and making sure signals didn’t get ahead of one another. I also realized that BRAM can be used as an interface between clock domains, something I’d never thought about before. As far as the implementation process, I’m grateful that Alejandro and I started early and did a test integration two weeks before the project was due rather than integrate for the first time right before checkoff.

Alejandro

Some of the most interesting code was reading from the lookup table through each 9 bit segment every few couple cycles and creating note_blobs using a rotating index. Mixing initially posed a possible problem, but adding the signals together in a simple fashion, with a bit of scaling, involved proved to work very well. Some of the code was very repetitive such as anything to do with x positions of notes and detecting those intersections; declaring all variables; and instances of note_blobs and sine_generators. To mitigate this I generated some python scripts to create all these instances.

Music creating was very tiresome. Basically it was just me sitting down and looking at sheet music and trying to convert it to my format. This was complicated by the fact that notes don't stream from the top, but rather they appear with their full length and then stream down which requires a little reverse thinking for rhythms. If I were to do this again I would take a page from MIDI and just start notes and then end them at a certain point instead of specifying a fixed length. This format simply makes much more sense in many ways. What initially stopped me from doing this was that note_blobs don't work well with this schema because there would have to be some scaling or offset when the notes are generated from the top. However, one workaround would be to store a 61(possible notes) by 768(falling note distance) grid and fill it with 1's starting at the top depending on the notes being played and their respective pitches. The grid would shift down and a new 1 would fill the place of the last 1 if the note is continuing to be played, otherwise it would be filled with a 0. This would require some storage, but altogether not too much. Then there would be a module that converts this grid to actual pixels on the screen. The rest of the game would not have to be changed significantly. I would also either read MIDI format songs directly or make conversion from MIDI to our music storage format very straightforward.

Altogether I liked bringing together all the different modules. I thought it was interesting to have the note generation and the audio generation completely disjoint. The note generation happens on its own and based on hand note intersections and x position the notes are relayed to the audio generation and display. There were also some timing aspects that were fairly interesting in the note generation. It was rewarding to be able to bring together the tracking to play our game and make a final product that Sarah and I are both proud of.

Acknowledgments

We'd like to thank Joe Steinmeyer, Gim Hom, Sarah Flanagan, and the rest of the 6.111 staff for advising us on this project and somehow always being in lab to help out. We don't know how they do it. Thank you!

Thank you also to the other 6.111 students who spent long days and nights alongside us. We made a little home away from home this semester in 38-600.

Appendix

GitHub repo (Verilog files, Python)

https://github.com/mitdevelop/FPGuitAr

- Python files/scripts are in the main folder, the sv files for the project are inside FPGuitAr/project_2/project_2.srcs/sources_1/new/

Microcontroller code for camera

```
/*
  Wire - I2C Scanner
  The WeMos D1 Mini I2C bus uses pins:
  D1 = SCL
  D2 = SDA
*/

#include <Wire.h>

/*These are settings some of which have been found empirically and/or found
from random internet sites. When you see that there's a "magic" number it
isn't a magic number like in comp sci or something...it just means we have
no idea why this register value seems to help since the data sheet doesn't
give a ton of guidance.  I'm sure there's rational explanations for many of
these numbers, but sometimes I've just got bills to pay and life to live
and don't have time to figure out why. You know the deal.
*/

const byte ADDR = 0x21; //name of the camera on I2C

uint8_t settings[][2] = {
 {0x12, 0x80}, //reset
 {0xFF, 0xF0}, //delay
 {0x12, 0x14}, // COM7,     set RGB color output (QVGA and test pattern 0x6...for RGB
video 0x4)
```

```
 {0x11, 0x80}, // CLKRC     internal PLL matches input clock
 {0x0C, 0x00}, // COM3,     default settings
 {0x3E, 0x00}, // COM14,    no scaling, normal pclock
 {0x04, 0x00}, // COM1,     disable CCIR656
 {0x40, 0xd0}, //COM15,     RGB565, full output range
 {0x3a, 0x04}, //TSLB       set correct output data sequence (magic)
 {0x14, 0x18}, //COM9       MAX AGC value x4
 {0x4F, 0xB3}, //MTX1       all of these are magical matrix coefficients
 {0x50, 0xB3}, //MTX2
 {0x51, 0x00}, //MTX3
 {0x52, 0x3d}, //MTX4
 {0x53, 0xA7}, //MTX5
 {0x54, 0xE4}, //MTX6
 {0x58, 0x9E}, //MTXS
 {0x3D, 0xC0}, //COM13      sets gamma enable, does not preserve reserved bits, may be
wrong?
 {0x17, 0x14}, //HSTART     start high 8 bits
 {0x18, 0x02}, //HSTOP      stop high 8 bits //these kill the odd colored line
 {0x32, 0x80}, //HREF       edge offset
 {0x19, 0x03}, //VSTART     start high 8 bits
 {0x1A, 0x7B}, //VSTOP      stop high 8 bits
 {0x03, 0x0A}, //VREF       vsync edge offset
 {0x0F, 0x41}, //COM6       reset timings
 {0x1E, 0x00}, //MVFP       disable mirror / flip //might have magic value of 03
 {0x33, 0x0B}, //CHLF       //magic value from the internet
 {0x3C, 0x78}, //COM12      no HREF when VSYNC low
 {0x69, 0x00}, //GFIX       fix gain control
 {0x74, 0x00}, //REG74      Digital gain control
 {0xB0, 0x84}, //RSVD       magic value from the internet *required* for good color
 {0xB1, 0x0c}, //ABLC1
 {0xB2, 0x0e}, //RSVD       more magic internet values
 {0xB3, 0x80}, //THL_ST
 //begin mystery scaling numbers. Thanks, internet!
 {0x70, 0x3a},
 {0x71, 0x35},
 {0x72, 0x11},
 {0x73, 0xf0},
 {0xa2, 0x02},
 //gamma curve values
 {0x7a, 0x20},
 {0x7b, 0x10},
 {0x7c, 0x1e},
 {0x7d, 0x35},
 {0x7e, 0x5a},
 {0x7f, 0x69},
 {0x80, 0x76},
 {0x81, 0x80},
 {0x82, 0x88},
 {0x83, 0x8f},
 {0x84, 0x96},
 {0x85, 0xa3},
 {0x86, 0xaf},
 {0x87, 0xc4},
 {0x88, 0xd7},
 {0x89, 0xe8},
 //WB Stuff (new stuff!!!!)
 {0x00, 0x00}, //set gain reg to 0 for AGC
 {0x01, 0x32}, //blue gain (default 80)
 {0x02, 0x32}, //red gain (default 80)
 {0x6a, 0x32}, //green gain (default not sure!)
 {0x13, 0x00}, //disable all automatic features!! (including automatic white balance)
};
uint8_t output_state;

void setup()
{
 Wire.begin();
 Serial.begin(115200);
 Serial.println("Starting");
 delay(1000);
 Wire.beginTransmission(ADDR);
 Wire.write(0x0A);
 Wire.requestFrom(ADDR, 2);
 byte LSB = Wire.read();
 byte MSB = Wire.read();
 uint16_t val = ((MSB << 8) | LSB);
 Wire.endTransmission();
 Serial.println(val);
 for (int i = 0; i < sizeof(settings) / 2; i++) {
   Wire.beginTransmission(ADDR);
   Wire.write(settings[i][0]);
   Wire.write(settings[i][1]);
   //    Wire.write(RegValues[i][1]);
   //    Wire.write(RegValues[i][2]);
   Wire.endTransmission();
 }
 //  Wire.write(0x12);
 //  Wire.write(0x4);
 Serial.println("OV7670 Setup Done");
 pinMode(4, INPUT_PULLUP);
 output_state = 0;
}

void loop()
{


}



void writeByte(uint8_t target_reg, uint8_t val) {
 Wire.beginTransmission(ADDR);
 Wire.write(target_reg);
 Wire.write(val);
 Wire.endTransmission();
}

void readBytes(uint8_t target_reg, uint8_t* val_out, uint8_t num_bytes) {
 Wire.beginTransmission(ADDR);
 Wire.write(target_reg);
 Wire.requestFrom(ADDR, num_bytes);
 uint8_t* ptr_to_out;
 ptr_to_out = val_out;
 for (int i = 0; i < num_bytes; i++) {
   *ptr_to_out = Wire.read();
   ptr_to_out++;
 }
}
```