Field Piano Gate Array

1st Lucy Cai
Massachusetts Institute of Technology
Cambridge, MA, USA
jcai1@mit.edu

2nd Emily Zhang
Massachusetts Institute of Technology
Cambridge, MA, USA
emilysz@mit.edu

3rd Nicholas Ouyang
Massachusetts Institute of Technology
Cambridge, MA, USA
nyouyang@mit.edu

***Abstract*—We propose a Field Piano Gate Array (FPGA), that will track the positions of three human fingers relative to a printed piano keyboard, with the FPGA playing the respective chords as fingers touch the "piano keys". The system will utilize two FPGA boards that each have a camera attachment, where we call one of the FPGA's the "top camera" and the other as the "side camera". The "top camera" FPGA will be responsible for keeping track of the position of each finger with respect to the note it is above. The "side camera" FPGA will be responsible for keeping track of whether or not the fingers are making contact with the keyboard. In the case the fingers are making contact with the keyboard, a chord of three notes will be transmitted through a speaker connected to the audio jack of the "side camera" FPGA.**

***Index Terms*—digital systems, field programmable gate array.**

I. Physical Construction

Our final design will consist of:

- An FPGA with a camera attachment, which we call the top camera
- A second FPGA with a camera attachment and an audio device connected to its headphone jack, which we call the side camera
- Four wires connecting the two FPGA's to allow for SPI communication from the top camera to the side camera

For the system to work as desired, we require the following:

- A neatly printed picture of an octave of a piano keyboard
- Three pink colored gloves which fit on three fingers of the hand
- Correctly positioned "top camera" and "side camera" FPGA's

The whole setup is captured in Figure 1. The bottom left corner houses the side camera FPGA and a speaker connected to it. The printout of the piano keyboard lies directly under the top camera FPGA, oriented upside down as we flip the camera inputs. Both FPGA's are using cameras to detect the location of the pink foam that is lying on the keyboard printout. The FPGA's are communicating using the four physical wires between them.

II. Image Processing

Our first task is to detect key boundaries given an image of a piano octave. To do this, we utilize the sobel X and Y filters to give us the horizontal and vertical key boundaries.

A. Sobel Masking

First, we run the sobel X and Y filtered camera input through a sobel masking module, which, given a pixel, first converts it from RGB to YCrCb color. Then, if the luminance Y of the pixel is greater than the threshold $10'b0\ 000\ 111\ 111$, the pixel gets mapped to 1. Otherwise, it gets mapped to 0. We chose this threshold to most clearly match the luminance of the piano key boundaries, so that background noise is optimally reduced. See an example of the output of this module in Figure 2.

Upon reset, the user turns on sw[8], which captures two static images, one of the masked sobel X image and one of the masked sobel Y image. These two images get saved to two different BRAMs on the top camera FPGA, from which we then run an image processing algorithm to detect the placement of the lines.

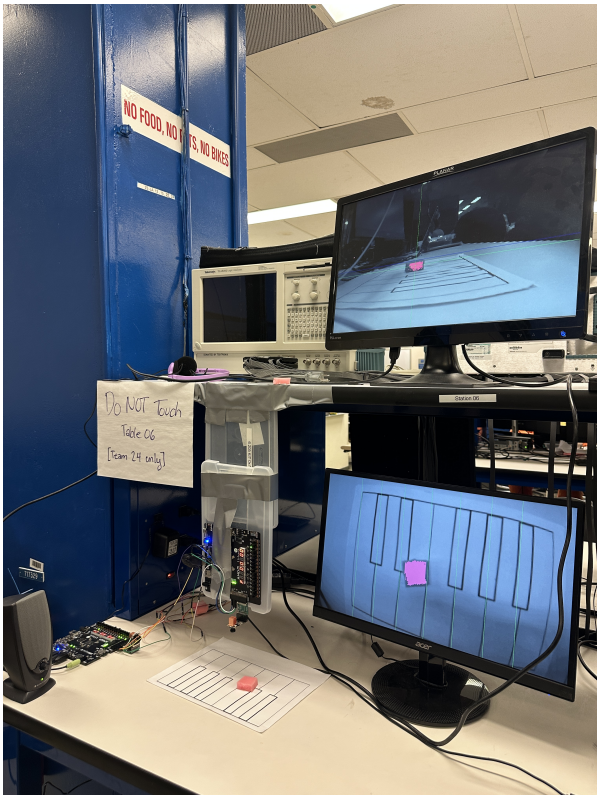

Fig. 1. Entire setup of our system with both FPGA's.

B. Detecting Key Boundaries

The algorithm works as follows. For the sobel X masked image, there should be 13 vertical lines in the upper half of the piano, demarcating the 7 white notes and 5 black notes. There should be 8 vertical lines in the bottom half of the piano, demarcating the 7 white notes. However, each line is not necessarily one white pixel wide – most of the time, each line may be 2 or 3 consecutive pixels wide. We scan through each row of the image, counting the number of white "clusters" encountered by counting the number of times where the previous pixel is 0 and the current pixel is 1. If this is true, we save the pixel's hcount coordinate in an unpacked array. At the end of each row, if the number of clusters encountered is 13, we output the 13 saved hcount values as our top 13 x coordinates. If the number of clusters encountered is 8, we output the 8 saved hcount values as our bottom 8 x coordinates.

For the sobel Y masked image, there should be 2 longer horizontal lines demarcating the top and bottom of the keyboard, and 5 short horizontal lines in the center demarcating the bottoms of the 5 black keys. We scan through each column of the image, counting again the number of white clusters encountered. At the end of each column, if the number of clusters encountered is 2, we output the 2 saved vcount values as our top/bottom boundaries. If the number of clusters encountered is 3, we output the 3 saved vcount values as our top/bottom and black key boundaries.

We acknowledge several limitations to this algorithm. The first is that it is not completely tolerant to background noise. For example, if there is an extra white pixel from a background object that coincides in the same row as a hole in a key boundary line, the resulting key boundaries will be computed incorrectly. However, one of the requirements of our setup is that the top camera is pointed directly above the piano octave, with no other objects in view. Given this, it is rare that extra white pixels will appear in the image, and also rare that masked key boundary lines will have holes.

The second limitation is that the key boundary lines are not completely straight– each one has a slight curve to it, so that the hcount of the top of a vertical key boundary may be several pixels off from the hcount of its center. To mitigate this, we chose to use a camera with the least fisheye lens effect. If, once integrating everything together, we still find this to be an issue, we may consider extending our algorithm to add an averaging step. For example, with the vertical key boundaries, instead of taking the first row with 13 clusters to be the key boundaries (which will occur near the very top of the piano), we can keep track of every row with 13 clusters, and average their results.

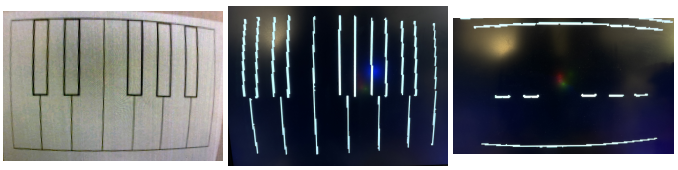

Fig. 2. Raw camera input, masked sobel x filter, and masked sobel y filter.

C. Key Association

Once we have the hcount and vcount values of our key boundaries, we need to associate each pixel in the camera input with a specific piano note. Note that we assume the piano paper does not move after sw[8] is flipped, so that our key boundaries and associations remain continually valid. This module takes in an x_com and y_com corresponding to some pixel, does computations based on the previously computed key boundaries, and outputs the piano note corresponding to the pixel's position. We encode piano notes in decimal and allow each to be 4 bits, so that an invalid note (outside the keyboard) is 0, C is 1, C sharp is 2, etc, and finally B is 12.

III. Center of Mass Tracking

Our next task is to track the positions of the three fingers playing the piano. We require the user to wear three pink glove tips on their thumb, middle, and pinky fingers, and adjust our Cr mask to optimally detect only the pink pixels.

A. k-Means

We first give our motivation for using the k-means algorithms and we follow with an introduction of our implementation of the k-means algorithm.

To track the center of mass of the pink glove tip on each of the three fingers, we propose the usage of the k-means clustering algorithm with $k = 3$. This choice is motivated by two choices. The first is the spatial distribution of the finger tips, where we expect them to be clearly separated from each other so there are distinct clusters present. The second is the fact that when the hand is moving, the new centroids will be very close to the previous centroids designated by the algorithm.

Our k-means algorithm works by guessing initial locations of centroids for each cluster, and then iteratively converging upon the actual centroid of each cluster. Formally, let $\mathcal{X} = \{x_1, \ldots, x_n\}$ represent the set of n 2-dimension data points, each representing a tuple of the x and y coordinates of one of the masked (pink) pixels.

The algorithm begins by initializing some centroids μ_1, μ_2, μ_3. Each centroid represents the center of mass of each distinct cluster within $\mathcal{X}$, where a cluster is loosely defined as points that are close together but far from other points. These centroids are simply the centroids determined by the algorithm in the immediately previous run of the algorithm. In the case that there are no previous runs of the algorithm (we have just booted our design up), then the centroids are assigned such that they split the space into halves vertically and fourths horizontally. Explicitly,

$$\mu_1 = [320, 360]$$
$$\mu_2 = [640, 360]$$
$$\mu_3 = [960, 360]$$

The next step of the algorithm is to assign each point $x \in \mathcal{X}$ to one of the centroids, using the rule

$$\text{ClusterAssignment}\,(x) = \underset{i \in \{1,2,3\}}{\arg\min} \|x - \mu_i\|_1$$

Note that we are using Manhattan distance (ℓ_1 norm) as our distance metric instead of the classical Euclidean distance, since it is a heuristic that sacrifices a minimal amount of accuracy for a great increase in speed.

The final step of the algorithm is to recompute the centroids of each cluster. This is calculated by a simple center of mass averaging given by

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

where C_i represents the set of points contained within cluster i. Essentially, we are averaging the x and y coordinate values for each point within a cluster, for all clusters.

In our case, we only run a single iteration of the algorithm, going through the steps of assignment and recalculation of the centroids only once for every fresh input of $\mathcal{X}$. This choice was supported by the fact that the fingers cannot possibly move far from it's previous position, relative to the clock speed of our system, so convergence should be able to be achieved in only one iteration if the algorithm is given access to the immediately prior centroids.

Realize that this implementation is scalable and allows us the freedom to set k to any number in the set $\{1, 2, 3, 4, 5\}$ for finger detection, if we so desire.

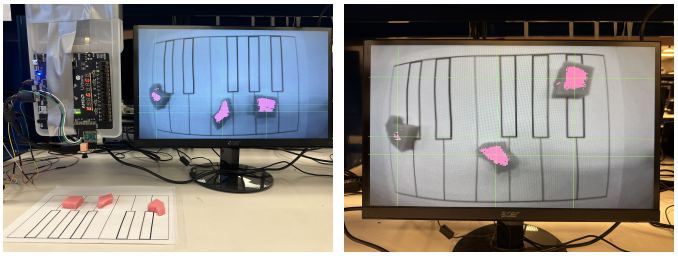

Fig. 3. Three note detection via k-means.

B. Side Camera Center of Mass

We used the algorithm implemented during Lab 5 center of mass calculation of a Cr mask to and lower and upper threshold values of A0 to F0 to optimally track only the pink pixels. We used the pink foam of the FPGA kit, cut into small cubes, to tape onto the user's fingers to be tracked. To see how the center of mass is being tracked, sw[6] can be utilized to see green crosshairs that track the center of mass. To compare it to the threshold line, the user can switch on both sw[6] and sw[7] to make it easiest to view.

IV. Audio Generation

A. The Physical Layer

We will utilize pulse-width modulation (PWM) to enable audio signal playback through the headphone jack present on the FPGA. Our design utilizes a 100 MHz system clock. The audio will be 8-bit audio, which ensures a balance between audio fidelity and space utilization on the BRAM of the FPGA.

B. Sine Wave Generation

We use the notion of direct digital synthesis (DDS) [1] to construct our audio signals. Since we are using 8-bit audio, a $2^8 = 256$ entry sine wave look-up table (LUT) is precomputed using a Python script. Since we require non-negative integer values for usage in our digital design, the ith entry of the LUT is explicitly designated as follows:

$$\text{LUT}[i] = \frac{\sin(\frac{2\pi i}{256}) + 1}{2} \cdot 255 \tag{1}$$

Let us examine what is exactly happening in (1). The $\sin(\frac{2\pi i}{256})$ simply computes the ith value of the sine function over the range of 256 equally spaced values on $[-1, 1]$. Adding 1 to this term shifts our range to $[0, 2]$. Similarly, dividing by 2 then shifts our range to $[0, 1]$. We are left with the problem of converting our values to integers. We rectify this problem with a multiplication by 255, since 8-bit values are represented by numbers from 0 to 255, inclusive.

We generate our notes by indexing into a singular LUT. To do this, we utilize a phase accumulator for each individual note. The phase accumulator is 32-bit which ensures good resolution of lookup index without much drawback on storage. Note that since our audio is 8-bit, we index into the LUT with the 8 MSB of the phase accumulator at each cycle.

Each cycle, the phase accumulator is incremented by a frequency tuning word (FTW), where each musical note in the octave is differentiated by a unique FTW. In essence, the FTW allows the phase accumulator to cycle through the fixed LUT at a rate such that it produces an audio signal at the exact frequency of the note the FTW corresponds to. The FTW values for each note are derived as follows, where f_N denotes the frequency of the musical note:

$$\text{FTW} = \frac{2^{32} \cdot f_N}{100 \cdot 10^6} \tag{2}$$

The 2^{32} term follows from the maximal value of a 32-bit phase accumulator, and the $100 \cdot 10^6$ term follows from the clock speed of the system (100 MHz). The FTW values are precomputed using another Python script and mapped to their corresponding notes.

C. Chord Mixing

In order to combine three notes into a chord audio signal, the sine wave amplitudes of each of the three notes at each cycle must be summed. To prevent distortion from integer overflow, the amplitudes are divided by 4 prior to summing, which guarantees no overflow on the basis that $\frac{3}{4}N \leq N$. This also reduces the volume of the audio to a tolerable level.

D. Pulse-Width Modulation

Given the final mixed chord value which we wish to output as audio, we pass it through a PWM module to convert the sine-wave signal to a square-wave signal. This signal is then passed to the digital-to-analog converter present on the FPGA in the form of a headphone jack, from where the audio is then output.

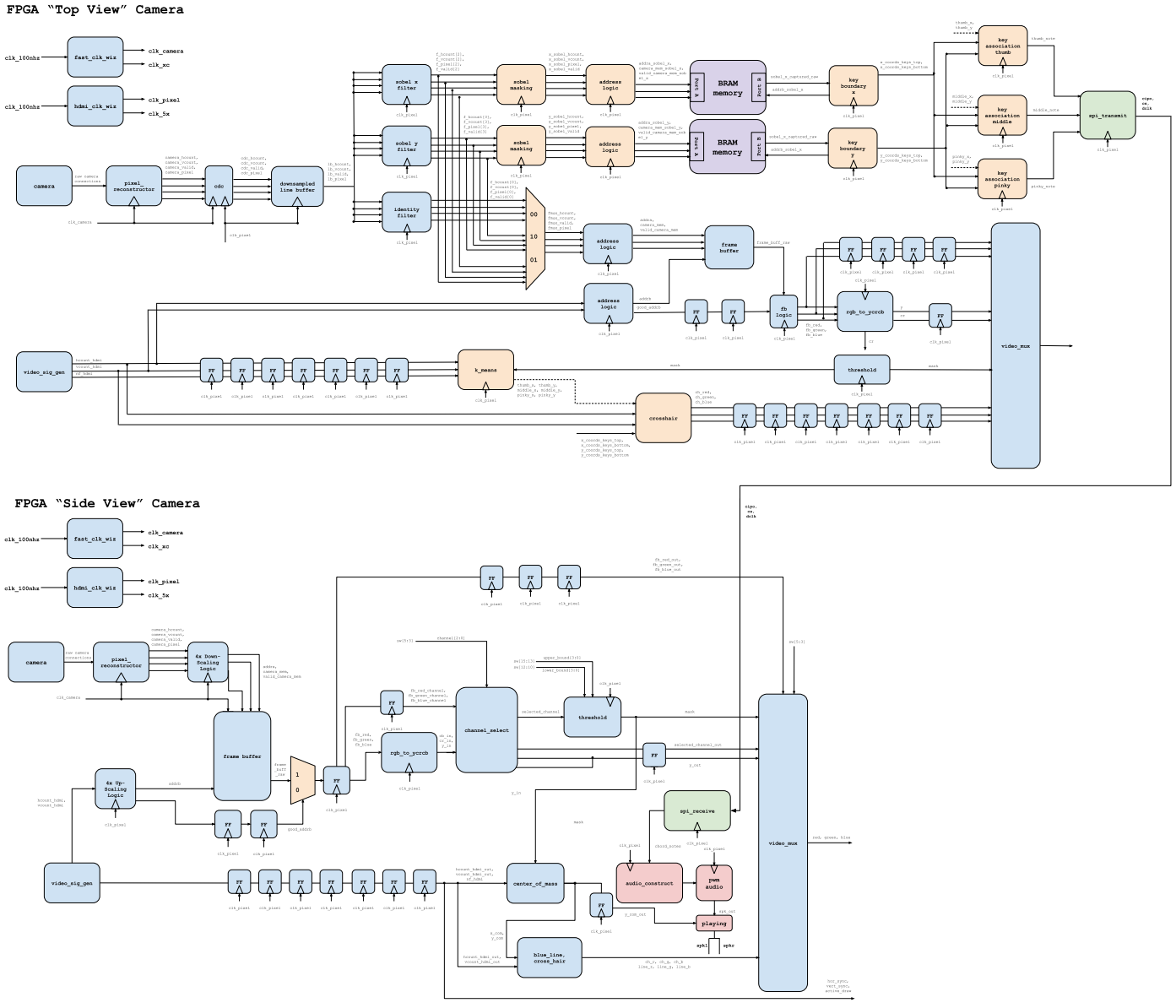

Fig. 4. Block diagram for the entire system. Blue blocks denote modules repurposed from past labs, orange blocks denote new modules written for this project, purple blocks denote memory allocation, green blocks denote communication between the FPGAs, and red blocks are used for audio output.

V. Side Camera

A. Startup Alignment

At startup, using a HDMI screen, the user can flip sw[7] on the side FPGA to see a blue line appear on the screen. The user can place the side camera on the table and point it toward the leftmost width of the paper camera. It should be pointed in the direction such that the right hand's thumb side view is seen. The blue line should be lined with the farther edge of the piano printout. Be sure to center the camera.

We consider playing if the fingers make contact with the page. Thus this blue line is used as a reference threshold when tracking the hand's movement. The FPGA will track the center of mass of the fingers such that if the fingers fall within 10 pixels or below the blue line, it is considered "playing". If it is above this blue line, the FPGA does not consider any extraneous movement to be "playing". This threshold was determined by testing the thumb on the furthest key from the camera, which in this case is the C note. We found that the center of mass algorithm, combined with consistent camera distance, has made it easiest to track whether the user is actually playing a note on the paper given.

B. FPGA-to-FPGA Communication

We utilized Serial Peripheral Interface (SPI) to transfer bits of data from the top camera FPGA, in our case, the Controller, to the side camera FPGA, the Peripheral. The top camera will

communicate the three-note chord, each note represented by 4-bits for a 12-bit long message.

The top camera, after tracking and assigning center of mass values to the three fingers, will identify the three notes of the chord that the fingers are hovering above. Each note is encoded as described in Section 2.C, identified as the thumb, middle finger, and pinky finger and encoded in the order of thumb note, middle note, and pinky note. After assignment, SPI transmission will be triggered per 1 MHz (once each 1 million clock cycles), and using a 50% duty cycle, each bit will be held on the line for 20 clock cycles (10 cycles high and 10 cycles low). This number was determined as the clock runs at 74.25 MHz, so the transmission must finish before this maximum time. Thus, to optimize for speed as we wish to have no identifiable delay, we found that 20 clock cycles were the smallest number to quickly transfer data. The Chip Select (CS) line will be held low for the duration of the transmission, allowing the Peripheral FPGA to know when the transmission itself is occurring. New bit values will be put on the Controlled Out Peripheral In (COPI) line on the falling edge of the Data Clock (DCLK). Bits are sent out by the most significant bit (MSB) first.

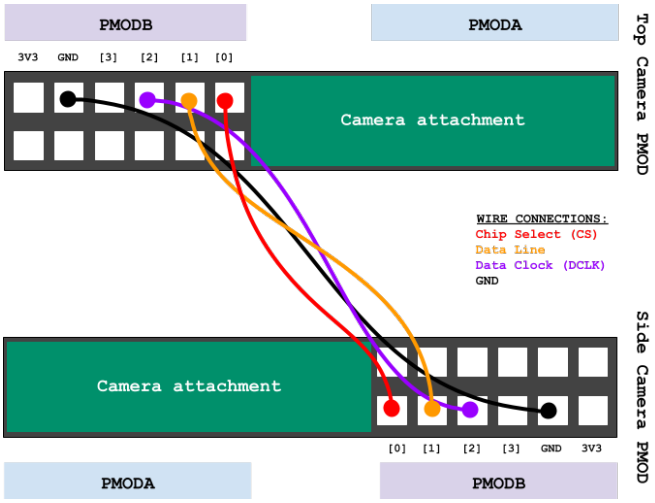

Fig. 5. PMODB diagram.

A four-wire connection is made between the top camera and side camera FPGA using PMODB connections as described in Figure 5 above. The GND pins are connected for synchronization purposes, and the additional wires to connect the output of the top camera DCLK, CS, COPI into inputs as DCLK, CS, and CIPO respectively. On the side camera FPGA side, when the CS line is pulled low, it begins to read from the transmission, sampling on the rising edge of the DCLK. Once all 12 bits are sent and sampled, the CS line should be pulled high and the side camera utilizes the values sent over the line to determine the notes played.

As we are trying to read synchronous data between the two FPGAs of real-time piano playing motion, after the two FPGAs fully process their own camera data input, there will be a delay before the chord notes are sent over from the top FPGA to the side FPGA. Each trigger of transmission is 1 MHz, so each chord data transmitted SPI is not necessarily from the newest data read. However, assuming that it is the newest data read, SPI would take at least 270 clock cycles to send over the data fully over the wires (12 bits total bits, 20 clock cycles per full duty cycle and additional duty cycles for the start and stop of transmission). There may also be additional delays with the implementation of audio as whether the notes are played out or not depends on the center of mass calculation. In that calculation, the divider has latency dependent on the numbers involved, and thus the latency cannot be determined without knowing the actual value. However, we felt that without pipelining to synchronize these two FPGAs fully, there was no discernible delay noticeable by the human ear, which was one of our main goals of the project. The advantage of the FPGAs is the speed at which all these calculations are made, so we felt additional pipelining to synchronize the two FPGAs fully was not necessary.

C. Audio Output

Transferred notes received by the side camera will be used to create chord note audio. If the center-of-mass tracking finds that the user is ”playing” the notes which is when the y center of mass goes below a threshold, the audio generated as described in Section 4 will be played out through the speaker/headphone jack of the FPGA. Otherwise, no audio is output.

VI. Evaluation

A. Timing

The worst negative slack of the top camera system is 0.587. The worst negative slack of the side camera system is 0.472. There are no strict timing requirements we must meet besides positive worst negative slack because we do not require high quality audio. There is no noticeable human delay between the states of playing and not playing and the changing of notes as you move around the FPGA which would be attributed to the speed of calculation of FPGAs and communication between the two FPGA's.

In terms of latency of the top camera FPGA, the initial frame read takes 12 cycles to save into the BRAM per pixel, and an additional 320*180 cycles to calculate the key boundaries for one frame, giving a latency of 8.53 milliseconds. In terms of the k-means algorithm, the calculation has a divider which is hard to estimate as the delay is dependent on the evaluation of the division. However, there is a 1 cycle delay to transmit to SPI and 7 cycle process to display cross-hairs on the video mux.

In terms of latency of the side camera FPGA, there is a 2 clock cycle delay from camera input into the frame buffer for pixel reconstruction and downscaling, 2 clock cycle delay for clock domain crossing between clk_pixel and clk_camera, in addition to an additional cycle to filter if the address is good, 1 clock cycle for RGB to YCrB conversion, 1 cycle

to include the threshold before outputting to the video mux, while HDMI h-count and v-count have been pipelined to be delayed by 7 clock cycles to account for this for each pixel. Additionally, if we use SPI of already received chord notes and use the current y center of mass, there is a 2 clock cycle delay between audio construction and output to the speaker. If we also wait 270 additional cycles for the SPI transmission, with the pixel clock running at 74.25MHz, the lowest latency of the side camera is 5.43 milliseconds. We acknowledge that this number may not be the most accurate due to the nature of the center of mass calculation and SPI communication.

B. Memory

The top camera FPGA utilizes a peak memory of 3744 MB, and the side camera FPGA utilizes a peak memory of 2927 MB. For the top camera in terms of BRAM usage, we calculated 1 frame of 320 x 180 resolution of 1 bit for the thresholded initial piano input, and an additional frame of of 320 x 180 resolution of 16 bits per pixel. Thus, the total bit usage is 1036800 bits stored in the BRAM of the FPGA. For the side camera FPGA, we calculated 1 frame of 320x180 resolution of 16 bits per pixel to be stored per frame plus an audio sine lookup table with 2^8 resolution audio at 2^7 sampling rate, totaling to 954368 bits of BRAM. Both of these values are below the 2.7Mbits of BRAM storage in the FPGAs. We thought about optimizing the memory further by compressing the sine wave into $1/4$th of its original size, since it stores all the necessary information in the uncompressed wave, by symmetry, however, in terms of BRAM usage we felt we were under the maximum and felt that our goals of processing speak and audio quality were both met without this change.

C. Use Cases

Our design handles the following use case: detect chords that fingers are hovering above and detect if the chord is actively being played using "top" and "side" cameras. If it is, play out audio of the corresponding key notes of three-finger chord.

This use case fits precisely the ideal goal that we set for this project. It is possible to modify our design to accommodate the use case of playing four or five finger chords. This can be done by changing the value of k in our k-means algorithm, which was designed to be easily modifiable.

VII. Reflections

In hindsight, we learned a few things that would have been helpful to know prior to starting the project.

- Camera distortion is real. Even though we positioned the top camera perfectly perpendicular to the printed keyboard below, there is still extremely noticeable warping that can be observed near the sides of the video. The design of the key boundary detection being done by the top camera works well in theory, but the distortion of the camera lens makes it hard to detect the correct boundaries of the keys. Given more time, it would perhaps be a useful investment to create a custom piano keyboard with customized spacing between distinct key boundaries depending on how far the boundaries are from the center of the keyboard. In this way, the spacings would be approximately equivalent.
- Correct pipelining matters. We had some trouble with a noticeable offset of the key boundaries being detected, which was constant for each boundary. Our initial thought was that it must have been due to incorrect pipelining of the system, perhaps because the assignment of the boundaries was occurring some number of cycles after it was "truly" detected. The same issue was observed with the mapping of the intervals of boundaries being mapped to incorrect notes, which was also resolved by fixing the pipelining of the system contained within the top camera FPGA.
- The pink foam is not detected perfectly by the cameras. The lighting was sometimes dim enough due to shadows that the pink would change shades and not be easy to detect by our cameras. This might have also been an effect of the camera quality. We could have considered tracking a different color but we were unsure if other colors would have been any easier to track than the "6.2050 pink" so we just stuck with it.

VIII. Appendix

A. Contributions

Lucy was responsible for implementing the top camera sobel filtering of the printed keyboard, mapping finger placement to the correct corresponding note being played, and the k-means algorithm. Emily was responsible for implementing the side camera logic necessary to determine if notes are actively being played and the transmission of information between the two FPGA's. Nick was responsible for implementing the chord audio generation and the k-means algorithm.

All three authors contributed equally to the writing of the report.

B. Acknowledgments

We thank Joe Steinmeyer for the camera, filtering, and top-level boilerplate code, as well as providing all the necessary hardware. We also thank TA's Kiran, Jan, and Andrew for their help on debugging and walking us through the concepts of DDS.

References

[1] J. Tierney, B. Rader, and C. M. Gold, "A Digital Frequency Synthesizer," *IEEE Transactions on Audio and Electroacoustics*, vol. 19, no. 1, pp. 48–57, March 1971.