The Soul Reader - 6.205 Final Report

Nick Hougardy
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: nickhoug@mit.edu

***Abstract*—In Texas Hold'em, when a player correctly identifies the other player's card with minimal information, it is typically called a soul read and is widely regarded as being an amazing player. Using this name as inspiration, the goal of this project is to implement a playing card reader through an FPGA and camera module. The FPGA will determine the card and communicate what card it is through the seven segment display on the FPGA. To do this, it will be determining the edges of the card, finding the top left hand corner, and performing a XOR the pixels with a predefined kernel.**

I. Introduction & Motivations

This project aims to identify a player's hole cards (the two cards a player holds face down) using the FPGA and camera module given to us in Lab 3. There are some open source software's that do this in Python on a Raspberry Pi, but the camera output is around 5 frames a second and processing the data is quite challenging. This process could be streamlined and quickened by using specialized hardware meant for the job. Once the card has been identified, it will be outputted on the seven segment displayed using characters defined in Figure 2.

The inspiration of this project came from me not completing it's predecessor over the summer. As said above, I could never get the image recognition working on the Raspberry Pi, but I built out the hardware (woodworking, routing, upholstery, etc.) before working on the software (Figures 1 and 2). Thus, I have a smart poker table just waiting for electronics.

II. Memory

The memory requirements are as follows: after the frame has been grabbed from the camera, it will then be transformed from a 16-bit RGB frame with size 240x320 to a 1-bit frame with size 240x320 (76800 pixels). From this frame, the corners of the card will be found and the hardware will grab the pixels from the top-left corner to preform a XOR operation on. Since there are 17 kernels, 13 with size 28x40 (1120 pixels) and 4 with size 28x29 (812 pixels), there will be 17 corresponding BRAM modules of the same size that will store the corner pixels. In total, the BRAM used is 112416 bits (14052 bytes), which is much less than what is available on the FPGA.

Fig. 1. Original Poker Table

Fig. 2. Reupholstered Poker Table

Fig. 3. Seven Segment Display Suits and Ranks

III. Image Processing

Figures 4 - 8 show the general steps that the FPGA takes to determine the card's suit and rank. The code can be found here: Nick Hougardy's GitHub .

A. Grayscale and Masking

From Figure 4 to Figure 5, this is where the FPGA does the threshold processing. The "threshold.sv" module takes in a pixel, converts it to Luminance by averaging each of the pixels evenly $\left(\mathrm{L} = \frac{\mathrm{R+G+B}}{3}\right)$, and then masks the bit to either 255 or 0 using a mask parameter fed into the module (Figure 6). In this example, the images were masked using a parameter of 210 and one place this project can go is cycling through a set of mask parameters and choosing the best one for the image; this is typically called adaptive thresholding and is supposed to be a useful technique when dealing with non-uniform backgrounds. Since the cards were tested on a dark black background, this wasn't needed. For the purposes of this project, the mask was able to be controlled using 8 of the FPGA's switches and could cycle through all values from 0 to 255.

B. Center-of-Mass Algorithm

From Figure 6 to Figure 7, the FPGA then performs a center-of-mass calculation to approximate the center of the card. The purpose of this is to grab a point within the card versus being no where near the card. This is needed for the edge detection algorithm to work. However, most of these cards are not symmetrical. Some are symmetrical across the vertical axis, like the Ace of Hearts, and some are symmetrical across the horizontal axis, like the Ace of Diamonds. But, most of these cards do not exhibit these properties. Because of this, one side of the card is weighted more than the other and causes the center-of-mass calculation to be incorrect and slightly askew. Even for the example image in Figure 7, the heart in the center has more black pixels on the top half of the card than the bottom half just because of the shape of the heart, meaning that more white pixels contribute to the bottom half, dragging the center of mass down. However, despite being slightly incorrect, it still yields a pixel within the center of the card 100% of the time and works for this application.

C. Finding the Edges

Using the Center of Mass, the "find_edges.sv" module grabs the pixels located underneath the pink cross-hair in Figure 7,

Fig. 4. Down-sampled (5:6:5)

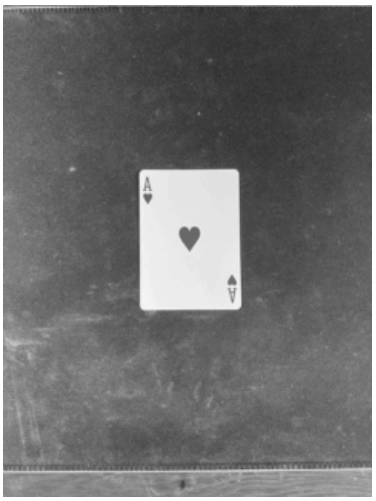

Fig. 5. RGB to Luminance

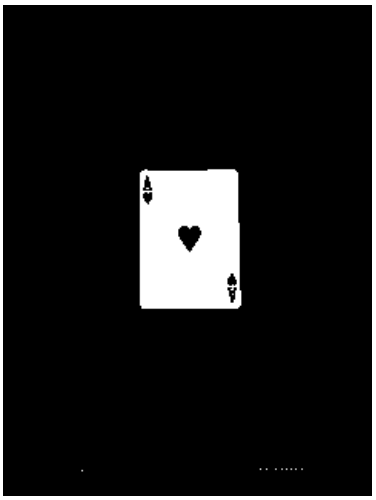

Fig. 6. Thresholded

stores them in a cache, and determines where the edges of the card are. This goes under the assumption that the card is perfectly straight and has no rotation. Using this algorithm, the edges are able to be found as seen in Figure 8.

D. Zooming into the Corner

Once the edges have been found, this also means the coordinates for the top left corner have also been found. Using this index, pixels are read from one BRAM into another using the hcount and vcount generated by the "VGA_gen.sv" module. If the pixels are apart of the corner, then they are stored. If not, they just pass through without any interaction

E. XOR Module

Once the top left hand corner of the card is detected, we will then perform a XOR operation on it with kernels stored on the FPGA. These kernels symbolize the 4 suits and 13 ranks (17 kernels total). Once this is done, we can add up the total pixels in the frame (White = 1, Black = 0). The one with the best score (closest to 0) gets outputted. The reason for this scheme is a XOR gate is a difference operation and the FPGA has a surplus of XOR gates. In order to prevent problems later on, it would benefit the project to plan on using XOR gates from the beginning to design the scoring scheme. Figure 9 is an example of how this would work. XORing our image with the correct kernel outputs all black pixels with a score of 0. When we XOR it with a worse fitting kernel, the score rises, telling the FPGA that the two images are dissimilar.

IV. Conclusions

In conclusion, the card reader works well. In fact, better than I expected. When determining the rank of the card, it is accurate near 100% of the time. Issues are outlined below in more detail, but to get this high accuracy, the card needs to be played with a bit to get it in just the right rotation to be read.

However, when determining suit, the card detector is correct about 50% of the time. Even at max zoom, the camera is such a low resolution that the input image of the top left corner is only 28x69 pixels. Without the rank, the suit is only 28x29 pixels. Given this, small changes in size are not clear enough once pipelined through the camera to have a distinguishable score. XORing a heart with the spade gives a score very close in magnitude when XORing a heart with a heart. Further work needs to be done in this area and further solutions are outlined below.

As for responsiveness, the difference is very clear. Working with the Raspberry Pi as before is much slower than the FPGA. As the card is getting in the right rotation, the FPGA picks it up nearly instantaneously and this is because of how the system is designed. Since the corner is being XORed against all of the kernels simutaneously, the process is quicker by a minimum factor of 17x.

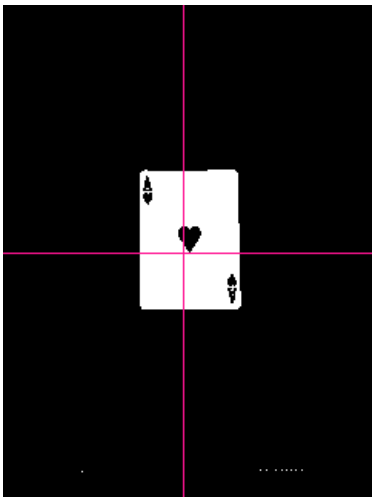

Fig. 7. Center of Mass

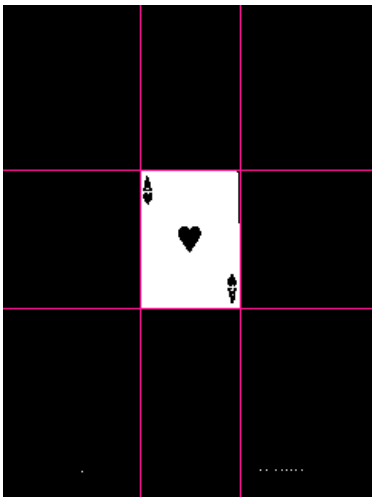

Fig. 8. Edges

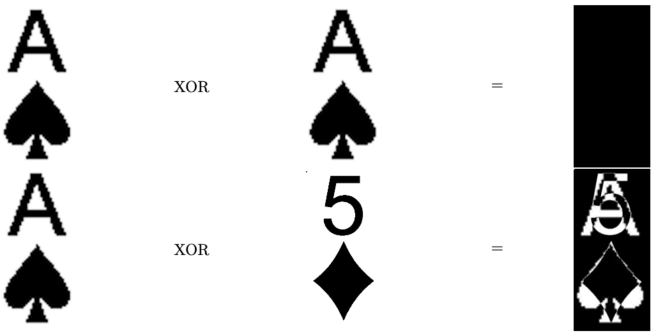

Fig. 9. XORing scheme

V. Block Diagram

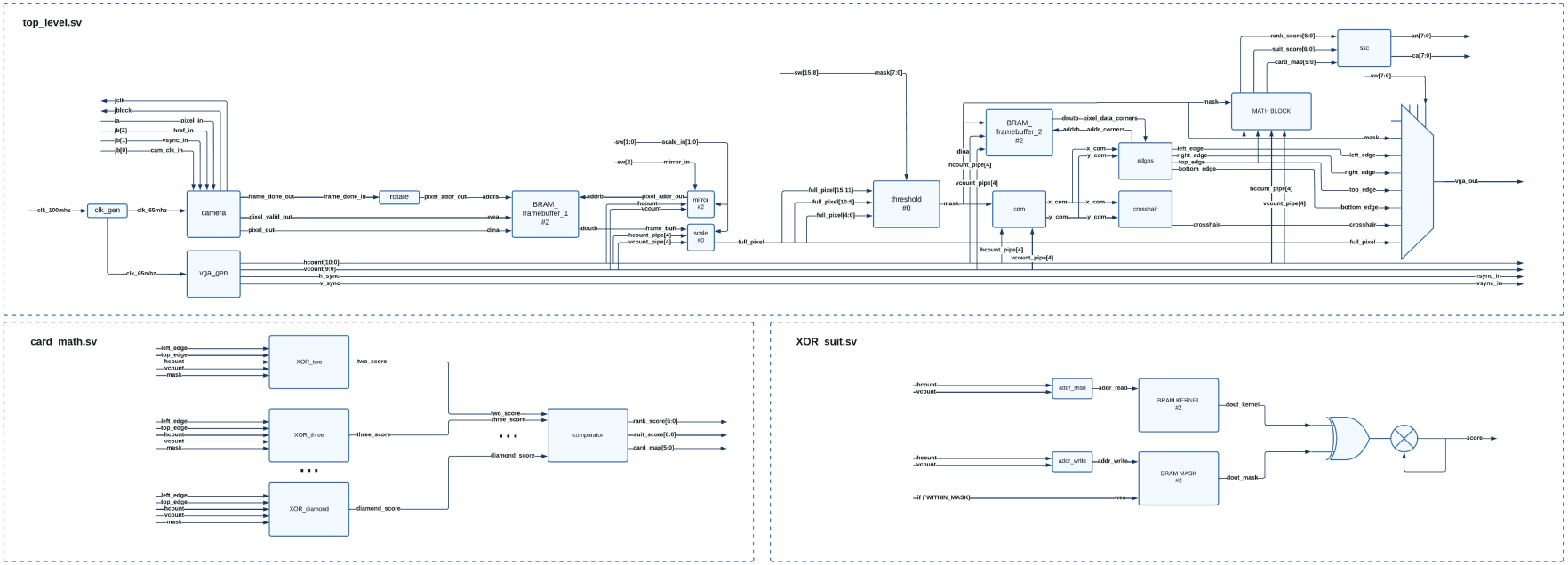

Fig. 10. Block Diagram

VI. Potential Improvements

A. Averaging frames

Greatly cutting down on the memory usage means that in order to improve the accuracy of the image recognition, we can take several frames and average them together, effectively forming a low-pass filter across the pixels. This would just serve as a redundancy and one of the issues with the main card reader is that it will recognize the correct suit/rank, and then flicker. Averaging several frames would reduce this high frequency flicker.

B. Using Weights and Colors

Specifically for suit, the card recognizer has a tough time with hearts and spades. Given that our camera won't get any better than it currently is, using color in conjunction with the kernels could nudge the device into determining which suit is correct.

Given that train of thought, the system as it stands is a "One layer linear neural net with no weights". The first ever neural net was used to recognize numbers and implementing what was learned into this project would be a good idea. To give an example, the card recognizer would often have trouble with 5's and 7's, which makes sense. Given the top half of each number, they are very similar. But, the bottom halves are wildly different. Splitting the kernel into sixteenths or even fourths and assigning different weights to each would greatly increase the performance of the device.

C. Communication over UART

One of the stretch goals for the project that didn't get done was using the card recognizer to detect the cards, but then communicating with a python script over UART to compute the odds of a specific set of two cards winning the hand. Implementing the UART protocol over USB seems straightforward and would be a nice addition to the project.

D. Rotation

A greedy algorithm to implement is to calculate the edges and locate the suit and rank. If nothing gets detected, then the starting image gets rotated and the process repeats until either the whole card has been rotated or the suit and rank have been found. Finding rotation would help eliminate the tediousness of finding the "perfect spot" for the card recognizer to work and would make the system more robust as a whole.

E. Dealing with Zoom

As it stands, the camera is a fixed distance away from the table where the cards are displayed. Using this fixed zoom level, I was able to determine the precise corner size and hard coded it into Verilog. Everything up until this point, the thresholding, the center of mass module, and the edge detection, is not hard coded and is not dependent on zoom. The main issue at hand is with the XORing scheme, both the corner pixels stored in the BRAM and the kernel stored initially need to be the same size. This way, each pixel is mapped to another and everything is accounted for. Dealing with scaling and determining a method for making both images the same size would solve this issue, but would take a considerable amount of thought. From a practical standpoint, if the camera does make it into the poker table, it will always be a fixed distance from the cards, but it would be better to make the system more robust in general.

What's nice about the cards used (found here) is that the horizontal distance of the corner compared to the horizontal distance of the card is $\frac{1}{7}$ and the vertical distance of the corner compared to the vertical distance of the card is $\frac{1}{4}$ (Figures 11 and 12). No doubt this has to do with some gambling regulation, but it makes the math straightforward on the FPGA and would be used to determine corner size given a variable zoom.

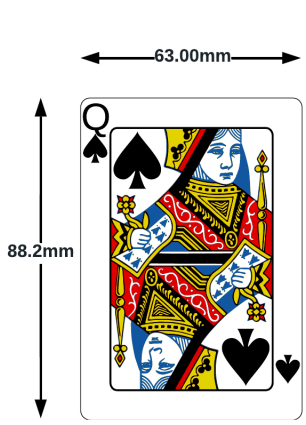

Fig. 11. Card Dimensions

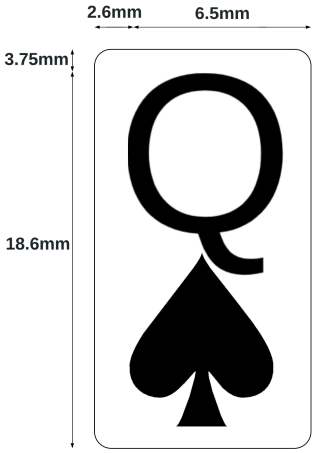

Fig. 12. Zoomed in Dimensions

Fig. 13. Live Demo with Cards shown