

FPGA Air Brush

6.111 Project Report

Oscar A. Guevara
Junior Neeranartvong

1. Table of Contents

1. Table of Contents	P.1
2. Introduction	P.2
3. Preparation & Motivation	P.3
4. <i>Air Brush</i> Manual	P.4
5. Technical Overview	P.6
6. Changes to Proposal	P.7
7. Hardware Choices	P.8
8. Block Diagram	P.10
9. Project Implementation	P.11
A. MicroSD Interface	P.21
B. Further Expansion	P.29
C. Timeline	P.30
D. Conclusion	P.31
E. Resources	P.33
F. Photo Gallery	P.34

2. Introduction

An artist requires three basic things in order to create. A platform, a means to change it, and inspiration to do it. Air Brush attempts to package all three requirements into a convenient and easy-to-use drawing application based on the Nexys 4 FPGA platform. The innovations? Through our application you draw via camera enabled motion-tracking, it features an easy-to-use graphic interface, and provides support for saving and loading your drawings. Through a blend of distinct technologies, we've developed a feature-rich innovative drawing toolbox with high potential for future expansion.

3. Preparation & Motivation

Professor Hom's meet-and-greet session was instrumental to the formation of our team. After a long conversation regarding the types of projects which we could possibly do, we realized we would be able to work together efficiently.

The team's initial meeting was held to discuss the core idea for our project. This not only was beneficial for getting us on the same page, but the ideas we brought to the table were scrapped in favor of new ones developed through in-person discussion. Our finalized idea *FPGA Air Brush* was a the result of pitching each other idealized goals for this project and refining the level of risk we were willing to take given the possible rewards.

The main motivation for *Air Brush* was a desire to build an interactive system. This opened the door to discussion on visual recognition systems as well as sensor based inputs. One important consideration was having freedom of movement at all times, so an implementation with tethered wires was out of the question. This limited the number of sensory devices we could use, or increased the size of them due to added space for wireless communication hardware. We thus decided to go with visual recognition, and the "Technical Overview" section discusses how we moved from gesture recognition to color detection.

The drawing application idea was the end result of a nostalgic appreciation of classic software. Initially, our idea was to recreate a game such as Tetris or Duck Hunt in the FPGA. However while the idea was interesting enough, we wanted something that would appeal to a larger audience and not get tiresome quickly. In the end our hand gestures, describing how this games might operate, produced the idea of a drawing application akin to MS Paint. The application was elegant in it's usefulness, and it contained a level of difficulty that we were looking for as well as a wide area in which to expand if needed.

Follow Up meetings were held to discuss module breakdown and graphic user interface design choices. These meetings were very informal, but proved important in keeping us on schedule. They also served as times to make necessary changes to our project and clarify points which we might have initially overlooked. These meetings, again, were essential to the development of our application.

4. *Air Brush* Manual (I/O)

I.) Considerations

Due to the applications reliance on visual input, for best results be sure to face the camera in a direction with no red or blue objects. In particular, be sure to not wear blue or red when using this application. Though inconvenient, the first phase of development is still a work in progress.

II.) The Glove (Input)

Start by setting the battery pack power slide-button to the on position. This switch is located on the pack itself, next to the outgoing wires.

Next, on the embedded gemma microcontroller, set the power slide-button to the on position. This should trigger a small on-chip green LED, as well as the main input (blue) LED located on the tip of the middle finger. This LED will serve as the input for your screen cursor and drawing tool.

Finally, in order to trigger the ‘click’ response, the glove has two copper contact points located between the thumb and index finger. Pressing the thumb against your index finger gently should trigger two input (red) LEDs on the tips of the ring and index fingers. These LEDs will serve as a binary selection input (click), on or off. Keep the LED on in order to draw or click, and turn it off to move on screen freely.

III.) The User Interface (Output)

The user interface contains six main features: Draw, Move, Change Color, Change Thickness, Clear, Save, and Load.

Startup: Be sure to hit the reset button (Center push button) located on Nexys 4, after first loading up the program for the first time. This ensures camera is configured properly to accept input, and eliminates any initial noise.

The Draw and Move features are closely related. If within the drawing area, defined by a white rectangle, the cursor will either move freely or draw using the current color and thickness which are visible to the user on the lower-right corner of the application. The application will draw if the user has selection triggered using the glove, otherwise you move freely.

Each of the other four features can be accessed in a similar way. Hover over any of the on-screen 'buttons' with the cursor and use the glove 'click' feature to make a selection. For example, clicking on the "Save 1" button using this procedure will store the current drawing 'as is' to the mounted microSD card.

IV) Feature Description

Change Color - Select color by clicking on desired color block (30 possible colors).

Change Thickness - Iterate by clicking the (+) or (-) until satisfied.

Clear - Selecting this option clears the screen. Cannot be undone.

Save 1 - Saves current image 'as-is' to microSD memory location 1.

Save 2 - Saves current image 'as-is' to microSD memory location 2.

Load 1 - Loads current image 'as-is' from microSD memory location 1.

Load 2 - Loads current image 'as-is' from microSD memory location 2.

5. Technical Overview

Air Brush project hopes to provide a base platform for FPGA computer-human interaction design. One of the key features of this project is its single-point tracking system. This implementation proves that high-fidelity tracking can be achieved with minimal processing on raw video input. This development was then used in combination with a smooth-line drawing algorithm to produce fluid motion tracking and continuous drawing.

A second key feature of this project is data storage and retrieval. Specifically for *Air Brush*, the focus was placed on saving and loading 640x480 pixel images for a total of 307,200 bytes. This data transfer was implemented between internal BRAM on the Nexys 4 FPGA, and an external microSD card. By thinking of this feature in terms of bytes, it becomes clear that this particular component of the project has the potential for adoption into a large number of other FPGA projects.

Once it became apparent that hand gesture recognition was out of the scope of the available timeline, the user interface was redefined to accept user selections through a ‘click’ based system, where the click was a second input (LED) received by the camera. This third key feature, derived from the first feature mentioned above, allowed for the redesigning and expansion of the different drawing parameter that could be made available to the end user. It informed the way in which our GUI was implemented, as well as the particular inputs to the camera we needed to track.

Other features, though not completely innovative, but nonetheless highly important to the success of the overall project include modifications to the provided camera input module and BRAM to VGA module.

In summary, *Air Brush* brings together distinct technical features that function in parallel to create an innovative take on personal art. However, the implications of this project go far beyond the artistic realm and may serve as the foundation of a whole range and different caliber of projects in the future.

6. Changes to Proposal

The initial proposal was limited by the team's early understanding of the constraints and available feature of the Nexys 4 FPGA. Through first-hand experience, we gained awareness of a few changes to our original design which we deemed necessary.

Foremost among these changes was the elimination of hand gesture as a viable user input given our timeline. Our final implementation relies the same principles as this original idea; mapping visual cues to on-screen application parameters. Due to the image-processing complexity of hand detection, a full project of it's own accord, we opted for color detection and tracking instead. The structure of our project allows us to go back and revisit a hand gesture implementation at a later date with minimal difficulty.

A secondary major change consisted in redefining the resolution of our application's user interface. Originally, the proposal called for a 640x480 display with and embedded 320x240 drawing area. Because we began exploring all our different module components using 640x480 resolutions, to achieve the proposed resolution we would have to downsample each module's output. This was fixed instead, by making the VGA display resolution 800x600 and the drawing area 640x480. This meant that not only did we have a higher resolution application, but we only had to make the change to a single module. The original proposed resolution was a conservative estimate of what we could display given our hardware constraints, however with better understanding of this components we were able to safely implement the needed changes.

A few other, smaller-scale, changes were made as the project went along. However, the overall proposal remains true to the original goals of the team.

7. Hardware Choices

I.) FPGA Platform

There were two serious FPGA contenders for the *Air Brush* application. The Nexys 4 or the 6.111 labkit.

The labkit comes with a database of relevant code from previous projects which might have been useful. It contains a good amount of memory, and a decent number of I/O peripherals. Most importantly, it comes with self-creating test benches for code verification. This FPGA came with pre-written code for interfacing the NTSC camera.

The Nexys is a newer platform with a larger amount of internal storage as well as the option for additional external storage. It also contains higher clock speeds. This FPGA also came with pre-written code for interfacing the OV7670 camera.

In the end, it came down to memory and the supported camera. Our application is memory intensive, and our desired features required external storage. The OV7670 offered us greater control of our raw input feed. It simply made more sense to go with the Nexys 4 for this project.

II.) VGA Interface

VGA is a common display standard which the team was familiar with from a previous 6.111 lab. For debugging purposes this was one of the top choices as VGA compatible screens are readily available everywhere. The resolution choice is talked more in depth under the ‘Changes to Proposal’ section.

III.) OV7670 Camera

This choice was made primarily due to seeing the two camera contenders in action during in-class demonstrations. The OV7670 camera offered a larger amount of features, including pre-processing, in a smaller package than its competitor, the NTSC camera.

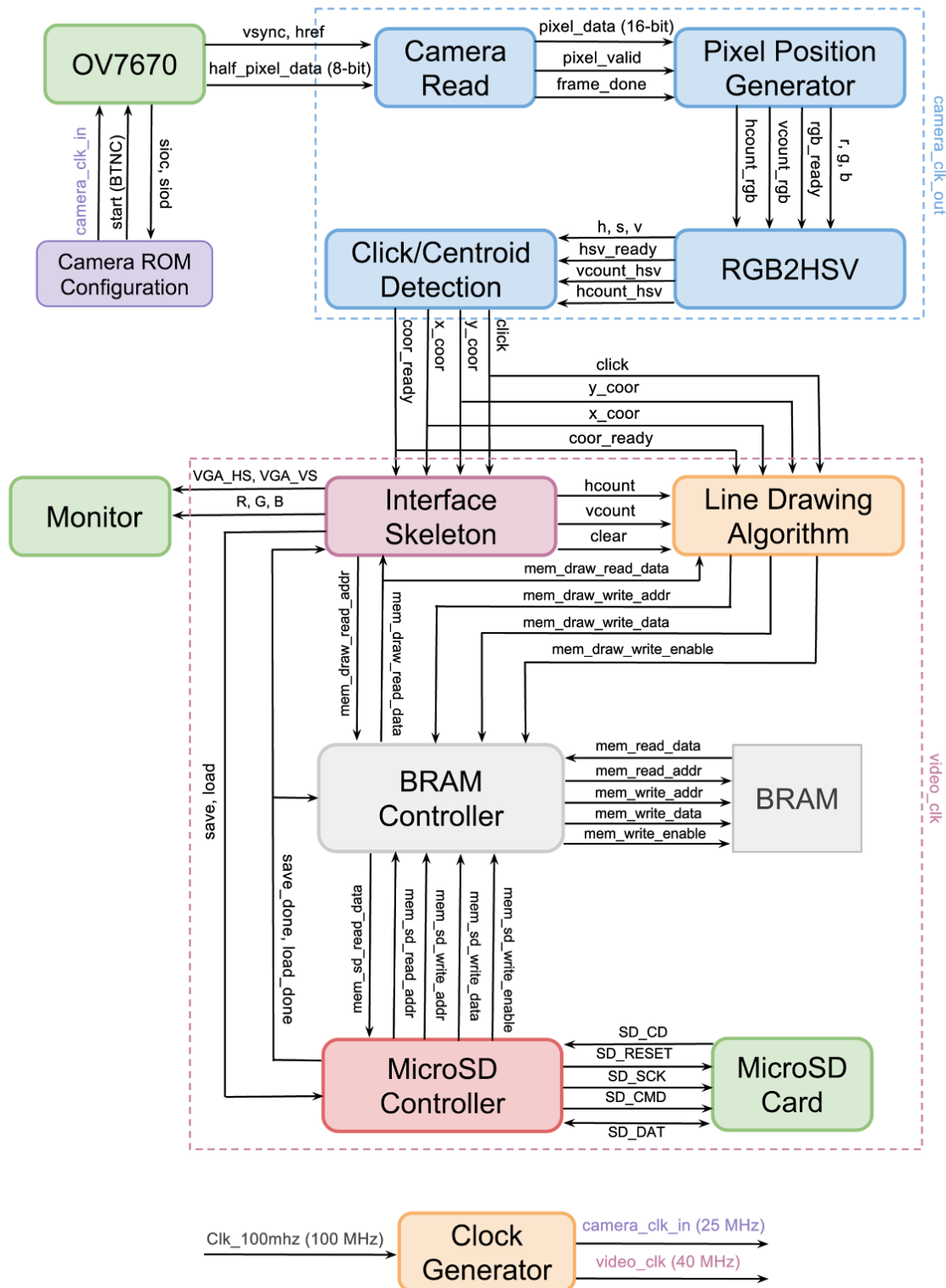
IV.) SPI Communication Interface

Since our external memory was a microSD card, this was the only viable option possible.

V.) Gemma & LED Inputs

For a minimal input interface, we used conductive thread to hack a simple circuit using a Gemma MC to drive three LEDs. This were used as user input.

8. Block Diagram



9. Project Implementation

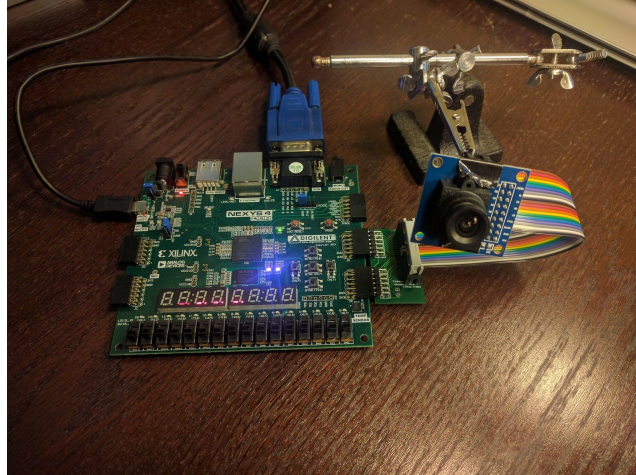
I.) Glove and LED



On the glove, we have installed three LEDs on each finger: one blue LED on the middle finger and two red LEDs on the index and ring fingers. The blue LED is always on and is intended to track a cursor on the screen. We have made an open circuit for the red LEDs and put a copper conductive tape at side areas of the index finger and the thumb. The red LED circuit will be closed if and only if the index finger and the thumb are closer together (when the copper tapes of both sides contact), resulting in that the red LEDs are on. We intend to use the red LEDs to determine whether we are going to click or draw on the screen. Since we want the cursor to be able to reach at the edge of the screen and draw at the same time, we installed 2 red LEDs to the left and right of the blue LED.

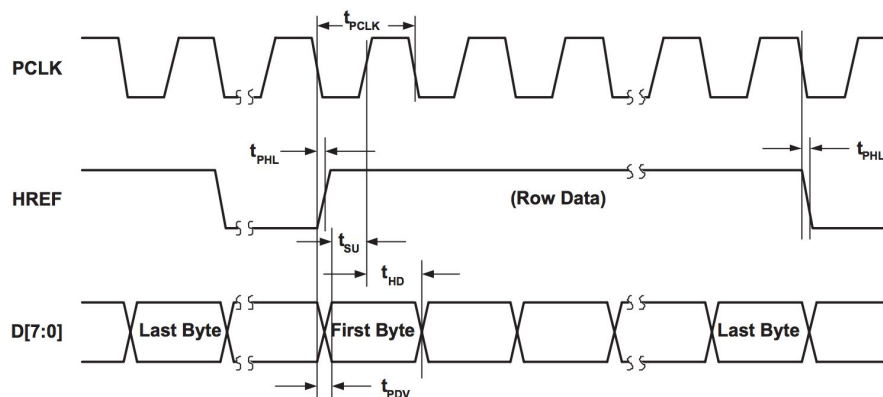
II.) Camera and Image Feeding

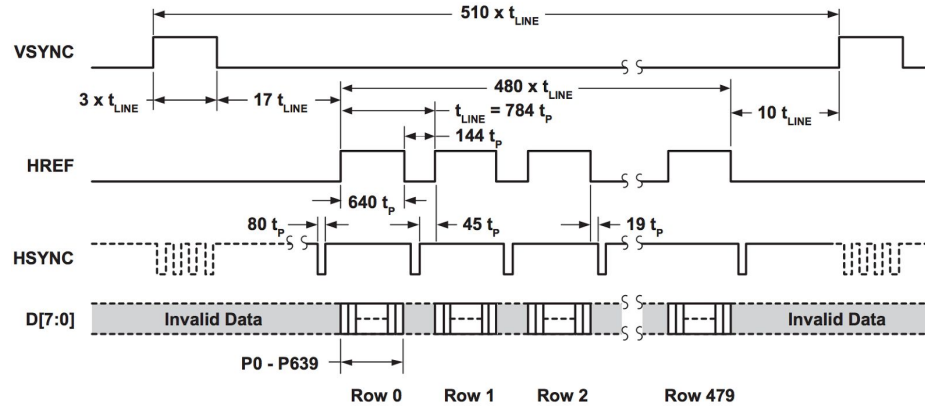
We connected the OV7670 camera to 2 PMOD connectors (JA and JB) on the Nexys 4 board. This connection is not direct, instead it requires pull up resistors at the SIOC and SIOD outputs from the camera. To communicate with the camera, we use the Serial Camera Control Bus (SCCB) interface. Setting up this SCCB communication is no easy task but fortunately, Weston Brawn, our TA, had developed tools (both Verilog codes and hardware) to deal with the camera communication on both sides.



In order to receive desired data from the camera, we need to configure its ROM/registers. We will instruct the camera to send each pixel in the format of RGB565 at a resolution of 640×480 (VGA) and a framerate of 30fps. We also need the camera to send each the frame from right to left, and vice versa. By looking at the addresses on the camera datasheet (<http://www.voti.nl/docs/OV7670.pdf>), we will be able to configure the camera's ROM and get the desired output data.

Note that in order to achieve the frame rate of 30fps, we generate a clock at speed 25MHz from the Nexys 4 and send to the input of the camera. The camera will send its own clock PCLK, VSYNC to determine whether the frame starts, HSYNC to determine whether the row starts, HREF to determine which parts of the row give valid data, and D[7:0] for 8-bit data. Since the data for each pixel will be RGB565, we need 2 bytes, or equivalently 2 PCLK cycles to get each pixel. We have attached images from the camera datasheet (<http://www.voti.nl/docs/OV7670.pdf>) describing the signal schemes below.

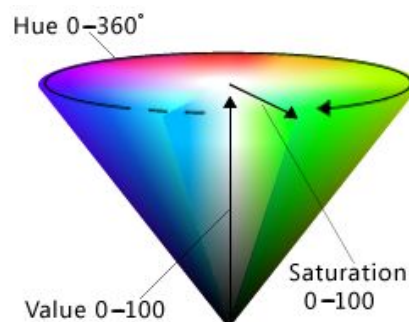




The next task is to transform the signals we get from the camera into a form that we can work with. After each high VSYNC, we know that a new frame is sent. Based on HREF, we can keep track on the positions on the image frame 640×480 the camera is sending with the pixel color data. In other words, we can keep track of hcount, vcount, and 2-byte RGB565 color data corresponding to the position/pixel on the image frame. Note that hcount is the position of a pixel in the row, counting from 0 to 639 from left to right of the frame, and vcount is the position of the pixel in the column, counting from 0 to 479 from top to bottom of the frame. We also output the pixel_ready variable to know when the 2 bytes data is completely fetched. We decide to pass the data to the next module and will not store an image frame into any BRAM due to Nexys 4 memory limitation of around 4.7 Megabits used by the drawing frame.

III.) RGB Pixels to HSV Pixels

Our goal is to get an accurate position of the blue LED color as computered by the camera. We know that in order to detect the blue color using RGB, we would have to set a threshold interval (rectangular box on 3D) on RGB axes, however this not a trivial task. After further research, we found that HSV (Hue-Saturation-Value) color is easier to work with. Basic visualization of HSV colors is shown below.



(Image from <https://i-msdn.sec.s-msft.com/dynimg/IC726004.png>)

By working with HSV, we were able to set the threshold on the blue color more easily using only hue value. Note that the white color is viewed the same by all hue values. We want to avoid the white color by setting the threshold for saturation at some minimum value. The same reasoning applies for the darkness value of each color, so we also set that to some minimum value. Values were determined empirically.

We use the LEDs as they can be easily distinguished from the background which is usually darker. The tradeoff is that the center of the LED is viewed as white by the camera, instead of the blue/red color we see with our eyes. However, the brightness is decreased as the distance from the center of the LED is higher, and at some radius, the camera will recognize blue/red color at the circumference of the LED. By knowing the circumference of a circle centered at the LED, we can compute the center position by performing a centroid calculation. This will be discussed later.

We have modified RGB2HSV module on the class website. The module requires the use of IP Divider, which can be generated by Divider Generator on Vivado. The original module takes 22 clock cycles (3 clock cycles for setting up variables, 18 clock cycles for the divider, and 1 clock cycle for concluding the result) to compute HSV value from RGB value. Since one pixel corresponds to 2 clock cycles, we have modified the module so that the computation is held when the full 2 bytes pixel are ready, we instead used 26 clock cycles (3×2 clock cycles for setting up variables, 18 clock cycles for the divider, and 1×2 clock cycle for concluding the result) to generate HSV pixel value. Note that each of H, S, and V is 8-bit. In order that the output HSV pixel matches the position on the frame, we have to delay hcount, vcount, and pixel_ready by 26 clock cycles.

IV.) Click and Centroid Detection

From the last module, we have HSV pixel data and its corresponding corrected hcount, vcount, and pixel_ready. For each frame, we would like to determine whether the red LEDs appear on the frame and extract the center coordinate of the blue LED.

To determine whether the pixel is red or blue based on HSV color. We set the following threshold for blue and red colors.

Blue	iff	$(144 < H < 176)$	and	$(S > 50)$	and	$(V > 80)$
Red	iff	$(5 < H < 10)$	and	$(S > 50)$	and	$(V > 130)$

To determine whether the red LEDs appear on the frame, we first count the number of red pixels falling into the specified range. We end counting when the frame is done (when hcount = 639 and vcount = 479). Since the frame seen by camera is not perfect and noisy, we set the decision rule so that if we see the number of red pixels more than 25, we decide that the red LEDs appear on the frame, meaning that the user wants to

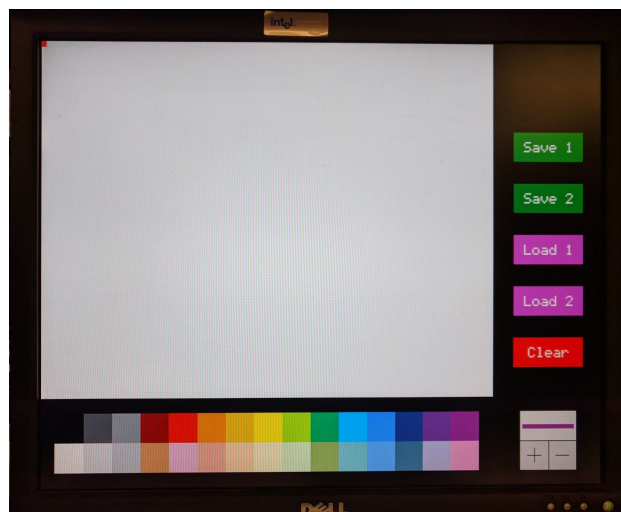
click. The click signal will be sent to the next module at the appropriate delay (to match with the clock cycle required by detecting blue centroid).

Next, we are going to determine the centroid of the blue pixels. For each frame, as the pixel HSV color data and its corresponding hcount and vcount come in, we actively sum hcount and vcount separately, and count the number of blue pixels on the screen. After the frame is done ((when hcount = 639 and vcount = 479), we send the sum of hcount as a numerator and the number of blue pixels as a denominator to the IP divider module generated by Divider Generator, and wait for 30 clock cycles to get the resulting quotient, which will be the mean of the hcount or the x-coordinate position of the blue LED. The same procedures are applied to the sum of vcount and the number of blue pixels and give the y-coordinate position of the blue LED. Since the frame is noisy, to achieve drawing smoothness, we will instead output the average of the last 8 coordinates to the next module.

In summary, now we have a binary value telling whether the user wants to click as well as a (smooth) x-coordinate and y-coordinate representing the blue LED position.

V.) Graphical Interface (Paint Logic) - Part I

The interface is what the user sees displayed on the monitor. We aimed to have the white drawing area of size 640×480 at the upper left of the screen. Therefore, we chose the output resolution of 800×600 (SVGA) at a refresh rate of 60Hz for the monitor. Therefore, our video clock will be 40 MHz. We will use the rest of the screen for color pad, thickness control, and save/load/clear buttons shown in the figure below.



Based on the x-coordinate and y-coordinate (in the frame of 640×480) of the hand positions, we can map the hand positions so that the user can reach 800×600 screen by

multiplying each coordinates by $5/4$ and rounding to the nearest integer. Note that the closest integer of a float number can be found by taking a floor function on that number plus 0.5. Therefore, we map the coordinate x to $x + (x+2) \gg 2$ and y to $y + (y+2) \gg 2$.

If the user wants to change the color, he/she can hover the cursor over one of the colors and click to select the desired color. The selected color will be shown on the thickness box. We store the current color in a register.

If the user wants to change the thickness of the line, he/she can hover the cursor over the plus or minus buttons and click to change it. When increasing/decreasing the thickness, note that we designed this feature such that the user cannot hold the click button to continuously change the thickness. Instead, the user needs to click and unclick to change one level of thickness. The current thickness is visualized as the thickness of the line in the thickness box and as the size of the square cursor.

Let's come back to the output to the screen part. In this module, we have to generate our own $hcount$, $vcount$, and their corresponding $hsync$, $vsync$, and blank signals to be the monitor input. What we have left is the pixel RGB444 data. For each pair of $(hcount, vcount)$, we first determine whether the position falls into the drawing area, color pad, thickness control, or save/load/clear buttons.

We will store a drawing image data into the BRAM. The BRAM can be created by the Block Memory Generator in Vivado. We use simple 2-port BRAM for our application (one port for writing and one port for reading). More information about BRAM will be described in the next paragraph.

If the position $(hcount, vcount)$ falls into the drawing area, the module will fetch the current drawing image stored in the BRAM. Our BRAM will have 76,800 rows where each row stores the RGB332 data of 4 pixels. The top row of BRAM corresponds to the 4 horizontal bottom left pixels in the drawing image. The second top row corresponds to the 4 pixels to the right of the first 4 bottom left pixels. Each row of the drawing image takes 160 rows on the BRAM. In the BRAM, we go and store data from bottom row to top row of the image. Therefore, the last row of the BRAM will represent the 4 horizontal upper right pixels in the drawing image. The intention is to match the scheme developed for SD card module as we are going to share the same BRAM due to memory limitation. This will be repeated again for clarification in the MicroSD interface, therefore for a visualization of the BRAM memory see the next section below. Note that it takes 2 clock cycles for fetching the data from the BRAM. We will map RGB332 data to RGB444 data for each pixel. We will not output this pixel data immediately to BRAM.

The user wants to see an updated drawing on the screen and not the previous drawing stored in BRAM. Therefore, for each pixel, we need to decide whether the pixel color needs to be updated. Suppose that the user wants to draw (click = 1) and he/she moves his/her hand around. We can keep track of the last two positions and connect the line between those two coordinates. This will be handled by the Line Drawing module and will be discussed in the next module. The decision takes 26 clock cycles to complete.

Based on the Line Drawing module, we will overwrite the new pixel color data back into the BRAM. Since we store 4 pixels for each BRAM row, we will wait until the 4 pixels are present and write to the BRAM at the same time. We set the write enable signal to high only when the 4 pixels are present. The updated pixels are going to be outputted and appeared on the screen too (adding the small square cursor).

If the position (hcount, vcount) falls into the colorpad, we create a logic determining the color of that position. If the position (hcount, vcount) falls into the thickness box, we separate into 3 parts: one top rectangle and two other squares. We manually put the plus and minus signs on the box and the line with adjustable thickness on the top rectangular box.

Lastly, if the position (hcount, vcount) falls into the Save/Load/Clear button, we manually make a rule and place each character's individual pixel by pixel on the button. Note that there are much better ways to do this, such as initially storing the characters on another BRAM. However, we did not need many characters and this brute approach saved the valuable time that a full fledged solution might have taken..

At this point, we are almost ready to send the signals to the screen. One reminder is that we need to delay the pixel data appropriately if it does not fall into the drawing area, as the pixel output from the drawing area takes additional 26 clock cycles. There is a simple remedy for this problem. We feed all the signals into the Line Drawing module and it will automatically ignore the pixel data with out-of-range position (hcount, vcount), while still delay all the signals so that their times are matched.

Lastly, before we output the signal to the monitor, we have to add the square cursor. This can be done by a basic decision rule without spending any clock cycles. Basically, we check whether the pixel falls within the area defined by the current hand coordinate plus/minus the current thickness.

We are still not done with the interface module yet. We will talk about how the interface save/load/clear the drawing image after the Line Drawing section.

VI.) Line Drawing Module

The user will be able to draw if and only if he/she clicks and the cursor is above the drawing area. Since we need the user to draw a continuous curve, we keep track of the last two cursor positions. Our task is to connect the line between those last two cursor positions.

It is straightforward to draw the line using a ruler and a pencil on a paper. However, it is not straightforward to draw a line on the screen with finite number of pixels. We will use the famous Bresenham's line drawing algorithm to determine whether each pixel is on the line.

Suppose we would like to determine which pixels are on the line connecting (x_1, y_1) and (x_2, y_2) on the screen. Note that $(0,0)$ represents the pixel at the top left of the drawing area and $(639,479)$ represents the pixel at the bottom right of the drawing area. Without loss of generality, let $x_1 \leq x_2$. We consider the following cases and determine whether a pixel (x,y) is on the line.

Case 1: Overlapping Points $x_1 = x_2$ and $y_1 = y_2$

Change the color to the pixel iff $(x,y) = (x_1, y_1)$.

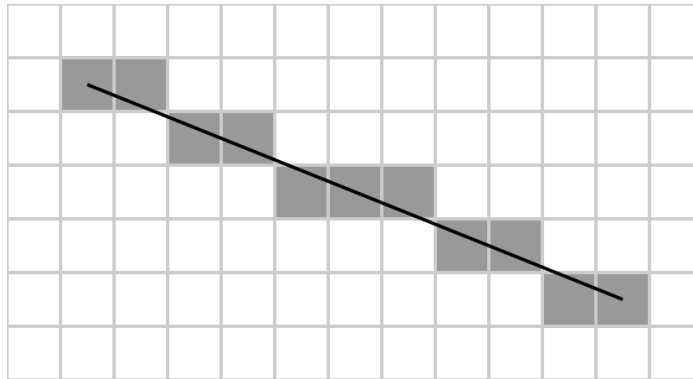
Case 2: Diagonal Line (including vertical/horizontal line) $x_1 \neq x_2$ or $y_1 \neq y_2$

Suppose that $y_1 \leq y_2$. We first see which of Δx or Δy is larger. Suppose that Δx is larger. For each x falling in the range from x_1 to x_2 , there will be only one y so that (x,y) is on the line. That y can be computed by the closest integer to $y_1 + (y_2 - y_1)(x - x_1)/(x_2 - x_1)$. Remind that the closest integer of a float number can be found by taking a floor function on that number plus 0.5. In the implementation, we can instead find the floor value of

$$y_1 + [(y_2 - y_1)(x - x_1) + (x_2 - x_1) \gg 1] / (x_2 - x_1).$$

From here, we need to use a divider module generated by the Divider Generator to compute that y . The divider takes 22 clock cycles to compute the resulting quotient y . Now we know which position (x,y) we have to change its color. If (x,y) is out of the rectangle formed by (x_1, y_1) and (x_2, y_2) , we can ignore it. If (x,y) is in the rectangle formed by (x_1, y_1) and (x_2, y_2) , we use the aforementioned procedure to determine whether the color of (x,y) should be changed. Note that it is not expensive in terms of the number of clock cycles to compute the new $y_1 + [(y_2 - y_1)(x - x_1) + (x_2 - x_1) \gg 1] / (x_2 - x_1)$ for (x,y) every time so that the decision result for all (x,y) comes out of the divider after the same clock cycles, which is easy to manipulate (all have the same delay).

Below is an example from Wikipedia where $(x_1, y_1) = (0, 0)$ and $(x_2, y_2) = (7, 5)$. We see that $\Delta x = 7$ is larger than $\Delta y = 5$. Therefore, we will go through each x-coordinate and determine which y-coordinate should be on the line. Going through x-coordinates will give us higher resolution than going through y-coordinates as $\Delta x > \Delta y$. If $\Delta y > \Delta x$, we will instead go through y-coordinates and determine which x-coordinate should be on the line.



(https://en.wikipedia.org/wiki/Bresenham's_line_algorithm#/media/File:Bresenham.svg)

The above description is only for the line thickness of 1 pixel. Now we can consider the case where the line thickness is more than 1 pixels. The rough idea is to check whether a pixel (x, y) is in/near a rectangle (x_1, y_1) and (x_2, y_2) within half of the line thickness distance. If not, we can ignore changing that pixel's color. If yes, we repeat our steps on (x, y) and check whether y is in the interval

$$y_1 + [(y_2 - y_1)(x - x_1) + (x_2 - x_1) \gg 1] / (x_2 - x_1) - (\text{thickness} \gg 1),$$

$$y_1 + [(y_2 - y_1)(x - x_1) + (x_2 - x_1) \gg 1] / (x_2 - x_1) + \text{thickness} - 1 - (\text{thickness} \gg 1)].$$

If yes, then we change the color of that (x, y) position. After the module completes the decision rule for the line drawing algorithm, it sends the new pixel information back to the interface as described.

VII.) Graphical Interface (Paint Logic) - Part II

Now, let's come back to the remaining parts of the interface. Based on the interface, the clear button is intended to clear everything from the drawing area to white. When the user clicks the clear button, it will instruct the line drawing module to instead change every pixel in the drawing area to white. The white pixel data will be sent out so that they are appeared on the screen and are written to the BRAM.

The application allows the user to save an image or load an image from two locations in the SD card. If the user clicks Load 1 button, the interface will send the signal out asking the MicroSD module to fetch the data from the sectors 0-599 of the

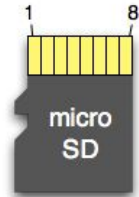
MicroSD card to be stored into the BRAM. The challenging part is that there is only one port that can read from the BRAM and only one port that can write to the BRAM accessing the same data. When the MicroSD module wants to write the data to the BRAM, the interface needs to give up the ability to write the data to the BRAM for a while, which makes sense. The interface can then regularly read the loaded data from BRAM and write to BRAM, after MicroSD module finishes loading data into the BRAM and sends the completed signal to the interface. If the user clicks Load 2 button, the same procedures apply except that the data will be fetched from the sectors 600-1199.

If the user clicks Save 1 button, the interface will instruct the MicroSD module to copy the drawing image data storing in BRAM to the MicroSD card at the sectors 0-599. The same procedures apply when the user clicks Save 2 button, except that the Sectors 600-1199 will be overwritten. Note that the interface must give up the ability to write and read when the MicroSD module is loading the data from BRAM. The interface will be in the regular mode when the MicroSD module is done. One side note is that the BRAM storing image data is shared between MicroSD module and the interface. A lot more details on the MicroSD module will be demonstrated in the MicroSD Interface section.

One cool thing is that the user can pre-load his/her image from the computer. The user can prepare 24-bit RGB bitmap image of size 640×480 pixels. He/she can run a basic python script transforming the bitmap image file into hex data (8 bit per pixel). Then he/she can use HxD, a hex editor tool that can perform raw disk editing, to copy hex data into the sectors 0-599 or 600-1199 of the SD card. In addition, after saving a drawing image to the SD card, the user can transform the hex data into regular bitmap image file by another basic python script.

A. MicroSD Interface

Note: The following implementation here was used for the storage and loading of the raw hex data of a 640x480 pixel bitmap image. Each pixel is described by a byte, of the form RRRGGGBB. The transformation from bitmap to hex, was done using a simple python script.



Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X

I.) MicroSD Overview

This section explains in detail how this project was able to achieve read and write capabilities between the Nexys 4 internal BRAM and a 2GB microSD (Secure Digital, SD) external memory. It should be mentioned that in order to interact with the SD card, strict adherence to the Serial Peripheral Interface (SPI) protocol must be followed.

SPI is a serial protocol which bridges the SD to its host, in our case the Nexys 4. This protocol has been in use for almost twenty years and its popularity is due to its reliability and high data transfer rates over short distance communications especially between embedded systems.

At a more technical level, SPI communicates using a full duplex master-slave architecture. Data is received and sent using two 8-bit shift registers due to it being serial communication, this means that in order to store and load in memory, data is processed a single bit at a time, for each byte, in each sector given, for the entire data available. The SD card initialization is a very specific sequence of commands and thus can be abstracted away into a controller module. The two other main commands we care about, read and write, are also managed by this controller.

In order to take advantage of the SPI protocol, we did just that and implemented the *SD Controller* module. This module is responsible for asserting the correctness of the the SPI protocol, specifically it allows for the reading and writing of 512-byte long blocks of memory located on the SD card. We write only in 512-byte blocks, referred to as sectors, due to the underlying physical NAND memory implementation inside the SD

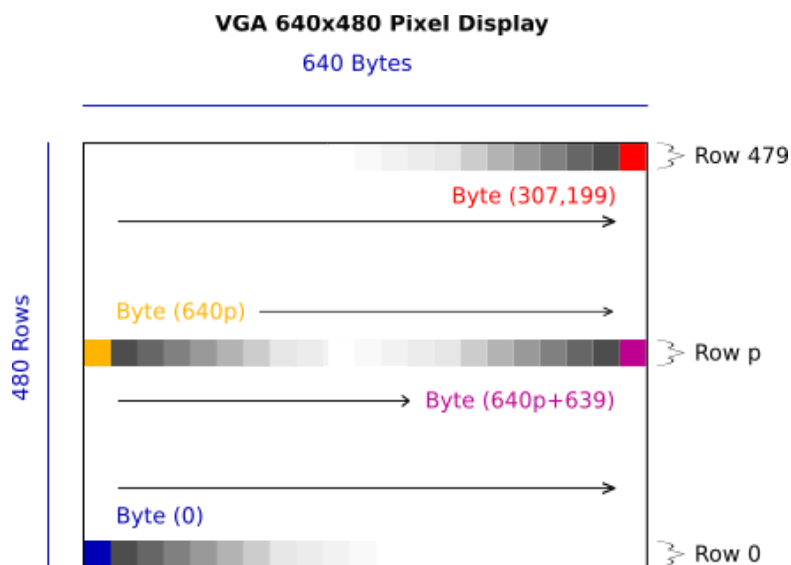
card. Essentially, the module ensures that data is transferred correctly by meeting the time constraints and sending the right sequential commands to keep proper data flow.

This sequential set of commands are 7-byte long words. Byte 0 is a series of 8 '1' bits, nothing is mapped there. Byte 1, makes use of the previous sequence of '1' bits to distinguish the command. It starts with bit sequence '01' to signal the initiation of the command, the remaining 6 bits encode the actual command. Bytes 2-5 contain the 32-bit address to the SD card memory location. Byte 6, again, is simply a series of 8 '1' bits to signify the end of the 7 byte command frame .

II.) Design Choices and Memory Addressing

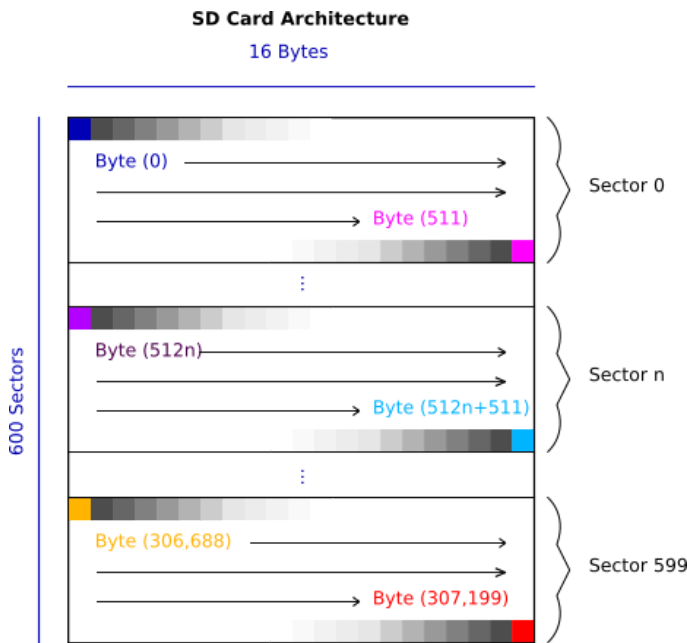
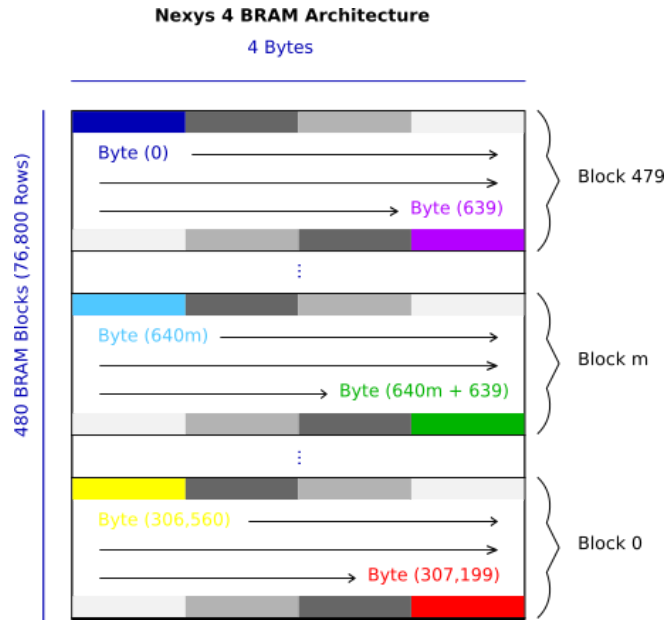
The above overview describes the controller module, this satisfies one side of the data transfer contract. The second part involves producing the correct sequential memory address to be given to the SD card.

However, in order to successfully implement the second part of the contract, the design choice for three different memory representations must be finalized. That is, how data is displayed via the VGA monitor, how data is stored in BRAM, and finally how data is stored in SD memory. Keep in mind that all code was implemented for a 640x480 display, where we define each pixel by a single byte (RRRGGGBB). The following diagrams thus will refer to pixels as bytes. Keep in mind that the same principles can be applied to other dimensions, by modifying the transformations.



Each frame of this project contains 307,200 pixels (640x480). For this project, we define the 0th byte as the lowermost-left byte (blue). By default then, the 307,199th byte is then located in the uppermost-right byte. These definitions could have been defined differently, but for this project are as shown.

The Nexys 4 internal memory is called BRAM. It is broken down into blocks. Using the integrated IP catalogue in Vivado, a BRAM of 32 bits (4 bytes) by 76,800 rows (479 blocks) was created. For this project we used the Dual Port RAM configuration in order to read and write concurrently. There is a one-to-one address mapping from the previous VGA frame display to this BRAM. To represent this, blue and red colors are used once more.



The microSD card architecture is in turn broken down by sectors of 512 bytes each. For each frame (640x480 bytes) stored, 600 sectors are required. Only one frame is represented below, but more can be stored by shifting addressing by increments of 600 sectors. Again, there is a one-to-one address mapping from each of the previous diagrams to this one. Similarly, to represent this correspondence, blue and red colors are used.

By defining each object in this way, and noting the zero-indexing, we can transfer bytes sequentially through a simple linear transformation of addresses between these three well defined objects.

III.) Starting Point & Challenges

Due to its popularity, we were able to find an *SD Controller* version that implemented the read functionality for the Nexys 4 platform though with minor glitches. Namely a previous year 6.111 project, *Live-Action RC Mario Kart™*, which itself borrowed the module from Steven Merrifield's original VHDL implementation.

However, a few modifications to the *Write_Block* states in the *SD Controller* FSM had to be made to properly configure the write functionality. Namely, the signals which trigger a new byte to be read into SD memory, were triggered at incorrect times and for incorrect durations. Under the assumption that provided *SD Controller* module was correct, debugging this was an extremely time consuming experience, and the recommendation is to fully analyze the SPI protocol and either write your own module or closely examine existing versions for correctness.

On a different topic, a misunderstanding of how to properly configure the BRAM IP generator led to an initial glitch on the read module. The glitch consisted of a vertical striped pattern displayed on screen where pixels alternated between correct ones and black ones. After a couple hours of debugging, changing the parameters of the BRAM generator, and reconfiguring solved this problem.

At the end of the project, a second less noticeable glitch was found in the read module. Where non-deterministic horizontal lines of 10-15 pixels appeared on screen after a read to the SD card was performed. Because the glitch was discovered just before the presentation deadline, there was not enough time to debug the problem.

IV.) Reading from SD to BRAM Module

Assuming a verified implementation of the *SD Controller*, the *SD_to_BRAM* module is the most straightforward of the two data transfer modules.

We defined our BRAM memory above to contain 76,800 rows containing one 32-bit word each. This means that each row (word) contains 4 bytes, this is important because as was previously mentioned, one byte is equivalent to one pixel.

The way BRAM works, is as follows. BRAM takes one 4-byte word on its input port and stores it at the location specified at its address port. Simple. This means you read in 4 bytes from the SD card, concatenate them into a 4-byte word, and proceed to store the word at address m (see diagram) of BRAM. You do this until the entire frame is loaded, that is, until you store the last word at the 76,800th row. Before each word can be

written, the BRAM write-enable pin must asserted, it is then brought low after the word has been read, this is repeated for each word that is being written.

Addressing between BRAM and SD can be made trivial by incrementally keeping track of the current row in BRAM, and the current pixel in that row. The applying this simple transformation, will yield the corresponding address for the SD.

$$address \leq loaded_rows \ll 2 + num_pix_loaded;$$

or more understandably,

$$sd_address \leq (current_BRAM_row \ll 2) + num_pixels_in_BRAM_row;$$

One limitation is that you cannot read out individual bytes from SD, instead you have to read an entire sector (512 bytes) at a time. However, because data transfer from SD is sequential, you can perform a write to BRAM for every 4 bytes read from the 512-byte stream. You continue in this manner for 600 sectors, which coincides with the 76,800th row of BRAM and the end of the frame.

V.) Reading from BRAM to SD Module

Assuming a verified implementation of the *SD Controller*, the *BRAM_to_SD* module may be implemented as follows. Keep in mind, that the process of writing to the SD, although similar to reading from it, has a few small important considerations that should not be overlooked.

Once again, our BRAM is defined by 76,800 4-byte rows, one 32-bit word per row accordingly. The goal now is to take each of those bytes sequentially and write them to the SD one sector at a time, until we complete 600 sectors. Essentially, we now need a byte from BRAM and the SD address to where it will be written.

Our first challenge deals with how we store each byte in BRAM. Our definition calls for four bytes to be stored per BRAM address. This problem can be solved by visiting the same address four times while using the two least significant bits of the SD address to decide which of the four bytes we want on that iteration. For this particular case we have the following simple linear transformation to obtain the BRAM address from which to read to SD. The SD address where the data is stored, itself is only incremented by one sector every 512 bytes.

```
assign read_address = y*BRAM_frame + (x >> 2);
```

or more understandably,

```
read_address <= num_BRAM_blocks*BRAM_block_size + (BRAM_rows >> 2)
```

Where BRAM_block size is equal to 160 rows or equivalently 640 bytes. BRAM_rows is unique to each word, or the same per 4 bytes.

The protocol for writing BRAM to SD is a bit more intricate than its inverse, however it is easily explained. First, in order to write, you need to assert the *wr* signal of the *SD Controller* and send the first byte to its *din* data input port at the address. Following that you set *wr* low and pass in 512 bytes to SD. Then update address and repeat process until all 600 sectors are written to.

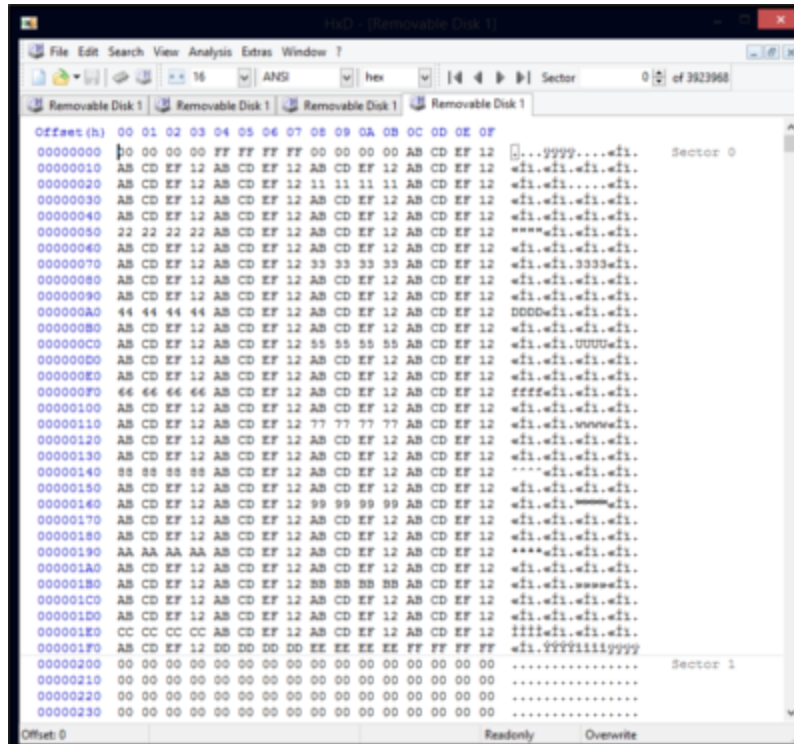
On the *SD Controller* module, one of the easiest things to mess up is the sequence in which *ready_for_next_byte* is asserted. That is, only assert that specific signal when a new and valid data byte is being written to SD.

VI.) Debugging the Modules

Initially, in order to debug the module that read data from SD to BRAM, we sent the data via VGA to a monitor. This allowed us to visually confirm that what we were reading in, was for the most part accurate. This BRAM to VGA module was the initial structure used to implement the BRAM to SD module, as extracting each byte from BRAM was the same in each case.

This visual test on the SD to BRAM module were good enough for a start, but not foolproof. A robust way to truly verify the correctness of a read into BRAM, is to write the data back to SD and manually check its hex values. However, the write module had yet to be implemented at this stage.

Debugging the BRAM to SD module was a design challenge all on its own and consisted of two main design choices. The first, was the use of the hex editor application HxD in order to visually read the raw contents stored in the SD card. This is a Windows specific application, and must be run with administrator privileges. An screenshot is shown below.



Initially, we hacked the SD to BRAM module to write in a single color at each pixel location, by simply ignoring the incoming data and replacing each byte with a fixed hex. The idea behind this was that if we could see 600 sectors with the same hex value at each byte, then our write module was working. This turned out to be a naive approach. Instead, our second important design choice was the creation of a .coe file, specifically for debugging the BRAM to SD module, this file can be added to BRAM as an initiation file during the configuration process. Our .coe file contained distinct hex values every 10 bytes. The file was defined in this way because many of our errors were due to data being skipped when writing or off-by-one. Having distinct hex values at predetermined locations, allowed for easier points of reference when debugging. You can see from the screenshot, the values used in the .coe debugging file. The file is read top to bottom, left to right. This visual representation of data was key to debugging our program and is defined simply as follows. In our case the initialization vector is 512 bytes long, corresponding to one sector of SD.

```
memory_initialization_radix=16;
memory_initialization_vector= 00000000 ffffffff .... eeeeeeee ffffffff;
```

VII.) Final Thoughts

The solution for the implementation of each module described above appears trivial in retrospect. Follow a standardized protocol and ensure proper addressing transformations while meeting correct timing constraints. However, a lot of the initial work went into understanding the architecture of each memory, as well as the protocols that allow communication between them. The diagrams above are the latest version of what turned out to be a great deal of research and empirical testing. We hope that these diagrams reduce this initial learning curve for these architectures. The SPI protocol is static, and we hope as well to have provided a good general overview of what it is, enough for a successful implementation or at least a decent guide to understanding the *SD Controller* module.

Finally, we try to make it clear that while our design choices shown here work, they are not the only ones. That being said, we believe that this guide lends itself to future modifications and alternate definitions of these memory architectures. So long as the one-to-one address mapping is conserved and protocols followed, other implementations should work just as well.

B. Further Expansion

I.) Multiplayer Drawing

Implementing this would not be difficult, it would require copying a section of our code and finding new thresholds for the new LED colors we decide to choose for the second user. However, we did not attempt to implement this in the very narrow timeframe between our working prototype and our final check-off.

II.) Non-Light based Hand Gesture Recognition

As discussed in this paper, one of the of the original ideas which had to be discarded due to timeline constraints was hand gesture detection. Given more time, we might be able to discard the glove completely, and use the bare hand to signal the application. This is a very reasonable goal, though we would expect a slight decrease in fidelity.

III.) Sprite Strokes

A feature which we were unable to implement was the ‘undo’ function. This requires that you store in memory the values of each continuous line you make sequentially. In order to achieve this, you must take into consideration the available memory as well as an efficient way to store the data. Not an easy process, but one that might be doable given enough time.

IV.) Direct Printing

As a future stretch goal, it would be interesting to be able to connect to a printer using the USB standard. This might be a difficult task, especially the initial research. However, after finding a compatible printer it should definitely be possible to add to our implementation.

C. Timeline

Task	Oct 31	Nov 7	Nov 14	Nov 21	Nov 28	Dec 5	Dec 12
Final Proposal Draft							
Block Diagram Meeting							
Project Design Presentation							
Figure out the solid idea/plan on how to import data from camera, image processing technique to detect colors, how to distinguish different colors for control, how to read/write memory, how to save/load file, and drawing data structure							
Checklist Meeting							
Finish camera connection							
Finish memory (BRAM/SD) connection							
Finish gesture control from color detection							
Finish color detection							
Finish GUI							
Finish reading/writing data to memory							
Optimization/Testing							
Polishing/Wiring Glove							
Checkoff/VDO Recording							

D. Conclusion

Our team set out to create an application which would allow the user to draw out their ideas in thin air and be able to visualize them on screen. We accomplished this. Our goals were clearly stated, and we were able to tackle them sequentially.

From a non-technical point of view, our application is not complex. It is not meant to be. Our biggest concern in this area was ease of use. We gave the user two direct inputs, a cursor and a ‘click button’. This was design choice turned out to be the right one, as it allows the developers full access to modify the on-screen features or even change the application itself.

At a technical level, synchronizing the various modules was one of the biggest challenges. Understanding how the various constraints of our hardware and the different modules we created could work together was quite the learning experience. However, we found it rewarding to have understood our project requirements well enough to achieve our goals.

Perhaps the biggest lesson of this project was the idea of design tradeoffs. We made starting from the choice of hardware to deciding on how the graphic user interface would look like. In order to do this, we had to consider the entire project before even writing the first module. Having clear goals really went a long way to helping us make the ‘right’ choices for our particular application.

A second lesson might speak to starting a project using a previous project’s code. It is a good start to get into the FPGA platform, but you cannot make the assumption that it will be correct one-hundred percent of the time. Specifically, this delayed our progress of writing to the microSD card. Since the read protocols from our borrowed SD Controller module ‘worked’ we acted under the impression that the write protocols were correct as well. This was not the case, the previous project in question had not needed the write protocols and did not bother to check its functionality. Later we also realized that the read functionality had glitches. So again, test thoroughly all the code you want to use.

Air Brush was an exciting learning experience for us on how to take a project from the initial conception of an idea to a full implementation. We are thrilled to have successfully worked on this project together, and look forward to using our newly acquired knowledge in future project. More importantly as engineers, we do not simply care about our own success, we want to see our work have an impact on the world. We hope that our project will inspire other students to pursue their own unique projects

whether these be in digital system development, the arts, or any other topic of their choosing. That they might go out there and build, create, and learn.

E. Resources & Datasheets

[Nexys 4 DDR Artix-7 FPGA Site](#)

[VGA Video Output Tutorial](#)

[OV7670 CameraChip Implementation Guide](#)

[OV7670 CameraChip Datasheet](#)

[SD Card Standards](#)

[SPI Introduction](#)

[Gemma LED Starting Pack](#)

Photo Gallery

