Encryption with Chaotic Maps Using a DE2-115 Development Board

Jihoon Lee

Zachary Splingaire

May 8, 2016

ECE 395: Advanced Digital Projects Lab

University of Illinois at Urbana-Champaign

Abstract

The goal of our project was to implement an image encryption algorithm which uses chaotic mapping functions on the DE2-115 development board. There were many algorithms to choose from, but we decided on the algorithm defined in Image Encryption Based on Diffusion and Multiple Chaotic Maps by G. A. Sathishkumar, Dr. K. Bhoopathy Bagan, and Dr. N. Sriraam. [1] The encryption algorithm takes an initial key and uses a combination of chaotic maps to generate subkeys. These subkeys control the encryption process, where transformations are applied to the image. The chaotic maps are an important part of this algorithm due to their sensitivity to initial conditions. We also interface our DE2-115 board with an OV7670 camera so we can capture our own images.

Setup

The main component of our project is the Altera DE2-115 development board, on which we programmed the entire encryption algorithm. Our design includes an OV7670 camera, which is can capture images and output them in RGB565 format. Our design requires a VGA compatible monitor to display encrypted and unencrypted images. We also ended up using a breadboard to more easily make connections to the camera and the DE2-115 board. We supplied the camera's voltage input using a voltage generator and its system clock using a waveform generator. These generators allowed us to better control the inputs to the camera since we've experienced multiple fried cameras. Figure 1 displays the pins of the OV7670 camera.

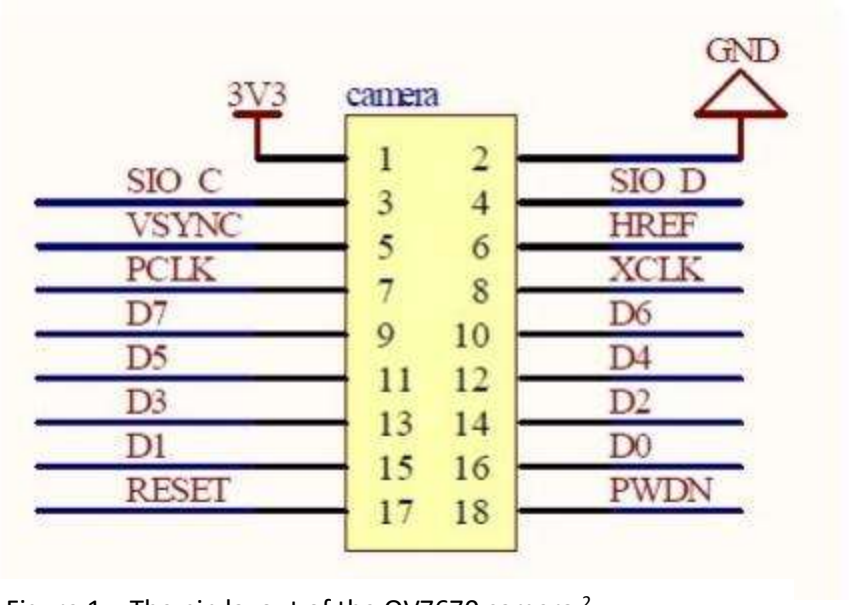

Figure 1 – The pin layout of the OV7670 camera.[2]

SIOC is the clock input to the Serial Camera Control Bus (SCCB) interface. SIOD is the input/output data line of the SCCB interface. VSYNC is the vertical sync output, which denotes the start and end of a frame. HREF is the horizontal reference output; it is at a logical 1 when valid row data is being output. PCLK is the output of the camera's pixel clock. XCLK is the system clock input. D7 through D0 are the output data pins. RESET clears the camera's internal registers and restores their default values. PWDWN determines whether the camera operates in normal mode or power down mode. RESET and PWDWN have internal pull-down resistors.

The 3V3 and XCLK pins were connected to the two generators. We supplied the 3V3 pin with 2.8 volts. The XCLK pin was supplied with a square wave with a frequency of 25 MHz, an amplitude of 1.6 volts peak to peak, and an offset of .8 volts. The GND pin was connected to a common ground shared between the DE2-115 board, camera, voltage generator, and waveform generator. The RESET and PWDN pins were left disconnected and disabled since they

were not utilized in our design. All other pins were connected to GPIO pins on the DE2-115 board.

Interfacing with the OV7670 Camera

Our initial stage of our project involved properly capturing and transferring an image from the camera to the DE2-115 board. By default, the OV7670 camera outputs in YCbCr format. Using the SIOC and SIOD pins, we can write values into the camera's internal registers so that the output data of the camera is in RGB565 format instead. We chose the RGB565 format over the YCbCr format for two reasons: RGB565 can be easily converted to RGB888, which is the format our VGA output requires; and the data for a single pixel spans only two bytes in RGB565, unlike YCbCr where the data for a single pixel spans three bytes.

In order for the camera to output in RGB565 format, we needed to write specific values to its internal registers. The OV7670 datasheet specifies which registers needed to be changed and what values they needed to be changed to. The list of all internal registers and their functions can be found in the OV7670 datasheet. Bits 2 and 0 of the COM7 register needed to be changed to 1 and 0, respectively. Likewise, bits 5 and 4 of the COM15 register needed to be 1 and 1. The SIOC and SIOD pins form the 2-wire Serial Camera Control Bus (SCCB) interface of the OV7670. In order to write to the camera's internal registers, we needed to design a module on the DE2-115 board to communicate through the camera's SCCB interface according to the specifications of the 2-wire SCCB interface.

Besides output format, registers govern input gamma ray reception curve, color conversion, and DSP. We were only able to figure out the most crucial registers and some other color conversion values and gamma curve values. However, the latter (which are less important) values are still experimental and needs improvement.

Some crucial settings were concealed by Omnivision as corporate secret, and they are only released upon signing agreement. Yet those are still not available to individuals. We had to experiment on our own and refer to other open source projects to figure out.

Register (Address)	Data	Content
COM7 (0X12)	0x04	Use RGB output (have options of other formats like YUV)
COM15 (0X40)	0xd0	Select RGB 565 version
TSLB(0X3A)	0X04	Secret value not revealed by Omnivision. Data sheet does not reveal the function of the register, but camera simply does not work right without this setting.
CLKRC (0X11)	0X80	No Pixel Clock Scaling
COM13 (0x3D)	0XC0	Maximum UV level adjustment

Reference Documents:

The OV7670 datasheet [3]

The OmniVision SCCB Functional Specification [4]

Once we were able to change the output format of the OV760, the next step was to design the interface to write the data from the OV7670 into memory and display that image to the monitor. We decided to use the DE2-115 SRAM memory due to its capacity and access time. Figure 2 shows how the data for RGB565 format will be output. For one pixel, the red component and two bits of the green component will be output on the first cycle of PCLK. The remaining bits of the green component and the blue component will be output on the next cycle of PCLK. The size of a single pixel in RGB565 format is two bytes.

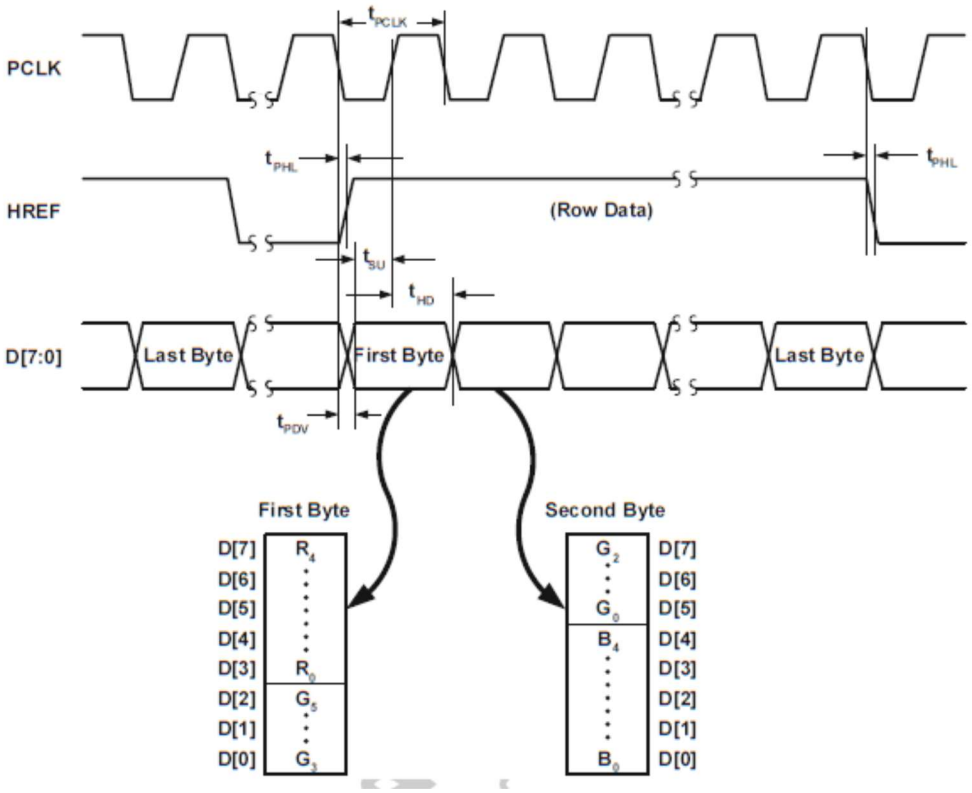

Figure 2 – The RGB565 Timing diagram. [3]

The word size of SRAM is two bytes, so we were able to store one pixel per memory address. In total, a 640 by 480 pixel image takes up 307,200 addresses. Since the camera outputs image data row by row starting from the top left of the image, each row is written into

memory from left to right, and the while images is written from the top row to the bottom row. By following the specified timing of the camera's output, our design writes a pixel to memory at every other byte sent from the camera. Displaying the image onto the monitor is as simple as reading each pixel from memory consecutively because of the layout of the image in memory.

We used code to generate our VGA output signals from lab8 of ECE 385 written by Christine Chen. The vertical sync and horizontal sync follow the industry standard timing for 640x480 VGA signals. [5]

Subkey Generation

Our encryption algorithm is based strongly on the algorithm defined in the paper by G. A. Sathishkumar, Dr. K. Bhoopathy Bagan, and Dr. N. Sriraam. [1] However, we made some modifications so that the algorithm would be compatible with our image format. The algorithm involves using chaotic maps to generate subkeys, which would then be used to encrypt the image. The following are the chaotic maps we used:

Logistic Map:

$$x[n+1] = \mu * x[n] * (1 - x[n])$$

The logistic map exhibits chaotic behavior for certain values of μ where $0 \leq \mu \leq 4$.

We chose μ = 3.75.

Tent Map:

$$x[n+1] = \begin{cases} \mu * x[n] \ \ for \ \ x[n] < \frac{1}{2} \\ \mu * (1 - x[n]) \ \ for \ \frac{1}{2} < x[n] \end{cases}$$

For the tent map to show chaotic behavior, we chose μ to be equal to 1.5.

Quadratic Map:

$$x[n+1] = x[n]^2 + c$$

With c equal to .375, we were satisfied with the behavior of the quadratic map.

Bernoulli Map:

$$x[n+1] = \begin{cases} 2 * x[n], & 0 \leq x[n] < \frac{1}{2} \\ 2 * x[n] - 1, & \frac{1}{2} \leq x[n] < 1 \end{cases}$$

Since all four of the chaotic maps are only defined for x[n] between 0 and 1, we came up with a representation of this range of numbers. This representation is essentially a reverse form of the standard unsigned representation of integers. In the unsigned representation, the least significant bit represents 2^0, the next bit to the left represents 2^1, etc. For our representation, the most significant bit represents 2^0, the next bit to the right represents 2^{-1}, etc. Fortunately, addition and subtraction between numbers in our representation use the same process as addition and subtraction between unsigned numbers, so there was no need for us to design adders or subtractors for our representation. We did need to design a module to multiply two numbers in our representation.

We decided on a subkey length of 80 bits. Our subkey generator takes an initial subkey value and a seed as inputs, which are both taken from the key that the user inputs. The initial subkey is the initial value that will be sent through a series of chaotic maps. The seed determines how long a single map is repeated and the order of mapping functions. Since the range of each of the chaotic maps is the same, the output of one chaotic map can be directly used as the input of another. After this sequence of chaotic maps, a subkey will be generated with a seemingly random value. Its value is not truly random since the subkey generator is deterministic.

Row Shifting

The first stage of our encryption algorithm is to perform row shifting along all rows of the image. The image is treated as a matrix of 8x8 subblocks of pixels. The entire image is composed of 80x60 of these subblocks. First, we generate one 80-bit subkey for this process. For each row of subblocks (60 in total), if the most significant bit (MSB) of the subkey is 1, the current row of subblocks is circular shifted right by one block. Otherwise the first column of subblocks is circular shifted down by one block. Then the subkey is shifted left by one bit. It is important to note that only the very first column is shifted when the MSB of the subkey is 0, regardless of what the current row is. The next stage in the process is to perform a similar process with the columns.

Column Shifting

The column shifting stage is similar to the row shifting stage. The image is still treated as a matrix of 8x8 subblocks of pixels. A new 80-bit subkey is generated for this stage. For each column of subblocks (80 in total), if the MSB of the subkey is 0, the current column of subblocks

is circular shifted down by one block. Otherwise the first row of subblocks is circular shifted right by one block. Then the subkey is shifted left by one bit.

Quadrant Shifting

The original intent was to make diagonal shift for 16 blocks of image chunk, it was very difficult to make HDL code for the routine. We could not make a big enough array for easy code because synthesis would have taken hours if we have done so (It would've been very easy for software languages like C that has array function). So we only divided image into 4 quadrants only, and depending on the specific bits of the key, the direction of the shift is decided.

XOR

For every 5 pixels, we could make XOR mask with a single key. (80 bits Key, 16bits each for a single pixel) By changing keys and pixel orientation constantly, we could make effective pixel value mask.

Encryption Cycles

One encryption cycle consisted of row shifting, column shifting, quadrant shifting, and then the XOR stage. We initially had our design perform four cycles automatically. We ran into problems with this design, so we modified it so that the cycles would be manually controlled by the switches on the DE2-115 board.

Decryption

Our decryption process involves simply reversing the operations done in the encryption process. The decryption process uses the key from the user to generate the same subkey values used in the encryption process. The first cycle of decryption uses the same subkeys from the last cycle of encryption, the second cycle of decryption uses the same subkeys from the second to last cycle of encryption, etc. For one decryption cycle, XOR is applied first. The XOR stage is the same as the XOR stage in encryption since the XOR operation undoes itself when using the same subkey. Then the quadrants of the image are shifted in the opposite direction that they would be shifted during encryption. Column shifting starts from the last column of 8x8 subblocks of pixels. If the least significant bit (LSB) of the subkey is 0, the current column of subblocks is circular shifted up. Otherwise, the first row of subblocks is circular shifted left. Then the subkey is shifted right. This process is repeated for all 80 columns of subblocks. Row shifting starts from the last row of 8x8 subblocks of pixels. If bit number 59 (since row shifting only uses the first 60 bits of the subkey) of the subkey is 1, the current row of subblocks is circular shifted left. Otherwise, the first column of subblocks is circular shifted up. The subkey is shifted right. This process is repeated for each row of subblocks

Design Overview

An overview of our design is shown in figure 3.

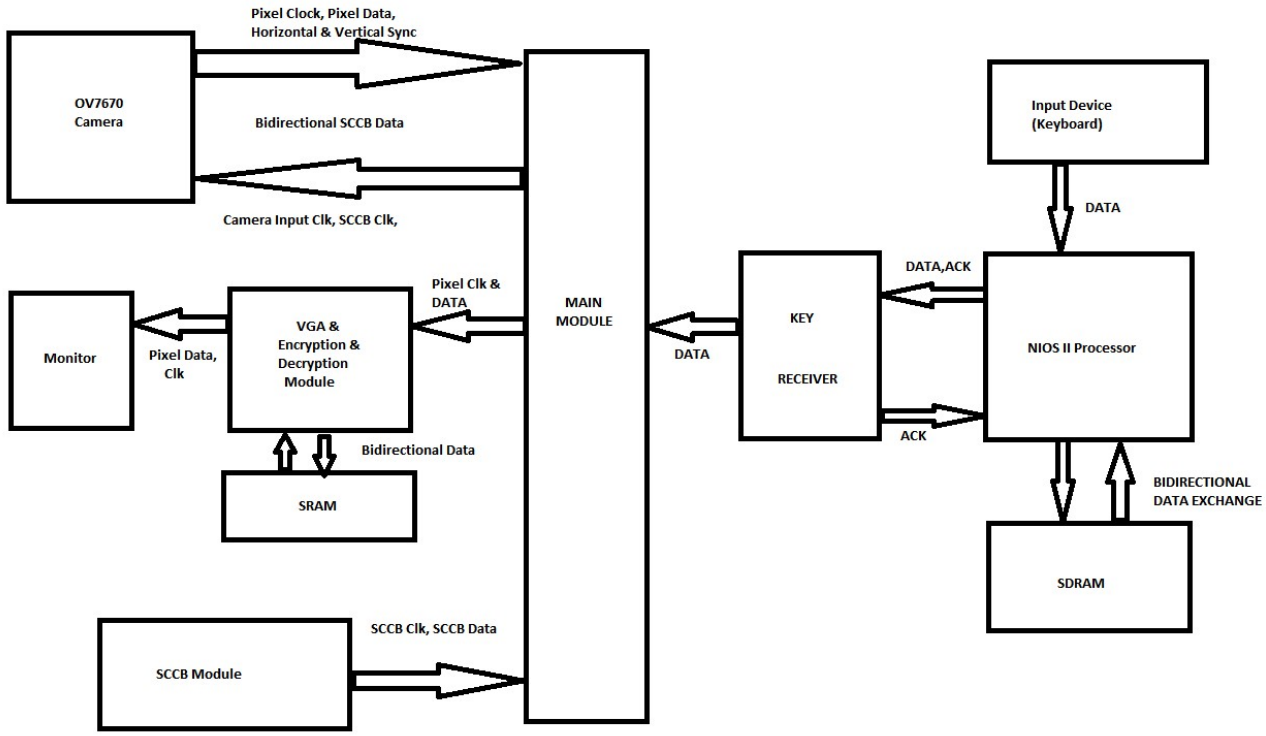

Figure 3 – The general design of our project.

One-Cycle Encryption and Decryption

The following figures show the encryption and decryption process for a single cycle.

Figure 4 – An unencrypted image taken by our camera.

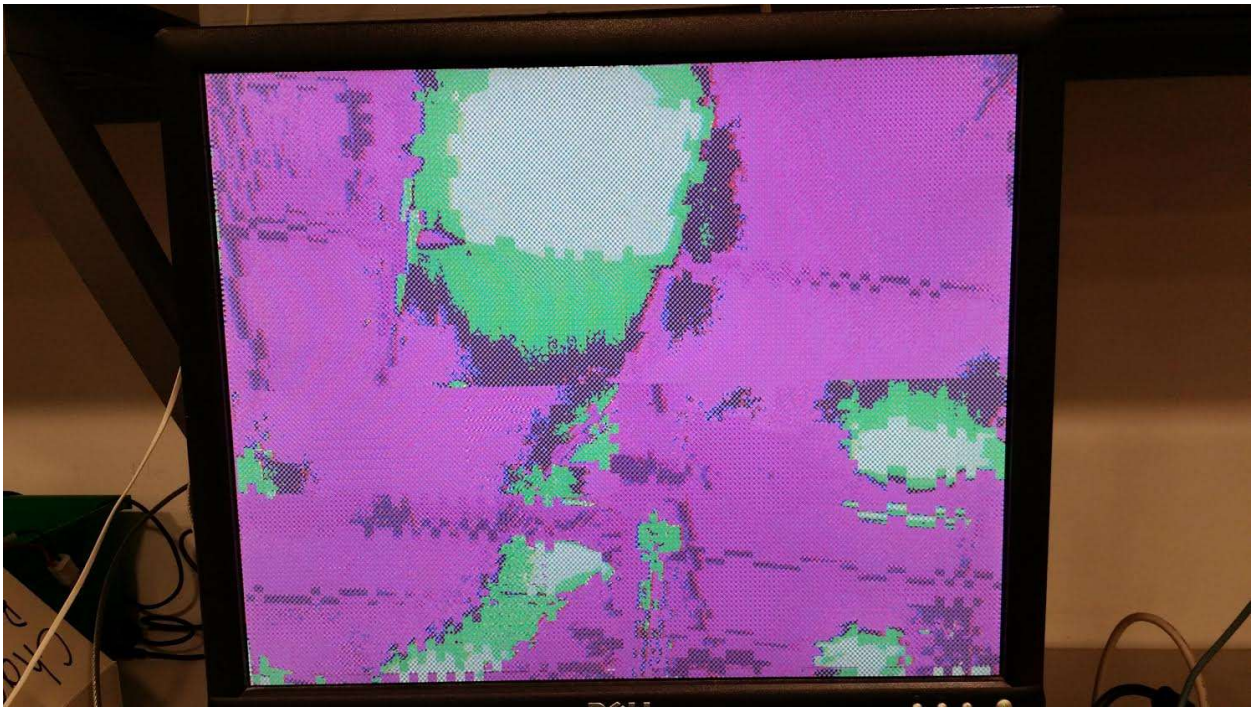

Figure 5 – The image after one cycle of encryption.

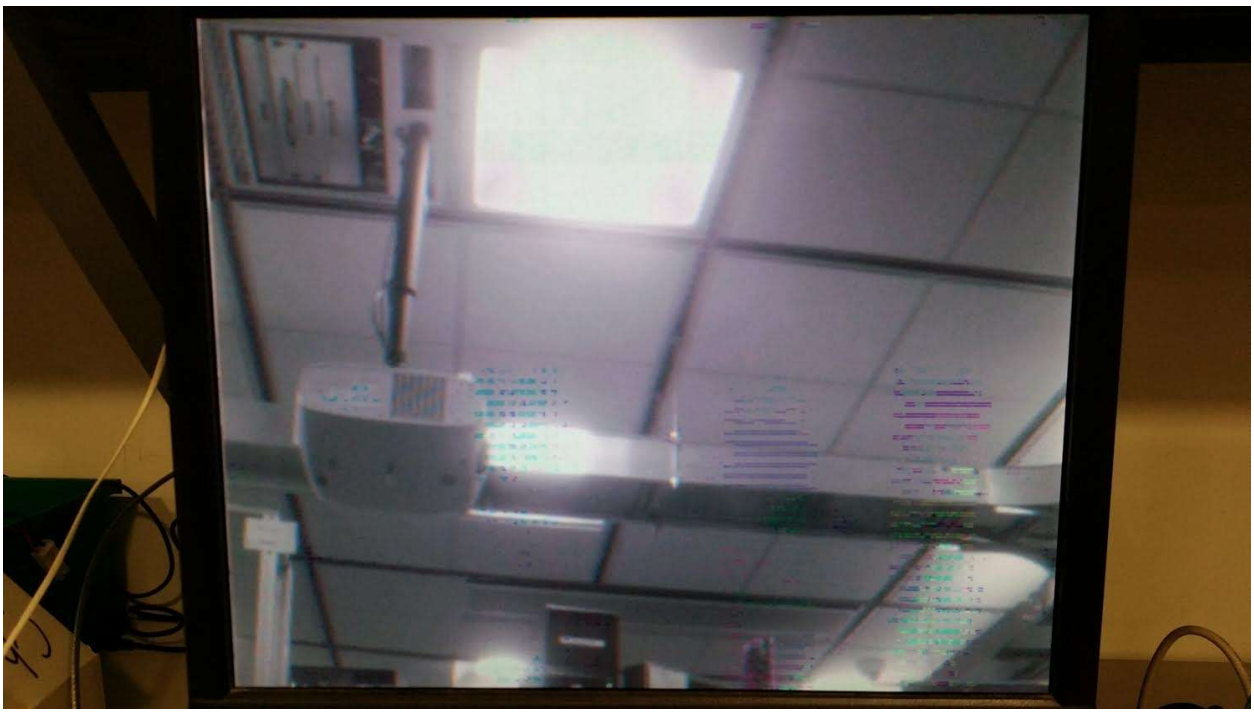

Figure 6 – The encrypted image after one cycle of decryption.

Multi-Cycle Encryption and Decryption

The following figures show the encryption and decryption process for repeated cycles.

Figure 7 – An unencrypted image of us taken by our camera.

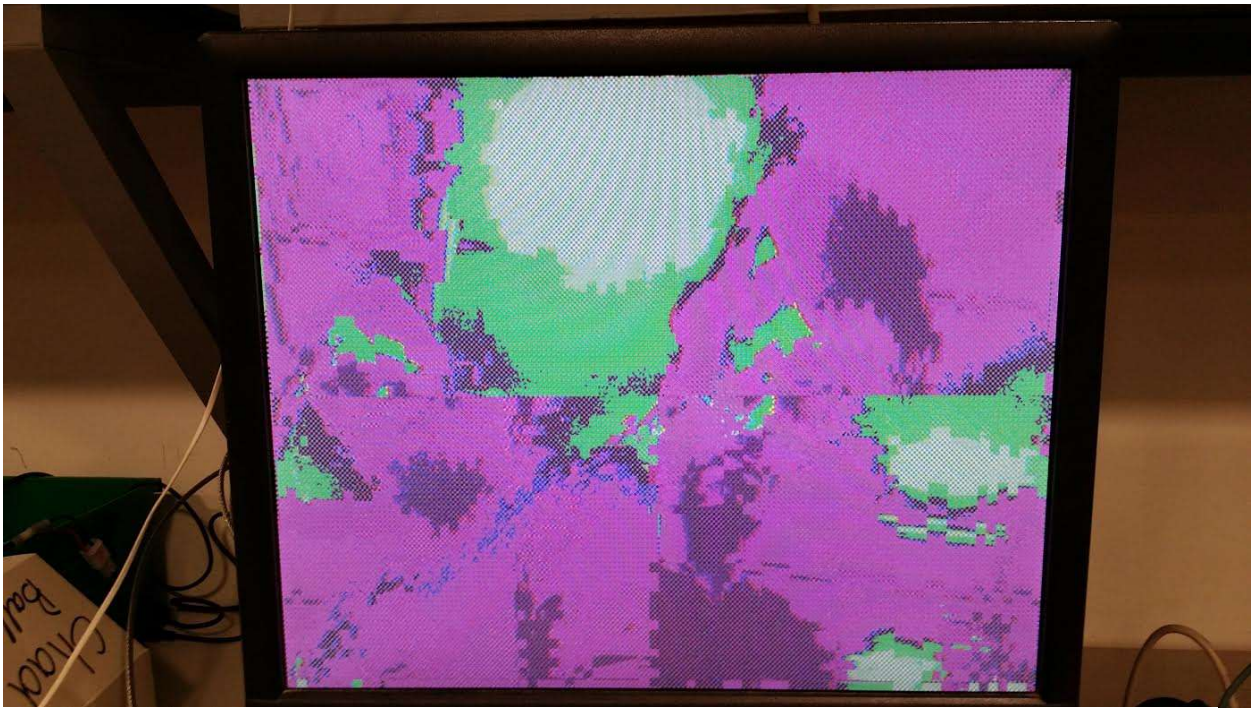

Figure 8 – The image of us after one cycle of encryption.

Figure 9 – The image of us after two cycles of encryption.

Figure 10 – The image of us after three cycles of encryption.

Figure 11 – The encrypted image of us after one cycle of decryption.

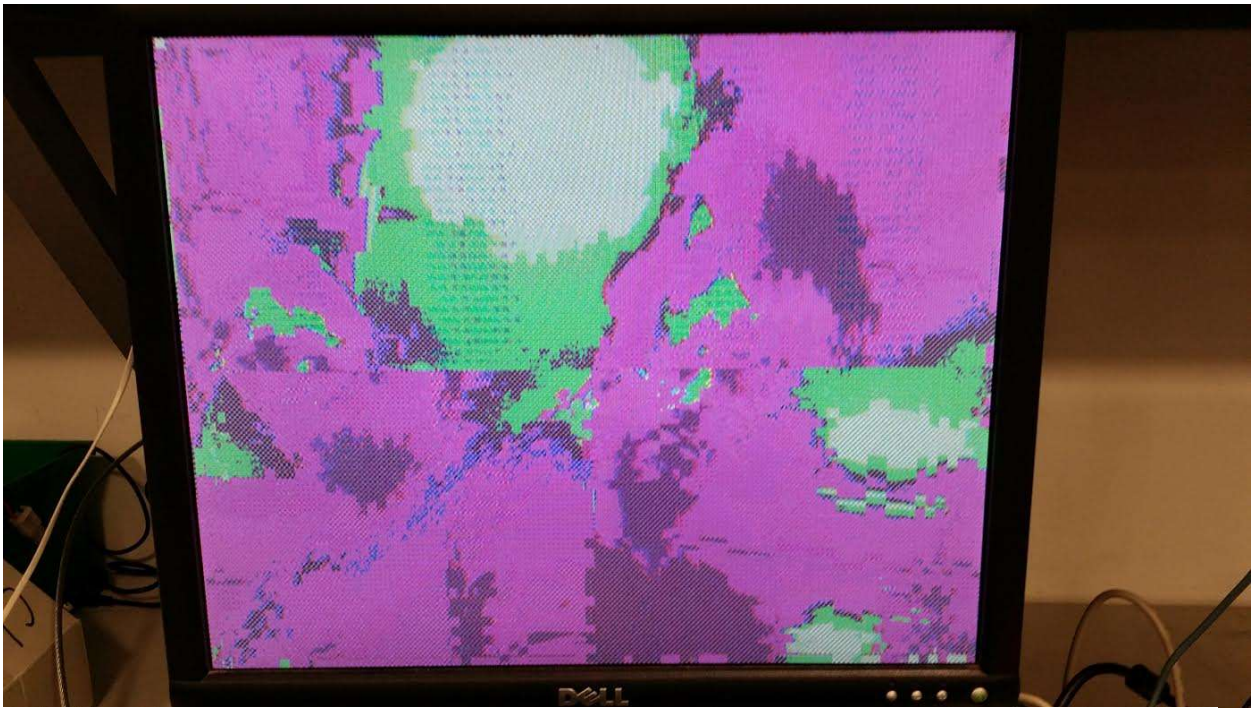

Figure 12 – The encrypted image of us after two cycles of decryption.

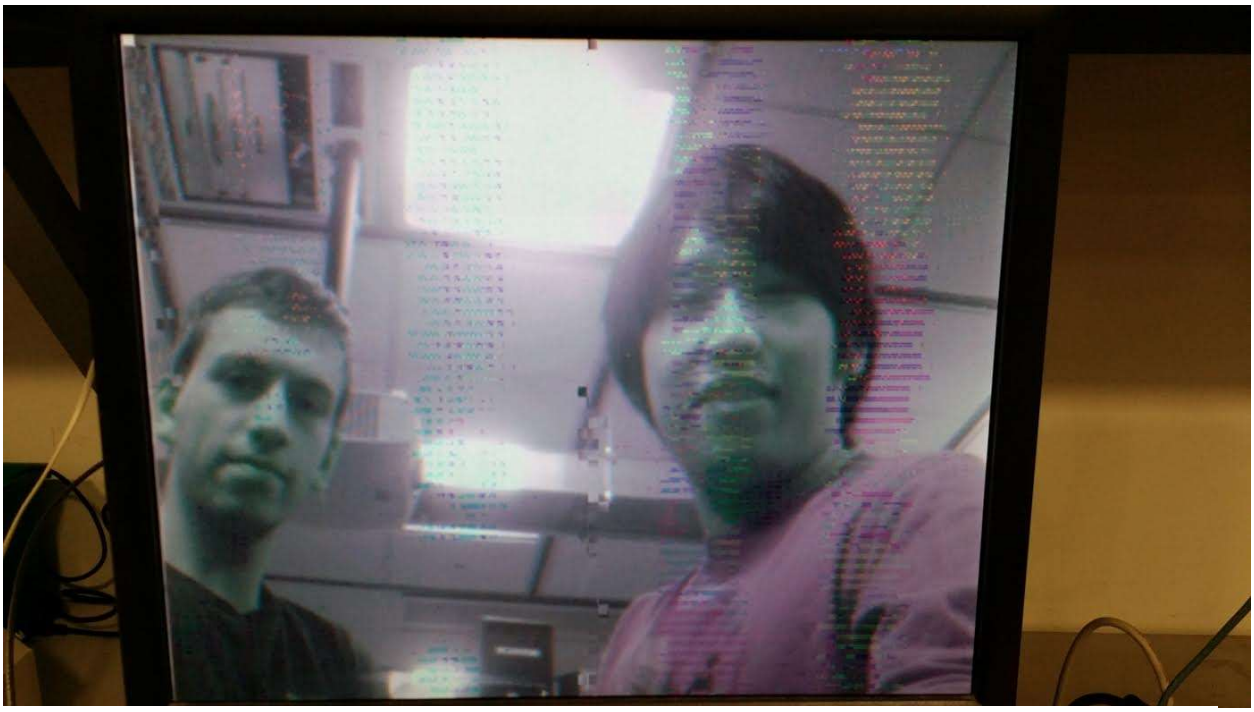

Figure 13 – The encrypted image of us after three cycles of decryption.

Areas of Improvement

One major area of improvement is the loss of data by our design. It can be seen that there are locations in the decrypted image where the data is not the same as the original image. (Artifacts). This was definitely not the error in algorithm or systematical aspects, since it was not present in some occasions, and it would suddenly come up in random moments. Unfortunately, we were not able to figure out where this error was coming from by the end of the semester. Additionally, the design could use more improvement regarding the actual encryption process. Currently, there is still noticeable correlation between neighboring pixels in the encrypted image. Increasing the amount of encryption cycles could remedy this occurrence, or the general algorithm could be modified.

Possible fix for the artifacts would be enhancing timing constraint file by manual inspection. Our timing analyzer gave some warnings but we spent most of time getting camera to work so we were not able to figure it out in time. Also, since the image encryption is done purely through SRAM, we can try to get more special operation range (or wiggle room for operation) for SRAM writing/reading, since timing is tightly implemented in our algorithm.

Also, we replaced diagonal shift with quadrant shift because of time management for deadline. Wring the routine was highly meticulous for hardware description language. It would have been a lot more (still understatement) easier in software language. But NIOS II was very slow with 50 Mhz system clock so we did not want to risk the efficiency for final demo. Taking more time into it, we would be able to finish the routine.

References

[1] https://arxiv.org/ftp/arxiv/papers/1103/1103.3792.pdf

[2] http://www.electrodragon.com/w/OV7670_Module

[3] http://www.voti.nl/docs/OV7670.pdf

[4] http://www.ovt.com/download_document.php?type=document&DID=63

[5] http://tinyvga.com/vga-timing/640x480@60Hz