Reconfigurable Hardware Color Depth Changer

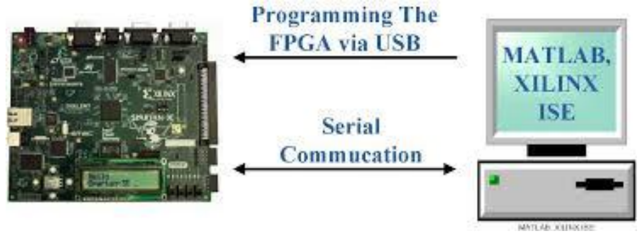
Timothy Kidd[1], Daniel Heer[1], David Bui[1], Aaaron Nguyen[1], Reinhart Paghunie[1], Dr. Mohamed El-Hadedy[1]
[1]Department of Electrical & Computer Engineering
California State Polytechnic University, Pomona, CA USA
{twkidd, dmheer, davidb, aaronknguyen, rjpaghunie, mealy}@cpp.edu

Abstract—**This paper discusses reconfigurable hardware in terms of changing an images color depth. Reconfigurable hardware is a term used for merging the two different mediums of hardware and software and how they interact with one another. This computing technique performs calculations in a hardware platform while presenting the solution in the format of software in order to boost performance. This allows developers to have a fast, reliable and accurate solution to solve many system problems. Our reconfigurable hardware in this context is an FPGA (field programmable gate array) board. An FPGA board acts as a powerful computing solution in order to load, calculate, and process information. The FPGA was specifically chosen since it is completely reconfigurable and allows for different kinds of logic depending on the solution needed. The project aims to use this reconfigurable hardware solution in the realm of computer graphics for color quantization in which an image's original RGB colors are reduced and the image is then reproduced to a similar looking image but at a smaller size. Our project is a 4-stage process in which our hardware (FPGA) stores an image of any color depth and size, reduces this color depth and outputs it to MATLAB in order to be displayed. This project in turn, aims to solve problems in system incompatibility in terms of an image being unable to be displayed on a screen.**

Keywords—FPGA; VHDL; color quantization ; RGB; color depth; reconfigurable hardware

I. INTRODUCTION

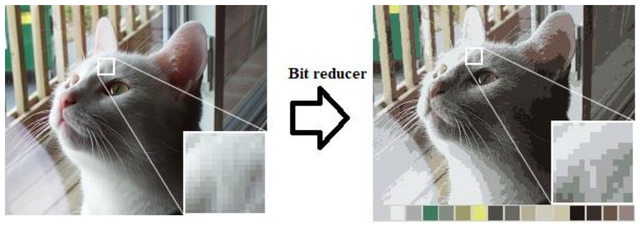
This paper will describe the design and implementation of a reconfigurable color depth changer. The design of our project is inspired by the concepts of reconfigurable hardware and color quantization. To solve the problem of display incompatibility on devices of different memory sizes, a host image's color depth and data size would need to be changed accordingly to the device that will display the image with the use of hardware which will act as a reconfigurable computing device. In [1], Reconfigurable computing allows different logic gates on the board to be changed according to the work that needs to be done during run-time. Thus, by allowing different circuits on the board to be adjusted accordingly at run time, a designer has a powerful device to custom make solutions. The hardware specifically chosen, was an FPGA board since this platform offers this power functionality that can give tailored solutions by adjusting its datapath. In [2], an FPGA does not offer a single static solution like a microprocessor. The FPGA offers a distinct advantage of updating the chip on the go with software updates. The FPGA offers powerful solutions in the form of look up table (LUTs) and the configured logic gates on the board. These are different tables that are configured solely by the designer in order to determine how the logic gates will be configured for a specific solution to a problem. Work can thus be done in parallel through pipelining, allowing for multiple tasks to be done at once; such as in the case of compressing an image. With our project, the FPGA board is mainly used with pseudo image compression where the image rendered into a text file will be compressed to a smaller size as shown in (a).



(a)

Figure 1. (a) Reconfigurable computing via FPGA.

Our original image from MATLAB will be compressed into our FPGA will be assessed using its color depth. In [3], color depth is the bits available that is used to represent that image. This amount of data defines how clear or unclear an image will become. In [4], this color depth or bit depth will have a higher quality image if the number of bits for the image is higher since it can have more colors. Our simplest image contains a range of bits between zero and one which represents black or white. If you want a higher amount of quality, you will need to increase your bit depth to 8 or even 32 bits. With 8 bits, you will have 2^8 or 256 colors available to define an image. With our project, this bit depth will be reduced form a very high number of bits, say 24 bits and reduced to say 8 bits of data from the red, green, and blue values of an image. We are only concerned with these three colors of an image, since these colors can be mixed in different quantities to make up an infinite amount of color ranges from the color map table. This process of reducing an image's bit depth follows the idea of color quantization. With color quantization, the idea is trying to resemble the same image with as little amount of color as possible. In [4], color quantization analyzes these colors as a set of vectors where some vector in a set sample space has a minimum error if that sample of vectors is much less than the dimensional space. In three dimensions, our red, green, blue values represent this vector space. Color quantization follows a set of processes in order to represent an image by a close approximation of a smaller size. First, a sample space must be established for the original image, second, some color map must be established such as the size of its red, green, and blue values. Third, these quantities must be mapped and lastly, the size must be reduced and established into a new image either uniformly or non-uniformly This can be seen in (b).



(b)

Figure 2. (b) Color Quantization.

II. 2-D ARRAY

For our project, an image of any size and color depth will have its individual RGB elements that make up the pixels of the image assessed and converted into binary using MATLAB's built-in function of A=imread(filename) and dec2bi. The input of our system is the RGB values of the original high-quality image in terms of its bit values that is loaded into a text file. In order to perform color quantization of these values, a method of storing these values was needed in VHDL. A two-dimensional array was proposed for this use. In order to create a two-dimensional array of any size, a generic component in the code was needed so our arrays can change in size according to the input image's size. Our ports were dependent on the clock, an enable signal, and a generic output vector. Our two-dimensional array was declared based on the row and column size of our image, where three internal signals would be created. The first signal is an output for our induvial word in the array, next there is two pointer signals created in order check the position of the row and column, and finally two check signals are created for our internal counter process. To iterate through the two- dimensional array, two for loops were originally planned. However, a problem was encountered since the for-loop implementation in VHDL cannot increment. Therefore, on the positive edge of the clock, only one slot value in the array was outputted. In order to solve this problem, the for loop was reconstructed with a series of if-else statements incremented by our internal clock. On the positive edge of the clock, the counter would begin to increment, and if our enable signal was high our output signal would send out the array based on the row and column pointers. These pointers would then be incremented if they were less than their max value and reset back to zero once the max value was reached. The concept of a two-dimensional array can be seen in figure 3.

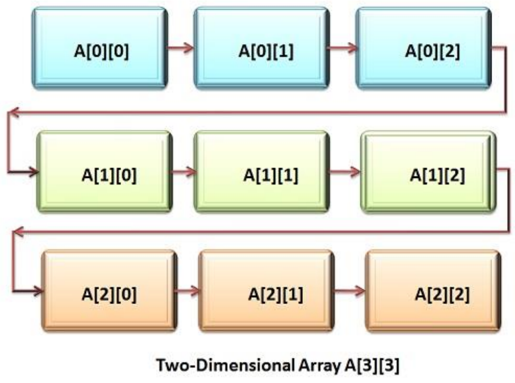


Figure 3. Two-Dimensional Array.

III. BIT REDUCER

After the input of the text file is stored in the two-dimensional arrays, a method of reducing its size was needed. In [6], the concept of color quantization, has two prevalent methods for reducing color depth are either uniform or non-uniform quantization. In uniform quantization, the color sample set is divided into an equal space less than a max value. Our bit reducer is based on this concept. Our input size is divided by three in order to get the size of the red, blue, and green values. Our output size is divided by three in order to get these values at the output. In order to determine how much to divide by, you need to use the formula of $2^{(\text{input size/3-output size/3})}$. For an example, if we have an input size of 24 bits, with an output size of 12 bits, our individual red, green, and blue values will be 24 divided by 3, or 8 bits per color. The output value will be 12 divided by 3, or 4 bits per color. The value we want to divide by will be $2^{(8-4)}$, or 16. We can accomplish this division through shifting (8-4) times and take the least significant (8-4) bits. Our output is the red, green, blue values concatenated together. An image has 3 values or color channels per pixel (R,G,B). Each color channel is represented by a number of bits and this determines how many possible colors there can be. An image with 8 bits per color channel has a total of 24 bits per pixel which means there are a possible 2^{24} colors per pixel
What we're doing is reducing the number of possible colors by reducing the number of bits per color channel. To do this we're dividing by a power of 2 that gives us the number of bits we want. The power of 2 we divide by is the difference between the original color channel bit size and the reduced color channel bit size.
So, if we want to go from 8 bits per color channel to 4 bits per color channel, we divide each color channel value by $2^{(8-4)} = 16$. Each color channel is now represented with a value in the range (0,15) instead of (0,255). This means we now have a total of 12 bits per pixel which means we've reduced the number of possible colors for our image from 2^{24} to 2^{12}.

IV. TEXT FILE SIMULATION

Next, in order to peace together the reduced red, green, blue values in a readable format for MATLAB, simulation of text files was needed. Originally, a testbench was made that used four text files. Input_vectors.txt was used to represent typical inputs to the bit reducer. Output_red.txt, Output_green.txt, Output_blue.txt files were used to hold the resulting output values for red, green and blue. The format for the text files were originally made to have each entry represent a pixel and the entries placement in the 2-dimensional array would be that pixel's location. The testbench was later changed to read separate text files for each color formatted into one column per color. The testbench concatenates the red, green, and blue values together to feed into the bit reducer and then the results are fed to the corresponding output text files: output_red.txt, output_green.txt, and output_blue.txt. The format for the output matches that of the input, but the bit length for the color values are different depending on what was set beforehand.

V. MATLAB RECONSTURCTION

In this part of the project, the desired output, which created from our simulations of our hardware in Vivado, was exported into MATLAB. Once in the MATLAB environment, the image was reconstructed. This was accomplished by making sure the simulations outputted into the format of a binary number for the first part of the new line and next value in the array. The files output red, green, and blue, were all in this format. The code was made generic so that we could easily switch to any resolution of the image along with changing the file name. Next, we stored our string of binary characters into their own matrices representative of the text file. We then had a generic for loop setup that will write a matrix in the style of the image. What is unique about MATLAB is language is in big endian when formatting to binary, so we had to swap ordering since Vivado reads in little endian. On top of this, we had to multiply each value in the matrices by sixteen in order to scale our amplitudes back up to 8 bits of color depth for MATLAB to display the image. The matrices we created then had to be force casted to unsigned 8-bit integer and then stored in a larger three dimensions that allowed for our red, green, and blue matrices to exist in one big matrix. From there, we used MATLAB's imshow function to display the outcome, and for further comparison we made a montage using our original image. The result can be seen in figure 4.

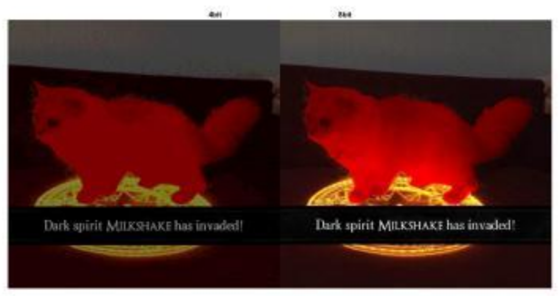

Dark spirit MILKSHAKE has invaded!
Dark spirit MILKSHAKE has invaded!

Figure 4. Reconstructed Image.

VI. CONCLUSION AND FUTURE WORK

The final project was successful. We first took an image of high quality, used MATLAB to translate the red, green, and blue values into a binary array. This was then stored in a two-dimensional array using the Vivado software on the FPGA board. The output of this array was then bit reduced and stored into a text file where MATLAB would then reconstruct the image file at a reduced color depth in order to emulate the quantization of color. Future work has been considered to adapt our output to a VGA display. However, the VGA display outputs images in 640 by 480 resolution and care will have to be given in order to make the code compatible with this system.

REFERENCES

[1] S. Hauck and DeHon André, *Reconfigurable computing the theory and practice of FPGA-based computation*. Boston: Morgan Kaufmann, 2008.

[2] D. J. Barry, "Enabling reconfigurable computing with field-programmable gate arrays," *Network World*, 30-Oct-2017.[Online].Available:https://www.networkworld.com/article/3234796/enabling-reconfigurable-computing-with-fpgas.html. [Accessed: 09-Dec-2019].

[3] "Color Depth," *reaConverter*. [Online]. Available: https://www.reaconverter.com/features/image-editing/color-depth.html. [Accessed: 09-Dec-2019].

[4] S. Segenchuk, "An Overview of Color Quantization Techniques," *An Overview of Color Quantization Techniques*.[Online].Available:https://web.cs.wpi.edu/~matt/courses/cs563/talks/color_quant/CQindex.html. [Accessed: 09-Dec-2019].