Smoothen Image - Hardware Defined Language

Alfonso de la Morena
Ingram School of Engineering
Texas State University
Austin, Texas
a_d426@txstate.edu

***Abstract*—This paper processes an image using an HDL (Verilog) to all the images RGB values and save the corresponding output to a file. The file read will be converted to hexadecimal values using MATLAB. The smoothening and edge detection of the file values will be done in XILINX compiler.**

***Keywords*—bit map image, pixel, hardware descriptive language, module, hexadecimal file.**

I. Introduction

This lab makes use of the MATLAB academic software to model various characteristic of a digital system. MATLAB is a language used for technical computing. It integrates computation, visualization, and programming in an environment where problems and solutions to be represented in mathematical notation.

In addition, this lab also makes use of the XILINX compiler to run Verilog code that will simulate hardware modules that could be built to achieve the simulated results in the real world.

The goal of this lab is to take an image, convert that image to a hexadecimal file that represents all its RGB values using MATLAB and finally pass the hexadecimal file to XILINX it will be processed. One image will be smoothened and the other will pass through an edge detection module.

II. Results

A. *Converting an Image to Hex Values MATLAB*

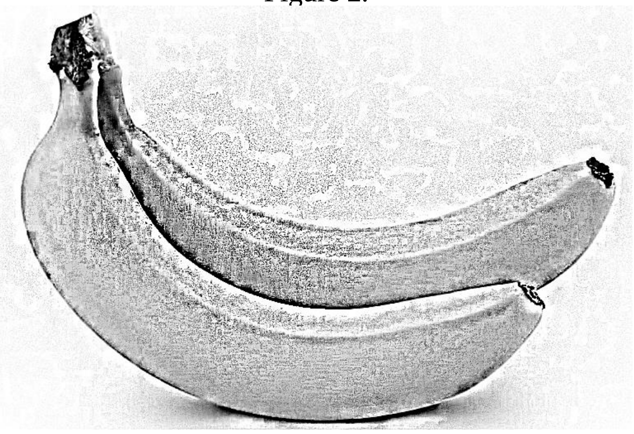
All the images that will be dealt with in this lab will be bit map images which make use of the '.bmp' extension. Figure 1 shows how to convert a bmp file with specified height and width to hexadecimal RGB values.

Figure 1.

```
b=imread('path\myfile.bmp'); % 24-bit BMP image RGB888
k=1;
for i=512:-1:1
for j=1:768
a(k)=b(i,j,1);
a(k+1)=b(i,j,2);
a(k+2)=b(i,j,3);
k=k+3;
end
end
fid = fopen('hedgehog.hex', 'wt');
fprintf(fid, '%x\n', a);
disp('Text file write done');disp(' ');
fclose(fid);
```

Any image with the specified height and width will work with the code provided at the end of the report. However, changing the height or width values require very little modification to the provided code. The image processed for this part of the lab can be seen in Figure 2.

Figure 2.



B. *Smoothening the Image using XILINX*

The next part of the lab was to create a module using the Xilinx compiler that could smoothen all the colors on the image. To do this, a 3x3 matrix around each pixel had to be passed to the module so that it could perform an operation according to surrounding values each pixel had. Extra care had to be taken to avoid the corners of the image since it might only have anywhere from 3 to 7 pixels instead of 9. These operations needed to achieve this are modeled in Figure 3.

Figure 3.

```
top_check = read_pointer < (TOTAL_DATA - WIDTH*3);
bot_check = read_pointer > WIDTH*3;
left_check = read_pointer %(WIDTH*3);
right_check = read_pointer % ((WIDTH*3) -1);

RGB [7:0]     = memory_storage[read_pointer][7:0]; // blue
RGB [15:8]    = memory_storage[read_pointer + 1][7:0]; // green
RGB [23:16]   = memory_storage[read_pointer + 2][7:0]; // red

// Check for edges if we are at the edge then pass RGB instead as the taking the average of itself will cancel out.
if(top_check)                begin top       = memory_storage[read_pointer + (768*3)][7:0]; end else begin top       = RGB [7:0]; end
if(bot_check)                begin bot       = memory_storage[read_pointer - (768*3)][7:0]; end else begin bot       = RGB [7:0]; end
if(~left_check)              begin left      = memory_storage[read_pointer - 3][7:0];       end else begin left      = RGB [7:0]; end
if(~right_check)             begin right     = memory_storage[read_pointer + 3][7:0];       end else begin top       = RGB [7:0]; end
if(bot_check && ~left_check)  begin bot_left  = memory_storage[read_pointer - (767*3)][7:0]; end else begin bot_left  = RGB [7:0]; end
if(top_check && ~left_check)  begin top_left  = memory_storage[read_pointer + (767*3)][7:0]; end else begin top_left  = RGB [7:0]; end
if(bot_check && ~right_check) begin bot_right = memory_storage[read_pointer - (769*3)][7:0]; end else begin bot_right = RGB [7:0]; end
if(top_check && ~right_check) begin top_right = memory_storage[read_pointer + (769*3)][7:0]; end else begin top_right = RGB [7:0]; end
read_pointer  = read_pointer + 3;
write_pointer = read_pointer + 3;
```

Since the hexadecimal values were already calculated using MATLAB in the previous section, once the matrix had been passed to the module the smoothened result could be calculated with the formula seen in Figure 4.

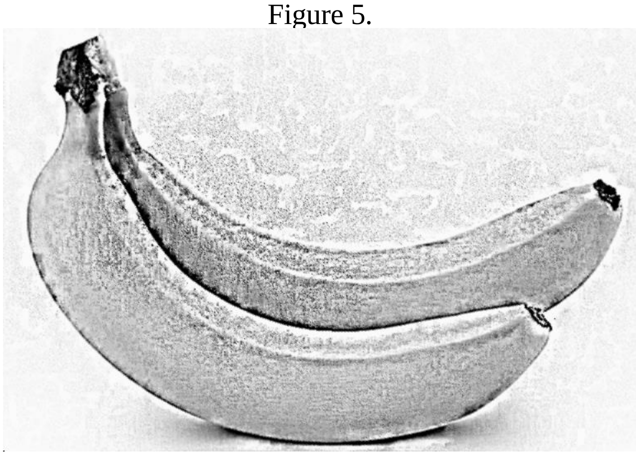
Figure 4.

```
transform =  RGB_Smooth[15:8] +
             top +
             bot +
             left +
             right +
             top_left +
             top_right +
             bot_left +
             bot_right;

// To approximate dividing by 9 we use transform >> 3 - transform >> 6 which
// equals .109375 times its original value
RGB_Smoothened[7:0]=(transform >> 3) - (transform >> 6); // blue
RGB_Smoothened[15:8]=(transform >> 3) - (transform >> 6); // green
RGB_Smoothened[23:16]=(transform >> 3) - (transform >> 6); // red
```

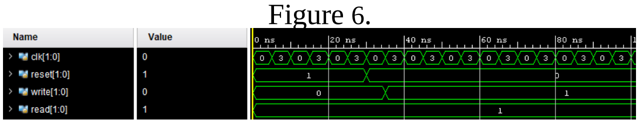
A smoothing operation with right shifts was used because it more closely resembles how the hardware would handle the problem at the machine level. Once the program ran through all the pixels in the hex file it produced the output seen in Figure 5.

Figure 5.



Thus, smoothing was all handled by the *smoothen* module. The rest of the operations required by hardware were simply a pointer to each of the write and read locations. This was done to pipeline the actions so that machine could read and write in one clock cycle. To do so the first cycle had to only perform a read and the last cycle only perform a write. The model for this can be seen in Figure 6.

Figure 6.



As can be seen, read is set to 1 initially so that the first value can be stored and then set to 0. Then once the reset is done and the machine is ready to start writing then both read and write are set to 1 and the process is pipelined to improve the cycles per instruction. Also, note the value of red, green and blue which all have a constant value after passing through the processing element which is the weighted average that was discussed previously.

C. *Edge Detecting the Image using XILINX*

The final part of the lab was to create a module using the Xilinx compiler that could detect the edges of an image. To do this, a 3x3 matrix around each pixel had to be passed to the module so that it could perform an operation according to surrounding values each pixel had. Again, extra care had to be taken to avoid the corners of the image since it might only have anywhere from 3 to 7 pixels instead of 9. These operations needed to achieve this are modeled in Figure 7.

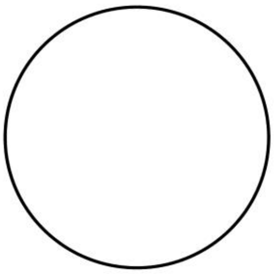
Figure 7.

```
// Operation
transform =  (RGB[15:8] << 3) -
             top -
             bot -
             left -
             right -
             top_left -
             top_right-
             bot_left -
             bot_right;

RGB_Transformed[7:0]=transform; // blue
RGB_Transformed[15:8]=transform; // green
RGB_Transformed[23:16]=transform; // red
end
```

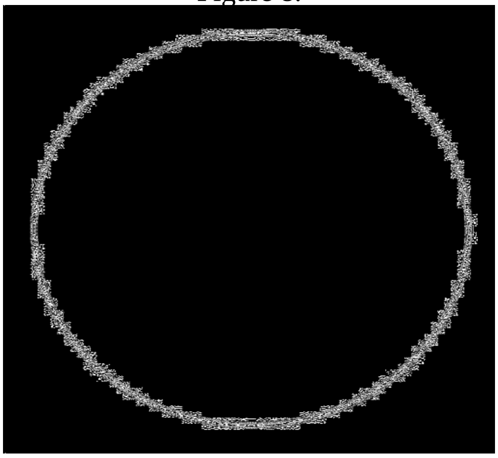
The process was very similar to that of smoothening the image. Once the hexadecimal values had been acquired all that needed to be done was pass them through the module making the changes that were represented in Figure 7. The image processed for this part of the lab can be seen in Figure 8.

Figure 8.



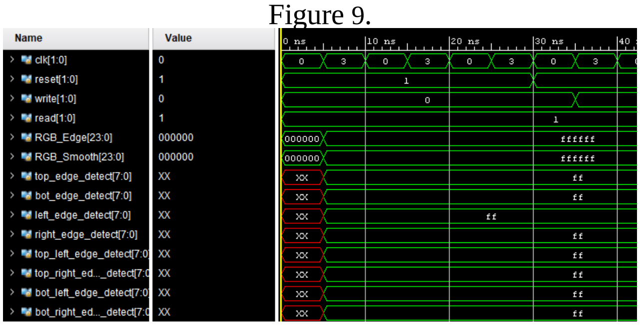
Edge detection worked very similar math to the smoothening module. The operation being to multiply the pixel of interest times 8 and then subtract all the surrounding pixels. When one of those pictures was unavailable because the pixel of interest was at the edge, the module instead passed the value of the pixel of interest so that it would average out. The final image after having passed through the edge detection module can be seen in Figure 9.

Figure 8.



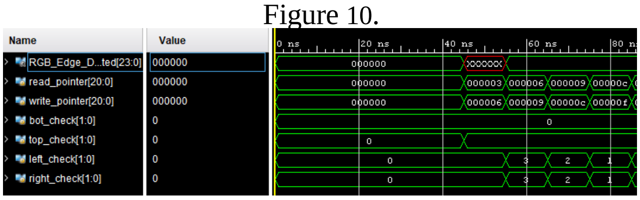
As can be observed, anywhere where there is an edge in original picture the pixels have a value other than 0. This is due to the module performing the operations discussed earlier. The waveform for the initial setup of the edge detect module can be seen if Figure 9.

Figure 9.



Very similar to the previous module. Initially, all values are reset so the read and write pointers do not advance. Once reset is set to 0 the module starts reading values. Initially there is a small lag while the pipeline is getting setup as there is no value to write if there has not been a read cycle yet. The behavior after this point can be further analyzed in Figure 10.

Figure 10.



In Figure 10. is the more typical behavior of the system. The read pointer is 3 hex values ahead of the write pointer as the process is pipelined. Bot check is set to 0 since the first 768*3 values are in the bottom of the page. Everything works as expected.

III. Conclusions

Very little deviation from calculated and experimental result. None the measured or experimental quantities, at any point, represented a danger to the stability of the circuit. All values were within the accepted 5% error. The experiment was a successful in all the sections it aimed to test the student. No issues were found in the experimental or calculation part of the experiment.

IV. Acknowldegements

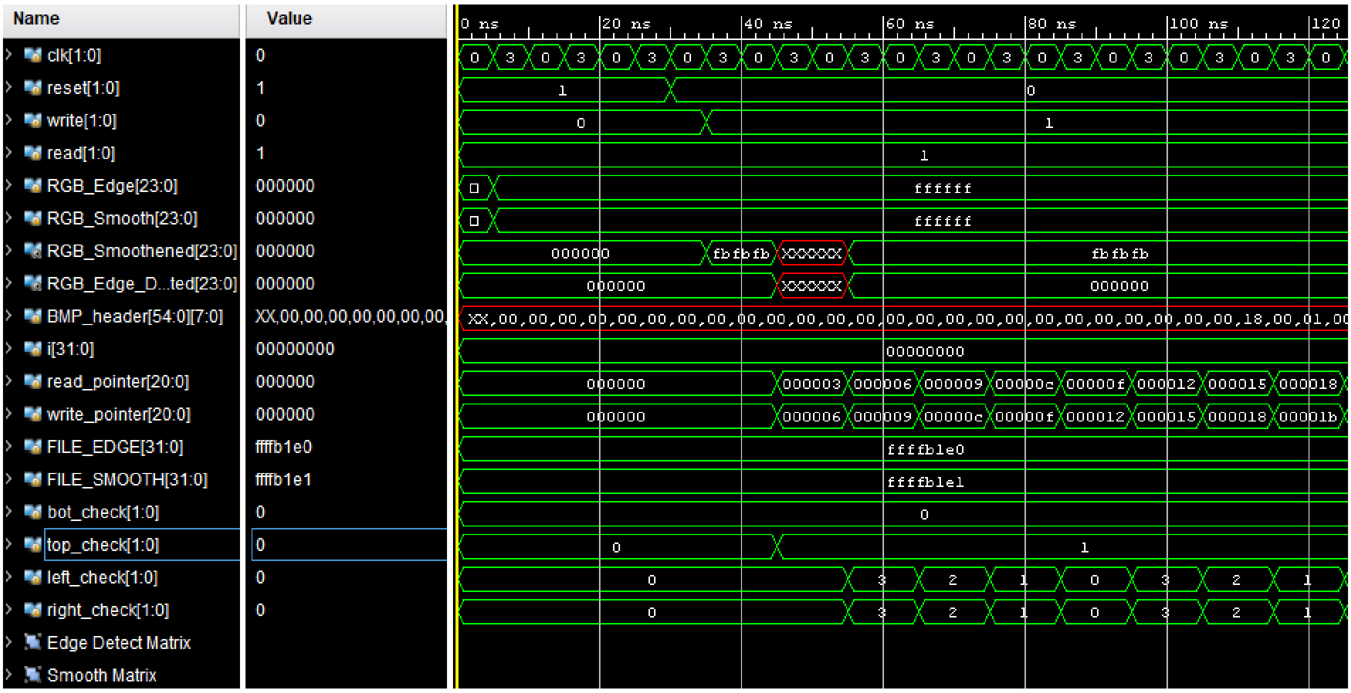
The authors wish to acknowledge Texas State University for providing a testing grounds for all calculations as well as many years of education and hard work that gave the authors the capacity to accomplish everything written down.

V. Next Steps

For those who wish to experiment with the biased circuits that were discussed. If you have any comments or wish to contact that author for other reasons, please use the email provided in the first page of this report. For the subject of your email please write the name of the report.

VI. Full Waveform

A. Beginning



B. End