ECE4063 - Image Thresholding

Emmanuel Jacyna - 24227498
James Anastasiou - 23438940

May 27, 2016

Contents

1 Introduction

This document presents our findings and designs for Assignment 2 of ECE4063. It is organised into three major sections, assumptions, solution documentation, and a discussion of potential improvements to the project. Testbench results, Signal Tap results, and RTL diagrams are included in the appendices.

The task at hand is to perform binary thresholding on an image for use in a bionic eye. The target platform is an Altera Cyclone II FPGA. In order to threshold the image, a histogram of the pixel greyscale values is calculated and used to find the 50th percentile grey scale value. This value is then used to decide whether to colour pixels white or black, performing an image threshold.

2 Assumptions

When thresholding images, we assume that an image where 99.9% of pixels are one value is meaningless to threshold. In the case of the Altera DE2 board with Terasic camera module, the small variation in value is likely to be because of pixel noise. We also believe that as the target is a human vision system, 100% accuracy is not necessarily the goal, as we prefer an image that makes sense to the human eye. This design decision allowed for a simplification of the logic circuit required to determine the most accurate thresholding value.

We also assumed that Model Sim testbenches are 100% completely accurate representations of reality. This assumption was routinely called into question, however after fixing a number of other assumptions, this sole assumption was found to be a valid assumption.

3 Documentation - Basic Requirements

3.1 RGB to Grayscale conversion

There are a number of ways to convert 12bit RGB values into 8Bit Grayscale values, these include taking the average between all 3 colour channels, desaturating the channels or performing a weighted average based on the NTSC standard. Each of these approaches has benefits and weaknesses, our design

decision was to approximate the NTSC standard weighted average via fixed shift additions. The decision to approximate the results meant the design did not need to include a multiplier or divider circuit, which significantly improved performance. Three multipliers were replaced with six additions, which if optimised appropriately could utilise Carry Save Adder hardware to improve the performance. Unfortunately as a result of approximating the NTSC values, the actual grayscale values produced to not match exactly with the theoretical values, this was found to be incorrect by a maximum of six units, which we determined to be not a significant loss in precision compared to the speedup and simplification of hardware seen, especially as the end goal of this system is image thresholding.

3.2 Total Histogram

The Total Histogram is the overall module responsible for managing the histogram generation, cumulative histogram generation, histogram storage and retrieval. This module is implemented as a two-tiered state machine, providing states and certain substates to handle edge cases that arose during testing. This module responds to changes in the frame valid and data valid signals to determine whether to calculate the histogram or the cumulative histogram. It is also able to provide direct read through capabilities, i.e. exposes the histogram RAM interfaces for display purposes.

3.2.1 Histogram Module

The histogram module provides a wrapper around a single port ram with read while write capabilities. The module is implemented via simple pipelining to minimise the latency of the circuit. The module uses a twenty bit wide RAM with an address bus width of eight bits to store the intermediate results of the histogram. The actual operation of the module is simple, with only a single edge case to handle. This is the case when the same value is read in consecutively. A race condition occurs where an old value is read from the ram before the new value has been written, this is overcome by checking for this condition, and incrementing the registered value by one before writing again, whilst also holding the registered value in a temporary register and checking if the same situation occurs again. By solving this race condition the histogram always produces the correct number of values, and behaves identically to the software (MATLAB) method.

The histogram module also handles clearing its own internal ram, which simplifies the interface for connection to the cumulative histogram module.

3.2.2 Cumulative Histogram Module

The cumulative histogram is significantly more complex than the histogram module, as it is responsible for a number of *administrative* or *house keeping* tasks. The cumulative histogram iterates over the entire contents of the Histogram ram, accumulating the output and storing the result into another RAM for reading by the histogram display modules. While the cumulative histogram module calculates the cumulative sum, it also writes the actual histogram values into one of two RAM's, essentially acting as a double buffered framebuffer. This enables the stored RAM to be used for display on the next frame whilst the new histogram is being calculated. Once the frame is no longer valid, a toggle is switched so that the correct RAM is overwritten with the new values, and the other RAM is kept for display on the next frame.

3.3 Thresholding Module

3.3.1 Description

The thresholding module is very simple. All it needs to do is take in an eight bit greyscale value and output either a white if the value is above the threshold, or a black if the value is below the threshold. This is accomplished by hooking up a comparator to a multiplexer.

3.3.2 RTL Diagram

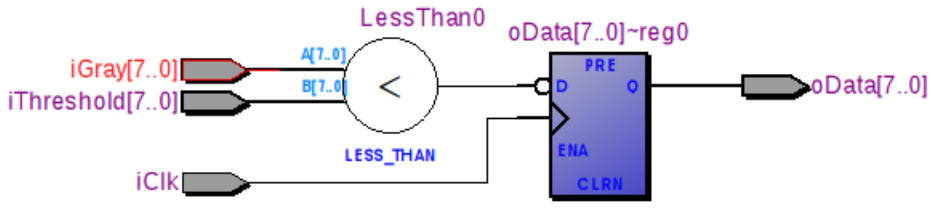

Figure 1: Thresholding module RTL

3.4 Displaying things

3.4.1 Description

In order to display the greyscale image, histogram, cumulative histogram, and thresholded image, we wrote a module to handle multiplexing between them using the switches on the DE2 board, called Arbitrator. This module takes in pixel outputs from the various modules and multiplexes them depending on the switch positions. In order to display images, we simply piggyback on the X_Cont and Y_Cont signals and modify the $wr1\_data$ and $wr2\_data$ inputs to the SDRAM with the appropriate pixel data.

Displaying the actual histogram data requires slightly more effort. First we need to extract the histogram data from the histogram RAM and convert the histogram bin contents into lines of appropriate length for display on the screen. To do this we have a module called HistogramDisplayer. This module takes in the Y_Cont signal and uses it to index the histogram RAM. Based on the value obtained from the RAM, it scales the histogram value, and uses the X_Cont signal to determine the length of the line to be displayed. The decision was made to display the histograms using the full width of the LCD screen. Whilst this does mean that some of the lowest values are not displayed (values less than ten), we found that the lowest black values, due to a quirk of the camera system, never drop below about twelve or thirteen. We thus decided to sacrifice those values in order to get a better view of the histogram as a whole.

3.4.2 RTL Diagram

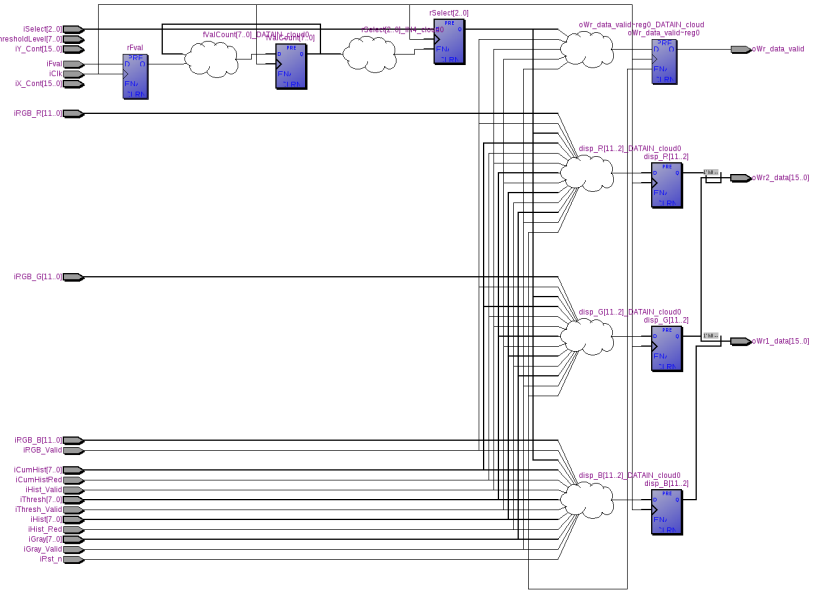

Figure 2: Arbitrator module RTL

3.4.3 RTL Diagram

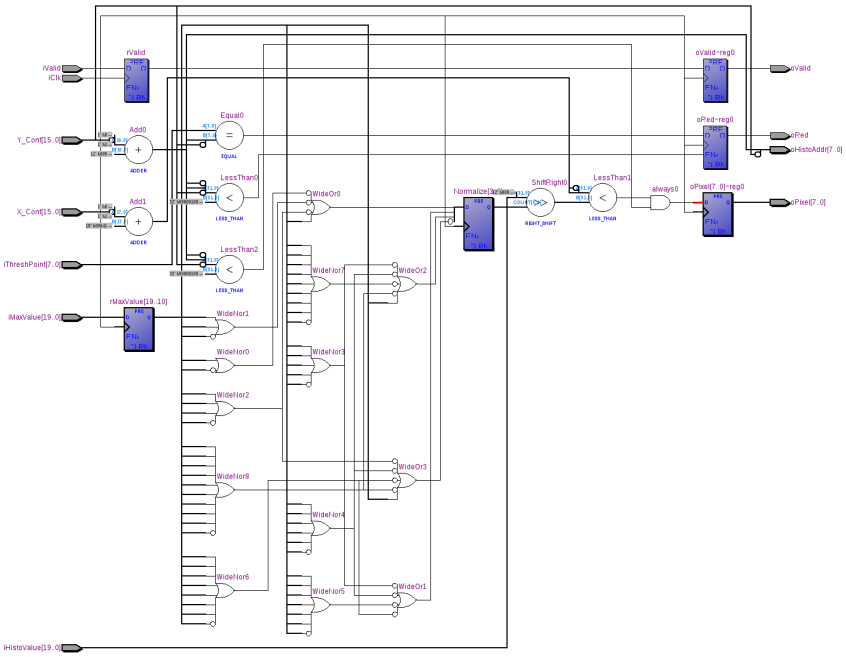

Figure 3: HistogramDisplayer module RTL

3.5 High Level Overview

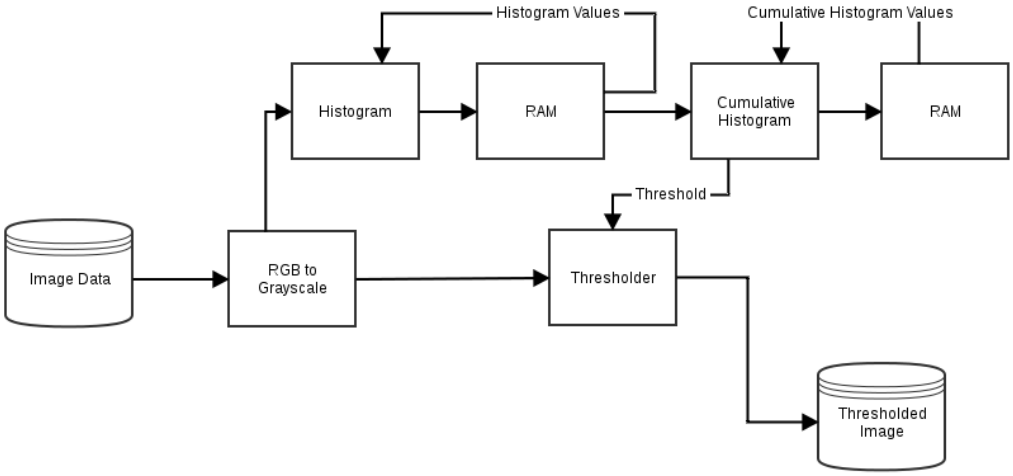

Figure 4: Overview of the image thresholding process

The toplevel module, `Total_Module` is fed RGB values from the camera. Those RGB values are then passed through an RGB to greyscale conversion to output greyscale values between 0 and 255. These greyscale values are then passed to the histogram module, which uses an internal RAM to count the number of times each greyscale value is encountered. Once the entire image has been processed, the cumulative histogram module is activated. This module uses its own internal RAM to store the cumulative sum of the histogram. Whilst it calculates the cumulative sum, it checks to see when the cumulative sum passes above the 50th percentile (the point when the sum is less than $800 * 480/2$). It saves this point as the threshold value. On the next frame, this value is then passed through to the thresholder module. The thresholder module uses the threshold it has been given to compare incoming greyscale values to the threshold. Values that are above the threshold are coloured white, and those below are coloured black, thus displaying a thresholded image.

4 Documentation - Advanced Requirements

This section discusses how we approached and achieved the advanced requirements specified in the project.

4.1 Threshold and Display the Correct Frame

In order to successfully threshold the frame the theshold was actually calculated for, we took advantage of the fact that the $wr1\_data$ and $wr2\_data$ wires were not fully packed. There were that there was exactly 8 bits of available space that was not utilised. We then recognised that in order to threshold the correct frame, the only information that needs to be delayed that can't be stored on chip was the grayscale values of the image. With this in mind we delayed the gray value and packed the $wr1\_data$ and $wr2\_data$ wires with the relevant grayscale values. So the wires are therefore containing data of the form `16'bXGGG-GGBB-BBBB-BBXX` and `16'bXGGG-XXRR-RRRR-RRXX` where `8'bXXXX-XXXX` is the 8 bit grayscale value, split across the 8 available bits of $wr1\_data$ and $wr2\_data$.

On the read side of the SDRAM FIFO, we intercepted the data, unpacked it and ran the resulting grayscale through a thresholder which utilises a stored threshold value generated by the total module. We only enable the display of the threshold interception when the appropriate switch is high, thus providing the delayed outcome.

4.2 Divided Threshold

In order to divide the display into two equal subwindows of half width the `Y_Cont` value was used to split the display in half, by a simple comparison with the halfway value of 240, if `Y_Cont` was found to be larger than 240, then the larger threshold was used, if it was found to be smaller, then the smaller threshold was to be used. This task only required a small alteration to the `Cumulative_Histogram` and `ThresholdDisplayer` modules, to account for the increased number of threshold values to be generated and used.

4.3 Smooth Divided Threshold

In order to remove the block artifacts generated by the significant step difference between the two thresholds, the middle 128 (in our final version,

actually 256) pixels were thresholded via a linear interpolation between the two thresholds. The step size required to linearly step from threshold two to threshold one was calculated as:

$$step = \frac{T2 - T1}{128} \tag{1}$$

The advantage to using an interpolation window size of 128 is that the division by 128 can instead be achieved by a right shift of 7 bits. However, using this method naively with 8 bit numbers quickly results in very poor interpolation, since the fractional bits disappear when the threshold values are close together. In order to resolve this problem, we left shifted values during calculations, and right shifted them back to the correct magnitude when using them for display.

To clarify, we first calculated the delta $T2 - T1$, then shifted it left by 24 bits to produce a 32 bit number. This provides 24 fractional bits, plenty to work with. We then divided delta by 128 by right shifting it 7 bits. Finally, when stepping through the image, we added the left shifted step size to the left shifted threshold as we progressed along the Y axis, right shifting the resulting threshold back to 8 bits for use in the thresholding module. Using this method allowed us to preserve the fractional bits, helping us to achieve a smooth linear interpolation. In the final product we used a window 256 pixels wide instead of only 128, since the larger window provided a much smoother interpolation effect.

Our implementation of fractional bits was very inefficient, significant improvements could be found by only shifting left by the absolutely minimum amount needed, the extra bits gained by a shift of 24 instead of 7 (or 8 in our case) provide no extra precision in our use case. The extra bits only serve to slow down that aspect of the circuit, which is a clear example of something that could be easily optimised away.

5 Acknowledgements

We strived to produce only original material, however, we do acknowledge the contributions of others. We obtained some inspiration for the histogram module from a paper published by Maggiani et.al. [1]. This paper suggested writing a naive histogram state machine that ignored the consecutive

edge case, and instead used multiple histogram cells, essentially independent histograms with indepent RAMs to hide the RAM access latency and avoid overwriting consecutive values. We did try this method initially, but believed that better use of resources could be obtained by limiting ourselves to one RAM and using a temporary register to handle the consecutive values case.

We also obtained ideas from the lectures provided by Dr. David Boland and Dr. Lindsay Kleeman; their suggestions were quite valuable and helped to keep us on track. We also had help from Sam Theodoulou, he gave us a helpful tutorial on the correct usage of Signal Tap.

6 Improvements

Get tbs working sooner. Test properly. Find out mroe about the touch tcon.

7 Conclusion

References

[1] L. Maggiani, C. Salvadori, M. Petracca, P. Pagano, and R. Saletti. Reconfigurable architecture for computing histograms in real-time tailored to fpga-based smart camera. In *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pages 1042–1046, June 2014.

8 Appendix A - Testbench Results

Testbenching was performed mainly by reading in image data stored in a text file, then writing the results of testbenching to another text file. This was then read in by MATLAB and MATLAB scripts were used to ensure that the testbench result matched the expected result. This appendix provides images of the results obtained from these testbenches.

8.1 RGB2GRAY

In order to test the `RGB2GRAY` module, the greyscale conversion, a testbench was written that took as inputs the RGB values of an image, and outputted a file containing the grey values calculated by the modelsim simulation. This was then compared to the MATLAB rgb2gray function results. The mean difference between the MATLAB and Verilog functions was only 6 units, not enough to visibly affect the image output. This was deemed to be well within tolerable bounds. The figures below show the MATLAB image, the Verilog image, and the difference between the two.

Figure 5: MATLAB rgb2gray result

Figure 6: RGB2GRAY.v result

Figure 7: Difference between the two

8.2 Histogram

In order to test the `Histogram` module, a testbench was written to read in precalculated greyscale values and output a histogram. This histogram was then compared to the histogram generated by matlab. The MATLAB code and Verilog code give identical histograms.

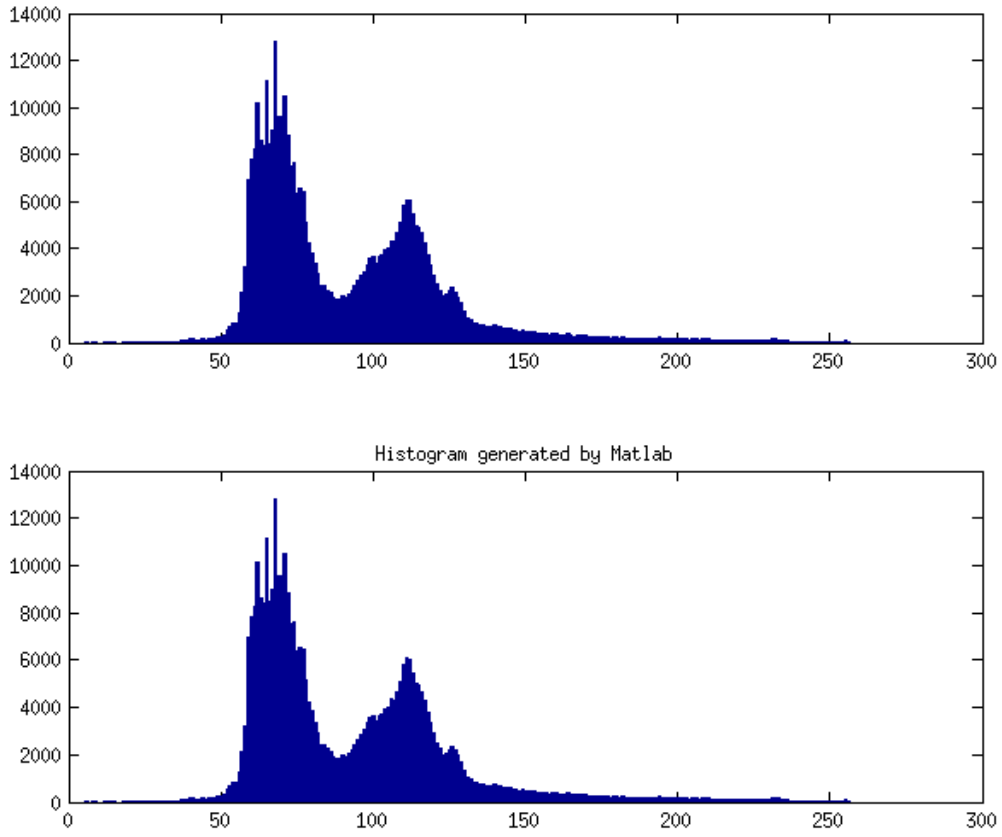

Figure 8: Comparison of histogram generated by Histogram module and by matlab

8.3 CumulativeHistogram

d In order to test the `CumulativeHistogram` module, a testbench was written to read in a histogram (generated by `Histogram` module). The cumulative histogram was then calculated and compared to that generated by matlab. The two cumulative histograms are identical.

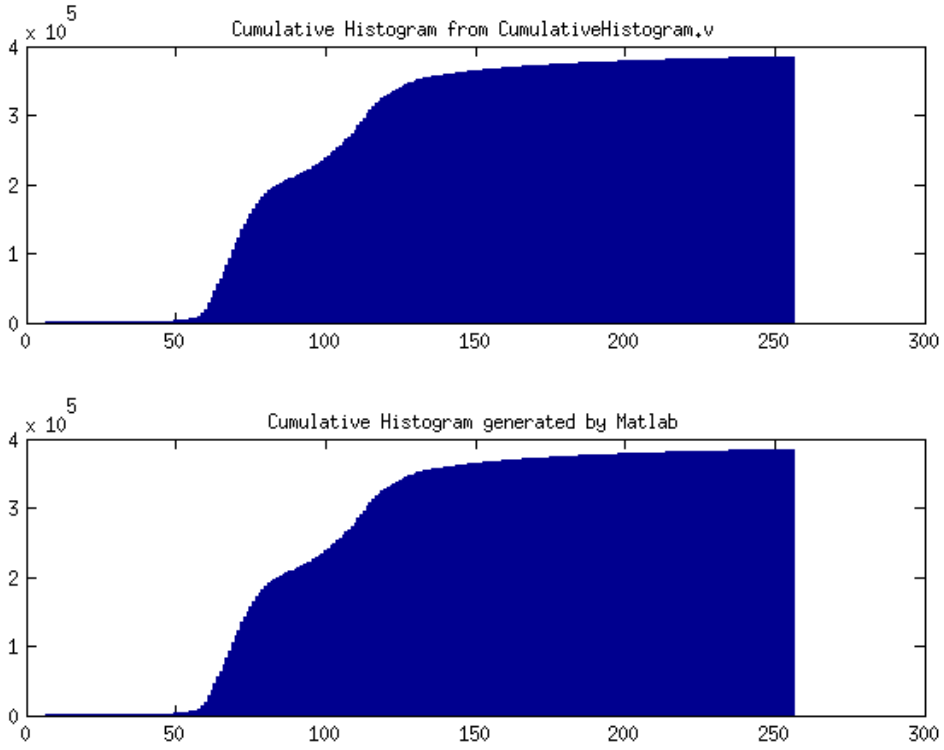

Figure 9: Comparison of cumulative histogram generated by Cumulative Histogram module and by matlab

8.4 Total_Histogram

In order to test the `Total_Histogram` module, a testbench was written to read in precalculated greyscale values and output a histogram and a cumulated histogram. The result is the same as those calculated with the separate Histogram and CumulativeHistogram testbenches.

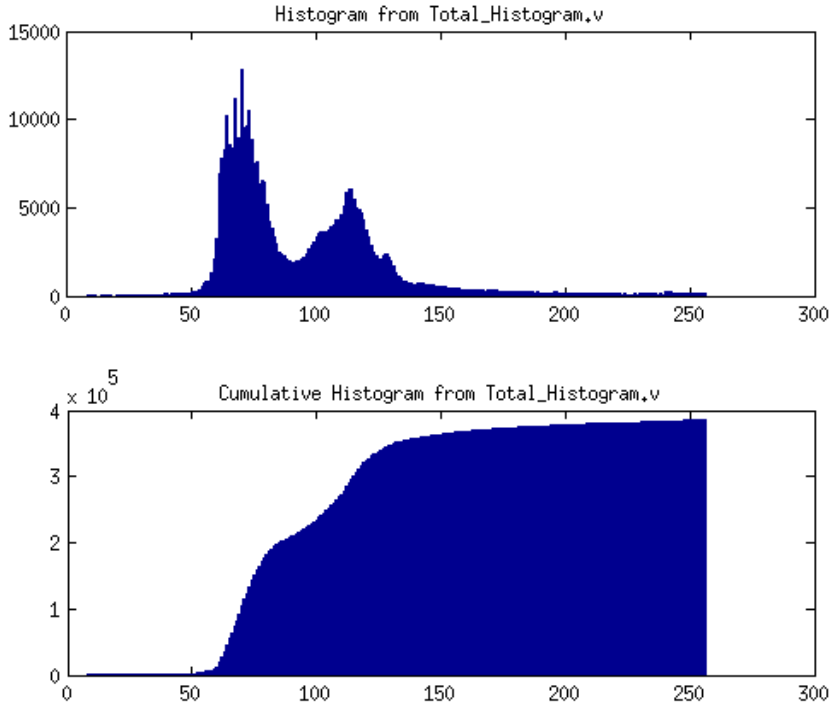

Figure 10: Cumulative Histogram and Histogram generated by `Total_Histogram` module

8.5 HistogramDisplayer

In order to test the `HistogramDisplayer` module, a testbench was written to read in histogram values and output the same image that would be outputted on the LCD screen. MATLAB was then used to display the image. The image displayed clearly reflects the histograms displayed above, ignoring normalization.

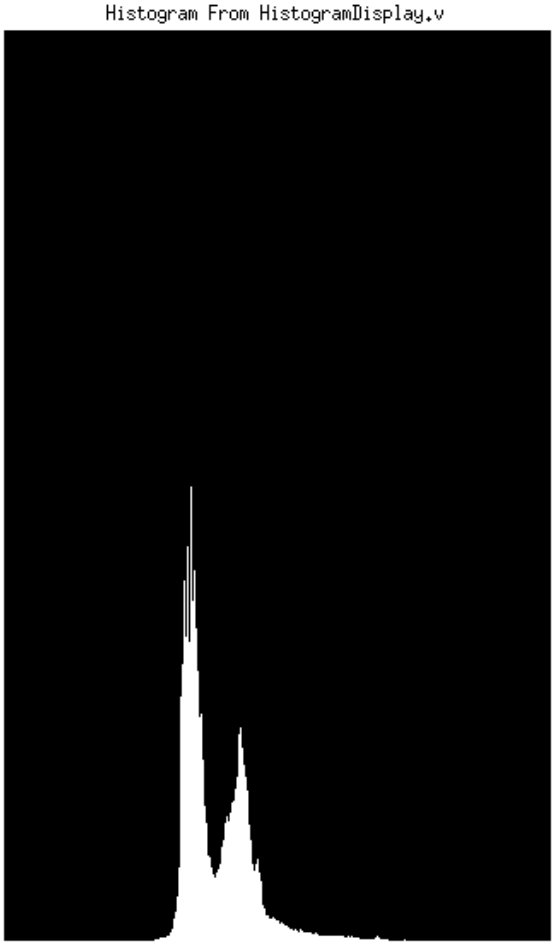

Figure 11: Image generated by the `Histogram Displayer module`

8.6 Total_Module

In order to test `Total_Module`, we wrote a testbench that reads in RGB values from a file, then passes those through to an instantiation of `Total_Module` along with appropriate data valid signals. The testbench then changes the switch settings and outputs image data to file. A matlab script is used to read in this data and display images. The results are below. Clearly the module behaves correctly for the christmas shark input image, confirming our work in simulation.

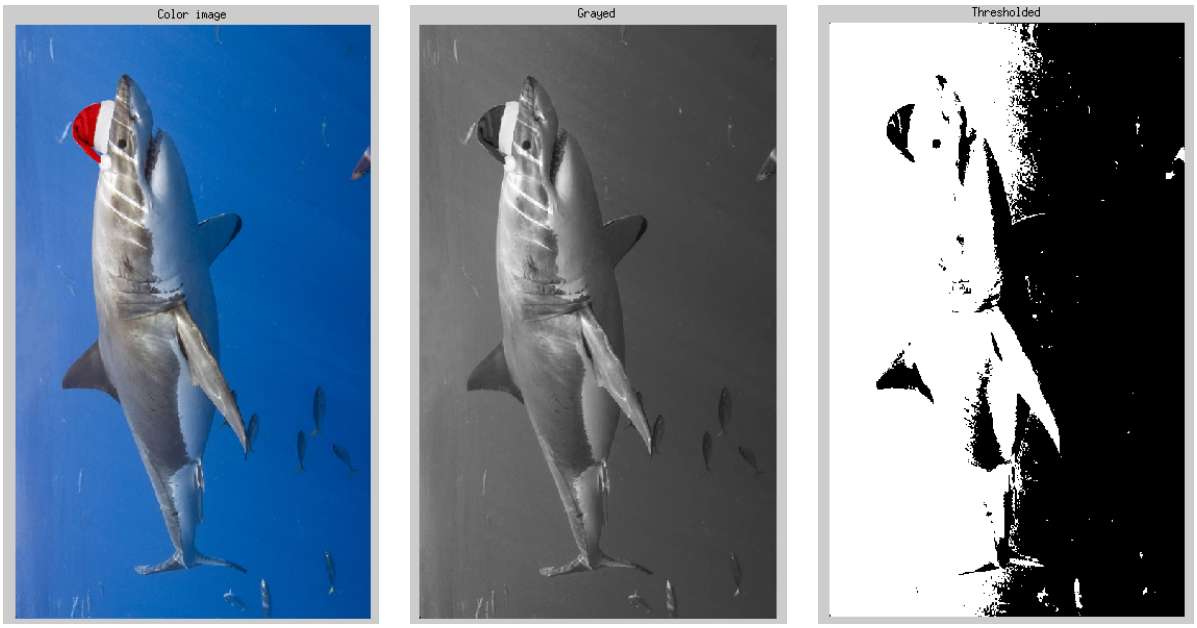

Figure 12: RGB pass through, grayscale image, and thresholded image

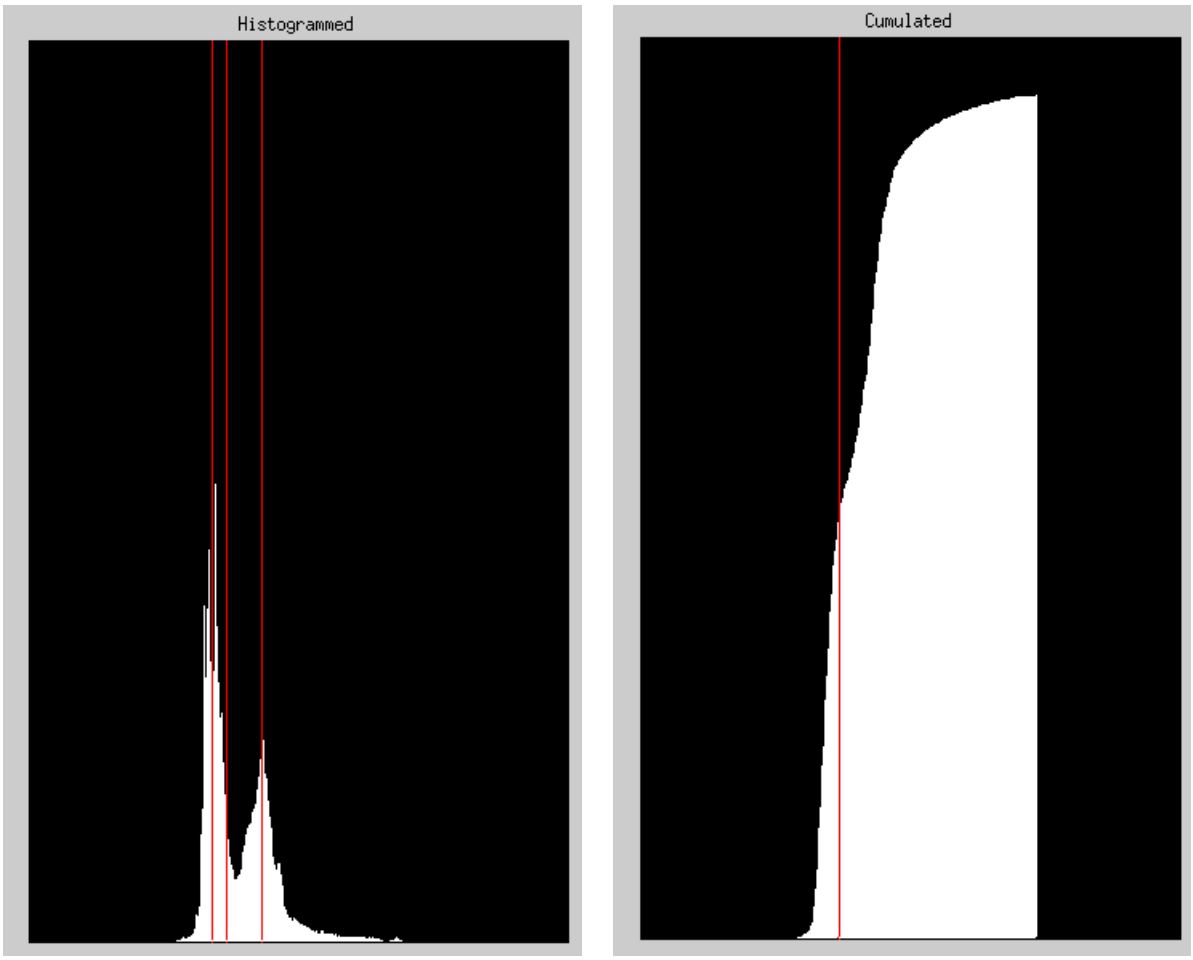

Figure 13: Histogram and Cumulated Histogram generated by `Total_Module`

Figure 14: Block Threshold and Interpolated Threshold generated by `Total_Module`

8.7 MultiThresh

In order to test the `MultiThresh` module, a testbench was written to read in greyscale values and run them through the MultiThresh module. After displaying the resulting image data in MATLAB we were able to verify that the module worked correctly.

Figure 15: Block Threshold and Interpolated Threshold generated by `MultiThresh`

8.8 Delayed Frame

In order to test the delayed frame setup, we needed to simulate the RAM and the image display device. Two RAM modules were instantiated and used to simulate the SDRAM. Then, a number of images were processed through the `Total_Module` and the results shown below. First a shark image was processed and displayed as greyscale. The module calculated the appropriate threshold for the shark image. Next, the lenna image was processed through the module and thresholded, however, the threshold used is for the shark threshold from prior. It can be clearly seen in figure 16 that the threshold is not appropriate for lenna. Finally, the shark frame was processed through the module again, however, the switch was set to delayed thresholding. The delayed values were read from the RAM and lenna was thresholded again, this time with the appropriate value. This result is as expected and demonstrates that the delayed thresholding works correctly.

Figure 16: Frame 1: Shark warmup frame. Frame 2: Lenna thresholded with Shark threshold. Frame 3: Lenna frame delayed and thresholded correctly.

9 Appendix B - Signal Tap and Simulation Results

9.1 Histogram

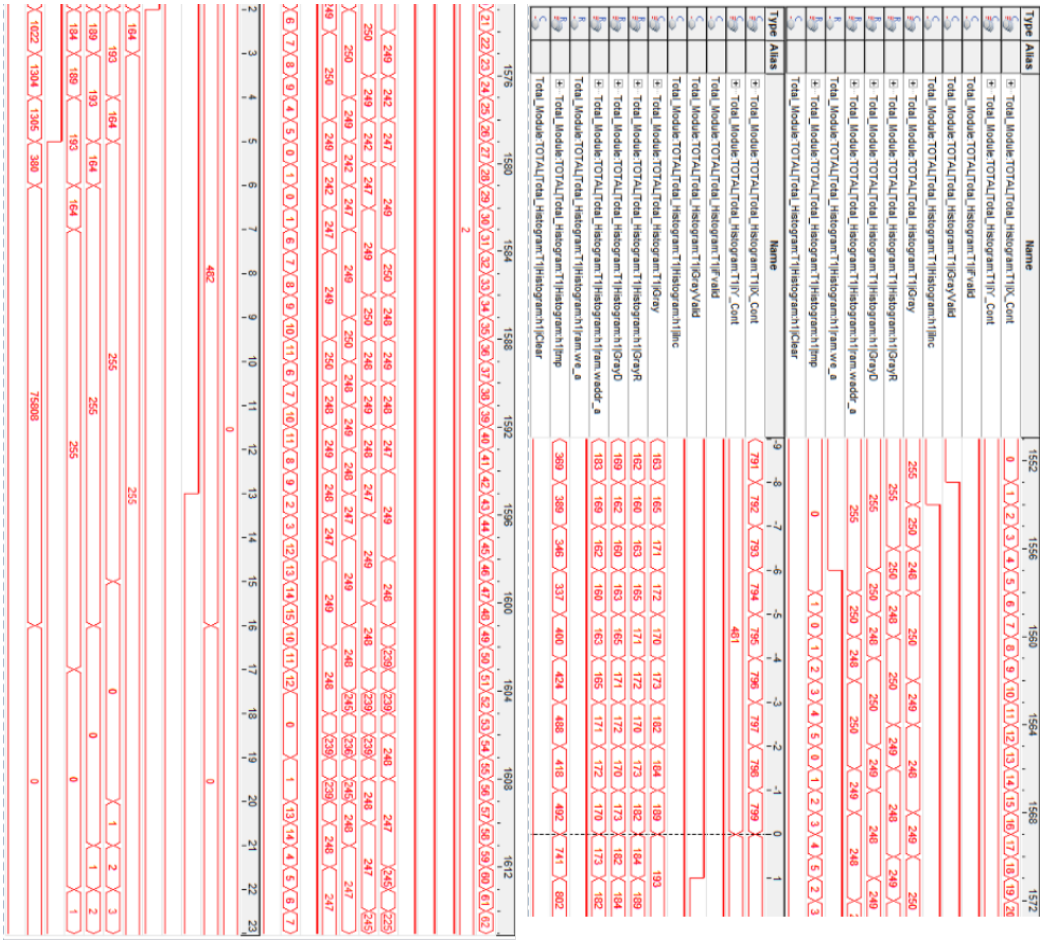

Figure 17: Normal Histogram Operation - Signal Tap

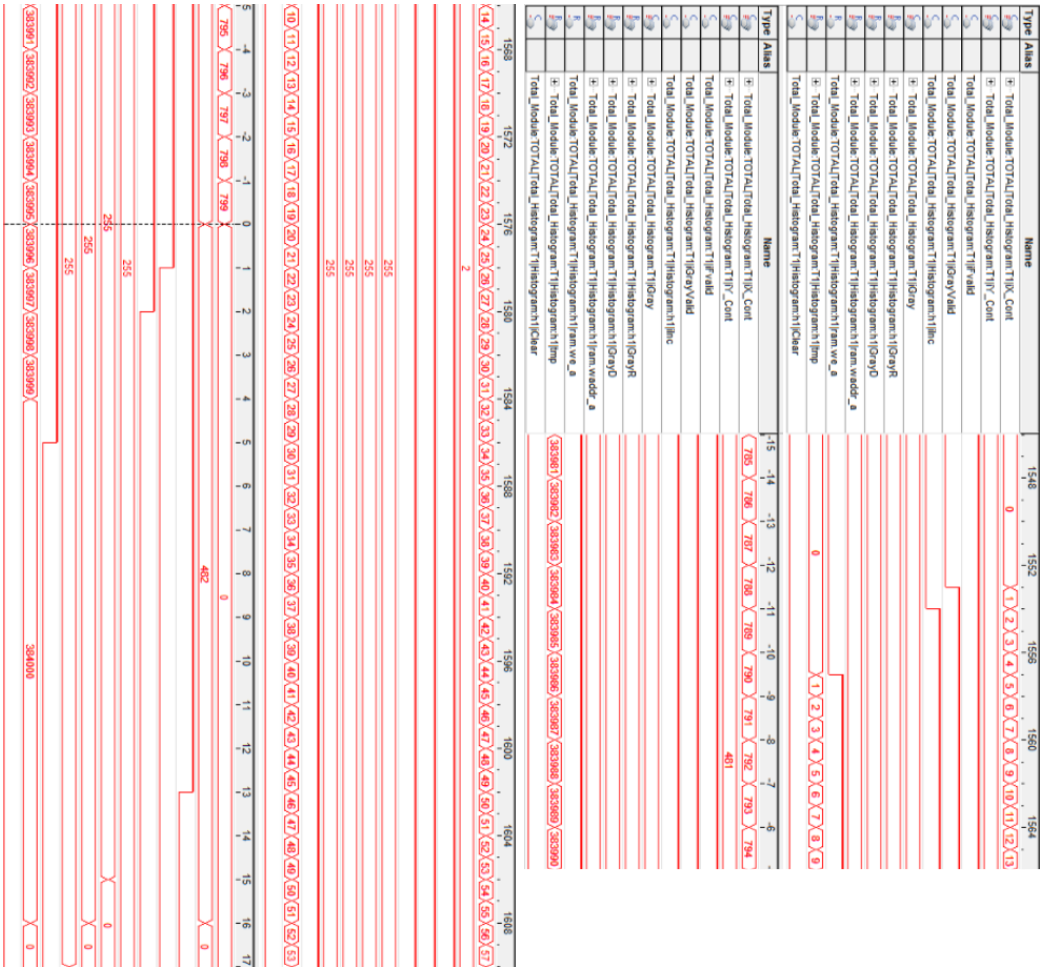

Figure 18: Edge Case Histogram Operation - Signal Tap

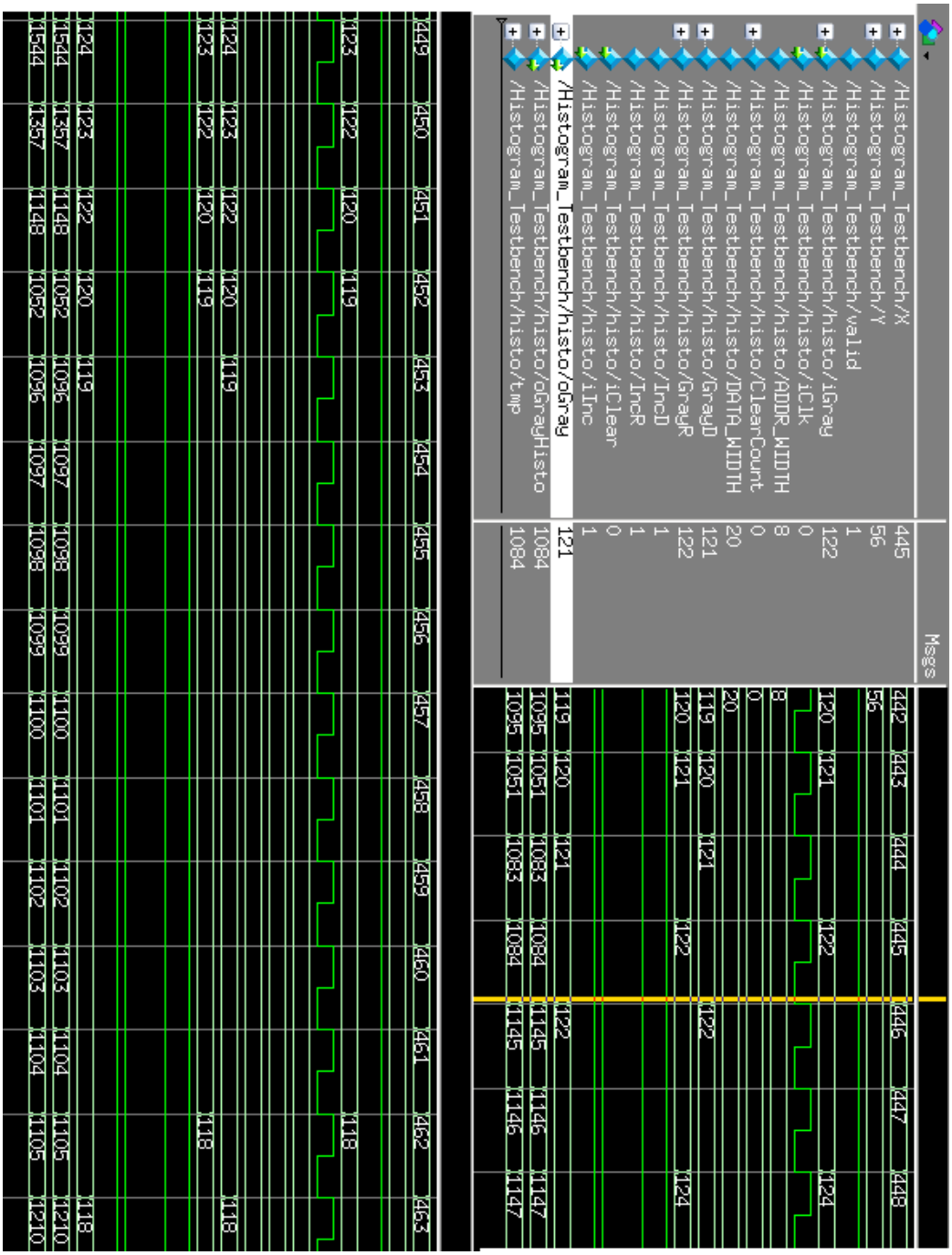

Figure 19: Histogram Operation - ModelSim

9.2 Cumulative Histogram

Figure 20: Cumulative Histogram Operation - Signal Tap

Figure 21: Cumulative Histogram Operation - ModelSim