Lab 3 Report

Group 3: Dennis Zang (704766877), Daniel Park (104809832),

Samuel Pando (904809319)

November 20, 2020

Image Processing

[Components and Implementation Logic]

In this lab, all modules shared a few basic similarities, so I'll discuss them here before diving into the individual modules themselves. Most of the modules (except for resize and brightness) shared the same inputs. First, all of them had a **clk** signal. Then there was **image_input[7:0]**, a byte sized input representing one pixel from the input image. The next was **enable**, which when set to 1 would continue to read the values from image_input into a register keeping track of all the values. And finally, **enable_process** which, when set to 1 while enable is 0, will process the stored input values according to the filter. The two outputs were **image_ouput[7:0]** (the same as input but for writing to a file) and **finish** (when all processes are done). As previously stated, all the modules made use of registers to hold the image files read in.

Figure 1.1: The original image of interest. This is the image that will go through the noise filter, and the noisy image will go through all the other filters.

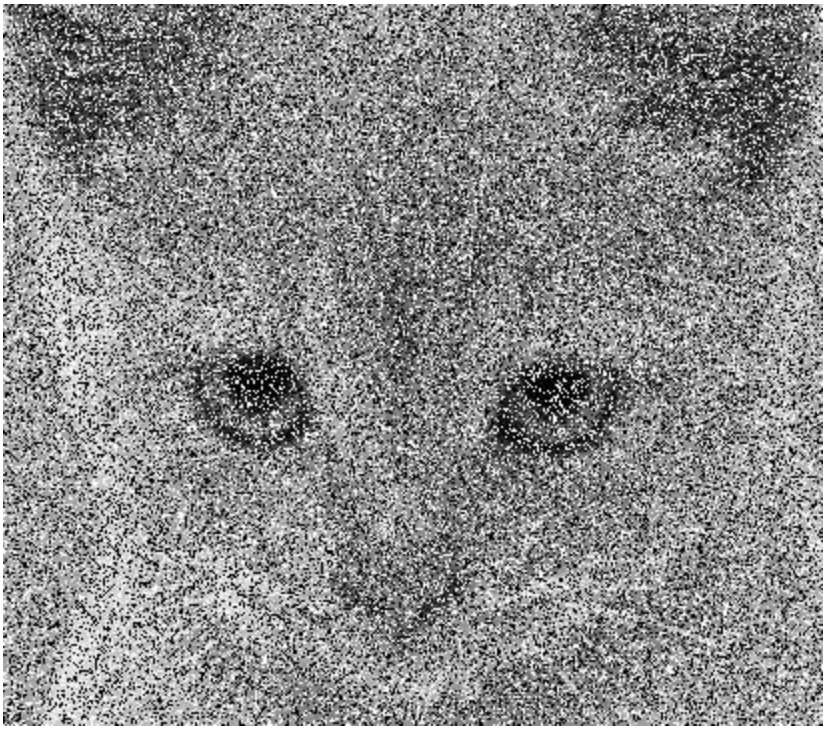

Figure 1.2: The image that corresponds to the noisy_image text file. As we can see, a decent amount of noise has been introduced in the image, so the filters will potentially show their advantages more prominently.

[Median]

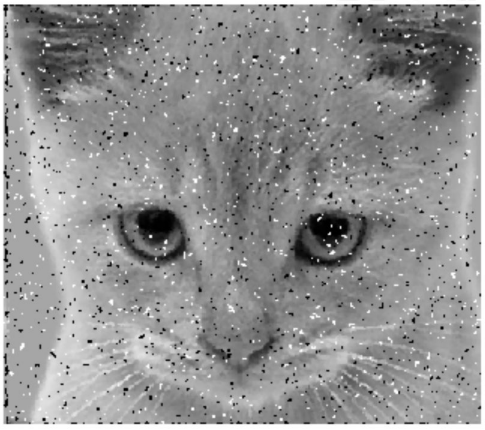

Figure 1.3: The image after applying the median filter. The image is much cleaner compared to the original noisy_image.

The median filter works by taking the median of the neighboring values of each pixel. Our implementation uses a 3x3 pixel square for finding neighbors and pixels outside the range of the image were considered to be 0 which slightly hampered the filter on the

image's edges. Although pepper flakes are still visible, the image is much more recognizable than previously.

[Low Pass]

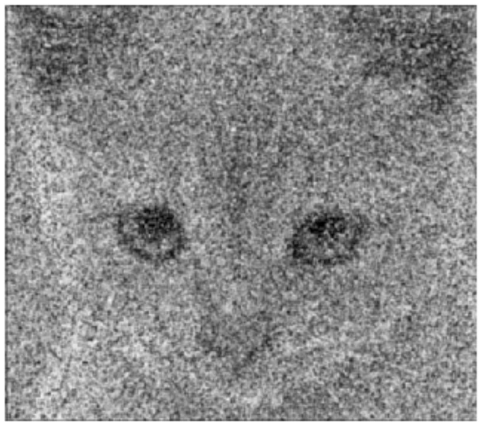

Figure 1.4: The image after applying the low pass filter. The image is recognizable, but is blurry and lacks contrast.

The low pass filter is similar to the median filter in that each pixel is influenced by its neighbors. However, the effect of each pixel is more exaggerated than before, resulting in a blurry, muddy image. Our implementation takes all values within a 3x3 pixel square around each pixel and averages the values to produce the pixel's new value. Values outside the range of the image were set to 0. This produces a thin darker border around the entire image.

[High Pass]

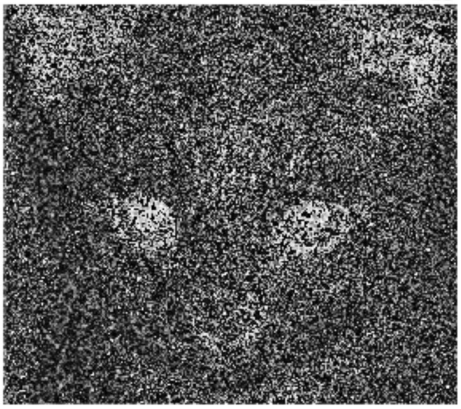

Figure 1.5: The image after applying the high pass filter. We can see that, as expected, the high pass filter emphasizes contrast.

The high pass filter is practically identical to the low pass filter. In fact, the only difference is in the convolution kernel used. Whereas low pass averages values to bring higher values down to the lower value levels, high pass emphasizes the higher values by giving them higher weight in the convolution color. This shows in the image, where except for the brightness values, everything else is muted. Since the pixels out of range of the window were set to 0, the pixels on the border will naturally stand out a bit more.

[Resize]

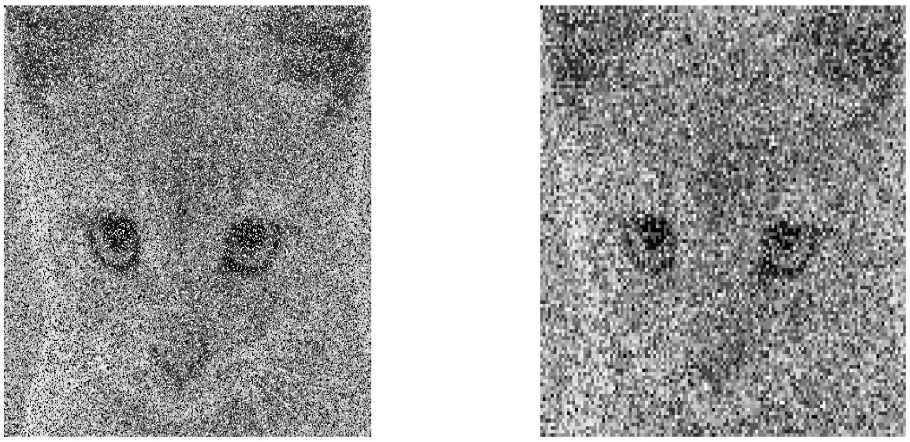

Figure 1.6: An expanded picture and a shrunken picture, side-by-side. The expanded picture keeps the original resolution, but the shrunken picture loses some resolution. Although the final picture generated from Matlab is the same size, the file sizes of the byte data Verilog output has a scale difference.

The resize filter is similar to the median filter and high/low-pass filters in that we need to make use of a window of neighbors for each pixel in order to scale the original image. In this case, we went with a simple integer scaling (both for shrinking and expanding the original image) by either generalizing a window with only its average into a single pixel for shrinking or multiplying an original pixel into a square window of a specified size. No matter if the picture is to be shrunk or expanded, the averages for each possible window (each window is associated with the pixel on the upper-left corner) is calculated, even though these averages would only be used for shrinking the image. Then depending on if the image is to be shrunk or expanded, the output is chosen by a single conditional/mux.

[Brightness]

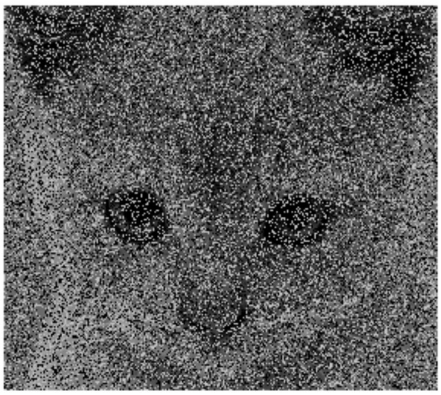

Figure 1.7: The image after applying the brightness filter. In this case, do_bright was set to 0, so instead of brightening the image, we actually darken it.

Brightness was the simplest filter. Building on the methods used to access the register from the other modules, the brightness module just had to add or subtract a value. Despite this, brightness did have two more inputs than most of the other modules. It had a byte sized **bright[7:0]** input that gave the amount to increase or decrease in brightness as well as a **do_bright** input that determined whether the filter would brighten or darken. Limits were set so that we could not darken to a negative value or brighten past the brightest possible value.

[Waveforms/Testbench]

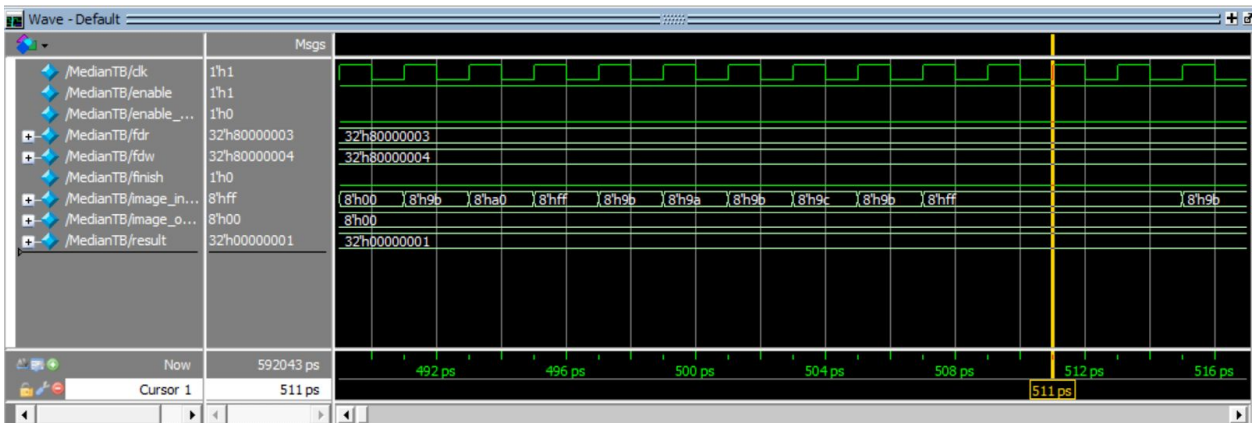

Figure 2.1: The waveform produced when passing bytes to be saved by the internal register. Notice the parameters controlling the inputs.

The waveform above depicts the primary state of this project which involved storing values within an internal register. The “enable” parameter instructs the module to store the passed “image_input,” and to delay outputting important information. Since the passed input is the noisy_image, there are many 8’h00 and 8’hff mixed into the data. Finally, “result” shows that the values are still being read from the text file.

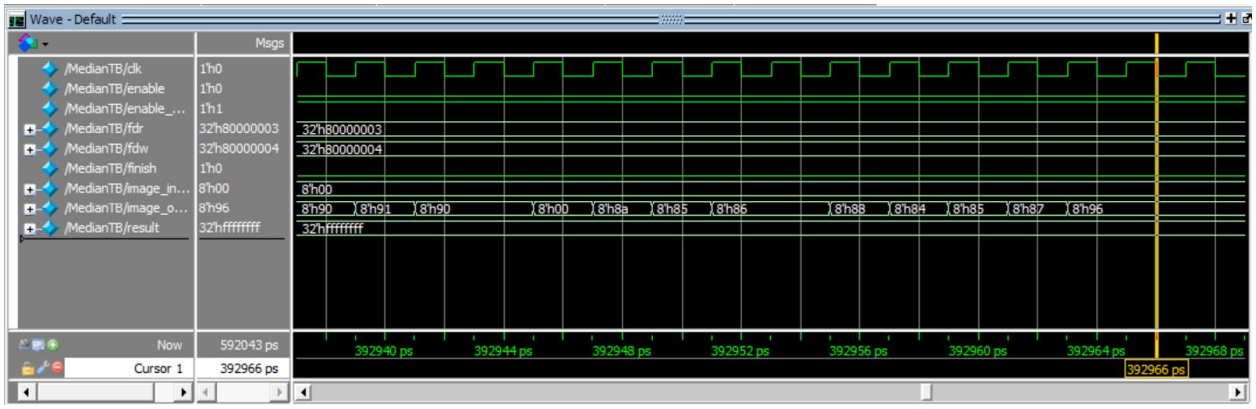

Figure 2.2: The waveform produced when receiving output from the module after applying the filter. Notice the parameters’ new values.

This waveform shows an example output when receiving data from the module after the filter has been applied. The “enable” parameter is swapped for the “enable_process” parameter. “image_input” no longer passes any data and instead “image_output” passes the new image byte by byte.

[Challenges Faced]

The main challenges of this project were implementing I/O and coding without familiar data structures. Although Verilog has many similarities to C, this project required I/O for both Verilog and Matlab which was unfamiliar at first. Furthermore, data structures commonly found in C/C++ were unavailable in Verilog which increased the difficulty of the project.

In addition, there were some issues that came up with unknown values being read into and out of some of the registers in our modules. There were different cases depending on the module. In one instance, unknown values were output at the beginning and the end of the text file, but they did not impact the actual image text itself. In another case, the unknown value did appear among the image bytes. Since we had so much difficulty with this, we ended up just building a net into our testbench to catch and replace unwanted unknown values. In the long run,

it seems like it ended up working out, though I wish we could have dealt with the problem more thoroughly (which we likely would have done if we had a bit more time).

Another interesting case that came up was the limitations on how many modules can be instantiated. An approach that was tried was instantiating multiple window modules for a row of pixels (about 500, done using a “generate” block) in order to greatly improve the throughput like with modern-day GPUs and their thousands of logical cores. However, this just led to Modelsim crashing due to lack of sufficient memory, which should have been expected.

Member Contributions:

Daniel Park: Median/Low Pass/Testbench

Dennis Zang: Resize

Samuel Pando: High Pass/Brightness