Minilab 2 Report

Harrison Doll
Jake Neau
ECE554 - UW-Madison

Github Repository

The team Github repository can be accessed via the following link:
https://github.com/fuzzy41316/ECE554_minilab2
.v or .sv files created and/or modified by us:

- testbench.sv
- DE1_SoC_CAMERA_ref\DE1_SoC_CAMERA.v
- DE1_SoC_CAMERA_ref\v\EDGE_DETECT.sv
- DE1_SoC_CAMERA_ref\v\Line_Buffer2.v
- DE1_SoC_CAMERA_ref\v\Line_Buffer3.v

.qip files created and used by us:

- DE1_SoC_CAMERA_ref\v\Line_Buffer2.qip
- DE1_SoC_CAMERA_ref\v\Line_Buffer3.qip

The rest of the files were copied from the minilab reference download.

Implementation

Grayscale implementation

We modified the main wrapper of the camera, DE1_SoC_CAMERA.v, to accept input from SW[1], SW[2], and SW[3]. SW[1] was used to swap the valid and pixel output values between our new image processing module, EDGE_DETECT, and RAW2RGB, which would be inputted into SDRAM to be fed into the screen. SW[1] turns on and off grayscale this way. Our grayscale implementation was very straightforward. We instantiated the line buffer used by RAW2RGB, which was Line_Buffer1.v This shift register had 1 tap spaced 1280 bits apart, the length of the row of pixels. We assigned our pixel data mDATA to that tap, and flopped it as well to mDATA_ff. Then, we took the input data, iDATA, and flopped that to iDATA_ff. To create grayscale, we needed to add those values together and take the average of them. (mDATA + mDATA_ff + iDATA + iDATA_ff) / 4. We then assigned that value to oRed, oGreen, and oBlue; of which was inputted to SDRAM to be outputted via VGA to the screen.

Edge detection implementation

Firstly, we input SW[2] to EDGE_DETECT and when turned on would enable our edge detection. SW[3] was used to select between vertical and horizontal edge detection in our module. To implement edge detection, we generated two more files: *Line_Buffer2*, and *Line_Buffer3*. Contrary to Line_Buffer1, these had a length of 640 in their line buffers, because converting to RGB or grayscale would halve the row and column pixels; and that's why we had to generate new line buffers. To perform the edge detection on this, we needed to create a 3x3 matrix to multiply with the provided Sobel kernels. We named this variable, *convolution_input*, which was a 12-bit wide 3x3 array. We assigned convolution_input[2][2] to the current gray pixel data, because that's where we start from when reading in pixels: the top right. The tap for that

shift register is then convolution_input[1][2]. Then, we routed that to the input of the next shift register, where we assigned the second tap to be convolution_input[0][2]. The valid enable signals for those two shift registers were pipelined from the valid enable of the grayscale calculations. Then we did this calculation on the pixels:

$$y[i][j] = \sum_{m=-1}^{1} \sum_{n=-1}^{1} x[i-m][j-n] \times h[m][n]$$

h[m][n] was the Sobel filters, and either vertical (first one) or horizontal filtering (second one):

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

To mimic this equation we assigned the pixel value as follows:

For horizontal edge detection:

*convolution_input[2][0] + 2 * convolution_input[2][1] + convolution_input[2][2] - convolution_input[0][0] - 2 * convolution_input[0][1] - convolution_input[0][2]*

For vertical edge detection:

*convolution_input[0][2] + 2 * convolution_input[1][2] + convolution_input[2][2] - convolution_input[0][0] - 2 * convolution_input[1][0] - convolution_input[2][0]*

To deal with the fact that this convolution may end up negative, we made our 12-bit pixel value 14-bits to solve the overflow. If the 14th bit was 1, then overflow occurred, and we applied two's complement. And finally, just like the grayscale, oRed, oGreen, and oBlue were assigned this value, and this was propagated to the screen.

Testbench

In the testbench I checked that SW[1] correctly switches the pixel values from RGB, to the values outputted by EDGE_DETECT which were grayscale, and that these values were pipelined correctly into the SDRAM module which dealt with the screen output. I also checked that when SW[1] was turned off, that it correctly resumed the RGB values again, without being reset. I also checked that SW[2] enables edge detection, performs the correct math on the pixels, and outputs corresponding to SW[3] as either vertical or horizontal edge detection. For further analysis of the testbench, simulation_log was added to the main folder of the github repository. This contains the simulation results in the form of a transcript, and how I tested them specifically.

Quartus Flow Summary

You can reference the full .RPT file in our github repository. It's called DE1_SoC_CAMERA-Flow Summary.rpt in the main folder. According to the flow summary, we synthesized and used 783 / 32,070 ALMs, 0/87 DSP blocks, and 89,992 / 4,065,280 BRAMs.

Problems & Solutions

We did not encounter any immediate problems due to the approach that was taken towards solving the problem. We chose to understand the problem conceptually, and learn how all of the modules were interconnected with each other. Then, we moved on to understanding how the pixels were loaded into the screen. This was done through trial and error by manually setting pixels to all red to see where they showed up. After figuring out where pixels come in, how they are loaded, and where they go in the camera wrapper, we finally started coding. We had a slight issue with the grayscale implementation. We forgot to set the green pixel output to our calculated result, and that caused the screen to have a purple filter over it instead of grayscale. That was a quick solve and fix. We also took the convolution calculations and application that I explained earlier to pen and paper before coding it. We didn't have any issues with the math, but we did have an error with the valid signal that we inserted into the line buffers. We were using the wrong one and that messed up our vertical and horizontal edges by duplicating them on the screen. It was slow to figure that out, but it was one signal that we were setting wrong, and once we fixed that, it worked.