

MP-3 Write-Up

Blake VERMEER

Kris HALL

Rohit ZAMBRE

November 5, 2014

Date Due: October 31, 2014

Instructors: Joseph Zambreno

1 Overview

In this lab we learned about the LEON processor on how to create and integrate a co-process with the LEON processor.

2 Software Grayscale Conversion

After getting acquainted with the build environment for the LEON processor, we dived into the *frameloop.c* code to see how it works. First we started by examining the main function and dissecting how it works. The main function of *frameloop.c* first sets up the environment by enabling the coprocessor, and configuring and enabling the SVGA controller. It primarily copies all (= NFRAMES) of the image frames from memory to the SVGA controller; it can do so via two modes: mode 1 and mode 2. After the completion of the transfer of all (= NFRAMES) image frames, it computes the Frame Per Second measure for the buffering process.

Next we looked at the two different modes that were present in *frameloop.c* to determine what they did. Mode 1 merely copies the current image from memory to the controller without applying any filters. Mode 2 is much more involved than mode 1. In mode 2, a filter is applied to each frame being transferred from memory to the buffer in the controller. Currently, two RGB values are being computed for each image frame and an OR is performed between the two RGB values to obtain the final the image frame that will be sent to the controller.

```
Program exited normally.
glib> load frameloop.leon
section: .text at 0x40000000, size 61392 bytes
section: .data at 0x4000efdc, size 3448 bytes
total size: 64840 bytes (756.6 kbit/s)
read 369 symbols
entry point: 0x40000000
glib> run
Allocated framebuffer at location 40010800
FPS: 0.025771

Program exited normally.
glib> █
```

Figure 1: Floating Point Software Grayscale Conversion Results

```

gllib> load frameloop.leon
section: .text at 0x40000000, size 61424 bytes
section: .data at 0x4000eff0, size 3448 bytes
total size: 64872 bytes (755.7 kbit/s)
read 370 symbols
entry point: 0x40000000
gllib> run
Allocated framebuffer at location 40010800
FPS: 4.250000

Program exited normally.
gllib> █

```

Figure 2: Fixed Point Software Grayscale Conversion Results

3 Hardware Coprocessor - Grayscale Conversion

In this section we were asked to read through the code in *coproc.vhd* and *coproc_core.vhd* and provide a description of how the controller sends data to the core. After reading over the code in *coproc.vhd* it became apparent that the LEON processor was passing the entire program instruction to the coprocessor. From there the coprocessor parses the instruction and extracts the op-code and source and destination register addresses. After getting the source register addresses it pulls the data directly from these registers. Therefore, the coprocessor has direct access to the LEON processor's registers.

We based our hardware coprocessor for the grayscale conversion on the logic we used to implement the software grayscale conversion. Here is a hardware diagram for the coprocessor we designed:

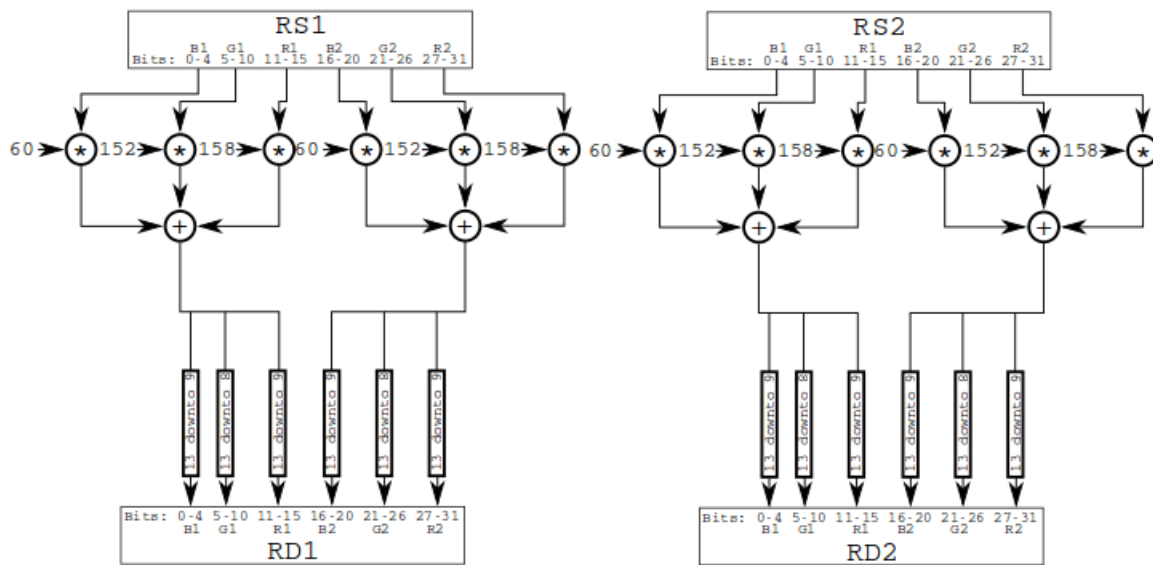


Figure 3: Hardware Grayscale Conversion Coprocessor Dataflow Diagram

The structure for the hardware coprocessor ended up being fairly simple and easily parallelized. Because of these qualities we were able to make the coprocessor complete the operation in one clock cycle. Additionally, since we were able to output a 64-bit value from the coprocessor, we are able to process four individual pixels in that single clock cycle. Including the load and storing operations (for some reason we could only get the 32-bit load/store operations to work correctly) it would only take 5 clock cycles (assuming the load and store operations take one cycle each).

```
glib> run
Allocated framebuffer at location 40010c00
FPS: 10.200000
Program exited normally.
```

Figure 4: Fixed Point Coprocessor Grayscale Conversion Results

3.1 Coprocessor Interface Instructions

The LDC - Load Coprocessor Register instruction loads a word (32-bits) from memory into a coprocessor register.

The LDDC - Load Double Coprocessor Register instruction loads a double-word (64-bits) from memory into a coprocessor register pair.

The STC - Store Coprocessor instruction stores the value of a coprocessor register into memory.

The STDC - Store Double Coprocessor instruction stores the value of a coprocessory register pair (with consecutive odd and even addresses) into memory.

The opcode for both the LD and ST class of operations is 11 but they are differentiated by their op3 value.

The CPOP1 - Coprocessor Operate 8 bits (opcode + op3) to denote that the instruction needs to be executed on the coprocessor. The remaining 24 bits are used to signify the 2 source registers, 1 destination register and the operation to be performed. All the register addresses refer to the coprocessor registers.

4 Software Edge Detection

For this part, we created a new Mode 5 to implement the edge detection in software. In this mode, we first called the software implementation of the grayscale conversion and then applied the Laplacian edge detector on the image, going left to right and top to bottom. We implemented this using two nested for loops, with the inner one traversing across the columns and the inner one traversing across the rows.

Here is a screenshot of our performance results of the software edge detection:

```
glib> run
Allocated framebuffer at location 40011000
FPS: 2.318182
Program exited normally.
```

Figure 5: Software Laplace Edge Detection Results

5 Hardware Coprocessor - Edge Detection

In this section, the task given is to modify the coprocessor to be able to perform greyscale edge detection. In order to implement this ability correctly, some changes had to be made to the files *coproc.h*, *coproc.vhd*, and *coproc_core.vhd*. The changes to the files were to provide two new instructions, CP_PRELOAD, and CP_EDGE_DETECT.

In order to increase throughput, and potentially accelerate the process, the conclusion was made that the best approach to performing edge detection is as follows. Since the laplacian edge detection operation performed on a pixel requires the values of the eight pixels that surround it, a design decision made was that edge detection will not be performed on the pixels on the edge of the screen. Another design decision that

was made was to load in six pixels at a time, to reduce the number of loads that were called, and allows for the loading and storing of 32-bit values. The six pixels that are loaded in to the coprocessor are the $[y-1][x-1]$, $[y-1][x]$, $[y][x-1]$, $[y][x]$, $[y+1][x-1]$, and $[y+1][x]$ pixels used for the calculation.

Due to the pixel loading requirement outlined above, each time the algorithm reaches the next row in the frame, a set of six pixels have to be loaded at the beginning to guarantee correct edge detection. With this in mind, the instruction CP_PRELOAD was created to perform just that, to load the first six pixels at the beginning of a new row.

Once the preload is complete, then the row is traversed to perform edge detection on each pixel in the row up to the second to last pixel. For each element in the row, grayscale conversion is performed on the six pixels, and the pixels are then loaded into the coprocessor through the CP_EDGE_DETECT instruction. The instruction performs the edge detection operation as shown in figure 6.

Another decision made was when the greyscale conversion of the pixels were performed, the greyscale pixels are stored in internal registers of the coprocessor as opposed to writing them out to a temporary frame-buffer. This decision was made so that cycles would not be wasted reading and writing to external memory, also, there was no need to wait for the greyscale conversion to finish before edge detection was performed on the pixels. Using the hardware to perform the greyscale conversion, the instruction had to be called twice. Once with the top four pixels, then again with the bottom two pixels with random junk that is not used. Once conversion is complete, the registers that the greyscaled pixels are at are set as inputs to the CP_EDGE_DETECT instruction. Edge detection is performed on two adjacent pixels, and the result is returned as a 32-bit result.

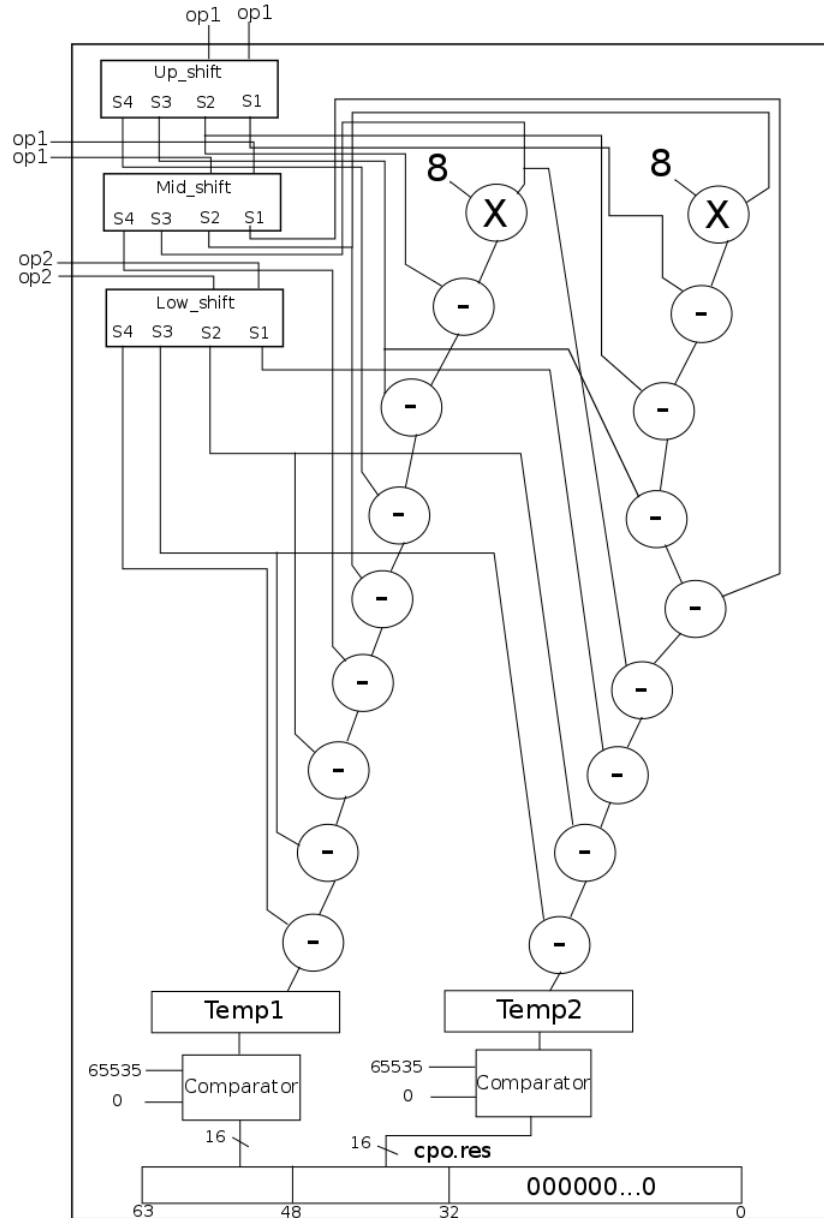


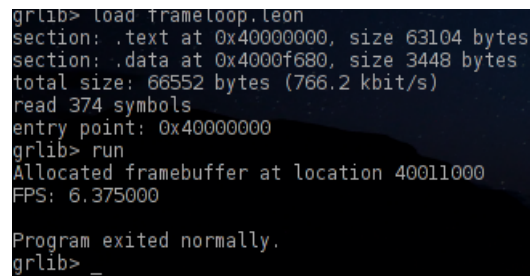
Figure 6: Hardware Laplace Edge Detection diagram

The decisions made that were outlined above have resulted in a speed up of 2.75 in the execution time of the operation, as shown in figure 7.

For the hardware edge detection implementation, we added two new opcodes - CP_PRELOAD and CP_EDGE_DETECT. In coproc.core, we essentially implemented 3 shift registers each being 64-bits long. Using CP_PRELOAD, we shift in the next pixels to be apply the Laplacian edge detection on. CP_EDGE_DETECT consists of two cycles. In the first cycle, it performs the internal shifting of the shift registers and in the next cycle, it performs the Laplacian edge detection computation. Since we have 64-bit (4 pixels) shift registers, we are able to compute the Laplacian value for two center pixels simultaneously.

We perform our grayscale conversion in line with the edge detection computation, unlike the software implementation in which we first converted all the images to grayscale and then performed the edge detection.

With the hardware implementation we were able to observe a 2.75 speedup when compared to the software implementation of the edge detection algorithm.



```
grlib> load frameloop.leon
section: .text at 0x40000000, size 63104 bytes
section: .data at 0x4000f680, size 3448 bytes
total size: 66552 bytes (766.2 kbit/s)
read 374 symbols
entry point: 0x40000000
grlib> run
Allocated framebuffer at location 40011000
FPS: 6.375000

Program exited normally.
grlib> _
```

Figure 7: Hardware Laplace Edge Detection Results

6 Conclusion

Through this Machine Problem assignment, we learned about a general hardware-software implementation and architecture.