Fall 2012 ECE498 Linux Assembly Language – Project 1

Due: 21 December 2012, 6PM

Background

In this project we will do some graphic manipulation in assembly language.

First we will load in a sample 320x320 graphics image. The original file was a .png image; this was converted to the much simpler .pcx format with the Linux `convert` utility, and then a provided executable converts this to an even simpler bitmap. The format for this file (`input.bin`) is two 32-bit values with the xsize and ysize, then xsize*ysize 32-bit values with color info. The color info is little-endian unsigned 32-bit integers with the format RRGGBBAA where the high 8-bits are the red value, the next are green, the next blue, and the last alpha (which we ignore in this project).

The code will do what is known as a convolution. In this case each pixel in the input buffer will be taken and averaged with the surrounding 8 pixels, according the values in a convolution matrix. This needs to be done separately for the R, G, and B components of the pixel. Then the averaged value is stored in the output buffer. This output buffer is then written to disk.

There are many possible convolution matrices. In this project we will use an "emboss" convolution. The matrix for it is described as $M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$. An example of the results are shown in Figure 1.

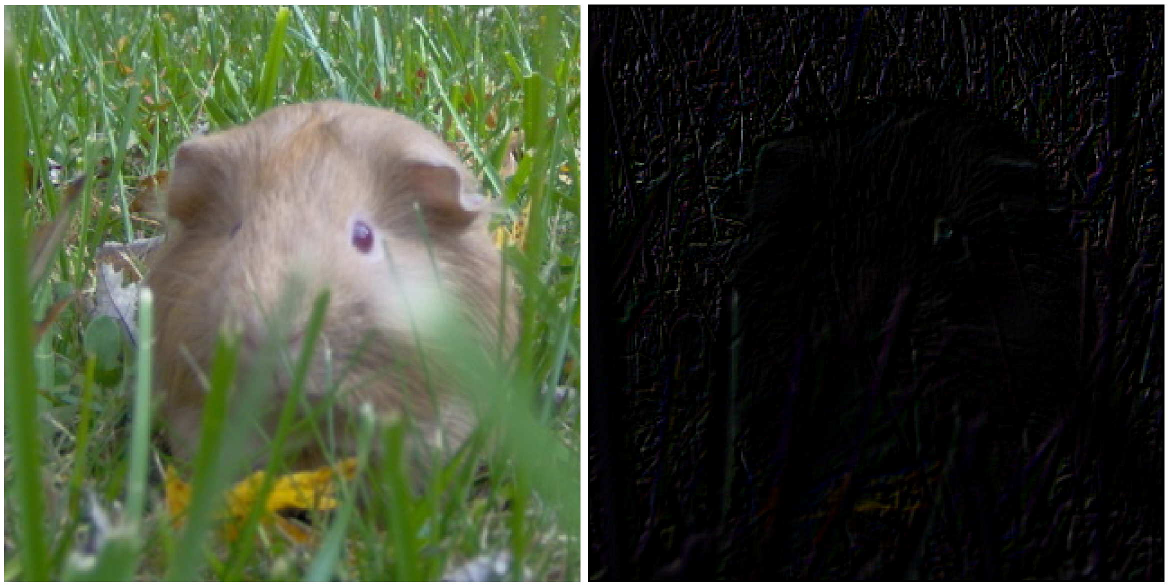

Figure 1: Image of a guinea pig before and after an emboss operation.

Setup

Download the Project 1 source code.

`http://www.eece.maine.edu/~vweaver/classes/ece498asm_2012f/project1_code.tar.gz`

Uncompress with `tar -xzvf`

Part 1 : x86 File I/O

The first part of this project is to read the image data in from a file and write it back out to a separate file, using 32-bit x86 assembly language.

1. First we will open our file, the equivalent of the C `fd=open("input.bin",O_RDONLY);`
   For convenience the test code has defined `O_RDONLY` for you; the definition of this file related defines can be found in `/usr/include/asm-generic/fcntl.h`
   On x86 the result of a syscall is passed in `%eax`. You will need to use the resulting file descriptor (fd) returned by `open()` in the following calls.

2. Next we will read in the xsize and ysize from the file. These were written as 32-bit little endian integers. You will need to do the `read()` syscall, expressed in C as `read(fd,&xsize,4);`.
   The second system call argument is a pointer to a buffer in memory.
   You can allocate such a buffer in either the `.data` or `.bss segments`, using `.int` to allocate a 32-bit value in data or `.lcomm var, size` to allocate space in bss. The provided code allocates space in bss that you can use.

3. Next read the ysize. Then read in the data.
   For the sake of this project the image will always be 320x320 32-bit integers. Again space for this is already provided in .bss in the provided test code.
   Well-written code would check the return values of the `read()` calls for errors; in this project for simplicity this is not required.

4. Now close the file descriptor with the `close(fd);` syscall.

5. Next we are going to write the buffer out to disk.

6. First open the file, this time with the following arguments: `fd=open("output.bin",O_WRONLY|O_CREA`
   The 0660 is an octal constant for file permissions, equal to group and user read/write access.

7. Write out the xsize and ysize you stored earlier.

8. Then write out the data.

9. Then close the file descriptor.

10. After you have done this, run your program `./x86_file_io`.
    If all went well you should have made a copy of your file. You can use the `diff` or `md5sum` programs to see if they are the same.

11. You can run `make output` which will run `./bin_to_pcx` to convert the result back to a PCX image `output.pcx` to verify the image is the same.

Part 2 : x86 Image Manipulation

In this part we will take the code from the previous section and modify it to do an "emboss" operation on the image before it is written back out.

Since the Matrix we use only needs two values (the current pixel, and the pixel to the lower right) we simplify the algorithm and just read and add these values. A proper algorithm would take any generic 3x3 matrix and apply the convolution, but getting code like that to work is assembly is much harder.

The 320x320 array is laid out linearly in memory, so the offset buffer[x][y] can be accessed as buffer[(y*xsize)+x].

You can implement the algorithm any way you want, but it might be easier if you use the following pseudo-code as a base:

Copy your finished `x86_file_io.s` file over `x86_convole.s`

Add the code that implements the following between the read in and write out routines. Make sure the write out routine writes out the proper buffer.

```
       put 321 in inptr   (start at buffer[1][1], not buffer[0][0], as the
       put 321 in outptr   convolution matrix would be out-of-bounds)

       put 318 in i
outer_loop:
       put 318 in j

inner_loop:
        color = inbuff[inptr]
        lowerright = inbuff[inptr + 321*4]

        inptr++

        blue_out=((color>>8)&0xff) - ((lowerright>>8)&0xff);
        green_out=((color>>16)&0xff) - ((lowerright>>16)&0xff);
        red_out=((color>>24)&0xff) - ((lowerright>>24)&0xff);


        /* saturate the results */

        if (blue_out<0) blue_out=0;
        if (green_out<0) green_out=0;
        if (red_out<0) red_out=0;

        if (blue_out>0xff) blue_out=0xff;
        if (green_out>0xff) green_out=0xff;
        if (red_out>0xff) red_out=0xff;



        outbuff[outptr]=(red_out<<24) | (green_out<<16) | (blue_out<<8);

        outptr++;

        j--;
        if (j>0) goto inner_loop;
```

```
inptr+=2;        Skip edges of image
outptr+=2;

i--;
if (i>0) goto outer_loop;
```

Part 3 : ARM File I/O

This part is the same as Part 1, but in ARM assembler rather than x86.

1. You can log into the pandaboard for this work, as you did for Homework 1.
2. Modify the provided `arm_file_io.s` to read in and write out the image data, the same as the x86 code did in Part 1.
3. On ARM the result of a system call is returned in `r0`.

Part 4 : ARM Image Manipulation

This part is the same as Part 2, but for ARM rather than x86.

1. Copy your working arm_file_io.s file from Part 3 over the arm_convolve.s file.
2. Modify the code to do the convolution as described in Part 3.

Part 5 : Analysis

Answer the following questions in the README file:

1. What do we store the image data in the .bss segment rather than .data?
2. If we added support for arbitrary sized graphic files, how would you allocate space for the buffers? What other changes would be needed in the code?
3. Which is bigger; your ARM code or the x86 code? Why?
4. Time how long it takes for your x86_convolve and arm_convolve programs to run. You can use do this by using the Linux `time` command (i.e. `time ./x86_convolve`. Report the results in the README.

Submitting

1. Be sure the final versions of your x86 and arm code are in your project1_code directory.
2. Answer all questions in Part5 and put the answers in the README file.
3. Run `make submit`
4. E-mail the generated `project1_submit.tar.gz` file to me.