# A Slightly More Advanced Assembly Exercise

## Exercise 1: Converting a Color Image (PPM) to Grayscale (PGM)

For this exercise, you will first write a C program to convert a color image (in PPM format) to grayscale (PGM format). Once you have thoroughly tested your C code to verify it is working correctly, you will convert it into a MIPS assembly program to do the same.

### PPM and PGM File Formats

For this exercise, we will use the ASCII file formats of type PPM (color) and PGM (grayscale). For a detailed description of these file formats, please see the[Wikipedia entry on Netpbm format.](http://en.wikipedia.org/wiki/Netpbm_format) A short description is given below.

The format of a PPM or PGM file is as follows:

* A PPM or PGM file begins with an identifier on the first line (called "magic number"). A magic number of "P3" means this file is an ASCII PPM file; "P2" means it is an ASCII PGM file.
* There may be an optional comment line following this, starting with "#". To make this assignment a little easier, the input files given to you will not have this optional comment line (or any other comment lines anywhere else in the input).
* Next, there are two numbers, the first indicating the number of columns, and the second the number of rows in the image.
* Next, there is a max value, which indicates the maximum value any pixel component (R, G, B, or gray value) may have. This value may be any positive integer, although values of 15 and 255 are quite typical.
* Finally, there are color or gray values for all the column \* row pixels. The values are ordered by rows from top to bottom, i.e., row 1 first, then row 2, etc., and within each row, the pixels are listed left to right.
  + For a PPM image, each pixel value is listed as the values of its three components, Red, Green and Blue.
  + For a PGM image, each pixel value is a single gray value.
* A whitespace (i.e., typically a space or newline character) must separate these integers. Other than that, the file is allowed to have any number of values on each line. For example, it is possible that the file has a single pixel value (R, G and B, separated by spaces) on a single line. Or, the file may have a single pixel color component per line (i.e., R on one line, G on the next, then B, then the next pixel, etc.). Or, one line in the file may have color values for multiple pixels, e.g., the RGB values for an entire row of pixels may be listed on a single line (as in the Wikipedia examples).
* When reading a PPM/PGM file, the above variations in format will not affect your C or assembly code. For instance, if you use **scanf("%d%d%d", &r, &g, &b)** in C, it will read the next three (decimal) integer values, regardless of any intervening space or newline characters.
* Likewise, your routine for printing the output for a PGM image can be formatted as you desire: one pixel gray value per line, or a whole row of values of per line, etc.

### PPM and PGM Viewers

There are a number of free/open-source viewers available for displaying PPM/PGM image files, including:

* Windows/Mac: GIMP (GNU Image Manipulation Program), available from [http://www.gimp.org](http://www.gimp.org/). You can also look into IrfanView, AyeView (fee trial), and ToyViewer (for Mac). I would recommend GIMP.
* Linux: I would recommend GIMP again, but many X Windows distributions already include a lighter-weight tool called XV.

### C Program

Write a C program that reads a PPM image (on its standard input, i.e., keyboard), and outputs its corresponding PGM image (on its standard output, i.e., console). Keep in mind the following:

* The input PPM image may be formatted in various ways, as mentioned above, i.e., one pixel per line, or one pixel row per line, or one pixel color component per line, etc. These variations in format will not affect your code. For instance, if you use **scanf("%d%d%d", &r, &g, &b)** in C, it will read the next three (decimal) integer values, regardless of any intervening space or newline characters.
* The max value for each color component can be specified to be any number 1 or greater. For this exercise, this number could be from 1 to 255 for the input PPM images.
* The max value for the gray values for the output *must be 255.* Thus, if the input PPM image has a max color component value of 15, then all of the values must be scaled appropriately. In particular, if the red, green and blue values of the pixel colors in the PPM image is R, G and B, respectively, and the max value is specified to be PPM\_MAX, then the corresponding gray value in the output PGM image must be computed exactly as follows:
* **Gray\_Value = ((R + G + B) \* 255) / (3 \* PPM\_MAX)**

Your C code (and assembly code below) must compute this expression exactly in the order indicated by the parentheses. The division is a truncated integer division (i.e., no rounding, throw away remainder).

* Reminder: The magic number of the output image should be "P2" since it is an ASCII PGM file.
* You can run your program and test it by typing (or cutting-and-pasting) sample inputs onto the console, and looking at the console output. For the larger examples, use input/output redirection:
* **ex1 < ex1in1.ppm > ex1result1.pgm**

Name the file containing your C program **ex1.c**, and test it on sample inputs provided. The sample output provided may differ from your program's output in terms of spaces and newlines. Be sure to check that they are otherwise identical, i.e., all the numbers contained in them are correct.

### Assembly Program

Once your C program is working correctly, write an equivalent MIPS assembly program that implements the same conversion from PPM to PGM images. *Do not begin work on the assembly program until you are sure that the C program is working correctly.* The best strategy would be to convert your C code (pretty much line-by-line) into its assembly equivalent. Run your program in MARS and verify that it is working correctly.

Name the file containing your assembly program **ex1.asm**, and test it on sample inputs provided. The sample output provided may differ from your program's output in terms of spaces and newlines. Be sure to check that they are otherwise identical, i.e., all the numbers contained in them are correct.

**Exercise 1**

For this exercise, you are to write an assembly program to convert red-green-blue (RGB) values for a set of pixels into a single *gray* value. First, study the C version of the program ([**ex1.c**](http://www.cs.unc.edu/~montek/teaching/Comp411-Spring17/Lab5/ex1.c)), and compile and run it.

//---------------------------------

// Lab 5: Pixel Conversion

//---------------------------------

#include<stdio.h>

int pixels[] = {0x00010000, 0x010101, 0x6, 0x3333, 0x030c, 0x700853, 0x294999, -1};

int main() {

int i=0;

int rgb, red, green, blue, gray;

printf("Converting pixels to grayscale:\n");

while (pixels[i] != -1) {

rgb = pixels[i];

blue = rgb & 0xff;

green = (rgb >> 8) & 0xff;

red = (rgb >> 16) & 0xff;

gray = (red + green + blue) / 3;

printf("%d\n", gray);

i++;

}

printf("Finished.\n");

}

For this programming assignment, you will read through this array of pixels, and for each pixel, convert the color pixel into a grayscale pixel using a simple formula: gray value = (red + green + blue) / 3. Note the division is integer divide and truncate (i.e., no rounding needed). For the above example, the gray value would be (1+255+34) / 3 or 96.

After calculating the gray value for a pixel, print it out to the console (only one element per line). The program keeps reading RGB values and printing the corresponding gray value, until it encounters an input of -1.

#---------------------------------

# Lab : Pixel Conversion

# Name: <YOUR-NAME-HERE>

# Onyen: <YOUR-ONYEN-HERE>

# --------------------------------

# Below is the expected output.

# Converting pixels to grayscale:

# 0

# 1

# 2

# 34

# 5

# 67

# 89

# Finished.

# -- program is finished running --

#---------------------------------

.data 0x0

startString: .asciiz "Converting pixels to grayscale:\n"

finishString: .asciiz "Finished."

newline: .asciiz "\n"

.align 2

pixels: .word 0x00010000, 0x010101, 0x6, 0x3333,

0x030c, 0x700853, 0x294999, -1

.text 0x3000

main:

ori $v0, $0, 4 #System call code 4 for printing a string

ori $a0, $0, 0x0 #address of startString is in $a0

syscall #print the string

addi $23,$0, 0

L1: sll $24,$23,2

lw $24,pixels($24)

la $15,-1

bne $24,$15,L2

b exit

L2: addi $30,$24, 0

la $24,255

and $20,$30,$24

sra $15,$30,8

and $21,$15,$24

sra $15,$30,16

and $22,$15,$24

addu $24,$22,$21

addu $24,$24,$20

la $15,3

div $19,$24,$15

printf: li $v0, 1 #System call code 4 for printing a strin

add $a0,$19, $0 #Displayed number is in $a0

syscall #System call

li $v0, 11 #System call code 11 for printing a character

addi $a0,$0, 0x0a #換行character is in $a0

syscall #System call

addi $23,$23, 1

b L1

exit:

ori $v0, $0, 4 #System call code 4 for printing a string

ori $a0, $0, 33 #address of finishString is in $a0; we computed this

# simply by counting the number of chars in startString,

# including the \n and the terminating \0

syscall #print the string

ori $v0, $0, 10 #System call code 10 for exit

syscall #exit the program