COMP 273

Assignment 3
School of Computer Science
McGill University
Available On: February 28th, 2020
Due Date: March 18th, 2020. 11:59pm.
Submit your solution in electronic form using MyCourses
Read the submission instructions at the end of the document
(late policy: 10 % off per day late, up to 2 days.
No submissions acceted after that.)

Things to consider

- Welcome to Assignment 3! We will be going through some fun image manipulation exercises in this one, developing some tools for your image processing toolbox. You may notice that FILE I/O will receive no partial marks. The reason for this is that every following question depends on FILE I/O functioning properly.

- Make sure to follow register and MIPS programming conventions! Beyond just readability and consistency between developers, the conventions also help developers avoid hours of debugging. If you have questions about conventions/debugging, feel free to ask any of the TAs.

- Even if your code does not produce the right image, make it produce something!

- MARS uses relative paths to find files. This means, if you have Mars in the same folder as your .txt and scripts, then the paths specified in the templates will work accordingly.
- Make sure to switch on "compile all files in directory" in the "settings" menu when assembling *main.asm* in MARS.
- GIMP is an image editing software package. Though you can use anything you'd like, GIMP is free software that can be used to view the .pgm file after generating the image is complete. You can download it from
https://www.gimp.org/
- In this assignment we will be working with '.pgm' files of type P5 and P2. Headers of .pgm files have a specific format. http://netpbm.sourceforge.net/doc/pgm.html contains information about proper '.pgm' formatting. In this assignment we will only use pgm files with a maximum value less than 256. So each value can be stored in one byte. We will also assume that there are no comment lines in the file, i.e., a line beginning with a '#' character.
- You may add any extra data or labels in .data segments of the files, but make sure you don't remove any existing data or labels. The data specified in the templates should be used accordingly.

1 FILE I/O (15 marks)

The template for this question is provided in *fileio.asm*. This question is to help you get started with reading and writing files.

(a) Using the template *fileIO.asm* as a starting point, write a MIPS procedure *read_file* that takes as its argument the address of a string that contains a valid filename, and then uses appropriate syscalls to read from that input file and then simply prints the content of that file to the screen. In the template there are two input files called test1.txt and test2.txt which you should test. To accomplish this task we allow you to create a large buffer, i.e., one that is larger than the expected number of ASCII characters (bytes) in the input file. The body of your code should work by calling the procedure you have written. The procedure should open the file, read its content as ASCII characters, store the content in the buffer, print the content to the screen, and then close the file. For your reference a more complete set of MIPS syscalls implemented in MARS, along with clear documentation on how to use each, is here: http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html. (10 marks)

(b) We shall now build on the above example. Following the process of reading the file *test1.txt*, after which the ASCII characters have been read into the buffer, we shall call a second MIPS procedure called *write_file*. This procedure should open a file called *copy.pgm*, and should then write the following information to that file:
P2
24 7
15
Then it should write out the content that was read into the buffer. It should then close the file.

Error statements should be printed if there are any errors in opening the file. If things are working properly when you view *copy.pgm* with a suitable image viewer, e.g., GIMP, you should see something interesting. (5 marks)

2 Reading and writing an Image (20 marks)

From this question onward you should switch on "assemble all files in directory" in the "settings" menu on MARS. You can then assemble *main.asm* and run to to test your subroutines. Note that template *main.asm* does not check validity of your solution. You may edit *main.asm* as you wish, adding more tests/checks to test your solutions. When grading the TAs will use a separate *main.asm* file with extensive tests. The Template for this question is *readImage.asm, writeImage.asm*
In this question we will implement functions to read and write image files.

(a) Starting with the template provided in *readImage.asm* implement a *read_image* procedure which takes in a '.pgm' image filename as input and reads the contents into a struct. An image will be read into a data structure containing information about the image and array of bytes containing the contents of the image. Specifically, an image should be represented as the following struct.

```
struct image{
    int width;                          // number of columns
    int height;                         // number of rows
    int max_value;                      // maximum value in the image
    char contents [width*height]; // image content as an array
}
```

Here contents is a `char` array of size $width \times height$ which stores values of the image as a 1D array. In addition, the width height and max_values have to be stored as 1 word(4 bytes) numbers each, at the beginning of the struct. Your subroutine should return the address of the image struct. The subroutine should support '.pgm' images of type 'P2' and 'P5'. Note that while contents is a 1D array, it represent a 2D image. Note that 2D arrays (or ND arrays in general) are an abstraction provided by a high language like C or Java while in reality

they are represented as 1D arrays in the memory. As you know, in a language like Java or C, we would simply specify array positions as *array*[i][j]. That is, we would let i represent the row we are currently at, and j represent the column we are currently at. In MIPS, however, our 2D array is stored as values in a 1D array. It is clear to see that for any position [i,j] in our 2D array, we can retrieve this position by simple computing (i * width) + j. Since i represents rows, whenever we add a width we are essentially going to the next row in our conceptual 2D array. j simply represents which column we are currently looking at. In the context of an image, the value of image array at a location say [i,j] is referred to as the *pixel value* of the image at [i,j] and location [i,j] is called the *pixel location*.

(b) In the second part, implement a subroutine *write_image* in file *writeImage.asm* which takes the image struct, a filename (of the form 'abcd.pgm') and a type (which can be 0 or 1) as inputs and writes it to a '.pgm' file. If the type input is 0 the function should write a '.pgm' file of type P5 otherwise it should write an image of type P2 if type is 1. You can verify that your subroutine is working correctly by opening the written '.pgm' file in GIMP. We suggest you start with implementing write for 'P5' as it is relatively easier. As a debugging tool you may also want to implement a helper function to print the contents of the image to the console. This will also be useful for the rest of this assignment, whether or not your *write_image* code is correct.

3 Image Accessor (20 marks)

Template: *getPixel.asm, setPixel.asm*

In this section we will implement subroutines to access a pixel value at any location in an image and change values of an image at a given location.

(a) Starting from 'getPixel.asm' as a template, write a subroutine '*get_pixel*' which takes an image struct, int row and int column as input and returns the pixel value of the image at location (row, column). If the pixel location (row,column) is outside the image your subroutine should return 0 and print out an error message to the console.

(b) Again using 'setPixel.asm' as a template, write a subroutine '*set_pixel*' which takes 4 inputs, an image struct, int row number, an int column number and an int value and writes the value to the image at location (row,column). If the input value is larger than 255 the value should be set to 255. If the pixel location (row, column) is outside the image region, your subroutine should not change any value. Instead just print an error message to the console.

4 Contrast Inversion (20 marks)

Template: *invertImage.asm*

(a) In this and the following questions we will implement some basic processing operations on an image. Starting with this question, using 'imageInvert.asm' as a starting point, write a subroutine '*invert_image*' which takes an image as an input and inverts the contrast of the image. Here by inverting the contrast we mean replacing the value at each pixel with '*max_value - old_value*'. The subroutine should return the inverted image. Note that after doing this operation both the '*max_value*' field of the image struct and the contents array of the resulting image will change. However, the width and height will remain the same. To test your result, you can write the image to a 'pgm' file using '*write_image*' subroutine implemented in earlier question and view it in GIMP.

5 Linear Contrast Stretching (25 marks)

Template : *rescale.asm*

(a) Write a subroutine '*rescale_image*' which rescales the values of an image to range from 0 to 255. Say you are given an image of $width = 24$ and $height = 7$ that has values ranging from 10 to 43. The rescaled image should be an image whose values range from 0 to 255, rescaling each value as follows. Say the pixel value at some pixel location (row,

column) is x, then the new rescaled value at the location (row, column) will be $new\_value = \frac{(x-min\_value)*255}{max\_value-min\_value}$. Here, '$min\_value$' and '$max\_value$' are the minimum and maximum values in the old image respectively. In our example the minimum and maximum will be 10 and 43. Note that rescalling can lead to non-integer values. So, you should use co-processor 1 (CP0) to perform the calculation using single precision floating point values. You should then round the result to the nearest integer value and write this to an appropriate location in the image. For example, if the pixel value at a location is 41, the new rescaled value will be $round\left(\frac{(41-10)*255}{(43-10)}\right) = round(239.54) = 240$. The subroutine should return a rescaled image struct, as in earlier questions. If the input image contains only one value, i.e $max\_value - min\_value = 0$, the rescaled image returned should contained the same value as the original, i.e./ pixels values should not be changed in this case. As an aside, you can play with the transformation function to get different contrast effects. If you are keen, an interesting one to do would be to change the transformation function to $new\_value = (old\_value)^{\gamma}$. See gamma correction and fast inverse square root.

Assignment Submission Instructions

By handing in this your assignment you acknowledge that the work you are submitting is your own, and that you have read the COMP 273 FAQs document under "Content" in *mycourses*.

1. In addition to subroutine templates provided in *fileIO.asm, readImage.asm, writeImage.asm, imageInvert.asm, getPixel.asm, setPixel.asm* and *transform.asm*, you may additionally add as many helper subroutines as you want. You may also include additional '.asm' files containing helper functions. However, do not include any images with your submission. Your submission will be tested on a separate wider set of input images. Testing will be done automatically, so make sure you don't change any labels and that you follow register conventions. If you fail to follow the conventions or change labels in the templates provided your may loose marks.
2. Submit your solution to *myCourses* before the due date. Hints, suggestions and clarifications may be posted on the discussion board on *mycourses* as questions arise. Even if you don't have any questions, it is a good idea to check the discussion board.
3. Take all your '.asm' files and zip them. The zipped file should be named <studentID>.zip. If my student ID is 123456789, then the zip file to submit will be named *123456789.zip*. **Do not upload individual files or any other compression format**. Additionally, make sure to add your student ID as a comment on top of all the files you submit. It is your responsibility to ensure that you upload the correct version of all the files before the assignment deadline.
4. Copying code from other students or from internet sources is strictly prohibited. The assignments will be checked for plagiarism using automatic plagiarism detection tools. When dealing with suspected plagiarism cases, we have no way of telling if you copied the code from someone or let others copy from you. We will simply report the matter to the disciplinary office of your faculty. Therefore it is far better to submit your own work, even if it is not complete. If you are tempted to copy, here is a guide on how to do this without getting caught. The upshot is that this might require more effort than doing the assignment on your own.
5. Comment your code in detail. If there are no comments, and the code is incorrect, you will get ZERO marks. If you make any special assumptions in your programs, or if you feel there are ideas that need explanation, describe them in your comments.
6. Your code *must* run and assemble, even if your implementation is not completely correct. If something is not working, comment-out the broken parts of code and write a comment or two about what you expect to happen, what is happening, and what the problem may be. You are expected to follow register and MIPs programming. *If your code does not assemble you will receive 0 points for that question.*