CS 240 – Computer Organization

Lab 5 – Image Processing Kernels on MIPS – 200 points

In this lab assignment, you will implement three image processing kernels in MIPS assembly. The input for all three subroutines is a square sized grayscale image. Refer to the last page of this document on memory layout to learn how a 2D array is stored in memory. In this case, images are stored in a 2D array where each element is a byte that represents each pixel's intensity (0x00 used for black to 0xFF for white. Output image for all parts will have the same size as the input image. We have included a utility function that load and store image data from images stored on disk with pgm file format (a simple raw format). Your task is to write the body of the functions that loads an array of pixels along with its dimensions and process them and stores the result back in memory.

Fig. 1. Image processing pipeline

Part 1) Image Thresholding (40 points + 5 points for good commenting):

Image thresholding is a simple way of partitioning an image into a foreground and background. This image analysis technique is a type of image segmentation that isolates objects by converting grayscale images into binary images. An example of such conversion is shown below:

Lena.png

Lena_binary.png

Your function should replace the value of each pixel in the grayscale image with the maximum possible value (0xFF, i.e. brightest value), if the intensity (value) of that pixel is greater than or equal to some constant T (function input), or otherwise with the minimum value (0x00, i.e. darkest value). Remember that these values are unsigned.

Part 2) Affine Transformation (85 points + 5 points for good commenting):

For the second part, you are implementing a function to apply an affine transformation to the input image. Such transformation is described using a 2x3 transform matrix (M) that map pixels at input image to different coordinates in output image. Affine transformations have this property that they map parallel lines in input image to parallel lines in output image. Examples of affine transformations include translation, rotation, reflection, and scale. You can see this youtube video for more details or this link as an essence of linear algebra.

The following algorithm demonstrates applying affine transformation to an input image. In this algorithm x and y represents current pixel coordinates -- x being the column index and y being the row index. Top-left pixel has coordinates of (0, 0).

$$for\ each\ input\ image\ pixel\ at\ x, y\ coordinate:$$
$$x_0 = M_{oo} \times x + M_{o1} \times y + M_{02}$$
$$y_0 = M_{1o} \times x + M_{11} \times y + M_{12}$$
$$output(x, y) = input(x0, y0)\ if\ cols > x_o \geq 0\ and\ rows > y_0 \geq 0\ otherwise\ 0$$

Although image pixels are bytes, matrix elements are integers. Here are some examples of affine transformations: (THESE ARE JUST EXAMPLES, YOUR IMAGES MAY LOOK A LITTLE DIFFERENT)

Sample input

1	0	0
0	1	0
0	0	1

Identity matrix

2	0	0
0	1	0
0	0	1

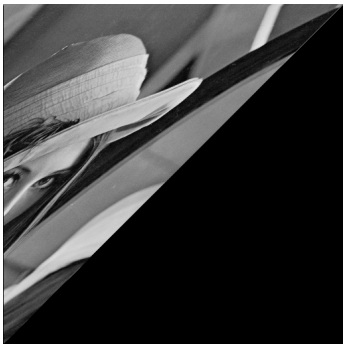
Scale matrix (x = 2)

0	1	0
1	0	0
0	0	1

Rotation matrix (+ translate)

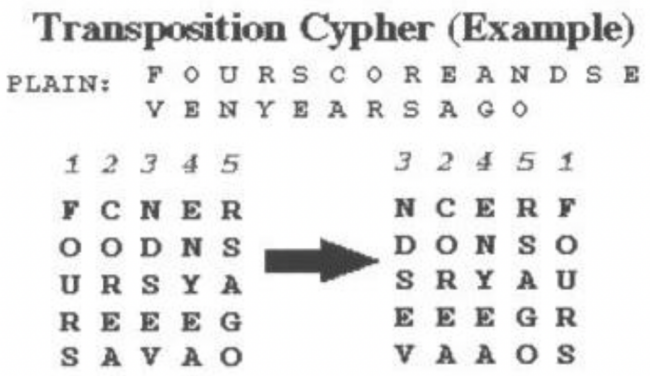
1	1	0
0	1	0
0	0	1

Shear matrix



Part 3) Image Cryptography (60 points + 5 points for good commenting):

In cryptography, a **transposition cipher** is a method of encryption which scrambles the positions of characters (*transposition*) without changing the characters themselves. In this lab, you will scramble the pixel values in an image using a **secret key** to hide the contents of the image.



For this lab – you the secret key - **4 3 1 5 2** to cipher the pixel contents.

You may assume that the image given to you would be square and a power a 5. This means that all the pixel values in a single row of an image would be jumbled within the same row.

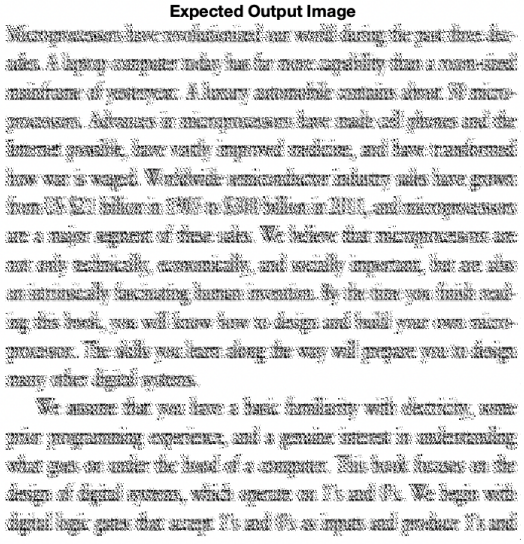
Expected output for the given input image is shown below:

Input Image - textfile.pgm

Microprocessors have revolutionized our world during the past three decades. A laptop computer today has far more capability than a room-sized mainframe of yesteryear. A luxury automobile contains about 50 microprocessors. Advances in microprocessors have made cell phones and the Internet possible, have vastly improved medicine, and have transformed how war is waged. Worldwide semiconductor industry sales have grown from US $21 billion in 1985 to $300 billion in 2011, and microprocessors are a major segment of these sales. We believe that microprocessors are not only technically, economically, and socially important, but are also an intrinsically fascinating human invention. By the time you finish reading this book, you will know how to design and build your own microprocessor. The skills you learn along the way will prepare you to design many other digital systems.

We assume that you have a basic familiarity with electricity, some prior programming experience, and a genuine interest in understanding what goes on under the hood of a computer. This book focuses on the design of digital systems, which operate on 1's and 0's. We begin with digital logic gates that accept 1's and 0's as inputs and produce 1's and

Expected Output Image



Your function should replace the value of each pixel in the grayscale image with the neighboring pixel value based on the secret key. Remember that these values are unsigned.

Pixel [0,0] = pixel [0,4] Pixel [0,1] = pixel [0,3] ….. Pixel [0,n] = pixel [0,n-3]
Pixel [1,0] = pixel [1,4] Pixel [1,1] = pixel [1,3] ….. Pixel [1,n] = pixel [0,n-3]
…
…
Pixel [n,0] = pixel [n,4] Pixel [n,1] = pixel [n,3] ….. Pixel [n,n] = pixel [0,n-3

Testing

Lab5.s comes with a tester which tests your function with proper parameters such as input arrays and images. If your program passes tests on arrays correctly, you are guaranteed to receive full points for this lab.

To test your code against images, you need to follow the same instructions as for lab3 and save mars, your code and the images in the same folder. “Lenna.pgm” is the input image and “lenna_thresh.pgm”, “lenna_scaled.pgm”, “lenna_shear.pgm” and “lenna_scale.pgm” are output images that will be generated by parts 1 and 2 of this lab. You need to verify them manually. (You can open your pgm file in this website for verification: http://paulcuth.me.uk/netpbm-viewer/), you can check the sample output files we are giving you before you run the code to see how they should look like

```
fin: .asciiz "lenna.pgm"
fout_thresh: .asciiz "lenna_thresh.pgm"
fout_rotate: .asciiz "lenna_rotation.pgm"
fout_shear: .asciiz "lenna_shear.pgm"
fout_scale: .asciiz "lenna_scale.pgm"
```

For Part 3 – the input image is “textfile.pgm” and the output image is “text_crypt.pgm”.

Submissions:

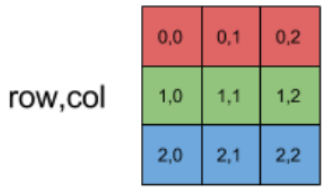
- Complete the provided lab5.s code and submit it to Gradescope
- Do NOT submit your project output files

Memory Layout of a 2D Array:

An image is essentially a 2 dimensional (2D) array where every element of that array represents a pixel. The value of that element determines the corresponding pixel appearance such as color or brightness (for grayscale images).

In order to process a 2D array in assembly, it's important to understand how a 2D array is laid out in memory. First, some notes on the nomenclature of this handout. Computer memory will be represented as a linear array with low addresses on the left and high addresses on the right. Also, we're going to use programmer notation for matrices: rows and columns start with zero, at the top-left corner of the matrix. Row indices go over rows from top to bottom; column indices go over columns from left to right.

The elements sit in the memory in a manner called row-major layout where the first row of the matrix is placed in contiguous memory, then the second, and so on:



Another way to describe row-major layout is that column indices change the fastest. This should be obvious by looking at the linear layout at the bottom of the diagram. If you read the element index pairs from left to right, you'll notice that the column index changes all the time, and the row index only changes once per row.
For programmers, another important observation is that given a row index (row_idx) and a column index (col_idx), the offset of the element they denote in the linear representation is:

$$\text{offset} = \text{row\_idx} * \text{num\_of\_columns} + \text{col\_idx}$$

Where num_of_columns is the number of columns per row in the matrix. It's easy to see this equation fits the linear layout in the diagram shown above.

Having the memory address of the base of an array and a pair of (row, col) indices, we are able to find the address of individual elements in a 2D array:

$$\text{element\_addr} = \text{base\_addr} + \text{element\_size\_in\_bytes} * \text{offset}$$