MIPSfpga by Imagination

Lab 4

More Programming Practice (Optional):

Pocket Hypnotizer, Memory Game, and Image Transformation

These materials produced in association with Imagination.
Join our University community for more resources.
community.imgtec.com/university

 © Imagination Technologies 2017

MIPSfpga Lab 4:

Pocket Hypnotizer, Memory Game, and Image Transformation

1. Introduction

This is an optional lab that gives you more practice writing C and Assembly programs for the MIPSfpga system. You will write three programs. First, you will write a program that scrolls the lights on the LEDs to create a pocket hypnotizer. Then (assuming the hypnotizer didn't work *too* well), you will write a memory game. Finally, you will write a program that performs several transformations on a given image.

2. Pocket Hypnotizer

You have been dispatched to the Atacama Desert to obtain secret information from a rebel leader. You'll have to get past the border guards to reach your destination. For this mission, you need to build a pocket hypnotizer.

Write a C program that causes a pattern on the LEDs to zip back and forth. When your program starts, it should turn on the far-right LED. Then it should turn off that LED and turn on the next one to the left. Continue until reaching the end of the LEDs, then go back to the right, and repeat indefinitely. You'll need to choose a suitable delay between steps to get the desired effect.

If you have difficulties, use Codescape's gdb debugger to step through your code and compare against your expectations. Tune the delay until it looks mesmerizingly good. Stare into the blinking lights as you repeat to yourself "I love MIPSfpga."

3. Memory Game

Now you will implement a Memory Game similar in spirit to Simon (see http://en.wikipedia.org/wiki/Simon_(game)).

The game should be played as follows:

1. First, the user presses any key (pushbutton) to begin playing the game.

2. Next, your program should create a series of 8 random LED flashes on the three right-most LEDs of the Nexys4 DDR board.

3. Then, the user enters keypresses on the Nexys4 DDR board using the left, center, and right pushbuttons (BTNL, BTNC, and BTNR). This should be an attempt to replicate the pattern of LED flashes. BTNR, corresponds to the right-most LED (LED[0]), BTNC to the center LED (LED[1]), and BTNL to the left-most LED (LED[2]).

   As the user presses a key, light up a new LED to indicate that the keypress has been detected. For example, after one keypress, the right-most LED lights up. After the second keypress, the right-most two LEDs light up, and so on until 8 LEDs light up. You will also have to deal with switch bounce, where a single keypress gives multiple transitions on the pushbutton input.

4. Then, your program should compute the user's *score* and display it on the LEDs. The program computes the score by comparing the record of the LED flashes with the keypresses the user entered. For each keypress that is correct, the user gets 1 point. The *score* is the number of correct keypresses. For example, suppose you generate the following pattern of 8 LED flashes:

   L, C, C, C, R, C, R, L (where L, C, and R stand for Left, Center, and Right).

   The user enters: L, C, C, R, R, C, R, R.

   The user got 6 out of 8 correct, so the 6 right-most LEDs should light up.

5. Finally, the user presses any key (pushbutton) to play again.

If you have difficulties, use Codescape's gdb debugger to step through your code and compare against your expectations.

4. Image transformation

4.1 Introduction

The aim of this exercise is to perform the transformation of an image with three color channels (RGB) to a grayscale image, and then transform it into a binary image. Figure 1 shows the result of this process on an example image.

Figure 1. Example image in color, grayscale and binary

Folder *Part1_Intro\Lab04_MoreProgramming\ImageTransformation* provides the source code for this exercise, with some empty functions that you have to complete as explained below.

An image is usually represented as a matrix, where each element of the matrix stores the value of a pixel within the image. In RGB, each pixel is defined by three values: level of red (R), level of green (G) and level of blue (B). Therefore, each element of the color image is represented with a vector of 3 elements. Specifically, for this lab, we use the following definition for a pixel in the RGB image (see file *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\trafo.c*):

```
typedef struct _pixel_RGB_t{
      unsigned char R;
      unsigned char G;
      unsigned char B;
} pixelRGB;
```

Note that, in this definition, each color channel uses 8 bits, thus each channel can represent 256 different values, and we have a total of 24 bits per pixel (24bpp).

In the case of a grayscale image, a single value is used to represent the brightness of each pixel, and in the case of a binary image, a single bit is necessary. In this lab, we use 8 bits to represent each pixel on both a grayscale and a binary image.

The following code defines three 128x128 images: an RGB image, a grayscale image and a binary image (see file *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\main.c*).

```
#define NROWS 128
#define NCOLS 128

pixelRGB ColorImage[NROWS*NCOLS];

unsigned char GrayImage[NROWS*NCOLS];

unsigned char BinaryImage[NROWS*NCOLS];
```

As an example, in order to access the pixel located at row 13 and column 24, you must index the images as follows:

RGB Image:

```
ColorImage[13*NCOLS + 24].R
ColorImage[13*NCOLS + 24].G
ColorImage[13*NCOLS + 24].B
```

Grayscale Image:

```
GrayImage[13*NCOLS + 24]
```

4.2 Transformation of an image from RGB to grayscale

In order to transform the RGB image into a grayscale image (function **RGB2GrayMatrix** in file *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\trafo.c*) we perform a simple

linear function, as shown in Figure 2. Thus, in your C program, you have to include the following command:

```
GrayImage[x*NC+y] = 0.2126*RGBImage[x*NC+y].R +
0.7152*RGBImage[x*NC+y].G + 0.0722*RGBImage[x*NC+y].B;
```

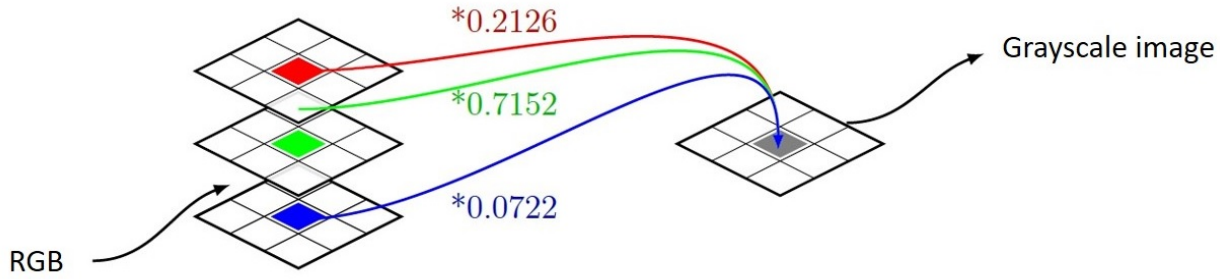

Figure 2. Example image in color, grayscale and binary.

Note that, in this equation, we need to perform floating point operations. However, the processor configuration that we are using in MIPSfpga does not include a floating point unit. Thus, you will transform the previous equation into integer arithmetic (see function **rgb2gray** in file *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\trafo.c*):

```
GrayImage[x*NC+y] = (3483*RGBImage[x*NC+y].R +
11718*RGBImage[x*NC+y].G + 1183*RGBImage[x*NC+y].B) / 16384;
```

4.3 Transformation of an image from grayscale to binary

The algorithms used for transforming an image from grayscale into binary format use to compare each pixel with a threshold: if the value is higher than the threshold, the pixel in the binary image is set to white (1), otherwise, it is set to black (0). Function **Gray2BinaryMatrix** in file *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\trafo.c* implements this functionality.

For selecting the threshold value you could use different techniques. In this lab, you will first create a function (called **computeHistogram** and defined in file *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\trafo.c*) which builds a histogram that accounts for the number of occurrences of each of the 256 possible values of light intensity (you can use the following array: `short int histogram[256];`). Then, function **computeThreshold** looks for the index of the maximum value in the first 128 elements (let´s call it *i*) and the index of the maximum value in the 128 second elements (let´s call it *j*), and calculates the threshold as follows:

```
Threshold = i + (j - i)/2
```

4.4 Exercise

The aim of this lab is to develop the transformations explained in the previous section in C and assembly languages, execute them on the board, and check the resulting images. We next describe the files that compose the project available in folder *Part1_Intro\Lab04_MoreProgramming\ImageTransformation*:

- **main.c**: In this file we define the images and invoke the functions that implement the image transformations. If *LENA* is defined, the RGB image is initialized with a 128*128 image widely used since the early works on image editing. If the image is not defined, a synthetic image is used. After the images are defined and initialized, the main function invokes the different functions for performing the RGB to grayscale and grayscale to binary transformations.
- **lena128color.h**: This file contains 128*128*3 values representing the R, G and B channels of each of the 128*128 pixels that make up Lenna image.
- **trafo.c**: This file contains the functions for performing the transformations, most of which you have to implement. Specifically:
    - *RGB2GrayMatrix*: This function performs RGB to grayscale image transformation as explained above and is already implemented. It invokes function *rgb2gray*, also described in the previous section, which you have to complete.
    - *computeHistogram*: As explained above, this function must compute the histogram. You also have to complete this function.
    - *computeThreshold*: In this function the threshold is computed as described in the previous section. You also have to complete this function.
    - *Gray2BinaryMatrix*: Using the threshold computed in the previous function, the grayscale image is transformed into a binary image. You also have to complete this function.

Once you have implemented all these functions in C language, you can compile, run and visualize the results by following the next steps:

- Download MIPSfpga on the board as explained in Step 1 – Lab 2.
- Compile the project as explained in Step 2 – Lab 2.
- Download the project onto the MIPSfpga system as explained in Step 3 – Lab 2.
- Debug the project as explained in Step 4 – Lab 2. Specifically, run the following commands in gdb:
    - `monitor reset halt`

    - `b *0x800002dc` (set a breakpoint after returning from the last function invoked at main.c, which is **Gray2BinaryMatrix**; note that it can be placed in a different address to the one used here)
    - `c`

  Execution should start and stop after returning from function **Gray2BinaryMatrix**. At this point, run in gdb the following commands, which will create two files containing the pixels of the grayscale and binary images:

    - `dump value gray.dat GrayImage`
    - `dump value binary.dat BinaryImage`

- These files are not in any common image format, but can be easily converted. For that purpose, you can use the file provided in *Part1_Intro\Lab04_MoreProgramming\ImageTransformation\dump2ppm.c*. Compile that file and execute it as follows:

    - `dump2ppm gray.dat gray.ppm 128 128 1`
    - `dump2ppm binary.dat binary.ppm 128 128 1`

  You should now be able to visualize the *.ppm* images (if you don´t have an application that can read this format, such as *gimp*, transform the *.ppm* image into a format that you can visualize).

Finally, as another interesting exercise, you can implement some of the functions performing the image transformations in MIPS assembly language. As an example, you can start by programming function **rgb2gray** in MIPS assembly language. For that purpose:

- File **assembly.c** defines subroutine *rgb2grayAssembly*, which you have to complete in MIPS assembly language. In addition, complete subroutine *div16K*, which should be invoked from *rgb2grayAssembly* for computing the division by 16384. Make sure you meet the MIPS convention for managing function calls, explained in detail in Section 6.4.6 of [1]. Finally, in function *RGB2GrayMatrix*, change invocation of function *rgb2gray* for function *rgb2grayAssembly*.

5 References

[1] “Digital Design and Computer Architecture”, 2nd Edition. David Money Harris and Sarah L. Harris. Morgan Kaufmann, 2012.