THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 4
Image Processing: C & Assembly

1. Introduction

In this lab, you will build RISC-V programming projects that perform image processing routines. The projects will include multiple source files, some of which are written in C and some in assembly. We will show how C functions can invoke assembly routines and vice versa.

2. Image Processing Tutorial

Begin this lab by examining a provided program that processes an RGB image (left side of Figure 1) and generates a greyscale version of that image (right side of Figure 1). The program is written in C and RISC-V assembly languages and is configured to run on the Catapult environment. It is available at:

[RVfpgaBasysPath]/Labs/Lab04/ImageProcessing

The source code is in the *src* subdirectory.

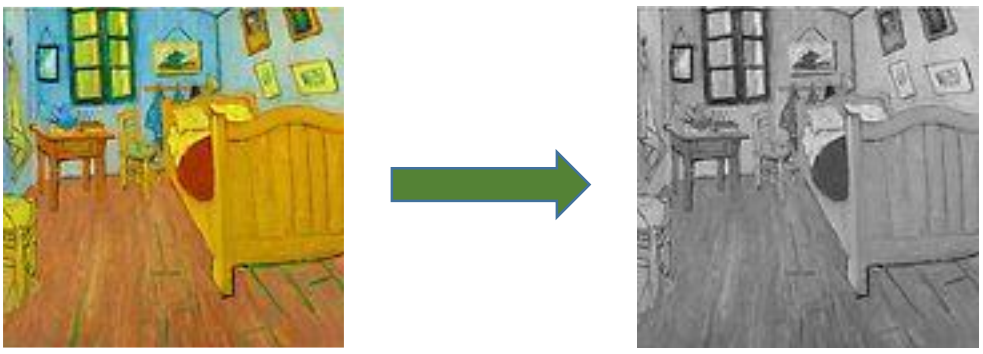

Figure 1. Transformation of an RGB image to a greyscale image

A. Project structure and *main* function

The program consists of the following source files: **Test.c**, **VanGogh_128.c** and **assemblySubroutines.S**. The *.c* files contain functions (such as the functions for performing the image transformations) and variable declarations (such as the input image, declared as an unsigned char array). The **assemblySubroutines.S** file contains an assembly language implementation of the function that transforms the image from RGB to grey scale called: *ColourToGrey_Pixel*.

Figure 2 shows the `main` function of this project. It first invokes function *initColourImage*, which creates an N x M matrix with the input image data. It then transforms the colour image to a greyscale image (function *ColourToGrey*). Finally, the function prints a message and enters an infinite loop (`while (1);`).

```
49  int main(void) {
50      // Create an NxM matrix using the input image
51      initColourImage(ColourImage);
52
53      // Transform Colour Image to Grey Image
54      ColourToGrey(ColourImage,GreyImage);
55
56      // Initialize Uart
57      uartInit();
58      // Print message on the serial output
59      printfNexys("Created Grey Image");
60
61      while(1);
62
63      return 0;
64  }
```

Figure 2. main function in ImageProcessing project

B. RGB and greyscale images

An image consists of a matrix of pixels, where each element of the matrix represents the value of a pixel in some given scale. In RGB, each pixel is composed of three values, which correspond to the luminous intensity of the red (**R**), green (**G**), and blue (**B**) components. Therefore, each pixel of a colour image will be a three-component vector. In this project, we use the following definition for the RGB pixel type:

```
typedef struct {
    unsigned char R;
    unsigned char G;
    unsigned char B;
} RGB;
```

This code defines a structure named *RGB*. In C, a `struct` data type is a collection of variables, possibly of different types, specified by a single name. This structure contains three fields, all of the same type (`unsigned char`), named *R*, *G*, and *B*. Thus, each colour channel (red, green, or blue) is represented by 8 bits, so that we can distinguish among 256 different intensity levels in each colour channel, for a total of 24 bits per pixel (24bpp). This is a common format in current digital image processing.

To represent a greyscale image, a single value (single channel) ranging from 0 to 255 indicates the brightness of each pixel. In this ImageProcessing project, we represent the greyscale image using a 2-dimensional array of characters:

```
unsigned char GreyImage[N][M];
```

C. Transforming a colour image into a greyscale image

The transformation between the two colour spaces (RGB and Greyscale) is performed using the following weighted sum:

```
grey = 0.299*R + 0.587*G + 0.114*B
```

This equation is based on the algorithms described at https://www.mathworks.com/help/matlab/ref/rgb2gray.html.

For each pixel, we calculate the greyscale value by multiplying each colour channel by the weight given in the equation. The sum of the weights (0.299+0.587+0.114) is one, so the resulting greyscale value will be within the range 0-255, and thus it can be represented with a single byte.

To use the weights given in the equation, we would need to operate with real numbers, however the **VeeR EL2** processor does not include floating-point support. One approach would be to use floating-point emulation, as in the DotProduct program shown in Section 5.H of the Getting Started Guide, however, in this lab, we use an approach based on integer arithmetic. The weights are converted to integers and the sum is a power of two (in our case, 2^{10}). To convert the weights to integers, we multiply each floating-point weight by 2^{10} and round to the nearest integer:

- $0.299*2^{10} = 306.176 \approx$ **306** **(weight for R)**
- $0.587*2^{10} = 601.088 \approx$ **601** **(weight for G)**
- $0.114*2^{10} = 116.736 \approx$ **117** **(weight for B)**

Of course, to reduce the final greyscale value to the range 0-255 we must divide the sum by 2^{10}, which is easily completed by shifting the value right by 10 bits. Thus, the final transformation is obtained using the following formula:

```
grey = (306*R + 601*G + 117*B) >> 10
```

Note that, given that the sum of the constants (306+601+117) is 1024, the resulting greyscale value will still be within the range 0-255.

Figure 3 illustrates the code for the `ColourToGrey` function (left side) and the `ColourToGrey_Pixel` subroutine (right side) that `ColourToGrey` invokes.

```
38  extern int ColourToGrey_Pixel(int R, int G, int B);
39
40  void ColourToGrey(RGB Colour[N][M], unsigned char Grey[N][M]) {
41      int i,j;
42
43      for (i=0;i<N;i++)
44          for (j=0; j<M; j++)
45              Grey[i][j] =  ColourToGrey_Pixel(Colour[i][j].R, Colour[i][j].G, Colour[i][j].B);
46  }
```

```
1   .globl ColourToGrey_Pixel
2
3   .text
4
5   ColourToGrey_Pixel:
6
7       li x28, 306
8       mul a0, a0, x28
9
10      li x28, 601
11      mul a1, a1, x28
12
13      li x28, 117
14      mul a2, a2, x28
15
16      add a0, a0, a1
17      add a0, a0, a2
18
19      srl a0, a0, 10
20
21      ret
22
23  .end
```

Figure 3. ColourToGrey function (see *Test.c*) and ColourToGrey_Pixel function (see *assemblySubroutines.S*).

In assembly language, symbols (variables and functions/subroutines) are local by default, i.e., they are invisible to other files. To turn those local symbols into global symbols, we must export them using the `.globl` assembler directive. On the right side of Figure 3, the first line (`.globl ColourToGrey_Pixel`) exports the `ColourToGrey_Pixel` function, so that it can be used by the `ColourToGrey` function, which is in a different file (*Test.c*). On the left side of Figure 3, the first line (`extern int ColourToGrey_Pixel(int R, int G, int B)`) declares the `ColourToGrey_Pixel` function as an external function to this file.

D. Execution of the program and visualization of the results

After the grey code conversion is complete, but before the end of the program's execution, we can dump the contents of some memory regions into files. To do this, we use the `dump` command of the GDB debugger. Follow the next steps for running the project code and obtaining the image results:

1. Open Catapult. On the top menu bar, click on *File → Open Folder* and browse into directory *[RVfpgaBasysPath]/Labs/Lab04*. Select directory *ImageProcessing* (do not open it, but just select it) and click OK at the top of the window. Catapult will now open the project.

2. Run the program using RVfpgaEL2-Whisper as explained in the GSG. (The memory available in RVfpgaEL2-Basys3 is not enough for the images to fit in it, so we run it in simulation only.)

3. After a short time (about 1 second), the program will have completed the greyscale image transformations described above, and it will have reached the infinite (`while(1);`) loop at the end (see Figure 2). Pause the execution by clicking on the *Pause* button.

4. Export the grey image (`GreyImage`), by running the following commands in the Debug Console:

```
cd
[RVfpgaBasysPath]/Labs/Lab04/ImageProcessing/AdditionalFi
les

dump value GreyImage.dat GreyImage
```

5. Transform the `.dat` file into `.ppm` file that you can view in your system.

In **LINUX:** do this by opening a terminal and typing the following commands:

```
cd [RVfpgaBasysPath]/Labs/Lab04/ImageProcessing/AdditionalFiles

gcc -o dump2ppm dump2ppm.c

./dump2ppm GreyImage.dat GreyImage.ppm 128 128 1
```

In WINDOWS: do this by either:

1. Using the *dump2ppm.exe* executable provided in *[RVfpgaBasysPath]\Labs\Lab04\ImageProcessing\AdditionalFiles*. Open a command shell, go into that folder, and run the executable with the same arguments as above:

```
dump2ppm.exe GreyImage.dat GreyImage.ppm 128 128 1
```

Or

2. Using Cygwin (if you installed it as described in the RVfpga Getting Started Guide) to compile the *dump2ppm.c* program. Then run the program (*dump2ppm.exe*) in the Cygwin terminal or in a command shell as in option 1 above.

6. Open the `.ppm` file using GIMP, the GNU Image Manipulation Program. If that program is not already installed, go to the following website to download the installer:

   https://www.gimp.org/downloads/

   The greyscale image should look like the one shown on the right side of Figure 1 (you can also access the input colour image at *[RVfpgaBasysPath]/Labs/Lab04/ImageProcessing/AdditionalFiles/VanGogh_128.ppm*, which should look like the one shown on the left side of Figure 1).

3. Exercises

Exercise 1. Execute the program on a different input image. You can use the image provided at: *[RVfpgaBasysPath]/Labs/Lab04/ImageProcessing/src/TheScream_256.c* (You can view the corresponding `.ppm` image at: *[RVfpgaBasysPath]/Labs/Lab04/ImageProcessing/AdditionalFiles/TheScream_256.ppm.* You will also create this image by running the program *dat2ppm/dat2ppm.exe*, as described earlier.)

Exercise 2. Create a C function that counts the number of close to white (>235) and close to black (<20) elements in the *VanGogh* greyscale image. Print the two numbers on the serial console using Western Digital's PSP and BSP libraries, as explained in Section 3 of Lab 1.

Exercise 3. Transform the `ColourToGrey_Pixel` assembly subroutine into a C function, and the C function `ColourToGrey` into an assembly subroutine that invokes the `ColourToGrey_Pixel` C function.

- In C, all functions and global variables are exported global symbols by default, so you can use the `ColourToGrey_Pixel` function in subroutine `ColourToGrey`.

- For accessing a matrix in assembly language, you must calculate the address of an element (`i,j`), given the starting address of the array. According to the ANSI C standard, two-dimensional arrays are stored in memory by rows. Thus, the address of the pixel in row `i` and column `j` is obtained by adding the starting address of the array and the offset (`i*M` + `j`)*`B`, where `M` is the number of columns and `B` is the number of bytes occupied by each pixel: three bytes in RGB and only one in greyscale.

Exercise 4. Apply a **Blur Filter** to the *VanGogh* colour image (you can find lots of information online; for example, you can use the information available at: https://lodev.org/cgtutor/filtering.html#Find_Edges).

Note that for transforming the `.dat` image into a `.ppm` image you must modify a bit the *dump2ppm/dum2ppm.exe* command invocation for considering 3 channels instead of only 1:

```
./dump2ppm FilterColourImage.dat FilterColourImage.ppm 128 128 3
```

Moreover, you can compare the filtered image with the original one, which is available at *[RVfpgaBasysPath]/Labs/Lab04/ImageProcessing/AdditionalFiles/VanGogh_128.ppm*