



PROGRAMA UNIVERSITARIO DE IMAGINATION

Práctica 5 RVfpga

Procesamiento de imágenes: C y ensamblador

1. INTRODUCCIÓN

En esta práctica construirá proyectos de programas para RISC-V que realizan rutinas de procesamiento de imágenes. Los proyectos incluirán múltiples archivos fuente, algunos de los cuales están escritos en C y otros en ensamblador. Se mostrará cómo las funciones C pueden invocar rutinas en ensamblador y viceversa.

2. TUTORIAL DE PROCESAMIENTO DE IMÁGENES

Comience esta práctica examinando un programa que procesa una imagen RGB (lado izquierdo de la Figura 1), y genera una versión en escala de grises de esa imagen (lado derecho de la Figura 1). El programa está escrito en los lenguajes C y ensamblador de RISC-V, está configurado para ejecutarse en el entorno de PlatformIO y está disponible en:

`[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing`

El código fuente se encuentra en el subdirectorio `src`.

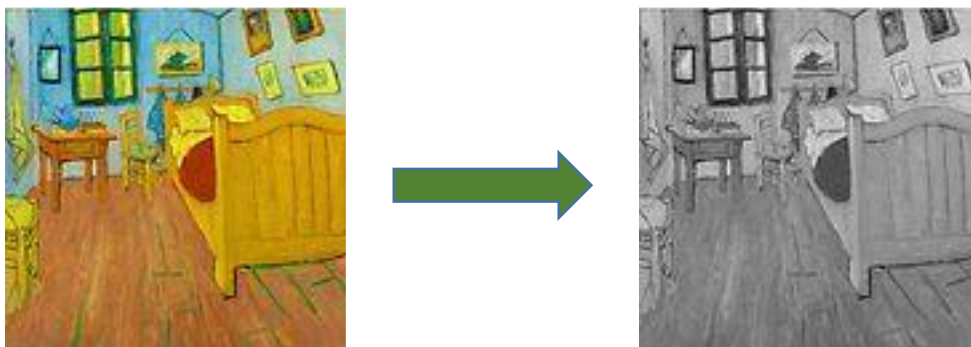


Figura 1. Transformación de una imagen RGB a una imagen en escala de grises.

A. Estructura y función *principal* del proyecto

El programa consta de los siguientes archivos fuente: **main.c**, **VanGogh_128.c** y **assemblySubroutines.S**. Los archivos `.c` contienen funciones (como las funciones para realizar las transformaciones de la imagen) y declaraciones de variables (como la imagen de entrada, declarada como un array de caracteres sin signo). El archivo **assemblySubroutines.S** contiene una implementación en lenguaje ensamblador de la función que transforma la imagen de rgb a escala de grises, llamada *ColourToGrey_Pixel*.

Figura 2 muestra la función `main` de este proyecto. En primer lugar, invoca a la función *initColourImage*, que crea una matriz $N \times M$ con los datos de la imagen de entrada. Luego transforma la imagen en color a una imagen en escala de grises (función *ColourToGrey*). Finalmente, la función imprime un mensaje y entra en un bucle infinito (`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  }

```

Figura 2. Función *main* en el proyecto ImageProcessing

B. Imágenes RGB y en escala de grises

Una imagen consiste en una matriz de píxeles, en donde cada elemento de la matriz representa el valor de un píxel en una escala dada. En RGB, cada píxel está compuesto por tres valores, que corresponden a la intensidad luminosa de los componentes rojo (**R**), verde (**G**) y azul (**B**). Por lo tanto, cada píxel de una imagen en color será un vector de tres componentes. En este proyecto se utiliza la siguiente definición para el tipo píxel RGB:

```

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

```

Este código define una estructura llamada *RGB*. En C, un tipo de datos `struct` es una colección de variables, posiblemente de diferentes tipos, especificada por un solo nombre. Esta estructura contiene tres campos, todos del mismo tipo (`unsigned char`), llamados *R*, *G* y *B*. Así, cada canal de color está representado por 8 bits, de modo que se puede distinguir entre 256 niveles de intensidad diferentes en cada canal de color, para un total de 24 bits por píxel (24bpp). Este formato es típico en el procesamiento actual de imágenes digitales.

Para representar una imagen en escala de grises, un único valor (un solo canal) que va de 0 a 255 indica el brillo de cada píxel. En este proyecto de procesamiento de imágenes, se representa la imagen en escala de grises utilizando un array de caracteres bidimensional:

```

unsigned char GreyImage[N][M];

```

C. Transformación de una imagen en color a escala de grises

La transformación entre los dos espacios de color (RGB y Escala de Grises) se realiza mediante la siguiente suma ponderada:

$$\text{gris} = 0,299 * R + 0,587 * G + 0,114 * B$$

Esta ecuación se basa en los algoritmos descritos en <https://www.mathworks.com/help/matlab/ref/rgb2gray.html>

Para cada píxel, se calcula el valor en escala de grises multiplicando cada canal de color por el peso dado en la ecuación. La suma de los pesos (0,299+0,587+0,114) es uno, por lo que el valor resultante en escala de grises estará dentro del rango 0-255, y por lo tanto puede ser representado con un solo byte.

Para usar los pesos dados en la ecuación, es necesario operar con números reales, sin embargo el procesador **SweRV EH1** no incluye soporte de punto flotante. Una posibilidad sería usar emulación de punto flotante, como en el programa DotProduct que se muestra en la Sección 6.H de la Guía de Inicio, sin embargo, en esta práctica se usará una estrategia basada aritmética de enteros. Los pesos se convierten a números enteros y la suma es una potencia de dos (en este caso, 2^{10}). Para convertir los pesos a números enteros, se multiplica cada peso de punto flotante por 2^{10} y se redondea al número entero más cercano:

- $0.299 \times 2^{10} = 306.176 \approx \mathbf{306}$ (peso para R)
- $0.587 \times 2^{10} = 601.088 \approx \mathbf{601}$ (peso para G)
- $0.114 \times 2^{10} = 116.736 \approx \mathbf{117}$ (peso para B)

Por supuesto, para reducir el valor final de la escala de grises al rango de 0-255 se debe dividir la suma por 2^{10} , lo que se puede realizar fácilmente desplazando el valor 10 bits a la derecha. Así, la transformación final se obtiene usando la siguiente fórmula:

$$\text{gris} = (306 \times R + 601 \times G + 117 \times B) \gg 10$$

Obsérvese que, dado que la suma de las constantes (306+601+117) es 1024, el valor resultante de la escala de grises seguirá estando dentro del rango 0-255.

Figura 3 muestra el código de la función *ColourToGrey* (lado izquierdo) y la subrutina *ColourToGrey_Pixel* (lado derecho) que *ColourToGrey* invoca.



```

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
    
```

Figura 3. Función *ColourToGrey* (implementada en el archivo *main.c*) y subrutina *ColourToGrey_Pixel* (implementada en el archivo *assemblySubroutines.S*).



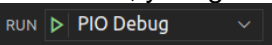
En lenguaje ensamblador, los símbolos (variables y funciones/subrutinas) son locales por defecto, es decir, son invisibles para otros archivos. Para convertir esos símbolos locales en símbolos globales, se debe exportarlos usando la directiva de ensamblador `.globl`. En la parte derecha de la Figura 3, la primera línea (`.globl ColourToGrey_Pixel`) exporta la función *ColourToGrey_Pixel*, de modo que pueda ser utilizada por la función *ColourToGrey*, que está en un archivo diferente (*main.c*). En el lado izquierdo de la Figura 3, la primera línea (`extern int ColourToGrey_Pixel(int R, int G, int B)`) declara la función *ColourToGrey_Pixel* como una función externa a este archivo.

D. Ejecución del programa y visualización de los resultados

Después de que la conversión a escala de grises finalice, pero antes de que termine la ejecución del programa, se puede volcar a archivos el contenido de algunas regiones de memoria. Para ello, se utilizará el comando `dump` del depurador GDB. Siga los siguientes pasos para ejecutar el código del proyecto y obtener los resultados de la imagen:


1. Abrir VSCode y PlatformIO.
2. En la barra del menú superior, haga clic en *File* → *Open Folder* y navegue hasta el directorio `[RVfpgaPath]/RVfpga/Labs/Lab5`. Seleccione la carpeta *ImageProcessing* (no la abra, sólo selecciónela) y haga clic en OK, en la parte superior de la ventana. PlatformIO abrirá el proyecto.
3. Abra `platformio.ini`, descomente la línea `board_build.bitstream_file` e introduzca la ubicación del directorio del archivo bitfile. Por ejemplo, utilice el archivo bitfile que creó en la Práctica 1.

```
board_build.bitstream_file =
[RVfpgaPath]/RVfpga/Labs/Lab1/Project1/Project1.runs/impl_1/rvfpganexys.bit
```

4. Abra todos los archivos fuente ubicados en el directorio `src` (`main.c`, `assemblySubroutines.S`) y analícelos para comprender cómo funciona el programa.
5. Descargue RVfpgaNexys en la placa Nexys A7 haciendo clic en el icono de PlatformIO en la barra del menú de la izquierda, luego expandiendo Project Tasks → `env:swervolf_nexys` → Platform y haciendo clic en Upload Bitstream. Recuerde que puede ejecutar el programa en simulación, tanto en Verilator como en Whisper.
6. Ejecute el programa en PlatformIO. Puede hacerlo en la placa (en cuyo caso debe descargar primero RVfpgaNexys a la Nexys A7, como se hizo en el paso anterior) o usando el simulador Whisper (como se describe en la Guía de Inicio de RVfpga). En cualquier caso, haga clic en el botón "Run" , que se encuentra en la barra lateral izquierda de PlatformIO, y luego inicie el depurador haciendo clic en el botón de reproducción . .

La ejecución se detendrá al principio de la función `main`, por lo que debe reanudarla haciendo clic en el botón "Continue" .

Después de un corto periodo de tiempo (alrededor de 1 segundo), el programa habrá finalizado la transformación de la imagen a escala de grises explicada anteriormente y habrá alcanzado el bucle infinito (`while(1);`) al final del programa (ver Figura 2).

Detenga la ejecución haciendo clic en el botón *Pause* .

7. Export la imagen en escala de grises (`GreyImage`), ejecutando los siguientes comandos en la Consola de Depuración (ver Figura 4, que muestra la ejecución de estos dos comandos):

```
cd AdditionalFiles

dump value GreyImage.dat GreyImage
```

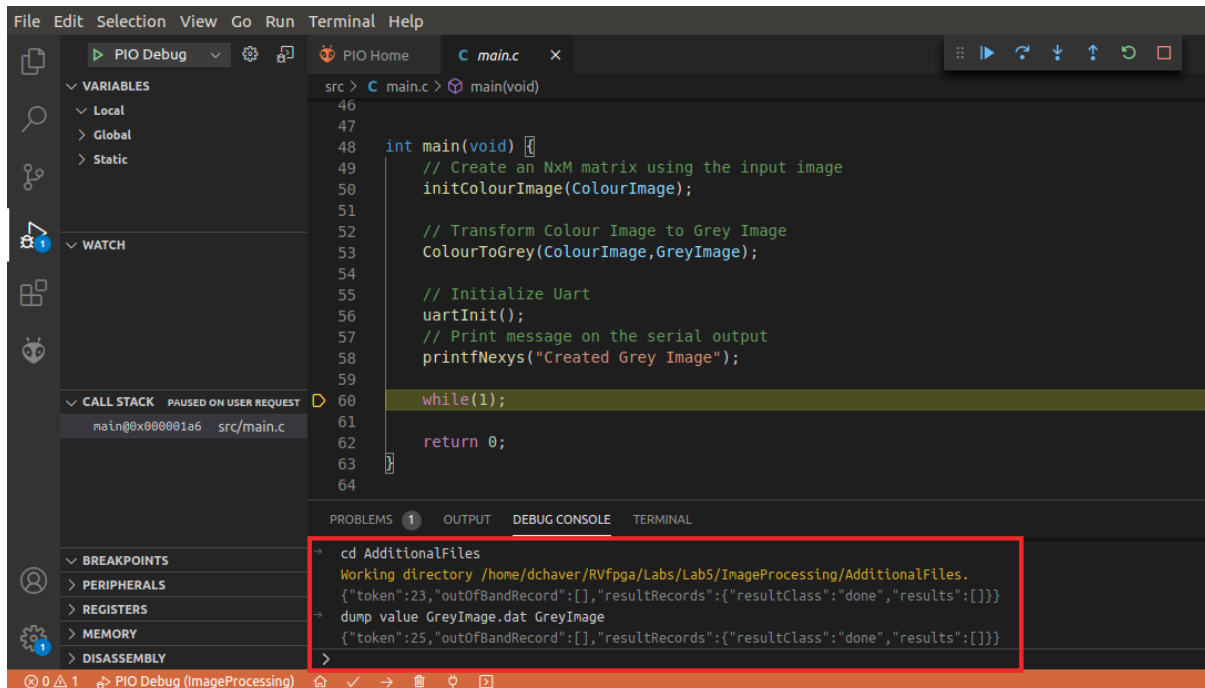


Figura 4. Exportar la imagen en escala de grises a un archivo

8. Transforme el archivo `.dat` en un archivo `.ppm` que pueda visualizar en su sistema.

En **LINUX**: Abra un terminal y escriba los siguientes comandos (ver Figura 5):

```
cd [RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/AdditionalFiles
gcc -o dump2ppm dump2ppm.c
./dump2ppm GreyImage.dat GreyImage.ppm 128 128 1
```

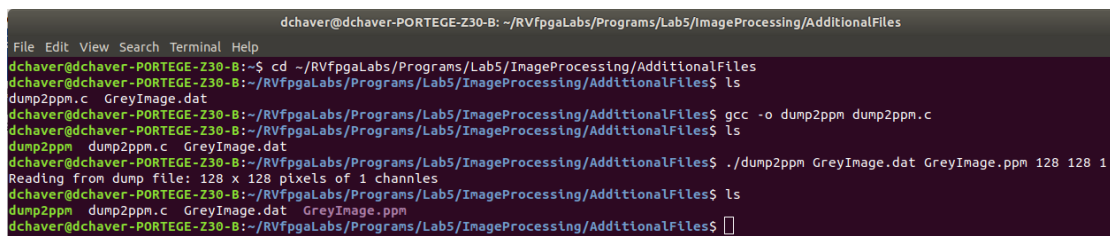


Figura 5. Transformación de la imagen a formato `.ppm`

En WINDOWS: Realice alguno de estos dos pasos:

1. Use el ejecutable `dump2ppm.exe` proporcionado en `[RVfpgaPath]\RVfpga\Labs\Lab5\ImageProcessing\AdditionalFiles`. Para ello abra un terminal de comandos, entre en esa carpeta y corra el ejecutable con los mismos argumentos indicados previamente:

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

O bien

2. Use Cygwin (si lo instaló como se indica en la Guía de Inicio de RVfpga) para compilar el programa `dump2ppm.c`. Luego ejecute el programa (`dump2ppm.exe`) en el terminal Cygwin o en un terminal de comandos como en la opción 1.

9. Abra el archivo `.ppm` con GIMP, el Programa de Manipulación de Imágenes de GNU. Si este programa no está ya instalado, acceda a este sitio web para descargar el instalador:

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

La imagen en escala de grises debe ser como la que se muestra en el lado derecho de la Figura 1 (también puede acceder a la imagen de entrada en color en `[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/AdditionalFiles/VanGogh_128.ppm`, que debe visualizarse como la que aparece en el lado izquierdo de la Figura 1).

3. Ejercicios

Ejercicio 1. Ejecutar el programa previo para una imagen de entrada diferente. Puede utilizar la imagen proporcionada en:
`[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/src/TheScream_256.c` (Puede ver la imagen `.ppm` correspondiente en:
`[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/AdditionalFiles/TheScream_256.ppm`.
 También puede crear esta imagen ejecutando el programa `dat2ppm`, como se describió anteriormente.)

Ejercicio 2. Crear una función en C que en la imagen *VanGogh* en escala de grises, cuente el número de elementos cercanos al blanco (> 235) y cercanos al negro (< 20). Imprima los dos números en la consola serie usando las bibliotecas PSP y BSP de Western Digital, como se explica en la Sección 3 de la Práctica 2.

Ejercicio 3. Transformar la subrutina de ensamblador **ColourToGrey_Pixel** en una función C, y la función C **ColourToGrey** en una subrutina de ensamblador que invoque la función C **ColourToGrey_Pixel**.

- En C, todas las funciones y variables globales se exportan como símbolos globales por defecto, por lo que se puede utilizar la función *ColourToGrey_Pixel* en la subrutina *ColourToGrey*.
- Para acceder a una matriz en lenguaje ensamblador hay que calcular la dirección de un elemento (i,j) , dada la dirección inicial de la matriz. Según la norma ANSI C, las matrices se almacenan en la memoria por filas. Así, la dirección del píxel de la fila i y la columna j se obtiene sumando la dirección inicial de la matriz y el desplazamiento $(i*M + j)*B$, donde M es el número de columnas y B es el número de bytes ocupados por cada píxel: tres bytes en RGB y sólo uno en escala de grises.

Ejercicio 4. Aplicar un **Filtro Blur** a la imagen *VanGogh* en color (puede encontrar mucha información sobre este filtro en la web; por ejemplo, se puede utilizar la información disponible en: https://lodev.org/cgtutor/filtering.html#Find_Edges).

Tenga en cuenta que para transformar la imagen `.dat` en una imagen `.ppm` hay que modificar ligeramente la invocación del comando `dump2ppm` para considerar 3 canales en lugar de sólo 1:

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

Además, el usuario puede comparar la imagen filtrada con la original, que está disponible en
`[RVfpgaLabsPath]/RVfpgaLabs/Programs/Lab5/ImageProcessing/AdditionalFiles/VanGogh_128.ppm`