Práctica 4

Proyectos con varios ficheros fuente

Mezclando C y ensamblador

4.1 Objetivos

En esta práctica profundizaremos en nuestros conocimientos sobre la programación en ensamblador sobre ARM creando proyectos con varios ficheros fuente, algunos de ellos escritos en lenguaje C y otros escritos en lenguaje ensamblador. Los objetivos concretos son:

- Comprender la diferencia entre variables locales y variables globales, en su almacenamiento a bajo nivel.
- Comprender el significado de símbolos estáticos (variables y funciones).
- Analizar los problemas que surgen cuando queremos utilizar varios ficheros fuente y comprender cómo se realiza la resolución de símbolos.
- Comprender la relación entre el código C que escribimos y el código máquina que se ejecuta.
- Saber utilizar desde un programa escrito en C variables y rutinas definidas en ensamblador, y viceversa.
- Comprender el código generado por el compilador gcc.

4.2 Pasando de C a ensamblador

La informática no se habría desarrollado hasta los niveles actuales si toda la programación se tuviera que hacer obligatoriamente utilizando lenguaje ensamblador. El programador debería utilizar un lenguaje más cercano al lenguaje natural y matemático, lo que le daría mayor legibilidad y mayor facilidad de codificación, y le permitiría una mejor estructuración de los datos y una mayos abstracción procedimental. A esto se le denomina Lenguaje de Programación de Alto Nivel, ejemplos de este lenguaje son C, C++, C#, Pascal, Basic …

Para que un lenguaje de alto nivel pueda ser ejecutado por una máquina concreta es necesario traducir ese lenguaje a ceros y unos para lo que se utiliza un compilador. Simplificándolo mucho podemos decir que un compilador es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación (generalmente ensamblador), generando un programa equivalente que la máquina es capaz de interpretar. Los compiladores se estructuran generalmente en tres partes: *front end* o *parser, middle end*, y *back end*. La primera parte se encarga de comprobar que el código escrito es correcto sintácticamente y de traducirlo a una representación intermedia independiente del lenguaje. El *middle end* se encarga de analizar el código en su representación intermedia y realizar sobre él algunas optimizaciones, que pueden tener distintos objetivos, por ejemplo, reducir el tiempo de ejecución del código final, reducir la cantidad de memoria que utiliza o reducir el consumo energético en su ejecución. Finalmente el *back end* se encarga de generar el código máquina para la arquitectura destino (generalmente dan código ensamblador como salida y es el ensamblador el que produce el código máquina final).

Dependiendo del compilador y de las opciones de compilación el código ensamblador generado puede variar tremendamente. En la Figura 4.1 y 4.2 se puede observar el resultado que se obtiene sobre el código de la práctica 2b con distintas opciones de compilación. En la Figura 4.1 se representa el código obtenido tras compilar el ejemplo de la práctica 2b el compilador con las opciones de depuración (que son las que se tienen por defecto en el *workspace* de la asignatura). Si no tenemos en cuenta las instrucciones necesarias para comenzar la ejecución, se genera un código de 232 instrucciones. ¿Cuántas instrucciones tiene el código ensamblador que presentasteis en la práctica 2b? En la Figura 4.1 se ha utilizado la opción de compilación –O1, en este caso el código generado está optimizado y se parece bastante más a el código que generasteis en la práctica 2b aunque por motivos de optimización las instrucciones no aparecen en el mismo orden que en el código C.

Los pros de programar en C (o cualquier otro lenguaje de alto nivel) es que es fácil de aprender, portable, independiente de la arquitectura y nos facilita la construcción y gestión de estructuras de datos. Sin embargo, no se puede acceder y gestionar las estructuras de datos internas a la arquitectura (registros y pila) y no se tiene control directo sobre la secuencia de generación de instrucciones (una instrucción de C no se corresponde con una única instrucción de ensamblador).

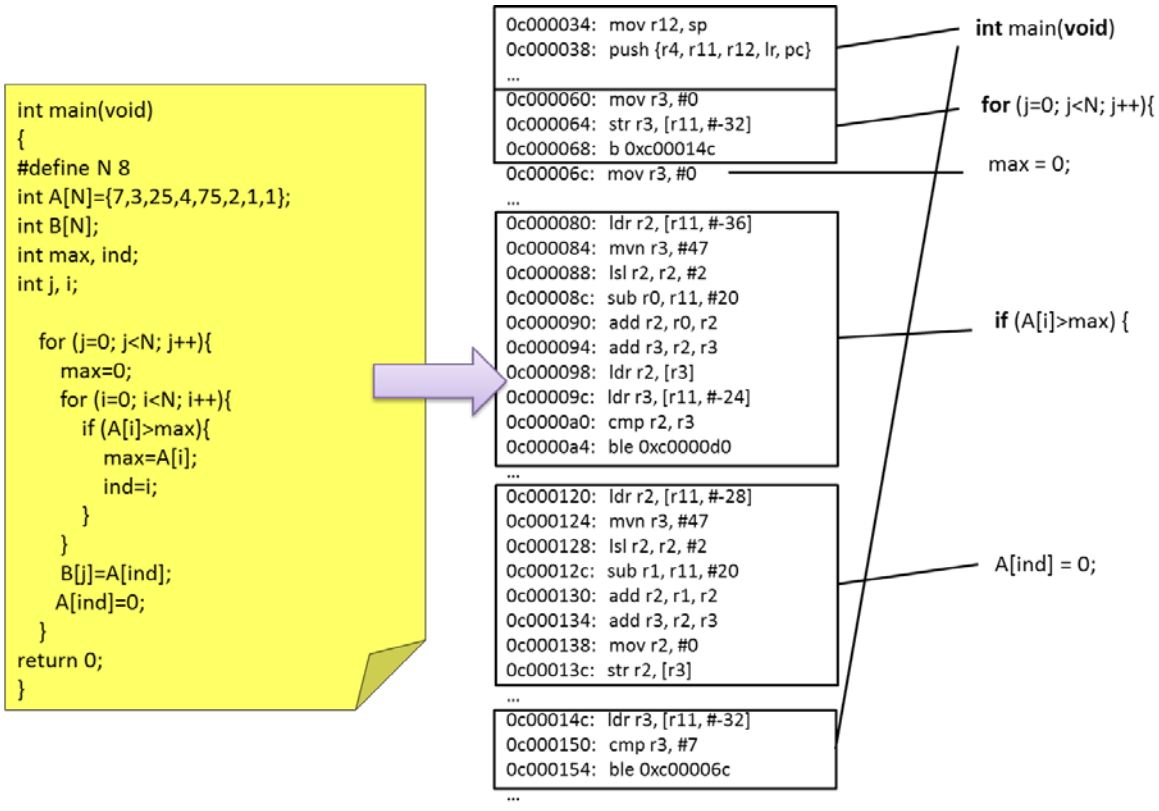

Figura 4.1. Código ensamblador sin optimizar, resultado por defecto.

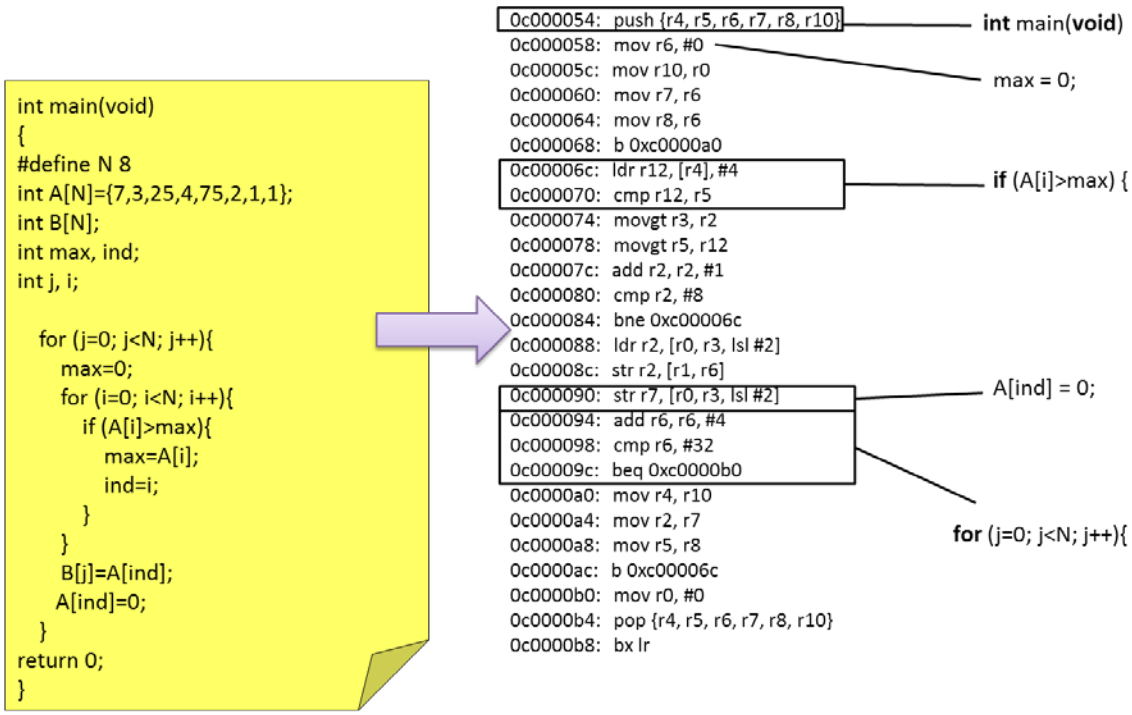

Figura 4.2. Código ensamblador compilado con optimización –O1

Mientras que si se programa en ensamblador tenemos un control directo sobre cada instrucción y sobre todas las operaciones con memoria y se pueden utilizar instrucciones que no genera el compilador de C. Sin embargo, es más difícil de aprender y no es nada portable, además la gestión de estructuras de datos complicadas se hace casi imposible.

Falsos mitos:

- Programando directamente en lenguaje ensamblador SIEMPRE se obtienen mejores códigos que los obtenidos tras compilar C.
- C con las adecuadas opciones de compilación SIEMPRE puede obtener un código que cumpla los requisitos de rendimiento sin necesidad de tocar el código ensamblador.

4.2.1 Compilando un proyecto en C

El programa a ejecutar no está sólo formado por las instrucciones necesarias para llevar a cabo una tarea específica, está formado por un conjunto de programas, bibliotecas, … necesarios para ejecutar la aplicación que se esté desarrollando. También hay varios tipos de datos: variables locales, variables globales y constantes (no tienen por qué ser sólo datos constantes, también pueden ser direcciones de memoria). Tenemos un proyecto con varios ficheros fuente, por lo tanto utilizaremos en alguno de ellos variables o subrutinas definidas en otro. Para solucionar este problema como vimos en la práctica 2 (Figura 2.1), la etapa de compilación se hace de

forma independiente sobre cada fichero y es una etapa final de enlazado la que combina los ficheros objeto formando el ejecutable. En ésta última etapa se deben resolver todas las referencias cruzadas entre los ficheros objeto.

4.2.1.1 Definición de variables globales y estáticas en C

Cuando se programa en C utilizando distintos ficheros puede ocurrir que necesitemos hacer referencias a símbolos, variable o funciones globales definidos en el otro. Una variable (símbolo) global es aquella que puede ser accedida por cualquier función, es decir, su ámbito son todas las funciones que componen el programa. Posteriormente, los ficheros objeto correspondientes se enlazarán para formar un único ejecutable.

- Puede ocurrir que queramos que esa variable sea accedida por las funciones de otro *fichero.c*

```
extern int var1;
```

- O que se accedida sólo por las funciones del *fichero.c* donde se encuentra

```
static int var2;
```

- Este mismo protocolo se puede aplicar a las funciones

En C todas las variables globales y todas las funciones son por defecto símbolos globales exportados. Para utilizar una función definida en otro fichero debemos poner una declaración adelantada de la función.

4.2.1.2 Generación de la tabla de símbolos

El contenido de un fichero objeto es independiente del lenguaje en que fue escrito el fichero fuente. Es un fichero binario estructurado que contiene una lista de secciones con su contenido, y una serie de estructuras adicionales. Una de ellas es la Tabla de Símbolos, que, como su nombre indica, contiene información sobre los símbolos utilizados en el fichero fuente. Las Tablas de Símbolos de los ficheros objeto se utilizan durante el enlazado para resolver todas las referencias pendientes. La tabla de símbolos de un fichero objeto en formato elf puede consultarse con el programa nm, por ejemplo así:

```
> arm-none-eabi-nm  -SP -f sysv ejemplo.o
Symbols from ejemplo.o:

Name        Value       Class  Type       Size        Line  Section
globalA     |00000000   | D   | OBJECT    |00000002|       |.data
globalB     |           | U   | NOTYPE    |        |       |*UND*
main        |00000000   | T   | FUNC      |00000054|       |.text
printf      |           | U   | NOTYPE    |        |       |*UND*
```

La información anterior nos dice:

- Hay un símbolo globalA , que comienza en la entrada 0x0 de la sección de datos (.data) y de tamaño 0x2 bytes. Es decir, será una variable de tamaño media palabra.
- Hay otro símbolo globalB que no está definido (debemos importarlo). No sabemos para qué se va a usar, debe estar definido en otro fichero.
- Hay otro símbolo main, que comienza en la entrada 0x0 de la sección de código (.text) y ocupa 0x48 bytes. Es la función de entrada del programa C.
- Hay otro símbolo printf, que no está definido (debemos importarlo de la biblioteca estándar de C).

4.3 Combinando C y ensamblador

La mayor parte de los proyectos reales se componen de varios ficheros fuente. La mayoría en un lenguaje de alto nivel C/C++, donde se trabajará con distintos ficheros. Además, se programa en ensamblador aquellas partes donde se tengan requisitos estrictos de eficiencia o bien porque se necesite utilizar directamente algunas instrucciones especiales de la arquitectura.

Para combinar código C con ensamblador resulta conveniente dividir el programa en varios ficheros fuente:

- Los ficheros en C deben ser compilados
- Los ficheros en ensamblador sólo deben ser ensamblados.

4.3.1. Definición de variables globales y estáticas en ensamblador

Para utilizar un símbolo exportado globalmente desde un fichero ensamblador dentro de un código C, debemos hacer una declaración adelantada del símbolo, como ya hemos visto en la sub-sección 4.2.1.1. Debemos tener en cuenta que el símbolo se asocia con la dirección del identificador. En el caso de que el identificador sea el nombre de una rutina en ensamblador esto corresponde a la dirección de comienzo de la misma.

El protocolo a seguir sería el siguiente:

– Para poder usar la subrutina en C, el símbolo (nombre de la rutina) debe haberse declarado como externo en el programa ensamblador

```
.global subrutinaARM
```

– Además debemos declarar en el fichero C una función con el mismo nombre que dicho símbolo, así como el tipo de todos los parámetros que recibirá la función y el valor que devuelve

```
extern int subrutinaARM(int, int);
```

– El mismo protocolo se sigue para las variables compartidas

Un ejemplo de utilización en un programa de C de variables y funciones definidas en ensamblador puede verse en la siguiente figura 4.3.

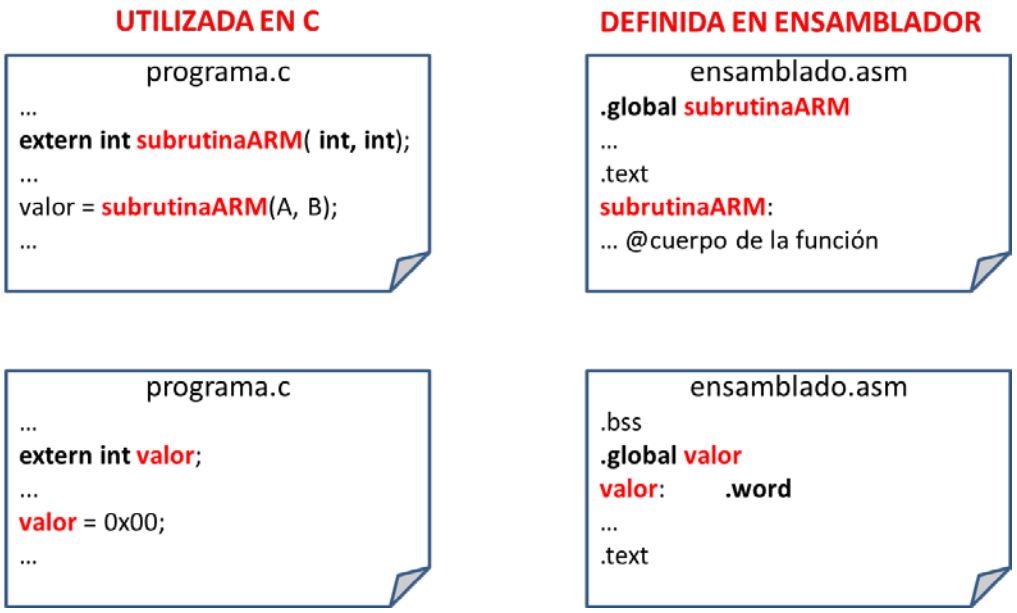

Figura 4.3. Utilizar en C subrutinas definidas en ensamblador

También se pueden utilizar funciones y variables definidas en C en un programa ensamblador, para lo que se sigue un protocolo muy parecido al anterior:

– Si queremos usar en ensamblador una rutina implementada en C, el nombre de la rutina debe haberse declarado como externo en C

```
extern int subrutinaC (int, int);
```

– Además debemos declarar a la rutina en C en el código ensamblador

```
.extern subrutinaC
```

– El mismo protocolo se sigue para las variables compartidas

Un ejemplo de utilización en un programa de ensamblador de variables y funciones definidas en C puede verse en la figura 4.4.

4.3.2. Código de Arranque

La función principal que constituye el punto de entrada al programa, suele llamarse *main*. Esta función no es diferente de ninguna otra función, en el sentido de que construirá en la pila su marco de activación y al terminar lo deshará y retornará, copiando en PC lo que tuviese el registro LR al comienzo de la propia función. La función *main* es la encargada de iniciar el programa, pero no de iniciar el sistema.

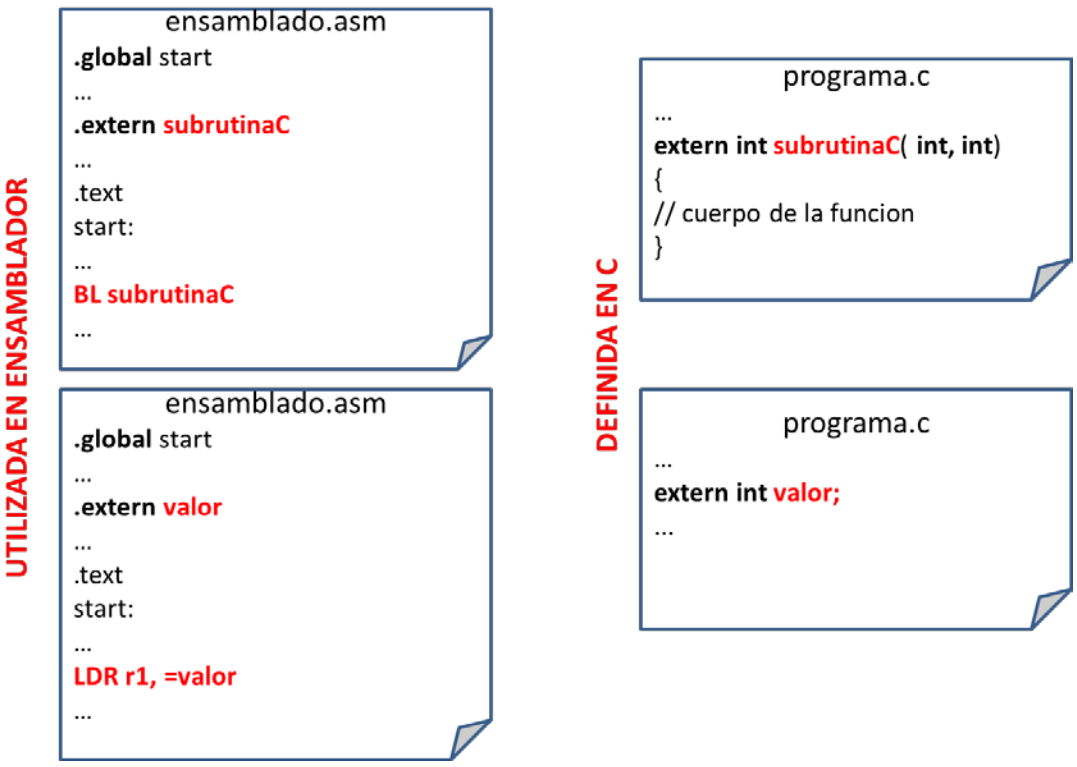

Figura 4.4. Utilizar en ensamblador subrutinas definidas en C

El programa encargado de iniciar el sistema, se denomina código de arranque, siendo este código un código en ensamblador que utiliza funciones en C .

El código de arranque se utiliza para "inicializar" la memoria: Inicializa la pila e inicializa el frame pointer. Pone parte de memoria a cero para aquellas variables que no se inicializan en tiempo de carga (*load time*). Además, para las aplicaciones de C que utilizan funciones como malloc(), este programa inicializa el *heap*

Después de estas inicializaciones, el programa (*C startup code)* salta la posición de memoria donde comienza el programa *main().* El compilador/enlazador es el encargado de insertar automáticamente el programa *C startup code.* Este programa no es necesario crearlo si se está trabajando directamente en ensamblador.

Ejemplo de un código de arranque en C:

```
        .extern main
        .extern _stack
        .global start
start:  ldr sp,=_stack
        mov fp,#0
        bl main
End:    b End
.end
```

Los compiladores de ARM denominan al *C startup code* como "_main," mientras que los compiladores de C en GNU lo suelen llamar "_start."

4.4 Desarrollo de la práctica

4.4.1 Qué hace el código

El código C de la práctica traduce una imagen RGB a una imagen formada por ceros y unos, y termina contando el número de unos que aparecen por filas en la imagen resultado.

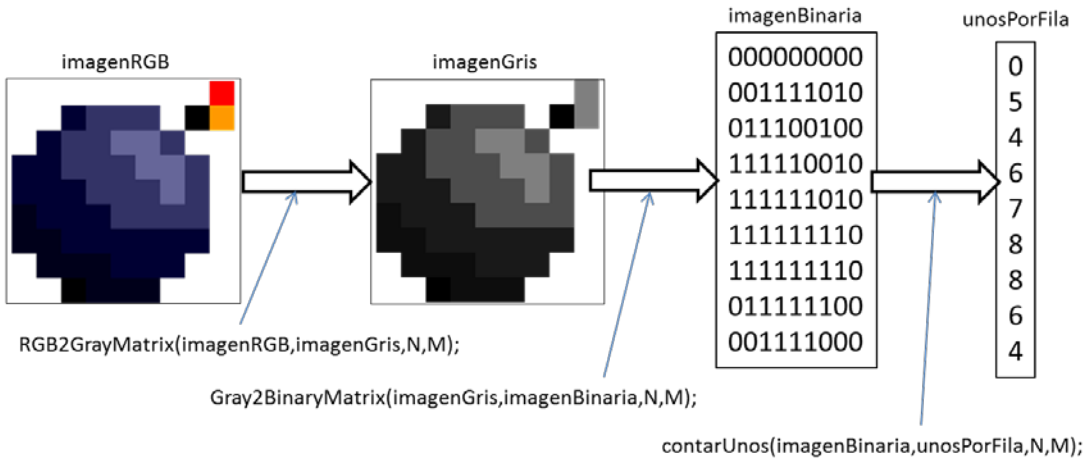

Figura 4.5. Esquema de funcionamiento de la práctica 4

Por lo tanto el código *main* de la práctica 4 será el siguiente:

```
int main() {

    // 1. Crear una matriz NxM de diferentes colores

       initRGB(imagenRGB,N,M);

    // 2. Traducir la matriz RGB a una matriz de grises

    RGB2GrayMatrix(imagenRGB,imagenGris,N,M);

    // 3. Traducir la matriz de grises a una matriz en blanco y negro

    Gray2BinaryMatrix(imagenGris,imagenBinaria,N,M);

    // Contar los unos que aparecen por filas en la matriz de blanco y negro

    contarUnos(imagenBinaria,unosPorFila,N,M);

    return 0;
}
```

- La función *InitRGB* crea una imagen por defecto de tamaño *NxM* pixels (esta función puede ser reemplazada por una función que lea una imagen de un fichero).

- La función RGB2GrayMatrix parte de una imagen en color (*imagenRGB*) y crea una imagen en grises (*imagenGris*) aplicando un algoritmo sencillo que da un peso a cada una de las componentes RGB para conseguir el gris:

imagenGris[i,j] = factorR*imagenRGB[i,j].R + factorG*imagenRGB[i,j].G+
factorB*imagenRGB[i,j].B

- La función *Gray2BinaryMatrix* parte de una imagen en grises (*imagenGris*) y la transforma en una imagen en ceros y unos (*imagenBinaria*), siendo uno si el gris supera un umbral y siendo cero si el gris no lo supera.
- Por último la función *contarUnos* devuelve un vector (*unosPorFila*) donde aparecen el número de unos que hay por filas en la matriz binaria (*imagenBinaria*).

Todas las funciones se encuentran definidas en el fichero trafo.h y el cuerpo de las funciones se encuentra en el fichero trafo.c.

4.4.2 Cómo lo hace

Para explicar cómo trabaja el código C y cómo se traduce a ensamblador vamos a utilizar la función RGB2GrayMatrix.

```
void RGB2GrayMatrix(pixelRGB orig[N][M], int dest[N][M], int nfilas, int
ncols) {
    int i,j;

    for (i=0;i<nfilas;i++)
        for (j=0; j<ncols; j++)
            dest[i][j] =  rgb2gray(orig[i][j]);

}

int rgb2gray(pixelRGB pixel) {
    return  ( (2048*pixel.R + 4096*pixel.G + 512*pixel.B) /(8*1024));
}
```

La función RGB2Gra2Matrix recibe como entrada cuatro parámetros:

- En r0 se almacena la dirección donde comienza la matriz imagenRGB (0x0C000000)
- En r1 se almacena la dirección donde comienza la matriz imagenGris (0x0C000180)
- En r2 se almacena el número de filas
- En r3 se almacena el número de columnas

La función realiza un recorrido pixel a pixel de la matrizRGB al que aplica la función que convierte a gris un valor RGB (rgb2gray).

Cuando en C se escribe orig[i][j], el ensamblador accede al componente i,j de dicha matriz que se encuentra almacenada desde la dirección a la que apunta r0. Y obtiene los tres elementos de la estructura guardando en r0 orig[i][j].R, en r1 orig[i][j].G y en r3 orig[i][j].B.

4.4.2.1 Cómo saber conociendo el valor de i y j y la dirección inicial dónde se encuentra el elemento en una matriz

C almacena la matriz por filas en forma de vector. Entonces una matriz NxM se convierte en un vector de NxM componentes donde los primeros M componentes se corresponden a la primera fila, los segundos M componentes a la segunda fila y así sucesivamente.

0	1	2	3
4	5	6	7
8	9	A	B

La matriz del ejemplo se convierte en el vector (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B). Para acceder al componente de la fila 2 columna 1, es decir 9, hay que sobrepasar la fila 0 (4 componentes), la fila 1 (4 componentes) y luego acceder al elemento de la columna 1. Esto es equivalente a realizar la siguiente operación matemática:

Matriz[i,j] = Vector[nºcolumnas*i + j]

Dada una matriz de 6x8 enteros que comienza en la dirección 0x0C000000. ¿Dónde se encuentra la componente (3, 2)?

La matriz puede ser también una matriz de estructuras, en ese caso la traducción se haría a un vector de estructuras, donde todos los elementos de la estructura asociados a un componente de la matriz se almacena de manera consecutiva. Si tuviéramos un matriz 2x3 de una estructura formada por dos enteros tendríamos la siguiente matriz.

0.Uno	0.Dos	1.Uno	1.Dos
2.Uno	2.Dos	3.Uno	3.Dos
4.Uno	4.Dos	5.Uno	5.Dos
6.Uno	6.Dos	7.Uno	7.Dos
8.Uno	8.Dos	9.Uno	9.Dos
A.Uno	A.Dos	B.Uno	B.Dos

La matriz del ejemplo se convierte en el vector (0.Uno, 0.Dos, 1.Uno, …, B.Dos). Para acceder al componente del fila 2 columna 1, es decir 5, hay que sobrepasar la fila 0 (4 componentes (de 2 elementos cada uno)), la fila 1 0 (4 componentes (de 2 elementos cada uno)) y luego acceder a la columna 1 (2 elementos más). Esto equivale a realizar la siguiente operación matemática:

Matriz[i,j] = Vector[(nºcolumnas*i + j)*nºelementos]

Dada una matriz de 6x8 estructuras de 3 enteros que comienza en la dirección 0x0C000000. ¿Dónde se encuentra la componente (3, 2)?

4.4.3 Trabajo sobre la práctica

a) Compilar el proyecto y comprobar su correcto funcionamiento. Conviene examinar el código ensamblador generado y que se comprende a grandes rasgos la funcionalidad de ese código ensamblador.

b) Traducir la función contarUnos a código ensamblador.

i) Borrar la descripción de la función en trafo.h
ii) Borrar el código de la función en trafo.c
iii) Indicar en main.c que la función contarUnos se encuentra en ensamblador
iv) Programar la función contarUnos en el archivo rutinas_asm.asm

```
.global contarUnos

.text
contarUnos:
        push {...}



        mov pc,lr
```