Práctica 4

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*.
- Conocer la representación de los tipos estructurados propios de los lenguajes de alto nivel.

En esta práctica el alumno tendrá que crear un programa, escribiendo unas partes en C y otras partes en ensamblador, de forma que las rutinas escritas en C puedan invocar rutinas escritas en ensamblador y viceversa.

4.2. Variables locales

Para introducir el concepto de variables locales vamos a partir del ejemplo de la siguiente declaración de una función C:

```
int funA( int a1, int a2, int a3, int a4, int a5);
```

Como sabemos, esta declaración nos dice que la función tendrá, al menos, 5 *variables locales* llamadas `a1-a5` respectivamente. No debemos confundir las variables locales con los parámetros de la llamada, estos son los valores iniciales que debemos dar a estas variables. Por ejemplo, una posible llamada en C podría ser `a = funA(1,3,6,4,b);`, donde pasamos los valores 1,3,6 y 4 con los que se deberá inicializar las variables locales `a1-4`, mientras que a `a5` le asignaremos como valor inicial lo que contenga la variable `b`.

Tendremos una versión privada de las variables locales para cada invocación de la función. Por ejemplo, supongamos que `funA` es una función recursiva, es decir, que se invoca a sí misma. Así, en una ejecución de `funA` tendremos varias *instancias activas* de `funA`. Cada una de estas instancias debe tener su propia variable `a1` (y del resto), accesible sólo desde esa instancia. Necesitamos por tanto un espacio privado para cada instancia o invocación de la función, donde podamos alojar estas variables (recordemos que una variable tiene reservado un espacio en memoria para almacenar su valor). El espacio más natural para ello es el marco de activación de la subrutina.

Por otro lado, las variables que hemos venido utilizando hasta ahora se conocen como variable globales. Recordemos que tienen un espacio de memoria reservado en las secciones *.data* o *.bss* desde que se inicia el programa, y persisten en memoria hasta que el programa finaliza. En cambio, las variables locales se crean en el prólogo de la subrutina, que reservará espacio para ellas en el marco de activación y se destruyen cuando finaliza la llamada a la subrutina. Como vemos debemos ampliar las funciones que asignamos en la práctica anterior al marco de activación.

4.2.1. Almacenamiento de variables locales

La figura 4.1 muestra el esquema detallado de la organización del marco de activación de una subrutina que presentamos en la práctica anterior. Cuando la subrutina crea su marco de activación, y tras salvar el contexto, reservará espacio para almacenar sus las variables locales. Para que se cumplan las restricciones del estándar este espacio tendrá un tamaño múltiplo de 4 bytes. La colocación de las variables locales en este espacio la decide el programador. En el cuerpo de la subrutina las variables locales suelen direccionarse con desplazamientos fijos relativos a `FP` (por tanto desplazamientos negativos).

Hay una excepción habitual a este emplazamiento de variables locales. Supongamos que una subrutina `SubA` invoca a una subrutina `SubB` que recibe 5 parámetros. Como sabemos, `SubA` debe pasar el quinto parámetro por memoria y debe reservar espacio en la cima de su propio marco de activación para almacenar este parámetro. En este caso, reservar otra vez espacio en el marco de `SubB` para almacenar la variable local correspondiente resulta un desperdicio de memoria. Por este motivo es habitual que `SubB` utilice el espacio reservado por `SubA` para almacenar su variable local. En este caso la variable se direcciona en el cuerpo de `SubB` con desplazamientos positivos a `fp`.

Tras el prólogo, lo primero que debe hacer la subrutina es inicializar las variables locales que tengan valor inicial. Entre estas estarán las variable locales declaradas en el prototipo de la función, que tendrán que ser inicializadas con los valores pasados como parámetros a la subrutina.

4.2.2. Ejemplo

Tomemos por ejemplo la siguiente función C, que devuelve el mayor de dos números enteros:

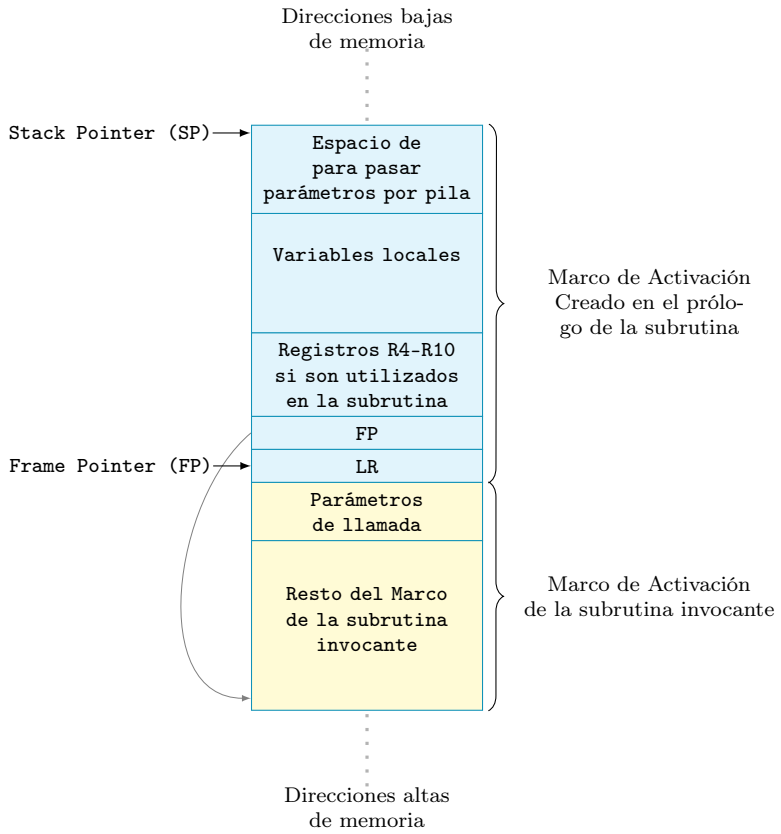


Figura 4.1: Estructura detallada del marco de activación de una subrutina.

```
int Mayor(int X, int Y){
  int m;
  if(X>Y)
    m = X;
  else
    m = Y;
  return m;
}
```

La función tiene tres variable locales, luego necesitaremos reservar 12 bytes adicionales en el marco para almacenarlas. Colocaremos por ejemplo `X` en los primeros 4 (`[fp, #-4]`), `Y` en los siguientes 4 (`[fp, #-8]`) y `m` en los últimos 4 (`[fp, #-12]`). El código ensamblador equivalente sería:

```
1  Mayor:
2       push {fp}
3       mov fp, sp
4       sub sp, sp, #12
5       str r0, [fp,#-4]  @ Inicialización de X con el primer parámetro
6       str r1, [fp,#-8]  @ Inicialización de Y con el segundo parámetro
7
8       @ if( X > Y )
9       ldr r0, [fp,#-4]  @ r0 ← X
10      ldr r1, [fp,#-8]  @ r1 ← Y
```

```
11      cmp r0, r1
12      ble ELS
13      @ then
14      ldr r0, [fp,#-4]  @ m = X;
15      str r0, [fp,#-12]
16      b RET
17      @ else
18  ELS: ldr r0, [fp,#-8]  @ m = Y;
19      str r0, [fp,#-12]
20
21  RET: @ return m
22      ldr r0, [fp,#-12] @ valor de retorno
23      mov sp, fp
24      pop{fp}
25      mov pc, lr
```

Como vemos hemos generado un código que es una traducción línea a línea del código C, siguiendo exactamente la semántica de cada línea del código C. Este código visto en ensamblador parece redundante y poco eficiente. Por ejemplo la línea 9 carga en `r0` la variable local `X`, cuando acabamos de inicializar dicha variable con el valor de `r0` y no hemos modificado el registro entre medias. Sin embargo, como se discute en la siguiente sección, este tipo de codificación es el más sencillo de seguir, entender y depurar. Por ello se recomienda al alumno el uso de este estilo de programación en ensamblador mientras esté aprendiendo.

4.3. Pasando de C a Ensamblador

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).

Cuando no se realiza ninguna optimización del código se obtiene un código ensamblador que podíamos decir que es una traducción literal del código C: cuando se genera el código para una instrucción C no se tiene en cuenta lo que se ha hecho antes con ninguna de las instrucciones, ignorando así cualquier posible optimización. La Figura 4.2 muestra un ejemplo de una traducción de la función `main` de un sencillo programa C, sin optimizaciones (`-O0`) y con nivel 2 de optimización `-O2`. Como podemos ver, en la versión sin optimizar podemos identificar claramente los bloques de instrucciones ensamblador por las que se ha traducido cada sentencia C. Para cada una se sigue sistemáticamente el mismo proceso: se cargan en registros las variables que están a la derecha del igual (loads), se hace la operación correspondiente sobre registros, y se guarda el resultado en memoria (store) en la variable que aparece a la izquierda del igual.

Este tipo de código es necesario para poder hacer una depuración correcta a nivel de C. Si estamos depurando puede interesarnos parar al llegar a una sentencia C (en la primera instrucción ensamblador generada por su traducción), modificar las variables con el depu-

rador, y ejecutar la siguiente instrucción C. Si el compilador no ha optimizado, los cambios que hayamos hecho tendrán efecto en la ejecución de la siguiente instrucción C. Sin embargo, si ha optimizado puede que no lea las variables porque asuma que su valor no ha cambiado desde que ejecutó algún bloque anterior. Por ejemplo esto pasa en el código optimizado de la Figura 4.2, donde la variable `i` dentro del bucle no se accede, sino que se usa un registro como contador. La variable `i` sólo se actualiza al final, con el valor de salida del bucle. Es decir, que si estamos depurando y modificamos `i` dentro del bucle, esta modificación no tendrá efecto, que no es lo que esperaríamos depurando a nivel de código C.

Otro problema típico, producido por la reorganización de instrucciones derivada de la optimización, es que podemos llegar a ver saltos extraños en el depurador. Por ejemplo, si el compilador ha desplazado hacia arriba la primera instrucción ensamblador generada por la traducción de una instrucción C, entonces cuando estemos en la instrucción C anterior y demos la orden al depurador de pasar a la siguiente instrucción C, nos parecerá que el flujo de ejecución vuelve hacia atrás, sin que haya ningún motivo aparente para ello. Esto no quiere decir que el programa esté mal, sino que estamos viendo un código C que ya no tiene una relación tan directa o lineal con el código máquina que realmente se ejecuta, y frecuentemente tendremos que depurar este tipo de códigos directamente en ensamblador.

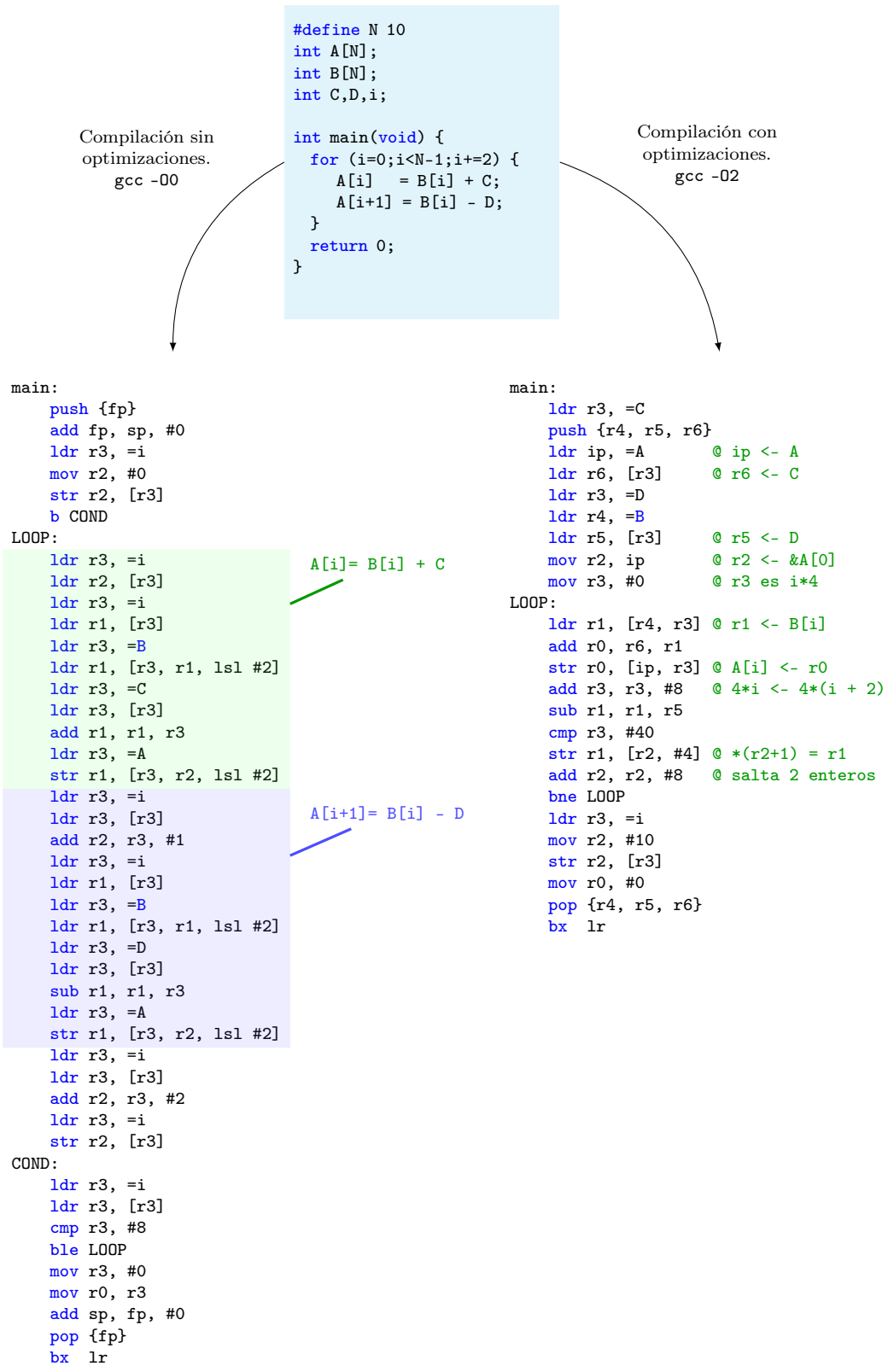
Por motivos didácticos, en esta asignatura es preferible que el alumno se acostumbre a escribir código ensamblador similar al mostrado en la parte izquierda de la Figura 4.2, a menos que se indique expresamente lo contrario. Cada variable que se modifique debería actualizarse en memoria tan pronto como se modifique su valor, evitando hacer por ejemplo lo que hace el código de la derecha con la variable global `i`. Del mismo modo se recomienda cargar de memoria el valor de cada variable cuando se vaya a utilizar, aunque ya tengamos su valor en algún registro por la ejecución de una operación anterior.

4.4. Accesos a memoria: tamaño y alineamiento

Hasta ahora todos los accesos a memoria que hemos venido haciendo con las instrucciones de load/store, han sido accesos tamaño palabra, es decir, llevarnos de un registro a memoria un dato de 32 bits o viceversa. Sin embargo los lenguajes de programación suelen ofrecer tipos básicos de otros tamaños, que en el caso de C son: `char` de tamaño byte, `short int` de 16 bits (media palabra), `long int` de 32 bits (equivalente a `int` en arquitecturas de 32 bits o más) y `long long int` de 64 bits. En todos los casos C admite el modificador `unsigned` para indicar que son enteros sin signo, ya sea de 8, 16, 32, o 64 bits.

Para trabajar con estos tipos de datos de forma eficiente el lenguaje máquina debe proporcionar instrucciones para realizar los accesos a memoria del tamaño apropiado. En ARMv4 existen instrucciones `ldr`/`str` para accesos de tamaño: byte, media palabra (*half word*, 16 bits), palabra (*word*, 32 bits) y doble palabra (*double word*, 64 bits). Para indicar el tamaño del acceso en lenguaje ensamblador se utilizan las siguientes instrucciones:

- `LDRB`: load de un entero sin signo de tamaño byte. Al escribirse el dato sobre el registro destino se extiende con ceros hasta los 32 bits.
- `LDRSB`: load de un entero con signo de tamaño byte. Al escribirse el dato sobre el registro destino se extiende con el bit de signo hasta los 32 bits.
- `STRB`: se escribe en memoria un entero de tamaño byte obtenido de los 8 bits menos significativos del registro fuente.




```
#define N 10
int A[N];
int B[N];
int C,D,i;

int main(void) {
  for (i=0;i<N-1;i+=2) {
     A[i]   = B[i] + C;
     A[i+1] = B[i] - D;
  }
  return 0;
}
```

```
main:
    push {fp}
    add fp, sp, #0
    ldr r3, =i
    mov r2, #0
    str r2, [r3]
    b COND
LOOP:
    ldr r3, =i
    ldr r2, [r3]
    ldr r3, =i
    ldr r1, [r3]
    ldr r3, =B
    ldr r1, [r3, r1, lsl #2]
    ldr r3, =C
    ldr r3, [r3]
    add r1, r1, r3
    ldr r3, =A
    str r1, [r3, r2, lsl #2]
    ldr r3, =i
    ldr r3, [r3]
    add r2, r3, #1
    ldr r3, =i
    ldr r1, [r3]
    ldr r3, =B
    ldr r1, [r3, r1, lsl #2]
    ldr r3, =D
    ldr r3, [r3]
    sub r1, r1, r3
    ldr r3, =A
    str r1, [r3, r2, lsl #2]
    ldr r3, =i
    ldr r3, [r3]
    add r2, r3, #2
    ldr r3, =i
    str r2, [r3]
COND:
    ldr r3, =i
    ldr r3, [r3]
    cmp r3, #8
    ble LOOP
    mov r3, #0
    mov r0, r3
    add sp, fp, #0
    pop {fp}
    bx  lr
```

```
main:
    ldr r3, =C
    push {r4, r5, r6}
    ldr ip, =A        @ ip <- A
    ldr r6, [r3]      @ r6 <- C
    ldr r3, =D
    ldr r4, =B
    ldr r5, [r3]      @ r5 <- D
    mov r2, ip        @ r2 <- &A[0]
    mov r3, #0        @ r3 es i*4
LOOP:
    ldr r1, [r4, r3] @ r1 <- B[i]
    add r0, r6, r1
    str r0, [ip, r3] @ A[i] <- r0
    add r3, r3, #8   @ 4*i <- 4*(i + 2)
    sub r1, r1, r5
    cmp r3, #40
    str r1, [r2, #4] @ *(r2+1) = r1
    add r2, r2, #8   @ salta 2 enteros
    bne LOOP
    ldr r3, =i
    mov r2, #10
    str r2, [r3]
    mov r0, #0
    pop {r4, r5, r6}
    bx  lr
```

Figura 4.2: C a ensamblador con gcc-4.7: optimizando (`-O2`) y sin optimizar (`-O0`)

- LDRH: load de un entero sin signo de 16 bits. Al escribirse el dato sobre el registro destino se extiende con ceros hasta los 32 bits.
- LDRSH: load de un entero con signo de 16 bits. Al escribirse el dato sobre el registro destino se extiende con el bit de signo hasta los 32 bits.
- STRH: se escribe en memoria un entero de 16 bits obtenido de los 16 bits menos significativos del registro fuente.
- LDRD: carga dos registros consecutivos con dos palabras (32 bits) consecutivas de memoria. Para facilitar la codificación de la instrucción se pone la restricción de que el primer registro debe ser un registro par y el segundo el registro impar siguiente, por ejemplo R8 y R9.
- STRD: es la operación contraria a la anterior.

En el caso de los accesos de tamaño byte contamos con los mismos modos de direccionamiento estudiados para las instrucciones ldr/str normales (tamaño palabra). En el caso de los accesos tamaño media palabra tan sólo se soportan los modos de direccionamiento indirecto de registro, indirecto de registro con desplazamiento inmediato e indirecto de registro con desplazamiento en registro.

Además, la arquitectura ARMv4 impone restricciones de alineamiento en los accesos a memoria. Se dice que un acceso está alineado si la dirección de memoria que se accede es múltiplo del tamaño del dato accedido en bytes. Así, un acceso tamaño byte no tiene ninguna restricción, ya que todas las direcciones serán múltiplo de 1. En cambio, un acceso tamaño media palabra sólo será válido si la dirección de memoria es múltiplo de 2, mientras que un acceso tamaño palabra o doble palabra necesitan direcciones múltiplo de 4 y 8 respectivamente. Si se viola la restricción de alineamiento, se produce una excepción *data abort*. El mecanismo y tratamiento de excepciones queda fuera del alcance de esta asignatura introductoria. Si el alumno comete un error de este tipo tendrá que reiniciar la simulación.

Ejemplos

```
LDRB R5,  [R9]          @ Carga en el byte 0 de R5 Mem(R9).
                        @ y pone los bytes 1,2 y 3 a 0
LDRB R3,  [R8, #3]      @ Carga en el byte 0 de R3 Mem(R8 + 3).
                        @ y pone los bytes 1,2 y 3 a 0
STRB R4,  [R10, #0x200] @ Almacena el byte 0 de R4 en Mem(R10+0x200)
STRB R10, [R7, R4]      @ Almacena el byte 0 de R10 en Mem(R10+R4)
LDRH R1,  [R0]          @ Carga en los byte 0,1 de R1 Mem(R0).
                        @ y pone los bytes 2,3 a 0
LDRH R8,  [R3, #2]      @ Carga en los byte 0,1 de R8 Mem(R3+2).
                        @ y pone los bytes 2,3 a 0
STRH R2,  [R1, #0X80]   @ Almacena los bytes 0,1 de R2 en Mem(R1+0x80)
LDRD R4,  [R9]          @ Carga en R4 una palabra de Mem(R9)
                        @ Carga en R5 una palabra de Mem(R9+4)
STRD R8,  [R2, #0x28]   @ Almacena R8 en Mem(R2+0x28)
                        @ y almacena R9 en Mem(R2+0x2C)
```

4.5. Utilizando varios ficheros fuente

Hasta ahora todos los proyectos que hemos realizado tenían únicamente un fichero fuente. Sin embargo esto no es lo normal, sino que la mayor parte de los proyectos reales se componen de varios ficheros fuente. Es frecuente además que la mayor parte del proyecto se programe en un lenguaje de alto nivel como C/C++, y solamente se programen en ensamblador aquellas partes donde sea estrictamente necesario, bien por requisitos de eficiencia o bien porque necesitemos utilizar directamente algunas instrucciones especiales de la arquitectura. Para combinar código C con ensamblador nos tendremos que dividir necesariamente el programa en varios ficheros fuente, ya que los ficheros en C deben ser compilados mientras que los ficheros en ensamblador sólo deben ser ensamblados.

Cuando tenemos un proyecto con varios ficheros fuente, utilizaremos en alguno de ellos variables o subrutinas definidas en otro. Como vimos en la práctica 2, 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. El objetivo de esta sección es estudiar este mecanismo y su influencia en el código generado.

4.5.1. 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`. Veamos lo que nos dice `nm` para un fichero objeto creado para este ejemplo:

```
> 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*
```

Sin conocer el código fuente la información anterior nos dice que:

- 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 en `ejemplo.o`, pero 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).

Todas las direcciones son desplazamientos desde el comienzo de la respectiva sección.

4.5.2. Símbolos globales en C

El cuadro 1 presenta un ejemplo con dos ficheros C. El código de cada uno hace referencias a símbolos globales definidos en el otro. Los ficheros objeto correspondientes se enlazarán para formar un único ejecutable.

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. Por ejemplo, si queremos utilizar una función `FOO` que no recibe ni devuelve ningún parámetro, definida en otro fichero, debemos poner la siguiente declaración adelantada antes de su uso:

```
extern void FOO( void );
```

donde el modificador `extern` es opcional.

Con las variables globales sucede algo parecido, para utilizar una variable global definida en otro fichero tenemos que poner una especie de declaración adelantada, que indica su tipo. Por ejemplo, si queremos utilizar la variable global entera `aux` definida en otro fichero (o en el mismo pero más adelante) debemos poner la siguiente declaración antes de su uso:

```
extern int aux;
```

Si no se pone el modificador `extern`, se trata como un símbolo `COMMON`. El enlazador resuelve todos los símbolos `COMMON` del mismo nombre por la misma dirección, reservando la memoria necesaria para el mayor de ellos. Por ejemplo, si tenemos dos declaraciones de una variable global `Nombre`, una como `char Nombre[10]` y otra como `char Nombre[20]`, el enlazador tratará ambas definiciones como el mismo símbolo `COMMON`, y utilizará los 20 bytes que necesita la segunda definición.

Si se quiere restringir la visibilidad de una función o variable global al fichero donde ha sido declarada, es necesario utilizar el modificador `static` en su declaración. Esto hace que el compilador no la incluya en la tabla de símbolos del fichero objeto. De esta forma podremos tener dos o más variables globales con el mismo nombre, cada una restringida a un fichero distinto.

4.5.3. Símbolos globales en ensamblador

En ensamblador los símbolos son por defecto locales, no visibles desde otro fichero. Si queremos hacerlos globales debemos exportarlos con la directiva `.global`. El símbolo `start` es especial e indica el punto (dirección) de entrada al programa, debe siempre ser declarado global. De este modo además, se evita que pueda haber más de un símbolo `start` en varios ficheros fuente.

El caso contrario es cuando queramos hacer referencia a un símbolo definido en otro fichero. En ensamblador debemos declarar el símbolo mediante la directiva `.extern`. Por ejemplo:

```
.extern FOO     @hacemos visible un símbolo externo
.global start   @exportamos un símbolo local
```

Cuadro 1 Ejemplo de exportación de símbolos.

// fichero fun.c

```
//declaración de variable global
//definida en otro sitio
extern int var1;

//definición de var2
//sólo accesible desde func.c
static int var2;

//declaración adelantada de one
void one(void);

//definición de two
//al ser static el símbolo no se
//exporta, está restringida a este
//fichero
static void two(void)
{
   ...
   var1++;
   ...
}

void fun(void)
{
   ...
   //acceso al único var1
   var1+=5;
   //acceso a var2 de fun.c
   var2=var1+1;
   ...
   one();
   two();
   ...
}
```

// fichero main.c

```
//declaración de variable global
//definida en otro sitio (más abajo)
extern int var1;

//definición de var2
//sólo accesible desde main.c
static int var2;

//declaración adelantada de one
void one(void);

//declaración adelantada de fun
void fun(void);

int main(void)
{
   ...
   //acceso al único var1
   var1 = 1;
   ...
   one();
   fun();
   ...
}

//definición de var1
int var1;

void one(void)
{
   ...
   //acceso al único var1
   var1++;
   //acceso a var2 de main.c
   var2=var1-1;
   ...
}
```

```
start:
    bl FOO
    ...
```

4.5.4. Mezclando C y 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 sección 4.5.2.

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 esto corresponde a la dirección de comienzo de la misma. En C deberemos declarar una función con el mismo nombre que el símbolo externo. Además, deberemos declarar el tipo de todos los parámetros que recibirá la función y el valor que devuelve la misma, ya que esta información la necesita el compilador para generar correctamente el código de llamada a la función.

Por ejemplo, si queremos usar una rutina `FOO` que no devuelva nada y que tome dos parámetros de entrada enteros, deberemos emplear la siguiente declaración adelantada:

```
extern void FOO( int, int );
```

Si se trata de una variable, el símbolo corresponderá a una etiqueta que se habrá colocado justo delante de donde se haya ubicado la variable, por tanto es la dirección de la variable. Este símbolo puede importarse desde un fichero C declarando la variable como `extern`. De nuevo seremos responsables de indicar el tipo de la variable, ya que el compilador lo necesita para generar correctamente el código de acceso a la misma.

Por ejemplo, si el símbolo `var1` corresponde a una variable entera de tamaño media palabra, tendremos que poner en C la siguiente declaración adelantada:

```
extern short int var1;
```

Otro ejemplo sería el de un código ensamblador que reserve espacio en alguna sección memoria para almacenar una tabla y queramos acceder a la tabla desde un código C, ya sea para escribir o para leer. La tabla se marca en este caso con una etiqueta y se exporta la etiqueta con la directiva `.global`, por tanto el símbolo es la dirección del primer byte de la tabla. Para utilizar la tabla en C lo más conveniente es declarar un array con el mismo nombre y con el modificador `extern`.

4.5.5. Resolución de símbolos y relocalización

La Figura 4.3 ilustra el proceso de resolución de símbolos y relocalización con dos ficheros fuente: `init.asm`, codificado en ensamblador, y `main.c`, codificado en C. El primero declara un símbolo global `MIVAR`, que corresponde a una etiqueta de la sección `.data` en la que se ha reservado un espacio tamaño palabra y se ha inicializado con el valor `0x2`. En `main.c` se hace referencia a una variable externa con nombre `MIVAR`, declarada en C como entera (`int`).

Estos ficheros fuentes son primero compilados, generando respectivamente los ficheros objeto `init.o` y `main.o`. La figura muestra en la parte superior el desensamblado de `main.o`. Para obtenerlo hemos seguido la siguiente secuencia de pasos:

```
> arm-none-eabi-gcc -O0 -c -o main.o main.c
> arm-none-eabi-objdump -D main.o
```

Se han marcado en rojo las instrucciones ensamblador generadas para la traducción de la instrucción C que accede a la variable `MIVAR`, marcada también en rojo en `main.c`. Como vemos es una traducción compleja. Lo primero que hay que observar es que la operación C es equivalente a:

```
MIVAR = MIVAR + 1:
```

Entonces, para poder realizar esta operación en ensamblador tendríamos primero que cargar el valor de `MIVAR` en un registro. El problema es que el compilador no sabe cuál es la dirección de `MIVAR`, puesto que es un símbolo no definido que será resuelto en tiempo de enlazado. ¿Cómo puede entonces gcc generar un código para cargar `MIVAR` en un registro si no conoce su dirección?

La solución a este problema es la siguiente. El compilador reserva espacio al final de la sección de código (`.text`) para una *tabla de literales*. Entonces genera código como si la dirección de `MIVAR` estuviese en una posición reservada de esa tabla. En el ejemplo, la entrada de la *tabla de literales* reservada para `MIVAR` está en la posición `0x44` de la sección de código de `main.o`, marcada también en rojo. Como la distancia relativa a esta posición desde cualquier otro punto de la sección `.text` no depende del enlazado, el compilador puede cargar el contenido de la tabla de literales con un `ldr` usando `PC` como registro base.

Como vemos, la primera instrucción en rojo carga la entrada `0x44` de la sección de texto en `r3`, es decir, tendremos en `r3` la dirección de `MIVAR`. La segunda instrucción carga en `r3` el valor de la variable y la siguiente instrucción le suma 1, guardando el resultado en `r2`. Finalmente se vuelve a cargar en `r3` la dirección de `MIVAR` y la última instrucción guarda el resultado de la suma en `MIVAR`.

Pero, si nos fijamos bien en el cuadro, veremos que la entrada 0x44 de la sección `.text` está a 0, es decir, no contiene la dirección de `MIVAR`. ¿Era esto esperable? Por supuesto que sí, ya habíamos dicho que el compilador no conoce su dirección. El compilador pone esa entrada a 0 y añade al fichero objeto una entrada para `MIVAR` en la tabla de relocalización (*realloc*), como podemos ver con la herramienta objdump:

```
> arm-none-eabi-objdump -r main.o

main.o:     file format elf32-littlearm

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000040 R_ARM_V4BX        *ABS*
00000044 R_ARM_ABS32       MIVAR
```

Como podemos ver, hay una entrada de relocalización que indica al enlazador que debe escribir en la entrada `0x44` un entero sin signo de 32 bits con el valor absoluto del símbolo `MIVAR`.

La parte inferior de la Figura 4.3 muestra el desensamblado del ejecutable tras el enlazado. Como vemos, se ha ubicado la sección `.text` de main a partir de la dirección de memoria `0x0C000004`. La entrada `0x44` corresponde ahora a la dirección `0x0C000048` y contiene el valor `0x0c000000`. Podemos ver también que esa es justo la dirección que corresponde al símbolo

`MIVAR`, y contiene el valor `0x2` con el que se inicializó en `init.s`. Es decir, el enlazador ha resuelto el símbolo y lo ha escrito en la posición que le indicó el compilador, de modo que el código generado por este funcionará correctamente.

4.6. Arranque de un programa C

Un programa en lenguaje C está constituido por una o más funciones. Hay una función principal que constituye el punto de entrada al programa, llamada `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.

Sin embargo para que la función main pueda comenzar, el sistema debe haber inicializado el registro de pila (`SP`) con la dirección base de la pila. Por este motivo (y otros que quedan fuera del alcance de esta asignatura) el programa no comienza realmente en la función `main` sino con un código de arranque, que en el caso del toolchain de GNU se denomina *C Real Time 0* o *crt0*. Este código está definido en el contexto de un sistema operativo. Sin embargo, en nuestro caso estamos utilizando herramientas de compilación para un sistema *bare metal*, es decir, sin sistema operativo. En este caso, debemos proporcionar nosotros el código de arranque. El cuadro 2 presenta el código de arranque que utilizaremos de aquí en adelante.

Cuadro 2 Ejemplo de rutina de inicialización.

```
    .extern main
    .extern _stack
    .global start

start:
    ldr sp,=_stack
    mov fp,#0

    bl main
End:
    b End
.end
```

4.7. Tipos compuestos

Los lenguajes de alto nivel como C ofrecen generalmente tipos de datos compuestos, como arrays, estructuras y uniones. Vamos a ver cómo es la implementación de bajo nivel de estos tipos de datos, de forma que podamos acceder a ellos en lenguaje ensamblador.

4.7.1. Arrays

Un array es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria. En C por ejemplo, cada elemento puede ser accedido por el nombre de la variable del array seguido de uno o más subíndices encerrados entre corchetes.

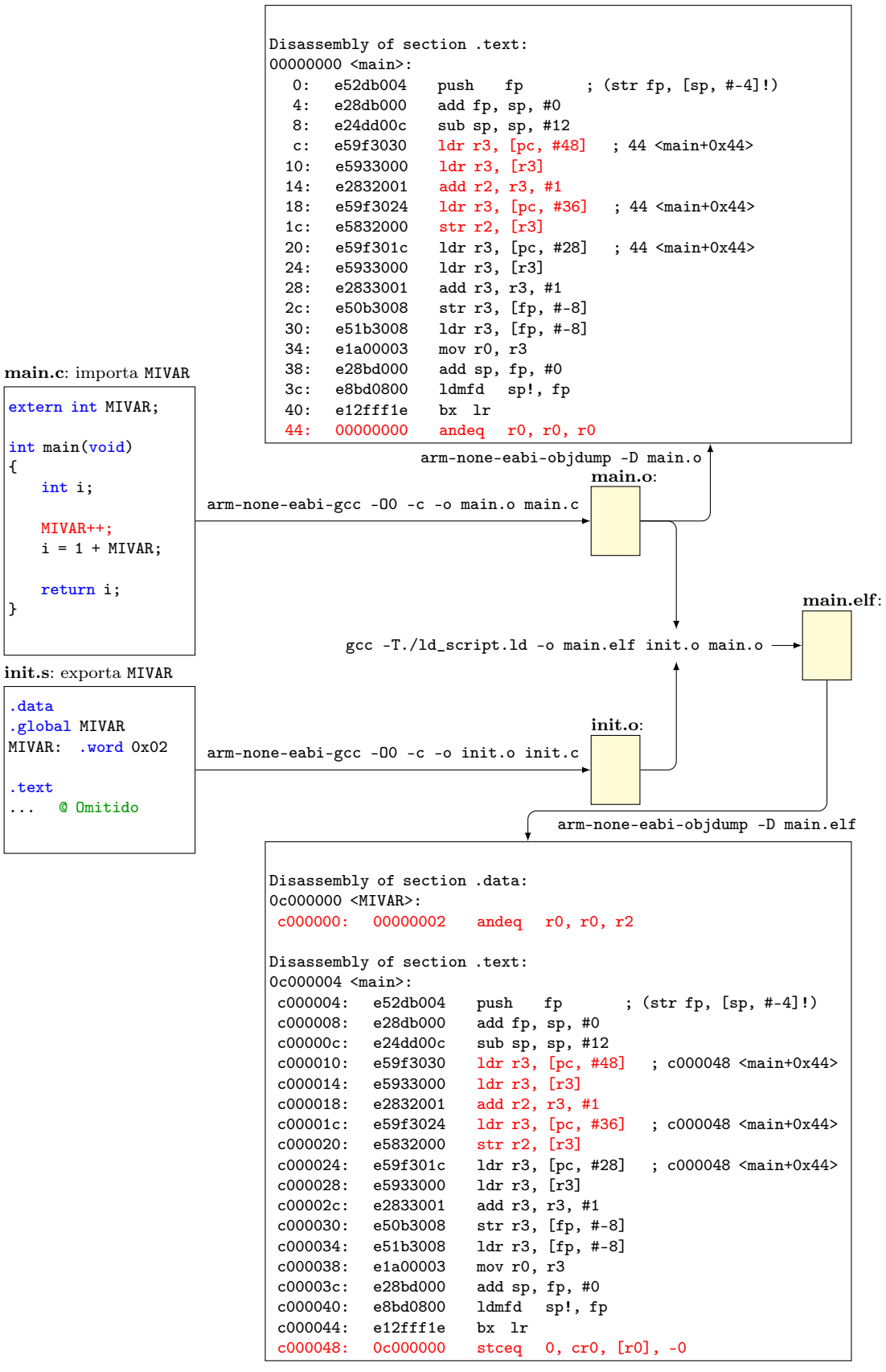


Figura 4.3: Ejemplo de resolución de símbolos.

Si declaramos una variable global cadena como:

```
char cadena[] = "hola␣mundo\n";
```

el compilador reservará doce bytes consecutivos en la sección `.data`, asignándoles los valores como indica la figura 4.4. Como vemos las cadenas en C se terminan con un byte a 0 y por eso la cadena ocupa 12 bytes. En este código el nombre del array, `cadena`, hace referencia a la dirección donde se almacena el primer elemento, en este caso la dirección del byte/caracter `h` (i.e. `0x0c0002B8`).

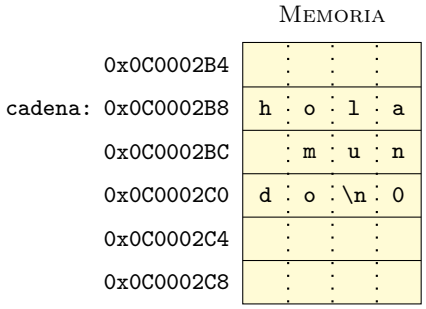


Figura 4.4: Almacenamiento de un array de caracteres en memoria.

La dirección de comienzo de un array debe ser una dirección que satisfaga las restricciones de alineamiento para el tipo de datos almacenados en el array.

4.7.2. Estructuras

Una estructura es una agrupación de variables de cualquier tipo a la que se le da un nombre. Por ejemplo el siguiente fragmento de código C:

```
struct mistruct {
    char primero;
    short int segundo;
    int tercero;
};

struct mistruct rec;
```

define un tipo de estructura de nombre `struct mistruct` y una variable `rec` de este tipo. La estructura tiene tres campos, de nombres: `primero`, `segundo` y `tercero` cada uno de un tipo distinto.

Al igual que sucedía en el caso del array, el compilador debe colocar los campos en posiciones de memoria adecuadas, de forma que los accesos a cada uno de los campos no violen las restricciones de alineamiento. Es muy probable que el compilador se vea en la necesidad de *dejar huecos* entre unos campos y otros con el fin de respetar estas restricciones. Por ejemplo, el emplazamiento en memoria de la estructura de nuestro ejemplo sería similar al ilustrado en la figura 4.5.

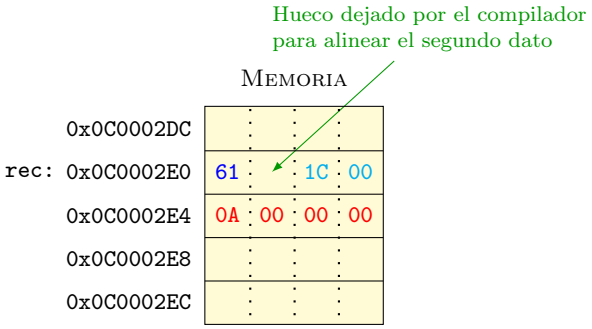


Figura 4.5: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a `0x61`, `0x1C` y `0x0A` respectivamente.

4.7.3. Uniones

Una unión es un tipo de dato que, como la estructura, tiene varios campos, potencialmente de distinto tipo, pero que sin embargo utiliza una región de memoria común para almacenarlos. Dicho de otro modo, la unión hace referencia a una región de memoria que puede ser accedida de distinta manera en función de los campos que tenga. La cantidad de memoria necesaria para almacenar la unión coincide con la del campo de mayor tamaño y se emplaza en una dirección que satisfaga las restricciones de alineamiento de todos ellos. Por ejemplo el siguiente fragmento de código C:

```
union miunion {
    char primero;
    short int segundo;
    int tercero;
};

union miunion un;
```

declara una variable `un` de tipo `union miunion` con los mismos campos que la estructura de la sección 4.7.2. Sin embargo su huella de memoria (*memory footprint*) es muy distinta, como ilustra la figura 4.6.

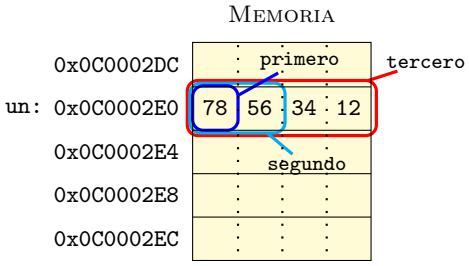


Figura 4.6: Almacenamiento de la union `un`, con los campos primero, segundo y tercero. Un acceso al campo primero da como resultado el byte `0x78`, es decir el carácter `x`. Un acceso al campo segundo da como resultado la media palabra `0x5678` (asumiendo configuración *little endian*), es decir el entero `22136`. Finalmente un acceso al campo tercero nos da como resultado la palabra `0x12345678` (de nuevo *little endian*), es decir el entero `305419896`.

Cuando se accede al campo `primero` de la unión, se accede al byte en la dirección `0x0C0002E0`, mientras que si se accede al campo `segundo` se realiza un acceso de tamaño media palabra a partir de la dirección `0x0C0002E0` y finalmente, un acceso al campo tercero implica un acceso de tamaño palabra a partir de la dirección `0x0C0002E0`.

4.8. Eclipse: depurando un programa C

Eclipse ofrece ciertas facilidades para la depuración de un programa C que conviene conocer. Como ejemplo, la Figura 4.7 ilustra una sesión de depuración de un programa C. En la parte superior derecha podemos ver el visor de variables. Para abrirlo basta con seleccionar `Widow→Show View→Variables`. Al principio sólo aparecerán las variables locales de la función que está ejecutándose en ese momento (mejor dicho, del marco de activación que tengamos seleccionado en la parte superior izquierda, donde aparece el árbol de llamadas actual). Podemos añadir las variables locales pinchando en el visor con el botón derecho del ratón y seleccionando `Add Global Variables...` del menú desplegado.

A veces resulta conveniente ver las variables en memoria, sobre todo en el caso de los arrays porque veremos varias posiciones del array cómodamente. Para esto debemos abrir el visor `Memory Monitor`, ilustrado en la parte inferior de la figura. En el ejemplo se ha creado un monitor en la dirección `0x0C000000`, y se han mostrado dos representaciones del mismo, una como enteros y otra como ASCII. Para ver las dos representaciones simultaneamente se ha pulsado el botón  . La pestaña `New Renderings` permite seleccionar distintos formatos de representación.

Finalmente, en ocasiones nos resultará conveniente el visor `Expressions`. Este visor, que se abrirá en la parte superior derecha junto al de variables, nos permite introducir cualquier expresión C válida y nos dará su valor. Por ejemplo, si introducimos la expresión `&num`, siendo `num` una variable de nuestro programa, veremos en el visor la dirección de memoria de `num`.

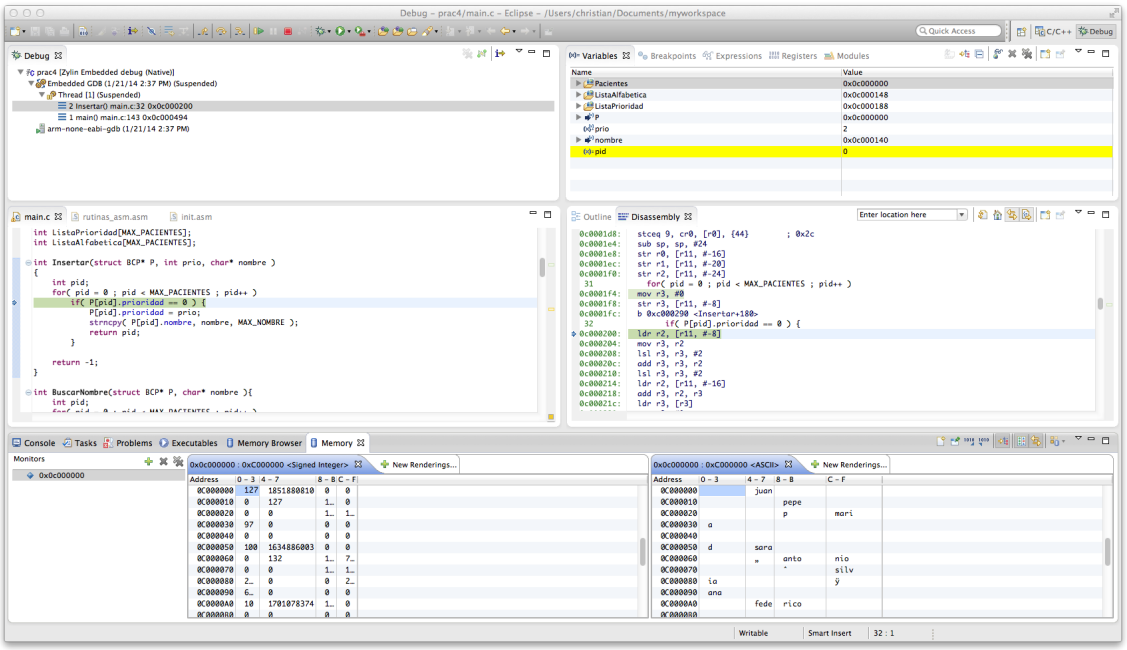


Figura 4.7: Depurando un programa C: variables y visor de memoria.

4.9. Desarrollo de la práctica

En esta práctica partiremos de un proyecto que permite ejecutar un programa C en nuestro sistema de laboratorio. Se trata de un programa que realiza la transformación de una imagen almacenada con tres canales de color (`RGB`) a una imagen en escala de grises para, posteriormente, transformarla en una imagen binaria, dónde cada punto se representa con negro o blanco.

(a) Imagen RGB (b) Imagen escala de grises (c) Imagen binaria

Figura 4.8: Ejemplo de transformación de una imagen en color a escala de grises y, finalmente, a imagen binaria.

Una imagen se puede representar como una matriz de *pixels*, en la que cada elemento de la matriz expresa el valor, en una determinada escala, de un *pixel*. En RGB, cada pixel se caracteriza por tres valores: intensidad luminosa del rojo (R), verde (G) y azul (B). Por tanto, cada elemento de una matriz que representa una imagen en color será un vector de tres elementos. Para la práctica usaremos la siguiente definición para el tipo *pixel* RGB:

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

Como vemos en la definición del tipo, cada canal de color se representa con 8 bits (`unsigned char`): podemos distinguir 256 niveles diferentes en cada canal, con un total de 24 bits por pixel (24bpp), habituales en muchos formatos de imagen actuales.

Por otro lado, para representar imágenes en escala de grises basta con un único valor (un solo canal) para indicar la luminosidad de cada *pixel*. En este caso utilizaremos 8 bits para representar cada *pixel* en una imagen en escala de grises.

Por tanto para definir dos imágenes de N filas y M columnas, una de ellas en color y la otra en escala de grises, bastaría con realizar la siguiente declaración:

```
#define N 128
#define M 128

pixelRGB imagenColor[N][M];
unsigned char imagenEscalaGrises[N][M];
```

A la hora de acceder a estas matrices en ensamblador debemos saber calcular la dirección de un elemento i, j, conocida la dirección de comienzo del array. Sabiendo que según el

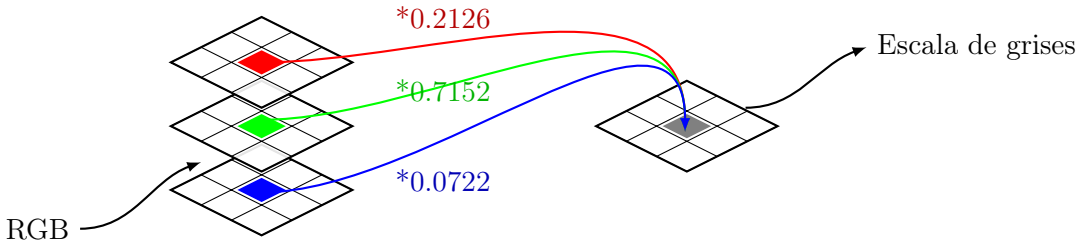


Figura 4.9: Transformación de `RGB` a escala de grises

estándar ANSI C los arrays de dos dimensiones se colocan en memoria por filas, la dirección del *pixel* de la fila i y columna j se obtendría sumando a la dirección del primer elemento $(i * M + j) * B$, dónde B es el número de bytes que ocupa cada pixel (3 en RGB y 1 en escala de grises).

4.9.1. Transformación `RGB` a escala de grises

La transformación entre ambos espacios de colores se realiza mediante una sencilla función lineal. Como se observa en la Figura 4.9 es necesario multiplicar el valor de cada canal de un pixel por una constante y, la suma de los productos será el valor en escala de grises:

```
gris = 0.2126*pixel.R + 0.7152*pixel.G + 0.0722*pixel.B;
```

Sin embargo, para poder usar esas constantes necesitaríamos poder operar con números reales, y en el modelo de procesador ARM con el que hemos estado trabajando no disponemos de representación en punto flotante, por tanto no podemos operar con números reales. Para solventarlo, haremos una aproximación usando aritmética entera, escogiendo además un divisor potencia de 2 (2^{14}) para facilitar la división (basta con desplazar a la derecha 14 bits).

```
gris =  (3483*orig.R + 11718*orig.G + 1183*orig.B) / 16384;
```

Aún nos encontramos con otra dificultad al utilizar ese código, ya que hemos decidido que representaremos, tanto los valores de cada canal de `RGB` como los de luminancia (grises) con tan solo 8 bits. Si bien el resultado de la operación anterior estará en el rango [0, 255], representable con 8 bits, las operaciones intermedias no lo están. Por ello, para hacer la conversión convertiremos cada canal a un entero de tamaño palabra, para hacer las multiplicaciones, la suma y la división. El resultado final será siempre menor o igual que 255, por lo que podremos guardar el resultado como un dato tamaño byte.

4.9.2. Transformación a imagen binaria

Existen múltiples algoritmos para *binarizar* una imagen en escala de grises, pero todos ellos coinciden en un aspecto: comparan cada *pixel* con un umbral y, si el valor del *pixel* es superior al umbral ese pixel se marcará como blanco; en caso contrario, será negro. La diferencia entre los diferentes métodos radica en el mecanismo para seleccionar el umbral, en ocasiones variable en función de la zona de la imagen que estemos procesando. En esta práctica utilizaremos un umbral fijo, con valor 127.

La matriz para representar la imagen binaria necesitaría únicamente un bit para almacenar cada *pixel*, pero dado que la mayoría de los formatos de representación estándar emplean un byte para cada pixel, en esta práctica usaremos un tipo `unsigned char` (tamaño 1 byte) para representar cada *pixel*, con valor 0 para el negro y 255 para el blanco. Por tanto, el siguiente código puede servir de base para realizar la transformación binaria con un umbral dado:

```
for (i=0;i<N;i++)
  for (j=0;j<M;j++)
    if  (imagenGris[i][j] > umbral)
        imagenBinaria[i][j]= 255;
    else
        imagenBinaria[i][j]= 0;
```

4.9.3. Estructura del programa

El código consta de varios ficheros fuente: `main.c`, `trafo.c`, `trafo.h`, `types.h`, `lena128.h`, `lena128.c`, `init.asm` y `rutinas_asm.asm`. Los ficheros `.h` contienen declaraciones adelantadas de funciones o variables, así como declaraciones de tipos y constantes. Los ficheros `.c` contienen las implementaciones de las funciones y las declaraciones de variables. El fichero `init.asm` contiene el código de inicialización que invoca `main` y el fichero `rutinas_asm.asm` contiene una implementación en ensamblador de `rgb2gray`.

El código de la función main será el siguiente:

```
int main(void) {
    // 1. Crear una matriz NxM a partir del array lena128
    initRGB(imagenRGB);

    // 2. Transformar la matriz RGB a una matriz de grises
    RGB2GrayMatrix(imagenRGB,imagenGris);

    // 3. Transformar la matriz de grises a una matriz en blanco y negro
    Gray2BinaryMatrix(imagenGris,imagenBinaria);

    // 4. Contar los blancos que aparecen por filas en imagenBinaria
    contarBlancos(imagenBinaria,blancosPorFila);

    return 0;
}
```

Todas estas funciones se encuentran declaradas en el fichero trafo.h y e implementadas en el fichero trafo.c.

4.9.4. Visualización de resultados

Una vez realizadas las transformaciones, pero antes de finalizar la ejecución (es decir, poniendo un breakpoint en la sentencia `return` de la función `main`) es posible volcar el contenido de regiones de memoria de la placa en ficheros del disco del PC del laboratorio. Para ello, usaremos la orden *dump* del depurador *GDB*:

- En la perspectiva de depuración, visualizar la ventana *Console*.

- Escribir en la consola el comando:

  `dump value` *nombreFicheroSalida matrizDeImagen*.

  Por ejemplo, para crear un fichero con la información contenida en la matriz `imagenBinaria` en el fichero `C:\hlocal\grises.dat`, deberíamos escribir:

  ```
  dump value C:\hlocal\grises.dat imagenBinaria
  ```

- Ese fichero no está en ningún formato habitual, pero es muy sencillo convertirlo. Para ello, se puede usar el programa *dump2ppm* proporcionado por el profesor. Ojo, este código NO hay que ejecutarlo en la placa, es un ejecutable de Windows que lee el fichero volcado y crea uno nuevo en formato PPM. Es un programa en línea de comandos, es decir, debemos invocarlo desde una ventana de comando de windows. Si ejecutamos el programa sin parámetros nos dará una ayuda de cómo debe usarse. Acepta los siguientes parámetros:

  - Nombre de fichero con el volcado de memoria
  - Nombre de fichero de salida
  - Número de filas de la imagen
  - Número de columnas de la imagen
  - Número de canales (1 ó 3).

  Por ejemplo, para obtener la imagen de grises a partir de su volcado la línea de órdenes sería:
  ```
  dump2ppm C:\hlocal\grises.dat C:\hlocal\grises.ppm 128 128 1
  ```

- Si se dispone de alguna herramienta para visualizar una figura en ese formato, podrá visualizarse haciendo doble clic sobre ella.

4.9.5. Trabajo sobre la práctica

1. 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. Para comprobar que funciona correctamente podemos descargar las imágenes y visualizarlas, tal y como explica la Subsección 4.9.4.

2. Obtener una función C equivalente de la subrutina rgb2gray (habrá que comentar la implementación en ensamblador de la subrutina).

3. Traducir la función RGB2GrayMatrix a ensamblador (habrá que comentar la implementación C de la función).