Práctica 1. Lenguajes de alto nivel y ensamblador

Índice general

1.1. Objetivos

En esta práctica profundizaremos en nuestros conocimientos sobre la programación en ensamblador sobre ARM aprendiendo a combinar código escrito en un lenguaje de alto nivel como C con código escrito directamente en lenguaje ensamblador. Los principales objetivos son:

- Recordar el convenio de paso de parámetros a funciones.
- Recordar los distintos ámbitos de variables, local y global.
- Comprender los tipos estructurados propios de los lenguajes de alto nivel.
- Comprender el código generado por el compilador *gcc*.

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.

1.2. Estructura de un programa en lenguaje ensamblador

Para explicar la estructura de un programa en lenguaje ensamblador y las distintas directivas del ensamblador utilizaremos como guía el sencillo programa descrito en el cuadro 1.

Cuadro 1 Ejemplo de programa en ensamblador de ARM.

```
        .global start

        .equ UNO, 0x01

        .data
DOS:    .word  0x02

        .bss
RES:    .space 4

        .text
start:
         MOV R0, #UNO
         LDR R1, =DOS
         LDR R2, [R1]
         ADD R3, R0, R2
         LDR R4, =RES
         STR R3, [R4]
FIN:     B .
        .end
```

Lo primero que podemos ver en el listado del cuadro 1 es que un programa en lenguaje ensamblador no es más que un texto estructurado en líneas con el siguiente formato:

```
etiqueta: <instrucción o directiva>    @ comentario
```

Cada uno de estos campos es opcional, es decir, podemos tener por ejemplo líneas con instrucción pero sin etiqueta ni comentario.

El fichero que define el programa comienza con una serie de órdenes (directivas de ensamblado) dedicadas a definir dónde se van a almacenar los datos, ya sean datos de entrada o de salida. A continuación aparece el código del programa escrito con instrucciones del repertorio ARM. Como el ARM es una máquina Von Neuman los datos y las instrucciones utilizan el mismo espacio de memoria. Como programa informático (aunque esté escrito en lenguaje ensamblador), los datos de entrada estarán definidos en unas direcciones de memoria, y los datos de salida se escribirán en direcciones de memoria reservadas para ese fin. De esa manera si se desean cambiar los valores de entrada al programa se tendrán que cambiar a mano los valores de entrada escritos en el fichero. Para comprobar que el programa funciona correctamente se tendrá que comprobar el valor almacenado en las posiciones de memoria reservadas para la salida una vez se haya ejecutado el programa.

Los términos utilizados en la descripción de la línea son:

- `etiqueta`: es una cadena de texto que el ensamblador relacionará con la dirección de memoria correspondiente a ese punto del programa. Si en cualquier otro punto del programa se utiliza esta cadena en un lugar donde debiese ir una dirección, el ensamblador sustituirá la etiqueta por el modo de acceso correcto a la dirección que corresponde a la etiqueta. Por ejemplo, en el programa del cuadro 1 la instrucción `LDR R1,=DOS` carga en el registro `R1` el valor de la etiqueta `DOS`, es decir, la dirección en la que hemos almacenado nuestra variable `DOS`, actuando a partir de este momento el registro R1 como si fuera un puntero. Debemos notar aquí que esta instrucción no se corresponde con ningún formato de `ldr` válido. Es una pseudo-instrucción, una facilidad que nos da el ensamblador, para poder cargar en un registro un valor inmediato o el valor de un símbolo o etiqueta. El ensamblador reemplazará esta instrucción por un `ldr` válido que cargará de memoria el valor de la etiqueta `DOS`, aunque necesitará ayuda del enlazador para conseguirlo, como veremos más adelante.

- `instrucción`: el mnemotécnico de una instrucción de la arquitectura destino (ver figura 1.2). A veces puede estar modificado por el uso de etiquetas o macros del ensamblador que faciliten la codificación. Un ejemplo es el caso descrito en el punto anterior donde la dirección de un load se indica mediante una etiqueta y es el ensamblador el que codifica esta dirección como un registro base más un desplazamiento.

- `directiva`: es una orden al propio programa ensamblador. Las directivas permiten inicializar posiciones de memoria con un valor determinado, definir símbolos que hagan más legible el programa, marcar el inicio y el fin del programa, etc (ver figura 1.1). Debemos tener siempre en cuenta que, aparte de escribir el código del algoritmo mediante instrucciones de ensamblador, el programador en lenguaje ensamblador debe reservar espacio en memoria para las variables, y en caso de que deban tener un valor inicial, escribir este valor en la dirección correspondiente. Las directivas más utilizadas son:

  - `.global`: exporta un símbolo para que pueda utilizarse desde otros ficheros, resolviéndose las direcciones en la etapa de enlazado. El comienzo del programa

se indica mediante la directiva .global start, y dicha etiqueta debe aparecer otra vez justo antes de la primera instrucción del programa, para indicar dónde se encuentra la primera instrucción que el procesador debe ejecutar.

- `.equ`: define un símbolo con un valor. De forma sencilla podemos entender un símbolo como una cadena de caracteres que será sustituida allí donde aparezca por un valor, que nosotros definimos. Por ejemplo, `.equ UNO, 0x01` define un símbolo `UNO` con valor `0x01`. Así, cuando en la línea `MOV R0, #UNO` se utiliza el símbolo, el ensamblador lo sustituirá por su valor.
- `.word`: se suele utilizar para inicializar las variables de entrada al programa. Inicializa la posición de memoria actual con el valor indicado tamaño palabra (también podría utilizarse .byte). Por ejemplo, en el programa del cuadro 1 la línea `DOS: .word 0x02` inicializa la posición de memoria con el valor `0x02`, dónde `0x` indica hexadecimal.
- `.space`: reserva espacio en memoria tamaño byte para guardar las variables de salida, si éstas no se corresponden con las variables de entrada. Siempre es necesario indicar el espacio que se quiere reservar. Por ejemplo, en el programa del cuadro 1 la línea `RES: .space 4` reserva cuatro bytes (una palabra (word)) que quedan sin inicializar. La etiqueta `RES` podrá utilizarse en el resto del programa para referirse a la dirección correspondiente a esta palabra.
- `.end`: Finalmente, el ensamblador dará por concluido el programa cuando encuentre la directiva `.end`. El texto situado detrás de esta directiva de ensamblado será ignorado.

- Secciones. Normalmente el programa se estructura en secciones, generalmente `.text`, `.data` y `.bss`. Para definir estas secciones simplemente tenemos que poner una línea con el nombre de la sección. A partir de ese momento el ensamblador considerará que debe colocar el contenido subsiguiente en la sección con dicho nombre.
  - `.bss`: es la sección en la que se reserva espacio para almacenar el resultado.
  - `.data`: es la sección que se utiliza para declarar las variables con valor inicial
  - `.text`: contiene el código del programa.
- `comentario`: una cadena de texto para comentar el código. Con `@` se comenta hasta el final de la línea actual. Pueden escribirse comentarios de múltiples líneas como comentarios C (entre `/*` y `*/`).

Las líneas que contienen directamente una instrucción serán codificadas correctamente como una instrucción de la arquitectura objetivo, ocupando así una palabra de memoria (4 bytes). Cuando se utiliza una *facilidad* del ensamblador, éste puede sustituirla por más de una instrucción, ocupando varias palabras. Si la línea es una directiva que implica reserva de memoria el ensamblador reservará esta memoria y colocará después la traducción de las lineas subsiguientes.

Los pasos seguidos por el entorno de compilación sobre el código presentado en el Cuadro 1 se resumen en la siguiente tabla:

En la Tabla 1.1 el código de la primera columna se corresponde con el código en lenguaje ensamblador tal y como lo programamos. Utilizamos etiquetas para facilitarnos

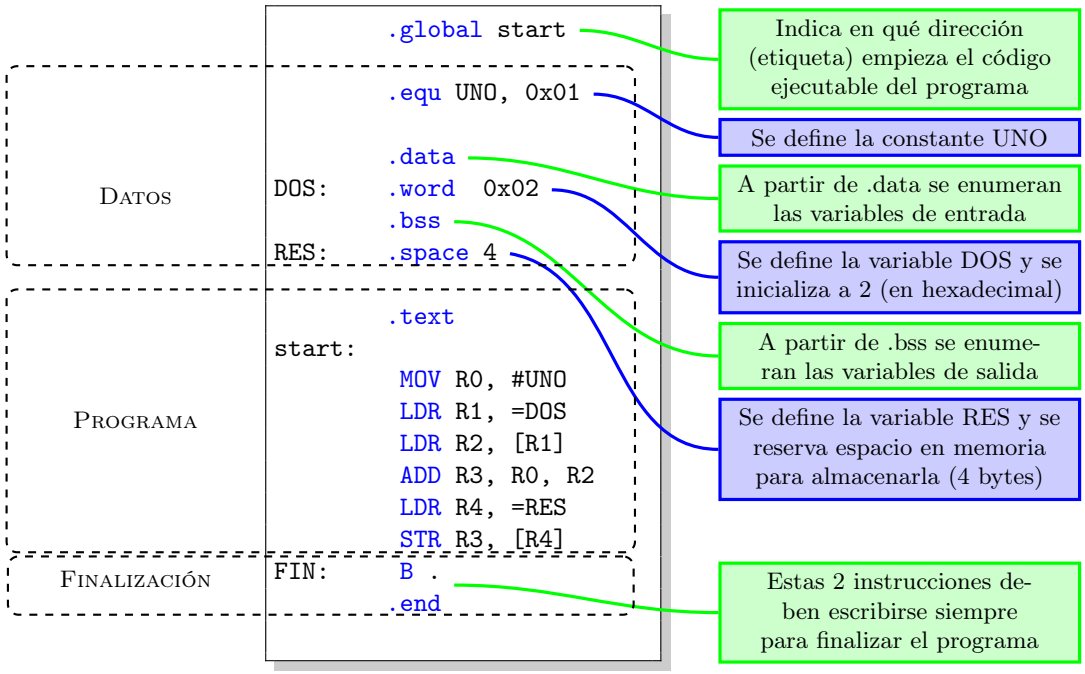


Figura 1.1: Descripción de la estructura de un programa en ensamblador: datos.

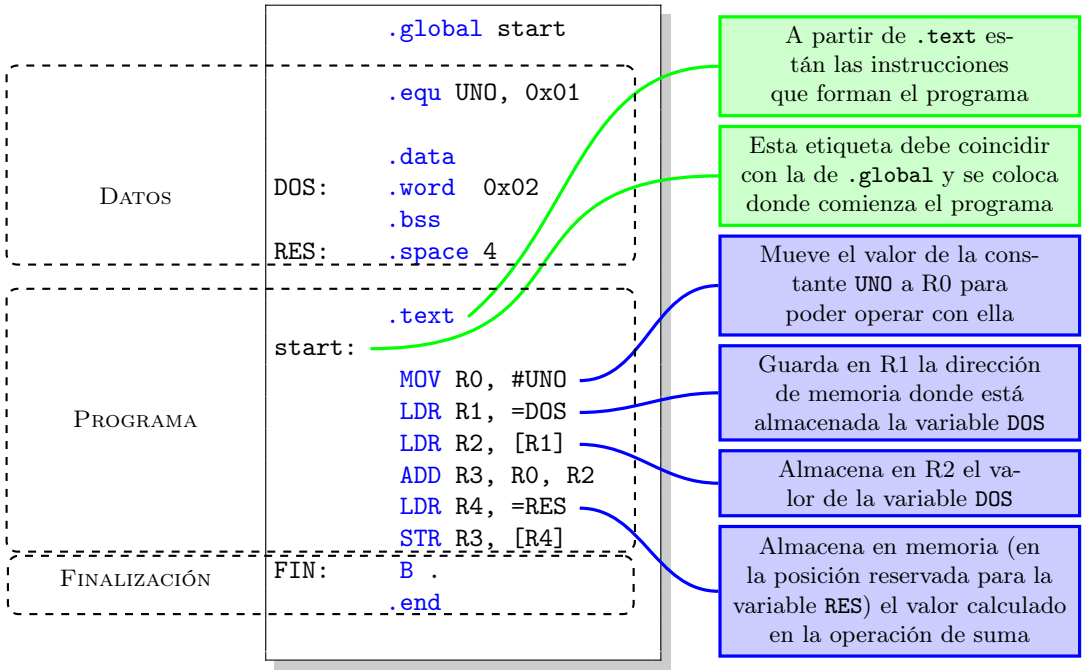


Figura 1.2: Descripción de la estructura de un programa en ensamblador: instrucciones.

Tabla 1.1: Traducción de código ensamblador a binario

Código Ensamblador	Código objeto desensamblado	Codificación (Hexadecimal)
`start:`		
`MOV R0, #UNO`	`MOV r0, #1`	`e3a00001`
`LDR R1, =DOS`	`LDR r1, [pc, #16]`	`e59f1010`
`LDR R2, [R1]`	`LDR r2, [r1]`	`e5912000`
`ADD R3, R0, R2`	`ADD r3, r0, r2`	`e0803002`
`LDR R4, =RES`	`LDR r4, [pc, #8]`	`e59f4008`
`STR R3, [R4]`	`STR r3, [r4]`	`e5843000`
`FIN: B .`		

una escritura rápida del algoritmo sin tener que realizar cálculos relativos a la posición de las variables en memoria para realizar su carga en el registro asignado a dicha variable. En la columna del centro aparece el código ensamblador generado por el programa ensamblador al procesar el código de la izquierda. Vemos que ya todas las instrucciones tienen la forma correcta del repertorio ARM. Las etiquetas utilizadas en el código anterior se han traducido por un valor relativo al contador de programa (el dato se encuentra X posiciones por encima o por debajo de la instrucción actual). Una vez obtenido este código desambiguado ya se puede realizar una traducción directa a ceros y unos, que es el único lenguaje que entiende el procesador, esa traducción está presente en la columna de la derecha, donde por ejemplo los bits más significativos se corresponden con el código de condición, seguidos del código de operación de cada una de las instrucciones.

1.2.1. Estructuras de control de flujo en ensamblador

A modo de recordatorio de todo lo trabajado en la asignatura de Fundamentos de Computadores, la tabla 1.2 presenta algunos ejemplos de programación de estructuras de control de flujo habituales, que pueden tomarse como punto de partida para las implementaciones propuestas.

Para consultas sobre el ensamblador de ARM, se aconseja consultar los documentos [ARMb] y [GNUa].

1.3. Generación de un ejecutable

La figura 1.3 ilustra el proceso de generación de un ejecutable a partir de uno o más ficheros fuente (en lenguaje C y/o ensamblador) y de algunas bibliotecas. Cada fichero en código C pasa por el preprocesador que modifica el código fuente original siguiendo directivas de preprocesado (`#define`, `#include`, etc.). El código resultante es entonces compilado, transformándolo en código ensamblador. El ensamblador genera entonces el código objeto para la arquitectura destino (ARM en nuestro caso). Si partimos de código ensamblador, sólo sería necesaria esta última etapa para generar el código objeto.

Tabla 1.2: Ejemplos de implementaciones de algunas estructuras de control habituales.

Pseudocódigo	Ensamblador

`if (R1 > R2)`
`  R3 = R4 + R5;`
`else`
`  R3 = R4 - R5`
`sigue la ejecución normal`
	`  CMP R1, R2`
	`  BLE else`
	`  ADD R3, R4, R5`
	`  B fin_if`
	`else:  SUB R3, R4, R5`
	`fin_if: sigue la ejecución normal`

`for (i=0; i<8; i++) {`
`  R3 = R1 + R2;`
`}`
`sigue la ejecución normal`
	`  MOV R0, #0  @R0 actúa como índice i`
	`for:  CMP R0, #8`
	`  BGE fin_for`
	`  ADD R3, R1, R2`
	`  ADD R0, R0, #1`
	`  B for`
	`fin_for: sigue la ejecución normal`

`i=0;`
`do {`
`  R3 = R1 + R2;`
`  i = i + 1;`
`} while( i != 8 )`
`sigue la ejecución normal`
	`  MOV R0, #0  @R0 actúa como índice i`
	`do:  ADD R3, R1, R2`
	`  ADD R0, R0, #1`
	`  CMP R0, #8`
	`  BNE do`
	`sigue la ejecución normal`

`while (R1 < R2)`
`{`
`  R2= R2-R3;`
`}`
`sigue la ejecución normal`
	`while:  CMP R1, R2`
	`  BGE fin_w`
	`  SUB R2, R2, R3`
	`  B while`
	`fin_w: sigue la ejecución normal`

Los ficheros de código objeto son ficheros binarios con cierta estructura. En particular debemos saber que estos ficheros están organizados en secciones con nombre. Como ya hemos indicado, suelen definirse, al menos, tres secciones: `.text` para el código, `.data` para datos (variables globales) con valor inicial y `.bss` para datos no inicializados.

Los ficheros objeto no son todavía ejecutables. El ensamblador ha generado el contenido de cada una de las secciones, pero falta decidir en qué direcciones de memoria se van a colocar dichas secciones y resolver los símbolos. Por ejemplo el valor de cada etiqueta no se conoce hasta que no se decide en qué dirección de memoria se va a colocar la sección en la que aparece.

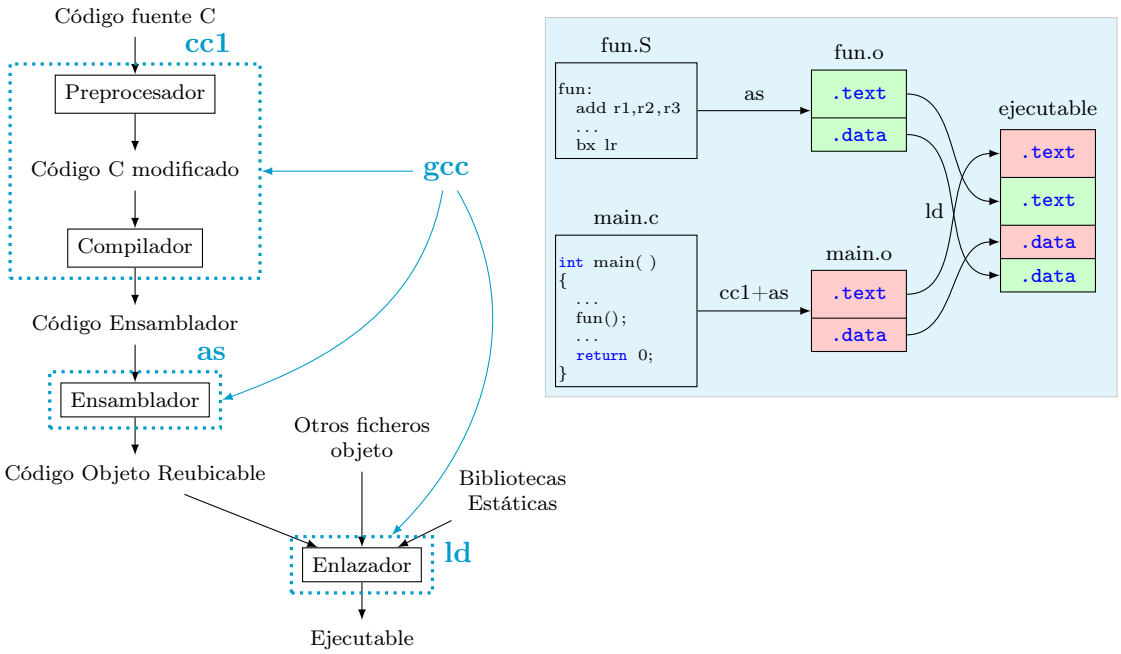


Figura 1.3: Proceso de generación de código ejecutable

Podríamos preguntarnos por qué no decide el ensamblador la dirección de cada sección y genera directamente el ejecutable final. La respuesta es que nuestro programa se implementará generalmente con más de un fichero fuente, pero las etapas de compilación y ensamblado se hacen fichero a fichero para reducir la complejidad del problema. Es más, utilizaremos frecuentemente bibliotecas de las que ni siquiera tendremos el código fuente. Por lo tanto, es necesario añadir una etapa de enlazado para componer el ejecutable final. El programa que la realiza se conoce como enlazador.

Como ilustra la Figura 1.3, el enlazador ([GNUb]) tomará los ficheros de código objeto procedentes de la compilación de nuestros ficheros fuente y de las bibliotecas externas con las que enlacemos, y reubicará sus secciones en distintas direcciones de memoria para formar las secciones del ejecutable final. En este proceso todos los símbolos habrán quedado resueltos, y como consecuencia todas nuestras etiquetas tendrán asignada una dirección definitiva. En esta etapa el usuario puede indicar al enlazador cómo colocar cada una de las secciones utilizando un *script* de enlazado.

1.4. Recordatorio: la pila de llamadas

Como hemos visto con anterioridad, el mapa de memoria de un proceso se divide en secciones de distinto propósito, generalmente, código (*text*), datos con valor inicial (*data*) y datos sin valor inicial (*bss*). El resto de la memoria puede utilizarse de distintas maneras. Una zona de memoria de especial importancia es la denominada *pila de llamadas* (*call stack* o simplemente *stack*).

La Pila es una región continua de memoria, cuyos accesos siguen una política LIFO (*Last-In-First-Out*), que sirve para almacenar información relativa a las rutinas activas del

programa: da soporte a la implementación de funciones y procedimientos. La región de la pila utilizada por una rutina en su ejecución se conoce como el Marco de Activación o Marco de Pila de la subrutina (*Activation Record* o *Stack Frame* en inglés).

La gestión de tipo LIFO se lleva a cabo mediante un puntero al último elemento (cima de la pila) que recibe el nombre de *stack pointer* (SP) y habitualmente es almacenado en un registro arquitectónico que es inicializado a la base de la región de pila cuando comienza el programa (pila vacía). Para facilitar el acceso a la información contenida en el marco, es habitual utilizar un puntero denominado *frame pointer* (FP) que apunta a una posición preestablecida del marco, generalmente la base.

La región de la pila utilizada por una rutina en su ejecución se conoce como el marco de activación o marco de pila de la rutina. En general el marco de pila de una rutina contiene:

- información sobre el estado de ejecución de la rutina (variables locales)
- información para restaurar el estado de la rutina que la invocó (copia de registros)
- parámetros que tenga que pasar a otra subrutina que invoque.

1.4.1. ARM Architecture Procedure Call Standard (AAPCS)

Como ya se estudió en cursos anteriores, el *ARM Architecture Procedure Call Standard* (AAPCS) es el estándar vigente que regula las llamadas a subrutinas en la arquitectura ARM [ARMa]. Especifica una serie de normas para que las rutinas puedan ser compiladas o ensambladas por separado, y que a pesar de ello puedan interactuar.

El estándar impone un modelo de pila *full-descending* que se caracteriza por:

- `SP` se inicializa con una dirección de memoria *alta* y crece hacia direcciones más bajas.
- `SP` apunta siempre a la última posición **ocupada**.

Además, se impone como restricción que el `SP` debe tener siempre una dirección múltiplo de 4.

Aunque el estándar **no impone ninguna restricción más a la hora de formar el marco de pila**, en esta asignatura seguiremos las siguientes pautas:

- Si nuestra subrutina invoca a otras subrutinas, tendremos que almacenar `LR` en la pila, ya que contiene la dirección de retorno pero lo tendremos que usar para pasar a su vez la dirección de retorno a las subrutinas invocadas. Esta acción (apilar `LR`) se llevará en el prólogo de la función.
- Vamos a usar siempre *frame pointer* (`FP`) para facilitar la depuración y el acceso al marco de activación. Por lo tanto, **tendremos que almacenar siempre el valor anterior de `FP` en la pila** (aunque el estándar no lo exige).
- Aunque `SP` es un registro que hay que preservar, **no será necesario incluirlo en el marco**, ya que en el epílogo podremos calcular el valor anterior de `SP` a partir del valor de `FP`.

- De los registros `r4-r10` insertaremos **sólo aquellos que modifiquemos en el cuerpo** de la subrutina: **no consideraremos válido apilar siempre todos los registros.**

La Figura 1.4 ilustra el marco de activación resultante de aplicar estas directrices a una subrutina que no es hoja, que es el que genera el compilador gcc-4.7 con depuración activada y sin optimizaciones[1]. Una de las ventajas de utilizar `FP`, aparte de la facilidad de codificación y depuración, es que se forma en la pila una lista enlazada de marcos de activación. Como ilustra la Figura 1.4, al salvar `FP` en el marco de una subrutina, estamos almacenando la dirección de la base del marco de activación de la subrutina que la invocó. Podemos así recorrer el árbol de llamadas a subrutina hasta llegar a la primera. Para identificar esta primera subrutina el programa principal pone `FP` a 0 antes de invocarla.

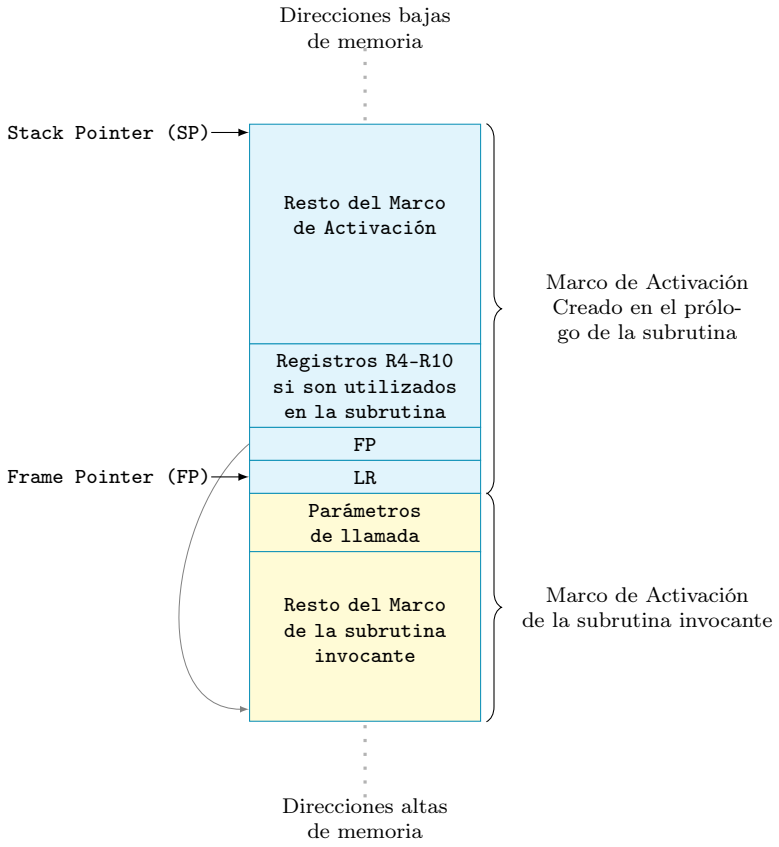


Figura 1.4: Estructura de Marco de Activación recomendada. En el caso de subrutinas hoja podremos omitir la copia de `LR`.

Por último, la tabla 1.3 recuerda los usos que se da a cada uno de los registros. El **estándar AAPCS impone el uso de los registros r0-r3 para el paso de los primeros cuatro argumentos** (si la rutina recibe más de cuatro argumentos, del quinto en adelante se apilarán**antes de realizar la llamada a la subrutina**. Esto es, se encargará

[1]Con los flags -g -O0

de hacerlo la rutina *invocante*). El AAPCS también especifica que el **resultado de una subrutina debe dejarse en el registro r0** [2].

Tabla 1.3: Uso habitual de los registros arquitectónicos

Registros	Alias	Deben ser Preservados	Uso según AAPCS
r0-r3	a1-a4	NO	Se utilizan para pasar parámetros a la rutina y obtener el valor de retorno.
r4-r10	v1-v7	SÍ	Se utilizan normalmente para almacenar variables debido a que deben ser preservados en llamadas a subrutinas.
r11	fp, v8	SÍ	Es el registro que debe utilizarse como *frame pointer*.
r12	ip	NO	Puede usarse como registro auxiliar en prólogos.
r13	sp	SÍ	Es el registro que debe utilizarse como *stack pointer*.
r14	lr	NO	Es el registro que debe utilizarse como *link register*, es decir, el utilizado para pasar la dirección de retorno en una llamada a subrutina.

Por tanto, en el marco de activación de una rutina concreta, podemos encontrar:

- El valor del registro `LR` al comenzar la rutina (sólo necesario en rutinas no-hoja)
- El valor del registro `FP` al comenzar la rutina (obligatorio)
- El valor de los registros r4-r10 si se van a utilizar en la rutina (sólo de aquellos que se modifiquen)
- Espacio para albergar las variables locales de la rutina. Dado que los argumentos de entrada se consideran variables locales, es conveniente escribir en la pila los valores de los argumentos pasados en los registros `r0-r3`. Esto ofrece más información de depuración en caso de errores imprevistos en ejecución.

1.4.2. Prólogo y epílogo

Tal y como se estudió en cursos pasados, el prólogo y epílogo de una rutina pueden implementarse con instrucciones de *load* y *store* múltiples. Para el caso concreto de la pila que implementaremos, existen dos instrucciones aún más sencillas de recordar: `push` y `pop`. El cuadro 2 describe un posible prólogo para la construcción del marco de activación representado en la Figura 1.4. La primera instrucción apila los registros a preservar sobre el marco de la rutina invocante, dejando `SP` apuntando a la nueva cima. Debemos incidir en que no es necesario apilar siempre todos los registros `R4-R10`, sólo aquellos que se vayan a modificar en la rutina. La siguiente instrucción deja el `FP` apuntando a la base del nuevo marco de activación. Para este caso concreto deberíamos sumar el valor `4*9 -4 = 32` a `SP` para obtener el valor de `FP`. La última instrucción reserva nuevo espacio en el marco: el necesario para las variables locales. Como decíamos con anterioridad, suele ser habitual guardar en la pila el valor actual de todos los argumentos recibidos como parámetros. Esto

[2] Hay casos especiales en los que se debe usar más de un registro para devolver el resultado de una subrutina, pero no los usaremos este curso

supondría escribir en la pila el contenido de los registros `r0-r3` que la rutina reciba como argumento. Ya que los valores de `FP` y `SP` ya han sido definidos en el prólogo, cualquier escritura en la pila posterior se realizará con una instrucción de `str` con dirección relativa a `FP` o `SP` (ver el ejemplo de la rutina `Mayor` a continuación, en la que se escriben en la pila los valores actuales de `X` e `Y`).

Cuadro 2 Prólogo para insertar en la pila el Marco de Activación de la Figura 1.4.

```
PUSH {R4-R10,FP,LR}                    @ Copiar registros en la pila
ADD  FP, SP, #(4*NumRegistrosApilados-4) @ FP ← dirección base del marco
SUB  SP, SP, #4*NumPalabrasExtra       @ Espacio extra necesario
```

El cuadro 3 presenta el correspondiente epílogo, que restaura la pila dejándola tal y como estaba antes de la ejecución del prólogo. Lo primero que hace es asignar a `SP` el valor que tenía tras la ejecución de la primera instrucción del prólogo. Esto lo consigue realizando la operación contraria a la segunda instrucción del prólogo. Con la segunda instrucción del epílogo se desapilan los registros apilados en el prólogo, dejando así la pila en el mismo estado que estaba antes de la ejecución del prólogo (incluyendo el valor de `SP`). En `R4-R10`, `FP` y `LR` se copian los valores que tenían al entrar en la subrutina, por lo que al terminar la ejecución del epílogo se restaura por completo el estado de la rutina invocante. Lo único que queda es hacer el retorno de subrutina, que es lo que hace la última instrucción del epílogo.

Cuadro 3 Epílogo para extraer de la pila el Marco de Activación de la Figura 1.4.

```
SUB  SP, FP, #(4*NumRegistrosApilados-4) @ SP ← dirección del 1er registro apilado
POP  {R4-R10,FP,LR}                    @ Desapila restaurando los registros
BX   LR                                @ Retorno de subrutina
```

1.4.3. Invocando una subrutina

Como se recordará, la instrucción más adecuada para invocar una subrutina es *BL*, ya que realiza un salto a la dirección indicada y guarda en el registro *LR* la dirección de la siguiente instrucción, de modo que, tras ejecutar la subrutina, se continúe la ejecución en el punto que se dejó.

Por otra parte conviene recordar que, si una rutina requiere más de 4 argumentos, **a partir del quinto se pasarán por la pila, y será la rutina invocante la encargada de apilar dichos argumentos antes de efectuar la llamada**. La figura 1.5 ilustra un ejemplo en el que una subrutina `SubA` debe invocar otra subrutina `SubB`, pasándole 7 parámetros que llamamos `Param1-7`. Como podemos ver, los cuatro primeros parámetros se pasan por registro (`r0-r3`), mientras que los últimos tres se pasan por pila, en la cima del marco de `SubA`. Si la subrutina `SubB` devolviese algún argumento, éste se encontraría en el registro `r0` tras la ejecución de la instrucción *bl*. Además, tras la llamada a la subrutina `SubB` debemos dejar el registro `SP` al valor anterior al proceso de llamada. Como debimos apilar tres registros (con los argumentos 5, 6 y 7) modificamos el valor `SP` antes de la instrucción y ahora debemos volverlo a dejar como estaba. Una opción (la usada en el ejemplo) es usar `pop`, aunque en realidad es poco probable que necesitemos restaurar el contenido de `r5-r7`.

Cuestión Propón otra alternativa, diferente a la instrucción `POP`, para restaurar el valor correcto de `SP` tras la llamada a subrutina; dicha alternativa no debe realizar ningún acceso a memoria (`POP` es realidad un *load* múltiple, por lo que implica accesos a memoria).

Cuestión Si la rutina invocante tiene valores que desea preservar en alguno de los registros r0-r3, ¿qué debe hacer para conseguirlo?. Completa el ejemplo de la figura 1.5 asumiendo que la rutina invocante desea preservar el contenido del registro `r2` tras la llamada a `subB`.

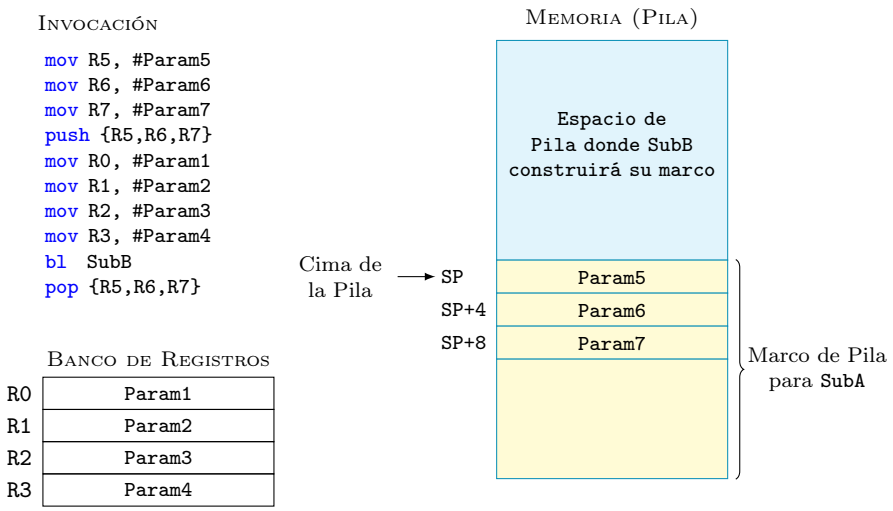


Figura 1.5: Paso de los parámetros `Param1-7` desde `SubA` a `SubB`. El código de invocación asume que los parámetros son constantes (valores inmediatos), en caso de ser variables habría que cargarlos con instrucciones `ldr` en lugar de `mov`.

1.4.4. Ejemplo: código completo de una subrutina

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

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

Como vemos, la función tiene tres variable locales (X, Y y m)[3] , 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       add fp, sp,#0
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      sub sp,fp,#0
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. **En el desarrollo de las prácticas no es necesario realizar una traducción tan literal**. En el código de este ejemplo pueden llamar la atención varios aspectos:

- Los argumentos de entrada, pasados por los registros `r0` y `r1` como exige el estándar AAPCS, se escriben en la pila tras el prólogo (líneas 5 y 6). Como hemos repetido anteriormente, esto lo exige la semántica de una variable local (y un argumento de entrada lo es): debe estar almacenado en la pila. De ese modo, si la aplicación termina abruptamente y se realiza un volcado de memoria, podemos comprobar el valor de las variables locales.

- Inmediatamente después de escribirlos en la pila, esos mismos valores se vuelven a leer (líneas 9,10, 14 y 18 y 22). Estas lecturas sí son prescindibles. Sólo son necesarios para la depuración del código C original, pues permite ejecutar cada sentencia C de forma explícita. En la siguiente sección hablaremos más sobre este aspecto del código.

[3] De acuerdo al estándar AAPCS, los dos argumentos de entrada se pasarán por los registros *r0* y *r1*; aún así las variables X e Y son consideradas locales y, por tanto, susceptibles de almacenarse en la pila

Ejercicio Propón una versión optimizada de código anterior evitando accesos innecesarios a memoria.

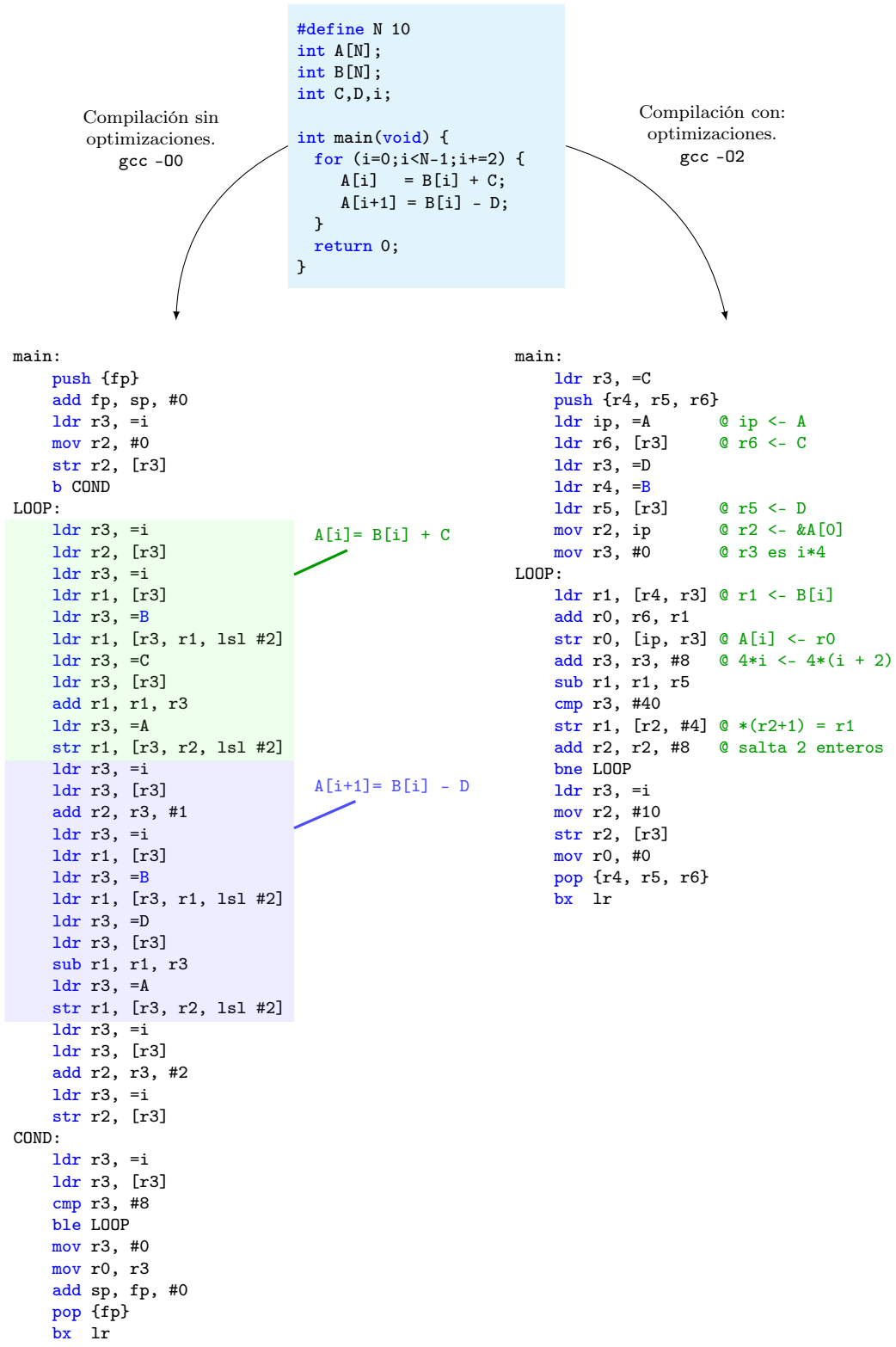
1.5. Pasando de C a Ensamblador

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 1.6 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 depurador, 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 1.6, 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




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

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.

Cuestión ¿Cuál es el nivel máximo de optimización en el compilador *gcc*? ¿Qué beneficios podremos observar al aplicar optimizaciones a nivel del compilador?

1.6. Accesos a memoria: tamaño y alineamiento

Los lenguajes de programación suelen ofrecer tipos básicos de varios 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 el modo de direccionamiento de las instrucciones `ldr`/`str` permite seleccionar 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 añaden sufijos a las instrucciones `ldr`/`str` normales, dando lugar a las siguientes variantes:

- `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.
- `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 todos los casos se soportan los modos de direccionamiento indirecto de registro, indirecto de registro con desplazamiento inmediato e indirecto de registro con desplazamiento en registro que se estudiaron el curso pasado.

Sin embargo, la arquitectura ARMv4 impone restricciones de alineamiento en los accesos a memoria. Se dice que un acceso está alineado si la dirección de memoria a la 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*.

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 R7 en Mem(R10+R4)
LDRH R1,  [R0]          @ Carga en los bytes 0,1 de R1 Mem(R0).
                        @ y pone los bytes 2,3 a 0
LDRH R8,  [R3, #2]      @ Carga en los bytes 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)
```

1.7. Utilizando varios ficheros fuente

La mayor parte de los proyectos reales se componen de varios ficheros fuente. Además, es frecuente 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 resulta conveniente dividir 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 anteriormente, 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.

1.7.1. 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 0x54 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.

1.7.2. Símbolos globales en C

El cuadro 4 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 );
```

Cuadro 4 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;
   ...
}
```

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 reservará 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.

1.7.3. 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

start:
    bl FOO
    ...
```

1.7.4. 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 1.7.2.

Supongamos que queremos usar en C una rutina implementada en ensamblador. El identificador usado como nombre de una rutina se corresponde a la dirección de comienzo de la misma. Para poder usarla en C, el símbolo (nombre de la rutina) debe haberse declarado como externo y deberemos declarar una función con el mismo nombre que dicho símbolo. 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[4].

Por ejemplo, si queremos usar una rutina `FOO` , implementada en ensamblador, que no devuelve nada y que toma 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 de 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`.

1.7.5. Resolución de símbolos

La Figura 1.7 ilustra el proceso de resolución de símbolos con dos ficheros fuente: `init.S`, 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

[4]En ensamblador no existe el concepto de tipo ni de declaración de una función. Pero en C, como en cualquier lenguaje tipado, es imprescindible

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. ¿Cabía esperar esto? 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 1.7 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.

1.8. 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

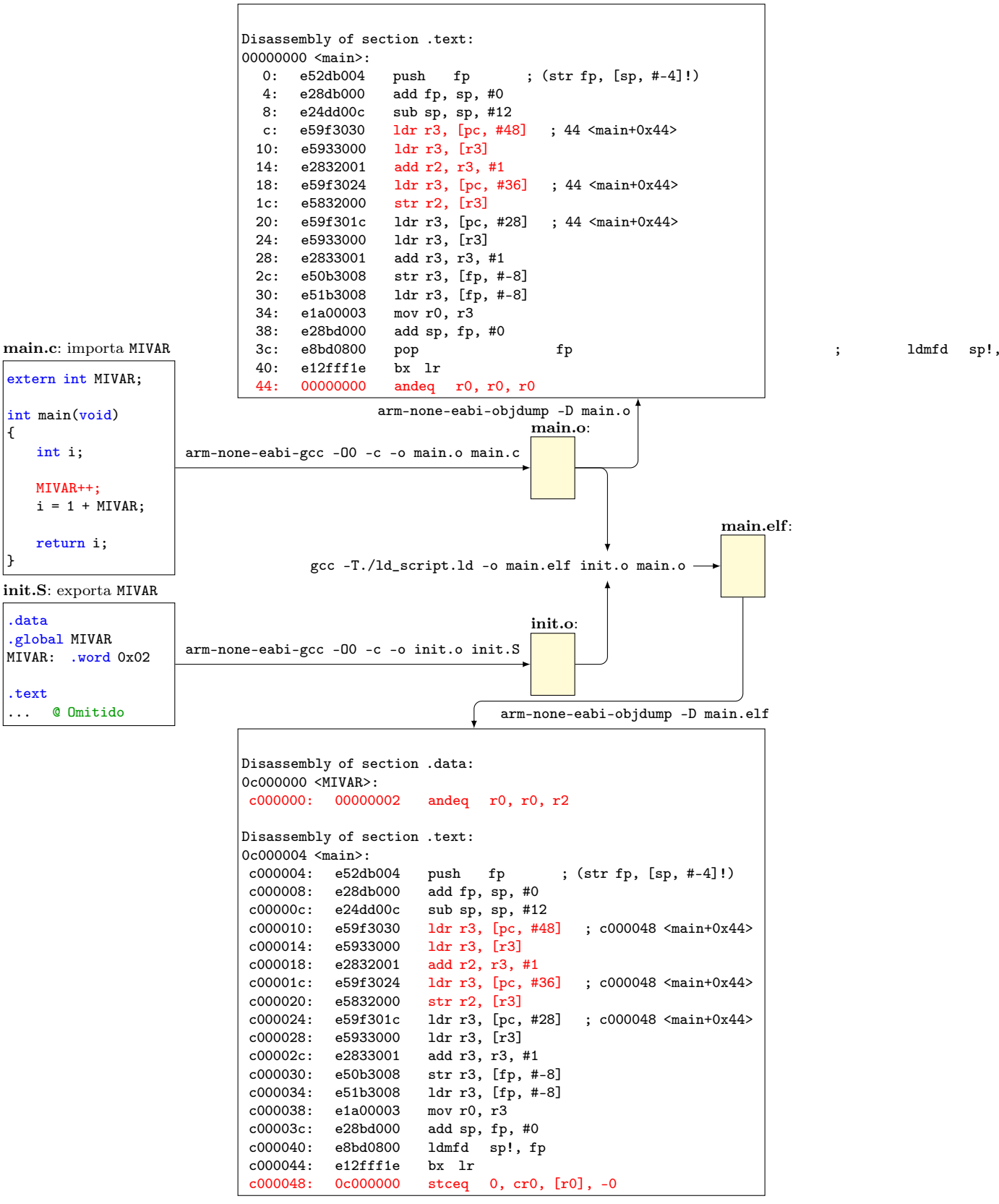


Figura 1.7: Ejemplo de resolución de símbolos.

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 5 presenta el código de arranque que utilizaremos de aquí en adelante.

Cuadro 5 Ejemplo de rutina de inicialización.

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

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

    bl main
End:
    b End
.end
```

1.9. 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.

1.9.1. 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.

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 1.8. 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`).

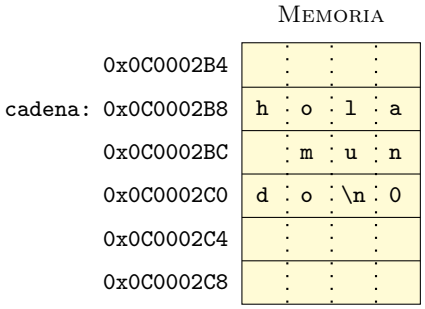


Figura 1.8: Almacenamiento de un array de caracteres en memoria.

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.

1.9.2. 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 1.9.

Cuestión ¿Cómo sería la disposición en memoria de un array cuyos elementos sean de tipo *struct*? Concretamente, imagine un tipo *struct* con tres campos tipo *unsigned char*. ¿Cuánto ocupa un array de 100 elementos de ese tipo? Si la dirección del primer elemento es 0x0C00, ¿cuál es la dirección del elemento 20?

1.9.3. 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

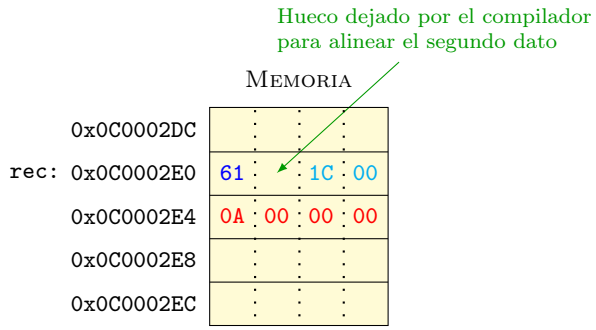


Figura 1.9: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a `0x61`, `0x1C` y `0x0A` respectivamente.

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 1.9.2. Sin embargo su huella de memoria (*memory footprint*) es muy distinta, como ilustra la figura 1.10.

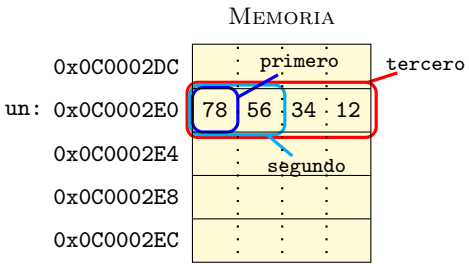


Figura 1.10: Almacenamiento de la union `un`, con los campos primero, segundo y tercero. Un acceso al campo primero da como resultado el byte `0x78`, es decir el carácter `x`. Un acceso al campo segundo da como resultado la media palabra `0x5678` (asumiendo configuración *little endian*), es decir el 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`.

1.10. Desarrollo de la práctica

El objetivo de la práctica será codificar la transformación de una imagen almacenada con tres canales de color (RGB) a una imagen en escala de grises para, posteriormente, transformarla una imagen binaria. La Figura 1.11 muestra el resultado de este proceso aplicado a una imagen RGB.

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

Figura 1.11: 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: cantidad de 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. En concreto, para la presente 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 8bits: 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 para indicar la luminosidad de cada *pixel*. En esta práctica, utilizaremos nuevamente 8 bits para representar cada *pixel* en una imagen en escala de grises.

Por tanto, para definir dos imágenes de 128 filas y 64 columnas de pixels, una de ellas en color y la otra en escala de grises, bastaría con realizar la siguiente declaración:

```
#define NFILAS 128
#define NCOLS 64

pixelRGB imagenColor[NFILAS*NCOLS];
unsigned char imagenEscalaGrises[NFILAS*NCOLS];
```

Si bien podríamos haber usado un tipo diferente (`pixelRGB imagenColor[128][64]`), hemos optado por una definición más sencilla que simplifica la comprensión del código generado. De este modo, y asumiendo que organizamos nuestras matrices por filas en memoria, para acceder al *pixel* de la fila 13, columna 24 de nuestras dos imágenes bastará con indexar como en este ejemplo:

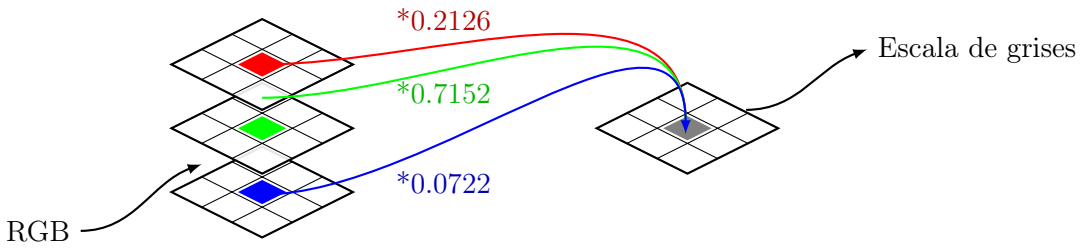


Figura 1.12: Transformación de RGB a escala de grises

```
imagenEscalaGrises[13*NCOLS + 24] = ... imagenColor[13*NCOLS + 24].R .... ;
```

1.10.1. 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 1.12 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:

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

Sin embargo, para poder usar esas constantes necesitaríamos poder operar con números reales. Pero en nuestro procesador ARM **no disponemos de representación en punto flotante** y por tanto no podemos operar con números reales. Para solventarlo, haremos una aproximación usando aritmética entera:

```
dest =  (2126*orig.R + 7152*orig.G + 722*orig.B) /10000;
```

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**. Cuando usamos el lenguaje C para codificar la sentencia anterior, se producen una serie de *castings* automáticos (cambios de tipo) para poder realizar las operaciones. Pero si hacemos cada una por separado, como en el siguiente ejemplo, el resultado sería **incorrecto**:

```
unsigned char aux_r, aux_g, aux_b;

aux_r =  2126*orig.R; // valor no codificable con 8 bits
aux_g = 7152*orig.G; // valor no codificable con 8 bits
aux_b =  722*orig.B; // valor no codificable con 8 bits
dest = (aux_r + aux_g + aux_b)/10000;
```

1.10.2. Transformación a imagen binaria

Existen múltiples algoritmos para *binarizar* una imagen en escala de grises, pero todas ellas coinciden en un aspecto: comparan cada *pixel* con un umbral y, si el valor *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.

Para esta práctica usaremos la técnica más sencilla de todas: un umbral único y constante para toda la imagen. La matriz que representará la imagen binaria necesita únicamente un bit para almacenar cada *pixel*, pero en esta parte de la práctica usaremos aún así un tipo `unsigned char` para representar cada *pixel*. Por tanto, el siguiente código puede servir de base para realizar la transformación binaria:

```
unsigned char imagenGris[N*M];
unsigned char imagenBinaria[N*M];
int i,j;
// umbral será una variable o constante de nuestro código

for (i=0;i<N;i++)  {
  for (j=0;j<M;j++) {
    if  (imagenGris[i*M + j] > umbral)
        imagenBinaria[i*M + j]= 1;
    else
        imagenBinaria[i*M + j]= 0;
  }
}
```

1.10.3. Trabajo a desarrollar (parte obligatoria)

El proyecto se desarrollará sobre los ficheros que se enumeran a continuación:

- `init.S` Este fichero, en lenguaje ensamblador, contendrá el símbolo global `start` y símplemente invocará la función `main()` que estará en el fichero `main.c`.**Este fichero se entregará completo** y no es necesario modificarlo.
- `main.c`. Este fichero definirá la función principal que se encargará de inicializar la matriz de color (RGB) e invocar a las funciones que implementan las dos transformaciones. Tras las dos transformaciones, se invocará a una última función `contarUnos` que se detallará a continuación. Asimismo, la declaración de las matrices necesarias para la práctica se realizará en este fichero. **Este fichero se entregará completo** y no es necesario modificarlo. A continuación se muestra la implementación de la función `main()` para ilustrar el flujo de ejecución del programa completo:

```
int main() {

    initRGB(imagenRGB,N,M);

    RGB2GrayMatrix(imagenRGB,imagenGris,N,M);

    Gray2BinaryMatrix(imagenGris,imagenBinaria,127,N,M);

    contarUnos(imagenBinaria,unosPorFila,N,M);

    return 0;
}
```

- `types.h`. Este fichero contiene la definición del tipo `imagenRGB`.
- `trafo.h`. Este fichero contiene el prototipo de las funciones utilizadas en la función `main()`, salvo `initRGB(...)` que se implementa en el fichero `main.c`. Incorpora también el prototipo de una función auxiliar `rgb2gray` que deberá implementase en ensamblador
- `trafo.c`. En este fichero **el alumno realizará la implementación** de las transformaciones explicadas anteriormente (salvo las funciones que se solicitan en ensamblador)
- `misc.S` En este fichero **el alumno deberá realizar la implementación, en ensamblador, de las funciones** `rgb2gray` y `contarUnos`.

Resumiendo, el trabajo del alumno consistirá en:

- Implementar, en **ensamblador**, la función `unsigned char rgb2gray(pixelRGB* pixel);` Deberá aplicar la función indicada anteriormente a los tres valores del pixel recibido como parámetro[5] para obtener el correspondiente valor en escala de grises. Como en el **repertorio ARM disponible no hay una instrucción de división**, se proporciona una rutina `div10000` que recibe un argumento (el numerador de la división) y devuelve una aproximación del cociente de dividir dicho argumento entre 10000. Esto convierte a la rutina `rgb2gray` en *no-hoja*. Se desarrollará en el fichero `misc.S`.
- Implementar en C la función `void RGB2GrayMatrix(pixelRGB orig[], unsigned char dest[], int nfilas, int ncols);`. Esta función recorrerá todos los *pixels* de la matriz `orig` y llamará a la función `rgb2gray` para obtener el valor correspondiente a ese mismo *pixel* en la matriz `dest`. Se desarrollará en el fichero `trafo.c`.
- Implementar en C la función `void Gray2BinaryMatrix(unsigned char orig[], unsigned char dest[], unsigned char umbral, int nfilas, int ncols);` Esta función recorrerá todos los *pixels* de la matriz `orig` comparándolos con el `umbral` proporcionado, escribiendo el valor correspondiente (0 ó 1) en la matriz `dest`. Se desarrollará en el fichero `trafo.c`.
- Implementar, en **ensamblador**, la función `void contarUnos(unsigned char mat[], short int vector[],int nfilas, int ncols);`. Esta función recorrerá la matriz `mat` contando el número de unos que haya en cada una de sus filas. El resultado lo guardará en el array `vector`, de modo que `vector[i]` contendrá el número de 1's que hay en la *i-ésima* fila de `mat`. Se desarrollará en el fichero `misc.S`.

[5]Se recibe un puntero al pixel, esto es, la dirección de un pixel

Bibliografía

[ARMa] ARM. The arm architecture procedure call standard. Accesible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042d/index.html. Hay una copia en el campus virtual.

[ARMb] ARM. Arm architecture reference manual. Accesible en http://www.arm.com/miscPDFs/14128.pdf. Hay una copia en el campus virtual.

[GNUa] GNU. Gnu arm assembler quick reference. Accesible en http://microcross.com/GNU-ARM-Assy-Quick-Ref.pdf. Hay una copia en el campus virtual.

[GNUb] GNU. Using ld, the gnu linker. Accesible en http://www.zap.org.au/elec2041-cdrom/gnutools/doc/gnu-linker.pdf. Hay una copia en el campus virtual.