Práctica 4

Paso de lenguaje de alto nivel a ensamblador

4.1. 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:

- Conocer el convenio para el paso de parámetros de ARM.
- Comprender el código generado por el compilador *gcc*.
- Saber utilizar en C variables o rutinas definidas en ensamblador y viceversa.
- Comprender los distintos ámbitos de variables, local y global.
- 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. Revisión del concepto *Pila de llamadas*

Ya hemos visto en la práctica anterior que para utilizar subrutinas necesitamos introducir el concepto de pila de llamadas (*call stack* en inglés). Se trata de una región de memoria gestionada como cola LIFO (Last In, First Out) utilizada con tres fines:

- Copiar parámetros de la llamada a una subrutina
- Copiar valores de registros para poder restaurarlos al terminar
- Utilizarla como memoria local a la rutina

Pongamos un ejemplo sencillo, por ahora sin atender a las particularidades de la arquitectura destino. Supongamos que en un punto dado del código queremos invocar una rutina `funA` cuya declaración en C fuese:

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

Se trata de una función (en C todas las rutinas son funciones) que toma 5 parámetros enteros (tamaño palabra) y devuelve un resultado entero. En una llamada concreta a la función debemos pasarle los valores para los parámetros `a1-a5`. Dos llamadas diferentes a la función pueden pasar valores distintos para estos parámetros. 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 para `a1-4` respectivamente, y a `a5` le asignamos lo que contenga la variable `b`. Lo que devuelve la función lo almacenamos en la variable `a`.

Además la declaración de la función nos dice otras cosas. La función tendrá, al menos, 5 *variables locales*, llamadas `a1-a5` respectivamente. Tendremos una versión privada de cada

una 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).

Debemos ser conscientes de que aunque la rutina se ha escrito como una función C, al final el compilador debe traducir el código C a un código ensamblador utilizando el repertorio de instrucciones de la arquitectura destino, ARM en nuestro caso. Este código utilizará una serie de registros para hacer los cálculos propios de la rutina, machacando el valor que tuviesen antes de la invocación de la rutina.

Por ejemplo, supongamos que el código de `funA`, traducido a ensamblador, contiene las instrucciones siguientes:

```
ldr   r4, [fp, #-16]
ldr   r5, [fp, #-20]
add   r6, r4, r5
```

En este caso los registros `r4`, `r5` y `r6` son machacados, escribiéndose un valor nuevo en ellos. Sin embargo quien utiliza la función no tiene por qué saber esto, al menos eso es lo que nos interesa, y si tenía algo importante en estos registros antes de la llamada a `funA` querrá seguir teniéndolo después. Por ello `funA` debería copiar estos registros en memoria antes de utilizarlos, y restaurar su valor antes de terminar la función.

Parece entonces claro que debe establecerse un convenio entre el que escribe una rutina (`funA` en nuestro ejemplo) y el que la va a invocar, que debe determinar:

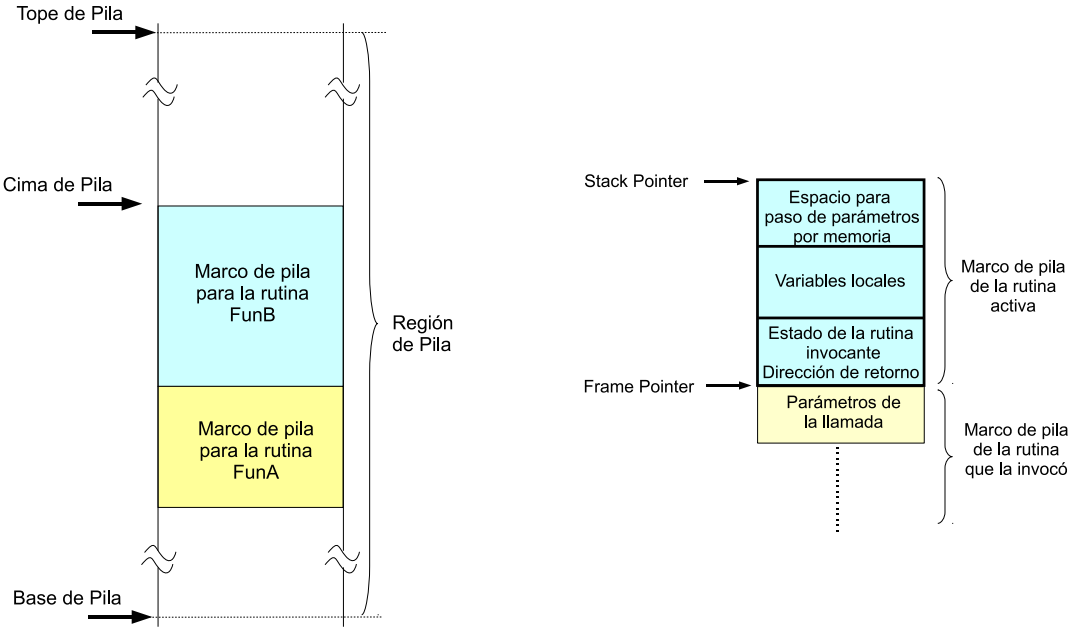
1. Cómo se pasarán los argumentos en la llamada a la rutina y cómo se obtendrá el valor de retorno
2. Qué registros deben preservarse
3. Como se gestionará la pila de llamadas, que contendrá los registros salvados, las variables locales y alguna otra memoria privada que pueda necesitar la rutina para su ejecución.

Este convenio es dependiente de la arquitectura destino y es imprescindible respetarlo si queremos mezclar código generado por distintas herramientas. Este es el caso al que nos enfrentamos en la práctica, mezclar un código generado por un compilador C con el código programado por nosotros directamente en ensamblador.

4.2.1. Marcos de Activación

La figura 4.1a ilustra el estado en el que estaría la pila de llamadas en el caso de que una rutina `FunA` invocase a otra rutina `FunB` y esta última estuviese en ejecución. En este caso, hay al menos dos rutinas activas (`FunA` y `FunB`) y la pila mantendrá la información de ambas.

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. La figura 4.1b muestra un esquema general de



(a) Pilla de llamadas (b) Marco de Activación

Figura 4.1: Pila de llamadas. (a) Ejemplo del estado de la pila de llamadas en el caso de que una función FunA haya invocado la rutina FunB y esta última se encuentre en ejecución. (b) Estructura del marco de activación de FunB cuando está en ejecución.

la organización del marco de activación de una subrutina.

Para facilitar el acceso a la información contenida en el marco, es habitual utilizar dos punteros:

- *Stack Pointer* (SP): que apunta siempre a la cima del marco de pila de la rutina activa, es decir, a la cima de la pila.
- *Frame Pointer* (FP): que apunta a una posición preestablecida del marco, generalmente la base del marco.

Los punteros SP y FP se almacenan normalmente en registros de la arquitectura. Habitualmente se utiliza el registro FP para direccionar tanto las variables locales (por encima de FP en el propio marco de la rutina) como los parámetros de la llamada (por debajo de FP, en el marco de la rutina que la invocó). La estructura concreta del marco de activación se fija por convenio (estándar) para la arquitectura destino.

4.2.2. Estándar de llamadas a procedimientos para ARM

El ARM Architecture Procedure Call Standard (AAPCS) es el estándar que regula las llamadas a procedimientos en la arquitectura ARM [aap]. Especifica una serie de normas para que las rutinas puedan ser compiladas o ensambladas por separado, y que a pesar de ello, puedan interactuar entre ellas. En definitiva, supone un contrato entre la rutina que invoca y la rutina invocada que define:

Uso de registros arquitectónicos

Este aspecto se abordó ya en la práctica anterior, ahora sólo resumimos y damos algún detalle adicional:

- Los primeros cuatro registros (r0-r3 o a1-a4) se utilizan para pasar parámetros a la rutina y obtener el valor de retorno y no es necesario que la rutina los preserve.
- Los registros r4-r11 (o v1-v8) y r13 **deben ser preservados**.
  - r11 es el registro que debe utilizarse como FP.
  - r13 es el registro que debe utilizarse como SP.
- Los registros r12 (o IP) y r14 (o LR) no es necesario preservarlos
  - r14 (o LR), es el registro que debe utilizarse para pasar la dirección de retorno.

Modelo pila

La figura 4.2 ilustra el modelo *full-descending* impuesto por el AAPCS, que se caracteriza por:

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

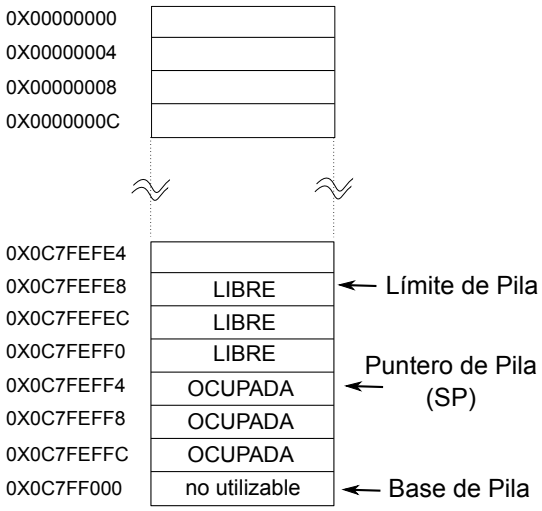


Figura 4.2: Ilustración de una pila *Full Descending*.

Mecanismo de llamadas a subrutina

La llamada a una rutina puede hacerse con cualquier secuencia de instrucciones que consiga:

- escribir en LR (R14) la dirección de retorno, y
- escribir en PC (R15) la dirección de comienzo de la rutina.

Las dos principales alternativas son:

- **Convencional**: propuesta en la práctica anterior, utilizando la instrucción BL con salto relativo a PC (ej: `BL FUNC`).
- **Alternativa**: Utilizando una secuencia de instrucciones, por ejemplo:

```
MOV LR, PC      @ LR <- dirección de retorno, valor leído de PC (. + 8)
LDR PC, =FUNC   @ La dirección FUNC se resuelve al enlazar
                @ Esta es la dirección de retorno
```

  Es preciso recordar aquí que, debido a la segmentación del procesador, cuando una instrucción lee el registro PC, éste contiene la dirección de la propia instrucción incrementada en 8 bytes (i.e. dos inst.).

La primera de las alternativas tiene un problema cuando el destino del salto (comienzo de la rutina) está en otra sección de código, ya que en ese caso el desplazamiento relativo a PC no se puede calcular hasta el momento de enlazado, que es cuando se decide la ubicación en memoria de todas las secciones. La segunda alternativa no presenta este problema y por ello es preferible utilizarla para invocar una rutina definida en otra sección.

Valor de retorno

Para aquellas subrutinas que devuelven un valor (funciones), el AAPCS especifica que debe devolverlo en R0 [1].

Paso de parámetros

Ya vimos en la práctica anterior que los cuatro primeros parámetros se deben pasar por registro, utilizando en orden los registros r0-r3. A partir del cuarto parámetro hay que pasarlos por memoria, escribiéndolos en la parte superior del marco de pila de la rutina invocante, en el orden siguiente:

- el primero de los parámetros restantes en $[SP]$,
- el siguiente en $[SP + 4]$,
- el siguiente en $[SP + 8]$,
- ...

La figura 4.3 ilustra un ejemplo, cuando FunA debe invocar la rutina FunB, 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 a través de la pila, en la cima del marco de FunA.

[1] Esta es una simplificación válida para valores de tamaño palabra o menor. El resto de los casos queda fuera del objetivo de este documento

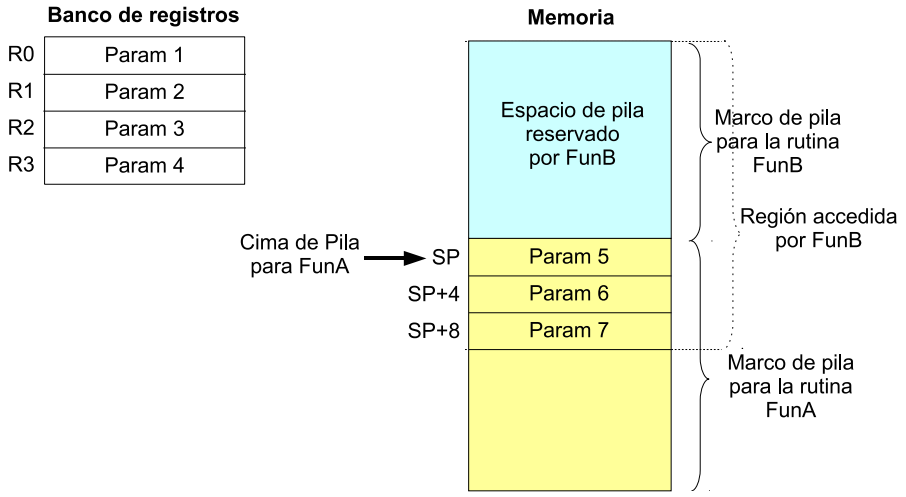


Figura 4.3: Paso de los parámetros Param1-7 a FunB desde FunA.

Estructura de una rutina

El cuerpo de las rutinas suele estructurarse en tres partes:

```
Código de entrada (prólogo)
Cuerpo de la rutina
Código de salida (epílogo)
```

El **prólogo** construye el marco de activación sobre la cima de la pila.

El **epílogo** restaura el estado de la rutina invocante, es decir, registros y pila, y copia en PC la dirección de retorno.

El cuadro 1 describe el prólogo generado por compilador de GNU (gcc), cuando no se activa ninguna opción de optimización. La figura 4.4 ilustra la estructura del marco de pila resultante.

Cuadro 1 Prólogo de función C generado por gcc.

```
MOV     IP, SP                          @ Copiar en IP la cima actual
STMDB   SP!, {r4-r10,FP,IP,LR,PC}       @ Copiar registros en la pila
SUB     FP, IP, #4                      @ FP <- dirección base del marco
SUB     SP, #EspacioVariablesYParams    @ Reservar espacio para variables
                                        @ locales y parámetros de llamadas
```

Debemos resaltar y/o matizar algunas cosas:

- No se almacenan siempre todos los registros R4-R10, sólo aquellos que se vayan a utilizar en la rutina.
- El registro R12 (IP) no se preserva, su valor anterior se pierde en la primera instrucción del prólogo. Esto no causa problemas porque el AAPCS dice que no es necesario preservarlo. Debemos ser conscientes de ello al invocar una rutina C desde el ensamblador, si tenemos en R12 un contenido que deseemos preservar, debemos copiarlo

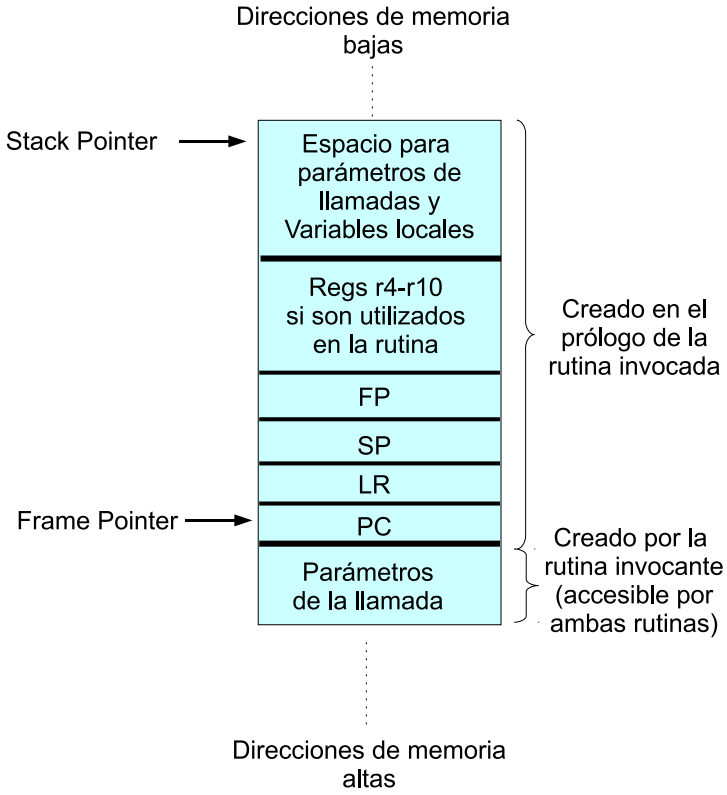


Figura 4.4: Marco de activación generado por gcc con opciones de depuración activadas.

antes de la llamada en un lugar seguro (lo mismo debemos hacer con r0-r3).

- En una compilación con opciones de depuración gcc almacena siempre el PC en la pila. Esto es útil para los depuradores.

El cuadro 2 presenta el correspondiente epílogo generado por gcc. Comparando la lista de registros de la instrucción `LDMDB` con la del `STMDB` del prólogo, podemos ver que se ha suprimido LR. Esto hace que el valor copiado en PC sea el valor que tenía LR al entrar en la función, realizándose correctamente el retorno. En R4-R10, FP y SP se copian los valores que tenían al entrar en la subrutina, por lo que al terminar la ejecución de esta instrucción se restaura por completo el estado de la rutina invocante. Al restaurar FP y SP el marco activo vuelve a ser el de la rutina invocante.

Cuadro 2 Epílogo para funciones C generado por gcc.

```
LDMDB   FP, {r4-r10,FP,SP,PC}
```

4.3. Variables locales y globales

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 organización

del programa permite definir variables en dos ámbitos diferentes: *global*, variables que son accesibles desde cualquier función, y *local*, variables que son accesibles sólo dentro de una determinada función.

Las variables globales 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. Las variables locales son almacenadas en la pila, dentro del marco de activación de la función, como ilustra la figura 4.4. Este espacio es reservado por el código de entrada de la función (prólogo) y es liberado por el código de salida (epílogo).

4.4. Utilizando varios ficheros fuente

Para combinar código C con ensamblador nos interesará dividir un programa en dos o más ficheros fuente, algunos escritos en C y otros en ensamblador. Pero los ficheros no serán independientes, querremos utilizar en alguno de ellos variables o rutinas definidas en otro. Sin embargo, vimos en la práctica 2 que los compiladores generan un fichero objeto por cada fichero fuente, sin tener en cuenta otros ficheros. Finalmente, una etapa de enlazado combina los ficheros objeto en un ejecutable. En ésta última etapa se deben resolver todas las *referencias cruzadas* entre los ficheros objeto. El objetivo de esta sección es entender este mecanismo y su influencia en el código generado.

4.4.1. 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, siendo la *Tabla de Símbolos* una de las más importantes. Esta tabla contiene información de los identificadores utilizados en el fichero fuente que tienen visibilidad externa. 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` (estándar GNU) puede consultarse con el programa `nm`. Veamos lo que nos dice `nm` para un fichero objeto creado para este ejemplo:

```
> arm-elf-nm  -SP ejemplo.o
globalA D 00000000 00000002
globalB U
main T 00000000 00000048
printf U
```

Sin conocer el código fuente sabemos 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 (texto) y ocupa 0x48 bytes. Es la función de entrada del programa C.

- Hay otro símbolo `printf`, que no está definido (debemos importarlo).

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

4.4.2. Símbolos globales en C

El cuadro 3 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. Esto quiere decir que el enlazador buscará en otro de los ficheros a enlazar una definición, y si no la encuentra en ninguno se añade al final a la sección `.bss` del ejecutable final. Si se usa el modificador `extern` se obliga a que el símbolo sea definido en alguno de los ficheros a enlazar.

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.

4.4.3. 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:

Cuadro 3 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;
   ...
}
```

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

start:
      mov lr, pc
      ldr pc,=FOO
      ...
```

4.4.4. 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 4.4.2.

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`.

4.4.5. Resolución de símbolos

Vamos a estudiar con un poco más de detalle como es este proceso de resolución de símbolos que lleva a cabo el enlazador. El proceso se ilustra en el cuadro 4. En la parte superior izquierda podemos ver parcialmente el código de dos ficheros: `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.

En el cuadro de la derecha se muestra el desensamblado del código objeto generado por gcc al compilar `main.c`. Para obtenerlo hemos seguido la siguiente secuencia de pasos:

```
> arm-elf-gcc -O0 -c -o main.o main.c
> arm-elf-objdump -D main.o
```

Se realza en rojo las instrucciones ensamblador generadas para la traducción de la instrucción C marcada en rojo. Como vemos es una traducción compleja. Lo primero que hay que observar es que la operación C es equivalente a:

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

Entonces lo primero que habría que hacer es conseguir 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 0x40 de la sección de código de `main.o`, marcada también en rojo.

Como vemos, las dos primeras instrucciones marcadas en rojo hacen lo mismo, cargan en r2 y r3 respectivamente la entrada 0x40 de la sección de texto, es decir, la dirección de `MIVAR`. La tercera instrucción carga en r3 el valor de la variable y la siguiente instrucción le suma 1. Finalmente se escribe el resultado en la dirección almacenada en r2 (la de `MIVAR`).

Pero si nos fijamos bien en el cuadro, veremos que la entrada 0x40 de la sección de código 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 en la tabla de *reubicación* para `MIVAR`, como podemos ver con la herramienta objdump:

```
> arm-elf-objdump -r main.o

main.o:     file format elf32-littlearm

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000040 R_ARM_ABS32       MIVAR
```

La única entrada que hay es una indicación para el enlazador. Le dice que cuando resuelva el símbolo `MIVAR`, lo escriba en la entrada 0x40 como un valor entero sin signo de 32 bits.

La parte inferior del cuadro 4 muestra el desensamblado del ejecutable tras el enlazado. Como vemos, se ha ubicado la sección de código de Main a partir de la dirección de memoria 0x0C000020. La entrad 0x40 corresponde ahora a la dirección 0x0C000060 y contiene el valor 0x0c000064. Vemos que justo esa dirección es la que corresponde al símbolo `MIVAR`, y contiene el valor 0x2 con el que se inicializó en `init.s`.

Cuadro 4 Ejemplo de resolución de símbolos.

```
@ Fichero init.s

.data
.global MIVAR
MIVAR:  .word 0x02

.text
...    @ Omitido
```

```
// Fichero main.c:
// importa MIVAR

extern int MIVAR;

int Main(void)
{
    int i;

    MIVAR++;
    i = 1 + MIVAR;

    return i;
}
```

```
//desensamblado del código objeto main.o
//generado por gcc al compilar main.c

Disassembly of section .text:

00000000 <Main>:
   0:   e1a0c00d    mov ip, sp
   4:   e92dd800    stmdb   sp!, fp, ip, lr, pc
   8:   e24cb004    sub fp, ip, #4  ; 0x4
   c:   e24dd004    sub sp, sp, #4  ; 0x4
  10:   e59f2028    ldr r2, [pc, #40]   ; 40 <Main+0x40>
  14:   e59f3024    ldr r3, [pc, #36]   ; 40 <Main+0x40>
  18:   e5933000    ldr r3, [r3]
  1c:   e2833001    add r3, r3, #1  ; 0x1
  20:   e5823000    str r3, [r2]
  24:   e59f3014    ldr r3, [pc, #20]   ; 40 <Main+0x40>
  28:   e5933000    ldr r3, [r3]
  2c:   e2833001    add r3, r3, #1  ; 0x1
  30:   e50b3010    str r3, [fp, -#16]
  34:   e51b3010    ldr r3, [fp, -#16]
  38:   e1a00003    mov r0, r3
  3c:   e91ba800    ldmdb   fp, fp, sp, pc
  40:   00000000    andeq   r0, r0, r0
```

```
//desensamblado del código tras la fase de enlazado

Disassembly of section .text:
... // Omitido
0c000020 <Main>:
 c000020:   e1a0c00d    mov ip, sp
 c000024:   e92dd800    stmdb   sp!, fp, ip, lr, pc
 c000028:   e24cb004    sub fp, ip, #4  ; 0x4
 c00002c:   e24dd004    sub sp, sp, #4  ; 0x4
 c000030:   e59f2028    ldr r2, [pc, #40]   ; c000060 <Main+0x40>
 c000034:   e59f3024    ldr r3, [pc, #36]   ; c000060 <Main+0x40>
 c000038:   e5933000    ldr r3, [r3]
 c00003c:   e2833001    add r3, r3, #1  ; 0x1
 c000040:   e5823000    str r3, [r2]
 c000044:   e59f3014    ldr r3, [pc, #20]   ; c000060 <Main+0x40>
 c000048:   e5933000    ldr r3, [r3]
 c00004c:   e2833001    add r3, r3, #1  ; 0x1
 c000050:   e50b3010    str r3, [fp, #-16]
 c000054:   e51b3010    ldr r3, [fp, #-16]
 c000058:   e1a00003    mov r0, r3
 c00005c:   e91ba800    ldmdb   fp, fp, sp, pc
 c000060:   0c000064    stceq   0, cr0, [r0], 100
Disassembly of section .data:

0c000064 <MIVAR>:
 c000064:   00000002    andeq   r0, r0, r2
```

4.5. Arranque de un programa C

Como hemos explicado más arriba, un programa C siempre ha de tener una función `main` por la que debe empezar la ejecución del programa. 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.

En nuestro contexto, donde no hay sistema operativo y cargamos directamente el programa en memoria para su ejecución, hay dos motivos por los que no podemos iniciar el programa en la función main:

- La pila no estaría inicializada, es decir, habría basura en el registro SP, y
- No se habría pasado una dirección de retorno, es decir, habría basura en LR.

El segundo motivo lo podríamos solventar haciendo que `main` no terminase (esperase en un bucle infinito), pero seguiríamos teniendo el problema de la inicialización de la pila. Por ello, siempre que hagamos un proyecto en C añadiremos un fichero ensamblador con la rutina de inicio. Esta rutina se encargará de preparar el sistema, luego invocará la función `main` y retomará el control cuando termine dicha función. Finalmente esperará en un bucle infinito. El cuadro 5 nos muestra un posible código para esta rutina de inicialización.

Cuadro 5 Ejemplo de rutina de inicialización.

```
    .extern main
    .global start
    .equ STACK, 0x0c7ff000

    .text
start:
    ldr sp,=STACK
    mov fp,#0

    mov lr,pc
    ldr pc,=main

End:
    B End
.end
```

Como vemos, además de lo explicado anteriormente la rutina presentada en el cuadro 5 pone a 0 el registro FP antes de llamar a la función `main`. Esto se hace así para dejar una señal en el marco de activación de `main` que indique que es el primer marco de la pila. Esto lo necesita el depurador cuando le pedimos hacer una traza de la pila de llamadas (*call trace*).

4.6. 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.

4.6.1. 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 de datos con valor inicial, asignándoles los valores como indica la figura 4.5. 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`).

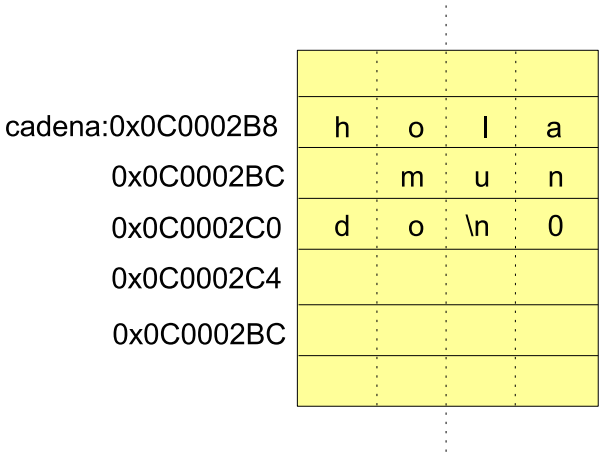


Figura 4.5: Almacenamiento de un array de caracteres en memoria.

Dependiendo de la arquitectura puede haber restricciones de alineamiento en el acceso a memoria. Este es el caso de la arquitectura ARM, donde los accesos deben realizarse a direcciones alineadas con el tamaño del acceso. Así, los accesos de tamaño byte pueden realizarse a cualquier dirección, en cambio los accesos a datos de tamaño palabra (4 bytes) sólo pueden realizarse a direcciones múltiplo de cuatro. 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.

4.6.2. 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 4.6.

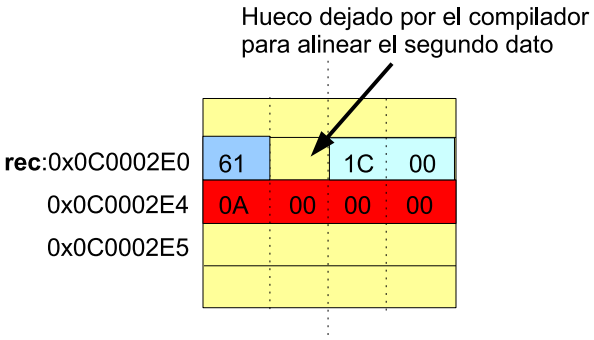


Figura 4.6: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a 0x61, 0x1C y 0x0A respectivamente.

4.6.3. 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 4.6.2. Sin embargo su huella de memoria (*memory footprint*) es muy distinta, como ilustra la figura 4.7.

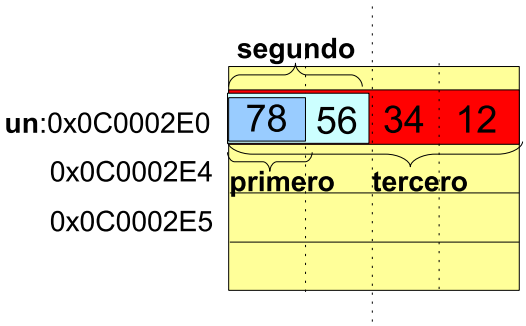


Figura 4.7: Almacenamiento de la union *un*, con los campos primero, segundo y tercero. Un acceso al campo primero da como resultado el byte *0x78*, es decir el carácter *x*. Un acceso al campo segundo da como resultado la media palabra *0x5678* (asumiendo configuración *little endian*), es decir el 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.

4.7. Desarrollo de la práctica

En esta práctica partiremos de un proyecto que permite ejecutar un programa C en nuestro sistema de laboratorio. Es un sencillo programa de ejemplo que declara en memoria un array `Pacientes`. Cada elemento del array es una estructura con dos campos: prioridad (prioridad con la que se debe tratar al paciente) y pid (identificador de paciente). El array tiene un máximo de 16 entradas, pero no tienen por qué estar todas rellenas. El array se llena siempre empezando en la posición 0 y el final viene indicado por la primera entrada con todos los bytes a 0 ({0,0}).

El programa se inicia con el array `Pacientes` inicializado en memoria. El array tiene una serie de entradas rellenas. El programa primero calcula el número de pacientes registrados, contando el número de entradas hasta encontrar la primera que está a 0.

Después ordena el array por prioridades, en orden creciente. El método de ordenación es muy sencillo: se van colocando los elementos empezando por la posición 0. En cada posición se elige el menor de los elementos que quedan. Pongamos un ejemplo con la siguiente lista:

$$5, 3, 7, 4, 8, 9, 7$$

Inicialmente no se ha seleccionado ningún elemento. Empezamos buscando el nuevo elemento para la primera posición, que será el menor de los no seleccionados, es decir, el 3. Para poder hacerlos sobre el mismo array intercambiamos el 3 con el 5, que es el elemento que estaba en la primera posición originalmente. Nos queda:

$$3, 5, 7, 4, 8, 9, 7$$

El siguiente paso es buscar el elemento para la segunda posición, que será el menor de los que quedan, es decir, todos menos el 3 seleccionado antes. De nuevo intercambiamos su posición con el que está en la segunda, y nos queda:

$$3, 4, 7, 5, 8, 9, 7$$

En el siguiente paso encontramos el 5, que intercambiamos con el 7, resultando:

$$3, 4, 5, 7, 8, 9, 7$$

A continuación encontramos que el menor es un 7. Nos quedamos, por ejemplo, con el primero de ellos, y el array no cambia. Después encontramos que el menor de los que quedan es otro 7, que se intercambia con el 8 resultando:

$$3, 4, 5, 7, 7, 9, 8$$

Un paso más dejaría el array en:

$$3, 4, 5, 7, 7, 8, 9$$

El algoritmo descrito se implementa muy fácilmente con dos funciones de apoyo:

```
int PosMinPrioridad(struct Descriptor* P,int ini, int num);
```

que nos da la posición en el array apuntado por `P` del elemento con menor prioridad a partir de la posición `ini`, siendo `num` el número de elementos en `P`; y

```
void Intercambiar(struct Descriptor* P, int i, int j);
```

que intercambia los elementos `i` y `j` en el array apuntado por `P`.

Al comenzar se dan estas rutinas implementadas, todas en C excepto `Intercambiar`, que se da en ensamblador. Hay que analizar el código y depurarlo paso a paso con el fin de afianzar los conceptos expuestos en la parte teórica. Después se proponen una serie de ejercicios.

Los pasos que debemos ir realizando son:

1. Abrir el Embest IDE y crear un workspace nuevo.
2. Crear un nuevo fichero, guardarlo con el nombre `ld_script.ld` y añadirlo al proyecto. Este será el fichero de entrada al enlazador, que determinará las secciones de nuestro ejecutable y su ubicación.

3. Copiar en el fichero ld_script.ld el siguiente contenido:

```
SECTIONS
{
    . = 0x0C000000;
    .text : { *(.text) }
    _bdata = .;
    .data : { *(.data) }
    _edata = .;
    .rodata : { *(.rodata) }
    _bbss = .;
    .bss : { *(.bss) }
    _ebss = .;
}
```

4. Crear un nuevo fichero con el nombre `init.s` y añadirlo a la carpeta *Project Source Files*. Este fichero será el encargado de inicializar la arquitectura para la ejecución de nuestro programa escrito en lenguaje C y de invocar la función de entrada a nuestro programa. Incluir en este fichero el código del cuadro 5.

5. Crear un nuevo fichero fuente, con el nombre `main.c`, añadirlo a la carpeta *Project Source Files* y copiar el siguiente contenido:

```
#define MAX_PACIENTES 16

struct Descriptor {
    unsigned int prioridad;
    unsigned int pid;
};


struct Descriptor Pacientes[MAX_PACIENTES] = { {127,1},{127,2},{112,3},
                        {100,4},{132,5},{136,6},{255,7},{10,8},{0,0}};

int NumPacientes(struct Descriptor* P)
{
    int i;

    for( i = 0; i < MAX_PACIENTES; i++ )
        if( (P[i].prioridad == 0) && ( P[i].pid == 0 ) )
            return i;

    return MAX_PACIENTES - 1;
}

int PosMinPrioridad(struct Descriptor* P,int ini, int num)
{
    int minpos = ini;
```

```
    unsigned int min = P[ini].prioridad;
    int i;

    for( i = ini+1; i < num; i++)
        if( P[i].prioridad < min ){
           minpos = i;
           min = P[i].prioridad;
        }

    return minpos;
}

extern void Intercambiar(struct Descriptor* P, int i, int j);

void OrdenaPorPrioridad(struct Descriptor*  A, int num)
{
    int i,j;

    for( i = 0; i < num ; i++ )
    {
        j = PosMinPrioridad(A,i,num);
        Intercambiar(A,i,j);
    }
}

int main(void)
{
    int num;

    num = NumPacientes(Pacientes);

    OrdenaPorPrioridad( Pacientes, num );

    return 0;
}
```

6. Añadir al proyecto un nuevo fichero `rutinas_asm.s` con el siguiente código:

```
    .global Intercambiar

Intercambiar:
    mov ip,sp
    stmdb sp!,{fp,ip,lr,pc}
    sub fp,ip,#4

    add r1, r0, r1, lsl #3
    add r2, r0, r2, lsl #3
    ldr r3, [r1]
    ldr r0, [r2]
    str r3, [r2]
    str r0, [r1]
    ldr r3,[r1,#4]
    ldr r0,[r2,#4]
    str r3,[r2,#4]
    str r0,[r1,#4]

    ldmdb fp,{fp,sp,pc}

    .end
```

7. Configurar el proyecto tal y como se hizo en las prácticas anteriores.
8. Compilar el proyecto y crear el ejecutable.
9. Conectarse al depurador. Si el procesador está corriendo (botón de stop en rojo) lo paramos pulsando el botón de stop.
10. Cargamos el programa y lo ejecutamos paso a paso analizando lo que sucede.
11. Se pide al alumno:
    a) Obtener una función C equivalente a `Intercambiar`.
    b) Reemplazar la función `OrdenaPorPrioridad` por una rutina codificada en ensamblador en el fichero `rutinas_asm.s`.

Bibliografía

[aap] 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.