Práctica 2

Paso de lenguaje de alto nivel a ensamblador

2.1. Objetivos

En la práctica anterior hemos estudiado los conceptos básicos de la arquitectura ARM y de su programación en lenguaje ensamblador. En esta práctica profundizaremos en nuestros conocimientos sobre la programación en ensamblador analizando su interacción con un lenguaje de alto nivel como C. Los principales objetivos son:

- Conocer el convenio de paso de parámetros a funciones.
- Comprender 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*.

2.2. La Pila de llamadas

Como hemos visto en la práctica anterior, 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*).

Esta región de memoria, cuyos accesos siguen una política LIFO (*Last-In-First-Out*, sirve para almacenar información relativa a las rutinas activas del programa. Por ejemplo si en un programa, una rutina FunA invoca a una rutina FunB, cuando FunB comience su ejecución, habrá al menos dos rutinas activas y la pila mantendrá la información de ambas. 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. En el caso de ARM se suele emplear el registro R13 para este propósito.

La pila suele organizase en sub-regiones denominadas marcos de activación o simplemente marcos de pila. Cada uno de estos marcos contiene la información de una rutina activa y su contenido depende tanto de la arquitectura del procesador como del tipo de código que se desee generar. Por ello, para poder combinar códigos compilados por separado, es necesario recurrir a estándares o convenios. El marco de una rutina generalmente contiene información tanto del estado de ejecución de la rutina (parámetros de la llamada y las variables locales, esencialmente) como información necesaria para restaurar el estado de la rutina que la invocó.

Para facilitar el acceso a la información contenida en el marco, es habitual utilizar un puntero adicional denominado *frame pointer* (FP) que apunta a una posición preestablecida del mismo, por ejemplo a aquella en la que se almacena la dirección de retorno de la rutina. Habitualmente los parámetros quedan almacenados en direcciones superiores a la apuntada por FP y las variables locales de la rutina, en direcciones inferiores al FP. Es también habitual que los compiladores hagan uso del FP para direccionar tanto los parámetros como las variables locales de la rutina. Siguiendo el ejemplo anterior, si la rutina FunA invoca a la rutina FunB, la estructura de la pila sería similar a la descrita en la figura 2.1.

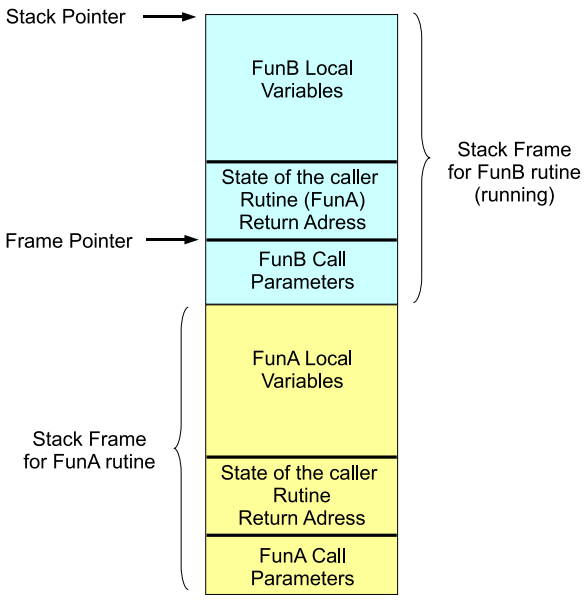

Figura 2.1: Estructura de la pila cuando la rutina funA ha invocado a la rutina funB.

2.3. Estándar de llamadas a procedimientos para ARM

En la práctica anterior hemos utilizado una rutina muy sencilla para realizar una pequeña tarea, pasando sus argumentos a través de registros. Si queremos poder ensamblar esta rutina de forma que otros puedan utilizarla en sus programas a partir del fichero objeto, debemos indicar cómo han de que pasarse los argumentos a la rutina. Lo mismo sucede si queremos invocar la rutina desde un código C, aunque este sea propio. El compilador debe saber cómo pasar los parámetros para traducir correctamente la llamada a la rutina.

Si cada rutina de cada desarrollador se comporta de forma diferente es imposible combinar los distintos códigos. Es por lo tanto necesario un estándar que homogeneice su comportamiento y garantice la interacción entre ellos. El ATPCS (ARM-Thumb Procedure Call Standard) es el estándar que regula las llamadas de procedimientos en la arquitectura ARM [atp]. Especifica una serie de normas para que las rutinas puedan ser compiladas y 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:

- Obligación para la rutina que invoca de crear un estado de memoria a partir del cual la rutina invocada pueda comenzar su ejecución (ej. paso de parámetros).
- Obligación para la rutina invocada de preservar, a lo largo de su ejecución, una parte del estado de memoria de la rutina que la invocó.
- Derechos de la rutina invocada a modificar parte del estado de memoria de la rutina que la invocó (ej. retorno de resultados).

El estándar define una serie de variantes que responden a distintas prioridades para la generación de código:

- Tamaño de código.
- Rendimiento del programa.
- Funcionalidad (por ejemplo facilidad de depuración, comprobaciones en tiempo de ejecución o soporte para bibliotecas de enlace dinámico).

Sin embargo, nosotros analizaremos exclusivamente el estándar básico. Para ampliar la información sobre las diversas variantes es preciso consultar el documento de referencia de ATPCS [atp].

2.3.1. Modelo de memoria

El estándar está diseñado para programas compuestos de un solo hilo de ejecución o proceso (en nuestro contexto, consideraremos ambos términos intercambiables). Cada hilo, o proceso, tiene un estado de memoria definido por el contenido de los registros arquitectónicos y el contenido de la memoria que puede direccionar, que puede ser de los siguientes tipos:

- Memoria de sólo lectura.
- Memoria de lectura y escritura asignada estáticamente (i.e. antes de ejecución).
- Memoria de lectura y escritura asignada dinámicamente (i.e. durante ejecución).
- Pila de llamadas.

La memoria que un proceso puede direccionar/acceder puede variar a lo largo de su ejecución. No obstante, la pila de llamadas siempre estará presente. Esta zona de memoria se distingue de las demás zonas de lectura y escritura por su importancia y especificidad.

Según el estándar ATPCS la pila debe ser de tipo *Full Descending*, lo que quiere decir que el puntero de pila apunta a la última posición ocupada de la pila, y que la pila crece hacia direcciones menos significativas, tal y como ilustrada la figura 2.2. La pila es una zona contigua de memoria dentro de la región comprendida entre la base (*stack base*) y el límite (*stack limit*). La zona ocupada de la pila está comprendida en el intervalo [*stack pointer*, *stack base*], mientras que la parte libre se define por el intervalo [*stack limit*, *stack pointer*). Es preciso aclarar, que no es estrictamente necesario que la pila ocupe la misma región de memoria en todo instante, y, que los valores de *stack base* y *stack limit* no tienen por qué están disponibles para el propio proceso.

En caso de haber varios procesos en ejecución, el estándar establece que un proceso no puede modificar la memoria de lectura/escritura de otro proceso ni la parte ocupada de su pila. Sin embargo, permite que la parte libre de la pila pueda ser escrita inesperadamente, previsiblemente debido a una rutina de tratamiento de excepciones.

2.3.2. Registros

La Tabla 2.1 muestra los registros visibles en modo usuario y describe su uso habitual definido por el estándar ATPCS.

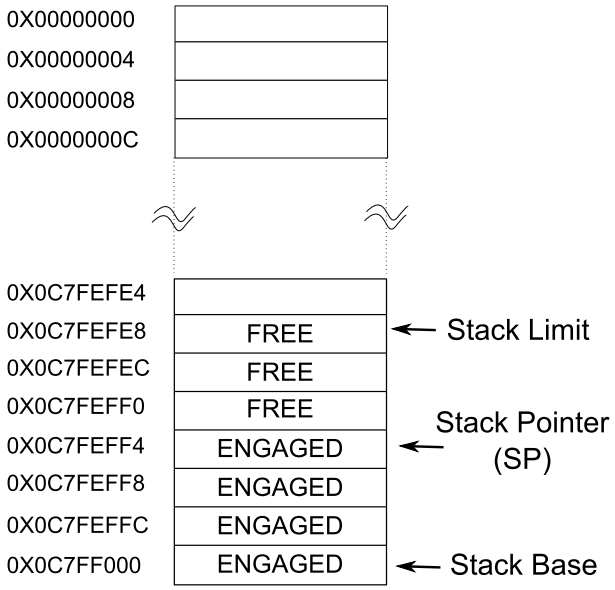

Figura 2.2: Ilustración de una pila *Full Descending*.

Los registros r0-r3 se utilizan para pasar parámetros a la rutina, para recoger el valor devuelto por está y para almacenar resultados temporales durante su ejecución. El registro r12, también llamado IP, se utiliza también para almacenar resultados temporales. Los registros r4-r11 se utilizan típicamente para almacenar el valor de variables locales de la rutina, por lo que también reciben los nombres v1-v8. Los registros r12-r15 juegan un papel especial, por el que reciben su nombre alternativo: IP, SP, LR y PC respectivamente. En algunas variantes del ATPCS los registros r9 y r10 también ejercen roles especiales, recibiendo en estos casos los nombres de SB y SL respectivamente. En el estándar básico, las rutinas deben preservar los valores de los registros r4-r11 y r13 (SP), pero no tienen obligación de preservar los de los registros r0-r3, r12 (IP) y r14 (LR).

Aunque también existen registros de punto flotante, por el momento los ignoraremos. Para más información sobre ellos es preciso consultar el documento de referencia del estándar [atp, arm].

2.3.3. Llamada a subrutina

La llamada a una rutina puede hacerse con cualquier secuencia de instrucciones que lleve al registro LR (r14) la dirección de retorno y al PC (r15) la dirección de comienzo de la rutina. Las dos principales alternativas son[1]:

Convencional Utilizando la instrucción BL que permite realizar un salto relativo al PC:

```
BL FUNC     @ La dirección se traduce en ensamblado por FUNC - (PC + 8)
            @ BL guarda en LR el valor PC - 4
```

[1]Es preciso recordar que, debido a la segmentación, cuando una instrucción lee el registro PC, este contiene la dirección de la propia instrucción incrementada en 8 bytes (i.e. dos inst.).

Tabla 2.1: Descripción del uso de los registros arquitectónicos según el estándar ATPCS.

Registro	Sinónimo	Especial	Descripción
r0	a1		Argumento 1/Resultado/Scratch
r1	a2		Argumento 2/Resultado/Scratch
r2	a3		Argumento 3/Resultado/Scratch
r3	a4		Argumento 4/Resultado/Scratch
r4	v1		Variable 1.
r5	v2		Variable 2.
r6	v3		Variable 3.
r7	v4		Variable 4.
r8	v5		Variable 5.
r9	v6	SB	Variable 6. Puntero base utilizado con bibliotecas dinámicas.
r10	v7	SL	Variable 7. Puntero límite de pila
r11	v8	FP	Variable 8. Puntero marco de pila (*Frame Pointer*).
r12		IP	Registro auxiliar utilizado en las llamadas a subrutina (*Intra-Procedure scratch*).
r13		SP	Puntero de pila (*Stack Pointer*).
r14		LR	Registro de enlace (*Link Register*) utilizado para almacenar la dirección de retorno en un salto a subrutina.
r15		PC	Contador de programa.
		CPSR	Registro de estado actual (*Current Program Status Register*).

Alternativa Utilizando una secuencia de instrucciones:

```
LDR r4, =FUNC     @ La dirección FUNC se resuelve al enlazar
MOV LR, PC        @ LR almacena el valor actual del PC (. + 8)
MOV PC, r4        @ En PC se carga la dirección FUNC
```

De las dos alternativas anteriores, solamente la segunda permite hacer un salto a una rutina ubicada en una sección distinta, ya que no requiere el cálculo de un desplazamiento relativo al PC. Este cálculo se lleva a cabo durante el ensamblado, cuando el emplazamiento de la secciones está aún por determinar. Del mismo modo, si queremos saltar a una rutina ensamblada (o compilada) por separado sólo podremos utilizar el segundo método.

2.3.4. Paso de parámetros

La forma de pasar los parámetros a una rutina se define en términos de los tipos fundamentales de la arquitectura (i.e. tipos de los registros arquitectónicos):

1. Enteros de 32 bits.
2. Punto flotante de simple precisión.

3. Punto flotante de doble precisión.

Cualquier parámetro de un lenguaje fuente que no se corresponda con un tipo fundamental se convierte a uno o más valores de tipo fundamental. Por ejemplo, en el caso de los enteros de menos de 32 bits, se les hace una extensión a 32 bits conservando el signo, y si son de 64 bits se tratan como dos de 32 bits. Para más información sobre la conversión consultad el documento de referencia del estándar [atp].

Hay dos esquemas distintos para el paso de parámetros, dependiendo de si la rutina acepta un número variable o fijo de parámetros. Nosotros estudiaremos solamente el caso de un número fijo de parámetros de tipo entero. Para información sobre el caso variable y/o sobre parámetros de punto flotante es preciso consultar el documento de referencia del estándar [atp].

El paso de parámetros a una rutina con un número fijo de parámetros enteros se realiza según las siguientes reglas:

- Los primeros 4 valores enteros se asignan a los registros a1-a4 (r0-r3). Si hay menos de cuatro, se ocupan en orden (a1-a3, a1-a2 o a1).
- Los valores restantes se insertan en la pila en orden inverso a como aparecen en la lista de parámetros de la rutina. De este modo se pueden sacar de la pila en orden.

Recapitulando, siguiendo el estándar ATPCS, para realizar correctamente una llamada a una rutina con un número fijo de parámetros enteros, debemos copiar el valor de los cuatro primeros parámetros en los registros a1-a4, en orden, y si hay más parámetros debemos meterlos en la pila en orden inverso. Después hacemos la llamada con BL o con una secuencia de instrucciones como se explicó en la sección 2.3.3.

2.3.5. Valor devuelto

Los procedimientos no devuelven ningún valor. En cambio, las funciones devuelven un único valor de tipo entero o flotante.

Una función de tipo entero puede devolver:

- Un valor entero de una palabra en a1 (r0).
- Un valor entero de 2-4 palabras en a1-a2, a1-a3 o a1-a4 respectivamente.
- Un valor entero más grande, pero ha de hacerlo indirectamente, mediante una variable alojada en memoria, cuya dirección se haya pasado como parámetro de la rutina.

Para información sobre las funciones de punto flotante es preciso consultar el documento de referencia del estándar [atp].

2.3.6. Estructura de una rutina

Gracias a la información descrita en las secciones anteriores podemos invocar correctamente cualquier rutina que haya sido diseñada siguiendo el estándar ATPCS. Asimismo,

sabemos lo que debe respetar para que satisfaga el estándar y por lo tanto podríamos implementar cualquier rutina para que fuese invocada por otro código. No obstante, el estándar contempla además una serie de reglas que determinan la estructura que deben tener los marcos de activación para que los depuradores o cualquier código que tenga que recorrer la lista de llamadas en sentido inverso, como los manejadores de excepciones de C++, puedan hacerlo. Este proceso se conoce como desenrollado de pila (*stack unwinding* o *back trace*).

Esta gestión del marco de pila implica además que el código de toda rutina debe dividirse en tres partes:

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

El código de entrada prepara el marco de activación, guardando en la pila los registros cuyo valor debe conservarse (ver sección 2.3.2) y reservando espacio espacio en esta última para almacenar las variables locales de la rutina, asignándoles un valor inicial si procede. El código de salida restaura el estado de la rutina invocante, recuperando el valor de los registros. Es preciso tener en cuenta que existe la posibilidad de que una rutina pueda tener varios puntos de entrada (por ejemplo en C++ con funciones que admiten parámetros con valores por defecto) y varios puntos de salida. En tal caso, es necesario replicar el prólogo/epílogo según convenga.

El estándar plantea dos alternativas para la construcción de los marcos de activación:

- Marcos de activación de tamaño fijo, que obedecen la condición *stack-moves-once*.
- Marcos de activación con puntero de marco o *frame pointer*.

La primera alternativa exige que el valor de SP en el cuerpo de la rutina tenga el mismo valor que a la salida del código de entrada (prólogo). La segunda alternativa exige que como *frame pointer* se emplee un registro que se conserve en llamadas a rutinas, generalmente el registro r11 (FP). El compilador *gcc* para ARM, que es el que emplearemos en el laboratorio, aunque mantiene constante el puntero de pila, genera código basado *frame pointer* y por eso nos centraremos en está alternativa.

El desenrollado de la pila empleando punteros de marco se realiza recorriendo una lista enlazada de estructuras de *backtrace*. La información contenida en estas estructuras es la siguiente (en orden ascendente de dirección de memoria):

```
dir baja ----> Registros a preservar (r10-14)
               Valor de FP de retorno          [fp, #-12]
               Valor de SP de retorno          [fp, #-8]
               Dirección de retorno (LR)       [fp, #-4]
dir alta ----> PC guardado                     [fp]
```

El *frame pointer* (FP) debe apuntar a la estructura de *backtrace* de la rutina que está actualmente en curso, en concreto a la dirección más alta de la misma. El valor del FP de

retorno (almacenado en la estructura) debe ser cero o apuntar al comienzo de la estructura de la rutina que la invocó, y así sucesivamente. Por ejemplo, supongamos el siguiente código:

```
#include <stdio.h>

void one(void);
void two(void);
void zero(void);

int main(void)
{
   one();
   return 0;
}

void one(void)
{
   zero();
   two();
   return;
}

void two(void)
{
   printf("main...one...two\n");
   return;
}

void zero(void)
{
   return;
}
```

La Figura 2.3 ilustra como quedan enlazados los marcos de activación a través de las estructuras de *backtrace*, en el momento en el que la rutina `two` escribe el mensaje por pantalla.

El marco de activación completo, incluyendo la estructura de *backtrace*, está descrito en la figura 2.4. Este marco se puede construir mediante la siguiente secuencia de código (prólogo):

```
MOV     IP, SP                          @ Guardar valor de retorno de SP
STMDB   SP!, {r4-r10,FP,IP,LR,PC}       @ Guadar registros en la pila
SUB     FP, IP, #4                      @ Actualizar FP
SUB     SP, #SpaceForLocalVaribles      @ Reservar espacio
```

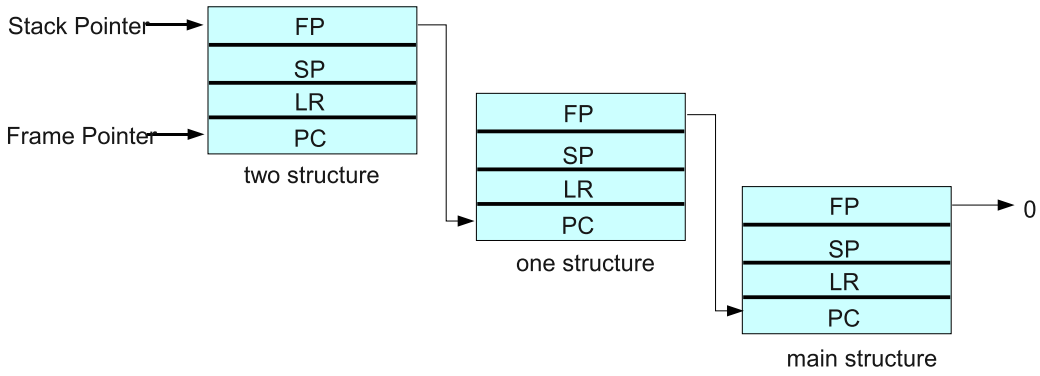

Figura 2.3: Lista enlazada de estructuras de *backtrace*.

Para deshacer el marco de activación, devolviendo correctamente el control a la dirección de retorno, podemos utilizar un código como el siguiente (epílogo):

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

El valor copiado en PC es el valor que tenía LR al entrar en la función, que según el estándar es la dirección de retorno. En FP y SP se copian los valores guardados al entrar en la rutina, por lo que el estado al volver de la rutina es el mismo que tenía la rutina invocante antes de la llamada. Por otra parte, el valor de PC almacenado en la estructura de *backtrace* permite localizar el punto por el que se entró a la rutina, información que puede ser muy útil en depuración.

El código generado por *gcc* evita modificar SP en el cuerpo de la rutina. Para lograrlo, reserva en la pila el espacio necesario para pasar los argumentos a las funciones invocadas en el cuerpo de la rutina.

2.4. 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 2.4. Este espacio es reservado por el código de entrada de la función (prólogo) y es liberado por el código de salida (epílogo).

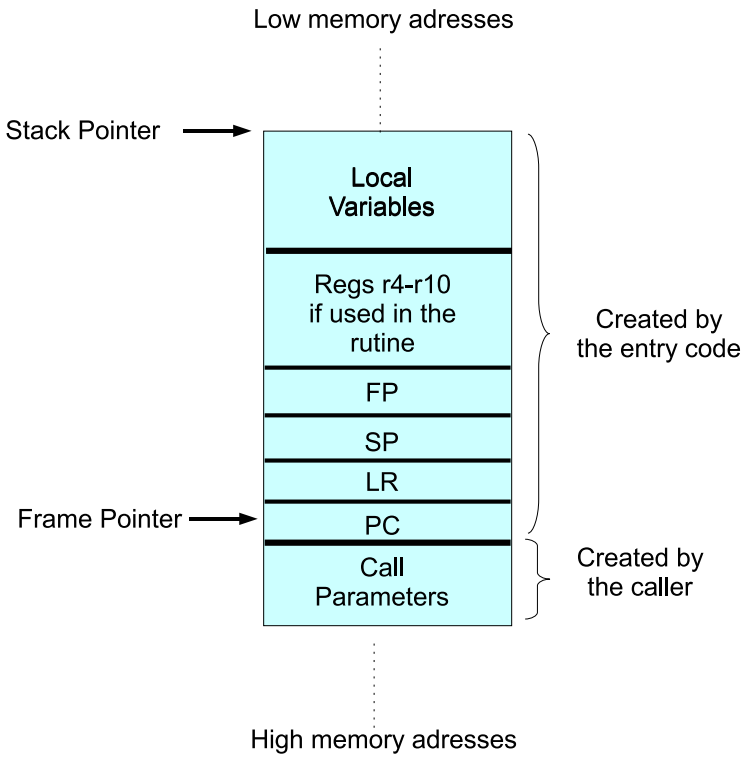

Figura 2.4: Contenido del marco de activación incluyendo la estructura para *backtrace*.

2.4.1. Símbolos globales

Durante el proceso de enlazado es preciso resolver los símbolos (variables o funciones) definidos en otros ficheros. A estos símbolos también se les denomina globales y es preciso identificarlos de manera explícita en el código fuente.

Cuando se declara una variable global en un fichero C y se quiere que sea accesible desde otro fichero con el que vamos a enlazar, debemos exportarla. En C, esto se realiza declarando la variable con el modificador `extern`. Con las funciones sucede lo contrario, por defecto los nombres de función son símbolos globales. Si se quiere restringir la visibilidad de una función al fichero donde ha sido declarada, es necesario utilizar el modificador `static` en su declaración.

En ensamblador los símbolos globales debemos también exportarlos de forma explícita utilizando la directiva `.global` en su definición. Esto es válido para cualquier símbolo. Por ejemplo, el símbolo `start`, que como vimos en la práctica anterior es especial e indica el punto 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. En ensamblador, cuando queramos hacer referencia a un símbolo definido en otro fichero utilizaremos la directiva `extern` en cada declaración local.

Interacción entre 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. Por ejemplo, si queremos usar una rutina `FOO`, sin parámetros de entrada ni valor de retorno, deberemos tener la siguiente declaración adelantada para que el compilador sepa generar el código de llamada:

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

Cuando lo que se usa es una variable global, deberemos hacer una declaración `extern` de un puntero. Así, si en un fichero ensamblador exportamos la etiqueta `TABLE`, en el fichero C dónde queramos utilizarla deberemos hacer la siguiente declaración:

```
extern char* TABLE;
```

2.5. 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. En C++ tenemos además los objetos, pero por ahora no vamos a considerarlos.

2.5.1. 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 2.5. 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`).

Dependiendo de la arquitectura puede haber restricciones de alineamiento en el acceso a memoria. Este es el caso de la arquitectura ARM. Como vimos en la práctica anterior, en esta arquitectura (al menos en la versión v4T) los accesos deben realizarse a direcciones alineadas con el tamaño del acceso. En este tipo de arquitecturas, 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. Esto hace que la dirección de comienzo de un array no pueda ser cualquiera, sino que debe ser una dirección

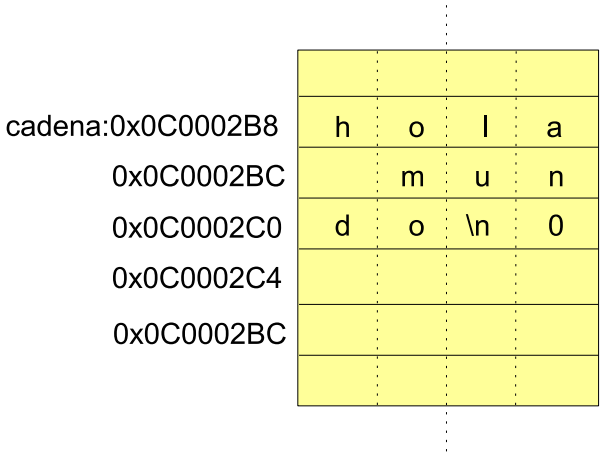

Figura 2.5: Almacenamiento de un array de caracteres en memoria.

que satisfaga las restricciones de alineamiento, en función del tipo de datos almacenados en el array. El compilador por tanto seleccionará una dirección de comienzo para el array que satisfaga estas restricciones.

2.5.2. 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 2.6.

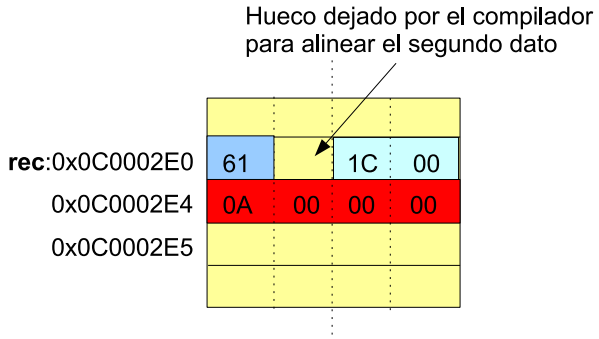

Figura 2.6: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a 0x61, 0x1C y 0x0A respectivamente.

2.5.3. 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 2.5.2. Sin embargo su huella de memoria (*memory footprint*) es muy distinta, como ilustra la figura 2.7.

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.

2.6. Desarrollo de la práctica

La práctica está organizada en dos partes, una primera guiada en la que iremos viendo con ejemplos todo lo que hemos explicado anteriormente y una segunda en la que se propondrán al alumno ejercicios a resolver. La intención es que el alumno afiance los co-

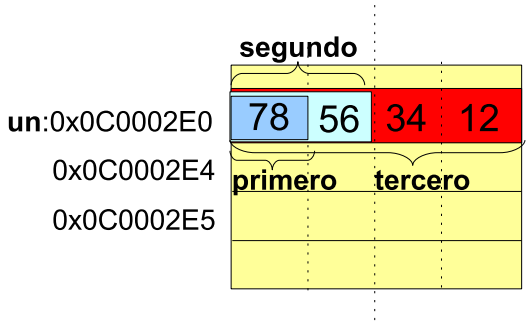

Figura 2.7: Almacenamiento de la union *un*, con los campos primero, segundo y tercero. Un acceso al campo primero da como resultado el byte *0x78*, es decir el carácter *x*. Un acceso al campo segundo da como resultado la media palabra *0x5678* (asumiendo configuración *little endian*), es decir el entero *22136*. Finalmente un acceso al campo tercero nos da como resultado la palabra *0x12345678* (de nuevo *little endian*), es decir el entero *12345678*.

nocimientos teóricos en la primera parte y sea capaz de demostrarlos en la segunda. A lo largo del guión se irán planteando algunas preguntas a los alumnos para que las respondan.

2.6.1. Parte guiada

Los pasos que debemos ir realizando en esta parte son:

1. Abrir el Embest IDE y crear un workspace nuevo.
2. Al workspace le añadimos como en la práctica anterior una carpeta common.
3. Crear un nuevo fichero y guardarlo con el nombre `ld_script.ld`. Añadirlo a la carpeta common. Este será el fichero de entrada al enlazador, que determinará las secciones de nuestro ejecutable y su ubicación.
4. Copiar en el fichero ld_script.ld el siguiente contenido:

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

Como podemos ver se definen tres secciones con los nombres habituales para código, datos con valor inicial, datos de sólo lectura y datos sin valor inicial. Además se

definen unos símbolos que nos permitirán luego conocer las posiciones inicial y final de cada una de estas secciones.

5. 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.

6. Añadir el siguiente contenido al fichero:

```
.global start

.equ STACK, 0x0c7ff000    @ Valor inicial para el puntero de pila

.text
start:
    LDR sp,=STACK
    MOV fp,#0

.extern Main

    ldr r0,=Main
    mov lr,pc
    mov pc,r0

End:
    B End
.end
```

   Como podemos ver, el programa primero inicializa SP y FP. Luego realiza un salto a la rutina `Main`, punto de entrada al programa escrito en C. El símbolo está definido en otro fichero fuente por lo que se declara como `extern`. Cuando termina la ejecución de la rutina el programa se queda en un bucle infinito.

   Responded a las siguientes preguntas:

   - ¿Por qué se inicializa FP a 0?
   - ¿Es necesario que el salto sea mediante macro de LDR? ¿Por qué?

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

```
//hacemos visibles los símbolos creados en
//el script de configuración del ld
extern char _bdata[];
extern char _edata[];
extern char _bbss[];
```

```
extern char _ebss[];

// variables globales
char * inidata = (char *) _bdata;
char * enddata = (char *) _edata;
char * inibss  = (char *) _bbss;
char * endbss  = (char *) _ebss;

int Res;

// Funciones

int sum2(int a1, int a2)
{
    int sum_local;

    sum_local = a1 + a2;

    return sum_local;
}


// Función principal
Main(void)
{
    int res;

    res = sum2(1,2);
    Res = res;
}
```

Como vemos el programa declara una variable global `Res` y una variable local `res`, ejecuta la función `sum2` con parámetros 1 y 2, guarda el resultado en `res` y finalmente copia este valor en `Res`.

8. Configurar el proyecto tal y como se hizo en la práctica anterior.

9. Compilar el proyecto y crear el ejecutable.

10. Conectarse a la placa. Si el procesador está corriendo (botón de stop en rojo) lo paramos pulsando el botón de stop.

11. Bajamos el programa a la placa y lo ejecutamos paso a paso analizando lo que sucede.

12. Responder razonadamente a las siguientes preguntas:

    - ¿Cuál es la dirección de la variable Res? ¿Se almacena esta dirección en algún sitio? ¿En qué región de memoria está Res?

- ¿Cuál es la dirección de la variable res? ¿En dónde se almacena? ¿Cómo se accede a ella en el código ensamblador?
- ¿Se almacenan en algún lado los valores 1 y 2 que se pasan como parámetros a la función sum2? ¿Cómo se pasan?
- En sum2, ¿dónde se almacena la variable local sum_local? ¿Y las variables a1 y a2? Indicar sus direcciones.
- ¿Cómo se devuelve el resultado de la suma?

13. Añadir la siguiente función al programa:

```
int sum7(int a1, int a2, int a3, int a4,int a5, int a6, int a7)
{
    int sum_local;

    sum_local = sum2(a1,a2) + sum2(a3,a4) + sum2( sum2(a5,a6),  a7);

    return sum_local;
}
```

y modificar la función Main de la siguiente manera:

```
Main(void)
{
    int res;

    res = sum2(1,2);
    Res = sum7(res,3,4,5,6,7,8);
}
```

14. Compilar y bajar de nuevo el programa a la placa.

15. Responder razonadamente a las siguientes preguntas:

- ¿Cómo se pasa cada uno de los argumentos a sum7? Detallad la respuesta.
- ¿Qué dirección de memoria tienen asignadas las variables locales a1-a7 en la función sum7?

16. Describir el contenido de la pila cuando para sumar los argumentos a1 y a2 se entra en sum2. Detallar el valor de las estructuras de *backtrace* en esta situación.

17. Añadir las siguientes variables globales al programa, justo debajo de Res:

```
char cadena[12] = "hola mundo\n";
char cadena2[12] ;

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

```
    int tercero;
};
struct mistruct rec;
char michar = 'a';
```

y añadir al final de la función Main el siguiente código:

```
    rec.primero = michar;
    rec.tercero = Res;
    rec.segundo = (short int) res;

for( i = 0; i < 12 ; i++ )
cadena2[i] = cadena[i];
```

18. Compilar el programa y descargarlo a la placa. Ejecutarlo paso a paso. Poner un *watch* sobre las variables cadena, cadena2 y rec.

19. Responder razonadamente a las siguientes preguntas:

    - ¿En qué región de memoria están almacenadas cadena, cadena2, rec, _bdata, _edata, _bbss y _ebss? ¿Cuales son sus direcciones? ¿Qué direcciones abarca cada región?
    - ¿Qué direcciones ocupa el array cadena? ¿y el array cadena2?
    - ¿Qué direcciones ocupa cada uno de los campos de rec? ¿Hay alguna separación entre campos?
    - Convertir la estructura en una unión. ¿Qué direcciones ocupa cada campo? ¿Cuál es el valor final almacenado en la union?

2.6.2. Parte no guiada

En este apartado partimos de un proyecto, escrito en su mayoría en C, que implementa una pila estática.

Como de costumbre tendremos un programa `init.s` que se encarga de inicializar convenientemente el estado del proceso para la ejecución de nuestro código C. Además de esto, el fichero contiene el código de una rutina `FOO`, que es invocada desde la función `Main`:

```
#Program entry
.global start

.text
start:
    LDR sp,=0x0C7ff000
    MOV fp,#0

.extern Main
    ldr r0,=Main
    mov lr,pc
```

```
    mov pc, r0

End:
    B End
.extern Vacia
.global FOO

FOO:
    mov ip, sp
    stmfd sp!, {fp,ip,lr,pc}
    sub fp, ip, #4
    sub sp, sp, #4

    str r0, [fp,#-16]
    ldr r2,=Vacia
    mov lr,pc
    mov pc,r2
    cmp r0, #0
    bne RETURN

    ldr r0, [fp,#-16]
    add r1, r0, #4
    bic r1, r1, #3
    ldrb r2, [r0]
    sub r2, r2, #1
    ldr r0,=screen
LOOP:
    cmp r2, #0
    blt RETURN
    ldr r3, [r1, r2, lsl #2]

    str r3, [r0], #4
    sub r2, r2, #1
    b   LOOP

RETURN:
    ldmdb fp, {fp,sp,pc}

.global screen
screen: .space 1024

.end
```

La función `Main`, inicializa una variable global `P` que es de tipo `PilaInt`. El tipo se define en el fichero pila.h y las funciones que operan sobre esta pila se definen en el fichero pila.c.

- El fichero main.c contiene:

```
#include "pila.h"

PilaInt P;

void FOO(PilaInt* P);

int Main(void)
{
    int i;

    Inicializar( &P );

    for(i = 0 ; i < 10 ; i++ ) {
        Insertar( &P, i );
    }

    FOO( &P );

    while( !Vacia( &P ) ) {
        Extraer( &P, &i );
    }

    return 0;
}
```

- El fichero pila.h contiene:

```
#define N 255

typedef struct {
    unsigned char num;
    int Buffer[N];
} PilaInt;

void Inicializar( PilaInt* P );

int Llena( PilaInt* P );

int Vacia( PilaInt* P );

int Insertar( PilaInt* P, int elem );

int Extraer( PilaInt* P, int* elem );
```

- y el fichero pila.c:

```
#include "pila.h"

void Inicializar( PilaInt* P )
{
    (*P).num = 0;
}

int Llena( PilaInt* P )
{
    return (*P).num == N;
}

int Vacia( PilaInt* P )
{
    return (*P).num == 0;
}

int Insertar( PilaInt* P, int elem )
{
    if( Llena( P ) )
        return -1; // error

    (*P).Buffer[ (*P).num ] = elem;
    (*P).num++;

    return 0;  // no hay error
}

int Extraer( PilaInt* P, int* elem )
{
    if( Vacia( P ) )
        return -1; // error

    (*P).num--;
    *(elem) = (*P).Buffer[ (*P).num ];

    return 0;
}
```

Se pide al alumno:

1. Reemplazar la función Insertar por una rutina codificada en ensamblador, siguiendo el estándar ATPCS y modificar la función `Main` para que invoque a esta rutina.

2. Obtener una función C equivalente a `FOO`, modificando la función `Main` para que invoque a esta nueva función C (sólo cambia el nombre de la función en la llamada).

Bibliografía

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

[atp] The arm-thumb procedure call standard. Accesible en http://www.cs.cornell.edu/courses/cs414/2001fa/armcallconvention.pdf. Hay una copia en el campus virtual.