Práctica 1. Lenguajes de alto nivel y ensamblador

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:

- 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*.

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. La Pila de llamadas

Como hemos visto en el capítulo de introducción, 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. El ejemplo de la figura 1.1 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 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.

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), y
- parámetros que tenga que pasar a otra subrutina que invoque.

La figura 1.2 muestra un esquema general de la organización del marco de activación de una subrutina. 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

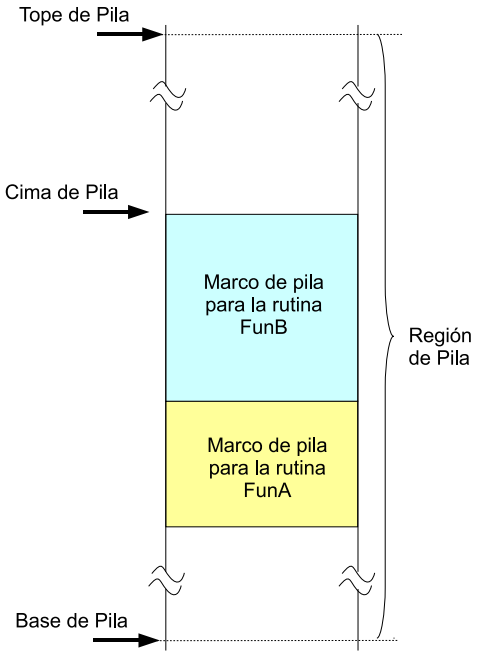

Figura 1.1: Ejemplo del estado de la pila de llamadas en el caso de que una función FunA haya invocado la rutina FunB y ésta última se encuentre en ejecución.

del marco. Generalmente el *FP* separa los parámetros de la llamada del resto de la información contenida en el marco. En este caso, direccionaríamos los parámetros de llamada con desplazamientos relativos a *FP* de un signo, y las variables locales con desplazamientos del signo contrario.

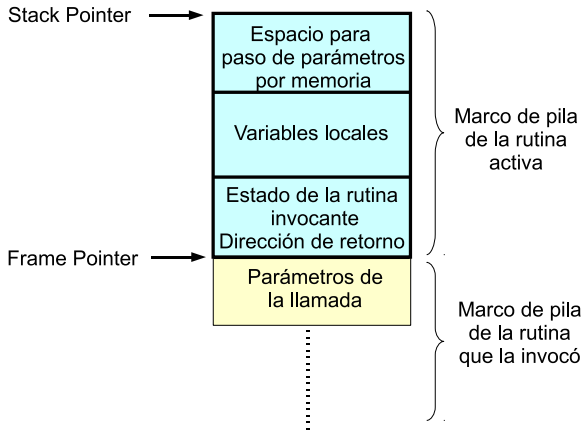

Figura 1.2: Estructura general de los marcos de activación o marcos de pila.

La estructura concreta del marco de activación se fija por convenio (estándar) para la arquitectura destino. El establecimiento de este estándar permite que códigos escritos en distintos lenguajes y compilados por separado puedan ser luego enlazados en un ejecutable final. El objetivo de la siguiente sección es conocer el estándar para la arquitectura ARM.

1.3. 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:

- El uso de registros arquitectónicos
- Un modelo de proceso, memoria y pila de llamadas
- El mecanismo de llamada a subrutina o función.
- Valor de retorno.
- Paso de parámetros

Nosotros vamos a estudiar el estándar básico, sin tener en cuenta las particularidades relacionadas con el uso de coprocesadores, extensiones SIMD o instrucciones en punto flotante. La descripción completa del estándar está en [aap].

1.3.1. Registros

Como vimos en la práctica anterior, la arquitectura ARM tiene 16 registros de 32 bits nombrados R0-R15 o r0-r15. El AAPCS otorga un papel especial a estos registros en el proceso de llamada a subrutina, asignándoles un nombre alternativo (sinónimo o alias) relacionado con dicho papel. Estos usos y nombres se dan en la Tabla 1.1.

Los primeros cuatro registros (R0-R3 o a1-a4) se utilizan para pasar parámetros a la rutina. R0 además se utiliza para recoger el valor devuelto por una función. Además, estos registros pueden ser utilizados dentro de una rutina para almacenar resultados temporales durante su ejecución.

El registro R12, también llamado IP, se utiliza para almacenar valores temporales entre llamadas. Por ejemplo, veremos como utilizarlo para componer el marco de pila de la rutina.

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.

Como comprobaremos más adelante, los registros R11-R14 juegan un papel especial en el mecanismo de llamadas a subrutina, por el que reciben sus nombres alternativos (FP, IP, SP y LR).

El estándar AAPCS básico impone que **cualquier subrutina debe preservar los contenidos de los registros R4-R11 y R13 (SP)**. Sin embargo no tiene obligación de preservar el resto de los registros: R0-R3, R12 (IP), R14 (LR) y R15 (PC), por lo que si alguno de estos registros contiene un valor que deba ser preservado, es responsabilidad de la rutina invocante copiarlo en un lugar seguro.

Tabla 1.1: Descripción del uso de los registros arquitectónicos según el estándar AAPCS.

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*).

1.3.2. Modelo de memoria y pila

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

Según el estándar, la memoria direccionable por un proceso se clasifica normalmente en las siguientes cinco categorías, algunas de las cuales hemos visto ya:

- Región de código. Debe poderse leer pero puede no ser accesible para escritura.
- Región de datos estáticos de sólo lectura. Suele contener variables globales constantes.
- Región de datos estáticos de lectura y escritura. Suele contener variables globales.
- El montículo o *heap*.
- La pila o *stack*.

La pila es la única región que debe ser continua, el resto puede estar fragmentada. El *heap* debe ser gestionado por el propio proceso (por ejemplo, como en C, a través de

las fuciones de la biblioteca estándar `malloc` y `free`), y suele utilizarse para la creación dinámica de objetos.

Cualquier programa que cumpla el estándar debe ejecutar únicamente instrucciones contenidas en regiones de código.

La Pila

La pila es una región continua de memoria que se utiliza para almacenar variables locales y los parámetros de llamada, además de para salvar los registros de la rutina invocante que deben ser preservados.

Según el AAPCS la pila debe ser *full-descending*, utilizando el registro R13 (SP) para almacenar la dirección de la cima de la pila. La figura 1.3 ilustra el concepto de pila *full-descendig*. En general la pila tendrá una base y un límite. En todo momento deben cumplirse las siguientes restricciones:

- Límite $<$ SP $<=$ Base (recordemos que crece hacia direcciones pequeñas).
- SP $mod\ 4 = 0$. La pila debe estar alineada en una dirección de palabra.
- Un proceso sólo puede direccionar la región de la pila en el intervalo $[\text{SP}, \text{Base} - 1]$

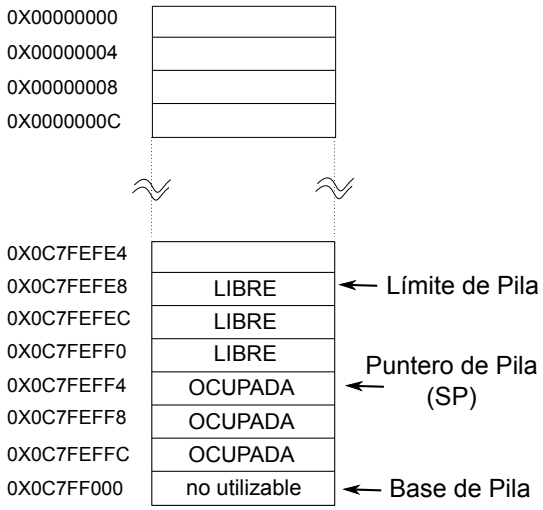

Figura 1.3: Ilustración de una pila *Full Descending*.

1.3.3. Llamadas 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][2]

[1] Consultar el manual del AAPCS para ver los detalles en el cambio a modo Thumb en una llamada a subrutina.

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

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

```
BL FUNC     @ El desplazamiento del branch se traduce por:
            @   FUNC - PC (. + 8)
            @ BL guarda en LR el valor PC - 4
```

- **Alternativa**: Utilizando una secuencia de instrucciones:

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

La primera de las alternativas tiene un problema. Cuando el destino del salto (comienzo de la rutina) está en otra sección de código, el desplazamiento relativo a PC no se puede calcular hasta el momento de enlazado. Si se utiliza, el ensamblador codifica el salto con un desplazamiento 0, y añade información en la tabla de símbolos para que el enlazador lo pueda modificar posteriormente. Sin embargo, si el enlazador determina que el desplazamiento no puede codificarse con los 24 bits disponibles en la instrucción `BL` [arm] nos dará un error de enlazado.

La segunda alternativa no presenta este problema, ya que en este caso el enlazador escribirá la dirección destino del salto en memoria (32 bits) y el ensamblador codificará para el load un acceso relativo a PC a la entrada correspondiente de la tabla de literales. Por este motivo, es preferible emplear esta alternativa cuando el salto haga referencia a una rutina definida en otra sección.

1.3.4. Valor de retorno

Para aquellas subrutinas que devuelven un valor (funciones), el AAPCS especifica el mecanismo de retorno de la siguiente forma:

- Un valor de un tipo fundamental de ancho menor a una palabra se extiende con ceros o con el bit de signo hasta los 32 bits y se devuelve en R0.
- Un valor de un tipo fundamental de tamaño palabra se devuelve en R0.
- Un valor de tipo fundamental de 2-4 palabras se devuelve en R0-R3.
- Un valor compuesto de hasta 4 bytes se devuelve en R0. El formato es como si se hubiese copiado en memoria en una dirección alineada a tamaño palabra y esta se hubiese después cargado en R0 con una instrucción LDR.
- Un valor compuesto de más de 4 bytes (o cuyo valor no se conoce en tiempo de compilación), se copia en memoria en una dirección pasada como un argumento extra en la llamada a la subrutina. Este parámetro extra se pasará como primer parámetro a la rutina.

1.3.5. Paso de parámetros

Para realizar el paso de parámetros a una subrutina, primero debe establecerse una conversión entre los tipos de datos de los lenguajes de alto nivel y los tipos fundamentales soportados por la arquitectura. Esta conversión es dependiente del lenguaje. Por ejemplo, en el caso de C a los enteros de menos de 32 bits (short int) se les hace una extensión de signo a 32 bits, y en el caso de enteros de 64 bits (long long int) se tratan como dos palabras de 32 bits. Para el caso de C, C++ puede encontrarse un relación completa en [aap].

Una vez realizada la conversión de valores anterior, obtenemos una lista de parámetros de tamaño palabra que debemos pasar a la subrutina. El AAPCS impone el siguiente mecanismo:

- Si la rutina es una función que devuelve un valor en memoria (último item de la lista de la sección 1.3.4):
  - La dirección donde debe almacenarse ese valor se debe pasar en R0.
  - Los tres primeros parámetros se pasan por registro, utilizando en orden los registros R1-R3.
- En otro caso, los cuatro primeros parámetros se pasan por registro, utilizando en orden los registros R0-R3.
- Si hay más parámetros, estos deben pasarse por memoria, escribiéndolos en la pila en el orden siguiente:
  - el primero de los parámetros restantes en $[SP]$,
  - el siguiente en $[SP + 4]$,
  - el siguiente en $[SP + 8]$,
  - ...

  Si se insertasen en la pila con instrucciones tipo *push* (`STR Rf, [SP,-#4]!` o `STMFD SP!, {Rf}`), se haría en orden inverso. Sin embargo, como el estándar impone que el tamaño del marco se fije desde el comienzo de la rutina, para adherirse al estándar la rutina invocante debe insertar los parámetros sin modificar su puntero de pila, por ejemplo con una secuencia similar a esta:

```
LDR R0, =Param5
STR R0, [SP]
LDR R0, =Param6
STR R0, [SP,#4]
LDR R0, =Param7
STR R0, [SP,#8]
...
```

  dejando los parámetros en la pila como si se hubiesen insertado en orden inverso.

La figura 1.4 ilustra el mecanismo de paso de parámetros especificado en el AAPCS para nuestro ejemplo, cuando FunA debe invocar la rutina FunB, pasándole 7 parámetros que llamamos Param1-7. En la figura podemos ver que los cuatro primeros parámetros se pasan por registro (registros R0-R3), mientras que los últimos tres se pasan a través de la pila. FunA escribe estos parámetros en la cima de su propio marco de pila. Es decir, cuándo comienza la ejecución de FunB el valor de SP tendrá la dirección de Param5. FunB construirá su marco de pila reservando el espacio que necesite para: el paso de parámetros a las subrutinas que invoque, almacenar sus variables locales y para salvar el contenido de los registros arquitectónicos que necesite, con el fin de poder restaurar el estado de FunA al finalizar su ejecución.

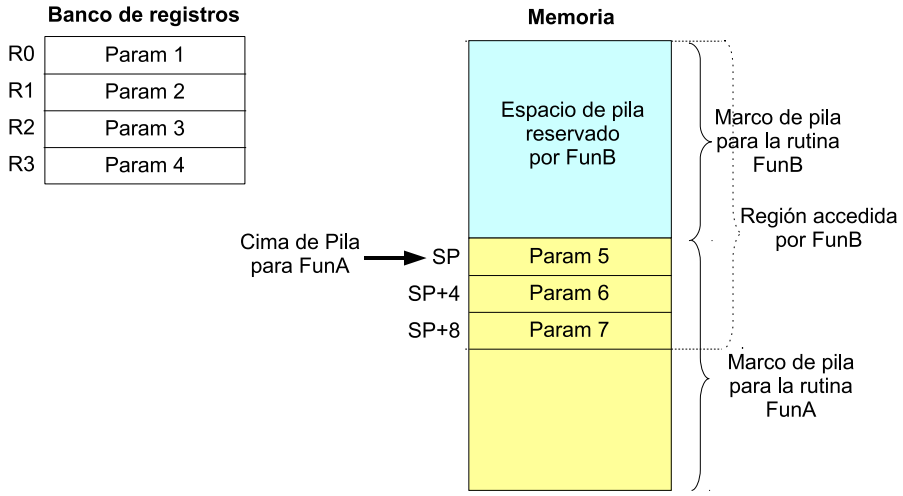

Figura 1.4: Paso de los parámetros Param1-7 a FunB desde FunA.

Debemos notar un par de cosas:

- Cuando una rutina completa la construcción de su marco de activación, debe reservar espacio suficiente para escribir en la cima el valor de los parámetros de las llamadas a subrutina que vaya a realizar. Por ejemplo, en la figura 1.4 la rutina FunA había dimensionado su marco de activación teniendo en cuenta que debía pasar tres parámetros por pila cuando fuese a llamar a la rutina FunB. Si va a invocar a más de una subrutina tendrá que evaluar el número máximo de parámetros que debe pasar por pila.

- Aunque en general una rutina accede sólo a su marco de activación, para leer los parámetros de la llamada (con los que inicializará sus variables locales) debe acceder también a la cima del marco de activación de la rutina que la invocó (desplazamientos positivos a partir de FP).

En la próxima sección veremos cómo puede construirse un marco de pila que respete el AAPCS, siguiendo el ejemplo de los marcos construidos por gcc.

1.4. Estructura de una rutina

En esta sección vamos a ver cómo podemos diseñar una rutina para que respete el AAPCS, siguiendo la descripción dada en las secciones anteriores. Para ello el cuerpo de las rutinas suele estructurarse en tres partes:

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

El **código de entrada** completa la construcción del marco de activación de la rutina. En concreto, se encarga de:

- Guardar en la pila la dirección de retorno, que se obtiene del registro R14 (LR). Esto es necesario si R14 se utiliza en el cuerpo de la rutina para algún cálculo intermedio o si se realiza alguna llamada a otra subrutina, pero no es obligatorio en caso contrario. Si se compila con opciones de depuración activadas, gcc copia siempre el contenido de LR en la pila.
- Guardar en la pila el contenido de aquellos registros que son utilizados en el código de la rutina y que el AAPCS obliga a preservar: R4-R11 y R13 (SP).
- Reservar espacio en la pila para almacenar las variables locales de la rutina, asignándoles un valor inicial si procede.
- Reservar espacio en la pila para almacenar el valor de los parámetros de las subrutinas invocadas en el cuerpo de la rutina.

El **código de salida** restaura el estado de la rutina invocante, recuperando el valor de los registros, y copia en PC la dirección de retorno.

Como ejemplo veamos la estructura de las funciones C compiladas con gcc. El compilador de GNU genera un código basado en *frame pointer*, es decir, realiza los accesos a las variables locales y los parámetros de llamada utilizando como registro base R11 (FP), generándo el prólogo que se describe en el cuadro 1. La figura 1.5 ilustra la estructura del marco de pila obtenido.

Cuadro 1 Prólogo de función C generado por gcc.

```
MOV     IP, SP                          @ Guardar valor de retorno de SP
STMDB   SP!, {r4-r10,FP,IP,LR,PC}       @ Guardar registros en la pila
SUB     FP, IP, #4                      @ Actualizar FP
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. Es necesario porque el estándar dice que deben ser preservados,

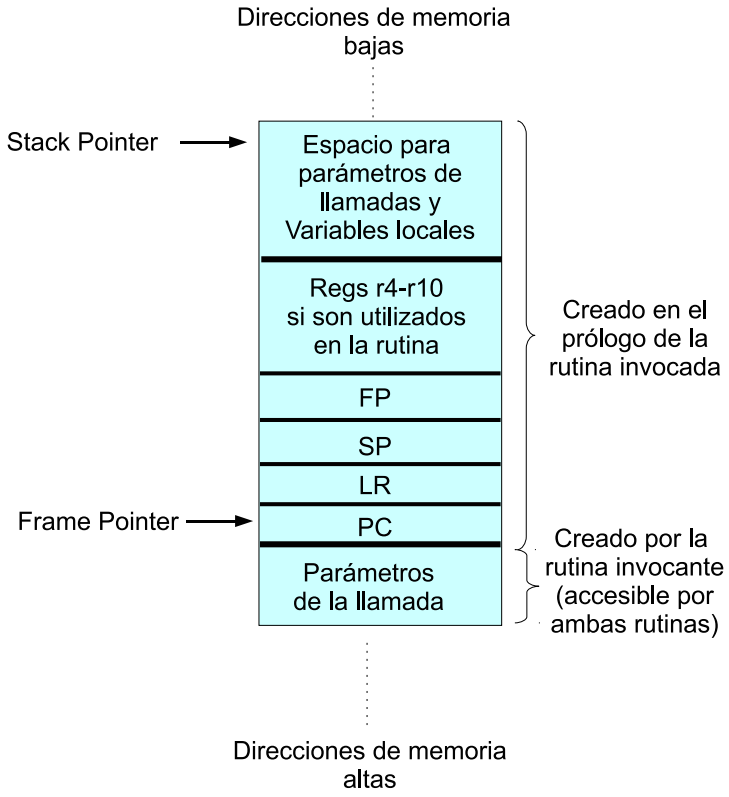

Figura 1.5: Marco de activación generado por gcc con opciones de depuración activadas.

por lo que necesitamos almacenar, su valor original para poder recuperarlo antes de realizar el retorno de subrutina.

- El registro R11 (FP) se utiliza como *frame pointer*, como el AAPCS dice que debe preservarse su valor se almacena en la pila.

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

- El registro R13 (SP) se utiliza como puntero de pila, como el AAPCS dice que debe preservarse, su valor se almacena en la pila.

- El registro R14 (LR) contiene la dirección de retorno. Si en la rutina vamos a utilizar R14 para algo (guardar un resultado o invocar una subrutina), tenemos que guardar la dirección de retorno en la pila. Sin embargo, si en la rutina no vamos a utilizar R14 para nada, no es necesario almacenarlo en la pila ya que el AAPCS no dice que deba ser preservado. En una compilación con opciones de depuración gcc siempre almacena la dirección de retorno en la pila, como vemos en el cuadro 1.

- En una compilación con opciones de depuración gcc almacena siempre el PC en la

pila. Esto permite obtener el punto de entrada a la función, muy útil en la depuración de rutinas con varios puntos de entrada.

- Tras el prólogo, el registro FP queda apuntando a la dirección de memoria que contiene el PC, justo el primer elemento que no pertenece al marco de activación de la rutina invocante. Los parámetros de la llamada son alcanzables a partir de FP con desplazamientos positivos, mientras que las variables locales serán alcanzables con desplazamientos negativos.

Para deshacer el marco de activación, devolviendo correctamente el control a la dirección de retorno, gcc genera un epílogo como el que describe el cuadro 2. El valor copiado en PC al ejecutar este epílogo es el valor que tenía LR al entrar en la función, es decir, la dirección de 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}
```

Trazado inverso de la pila

Parte de la información que gcc almacena en el marco de activación de la rutina tiene como misión facilitar el trabajo de los depuradores. En concreto permite recorrer fácilmente el grafo de llamadas a subrutina en orden inverso. A esto se le llama trazado inverso de pila o *stack backtrace*. Como hemos visto más arriba, la información contenida en el marco es la siguiente (en orden ascendente de dirección de memoria):

```
dir baja ----> Registros a preservar (r4-10)
               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) apunta al marco de la rutina que está actualmente en curso. Sin embargo, en la posición `[FP, -#12]` tenemos almacenada la base del marco de activación de la rutina que invocó a la actual, lo que permite al depurador localizar su marco. Como el valor del PC al comienzo de la rutina está almacenado en esa dirección, puede localizar el punto de entrada a la rutina, muy útil en caso de que existan varios puntos de entrada. Puede repetir este proceso hasta alcanzar la función **main**.

Por ejemplo, la Figura 1.6 ilustra como quedan enlazados los marcos de activación para el código del cuadro 3, en el momento en el que la rutina `two` escribe el mensaje por pantalla. El depurador puede deshacer el camino de llamadas hasta llegar a **main**. Ésta función (**main**) es especial, es el punto de entrada a cualquier programa escrito en C. Sin embargo, antes de comenzar la ejecución de dicha función, se ejecuta normalmente un

código cuya misión es preparar el sistema para la ejecución del programa. El comienzo de este fragmento de código es el punto de entrada real, marcado por el símbolo `start`. Justo antes de saltar a **main**, este código pone el registro FP a 0, lo que permitirá al depurador detectar la raíz del árbol de llamadas cuando realice el *backtrace*.

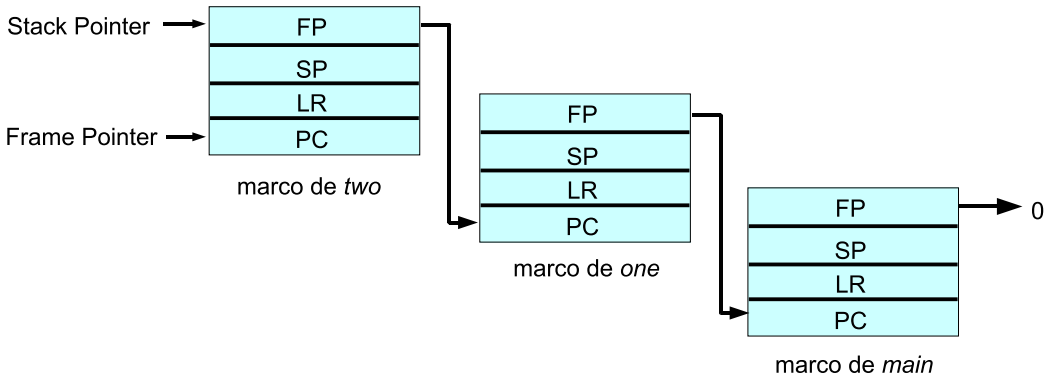

Figura 1.6: Lista enlazada de marcos de activación.

Cuadro 3 Ejemplo de programa C para ilustrar el desenrollado de pila.

```
#include <stdio.h>

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

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

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

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

void zero(void) { }
```

1.5. 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 1.5. Este espacio es reservado por el código de entrada de la función (prólogo) y es liberado por el código de salida (epílogo).

1.6. Símbolos globales

Durante el proceso de enlazado hay que 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. La discusión se ilustra en el cuadro 4.

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;
```

En este caso el modificador `extern` es **obligatorio**. 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 podamos tener dos o más variables globales con el mismo nombre, cada una restringida a un fichero distinto.

En ensamblador los símbolos globales debemos 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.

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;
   ...
}
```

1.6.1. 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, como ya hemos visto en la sección 1.6. Por ejemplo, si queremos usar una rutina `FOO`, sin parámetros de entrada ni valor de retorno, deberemos emplear la siguiente declaración adelantada para que el compilador sepa generar el código de llamada:

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

En ensamblador una variable global se identifica a través de una etiqueta asociada a la dirección de la variable. Cuando exportamos la etiqueta mediante `.global` se crea el símbolo correspondiente. Este símbolo puede importarse desde un fichero C declarando la variable como `extern`. El proceso se ilustra en el cuadro 5 mediante un ejemplo, compilado con gcc y enlazado con ld. En este ejemplo, el compilador crea una tabla de literales reservando una posición para la dirección de `MIVAR`. Posteriormente, el enlazador, cuando resuelve los símbolos, escribe la dirección de `MIVAR` en la entrada correspondiente de la tabla de literales y la variable puede ser correctamente haciendo uso de esta dirección.

Para recorrer desde C una región de memoria cuyo comienzo se ha definido en ensamblador, se puede emplear un array o simplemente un puntero. En ambos casos es necesario utilizar el modificador `extern`. Por ejemplo, si la etiqueta `TABLA` se ha utilizado para definir el comienzo de una región de memoria que contiene una tabla de valores `short int` (media palabra), en C deberíamos declarar el array como:

```
extern short int TABLA[];
```

1.7. Tipos compuestos

Los lenguajes de alto nivel como C ofrecen generalmente tipos de datos compuestos, como arrays, estructuras y uniones. Vamos a ver cómo es la implementación de bajo nivel de algunos 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.

1.7.1. Arrays

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

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 1.7. Como vemos las cadenas en C se terminan

Cuadro 5 Ejemplo de exportación de símbolos entre C y AS.

```
// fichero C que importa
// una variable entera
// MIVAR definida en un
// fichero ensamblador
// como:
//
//MIVAR:  .word 0x02

extern int MIVAR;

void Main(void)
{
    int i;
    MIVAR++;
    ...
}
```

```
//desensamblado del código objeto
//generado por el compilador

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:  e59f3084   ldr r3, [pc, #132]  ; 9c <.text+0x9c>
 14:  e5933000   ldr r3, [r3]
 18:  e2832001   add r2, r3, #1  ; 0x1
 1c:  e59f3078   ldr r3, [pc, #120]  ; 9c <.text+0x9c>
 20:  e5832000   str r2, [r3]
   ...
 9c:  00000000    //el compilador lo deja a 0
 ...
```

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

Disassembly of section .text:

0c000000 <Main>:
 c000000:  e1a0c00d     mov ip, sp
 c000004:  e92dd800     stmdb   sp!, {fp, ip, lr, pc}
 c000008:  e24cb004     sub fp, ip, #4  ; 0x4
 c00000c:  e24dd004     sub sp, sp, #4  ; 0x4
 c000010:  e59f3084     ldr r3, [pc, #132]  ; c00009c <.text+0x9c>
 c000014:  e5933000     ldr r3, [r3]
 c000018:  e2832001     add r2, r3, #1  ; 0x1
 c00001c:  e59f3078     ldr r3, [pc, #120]  ; c00009c <.text+0x9c>
 c000020:  e5832000     str r2, [r3]
    ...
 c00009c:  0c00074c     stceq7, cr0, [r0], {76}
    ...
0c00074c <MIVAR>:
 c00074c:  00000002     andeq   r0, r0, r2
 ...
```

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`).

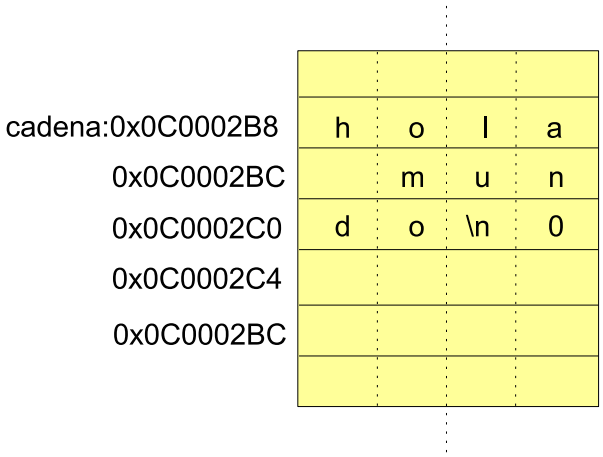

Figura 1.7: Almacenamiento de un array de caracteres en memoria.

Dependiendo de la arquitectura puede haber restricciones de alineamiento en el acceso a memoria. Este es el caso de la arquitectura ARM. 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 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.

1.7.2. Estructuras

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

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

struct mistruct rec;
```

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

Al igual que sucedía en el caso del array, el compilador debe colocar los campos en posiciones de memoria adecuadas, de forma que los accesos a cada uno de los campos no violen las restricciones de alineamiento. Es muy probable que el compilador se vea en la necesidad de *dejar huecos* entre unos campos y otros con el fin de respetar estas restricciones. Por ejemplo, el emplazamiento en memoria de la estructura de nuestro ejemplo sería similar al ilustrado en la figura 1.8.

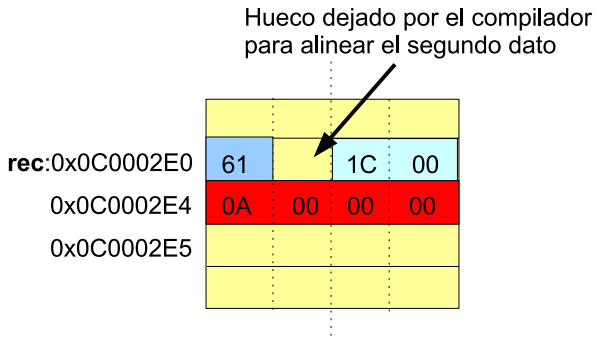

Figura 1.8: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a 0x61, 0x1C y 0x0A respectivamente.

1.8. Desarrollo de la práctica

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

1.8.1. 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 cuatro 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 al depurador. Si el procesador está corriendo (botón de stop en rojo) lo paramos pulsando el botón de stop.

11. Cargamos el programa 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? ¿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? ¿Qué sucede si compila con nivel de optimización `-O1`?
    - 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 cargar de nuevo el programa.

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 la declaración de una variable entera `i` al comienzo de la función Main y al final de la misma 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 cargarlo de nuevo. 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? ¿Cuáles 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?

1.8.2. Parte no guiada

En este apartado partimos de un proyecto, escrito en su mayoría en C, que implementa y utiliza un buffer estático circular.

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
.extern Main

.text
start:
    LDR sp,=0x0C7ff000
    MOV fp,#0
    ldr r0,=Main
    mov lr,pc
    mov pc, r0
End:
    B End

    .global FOO
FOO:
    str     r4, [sp, #-4]!
    ldmib   r0, {r2, r3}
    cmp     r2, r3
    ldreq   r4, [sp], #4
    moveq   pc, lr
    add     ip, r0, #12
    add     r4, r0, #524
LOOP:
    rsb     r3, ip, r2
    mov     r3, r3, asr #1
    strb    r3, [r1]
    ldrh    r3, [r2], #2
    strh    r3, [r1, #2]
    cmp     r2, r4
    movcs   r2, ip
    add     r1, r1, #4
    ldr     r3, [r0, #8]
    cmp     r2, r3
    bne     LOOP
    ldr     r4, [sp], #4
    mov     pc, lr

.global screen
screen: .space 1024,0x0
.end
```

La función `Main`, inicializa una variable global `B` que es de tipo `CBuffer`. El tipo se define en el fichero `cbuffer.h` y las funciones que operan sobre este buffer se definen en el fichero `cbuffer.c`.

- El fichero `main.c` contiene:

```
#include "cbuffer.h"
extern sdump screen[];

CBuffer B;

int Main(void)
{
    short int i;

    Initialize( &B );

    for( i = 0 ; i < 10 ; i++ ) {
        Insert( &B, i );
    }

    FOO( &B, screen );

    while( !Empty( &B ) ) {
        Extract( &B, &i );
    }

    return 0;
}
```

- El fichero `cbuffer.h` contiene:

```
#define N 256

typedef struct {
    short int count;
    short int* start;
    short int* end;
    short int Buffer[N];
} CBuffer;

typedef struct{
    char pos;
    short int value;
} sdump;

void Initialize( CBuffer* B );

int Full( CBuffer *B );

int Empty( CBuffer* B );

int Insert( CBuffer* B, short int elem );
```

```
int Extract( CBuffer* B, short int* elem );

void FOO( CBuffer* B, sdump* screen );
```

- y el fichero `cbuffer.c`:

```
#include "cbuffer.h"
#include <stdio.h>

extern void FOO( CBuffer* B, sdump* screen );

void Initialize( CBuffer* B )
{
    B->count = 0;
    B->start = B->Buffer;
    B->end   = B->start;
}

int Full( CBuffer *B )
{
    return B->count == N;
}

int Empty( CBuffer* B )
{
    return B->count == 0;
}

int Insert( CBuffer* B, short int elem )
{
    if( Full( B ) )
        return -1; // error

    *(B->end) = elem;
    B->end++;

    if( B->end >= B->Buffer + N )
        B->end = B->Buffer;

    B->count++;

    return 0;  // no hay error
}

int Extract( CBuffer* B, short int* elem )
{
```

```
    if( Empty( B ) )
        return -1; // error

    *elem = *(B->start);

    B->start++;

    if( B->start >= B->Buffer + N )
        B->start = B->Buffer;

    B->count--;

    return 0;
}
```

Se pide al alumno:

1. Reemplazar la función `Extract` por una rutina codificada en ensamblador, siguiendo el estándar AAPCS, 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). Explicar qué hace dicha función.

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.

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