Práctica 3

Programación con subrutinas

3.1. Objetivos

El objetivo de esta práctica es introducir el concepto de subrutina y familiarizarse con su uso. Para ello resulta necesario conocer:

- El soporte del ensamblador del ARM para la gestión de subrutinas.
- Conceptos de pila de llamadas, marco de activación, su construcción y su destrucción (prólogo/epílogo).
- Convenio de paso de parámetros a subrutinas y valor de retorno.

3.2. Subrutinas y Pila de Llamadas

Se conoce como subrutina a un fragmento de código que podemos invocar desde cualquier punto del programa (incluyendo otra o la propia subrutina), retomándose la ejecución del programa, en la instrucción siguiente a la invocación, cuando la subrutina haya terminado. La Figura 3.1 presenta un ejemplo, con un programa principal que invoca la subrutina `SubRut1`, las flechas discontinuas indican el cambio en el flujo de ejecución, tanto en la invocación como en el retorno. Las subrutinas se usan para simplificar el código y poder reusarlo, y son el mecanismo con el que los lenguajes de alto nivel implementan procedimientos, funciones y métodos.

Una subrutina puede estar parametrizada, esto es, el algoritmo que implementa puede especificarse en función de unos parámetros, que podemos inicializar en la invocación. Por ejemplo, la subrutina podría tener un bucle desde 1 hasta `N`, siendo `N` un parámetro. Cuando se invoca la subrutina se inicializa este `N` con un valor. Se dice que se pasan los parámetros en la llamada o invocación. Una subrutina puede además retornar un valor.

Como ilustra Figura 3.1 sólo existe una copia del código de una subrutina, que debe colocarse fuera de la región del programa principal, y resulta conveniente identificar la primera instrucción de la subrutina con una etiqueta, ya que entonces para invocarla basta con hacer dos cosas:

- indicar de alguna manera la dirección de la instrucción por la que debe continuar la ejecución cuando termine la subrutina, y

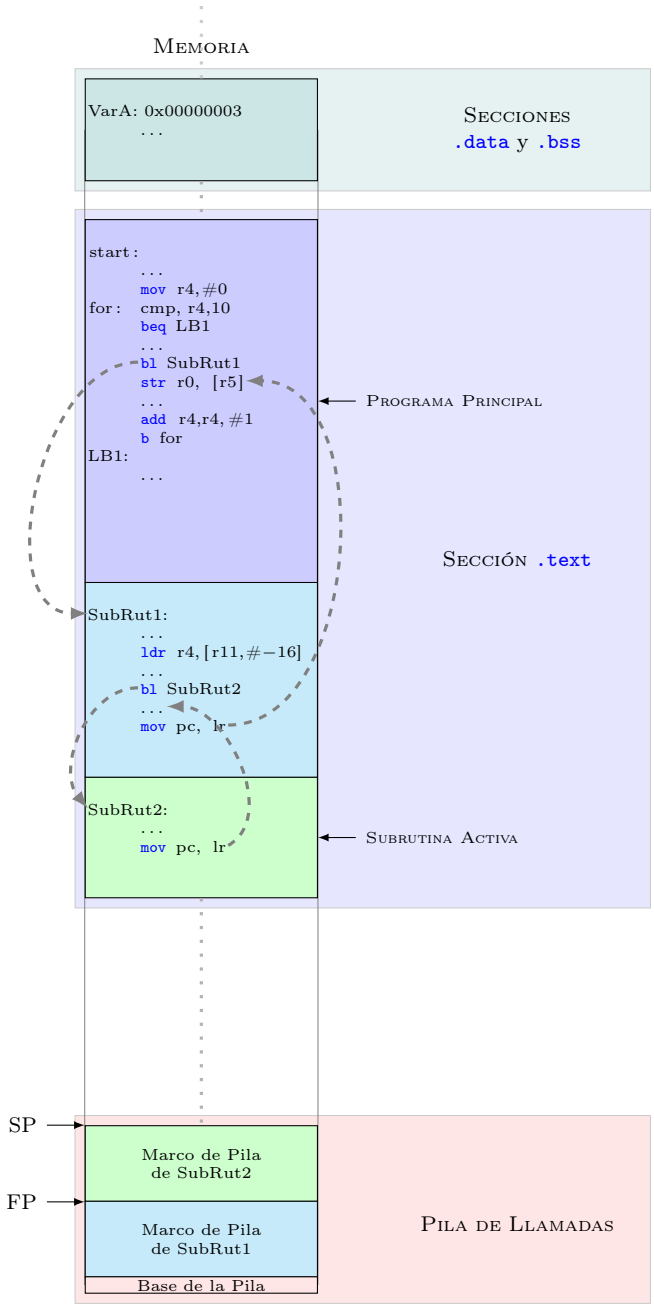


Figura 3.1: Subrutinas y Pila de Llamadas. El programa principal invoca la subrutina `SubRut1`, que a su vez invoca a `SubRut2`, que pasa a ser la subrutina activa. Los punteros `SP` y `FP` delimitan el marco de la subrutina activa.

- hacer un salto a la etiqueta que indica el comienzo de la subrutina.

Esto puede hacerse de varias maneras, pero generalmente todos los repertorios de instrucciones proporcionan una instrucción especial para la invocación de subrutinas. En el caso de ARM esta instrucción es el *Branch and Link*, cuya sintaxis en ensamblador es:

```
BL Etiqueta
```

Esta instrucción hace dos cosas: guarda en el registro `R14` la dirección siguiente a la instrucción `BL` (la dirección de retorno) y salta a la dirección indicada por `Etiqueta`. Cuando se hace este uso del registro `R14` se le denomina registro de enlace (*link register*), y por eso el ensamblador permite referirse a `R14` como `LR`.

Para retornar de la subrutina basta con escribir el valor contenido en `LR` (`r14`) sobre `PC`. Esto suele hacerse como en la Figura 3.1 con la instrucción `MOV`:

```
mov pc, lr
```

o con la instrucción `BX` (*brach and exchange*):

```
bx lr
```

Sin embargo, para que una subrutina pueda ser invocada con seguridad desde cualquier punto del programa, es imprescindible que la subrutina preserve el estado arquitectónico, es decir, el valor de los registros visibles al programador. La Figura 3.1 ilustra este problema. El programa principal recorre un bucle `for` usando `r4` para almacenar el iterador. En el cuerpo del bucle se llama a la subrutina `SubRut1` que contiene la instrucción:

```
ldr   r4, [r11, #-16]
```

El registro `r4` es machacado, con el indeseado efecto lateral de modificar el iterador del bucle. Obviamente esto no pasaría si el programador no hubiese escogido `r4` para almacenar el contador del bucle. Sin embargo nos interesa que quién utilice la subrutina no tenga que saber cómo está implementada.

La solución a este problema es sencilla, toda subrutina debe copiar temporalmente en memoria los registros que vaya a modificar, y restaurarlos justo antes de hacer el retorno, una vez que haya completado su trabajo.

Por tanto una subrutina necesitará temporalmente para su ejecución un poco de memoria donde poder guardar los registros que va a modificar. Fijémonos que esta reserva de memoria no puede ser estática, ya que no sabemos cuántas invocaciones de una subrutina puede haber en curso. Pensemos por ejemplo en una implementación recursiva del factorial de un número natural n. Tendremos al final n invocaciones en vuelo y por tanto necesitaremos memoria para salvar n veces el contexto, pero n es un dato de entrada. La mejor solución es que cada subrutina reserve la memoria que necesite para guardar el contexto al comenzar su ejecución (cada vez que se invoca) y la libere al terminar.

Para esto se utiliza una región de memoria conocida como la pila de llamadas (*call stack*). Es una región continua de memoria cuyos accesos siguen una política LIFO (*Last-In-First-Out*), que sirve para almacenar información relativa a las subrutinas activas del programa. La pila suele ubicarse al final de la memoria disponible, en las direcciones más altas, y suele crecer hacia las direcciones bajas.

La región de la pila utilizada por una rutina en su ejecución se conoce como el Marco de Activación o Marco de Pila de la subrutina (*Activation Record* o *Stack Frame* en inglés). El ejemplo de la Figura 3.1 muestra el estado de la pila cuando la subrutina `SubRut1` ha invocado a la subrutina `SubRut2` y esta última está en ejecución. La pila contiene entonces los marcos de activación de las subrutinas activas.

Las subrutinas suelen estructurarse en tres partes:

```
Código de entrada (prólogo)
Cuerpo de la subrutina
```

```
Código de salida (epílogo)
```

Comienzan con un prólogo, un código encargado de construir el marco de activación de la subrutina y de insertarlo en la pila. Finalizan con un epílogo, un código encargado de extraer el marco de activación de la pila, dejándola tal y como estaba antes de la ejecución del prólogo.

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

La subrutina usará el marco de activación como una zona de memoria privada, sólo accesible desde el contexto de la propia subrutina. Una parte del marco la utilizará para guardar una copia de los registros que vaya a modificar durante su ejecución, para restaurarlos en la ejecución del epílogo. Asimismo, la subrutina podrá utilizar también su marco de activación para almacenar sus variables locales, que se crearán con el marco y dejarán de ser accesibles cuando la subrutina finalice y extraiga su marco de la pila. Haremos hincapié en este tipo de variables en la próxima práctica.

Finalmente, el marco de activación se utiliza también para pasar por memoria los parámetros a la subrutina. El programa copia los parámetros en la cima de la pila. La subrutina inserta su marco de activación en la pila, justo por encima de los parámetros de la llamada, pudiendo acceder a ellos con desplazamientos relativos a `FP`. Existe una alternativa al paso de parámetros por pila, pasarlos por registro.

Es evidente que debe haber un acuerdo entre el programador que escribe la subrutina y el programador que escribe el programa que la invoca, deben estar de acuerdo en cómo se pasarán los parámetros y qué registros y/o direcciones de memoria se usarán para cada parámetro. Debemos pensar que lo habitual es utilizar código escrito por otras personas o código generado por un compilador a partir de un programa que escribamos en algún lenguaje de alto nivel, y por lo tanto necesitamos una serie de normas generales para la invocación y la escritura de subrutinas. Estas normas las especifica el fabricante, ARM en este caso, en el estándar de llamadas a subrutinas, que forma parte de lo que se denomina el *Application Binary Interface* (ABI). En la siguiente sección estudiaremos el estándar de ARM.

3.3. Estándar de llamadas a subrutinas

El *ARM Architecture Procedure Call Standard* (AAPCS) es el estándar vigente que regula las llamadas a subrutinas en la arquitectura ARM [aap] (sustituye a los estándares anteriores APCS y ATPCS). Especifica una serie de normas para que las rutinas puedan ser compiladas o ensambladas por separado, y que a pesar de ello puedan interactuar. En definitiva, supone un contrato entre el código que invoca y la subrutina invocada. Las secciones siguientes presentan de forma resumida los aspectos más relevantes que cubre el estándar. No obstante debemos notar que dicho estándar está en constante revisión y evolución, por ello es importante en la práctica comprobar la versión del ABI con la que se vaya a trabajar y estudiar el estándar correspondiente.

3.3.1. Uso de registros arquitectónicos

El estándar determina qué registros arquitectónicos debe preservar una subrutina y cuales no es necesario que preserve. Como mencionamos arriba, si la subrutina necesita utilizar alguno de los registros que debe preservar tendrá que copiarlo antes en su marco de activación y restaurar su valor antes del retorno. Por otro lado, el código que invoca a una subrutina debe ser muy consciente de que la subrutina no tiene la obligación de preservar algunos registros, y debe considerar siempre que el valor que tengan estos registros se perderá a través de la llamada.

La tabla 3.1 describe los usos que el AAPCS da a cada uno de los registros. Como vemos los registros reciben un nombre alternativo (alias), que nos recuerda el uso que les asigna el AAPCS. Observemos que aunque no sea necesario preservar el valor de `LR`, este registro es el que contiene la dirección de retorno. Si la subrutina hace llamadas a otras subrutinas (o la misma si es recursiva) tendrá que utilizar `LR` para escribir una nueva dirección de retorno (recordemos que esto lo hace `BL` automáticamente). Esto sucede en el ejemplo de la Figura 3.1 con `SubRut1`, que invoca a `SubRut2`. Entonces `SubRut1` deberá copiar también `LR` en su marco de activación para no perder la dirección de retorno al programa principal. Esto no es necesario si la subrutina no invoca otras subrutinas (`SubRut2`). Esas subrutinas se dice que son subrutinas hoja.

Tabla 3.1: Uso de los registros arquitectónicos según el AAPCS

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

3.3.2. Modelo de pila

La figura 3.2 ilustra el modelo *full-descending* impuesto por el AAPCS, que se caracteriza por:

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

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

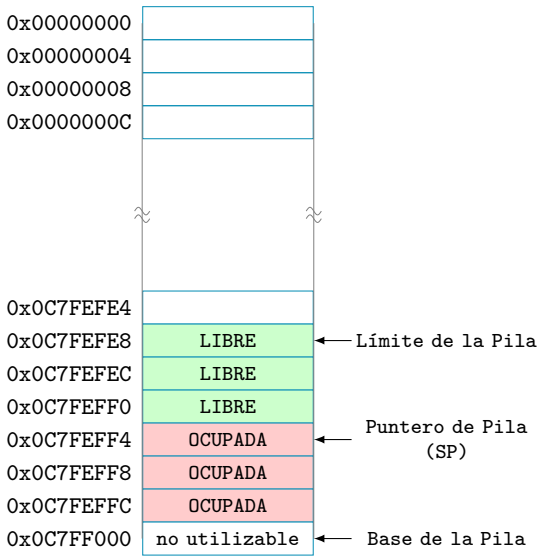


Figura 3.2: Ilustración de una pila *Full Descending*.

3.3.3. Paso de parámetros

Según el estándar, los cuatro primeros parámetros de una subrutina se deben pasar por registro, utilizando en orden los registros `r0-r3`. A partir del cuarto parámetro hay que pasarlos por memoria, escribiéndolos en orden en la cima de la pila:

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

De esta forma la subrutina siempre puede acceder a los parámetros que se le pasan por memoria con desplazamientos positivos a su frame pointer.

Observemos que esto supone colocarlos en la parte superior del marco de activación de la rutina invocante. Esto hay que tenerlo en cuenta en el prólogo para calcular correctamente el espacio total de marco que necesita la subrutina. La figura 3.3 ilustra un ejemplo en el que una subrutina `SubA` debe invocar otra subrutina `SubB`, pasándole 7 parámetros que llamamos `Param1-7`. Como podemos ver, los cuatro primeros parámetros se pasan por registro (`r0-r3`), mientras que los últimos tres se pasan por pila, en la cima del marco de `SubA`.

3.3.4. Valor de retorno

Para aquellas subrutinas que devuelven un valor (funciones en lenguajes de alto nivel), el AAPCS especifica que deben devolverlo en R0 [1].

[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

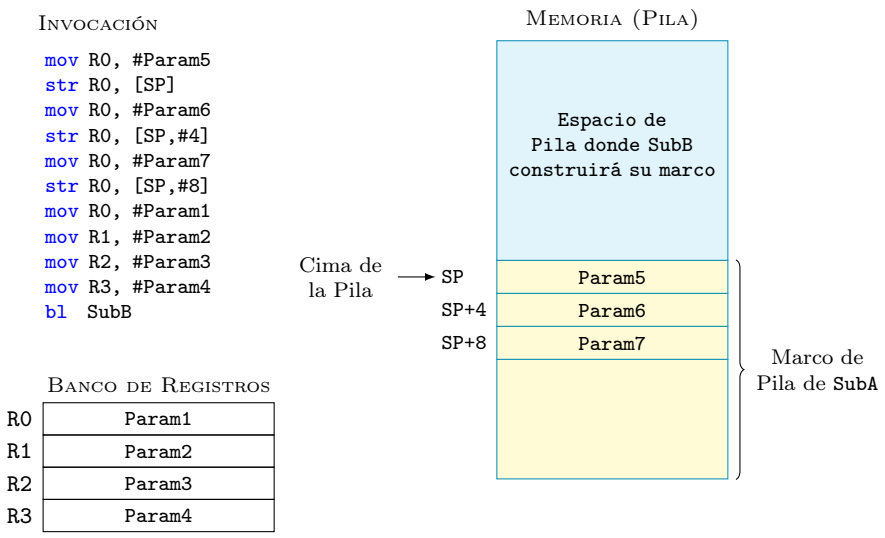


Figura 3.3: Paso de los parámetros `Param1-7` desde `SubA` a `SubB`. El código de invocación asume que los parámetros son constantes (valores inmediatos), en caso de ser variables habría que cargarlos con instrucciones `ldr` en lugar de `mov`.

3.4. Marco de Activación

Los estándares antiguos (APCS y ATPCS) daban algunas posibles estructuras para el marco de activación de una rutina. Sin embargo, con el AAPC desaparecen estas restricciones. Cualquier rutina que cumpla con las condiciones presentadas en las secciones anteriores estaría conforme al estándar. No obstante, en esta asignatura vamos a seguir siempre las siguientes pautas:

- Si nuestra subrutina invoca a otras subrutinas, tendremos que almacenar `LR` en la pila, ya que contiene la dirección de retorno pero lo tendremos que usar para pasar a su vez la dirección de retorno a las subrutinas invocadas.
- Vamos a usar siempre frame pointer, para facilitar la depuración y el acceso al marco de activación. Por lo tanto, tendremos siempre que almacenar el valor anterior de `FP` en la pila.
- Aunque `SP` es un registro que hay que preservar, no será necesario incluirlo en el marco, ya que en el epílogo podremos calcular el valor anterior de `SP` a partir del valor de `FP`.
- De los registros `r4-r10` insertaremos sólo aquellos que modifiquemos en el cuerpo de la subrutina.

La Figura 3.4 ilustra el marco de activación resultante de aplicar estas directrices a una subrutina que no es hoja, que es el que genera el compilador gcc-4.7 con depuración activada y sin optimizaciones[2]. Una de las ventajas de utilizar `FP`, aparte de la facilidad de codificación y depuración, es que se forma en la pila una lista enlazada de marcos de activación. Como ilustra la Figura 3.4, al salvar `FP` en el marco de una subrutina, estamos almacenando la dirección de la base del marco de activación de la subrutina que la invocó. Podemos así

[2]Con los flags -g -O0

recorrer el árbol de llamadas a subrutina hasta llegar a la primera. Para identificar esta primera subrutina el programa principal pone FP a 0 antes de invocarla.

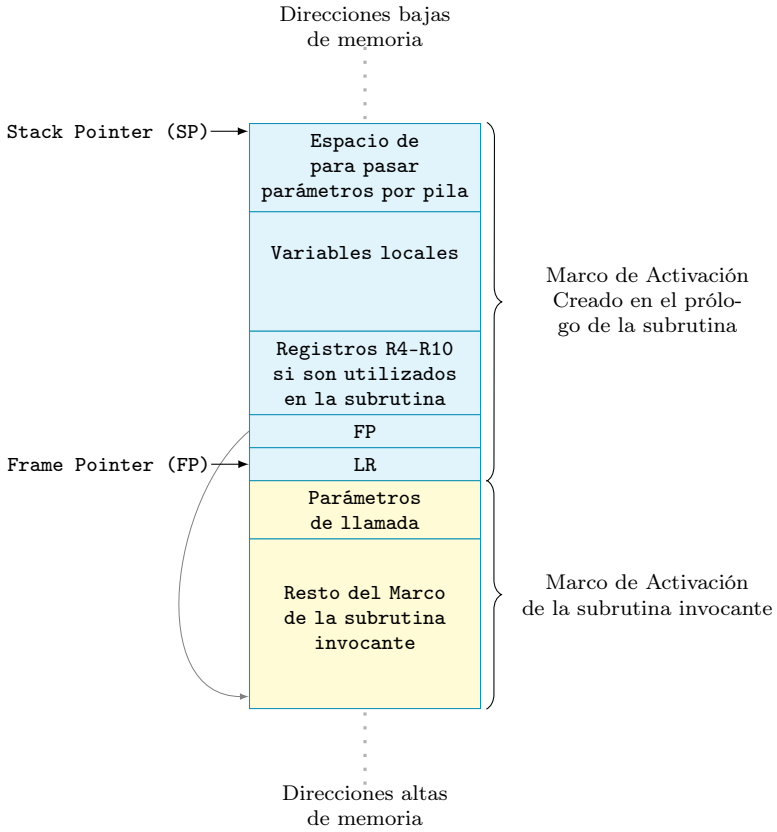


Figura 3.4: Estructura de Marco de Activación recomendada. En el caso de subrutinas hoja podremos omitir la copia de LR.

Para insertar el marco de activación en la pila (*push*) suelen utilizarse en ARM instrucciones de store múltiple, capaces de almacenar varios registros en memoria. Del mismo modo, para extraer el marco de la pila (*pop*) una vez terminada la subrutina suelen utilizarse instrucciones de load múltiple. En la siguiente sección analizaremos este tipo de instrucciones antes de presentar el código del prólogo y el epílogo que utilizaremos en nuestras subrutinas.

3.4.1. Instrucciones de Load y Store Múltiple

Además de las instrucciones de load y store convencionales, la arquitectura ARM ofrece instrucciones de load y store múltiple. Este tipo de instrucciones son muy útiles para la gestión de pila y las copias de bloques de datos en memoria. Además permiten reducir el tamaño del código.

El Load Múltiple (LDM) permite la carga simultánea de varios registros con datos ubicados en posiciones de memoria consecutivas, mientras que el Store Múltiple (STM) permite hacer la operación contraria, almacenar en posiciones de memoria consecutivas el valor de varios registros. La codificación de estas instrucciones en ensamblador es:

```
LDM<Modo> Rb[!], {lista de registros}
STM<Modo> Rb[!], {lista de registros}
```

Donde `Rb` es el registro base que contiene una dirección a partir de la cual se obtiene la dirección de memoria por la que comienza la operación. El signo `!` tras el registro base es opcional, si se pone, el registro base quedará actualizado adecuadamente para encadenar varios accesos de este tipo.

Existen cuatro modos de direccionamiento para estas instrucciones, dando lugar a 4 versiones de cada instrucción. En esta práctica vamos a limitarnos a estudiar sólo los dos modos que nos interesan para codificar el prólogo y el epílogo de nuestras rutinas, que son:

- *Increment After* (`IA`): La dirección de comienzo se toma del registro base `Rb`:

  ```
  dirección de comienzo = Rb
  ```

- *Decrement Before* (`DB`): La dirección de comienzo se forma restando al contenido del registro `Rb` cuatro veces el número de registros de la lista:

  ```
  dirección de comienzo = Rb - 4*NúmeroDeRegistros
  ```

El funcionamiento de las cuatro instrucciones a las que dan lugar se ilustra en la Figura 3.5. Empezando en la dirección de comienzo, se recorren todos los registros arquitectónicos en orden (primero `R0`, luego `R1`, ...). Si el registro está en la lista de registros de la instrucción se hace la operación correspondiente (load o store) con él y se incrementa en cuatro la dirección. Si no está se pasa al registro siguiente.

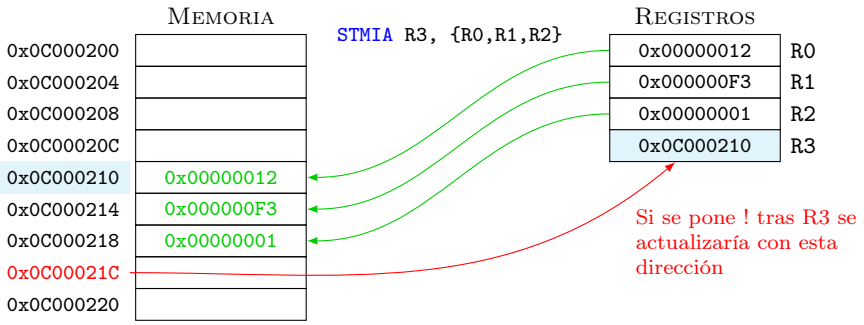
Las dos instrucciones que nos permiten implementar las operaciones de inserción (*push*) y extracción (*pop*) sobre una pila *full descending* son:

Inserción: `STMDB Rb!, {lista de registros}`. Esta instrucción es equivalente a un *push* de la lista de registros si utilizamos `SP` como registro base. De hecho podemos referirnos a esta instrucción con dos posibles alias:

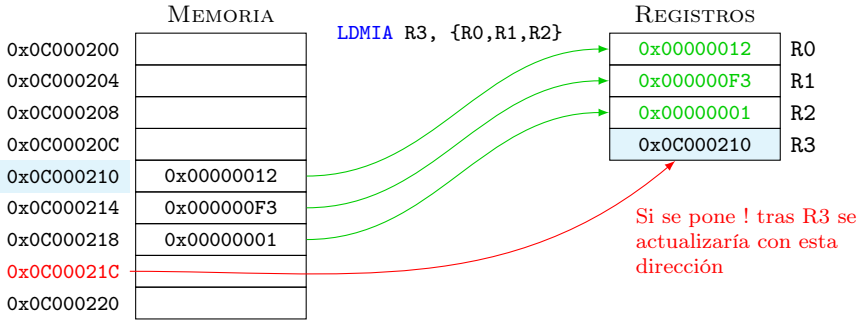
- `STMFD Rb!, {lista de registros}`, donde `FD` deriva de full descending.
- `PUSH {lista de registros}`. En este caso no podemos elegir el registro base, que forzosamente es `SP`. Éste es el formato que usaremos en los prólogos de las subrutinas.

Extracción: `LDMIA Rb!, {lista de registros}`. Esta instrucción es equivalente a un *pop* de la lista de registros si utilizamos `SP` como registro base. De hecho podemos referirnos a esta instrucción con dos posibles alias:

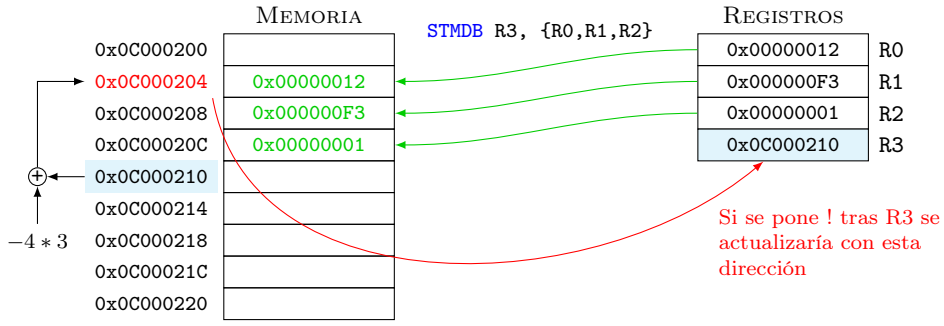
- `LDMFD Rb!, {lista de registros}`. Igual que antes `FD` deriva de full descending.
- `POP {lista de registros}`. En este caso no podemos elegir el registro base, que forzosamente es `SP`. Este es el formato que utilizaremos en los epílogos de las subrutinas.



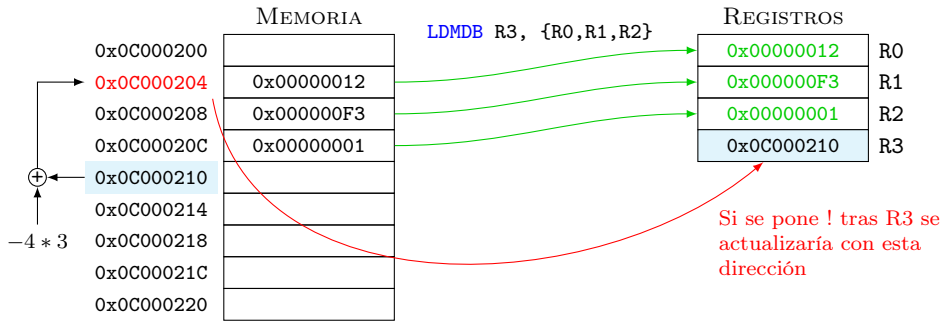
(a) STMIA



(b) LDMIA



(c) STMDB



(d) LDMDB

Figura 3.5: Instrucciónes de load y store múltiple

Ejemplos

```
STMDB SP!, {R4-R10,FP,LR}   @ Empezando en la dirección SP-4*9 copia el
                            @ contenido de los registros R4-R10, FP y LR.
                            @ SP queda con la dirección de comienzo (SP-4*9)
                            @ Se apilan los registros.
STMFD SP!, {R4-R10,FP,LR}   @ Lo mismo que la anterior
PUSH       {R4-R10,FP,LR}   @ Lo mismo que la anterior

LDMIA SP!, {R4-R10,FP,LR}   @ Empezando en la dirección SP copia el contenido
                            @ de la memoria en los registros R4-R10, FP y LR.
                            @ SP queda con la dirección original + 4*9
                            @ Se desapilan los registros.
LDMFD SP!, {R4-R10,FP,LR}   @ Lo mismo que la anterior
POP        {R4-R10,FP,LR}   @ Lo mismo que la anterior
```

3.4.2. Prólogo y Epílogo

El cuadro 1 describe un posible prólogo para la construcción del marco de activación representado en la Figura 3.4. La primera instrucción apila los registros a preservar sobre el marco de la rutina invocante, dejando `SP` apuntando a la nueva cima. Debemos incidir en que no es necesario apilar siempre todos los registros `R4-R10`, sólo aquellos que se vayan a modificar en la rutina. La siguiente instrucción deja el `FP` apuntando a la base del nuevo marco de activación. La última instrucción reserva nuevo espacio en el marco, el necesario para almacenar las variables locales y los parámetros que deba pasar por pila a las subrutinas que invoque. Existen algunos casos especiales en los que podemos simplificar este prólogo:

- Si no pasamos más de cuatro parámetros a ninguna subrutina y no vamos a utilizar variables locales, no tendremos que reservar espacio extra. Podemos entonces eliminar la tercera instrucción.
- Si sólo apilamos un registro (sucedería en una subrutina hoja que no modifique ninguno de los registros `r4-r10`) la segunda instrucción se convertiría en `ADD FP, SP, #0`. En este caso quizá quede más claro codificarla como `MOV FP, SP`.

Cuadro 1 Prólogo para insertar en la pila el Marco de Activación de la Figura 3.4.

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

El cuadro 2 presenta el correspondiente epílogo, que restaura la pila dejándola tal y como estaba antes de la ejecución del prólogo. Lo primero que hace es asignar a `SP` el valor que tenía tras la ejecución de la primera instrucción del prólogo. Esto lo consigue realizando la operación contraria a la segunda instrucción del prólogo. Otra alternativa sería hacer lo contrario de lo que hace la tercera instrucción del prólogo, asumiendo que no hemos alterado el valor de `SP` en el cuerpo de la subrutina, es decir, `ADD SP, SP, #4*NumPalabrasExtra`. Las dos instrucciones son igual de eficientes por lo que no hay un motivo a priori por el que elegir una de ellas. Hemos optado por la instrucción que aparece en el cuadro 2 porque es

la que utiliza el compilador gcc-4.7 que usamos en el laboratorio, y de esta forma el alumno estará más familiarizado con el código que genera el compilador y le resultará más fácil comprenderlo.

Con la segunda instrucción del epílogo se desapilan los registros apilados en el prólogo, dejando así la pila en el mismo estado que estaba antes de la ejecución del prólogo. Lo único que queda es hacer el retorno de subrutina, que es lo que hace la última instrucción del epílogo. En `R4-R10`, `FP` y `LR` se copian los valores que tenían al entrar en la subrutina, por lo que al terminar la ejecución del epílogo se restaura por completo el estado de la rutina invocante.

Cuadro 2 Epílogo para extraer de la pila el Marco de Activación de la Figura 3.4.

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

De nuevo, hay algunas situaciones en las que podemos simplificar este epílogo:

- Si sólo apilamos un registro la primera instrucción del epílogo se reduce a `SUB SP, FP, #0`. En este caso quizá quede más claro codificarla como `MOV SP, FP`.
- Si no hemos reservado espacio extra (`NumPalabrasExtra=0`) y no hemos alterado el valor de `SP` en el cuerpo de la subrutina, podemos incluso eliminar esta misma instrucción del epílogo, ya que `SP` tendrá el valor que le asignó la primera instrucción del prólogo.

3.5. Un ejemplo

El código de la Figura 3.6 ilustra un ejemplo de un programa que utiliza la subrutina `Recorre` para recorrer un array `A`. En cada posición `j` selecciona el mayor entre los elementos `A[j]` y `A[j+1]` utilizando la subrutina `Mayor`. El mayor de los dos se copia en la posición `j` de otro array `B`. A la izquierda podemos ver la implementación C y a la derecha la implementación en ensamblador. Observemos que al comienzo del programa se inicializa `SP` con el símbolo `_stack`, proporcionado por el enlazador al seguir las indicaciones del script de enlazado `ld_script.ld`. También se inicializa `fp` a cero, para que al llamar a la primera subrutina se meta el valor 0 en la base de su marco cuando salve `fp`, dejando una marca al depurador que indica que es la primera subrutina del árbol de llamadas.

3.6. Desarrollo de la práctica

Los alumnos deberán preparar en casa las soluciones a los apartados que se presentan a continuación. En el laboratorio se pedirá una **modificación** de estos apartados y se realizará un examen online para comprobar que el estudiante ha entendido correctamente todos los conceptos asociados a la práctica desarrollada.

a) Codificar en ensamblador del ARM la siguiente función en C encargada de buscar el valor máximo de un vector `A` de enteros positivos de longitud `longA` y devolver la posición de ese máximo (el índice):

Código C

```c
#define N 4
int A[N]={7,3,25,4};
int B[N];

void Recorre();
int Mayor();

void main(){
  Recorre (A, B, N);
}

void Recorre (int A[], int B[], int M){
  for(int j=0; j<M-1; j++){
    B[j] = Mayor(A[j],A[j+1]);
  }
}

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

Ensamblador

```
    .extern _stack
    .global start
    .equ N, 4
    .data
A:  .word 7,3,25,4
    .bss
B:  .space N*4

    .text
start:
    ldr sp, =_stack
    mov fp, #0
    ldr r0, =A
    ldr r1, =B
    mov r2, #N
    bl Recorre
    b .

Recorre:
    push {r4-r8,fp,lr}
    add fp, sp, #24 @ 24=4*7-4
    mov r4, r0      @ R4, A
    mov r5, r1      @ R5, B
    sub r6, r2, #1  @ R6, M-1
    mov r7, #0      @ R7, j
for:
    cmp r7, r6
    bge Ret1
    ldr r0, [r4, r7, lsl #2]
    add r8, r7, #1
    ldr r1, [r4, r8, lsl #2]
    bl Mayor
    str r0, [r5, r7, lsl #2]
    add r7, r7, #1
    b for
Ret1:
    pop {r4-r8,fp,lr}
    mov pc, lr

Mayor:
    push {fp}
    mov fp, sp  @ SP - 0
    cmp r0, r1
    bgt Ret2
    mov r0, r1
Ret2:
    pop {fp}
    mov pc, lr
    .end
```

Figura 3.6: Ejemplo con subrutinas

```
int i, max, ind;
int max(int A[], int longA){
  max=0;
  ind=0;
  for(i=0; i<longA; i++){
    if(A[i]>max){
      max=A[i];
      ind=i;
    }
  }
  return(ind);
}
```

NOTA: El código de este apartado se corresponde (excepto por las instrucciones de gestión de subrutina) con el bucle interno del apartado b de la práctica anterior. Nótese que la subrutina es de tipo hoja, por tanto sólo necesita salvar y restaurar el registro `fp` y los registros `r4` a `r10` que utilice.

b) Codificar en ensamblador del ARM un algoritmo de ordenación basado en el código del apartado anterior. Supongamos un vector `A` de `N` enteros mayores de 0, queremos rellenar un vector `B` con los valores de `A` ordenados de mayor a menor. Para ello nos podemos basar en el siguiente código de alto nivel:

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int B[N];
int j;
void main(){
  for(j=0; j<N; j++){
    ind=max(A,N)
    B[j]=A[ind];
    A[ind]=0;
  }
}
```

NOTA: El código de este apartado se corresponde (excepto por las instrucciones de llamada a subrutina) con el bucle externo del apartado b de la práctica anterior.

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.