CAPÍTULO 6

Gestión de subrutinas

Índice

Cuando se realiza una llamada a una subrutina en un lenguaje de alto nivel, los detalles de cómo se cede el control a dicha subrutina y la gestión de información que dicha cesión supone, quedan convenientemente ocultos.

Sin embargo, un compilador, o un programador en ensamblador, sí debe explicitar todos los aspectos que conlleva la gestión de la llamada y ejecución de una subrutina. Algunos de dichos aspectos se trataron en el capítulo anterior. En concreto, la transferencia de información entre el programa invocador y la subrutina, y la devolución del control al programa invocador cuando finaliza la ejecución de la subrutina.

Otro aspecto relacionado con la llamada y ejecución de subrutinas, y que no se ha tratado todavía, es el de la gestión de la información requerida por la subrutina. Esta gestión abarca las siguientes tareas:

Este capítulo forma parte del libro «Introducción a la arquitectura de computadores con Qt ARMSim y Arduino». Copyright © 2014 Sergio Barrachina Mir, Maribel Castillo Catalán, Germán Fabregat Llueca, Juan Carlos Fernández Fernández, Germán León Navarro, José Vicente Martí Avilés, Rafael Mayo Gual y Raúl Montoliu Colás. Se publica bajo la licencia «Creative Commons Atribución-CompartirIgual 4.0 Internacional».

- El almacenamiento y posterior recuperación de la información almacenada en determinados registros.
- El almacenamiento y recuperación de la dirección de retorno para permitir que una subrutina llame a otras subrutinas o a sí misma (*recursividad*).
- La creación y utilización de variables locales de la subrutina.

Salvo que la subrutina pueda realizar dichas tareas utilizando exclusivamente los registros destinados al paso de parámetros, será necesario crear y gestionar un espacio de memoria donde la subrutina pueda almacenar la información que necesita durante su ejecución. A este espacio de memoria se le denomina *bloque de activación de la subrutina* y se implementa por medio de una estructura de datos conocida coma *pila*. La gestión del bloque de activación de la subrutina constituye un tema central en la gestión de subrutinas.

Este capítulo está organizado como sigue. El primer apartado describe la estructura de datos conocida como pila y cómo se utiliza en ensamblador. El segundo apartado describe cómo se construye y gestiona el bloque de activación de una subrutina. Por último, se proponen una serie de problemas.

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.10 «*Subroutines and the Stack*» del libro «*Computer Organization and Architecture: Themes and Variations*» de Alan Clements. Conviene tener en cuenta que en dicho apartado se utiliza el juego de instrucciones ARM de 32 bits y la sintaxis del compilador de ARM, mientras que en este libro se describe el juego de instrucciones Thumb de ARM y la sintaxis del compilador GCC.

6.1. La pila

Una *pila* o *cola LIFO* (*Last In First Out*) es una estructura de datos que permite añadir y extraer datos con la peculiaridad de que los datos introducidos solo se pueden extraer en el sentido contrario al que fueron introducidos. Añadir datos en una pila recibe el nombre de apilar (*push*, en inglés) y extraer datos de una pila, desapilar (*pop*, en inglés).

Una analogía que se suele emplear para describir una pila es la de un montón de libros puestos uno sobre otro. Sin embargo, para que dicha analogía sea correcta, es necesario limitar la forma en la que se pueden añadir o quitar libros de dicho montón. Cuando se quiera añadir un libro, éste deberá colocarse encima de los que ya hay (lo que implica que no es posible insertar un libro entre los que ya están en el montón).

Por otro lado, cuando se quiera quitar un libro, solo se podrá quitar el libro que esté más arriba en el montón (por tanto, no se puede quitar un libro en particular si previamente no se han quitado todos los que estén encima de él). Teniendo en cuenta dichas restricciones, el montón de libros actúa como una pila, ya que solo se pueden colocar nuevos libros sobre los que ya están en el montón y el último libro colocado en la pila de libros será el primero en ser sacado de ella.

Un computador no dispone de un dispositivo específico que implemente una pila en la que se puedan introducir y extraer datos. La pila en un computador se implementa utilizando los siguientes elementos: memoria y un registro. La memoria se utiliza para almacenar los elementos que se vayan introduciendo en la pila y el registro para apuntar a la dirección del último elemento introducido en la pila (lo que se conoce como el *tope de la pila*).

Puesto que la pila se almacena en memoria, es necesario definir el sentido de crecimiento de la pila con respecto a las direcciones de memoria utilizadas para almacenar la pila. La arquitectura ARM sigue el convenio más habitual: la pila crece de direcciones de memoria altas a direcciones de memoria bajas. Es decir, cuando se apilen nuevos datos, éstos se almacenarán en direcciones de memoria más bajas que los que se hubieran apilado previamente. Por tanto, al añadir elementos, la dirección de memoria del tope de la pila disminuirá; y al quitar elementos, la dirección de memoria del tope de la pila aumentará.

Como ya se ha comentado, la dirección de memoria del tope de la pila se guarda en un registro. Dicho registro recibe el nombre de *puntero de pila* o `sp` (de las siglas en inglés de *stack pointer*). La arquitectura ARM utiliza como puntero de pila el registro `r13`.

Como se puede intuir a partir de lo anterior, introducir y extraer datos de la pila requerirá actualizar el puntero de pila y escribir o leer de la memoria con un cierto orden. Afortunadamente, la arquitectura ARM proporciona dos instrucciones que se encargan de realizar todas las tareas asociadas al apilado y al desapilado: «**`push`**» y «**`pop`**», respectivamente, que se explican en el siguiente apartado.

Sin embargo, y aunque para un uso básico de la pila es suficiente con utilizar las instrucciones «**`push`**» y «**`pop`**», para utilizar la pila de una forma más avanzada, como se verá más adelante, es necesario comprender en qué consisten realmente las acciones de apilado y desapilado.

La operación de apilar se realiza en dos pasos. En el primero de ellos se decrementa el puntero de pila en tantas posiciones como el tamaño en bytes alineado a 4 de los datos que se desean apilar. En el segundo, se almacenan los datos que se quieren apilar a partir de la dirección indicada por el puntero de pila. Así por ejemplo, si se quisiera apilar la palabra que contiene el registro `r4`, los pasos que se deberán realizar son: 1) decrementar el puntero de pila en 4 posiciones, «**`sub`** `sp, sp, #4`», y

II) almacenar el contenido del registro `r4` en la dirección indicada por `sp`, «**`str`** `r4, [sp]`». La Figura 6.1 muestra el contenido de la pila y el valor del registro `sp` antes y después de apilar el contenido del registro `r4`.

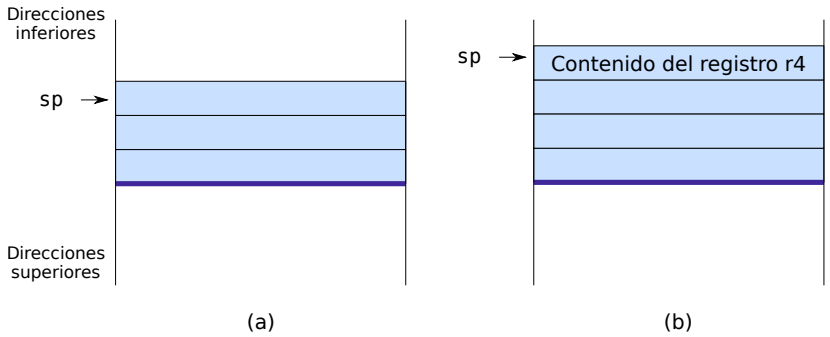


Figura 6.1: La pila (a) antes y (b) después de apilar el registro `r4`

La operación de desapilar también consta de dos pasos. En el primero de ellos se recuperan los datos que están almacenados en la pila. En el segundo, se incrementa el puntero de pila en tantas posiciones como el tamaño en bytes de los datos que se desean desapilar. Así por ejemplo, si se quisiera desapilar una palabra para cargarla en el registro `r4`, los pasos que se deberán realizar son: I) cargar el dato que se encuentra en la posición indicada por el registro `sp` en el registro `r4`, «**`ldr`** `r4, [sp]`», e II) incrementar en 4 posiciones el puntero de pila, «**`add`** `sp, sp, #4`». La Figura 6.2 muestra el contenido de la pila y el valor del registro `sp` antes y después de desapilar una palabra.

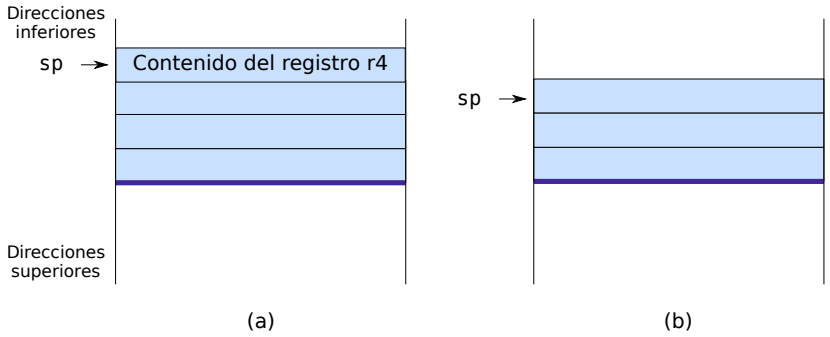


Figura 6.2: La pila (a) antes y (b) después de desapilar el registro `r4`

EJERCICIOS

Realiza los siguientes ejercicios relacionados con la operación apilar.

▶ **6.1** Suponiendo que el puntero de pila contiene el valor `0x7fffeffc` y que se desea apilar una palabra (4 bytes). ¿Qué valor deberá pasar a

tener el puntero de pila antes de almacenar la nueva palabra en la pila? ¿Qué instrucción se utilizará para hacerlo en el ensamblador ARM?

▶ **6.2** ¿Qué instrucción se utilizará para almacenar en la posición apuntada por el puntero de pila el contenido del registro `r5`?

▶ **6.3** A partir de los dos ejercicios anteriores indica qué dos instrucciones permiten apilar en la pila el registro `r5`.

El siguiente fragmento de código apila, uno detrás de otro, el contenido de los registros `r4` y `r5`:

subrutina-apilar-r4r5.s

```
1         .text
2 main:
3         mov r4, #10     @ r4<-10
4         mov r5, #13     @ r5<-13
5         sub sp, sp, #4  @ Actualiza sp (sp<-sp-4)
6         str r4, [sp]    @ Apila r4
7         sub sp, sp, #4  @ Actualiza sp (sp<-sp-4)
8         str r5, [sp]    @ Apila r5
9
10 stop:  wfi
11        .end
```

EJERCICIOS

Copia el programa anterior, cambia al modo de simulación y realiza los siguientes ejercicios:

▶ **6.4** Ejecuta el programa paso a paso y comprueba en qué posiciones de memoria, pertenecientes al segmento de pila, se almacenan los contenidos de los registros `r4` y `r5`.

▶ **6.5** Modifica el programa anterior para que en lugar de actualizar el puntero de pila cada vez que se pretende apilar un registro, se realice una única actualización del puntero de pila al principio y, a continuación, se almacenen los registros `r4` y `r5`. Los registros deben quedar apilados en el mismo orden que en el programa original.

EJERCICIOS

Realiza los siguientes ejercicios relacionados con la operación desapilar.

▶ **6.6** ¿Qué instrucción se utilizará para desapilar el dato contenido en el tope de la pila y cargarlo en el registro `r5`?

▶ **6.7** ¿Qué instrucción en ensamblador ARM se utilizará para actualizar el puntero de pila?

▶ **6.8** A partir de los dos ejercicios anteriores indica qué dos instrucciones permiten desapilar de la pila el registro `r5`.

▶ **6.9** Suponiendo que el puntero de pila contiene el valor `0x7fffeff8` ¿Qué valor deberá pasar a tener el puntero de pila después de desapilar una palabra (4 bytes) de la pila?

6.1.1. Operaciones sobre la pila empleando instrucciones «push» y «pop»

Como ya se ha comentado antes, si simplemente se quiere apilar el contenido de uno o varios registros o desapilar datos para cargarlos en uno o varios registros, la arquitectura ARM facilita la realización de las acciones de apilado y desapilado proporcionando dos instrucciones que se encargan de realizar todos los pasos vistos en el apartado anterior: «**push**» y «**pop**».

A modo de ejemplo, el siguiente fragmento de código apila el contenido de los registros `r4` y `r5` empleando la instrucción «**push**» y recupera los valores de dichos registros mediante la instrucción «**pop**»:

subrutina-apilar-r4r5-v2.s

```
1         .text
2 main:
3         mov r4, #10
4         mov r5, #13
5         push {r5, r4}
6         add r4, r4, #3
7         sub r5, r5, #3
8         pop {r5, r4}
9
10 stop:  wfi
11        .end
```

EJERCICIOS

Copia el programa anterior en el simulador y contesta a las siguientes preguntas mientras realizas una ejecución paso a paso.

▶ **6.10** ¿Cuál es el contenido del puntero de pila antes y después de la ejecución de la instrucción «**push**»?

▶ **6.11** ¿En qué posiciones de memoria, pertenecientes al segmento de pila, se almacenan los contenidos de los registros `r4` y `r5`?

▶ **6.12** ¿Qué valores tienen los registros `r4` y `r5` una vez realizadas las operaciones de suma y resta?

▶ **6.13** ¿Qué valores tienen los registros `r4` y `r5` tras la ejecución de la instrucción «**pop**»?

▶ **6.14** ¿Cuál es el contenido del puntero de pila tras la ejecución de la instrucción «**pop**»?

▶ **6.15** Fíjate que en el programa se ha puesto «**push** `{r5, r4}`». Si se hubiese empleado la instrucción «**push** `{r4, r5}`», ¿en qué posiciones de memoria, pertenecientes al segmento de pila, se hubieran almacenado los contenidos de los registros `r4` y `r5`? Según esto, ¿cuál es el criterio que sigue ARM para copiar los valores de los registros en la pila mediante la instrucción «**push**»?

6.2. Bloque de activación de una subrutina

Aunque la pila se puede utilizar para más propósitos, tiene una especial relevancia en la gestión de subrutinas, ya que es la estructura de datos ideal para almacenar la información requerida por la subrutina. Suponiendo, como se ha comentado anteriormente, que no sea suficiente con los registros reservados para el paso de parámetros.

Se denomina *bloque de activación* de una subrutina al segmento de la pila que contiene la información requerida por dicha subrutina. El bloque de activación de una subrutina cumple los siguientes cometidos:

- En el caso que la subrutina llame a otras subrutinas, almacenar la dirección de retorno original.
- Proporcionar espacio para las variables locales de la subrutina.
- Almacenar los registros que la subrutina necesita modificar y que el programa que ha hecho la llamada no espera que sean modificados.
- Mantener los valores que se han pasado como argumentos a la subrutina.

6.2.1. Anidamiento de subrutinas

El anidamiento de subrutinas tiene lugar cuando un programa invocador llama a una subrutina y ésta, a su vez, llama a otra, o a sí misma. El que una subrutina se llame a sí misma es lo que se conoce como recursividad.

Cuando se anidan subrutinas, el programa invocado se convierte en un momento dado en invocador, lo que obliga a ser cuidadoso con la gestión de las direcciones de retorno. Como se ha visto, la instrucción utilizada para llamar a una subrutina es la instrucción «**bl**». Dicha instrucción, antes de realizar el salto propiamente dicho, almacena la dirección de retorno del programa invocador en el registro `lr`. Si durante

la ejecución del programa invocado, éste llama a otro (o a sí mismo) utilizando otra instrucción «**bl**», cuando se ejecute dicha instrucción, se almacenará en el registro `lr` la nueva dirección de retorno. Por tanto, el contenido original del registro `lr`, es decir, la dirección de retorno almacenada tras ejecutar el primer «**bl**», se perderá. Si no se hiciera nada para evitar perder la anterior dirección de retorno, cuando se ejecuten las correspondientes instrucciones de vuelta, «**mov** `pc, lr`», se retornará siempre a la misma posición de memoria, la almacenada en el registro `lr` por la última instrucción «**bl**».

El siguiente fragmento de código ilustra este problema realizando dos llamadas anidadas.

subrutina-llamada-arm-v2.s

```
1            .data
2  datos:    .word 5, 8, 3, 4
3  suma1:    .space 4
4  suma2:    .space 4
5            .text
6
7  main:     ldr r0, =datos     @ Paso de parametros para sumatorios
8            ldr r1, =suma1
9  salto1:   bl sumatorios      @ Llama a la subrutina sumatorios
10 stop:     wfi                @ Finaliza la ejecucion
11
12 sumatorios: mov r7, #2
13           mov r5, r0
14           mov r6, r1
15 for:      cmp r7, #0
16           beq salto4
17           ldr r0, [r5]       @ Paso de parametros para suma_elem
18           ldr r1, [r5, #4]
19 salto2:   bl suma_elem       @ Llama a la subrutina suma_elem
20           str r2, [r6]
21           add r5, r5, #8
22           add r6, r6, #4
23           sub r7, r7, #1
24           b for
25 salto4:   mov pc, lr
26
27 suma_elem: add r2, r1, r0
28 salto3:   mov pc, lr
29
30           .end
```

EJERCICIOS

Edita el programa anterior, cárgalo en el simulador y ejecútalo paso

a paso.

▶ **6.16** ¿Dónde pasa el control del programa tras la ejecución de la instrucción etiquetada por «`salto1`»? ¿Qué valor se carga en el registro `lr` después de ejecutar la instrucción etiquetada por «`salto1`»?

▶ **6.17** ¿Dónde pasa el control del programa tras la ejecución de la instrucción etiquetada por «`salto2`»? ¿Qué valor se carga en el registro `lr` después de ejecutar la instrucción etiquetada por «`salto2`»?

▶ **6.18** ¿Dónde pasa el control del programa tras la ejecución de la instrucción etiquetada por «`salto3`»?

▶ **6.19** ¿Dónde pasa el control del programa tras la ejecución de la instrucción etiquetada por «`salto4`»?

▶ **6.20** Explica qué ocurre en el programa.

El ejercicio anterior muestra que es necesario utilizar alguna estructura de datos que permita almacenar las distintas direcciones de retorno cuando se realizan llamadas anidadas.

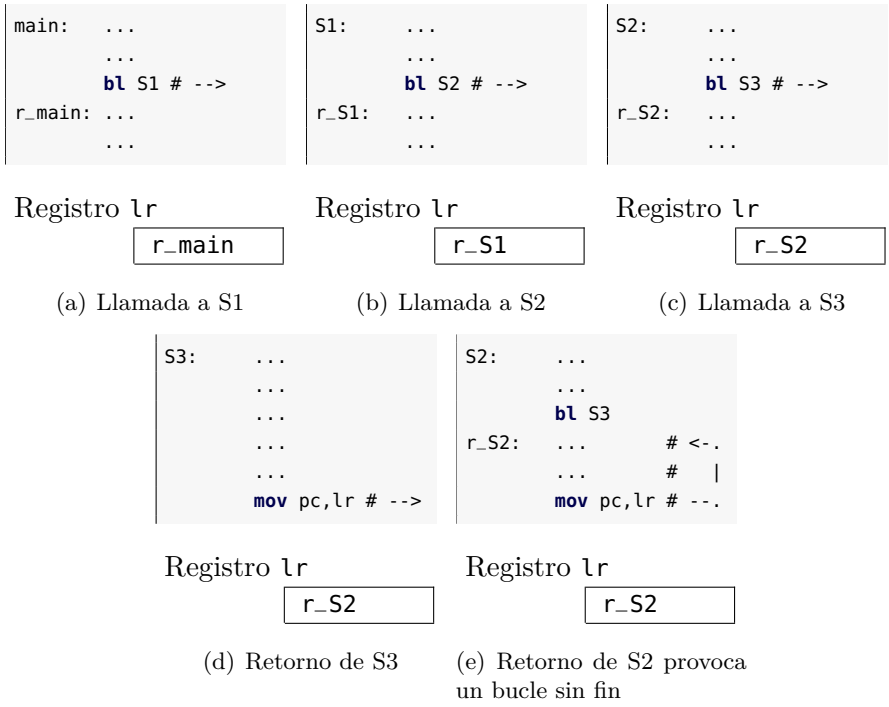
Dicha estructura de datos debe satisfacer dos requisitos. En primer lugar, debe ser capaz de permitir recuperar las direcciones de retorno en orden inverso a su almacenamiento (ya que es el orden en el que se producen los retornos). En segundo lugar, el espacio reservado para este cometido debe poder crecer de forma dinámica (la mayoría de las veces no se conoce cuántas llamadas se van a producir, ya que puede depender de cuáles sean los datos del problema).

La estructura de datos que mejor se adapta a los anteriores requisitos es la pila. Para almacenar y recuperar las direcciones de retorno utilizando una pila basta con proceder de la siguiente forma. Antes de realizar una llamada a otra subrutina (o a sí misma), se deberá apilar la dirección de retorno actual. Y antes de retornar, se deberá desapilar la última dirección de retorno apilada.

Por tanto, el programa invocador deberá apilar el registro `lr` antes de invocar al nuevo programa y desapilarlo antes de retornar. Es decir, en el caso de que se realicen llamadas anidadas, el contenido del registro `lr` formará parte de la información que se tiene que apilar en el bloque de activación de la subrutina.

La Figura 6.3 muestra de forma esquemática qué es lo que puede ocurrir si no se almacena la dirección de retorno. En dicha figura se muestra el contenido del registro `lr` justo después de ejecutar una instrucción «**`bl`**». Como se puede ver, a partir de la llamada a la subrutina S3, el registro `lr` solo mantiene almacenada la dirección de retorno del último «**`bl`**», «`dir_ret2`». Por tanto, todos los retornos que se produzcan a partir de la última llamada, rutina S3, retornarán a la posición «`dir_ret2`». El retorno de la rutina S3 será correcto, pero cuando se

ejecute el retorno de la rutina S2 se volverá de nuevo a la posición «`dir_ret2`», provocando la ejecución de un bucle infinito.



(a) Llamada a S1 (b) Llamada a S2 (c) Llamada a S3

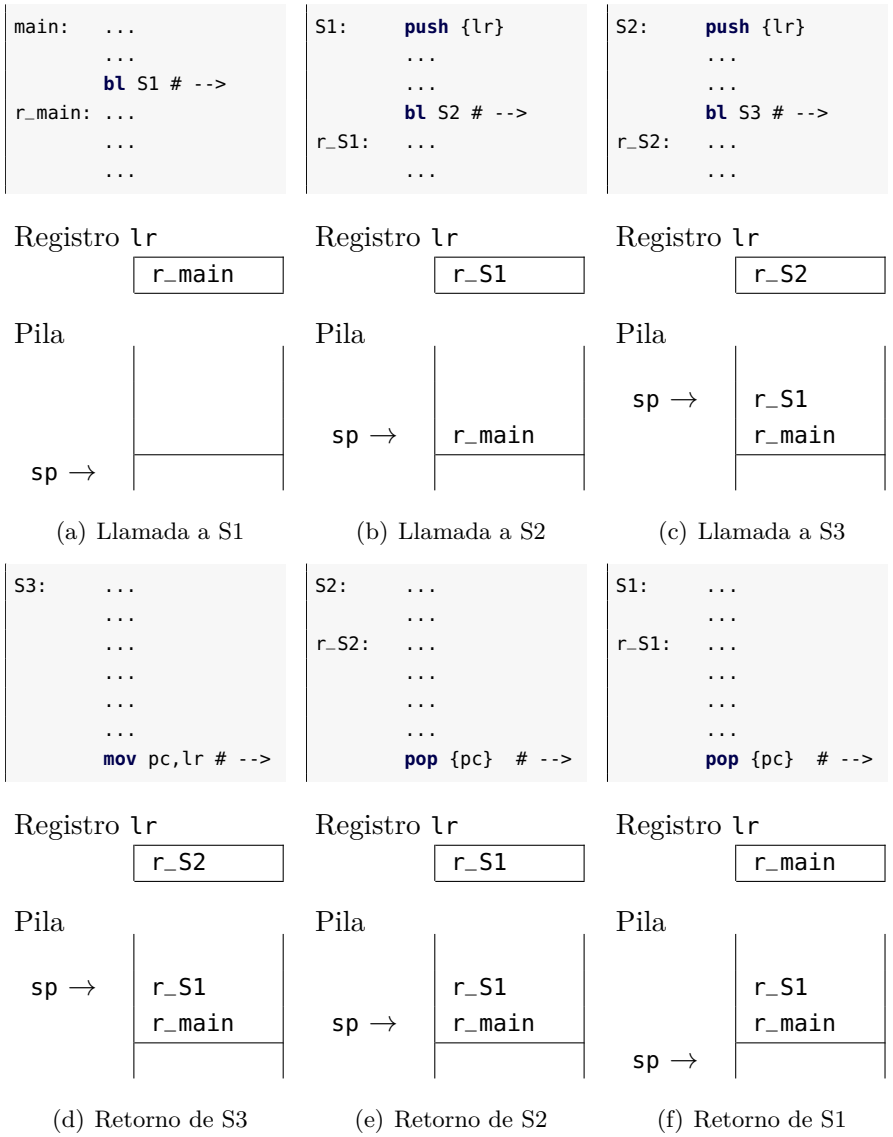
(d) Retorno de S3 (e) Retorno de S2 provoca un bucle sin fin

Figura 6.3: Llamadas anidadas a subrutinas (sin apilar las direcciones de retorno)

La Figura 6.4 muestra de forma esquemática qué es lo que ocurre cuando sí que se almacena la dirección de retorno. En dicha figura se muestra cuándo se debe apilar y desapilar el registro `lr` en la pila para que los retornos se produzcan en el orden adecuado. Se puede observar el estado de la pila y el valor del registro `lr` justo después de ejecutar una instrucción «**`bl`**». En las subrutinas S2 y S3 se apila el registro `lr` mediante «**`push {lr}`**». Para desapilar la dirección de retorno se emplea «**`pop {pc}`**», que almacena en el contador de programa la dirección de retorno para efectuar el salto. Por tanto, no se necesita ninguna otra instrucción para llevar a cabo el retorno de la subrutina.

EJERCICIOS

▶ **6.21** Modifica el fragmento de código anterior para que la dirección de retorno se apile y desapile de forma adecuada.



(a) Llamada a S1 (b) Llamada a S2 (c) Llamada a S3

(d) Retorno de S3 (e) Retorno de S2 (f) Retorno de S1

Figura 6.4: Llamadas anidadas a subrutinas (apilando las direcciones de retorno)

6.2.2. Variables locales de la subrutina

Una subrutina puede requerir durante su ejecución utilizar variables locales que solo existirán mientras se está ejecutando. Dependiendo del tipo de variables, bastará con utilizar los registros no ocupados o será necesario utilizar la memoria principal. En el caso de que sea necesario utilizar la memoria principal, dichas variables deberán almacenarse en el bloque de activación de la subrutina.

En el caso de datos de tipo escalar, siempre que sea posible, se utilizarán los registros del procesador que no estén siendo utilizados.

Pero en el caso de los datos de tipo estructurado, se hace necesario el uso de memoria principal. Es decir, las variables de tipo estructurado formarán parte del bloque de activación de la subrutina.

La forma de utilizar el bloque de activación para almacenar las variables locales de una subrutina es la siguiente. Al inicio de la subrutina se deberá reservar espacio en el bloque de activación para almacenar dichas variables. Y antes del retorno se deberá liberar dicho espacio.

6.2.3. Almacenamiento de los registros utilizados por la subrutina

Como se ha visto en el apartado anterior, la subrutina puede utilizar registros como variables locales, y por tanto, el contenido original de dichos registros puede perderse en un momento dado. Si la información que contenían dichos registros es relevante para que el programa invocador pueda continuar su ejecución tras el retorno, será necesario almacenar temporalmente dicha información en algún lugar. Este lugar será el bloque de activación de la subrutina.

La forma en la que se almacena y restaura el contenido de aquellos registros que vaya a sobrescribir la subrutina en el bloque de activación es la siguiente. La subrutina, antes de modificar el contenido de los registros, los apila en el bloque de activación. Una vez finalizada la ejecución de la subrutina, y justo antes del retorno, los recupera.

Este planteamiento implica almacenar en primer lugar todos aquellos registros que vaya a modificar la subrutina, para posteriormente recuperar sus valores originales antes de retornar al programa principal.

6.2.4. Estructura y gestión del bloque de activación

Como se ha visto, el bloque de activación de una subrutina está localizado en memoria y se implementa por medio de una estructura de tipo pila. El bloque de activación visto hasta este momento se muestra en la Figura 6.5.

Un aspecto que influye en la eficiencia de los programas que manejan subrutinas y sus respectivos bloques de activación es el modo en el

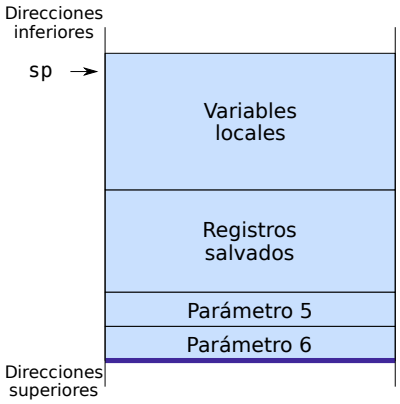


Figura 6.5: Esquema del bloque de activación

que se accede a la información contenida en los respectivos bloques de activación.

El modo más sencillo para acceder a un dato en el bloque de activación es utilizando el modo indexado. En el modo indexado la dirección del dato se obtiene sumando una dirección base y un desplazamiento. Como dirección base se puede utilizar el contenido del puntero de pila, que apunta a la posición de memoria más baja del bloque de activación (ver Figura 6.5). El desplazamiento sería entonces la posición relativa del dato con respecto al puntero de pila. De esta forma, sumando el contenido del registro `sp` y un determinado desplazamiento se obtendría la dirección de memoria de cualquier dato que se encontrara en el bloque de activación. Por ejemplo, si se ha apilado una palabra en la posición 8 por encima del `sp`, se podría leer su valor utilizando la instrucción «`ldr r4, [sp, #8]`».

6.2.5. Convenio para la llamada a subrutinas

Tanto el programa invocador como el invocado intervienen en la creación y gestión del bloque de activación de una subrutina. La gestión del bloque de activación se produce principalmente en los siguientes momentos:

- Justo antes de que el programa invocador pase el control a la subrutina.
- En el momento en que la subrutina toma el control.
- Justo antes de que la subrutina devuelva el control al programa invocador.
- En el momento en el que el programa invocador recupera el control.

A continuación se describe con más detalle qué es lo que debe realizarse en cada uno de dichos momentos.

Justo antes de que el programa invocador pase el control a la subrutina:

1. Paso de parámetros. Cargar los parámetros en los lugares establecidos. Los cuatro primeros se cargan en registros, `r0` a `r3`, y los restantes se apilan en el bloque de activación (p.e., los parámetros 5 y 6 de la Figura 6.5).

En el momento en que la subrutina toma el control:

1. Reservar memoria en la pila para el resto del bloque de activación. El tamaño se calcula sumando el espacio en bytes que ocupa el contenido de los registros que requiera apilar la subrutina más el espacio que ocupan las variables locales que se vayan a almacenar en el bloque de activación.

2. Almacenar en el bloque de activación aquellos registros que vaya a modificar la subrutina (incluido el registro `lr`).

Justo antes de que la subrutina devuelva el control al programa invocador:

1. Cargar el valor (o valores) que deba devolver la subrutina en los registros `r0` a `r3`.

2. Restaurar el valor original de los registros apilados por la subrutina (incluido el registro `lr`, que se restaura sobre el registro `PC`).

En el momento en el que el programa invocador recupera el control:

1. Eliminar del bloque de activación los parámetros que hubiera apilado.

2. Recoger los parámetros devueltos.

En la Figura 6.6 se muestra el estado de la pila después de que un programa haya invocado a otro siguiendo los pasos que se han descrito. En dicha figura aparece enmarcado el bloque de activación creado tanto por el programa invocador como por el invocado.

Como ejemplo de cómo se utiliza el bloque de activación se propone el siguiente programa Python3. Más adelante se muestra una versión equivalente en ensamblador.

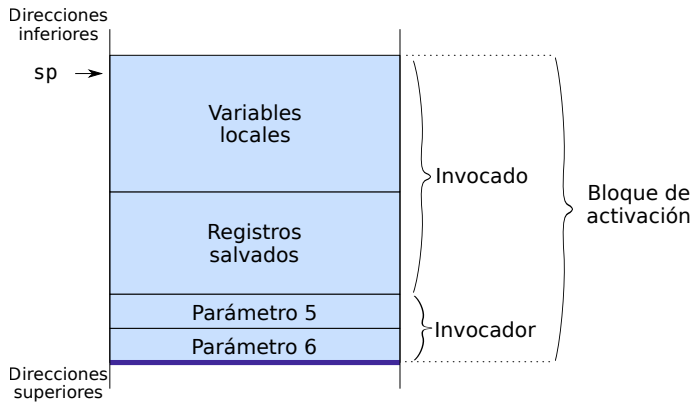


Figura 6.6: Estado de la pila después de una llamada a subrutina

```
1  def sumatorios(A, dim):
2        B = [0]*dim
3        for i in range(dim):
4             B[i] = sumatorio(A[i:], dim-i)
5
6        for i in range(dim):
7             A[i] = B[i]
8        return
9
10 def sumatorio(A, dim):
11       suma = 0;
12       for i in range(dim):
13            suma = suma + A[i]
14       return suma
15
16 A = [6, 5, 4, 3, 2, 1]
17 dim = 6
18 sumatorios(A, dim)
```

A continuación se muestra el equivalente en ensamblador ARM del anterior programa en Python3.

subrutina-varlocal-v0.s

```
1              .data
2  A:          .word 7, 6, 5, 4, 3, 2
3  dim:        .word 6
4              .text
5
6  @ Programa invocador
7  main:       ldr r0, =A
8              ldr r4, =dim
```

```
9             ldr r1, [r4]
10            bl sumatorios
11
12 fin:       wfi
13
14 @ subrutina sumatorios
15 sumatorios: @ --- 1 ---
16            push {r4, r5, r6, lr}
17            sub sp, sp, #32
18            add r4, sp, #0
19            str r0, [sp, #24]
20            str r1, [sp, #28]
21            mov r5, r0
22            mov r6, r1
23
24
25 for1:       cmp r6, #0
26            beq finfor1
27
28            @ --- 2 ---
29            bl sumatorio
30            str r2, [r4]
31
32            @ --- 3 ---
33            add r4, r4, #4
34            add r5, r5, #4
35            sub r6, r6, #1
36            mov r0, r5
37            mov r1, r6
38            b for1
39
40 finfor1:    @ --- 4 ---
41            ldr r0, [sp, #24]
42            ldr r1, [sp, #28]
43            add r4, sp, #0
44
45 for2:       cmp r1, #0
46            beq finfor2
47            ldr r5, [r4]
48            str r5, [r0]
49            add r4, r4, #4
50            add r0, r0, #4
51            sub r1, r1, #1
52            b for2
53
54 finfor2:    @ --- 5 ---
55            add sp, sp, #32
56            pop {r4, r5, r6, pc}
```

```
57
58 @ subrutina sumatorio
59 sumatorio:  push {r5, r6, r7, lr}
60            mov r2, #0
61            mov r6, r1
62            mov r5, r0
63 for3:       cmp r6, #0
64            beq finfor3
65            ldr r7, [r5]
66            add r5, r5, #4
67            add r2, r2, r7
68            sub r6, r6, #1
69            b for3
70 finfor3:    pop {r5, r6, r7, pc}
71
72            .end
```

EJERCICIOS

Carga el programa anterior en el simulador y contesta a las siguientes preguntas:

▶ **6.22** Localiza el fragmento de código del programa ensamblador donde se pasan los parámetros a la subrutina «`sumatorios`». Indica cuántos parámetros se pasan, el lugar por donde se pasan y el tipo de parámetros.

▶ **6.23** Indica el contenido del registro `lr` una vez ejecutada la instrucción «**`bl`** `sumatorios`».

▶ **6.24** Dibuja y detalla (con los desplazamientos correspondientes) el bloque de activación creado por la subrutina «`sumatorios`». Justifica el almacenamiento de cada uno de los datos que contiene el bloque de activación.

▶ **6.25** Indica el fragmento de código del programa ensamblador donde se pasan los parámetros a la subrutina «`sumatorio`». Indica cuántos parámetros se pasan, el lugar por donde se pasan y el tipo de parámetros.

▶ **6.26** Indica el contenido del registro `lr` una vez ejecutada la instrucción «**`bl`** `sumatorio`».

▶ **6.27** Dibuja el bloque de activación de la subrutina «`sumatorio`».

▶ **6.28** Una vez ejecutada la instrucción «**`pop`** `{r5, r6, r7, pc}`» de la subrutina «`sumatorio`» ¿Dónde se recupera el valor que permite retornar a la subrutina «`sumatorios`»?

▶ **6.29** Localiza el fragmento de código donde se desapila el bloque de activación de la subrutina «`sumatorios`».

▶ **6.30** ¿Dónde se recupera el valor que permite retornar al programa principal?

6.3. Problemas del capítulo

EJERCICIOS

▶ **6.31** Desarrolla dos subrutinas en ensamblador: «`subr1`» y «`subr2`». La subrutina «`subr1`» tomará como entrada una matriz de enteros de dimensión $m \times n$ y devolverá dicha matriz pero con los elementos de cada una de sus filas invertidos. Para realizar la inversión de cada una de las filas se deberá utilizar la subrutina «`subr2`». Es decir, la subrutina «`subr2`» deberá tomar como entrada un vector de enteros y devolver dicho vector con sus elementos invertidos.

(Pista: Si se apilan elementos en la pila y luego se desapilan, se obtienen los mismos elementos pero en el orden inverso.)

▶ **6.32** Desarrolla tres subrutinas en ensamblador, «`subr1`», «`subr2`» y «`subr3`». La subrutina «`subr1`» devolverá un 1 si las dos cadenas de caracteres que se le pasan como parámetro contienen el mismo número de los distintos caracteres que las componen. Es decir, devolverá un 1 si una cadena es un anagrama de la otra. Por ejemplo, la cadena «ramo» es un anagrama de «mora».

La subrutina «`subr1`» utilizará las subrutinas «`subr2`» y «`subr3`». La subrutina «`subr2`» deberá calcular cuántos caracteres de cada tipo tiene la cadena que se le pasa como parámetro. Por otra lado, la subrutina «`subr3`» devolverá un 1 si el contenido de los dos vectores que se le pasa como parámetros son iguales.

Suponer que las cadenas están compuestas por el conjunto de letras que componen el abecedario en minúsculas.

▶ **6.33** Desarrolla en ensamblador la siguiente subrutina recursiva descrita en lenguaje Python3:

```
1 def ncsr(n, k):
2       if k > n:
3           return 0
4       elif n == k or k == 0:
5           return 1
6       else:
7           return ncsr(n-1, k) + ncsr(n-1, k-1)
```

▶ **6.34** Desarrolla un programa en ensamblador que calcule el máximo de un vector cuyos elementos se obtienen como la suma de los elementos fila de una matriz de dimensión $n \times n$. El programa debe tener la siguiente estructura:

- Deberá estar compuesto por 3 subrutinas: «`subr1`», «`subr2`» y «`subr3`».
- «`subr1`» calculará el máximo buscado. Se le pasará como parámetros la matriz, su dimensión y devolverá el máximo buscado.
- «`subr2`» calculará la suma de los elementos de un vector. Se le pasará como parámetros un vector y su dimensión y la subrutina devolverá la suma de sus elementos.
- «`subr3`» calculará el máximo de los elementos de un vector. Se le pasará como parámetros un vector y su dimensión y devolverá el máximo de dicho vector.
- El programa principal se encargará de realizar la inicialización de la dimensión de la matriz y de sus elementos y llamará a la subrutina «`subr1`», quién devolverá el máximo buscado. El programa principal deberá almacenar este dato en la posición etiquetada con «`max`».

Bibliografía

[Adv95] Advanced RISC Machines Ltd (ARM) (1995). *ARM 7TDMI Data Sheet*.
URL http://www.ndsretro.com/download/ARM7TDMI.pdf

[Atm11] Atmel Corporation (2011). *ATmega 128: 8-bit Atmel Microcontroller with 128 Kbytes in-System Programmable Flash*.
URL http://www.atmel.com/Images/doc2467.pdf

[Atm12] Atmel Corporation (2012). *AT91SAM ARM-based Flash MCU datasheet*.
URL http://www.atmel.com/Images/doc11057.pdf

[Bar14] S. Barrachina Mir, G. León Navarro y J. V. Martí Avilés (2014). *Conceptos elementales de computadores*.
URL http://lorca.act.uji.es/docs/conceptos_elementales_de_computadores.pdf

[Cle14] A. Clements (2014). *Computer Organization and Architecture. Themes and Variations. International edition.* Editorial Cengage Learning. ISBN 978-1-111-98708-4.

[Shi13] S. Shiva (2013). *Computer Organization, Design, and Architecture, Fifth Edition*. Taylor & Francis. ISBN 9781466585546.
URL http://books.google.es/books?id=m5KlAgAAQBAJ