CAPÍTULO 4

Modos de direccionamiento y formatos de instrucción

Índice

Como se ha visto en capítulos anteriores, una instrucción en ensamblador indica qué operación se debe realizar, con qué operandos y dónde se debe guardar el resultado. En cuanto a los operandos, se ha visto que éstos pueden estar: I) en la propia instrucción, II) en un registro, o III) en la memoria principal. Con respecto al resultado, se ha visto que se puede almacenar: 1. en un registro, o 2. en la memoria principal.

Puesto que los operandos fuente de la instrucción deben codificarse en la instrucción, y dichos operandos pueden estar en la misma instrucción, en un registro, o en memoria, sería suficiente dedicar ciertos bits de la instrucción para indicar para cada operando fuente: I) el valor del operando, II) el registro en el que está, o III) la dirección de memoria en la que se encuentra. De igual forma, puesto que el resultado puede almacenarse en un registro o en memoria principal, bastaría con destinar otro conjunto de bits de la instrucción para codificar: I) el registro en el que debe guardarse el resultado, o II) la dirección de memoria en la que debe guardarse el resultado.

Sin embargo, es conveniente disponer de otras formas más elaboradas de indicar la dirección de los operandos [Bar14]. Principalmente por los siguientes motivos:

- Para ahorrar espacio de código. Cuanto más cortas sean las instrucciones máquina, menos espacio ocuparán en memoria, por lo que teniendo en cuenta que una instrucción puede involucrar a más de un operado, deberían utilizarse formas de indicar la dirección de los operandos que consuman el menor espacio posible.

- Para facilitar las operaciones con ciertas estructuras de datos. El manejo de estructuras de datos complejas (matrices, tablas, colas, listas, etc.) se puede simplificar si se dispone de formas más elaboradas de indicar la dirección de los operandos.

- Para poder reubicar el código. Si la dirección de los operandos en memoria solo se pudiera expresar por medio de una dirección de memoria fija, cada vez que se ejecutara un determinado programa, éste buscaría los operandos en las mismas direcciones de memoria, por lo que tanto el programa como sus datos habrían de cargarse siempre en las mismas direcciones de memoria. ¿Qué pasaría entonces con el resto de programas que el computador puede ejecutar? ¿También tendrían direcciones de memoria reservadas? ¿Cuántos programas distintos podría ejecutar un computador sin que éstos se solaparan? ¿De cuánta memoria dispone el computador? Así pues, es conveniente poder indicar la dirección de los operandos de tal forma que un programa pueda ejecutarse independientemente de la zona de memoria en la que haya sido cargado para su ejecución.

Por lo tanto, es habitual utilizar diversas formas, además de las tres ya comentadas, de indicar la *dirección efectiva* de los operandos fuente y del resultado de una instrucción. Las distintas formas en las que puede indicarse la dirección efectiva de los operandos y del resultado reciben el nombre de *modos de direccionamiento*.

Como se ha comentado al principio, una instrucción en ensamblador indica qué operación se debe realizar, con qué operandos y dónde se debe guardar el resultado. Queda por resolver cómo se codifica esa información en la secuencia de bits que conforman la instrucción. Una primera idea podría ser la de definir una forma de codificación única y general que pudiera ser utilizada por todas las instrucciones. Sin embargo, como ya se ha visto, el número de operandos puede variar de una instrucción a otra. De igual forma, como ya se puede intuir, el modo de direccionamiento empleado por cada uno de los operandos también puede variar de una instrucción a otra. Por tanto, si se intentara utilizar una forma de codificación única que englobara a todos los tipos de instrucciones, número de operandos y modos de direccionamiento, el tamaño de las instrucciones sería innecesariamente grande —algunos bits se utilizarían en unas instrucciones y en otras no, y al revés—.

Como no todas las instrucciones requieren el mismo tipo de información, una determinada arquitectura suele presentar diversas formas de organizar los bits que conforman una instrucción con el fin de optimizar el tamaño requerido por las instrucciones y aprovechar al máximo el tamaño disponible para cada instrucción. Las distintas formas en las que pueden codificarse las distintas instrucciones reciben el nombre de *formatos de instrucción* [Bar14]. Cada formato de instrucción define su tamaño y los campos que lo forman —cuánto ocupan, su orden y su significado—. Un formato de instrucción puede ser utilizado para codificar uno o varios tipos de instrucciones.

En este capítulo vamos a estudiar los modos de direccionamiento que forman parte de la arquitectura ARM. Además, como los modos de direccionamiento están relacionados con las instrucciones que los utilizan y los formatos de instrucción utilizados para codificar dichas instrucciones, iremos viendo para cada modo de direccionamiento algunas de las instrucciones que los utilizan y los formatos de instrucción utilizados para codificarlas.

Como referencia del juego de instrucciones Thumb de ARM y sus formatos de instrucción se puede utilizar el Capítulo 5 «*THUMB Instruction Set*» de [Adv95].

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.7 «*ARM Addressing Modes*» 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. También se puede consultar el Aparta-

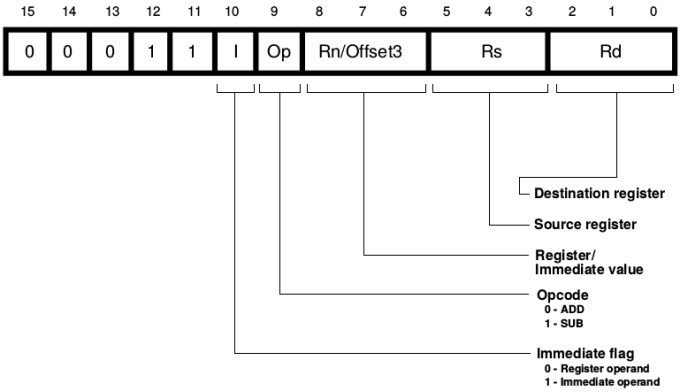

Figura 4.1: Formato de instrucción usado por las instrucciones «`add rd, rs, rn`» y «`sub rd, rs, rn`», entre otras

do 4.6.1 «*Thumb ISA*» de dicho libro para obtener más detalles sobre la arquitectura Thumb de ARM, en especial sobre los distintos formatos de instrucción.

4.1. Direccionamiento directo a registro

El direccionamiento directo a registro es el más simple de los modos de direccionamiento. En este modo de direccionamiento, el operando se encuentra en un registro. En la instrucción simplemente se debe codificar en qué registro se encuentra el operando.

Este modo de direccionamiento se utiliza en la mayor parte de instrucciones, tanto de acceso a memoria, como de procesamiento de datos, para algunos o todos sus operandos.

Se utiliza, por ejemplo, en las instrucciones «`add rd, rs, rn`» y «`sub rd, rs, rn`». Como se ha visto en el Capítulo 2, la instrucción «`add rd, rs, rn`» suma el contenido de los registros `rs` y `rn` y escribe el resultado en el registro `rd`. Por su parte, la instrucción «`sub rd, rs, rn`», resta el contenido de los registros `rs` y `rn` y almacena el resultado en el registro `rd`. Por lo tanto, las anteriores instrucciones utilizan el modo de direccionamiento directo a registro para especificar tanto sus dos operandos fuente, como para indicar dónde guardar el resultado.

El formato de instrucción utilizado para codificar estas instrucciones se muestra en la Figura 4.1. Como se puede ver, el formato de instrucción ocupa 16 bits y proporciona tres campos de 3 bits cada uno para codificar los registros `rd`, `rs` y `rn`, que ocupan los bits 2 al 0, 5 al 3 y 8 al 6, respectivamente. Puesto que los campos son de 3 bits, solo pueden codificarse en ellos los registros del `r0` al `r7`, que son los registros habitualmente disponibles para las instrucciones Thumb.

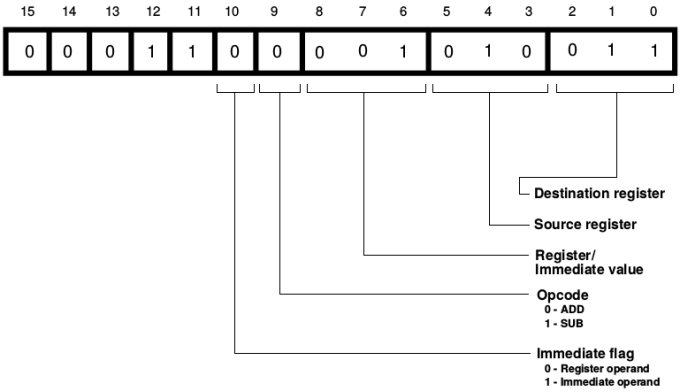

Figura 4.2: Codificación de la instrucción «`add r3, r2, r1`»

Así pues, la instrucción «`add r3, r2, r1`» se codificaría con la siguiente secuencia de bits: «`00011 0 0 001 010 011`», tal y como se desglosa en la Figura 4.2.

4.2. Direccionamiento inmediato

En el modo de direccionamiento inmediato, el operando está en la propia instrucción. Es decir, en la instrucción se debe codificar el valor del operando (aunque la forma de codificar el operando puede variar dependiendo del formato de instrucción —lo que normalmente está relacionado con para qué se va a utilizar dicho dato inmediato—).

Como ejemplo de instrucciones que usen este modo de direccionamiento están: «`add rd, rs, #0ffset3`» y «`sub rd, rs, #0ffset3`», que utilizan el modo direccionamiento inmediato en su segundo operando fuente.

Estas instrucciones utilizan de hecho el formato de instrucción ya presentado en la Figura 4.1, que como se puede ver permite codificar un dato inmediato, «`0ffset3`», pero de solo 3 bits. Por tanto, el dato inmediato utilizado en estas instrucciones deberá estar en el rango $[0, 7]$ (ya que el dato inmediato se codifica en este formato de instrucción como un número entero sin signo).

Conviene destacar que el campo utilizado en dicho formato de instrucción para codificar el dato inmediato en estas instrucciones es el mismo que se usa para codificar el registro `rn` en las instrucciones vistas en al apartado anterior. ¿Cómo puede saber el procesador cuando decodifica una instrucción en este formato si el número que hay en dicho campo es un registro o un dato inmediato? Consultando qué valor hay en el campo `I` (bit 10). Si el campo `I` vale 1, entonces el campo `rn/0ffset3`

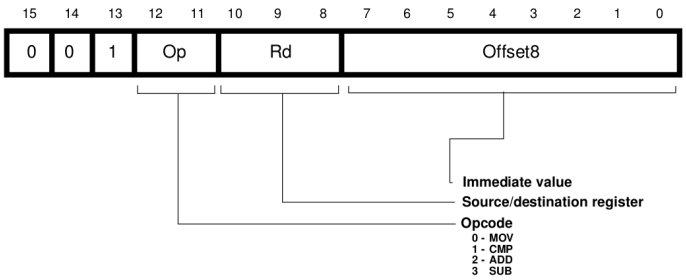

Figura 4.3: Formato de las instrucciones «`mov rd, #Offset8`», «`cmp rd, #Offset8`», «`add rd, #Offset8`» y «`sub rd, #Offset8`»

contiene un dato inmediato; si vale 0, entonces el campo `rn/Offset3` indica el registro en el que está el segundo operando fuente.

EJERCICIOS

▶ **4.1** ¿Qué instrucción se codifica como «`00011 1 0 001 010 011`»?

▶ **4.2** ¿Cuál será la codificación de la instrucción «`add r3, r4, r4`»?

▶ **4.3** ¿Cuál será la codificación de la instrucción «`add r3, r4, #5`»?

▶ **4.4** ¿Qué ocurre si se intenta ensamblar la siguiente instrucción: «`add r3, r4, #8`»? ¿Qué mensaje muestra el ensamblador? ¿A qué es debido?

Las siguientes instrucciones también utilizan el direccionamiento inmediato para especificar uno de los operandos: «`mov rd, #Offset8`», «`cmp rd, #Offset8`», «`add rd, #Offset8`» y «`sub rd, #Offset8`». Estas instrucciones se codifican utilizando el formato de instrucción mostrado en la Figura 4.3. Como se puede observar en dicha figura, el campo destinado en este caso para el dato inmediato, `Offset8`, ocupa 8 bits. Por tanto, el rango de números posibles se amplía en este caso a $[0, 255]$ (ya que al igual que en el formato de instrucción anterior, el dato inmediato se codifica en este formato de instrucción como un número sin signo).

A modo de ejemplo, y teniendo en cuenta el formato de instrucción de la Figura 4.3, la instrucción «`add r4, #45`», que suma el contenido del registro `r4` y el número 45 y almacena el resultado en el registro `r4`, se codificaría como: «`001 10 0 100 00101101`».

EJERCICIOS

▶ **4.5** ¿Cómo se codificará la instrucción «`sub r2, #200`»?

▶ **4.6** ¿Cómo se codificará la instrucción «`cmp r4, #42`»?

4.3. Direccionamiento relativo a registro con desplazamiento

En el modo de direccionamiento relativo a registro con desplazamiento, la dirección efectiva del operando es una dirección de memoria que se obtiene sumando el contenido de un registro y un desplazamiento especificado en la propia instrucción. Por tanto, si un operando utiliza este modo de direccionamiento, el formato de instrucción deberá proporcionar dos campos para dicho operando: uno para especificar el registro y otro para el desplazamiento inmediato.

Este tipo de direccionamiento se utiliza en las instrucciones de carga y almacenamiento en memoria para el operando fuente y destino, respectivamente. La instrucción de carga, «`ldr rd, [rb, #Offset5]`», realiza la operación $rd \leftarrow [rb + Offset5]$, por lo que consta de dos operandos. Uno es el operando destino, que es el registro `rd`. El modo de direccionamiento utilizado para dicho operando es el directo a registro. El otro operando es el operando fuente, que se encuentra en la posición de memoria cuya dirección se calcula sumando el contenido de un registro `rb` y un desplazamiento inmediato, `Offset8`. El modo de direccionamiento utilizado para dicho operando es el relativo a registro con desplazamiento.

Algo parecido ocurre con la instrucción «`str rd, [rb, #Offset5]`». Contesta las siguientes preguntas sobre dicha instrucción.

EJERCICIOS

▶ **4.7** ¿Cuántos operandos tiene dicha instrucción?

▶ **4.8** ¿Cuál es el operando fuente? ¿Cuál es el operando destino?

▶ **4.9** ¿Cómo se calcula la dirección efectiva del operando fuente? ¿Qué modo de direccionamiento se utiliza para dicho operando?

▶ **4.10** ¿Cómo se calcula la dirección efectiva del operando destino? ¿Qué modo de direccionamiento se utiliza para dicho operando?

El modo de direccionamiento relativo a registro con desplazamiento puede utilizarse por las siguientes instrucciones de carga y desplazamiento: «`ldr`», «`str`», «`ldrb`», «`strb`», «`ldrh`» y «`strh`». El formato de instrucción utilizado para codificar las cuatro primeras de las instrucciones anteriores se muestra en la Figura 4.4. Como se puede observar en dicha figura, el formato de instrucción está formado por los siguientes campos:

rd se utiliza para codificar el registro destino o fuente (dependiendo de si la instrucción es de carga o de almacenamiento, respectivamente).

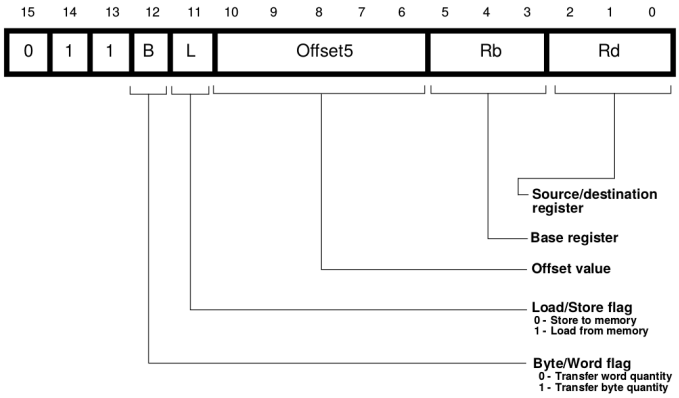

Figura 4.4: Formato de instrucción con direccionamiento relativo a registro con desplazamiento utilizado por las instrucciones «`ldr rd, [rb, #Offset5]`», «`str rd, [rb, #Offset5]`», «`ldrb rd, [rb, #Offset5]`» y «`strb rd, [rb, #Offset5]`»,

rb se utiliza para codificar el registro base, cuyo contenido se usa para calcular la dirección de memoria del segundo operando.

Offset5 se utiliza para codificar el desplazamiento que junto con el contenido del registro `rb` proporciona la dirección de memoria del segundo operando.

L se utiliza para indicar si se trata de una instrucción de carga ($L = 1$) o de almacenamiento ($S = 0$).

B se utiliza para indicar si se debe transferir un byte ($B = 1$) o una palabra ($B = 0$).

El desplazamiento inmediato `Offset5` codifica un número sin signo con 5 bits. Por tanto, dicho campo permite almacenar un número entre 0 y 31. En el caso de las instrucciones «`ldrb`» y «`strb`», que cargan y almacenan un byte, dicho campo codifica directamente el desplazamiento. Así por ejemplo, la instrucción «`ldrb r3, [r0,#31]`» carga en el registro `r3` el byte que se encuentra en la dirección de memoria dada por $r0 + 31$ y el número 31 se codifica tal cual en la instrucción: «11111_2».

Sin embargo, en el caso de las instrucciones de carga y almacenamiento de palabras es posible aprovechar mejor los 5 bits del campo si se tiene en cuenta que una palabra solo puede ser leída o escrita si su dirección de memoria es múltiplo de 4. Puesto que las palabras deben estar alineadas en múltiplos de 4, si no se hiciera nada al respecto, habría combinaciones de dichos 5 bits que no podrían utilizarse $(1, 2, 3, 5, 6, 7, 9, \ldots, 29, 30, 31)$. Por otro lado, aquellas combinaciones que sí serían válidas $(0, 4, 8, 12, \ldots, 24, 28)$, al ser múltiplos de 4 tendrían

los dos últimos bits siempre a 0 ($0 = 00000_2$, $4 = 000100_2$, $8 = 001000_2$, $12 = 001100_2 \ldots$). Además, el desplazamiento posible, si lo contamos en número de palabras, sería únicamente de $[0, 7]$, lo que limitaría bastante la utilidad de este modo de direccionamiento.

Teniendo en cuenta todo lo anterior, ¿cómo se podrían aprovechar mejor los 5 bits del campo `Offset5` en el caso de la carga y almacenamiento de palabras? Simplemente no malgastando los 2 bits de menor peso del campo `Offset5` para almacenar los 2 bits de menor peso del desplazamiento (que sabemos que siempre serán 0). Al hacerlo así, en lugar de almacenar los bits 0 al 4 del desplazamiento en el campo `Offset5`, se podrían almacenar los bits del 2 al 6 del desplazamiento en dicho campo. De esta forma estaríamos codificando 7 bits de desplazamiento utilizando únicamente 5 bits (ya que los 2 bits de menor peso son 0).

Cómo es fácil deducir, al proceder de dicha forma, no solo se aprovechan todas las combinaciones posibles de 5 bits, sino que además el rango del desplazamiento aumenta de $[0, 28]$ bytes a $[0, 124]$ (o lo que es lo mismo, de $[0, 7]$ palabras a $[0, 31]$ palabras).

Veamos con un ejemplo cómo se codificaría un desplazamiento de 20 bytes en una instrucción de carga de bytes y en otra de carga de palabras. En la de carga de bytes (p.e., «`ldrb r1, [r0,#20]`»), el número 20 se codificaría tal cual en el campo `Offset5`. Por tanto, en `Offset5` se pondría el número 20 (10100_2). Por el contrario, en una instrucción de carga de palabras (p.e., «`ldr r1, [r0,#20]`»), el número 20 se codificaría con 7 bits y se guardarían en el campo `Offset5` los bits 2 al 6. Puesto que $20 = 0010100_2$, en el campo `Offset5` se almacenaría el valor 00101_2 (o lo que es lo mismo, $20/4 = 5$).

EJERCICIOS

▶ **4.11** Utiliza el simulador para obtener la codificación de la instrucción «`ldrb r2, [r5,#12]`». ¿Cómo se ha codificado, en binario, el campo `Offset5`?

▶ **4.12** Utiliza el simulador para obtener la codificación de la instrucción «`ldr r2, [r5,#12]`». ¿Cómo se ha codificado, en binario, el campo `Offset5`?

▶ **4.13** Utiliza el simulador para obtener la codificación de la instrucción «`ldr r2, [r5,#116]`». ¿Cómo se ha codificado, en binario, el campo `Offset5`? ¿Cuál es la representación en binario con 7 bits del número 116?

▶ **4.14** Intenta ensamblar la instrucción «`ldrb r2, [r5,#116]`». ¿Qué mensaje de error proporciona el ensamblador? ¿A qué es debido?

▶ **4.15** Intenta ensamblar la instrucción «`ldr r2, [r5,#117]`». ¿Qué mensaje de error proporciona el ensamblador? ¿A qué es debido?

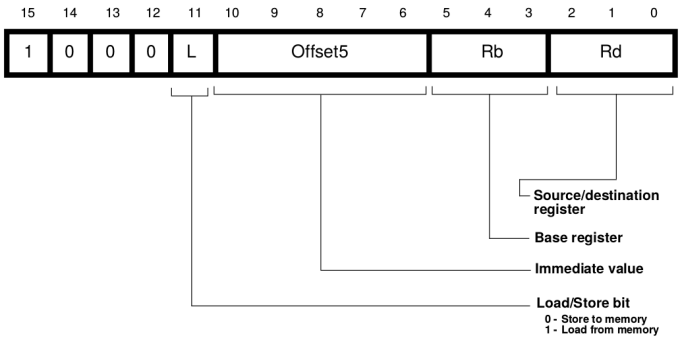

Figura 4.5: Formato de instrucción con direccionamiento relativo a registro con desplazamiento utilizado por las instrucciones «**ldrh** rd, [rb, #Offset5]» y «**strh** rd, [rb, #Offset5]»

Como se ha comentado antes, las instrucciones de carga y almacenamiento de medias palabras también pueden utilizar este modo de direccionamiento, el relativo a registro más desplazamiento. Sin embargo, estas instrucciones se codifican con un formato de instrucción (ver Figura 4.5) ligeramente distinto al de las instrucciones de carga y almacenamiento de bytes y palabras (tal y como se mostró en la Figura 4.4). Aunque como se puede observar, ambos formatos de instrucción tan solo se diferencian en los 4 bits de mayor peso[1]. En el caso del formato para bytes y palabras, los 3 bits más altos tomaban el valor 011_2, mientras que ahora valen 100_2. En cuanto al cuarto bit de mayor peso, el bit 12, en el caso del formato de instrucción para bytes y palabras, éste podía tomar como valor un 0 o un 1, mientras que en el formato de instrucción para medias palabras siempre vale 0.

En el caso de las instrucciones «**ldrh**» y «**strh**», el campo `Offset5` de solo 5 bits, siguiendo un razonamiento similar al descrito para el caso de las instrucciones de carga y almacenamiento de palabras, permite codificar un dato inmediato de 6 bits: los 5 bits de mayor peso se almacenan en el campo `Offset5` y el bit de menor peso no es necesario almacenarlo ya que siempre vale 0 (puesto que para poder leer o escribir una media palabra, ésta debe estar almacenada en una dirección múltiplo de 2). Así pues, los desplazamientos de las instrucciones de carga y almacenamiento de medias palabras estarán en el rango de $[0, 62]$ bytes (o lo que es lo mismo, de $[0, 31]$ medias palabras).

Otras instrucciones en las que también se utiliza el modo de direccionamiento relativo a registro con desplazamiento son las de carga relativa

[1] El campo formado por aquellos bits de la instrucción que codifican la operación a realizar —o el tipo de operación a realizar— recibe el nombre de *código de operación*.

al contador de programa[2], «`ldr rd, [PC, #Word8]`», y las de carga y almacenamiento relativo al puntero de pila, «`ldr rd, [SP, #Word8]`» y «`str rd, [SP, #Word8]`».

La instrucción «`ldr rd, [PC, #Word8]`» carga en el registro `rd` la palabra leída de la dirección de memoria especificada por la suma del `PC` + 4 alineado[3] a 4 y del dato inmediato `Word8`. Las instrucciones de acceso relativo al puntero de pila realizan una tarea similar (en este caso tanto para carga como para almacenamiento), pero apoyándose en el puntero de pila.

Como se puede ver en las Figuras 4.6 y 4.7, las anteriores instrucciones se codifican utilizando formatos de instrucción muy similares. Ambos formatos de instrucción proporcionan un campo `rd` en el que indicar el registro destino (o fuente en el caso de la instrucción «`str rd, [SP, #Word8]`») y un campo `Word8` de 8 bits donde se almacena el valor inmediato que se deberá sumar al contenido del registro `PC` o al del `SP`, dependiendo de la instrucción.

Puesto que dichas instrucciones cargan y almacenan palabras, el campo `Word8`, de 8 bits, se utiliza para codificar un dato inmediato de 10 bits, por lo que el rango del desplazamiento es de $[0, 1020]$ bytes (o lo que es lo mismo, de $[0, 255]$ palabras).

Es interesante observar que el registro que se utiliza como registro base, `PC` o `SP`, está implícito en el tipo de instrucción. Es decir, no se requiere un campo adicional para codificar dicho registro, basta con saber de qué instrucción se trata. Como ya se habrá intuido a estas alturas, los bits de mayor peso se utilizan para identificar la instrucción en cuestión. Así, y tal como se puede comprobar en las Figuras 4.6 y 4.7, si los 5 bits de mayor peso son 01001_2, se tratará de la instrucción «`ldr rd, [PC, #Word8]`»; si los 5 bits de mayor peso son 10011_2, de la instrucción «`ldr rd, [SP, #Word8]`»; y si los 5 bits de mayor peso valen 10010_2, de la instrucción «`str rd, [SP, #Word8]`».

EJERCICIOS

▶ **4.16** Con ayuda del simulador, obtén el valor de los campos `rd` y `Word8` de la codificación de la instrucción «`ldr r3,[pc,#844]`».

▶ **4.17** Con ayuda del simulador, obtén el valor de los campos `rd` y `Word8` de la codificación de la instrucción «`str r4,[sp,#492]`».

[2] Como se comentó en la introducción del capítulo, una de las ventajas de utilizar diversos modos de direccionamiento es la de conseguir código que pueda reubicarse en posiciones de memoria distintas. La carga relativa al contador de programa es un ejemplo de ésto.

[3] Cuando el procesador va a calcular la dirección del operando fuente de la instrucción «`ldr rd, [PC, #Word8]`» como la suma relativa al `PC` del desplazamiento `Word8`, el `PC` se ha actualizado ya a `PC`+4. Puesto que las instrucciones Thumb ocupan una o media palabra, `PC`+4 podría no ser múltiplo de 4, por lo que el bit 1 de `PC`+4 se pone a 0 para garantizar que dicho sumando sea múltiplo de 4 [Adv95].

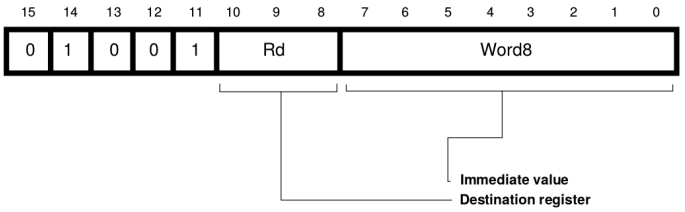

Figura 4.6: Formato de instrucción de carga relativa al contador de programa

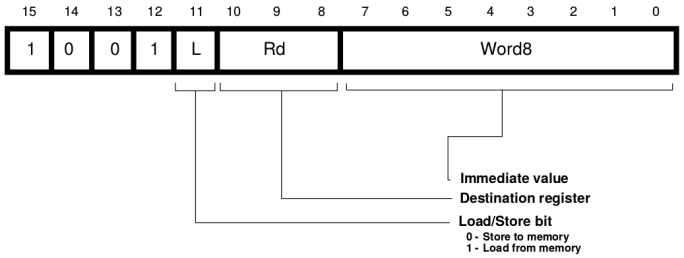

Figura 4.7: Formato de instrucción de carga y almacenamiento relativo al puntero de pila

4.4. Direccionamiento relativo a registro con registro de desplazamiento

En el modo de direccionamiento relativo a registro con registro de desplazamiento, la dirección efectiva del operando es una dirección de memoria que se obtiene sumando el contenido de dos registros. Por tanto, si un operando utiliza este modo de direccionamiento, el formato de instrucción deberá proporcionar dos campos para dicho operando: uno para cada registro. Como se puede ver, es muy similar al relativo a registro con desplazamiento. La diferencia radica en que el desplazamiento se obtiene de un registro en lugar de un dato inmediato.

Las instrucciones que utilizan este modo de direccionamiento son las instrucciones de carga y almacenamiento de palabras, medias palabras y bytes. En concreto, las instrucciones de carga que utilizan este modo de direccionamiento son:

- «`ldr rd, [rb, ro]`» Carga en el registro `rd` el contenido de la palabra de memoria indicada por la suma de los registros `rb` y `ro`.

- «**ldrh** rd, [rb, ro]» Carga en el registro rd el contenido de la media palabra de memoria indicada por la suma de los registro rb y ro. La media palabra de mayor peso del registro rd se rellenará con 0.
- «**ldsh** rd, [rb, ro]» Carga en el registro rd el contenido de la media palabra de memoria indicada por la suma de los registro rb y ro. La media palabra de mayor peso del registro rd se rellenará con el valor del bit 15 de la media palabra leída.
- «**ldrb** rd, [rb, ro]» Carga en el registro rd el contenido del byte de memoria indicado por la suma de los registros rb y ro. Los tres bytes de mayor peso del registro rd se rellenarán con 0.
- «**ldsb** rd, [rb, ro]» Copia en el registro rd el contenido del byte de memoria indicado por la suma de los registro rb y ro. Los tres bytes de mayor peso del registro rd se rellenarán con el valor del bit 7 del byte leído.

Conviene destacar que, como se puede observar en la relación anterior, las instrucciones de carga de medias palabras y bytes proporcionan dos variantes. Las instrucciones de la primera variante, «**ldrh**» y «**ldrb**», no tienen en cuenta el signo del dato leído y completan la palabra con 0. Por el contrario, las instrucciones de la segunda variante, «**ldsh**» y «**ldsb**», sí que tienen en cuenta el signo y extienden el signo del dato leído a toda la palabra.

Volviendo a las instrucciones de carga y almacenamiento que utilizan este modo de direccionamiento, las instrucciones de almacenamiento que lo soportan son:

- «**str** rd, [rb, ro]» Almacena el contenido del registro rd en la palabra de memoria indicada por la suma de los registros rb y ro.
- «**strh** rd, [rb, ro]» Almacena el contenido de la media palabra de menor peso del registro rd en la media palabra de memoria indicada por la suma de los registro rb y ro.
- «**strb** rd, [rb, ro]» Almacena el contenido del byte de menor peso del registro rd en el byte de memoria indicado por la suma de los registros rb y ro.

El formato de instrucción utilizado para codificar las instrucciones «**ldr**», «**ldrb**», «**str**» y«**strb**», se muestra en la Figura 4.8. Por otro lado, el formato utilizado para las instrucciones «**ldrh**», «**ldsh**», «**ldsb**» y «**strh**» se muestra en la Figura 4.9. Como se puede ver, en ambos formatos de instrucción aparecen los campos ro y rb, en los que se indican los registros cuyo contenido se va a sumar para obtener la dirección

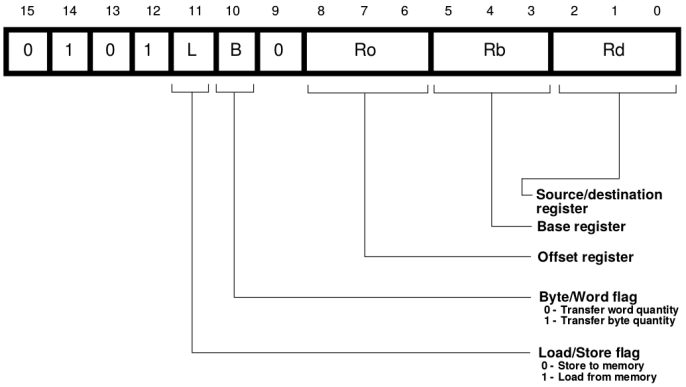

Figura 4.8: Formato de instrucción usado para codificar las instrucciones «**ldr**», «**ldrb**», «**str**» y «**strb**» que utilizan el modo de direccionamiento relativo a registro con registro de desplazamiento

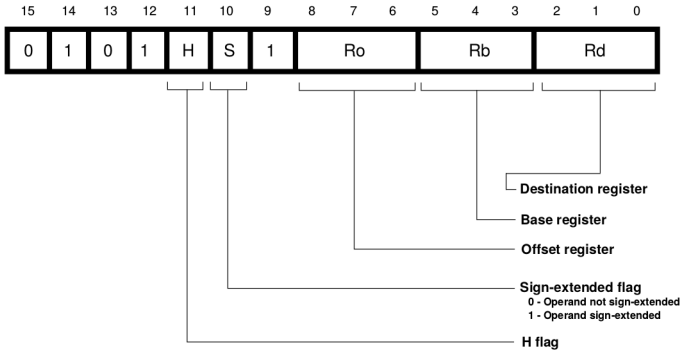

Figura 4.9: Formato de instrucción usado para codificar las instrucciones «**ldrh**», «**ldsh**», «**ldsb**» y «**strh**» que utilizan el modo de direccionamiento relativo a registro con registro de desplazamiento

de memoria del operando. También se puede ver que en ambas figuras aparece el campo `rd`, en el que se indica el registro en el que cargar el dato leído de memoria (en las instrucciones de carga) o del que se va a leer su contenido para almacenarlo en memoria (en las instrucciones de almacenamiento).

EJERCICIOS

▶ **4.18** Cuando el procesador lee una instrucción de uno de los formatos descritos en las Figuras 4.8 y 4.9, ¿cómo podría distinguir de qué formato de instrucción se trata?

▶ **4.19** Con ayuda del simulador comprueba cuál es la diferencia entre la codificación de la instrucción «**ldrh** `r3,[r1,r2]`» y la instrucción «**ldsh** `r3,[r1,r2]`».

▶ **4.20** Codifica a mano la instrucción «`ldsh r5, [r1,r3]`». Comprueba con ayuda del simulador que has realizado correctamente la codificación.

4.5. Direccionamiento en las instrucciones de salto incondicional y condicional

Uno de los operandos de las instrucciones de salto incondicional, «`b etiqueta`», y condicional, «`bXX etiqueta`», (ver Apartado 3.2) es justamente la dirección de salto. Como se puede ver en las instrucciones anteriores, la dirección de salto se identifica en ensamblador por medio de una etiqueta que apunta a la dirección de memoria a la que se quiere saltar.

¿Cómo se codifica dicha dirección de memoria en una instrucción de salto? Puesto que las direcciones de memoria en ARM ocupan 32 bits, si se quisieran codificar los 32 bits del salto en la instrucción, sería necesario recurrir a instrucciones que ocuparan más de 32 bits (al menos para las instrucciones de salto, ya que no todas las instrucciones tienen por qué ser del mismo tamaño).

Pero además, si se utilizara esta aproximación, la de codificar la dirección completa del salto como un valor absoluto, se forzaría a que el código se tuviera que ejecutar siempre en las mismas direcciones de memoria. Esta limitación se podría evitar durante la fase de carga del programa. Bastaría con que el programa cargador, cuando cargue un programa para su ejecución, sustituya las direcciones de salto absolutas por nuevas direcciones que tengan en cuenta la dirección de memoria a partir de la cual se esté cargando el código [Shi13]. En cualquier caso, esto implica que el programa cargador debe saber dónde hay saltos absolutos en el código, cómo calcular la nueva dirección de salto y sustituir las direcciones de salto originales por las nuevas.

Para que las instrucciones de salto sean pequeñas y el código sea directamente reubicable en su mayor parte, en lugar de *saltos absolutos* se suele recurrir a utilizar *saltos relativos*. Bien, pero, ¿relativos a qué? Para responder adecuadamente a dicha pregunta conviene darse cuenta de que la mayor parte de las veces, los saltos se realizan a una posición cercana a aquella instrucción desde la que se salta (p.e., en estructuras *if-then-else*, en bucles *while* y *for*...). Por tanto, el contenido del registro `PC`, que tiene la dirección de la siguiente instrucción a la actual, se convierte en el punto de referencia idóneo y los saltos relativos se codifican como saltos relativos al registro `PC`.

La arquitectura Thumb de ARM no es una excepción. De hecho, en dicha arquitectura tanto los saltos incondicionales como los condicionales

Figura 4.10: Formato de instrucción utilizado para codificar la instrucción «`b etiqueta`»

utilizan el modo de direccionamiento relativo al PC para codificar la dirección de salto.

El formato de instrucción utilizado por la instrucción de salto incondicional, «`b etiqueta`», se muestra en la Figura 4.10. Como se puede ver, el campo destinado a codificar el desplazamiento, `Offset11`, consta de 11 bits. Para poder aprovechar mejor dicho espacio, se utiliza la misma técnica ya comentada en el Apartado 4.3. Puesto las instrucciones Thumb ocupan 16 o 32 bytes, el número de bytes del desplazamiento va a ser siempre un número par y, por tanto, el último bit del desplazamiento va a ser siempre 0. Como se sabe de antemano el valor de dicho bit, no es necesario guardarlo en el campo `Offset11`, lo que permite guardar los bits 1 al 11 del desplazamiento. Pudiendo codificar, por tanto, el desplazamiento con 12 bits (en lugar de con 11), lo que proporciona un rango de salto de $[-2048, 2046]$ bytes[4] con respecto al `PC`.

Hasta ahora sabemos que la dirección de memoria a la que se quiere saltar se codifica como un desplazamiento de 12 bits con respecto al contenido del registro `PC`. Sin embargo, no hemos dicho mucho del contenido del registro `PC`. Teóricamente, el contenido del registro `PC` se actualiza al valor `PC`+2 al comienzo de la ejecución de la instrucción «`b etiqueta`», puesto que la instrucción «`b etiqueta`» ocupa 2 bytes. Por tanto, el valor del desplazamiento debería ser tal que al sumarse a `PC`+2 diera la dirección de salto.

En la práctica, cuando el procesador va a ejecutar el salto, el contenido del `PC` se ha actualizado al valor del `PC`+4. Esto es debido a que se utiliza una técnica conocida como precarga de instrucciones. Por tanto, en realidad el desplazamiento debe ser tal que al sumarse al `PC`+4 proporcione la dirección de memoria a la que se quiere saltar.

Para ilustrar lo anterior, fíjate que el siguiente código está formado por varias instrucciones que saltan al mismo sitio:

`mod-dir-b.s` ✍

[4]Si el bit 0 no se considerara como un bit implícito a la instrucción, el rango del desplazamiento correspondería al rango del complemento a 2 con 11 bits para números pares, es decir, $[-1024, 1022]$ bytes con respecto al `PC`.

```
1         .text
2 main:   b  salto
3         b  salto
4         b  salto
5         b  salto
6 salto:  mov r0, r0
7         mov r1, r1
8         b   salto
9         b   salto
10
11 stop:  wfi
```

EJERCICIOS

Copia el programa anterior y ensámblalo. No lo ejecutes (el programa no tiene ningún sentido y si se ejecutara entraría en un bucle sin fin, su único propósito es mostrar qué desplazamiento se almacena en cada caso).

▶ **4.21** ¿Cuál es la dirección memoria de la instrucción etiquetada con «`salto`»?

▶ **4.22** Según la explicación anterior, cuando se va a realizar el salto desde la primera instrucción, ¿qué valor tendrá el registro `PC`?, ¿qué número se ha puesto como desplazamiento?, ¿cuánto suman?

▶ **4.23** ¿Cuánto vale el campo `Ofsset11` de la primera instrucción? ¿Qué relación hay entre el desplazamiento y el valor codificado de dicho desplazamiento?

▶ **4.24** Observa con detenimiento las demás instrucciones, ¿qué números se han puesto como desplazamiento en cada uno de ellas? Comprueba que al sumar dicho desplazamiento al `PC`+4 se consigue la dirección de la instrucción a la que se quiere saltar.

Las instrucciones de salto condicional se codifican de forma similar a como se codifican las de salto incondicional. La única diferencia es que el formato de instrucción utilizado para codificar dichas instrucciones (ver Figura 4.11), tiene un campo, `SOffset8`, de 8 bits (en lugar de los 11 disponibles en el caso del salto incondicional). Esto es debido a que este formato de instrucción debe proporcionar un campo adicional, `Cond`, para codificar la condición del salto, por lo quedan menos bits disponibles para codificar el resto de la instrucción.

Puesto que el campo `SOffset` solo dispone de 8 bits, el desplazamiento que se puede codificar en dicho campo tendrá como mucho 9 bits en complemento a 2. Por tanto, el rango del desplazamiento de los saltos condicionales está limitado a $[-512, 510]$ con respecto al `PC+4`.

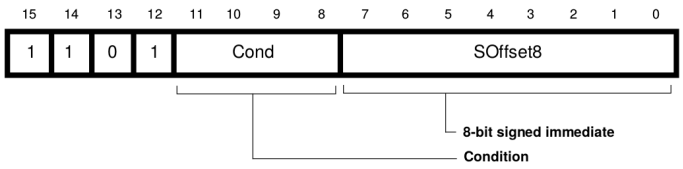

Figura 4.11: Formato de instrucción utilizado para codificar las instrucciones de salto condicional

4.6. Ejercicios del capítulo

EJERCICIOS

▶ **4.25** Supón un vector, V, de 26 palabras y dos variables x e y asignadas a los registros `r0` y `r1`. Asume que la dirección base del vector V se encuentra en el registro `r2`. Escribe el código ensamblador que implementaría la siguiente operación: $x = V[25] + y$.

▶ **4.26** Supón un vector, V, de 26 palabras y una variable y asignada al registro `r1`. Asume que la dirección base del vector V se encuentra en el registro `r2`. Escribe el código ensamblador que implementaría la siguiente operación: $V[10] = V[25] + y$.

▶ **4.27** Escribe un código en ensamblador que dado un vector, V, de n bytes, almacene en el registro `r1` la posición del primer elemento de dicho vector que es un 1. Debes realizar este ejercicio sin utilizar el direccionamiento relativo a registro con registro de desplazamiento.

Comprueba el funcionamiento del programa inicializando el vector V a $[0, 0, 1, 0, 1, 0, 1, 0, 0, 0]$ y, por tanto, n a 10.

▶ **4.28** Escribe un código en ensamblador que dado un vector, V, de n bytes, almacene en el registro `r1` la posición del último elemento de dicho vector que es un 1. Debes realizar este ejercicio sin utilizar el direccionamiento relativo a registro con registro de desplazamiento.

Comprueba el funcionamiento del programa inicializando el vector V a $[0, 0, 1, 0, 1, 0, 1, 0, 0, 0]$ y, por tanto, n a 10.

▶ **4.29** Escribe una nueva versión del programa del ejercicio 4.27, pero esta vez utilizando el direccionamiento relativo a registro con registro de desplazamiento.

▶ **4.30** Escribe una nueva versión del programa del ejercicio 4.28, pero esta vez utilizando el direccionamiento relativo a registro con registro de desplazamiento.

▶ **4.31** Escribe un código ensamblador cuya primera instrucción sea «`ldr r2, [r5, #124]`»; el resto del código deberá obtener a partir de la

codificación de esa instrucción, el campo `Offset5` y guardarlo en la parte alta del registro `r3`.

El formato utilizado para codificar «`ldr r2, [r5, #124]`» se puede consultar en la Figura 4.4.

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