CAPÍTULO 2

Instrucciones de procesamiento de datos

Índice

En el capítulo anterior se ha visto cómo inicializar determinadas posiciones de memoria con determinados valores, cómo cargar valores de la memoria a los registros del procesador y cómo almacenar en memoria la información contenida en los registros.

Traduciendo lo anterior a las acciones que habitualmente realiza un programa, se ha visto cómo definir e inicializar las variables del programa, cómo transferir el contenido de dichas variables de memoria a los registros, para así poder realizar las operaciones que se quiera llevar a cabo, y, finalmente, cómo transferir el contenido de los registros a memoria, para almacenar el resultado de las operaciones realizadas.

Lo que no se ha visto todavía es qué operaciones se pueden llevar a cabo. En éste y en el siguiente capítulo se verán algunas de las operaciones

básicas más usuales que puede realizar el procesador, y las instrucciones que se utilizan para indicar dichas operaciones.

Este capítulo se centra en las instrucciones proporcionadas por ARM para la realización de operaciones aritméticas, lógicas y de desplazamiento de bits.

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.5 «*ARM Data-processing Instructions*» 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.

2.1. Operaciones aritméticas

Para introducir las operaciones aritméticas que puede realizar la arquitectura ARM, el siguiente programa muestra cómo sumar un operando almacenado originalmente en memoria y un valor constante proporcionado en la propia instrucción de suma. Observa que para realizar la suma, es necesario cargar en primer lugar el valor que se quiere sumar en un registro desde la posición de memoria en la que se encuentra almacenado.

oper-add-inm.s

```
1         .data             @ Zona de datos
2 num:    .word 2147483647  @ Máx. positivo representable en Ca2(32)
3                           @ (en hexadecimal 0x7fff ffff)
4
5         .text             @ Zona de instrucciones
6 main:   ldr r0, =num
7         ldr r0, [r0]      @ r0 <- [num]
8         add r1, r0, #1    @ r1 <- r0 + 1
9
10 stop:  wfi
```

La instrucción «**add** `rd, rs, #Inm3`» suma dos operandos. Uno de los operandos está almacenado en un registro, en `rs`, y el otro en la propia instrucción, en el campo `Inm3`; el resultado se almacenará en el registro `rd`. «**add**» *(inm)*

Por su parte, la instrucción «**sub** `rd, rs, #Inm3`» resta el dato inmediato «`Inm3`» del contenido del registro `rs` y almacena el resultado en `rd`. «**sub**» *(inm)*

Hay que tener en cuenta que puesto que el campo destinado al dato inmediato es de solo 3 bits, solo se pueden utilizar estas instrucciones si el dato inmediato es un número entre 0 y 7.

Existe una variante de las instrucciones suma y resta con dato inmediato que utilizan el mismo registro como fuente y destino de la operación: «`add rd, #Inm8`» y «`sub rd, #Inm8`». Estas variantes permiten que el dato inmediado sea de 8 bits, por lo que se puede indicar un número entre 0 y 255.

EJERCICIOS

Copia el fichero anterior en el simulador, ensámblalo y ejecútalo.

▶ **2.1** Localiza el resultado de la suma. ¿Cuál ha sido? ¿El resultado obtenido es igual a $2.147.483.647 + 1$? (puedes utilizar `kcalc`, `python3` o modificar a mano el contenido de un registro para comprobarlo).

▶ **2.2** Localiza en la parte inferior del simulador el valor de los indicadores (*flags*). Comprueba que aparece lo siguiente: `N z c V`, lo que quiere decir que se han activado los indicadores `N` y `V`. ¿Qué significa que se hayan activado los indicadores `N` y `V`? (si dejas el ratón sobre el panel de los indicadores, se mostrará un pequeño mensaje de ayuda).

▶ **2.3** Recarga el simulador, escribe el valor «`0x7FFFFFFE`» en la posición de memoria `0x20070000`, y vuelve a ejecutar el código. ¿Se ha activado algún indicador? ¿Por qué no?

▶ **2.4** Vuelve al modo de edición y sustituye en el programa anterior la instrucción «`add r1, r0, #1`» por la instrucción «`add r1, r0, #8`», guarda el fichero y pasa al modo de simulación. ¿Qué ha ocurrido al efectuar este cambio? ¿Por qué?

▶ **2.5** Vuelve al modo de edición y modifica el programa original para que: I) la posición de memoria etiquetada con «`num`» se inicialice con el número 10, y II) en lugar de la suma con dato inmediato se realice la siguiente resta con dato inmediato: $r1 \leftarrow r0 - 2$. Una vez realizado lo anterior, guarda el nuevo fichero, vuelve al modo de simulación y ejecuta el programa, ¿qué valor hay ahora en `r1`?

Las operaciones aritméticas en las que uno de los operandos es una constante aparecen con relativa frecuencia, p.e., para decrementar en uno un determinado contador «`nvidas = nvidas - 1`». Sin embargo, es más habitual encontrar instrucciones en las que los dos operandos fuente sean variables. Esto se hace en ensamblador con instrucciones en las que se utiliza como segundo operando fuente un registro en lugar de un dato inmediato. Así, para sumar el contenido de dos registros, se puede utilizar la instrucción «`add rd, rs, rn`», que suma el contenido de los «`add`»

registros `rs` y `rn`, y almacena el resultado en `rd`.

Por su parte, la instrucción «**`sub rd, rs, rn`**», resta el contenido de `rn` del contenido de `rs` y almacena el resultado en `rd`. «**`sub`**»

El siguiente programa muestra un ejemplo en el que se restan dos variables, almacenadas en «`num1`» y «`num2`», y se almacena el resultado en una tercera variable, etiquetada con «`res`».

oper-sub.s

```
1         .data           @ Zona de datos
2  num1:  .word 10
3  num2:  .word 6
4  res:   .space 4
5
6         .text           @ Zona de instrucciones
7  main:  ldr r0, =num1
8         ldr r0, [r0]    @ r0 <- [num1]
9         ldr r1, =num2
10        ldr r1, [r1]    @ r1 <- [num2]
11        sub r2, r0, r1  @ r2 <- [num1] - [num2]
12        ldr r3, =res
13        str r2, [r3]    @ [res] <- [num1] - [num2]
14
15 stop:  wfi
```

EJERCICIOS

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

▶ **2.6** ¿Qué posición de memoria se ha inicializado con el número 10? ¿Qué posición de memoria se ha inicializado con el número 6?

▶ **2.7** ¿Qué hace el código anterior? ¿A qué dirección de memoria hace la referencia la etiqueta «`res`»? ¿Qué resultado se almacena en la dirección de memoria etiquetada como «`res`» cuando se ejecuta el programa? ¿Es correcto?

▶ **2.8** ¿Qué dos instrucciones se han utilizado para almacenar el resultado en la posición de memoria etiquetada con «`res`»?

▶ **2.9** Recarga el simulador y ejecuta el programa paso a paso hasta la instrucción «**`sub r2, r0, r1`**» inclusive. ¿Se ha activado el indicador `Z`? ¿Qué significado tiene dicho indicador?

▶ **2.10** Recarga de nuevo el simulador y modifica a mano el contenido de las posiciones de memoria `0x2007 0000` y `0x2007 0004` para que tengan el mismo valor, p.e., un 5. Ejecuta de nuevo el programa paso a paso hasta la instrucción «**`sub r2, r0, r1`**» inclusive. ¿Se ha activado ahora el indicador `Z`? ¿Por qué?

«**cmp**»

Otra operación aritmética que se utiliza con frecuencia es la comparación. Dicha operación compara el contenido de dos registros y modifica los indicadores en función del resultado. Equivale a una operación de resta, con la diferencia de que su resultado no se almacena. Como se verá en el capítulo siguiente, la operación de comparación se suele utilizar para modificar el flujo del programa en función de su resultado.

La instrucción «**cmp** `r0, r1`» resta el contenido del registro `r1` del contenido del registro `r0` y activa los indicadores correspondientes en función del resultado obtenido.

El siguiente programa muestra un ejemplo con varias instrucciones de comparación.

oper-cmp.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #10
3         mov r1, #6
4         mov r2, #6
5         cmp r0, r1
6         cmp r1, r0
7         cmp r1, r2
8
9 stop:   wfi
```

EJERCICIOS

Copia el programa anterior en el simulador y realiza los siguientes ejercicios.

▶ **2.11** Ejecuta paso a paso el programa hasta la instrucción «**cmp** `r0, r1`» inclusive. ¿Se ha activado el indicador `C`?
Nota: En el caso de la resta, el indicador `C` *se utiliza para indicar si el resultado cabe en una palabra (*`C` *activo) o, por el contrario, si no cabe en una palabra (*`c` *inactivo), como cuando se dice* «nos llevamos una» *cuando restamos a mano.*

▶ **2.12** Ejecuta la siguiente instrucción, «**cmp** `r1, r0`». ¿Qué indicadores se han activado? ¿Por qué?

▶ **2.13** Ejecuta la última instrucción, «**cmp** `r1, r2`». ¿Qué indicadores se han activado? ¿Por qué?

«**neg**»
«**mvn**»

Otras de las operaciones aritméticas que puede realizar un procesador son el cambio de signo y el complemento bit a bit de un número. La instrucción que permite cambiar el signo de un número es «**neg** `rd, rs`» ($rd \leftarrow -rs$). La instrucción que permite obtener el complemento bit a bit de un número es «**mvn** `rd, rs`» ($rd \leftarrow NOT\ rs$).

El siguiente programa muestra un ejemplo en el que se utilizan ambas instrucciones.

oper-neg.s

```
1         .data           @ Zona de datos
2  num1:  .word 10
3  res1:  .space 4
4  res2:  .space 4
5
6         .text           @ Zona de instrucciones
7  main:  ldr r0, =num1
8         ldr r0, [r0]    @ r0 <- [num1]
9
10        @ Cambio de signo
11        neg r1, r0      @ r1 <- -r0
12        ldr r2, =res1
13        str r1, [r2]    @ [res1] <- -[num1]
14
15        @ Complementario
16        mvn r1, r0      @ r1 <- NOT r0
17        ldr r2, =res2
18        str r1, [r2]    @ [res2] <- NOT [num1]
19
20 stop:  wfi
```

EJERCICIOS

Copia el programa anterior en el simulador, ensámblalo y realiza los siguientes ejercicios.

▶ **2.14** Ejecuta el programa, ¿qué valor se almacena en «`res1`»?, ¿y en «`res2`»?

▶ **2.15** Completa la siguiente tabla I) recargando el simulador cada vez; II) modificando a mano el contenido de la posición de memoria etiquetada con «`num`» con el valor indicado en la primera columna de la tabla; y III) volviendo a ejecutar el programa. Sigue el ejemplo de la primera línea.

Valor	[num1]	[res1]	[res2]
10	`0x0000000A`	`0xFFFFFFF6`	`0xFFFFFFF5`
-10			
0			
−252645136			

▶ **2.16** Observando los resultados de la tabla anterior, ¿hay alguna relación entre el número con el signo cambiado [*res*1] y el número complementado [*res*2]? ¿cuál?

La última de las operaciones aritméticas que vamos a ver es la multiplicación. El siguiente programa muestra un ejemplo en el que se utiliza la instrucción «`mul rd, rm, rn`», que multiplica el contenido de `rm` y `rn` y almacena el resultado en `rd`, que forzosamente tiene que ser `rm` o `rn`. De hecho, puesto que el registro destino debe coincidir con uno de los registros fuente, también es posible escribir la instrucción de la forma «`mul rm, rn`», donde `rm` es el registro en el que se almacena el resultado de la multiplicación. «`mul`»

oper-mul.s

```
1        .data           @ Zona de datos
2  num1: .word 10
3  num2: .word 6
4  res:  .space 4
5
6        .text           @ Zona de instrucciones
7  main: ldr r0, =num1
8        ldr r0, [r0]    @ r0 <- [num1]
9        ldr r1, =num2
10       ldr r1, [r1]    @ r1 <- [num2]
11       mul r1, r0, r1  @ r1 <- [num1] * [num2]
12       ldr r3, =res
13       str r1, [r3]    @ [res] <- [num1] * [num2]
14
15 stop: wfi
```

EJERCICIOS

▶ **2.17** Ejecuta el programa anterior y comprueba que en la posición de memoria etiquetada con «`res`» se ha almacenado el resultado de 10×6.

▶ **2.18** Vuelve al modo de edición y modifica el programa sustituyendo la instrucción «`mul r1, r0, r1`» por una igual pero en la que el registro destino no sea ni `r0`, ni `r1`. Intenta ensamblar el código, ¿qué ocurre?

▶ **2.19** Modifica el programa original sustituyendo «`mul r1, r0, r1`» por una instrucción equivalente que utilice la variante con dos registros de la multiplicación. Ejecuta el código y comprueba si el resultado es correcto.

2.2. Operaciones lógicas

La arquitectura ARM proporciona las siguientes instrucciones que permiten realizar las operaciones lógicas «y» (*and*), «o» (*or*), «o exclusiva» (*eor*) y «complemento» (*not*), respectivamente:

- «**and** `rd, rs`», $rd \leftarrow rd\ AND\ rs$ (y).
- «**orr** `rd, rs`», $rd \leftarrow rd\ OR\ rs$ (o).
- «**eor** `rd, rs`»: $rd \leftarrow rd\ EOR\ rs$ (o exclusiva).
- «**mvn** `rd, rs`»: $rd \leftarrow NOT\ rs$ (complemento).

La instrucción «**mvn**», que permite obtener el complemento bit a bit de un número, ya se ha descrito previamente (ver página 47).

En cuanto a las operaciones lógicas «y», «o» y «o exclusiva», éstas toman como operandos dos palabras y realizan la operación lógica correspondiente bit a bit. Así, por ejemplo, la instrucción «**and** `r0, r1`» almacena en `r0` una palabra en la que su bit 0 es la «y» de los bits 0 de los dos operandos fuente, el bit 1 es la «y» de los bits 1 de los dos operandos fuente, y así sucesivamente.

La tabla de verdad de la operación «y» para dos bits a y b es la siguiente:

a	b	$a\ y\ b$
0	0	0
0	1	0
1	0	0
1	1	1

Como se puede ver, solo cuando los dos bits, a y b, valen 1, el resultado es 1.

Por otro lado, también se puede describir el funcionamiento de la operación «y» en función del valor de uno de los bits. Así, si b vale 0, $a\ y\ b$ será 0, y si b vale 1, $a\ y\ b$ tomará el valor de a. Si expresamos lo anterior en forma de tabla de verdad, quedaría:

b	$a\ y\ b$
0	0
1	a

Así pues, y suponiendo que los registros `r0` y `r1` tuvieran los valores `0x0123 4567` y `0x0000 0070`, la instrucción «**and** `r0, r1`» realizaría la siguiente operación:

$$\begin{array}{r} 0000\,0001\ 0010\,0011\ 0100\,0101\ 0110\,0111_2 \\ \text{y}\ 0000\,0000\ 0000\,0000\ 0000\,0000\ 0111\,0000_2 \\ \hline 0000\,0000\ 0000\,0000\ 0000\,0000\ 0110\,0000_2 \end{array}$$

Como se puede observar, puesto que el segundo operando tiene todos sus bits a 0 excepto los bits 4, 5 y 6, los únicos bits del resultado que podrían ser distintos de 0 serían justamente dichos bits. Además, los bits 4, 5 y 6 del resultado tomarán el valor que tuvieran dichos bits en el primer operando. De hecho, si sustituyéramos el primer operando por otro número, podríamos observar el mismo comportamiento. Tan solo se mostrarían en el resultado los bits 4, 5 y 6 del nuevo número.

Cuando se utiliza una secuencia de bits con este fin, ésta suele recibir el nombre de *máscara de bits*; ya que sirve para «ocultar» determinados bits del otro operando, a la vez que permite «ver» los bits restantes.

máscara de bits

El siguiente programa implementa el ejemplo anterior:

oper-and.s

```
1         .data                   @ Zona de datos
2  num:   .word 0x01234567
3  mask:  .word 0x00000070
4  res:   .space 4
5
6         .text                   @ Zona de instrucciones
7  main:  ldr r0, =num
8         ldr r0, [r0]            @ r0 <- [num]
9         ldr r1, =mask
10        ldr r1, [r1]            @ r1 <- [mask]
11        and r0, r1              @ r0 <- r0 AND r1
12        ldr r3, =res
13        str r0, [r3]            @ [res] <- [num] AND [mask]
14
15 stop:  wfi
```

EJERCICIOS

▶ **2.20** Carga el anterior programa en el simulador y ejecútalo. ¿Qué valor, expresado en hexadecimal, se almacena en la posición de memoria «`res`»? ¿Coincide con el resultado calculado en la explicación anterior?

▶ **2.21** Modifica el código para que sea capaz de almacenar en «`res`» el valor obtenido al conservar tal cual los 16 bits más significativos del dato almacenado en «`num`», y poner a cero los 16 bits menos significativos, salvo el bit cero, que también debe conservar su valor original. ¿Qué valor has puesto en la posición de memoria etiquetada con «`mask`»?

▶ **2.22** Desarrolla un programa, basado en los anteriores, que almacene en la posición de memoria «`res`» el valor obtenido al conservar el valor original de los bits 4, 5 y 6 del número almacenado en «`num`» y

ponga a 1 los bits restantes. *(Pista: La operación lógica «y» no sirve para este propósito, ¿que operación lógica se podría utilizar?)*

2.3. Operaciones de desplazamiento

La arquitectura ARM también proporciona instrucciones que permiten desplazar los bits del número almacenado en un registro un determinado número de posiciones a la derecha o a la izquierda.

El siguiente programa presenta la instrucción de desplazamiento aritmético a derechas (*arithmetic shift right*). La instrucción en cuestión, «**asr** `rd, rs`», desplaza hacia la derecha y conservando su signo, el valor almacenado en el registro `rd`, tantos bits como indique el contenido del registro `rs`. El resultado se almacena en el registro `rd`. A continuación se muestra un programa de ejemplo.

oper-asr.s

```
1         .data                   @ Zona de datos
2  num:   .word 0xffffff41
3  res:   .space 4
4
5         .text                   @ Zona de instrucciones
6  main:  ldr r0, =num
7         ldr r0, [r0]            @ r0 <- [num]
8         mov r1, #4
9         asr r0, r1              @ r0 <- r0 >> 4
10        ldr r2, =res
11        str r0, [r2]
12
13 stop:  wfi
```

EJERCICIOS

▶ **2.23** Copia el programa anterior en el simulador y ejecútalo, ¿qué valor se almacena en la posición de memoria «`res`»? ¿Se ha conservado el signo del número almacenado en «`num`»? Modifica el programa para comprobar su funcionamiento cuando el número que se desplaza es positivo.

▶ **2.24** Modifica el programa propuesto originalmente para que realice un desplazamiento de 3 bits. Como se puede observar, la palabra original era `0xFFFF FF41` y al desplazarla se ha obtenido la palabra `0xFFFF FFE8`. Representa ambas palabras en binario y comprueba si la palabra obtenida corresponde realmente al resultado de desplazar `0xFFFF FF41` 3 bits a la derecha conservando su signo.

▶ **2.25** La instrucción «`lsr`», desplazamiento lógico a derechas (*logic shift right*), también desplaza a la derecha un determinado número de bits el valor indicado. Sin embargo, no tiene en cuenta el signo y rellena siempre con ceros. Modifica el programa original para que utilice la instrucción «`lsr`» en lugar de la «`asr`» ¿Qué valor se obtiene ahora en «`res`»?

▶ **2.26** Modifica el código para desplazar el contenido de «`num`» 2 bits a la izquierda. ¿Qué valor se almacena ahora en «`res`»? *(Pista: La instrucción de desplazamiento a izquierdas responde al nombre en inglés de* logic shift left*)*

▶ **2.27** Es conveniente saber que desplazar n bits a la izquierda equivale a una determinada operación aritmética (siempre que no se produzca un desbordamiento). ¿A qué operación aritmética equivale? ¿Según lo anterior, a qué equivale desplazar 1 bit a la izquierda? ¿Y desplazar 2 bits?

▶ **2.28** Por otro lado, desplazar n bits a la derecha conservando el signo también equivale a una determinada operación aritmética. ¿A qué operación aritmética equivale? (Nota: si el número es positivo el desplazamiento corresponde siempre a la operación indicada; sin embargo, cuando el número es negativo, el desplazamiento no produce siempre el resultado exacto.)

2.4. Problemas del capítulo

EJERCICIOS

▶ **2.29** Desarrolla un programa en ensamblador que defina el vector de enteros de dos elementos $V = [v_0, v_1]$ en la memoria de datos y almacene la suma de sus elementos en la primera dirección de memoria no ocupada después del vector.
Para probar el programa, inicializa el vector V con $[10, 20]$.

▶ **2.30** Desarrolla un programa en ensamblador que multiplique por 5 los dos números almacenados en las dos primeras posiciones de la memoria de datos. Ambos resultados deberán almacenarse a continuación de forma consecutiva.
Para probar el programa, inicializa las dos primeras palabras de la memoria de datos con los números 18 y -1215.

▶ **2.31** Desarrolla un programa que modifique el valor de la palabra almacenada en la primera posición de la memoria de datos de tal forma que los bits 11, 7 y 3 se pongan a cero mientras que los bits restantes conserven el valor original.

Para probar el programa, inicializa la primera palabra de la memoria de datos con `0x00FF F0F0`.

▶ **2.32** Desarrolla un programa que multiplique por 32 el número almacenado en la primera posición de memoria sin utilizar operaciones aritméticas.
Para probar el programa, inicializa la primera posición de memoria con la palabra `0x0000 0001`.

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