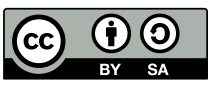
Prácticas de inntroducción a la arquitectura de computadores con QtARMSim y Arduino

Sergio Barrachina Mir
Maribel Castillo Catalán
Germán Fabregat Llueca
Juan Carlos Fernández Fernández
Germán León Navarro
José Vicente Martí Avilés
Rafael Mayo Gual
Raúl Montoliu Colás



Índice general

CAPÍTULO 1

Primeros pasos con ARM y Qt ARMSim

Índice

En este capítulo se introduce el lenguaje ensamblador de la arquitectura ARM y se describe la aplicación Qt ARMSim.

Con respecto al lenguaje ensamblador de ARM, lo primero que hay que tener en cuenta es que dicha arquitectura proporciona dos juegos de instrucciones diferenciados. Un juego de instrucciones estándar, en el que todas las instrucciones ocupan 32 bits; y un juego de instrucciones reducido, llamado Thumb, en el que la mayoría de las instrucciones ocupan 16 bits.

Uno de los motivos por el que la arquitectura ARM ha acaparado el mercado de los dispositivos empotrados ha sido justamente por pro-

porcionar el juego de instrucciones Thumb. Si se utiliza dicho juego de instrucciones es posible reducir a la mitad la memoria necesaria para las aplicaciones utilizadas en dichos dispositivos, reduciendo sustancialmente su coste de fabricación.

Cuando se programa en ensamblador de ARM, además de tener claro qué juego de instrucciones se quiere utilizar, también hay que tener en cuenta qué ensamblador se va a utilizar. Los dos ensambladores más extendidos para ARM son el ensamblador propio de ARM y el de GNU. Aunque la sintaxis de las instrucciones será la misma independientemente de qué ensamblador se utilice, la sintaxis de la parte del código fuente que describe el entorno del programa (directivas, comentarios, etc.) es diferente en ambos ensambladores.

Por tanto, para programar en ensamblador para ARM es necesario tener en cuenta en qué juego de instrucciones (estándar o Thumb) se quiere programar, y qué ensamblador se va a utilizar (ARM o GNU).

En este libro se utiliza el juego de instrucciones Thumb y la sintaxis del ensamblador de GNU, ya que son los utilizados por Qt ARMSim.

Por otro lado, Qt ARMSim es una interfaz gráfica para el simulador ARMSim[1]. Proporciona un entorno de simulación de ARM multiplataforma, fácil de usar y que ha sido diseñado para ser utilizado en cursos de introducción a la arquitectura de computadores. Qt ARMSim se distribuye bajo la licencia libre GNU GPL v3+ y puede descargarse desde la página web: «`http://lorca.act.uji.es/projects/qtarmsim`».

Este capítulo se ha organizado como sigue. Comienza con una breve descripción del ensamblador de ARM. El segundo apartado describe la aplicación Qt ARMSim. Los siguientes tres apartados proporcionan información sobre aquellas directivas e instrucciones del ensamblador de ARM que serán utilizadas con más frecuencia a lo largo del libro. En concreto, el Apartado 1.3 muestra cómo utilizar literales y constantes; el Apartado 1.4 cómo inicializar datos y reservar espacio de memoria; y el Apartado 1.5 las instrucciones de carga y almacenamiento. Finalmente, se proponen una serie de ejercicios adicionales.

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.4 «*ARM Assembly Language*» 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

[1] ARMSim es un simulador de ARM desarrollado por Germán Fabregat Llueca que se distribuye conjuntamente con Qt ARMSim.

en este libro se describe el juego de instrucciones Thumb de ARM y la sintaxis del compilador GCC.

1.1. Introducción al ensamblador Thumb de ARM

Aunque se irán mostrando más detalles sobre la sintaxis del lenguaje ensamblador Thumb de ARM conforme vaya avanzando el libro, es conveniente familiarizarse cuanto antes con algunos conceptos básicos relativos a la programación en ensamblador.

Es más, antes de comenzar con el lenguaje ensamblador propiamente dicho, es conveniente diferenciar entre «código máquina» y «lenguaje ensamblador».

El *código máquina* es el lenguaje que entiende el procesador. Una instrucción en código máquina es una secuencia de ceros y unos que el procesador es capaz de reconocer como una instrucción y, por tanto, de ejecutar.

Por ejemplo, un procesador basado en la arquitectura ARM reconocería la secuencia de bits 0001100010001011 como una instrucción máquina que forma parte de su repertorio de instrucciones y que le indica que debe sumar los registros `r1` y `r2` y almacenar el resultado de dicha suma en el registro `r3` (es decir, $[r3] \leftarrow [r1] + [r2]$, en notación RTL).

Cada instrucción máquina codifica en ceros y unos la operación que se quiere realizar, los operandos con los que se ha de realizar la operación y el operando en el que se ha de guardar el resultado. Por tanto, la secuencia de bits del ejemplo sería distinta si se quisiera realizar una operación que no fuera la suma, si los registros con los que se quisiera operar no fueran los registros `r1` y `r2`, o si el operando destino no fuera el registro `r3`.

Ahora que sabemos qué es una instrucción (en código) máquina, ¿qué es un programa en código máquina? Un programa en código máquina es simplemente una secuencia de instrucciones máquina que cuando se ejecutan realizan una determinada tarea.

Como es fácil de imaginar, desarrollar programas en código máquina, teniendo que codificar a mano cada instrucción mediante su secuencia de unos y ceros correspondiente, es una tarea sumamente ardua y propensa a errores. No es de extrañar que tan pronto como fue posible, se desarrollaran programas capaces de leer instrucciones escritas en un lenguaje más cercano al humano y de codificarlas en los unos y ceros que forman las instrucciones máquina correspondientes.

El lenguaje de programación que se limita a representar el lenguaje de la máquina, pero de una forma más cercana al lenguaje humano, re-

cibe el nombre de *lenguaje ensamblador*. Aunque este lenguaje es más asequible para nosotros que las secuencias de ceros y unos, sigue estando estrechamente ligado al código máquina. Así pues, el lenguaje ensamblador entra dentro de la categoría de *lenguajes de programación de bajo nivel*, ya que está fuertemente relacionado con el *hardware* en el que se puede utilizar.

El lenguaje ensamblador permite escribir las instrucciones máquina en forma de texto. Así pues, la instrucción máquina del ejemplo anterior, 0001100010001011, se escribiría en el lenguaje ensamblador Thumb de ARM como «`add r3, r1, r2`». Lo que obviamente es más fácil de entender que 0001100010001011, por muy poco inglés que sepamos.

Para hacernos una idea de cuán relacionado está el lenguaje ensamblador con la arquitectura a la que representa, basta con ver que incluso en una instrucción tan básica como «`add r3, r1, r2`», podríamos encontrar diferencias de sintaxis con el lenguaje ensamblador de otras arquitecturas. Por ejemplo, la misma instrucción se escribe como «`add $3, $1, $2`» en el lenguaje ensamblador de la arquitectura MIPS.

No obstante lo anterior, podemos considerar que los lenguajes ensambladores de las diferentes arquitecturas son más bien como dialectos, no son idiomas completamente diferentes. Aunque puede haber diferencias de sintaxis, las diferencias no son demasiado grandes. Por tanto, una vez que se sabe programar en el lenguaje ensamblador de una determinada arquitectura, no cuesta demasiado adaptarse al lenguaje ensamblador de otra arquitectura. Esto es debido a que las distintas arquitecturas de procesadores no son tan radicalmente distintas desde el punto de vista de su programación en ensamblador.

Como se había comentado anteriormente, uno de los hitos en el desarrollo de la computación consistió en el desarrollo de programas capaces de leer un lenguaje más cercano a nosotros y traducirlo a una secuencia de instrucciones máquina que el procesador fuera capaz de interpretar y ejecutar.

Uno de estos programas, el programa capaz de traducir *lenguaje ensamblador* a *código máquina* recibe el imaginativo nombre de *ensamblador*. Dicho programa lee un fichero de texto con el código en ensamblador y genera un fichero de instrucciones en código máquina que el procesador entiende directamente.

Es fácil darse cuenta de que una vez desarrollado un programa capaz de traducir instrucciones en ensamblador a código máquina, el siguiente paso natural haya sido el de añadir más características al lenguaje ensamblador que hicieran más fácil la programación a bajo nivel. Así pues, el lenguaje ensamblador también proporciona una serie de recursos adicionales destinados a facilitar la programación en dicho lenguaje. A continuación se muestran algunos de dichos recursos, particularizados para el caso del lenguaje ensamblador de GNU para ARM:

Comentarios Sirven para dejar por escrito qué es lo que está haciendo alguna parte del programa y para mejorar su legibilidad señalando las partes que lo forman.
Si comentar un programa cuando se utiliza un lenguaje de alto nivel se considera una buena práctica de programación, cuando se programa en lenguaje ensamblador es prácticamente imprescindible comentar el código para poder saber de un vistazo qué está haciendo cada parte del programa.
El comienzo de un comentario se indica por medio del carácter arroba («@»). Cuando el programa ensamblador encuentra el carácter «@» en el código fuente, éste ignora dicho carácter y el resto de la línea.
También es posible utilizar el carácter «#» para indicar el comienzo de un comentario, pero en este caso, el carácter «#» tan solo puede estar precedido por espacios. Así que para evitarnos problemas, es mejor utilizar «@» siempre que se quiera poner comentarios en una línea.
Por último, en el caso de querer escribir un comentario que ocupe varias líneas, es posible utilizar los delimitadores «/*» y «*/» para marcar dónde empieza y acaba, respectivamente.

Pseudo-instrucciones El lenguaje ensamblador proporciona también un conjunto de instrucciones propias que no están directamente soportadas por el juego de instrucciones máquina. Dichas instrucciones adicionales reciben el nombre de *pseudo-instrucciones* y se proporcionan para que la programación en ensamblador sea más sencilla para el programador.
Cuando el ensamblador encuentra una pseudo-instrucción, éste se encarga de sustituirla automáticamente por aquella instrucción máquina o secuencia de instrucciones máquina que realicen la función asociada a dicha pseudo-instrucción.

Etiquetas Se utilizan para posteriormente poder hacer referencia a la posición o dirección de memoria del elemento definido en la línea en la que se encuentran. Para declarar una etiqueta, ésta debe aparecer al comienzo de una línea y terminar con el carácter dos puntos («:»). No pueden empezar por un número.
Cuando el programa ensamblador encuentra la definición de una etiqueta en el código fuente, anota la dirección de memoria asociada a dicha etiqueta. Después, cuando encuentra una instrucción en la que se hace referencia a una etiqueta, sustituye la etiqueta por un valor numérico que puede ser la dirección de memoria de dicha etiqueta o un desplazamiento relativo a la dirección de memoria de la instrucción actual.

Directivas Sirven para informar al ensamblador sobre cómo debe interpretarse el código fuente. Son palabras reservadas que el ensamblador reconoce. Se identifican fácilmente ya que comienzan con un punto («.»).

En el lenguaje ensamblador, cada instrucción se escribe en una línea del código fuente, que suele tener la siguiente forma:

```
Etiqueta: operación oper1, oper2, oper3 @ Comentario
```

Conviene notar que cuando se programa en ensamblador no importa si hay uno o más espacios después de las comas en las listas de argumentos; se puede escribir indistintamente «`oper1, oper2, oper3`» o «`oper1,oper2, oper3`».

Sea el siguiente programa en ensamblador:

```
1 Bucle:  add  r0, r0, r1 @ Calcula Acumulador = Acumulador + Incremento
2         sub  r2, #1     @ Decrementa el contador
3         bne  Bucle      @ Mientras no llegue a 0, salta a Bucle
```

Las líneas del programa anterior están formadas por una instrucción cada una (que indica el nombre de la operación a realizar y sus argumentos) y un comentario (que comienza con el carácter «@»).

Además, la primera de las líneas declara la etiqueta «`Bucle`», que podría ser utilizada por otras instrucciones para referirse a dicha línea. En el ejemplo, la etiqueta «`Bucle`» es utilizada por la instrucción de salto condicional que hay en la tercera línea. Cuando se ensamble dicho programa, el ensamblador traducirá la instrucción «**`bne`** `Bucle`» por la instrucción máquina «**`bne`** `pc, #-8`». Es decir, sustituirá, sin entrar en más detalles, la etiqueta «`Bucle`» por el número «`-8`».

En el siguiente ejemplo se muestra un fragmento de código que calcula la suma de los cubos de los números del 1 al 10.

```
2 main:   mov r0, #0      @ Total a 0
3         mov r1, #10     @ Inicializa n a 10
4 loop:   mov r2, r1      @ Copia n a r2
5         mul r2, r1      @ Almacena n al cuadrado en r2
6         mul r2, r1      @ Almacena n al cubo en r2
7         add r0, r0, r2  @ Suma [r0] y el cubo de n
8         sub r1, r1, #1  @ Decrementa n en 1
9         bne loop        @ Salta a «loop» si n != 0
```

El anterior programa en ensamblador es sintácticamente correcto e implementa el algoritmo apropiado para calcular la suma de los cubos de los números del 1 al 10. Sin embargo, todavía no es un programa

que podamos ensamblar y ejecutar. Por ejemplo, aún no se ha indicado dónde comienza el código.

Así pues, un programa en ensamblador está compuesto en realidad por dos tipos de sentencias: *instrucciones ejecutables*, que son ejecutadas por el computador, y *directivas*, que informan al programa ensamblador sobre el entorno del programa. Las directivas, que ya habíamos introducido previamente entre los recursos adicionales del lenguaje ensamblador, se utilizan para: I) informar al programa ensamblador de dónde se debe colocar el código en memoria, II) reservar espacio de almacenamiento para variables, y III) fijar los datos iniciales que pueda necesitar el programa.

Para que el programa anterior pudiera ser ensamblado y ejecutado en el simulador Qt ARMSim, sería necesario añadir la primera y la penúltima de las líneas mostradas a continuación. (La última línea es opcional.)

introsim-cubos.s

```
1         .text
2 main:   mov r0, #0      @ Total a 0
3         mov r1, #10     @ Inicializa n a 10
4 loop:   mov r2, r1      @ Copia n a r2
5         mul r2, r1      @ Almacena n al cuadrado en r2
6         mul r2, r1      @ Almacena n al cubo en r2
7         add r0, r0, r2  @ Suma [r0] y el cubo de n
8         sub r1, r1, #1  @ Decrementa n en 1
9         bne loop        @ Salta a «loop» si n != 0
10 stop:  wfi
11        .end
```

La primera línea del código anterior presenta la directiva «**.text**». Dicha directiva indica al ensamblador que lo que viene a continuación es el programa en ensamblador y que debe colocar las instrucciones que lo forman en la zona de memoria asignada al código ejecutable. En el caso del simulador Qt ARMSim esto implica que el código que venga a continuación de la directiva «**.text**» se almacene en la memoria ROM, a partir de la dirección `0x0000 1000`.

La penúltima de las líneas del código anterior contiene la instrucción «**wfi**». Dicha instrucción se usa para indicar al simulador Qt ARMSim que debe concluir la ejecución del programa en curso. Su uso es específico del simulador Qt ARMSim. Cuando se programe para otro entorno, habrá que averiguar cuál es la forma adecuada de indicar el final de la ejecución en ese entorno.

La última de las líneas del código anterior presenta la directiva «**.end**», que sirve para señalar el final del módulo que se quiere ensamblar. Por regla general no es necesario utilizarla. Tan solo tiene sentido

hacerlo en el caso de que se quiera escribir algo a continuación de dicha línea y que ese texto sea ignorado por el ensamblador.

1.2. Introducción al simulador Qt ARMSim

Como se ha comentado en la introducción de este capítulo, Qt ARMSim es una interfaz gráfica para el simulador ARMSim, que proporciona un entorno de simulación basado en ARM. Qt ARMSim y ARMSim han sido diseñados para ser utilizados en cursos de introducción a la arquitectura de computadores y pueden descargarse desde la web: «`http://lorca.act.uji.es/projects/qtarmsim`».

1.2.1. Ejecución, descripción y configuración

Para ejecutar Qt ARMSim, basta con pulsar sobre el icono correspondiente o lanzar el comando «`qtarmsim`». La Figura 1.1 muestra la ventana principal de Qt ARMSim cuando acaba de iniciarse.

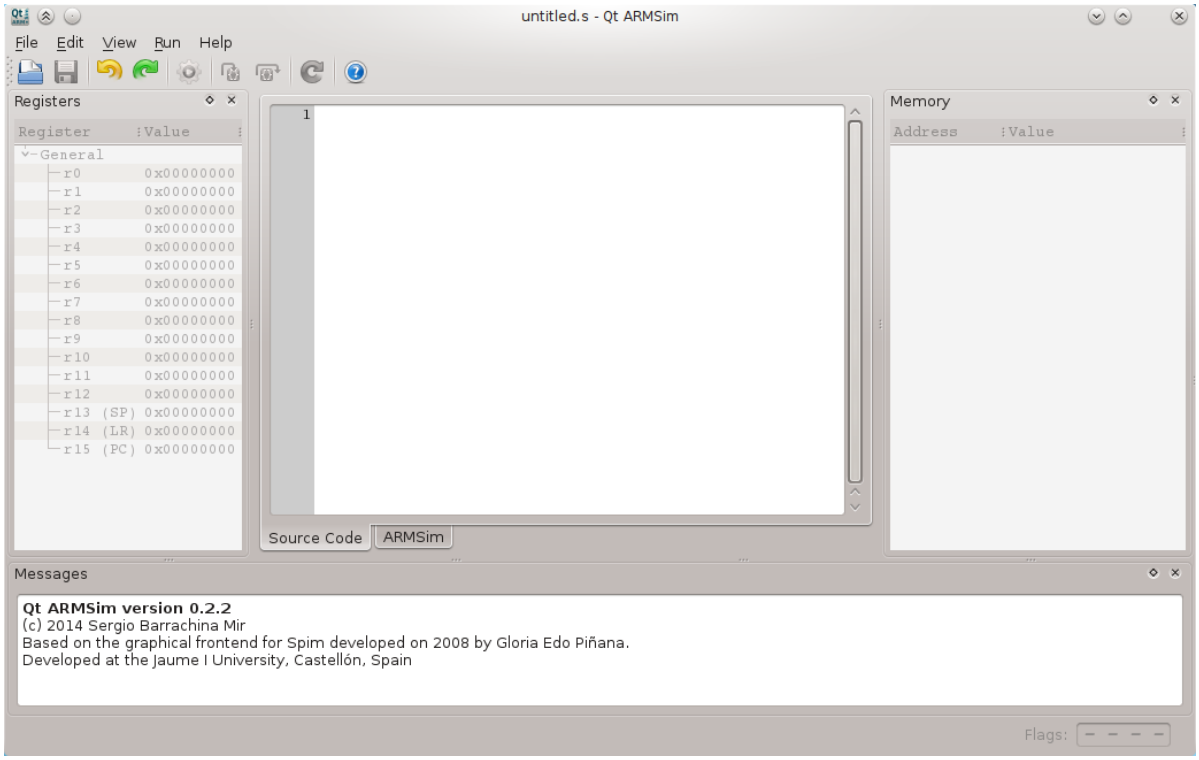


Figura 1.1: Ventana principal de Qt ARMSim

La parte central de la ventana principal que se puede ver en la Figura 1.1 corresponde al editor de código fuente en ensamblador. Alrededor

de dicha parte central se distribuyen una serie de paneles. A la izquierda del editor se encuentra el panel de registros; a su derecha, el panel de memoria; y debajo, el panel de mensajes. Los paneles de registros y memoria inicialmente están desactivados y en breve volveremos sobre ellos. En el panel de mensajes se irán mostrando mensajes relacionados con lo que se vaya haciendo: si el código se ha ensamblado correctamente, si ha habido errores de sintaxis, qué línea se acaba de ejecutar, etc.

Si se acaba de instalar Qt ARMSim, es probable que sea necesario modificar sus preferencias para indicar cómo llamar al simulador ARMSim y para indicar dónde está instalado el compilador cruzado de GCC para ARM. Para mostrar el cuadro de diálogo de preferencias de Qt ARMSim se debe seleccionar la entrada «`Preferences...`» dentro del menú «`Edit`».

¿Mostrar el cuadro de diálogo de preferencias?

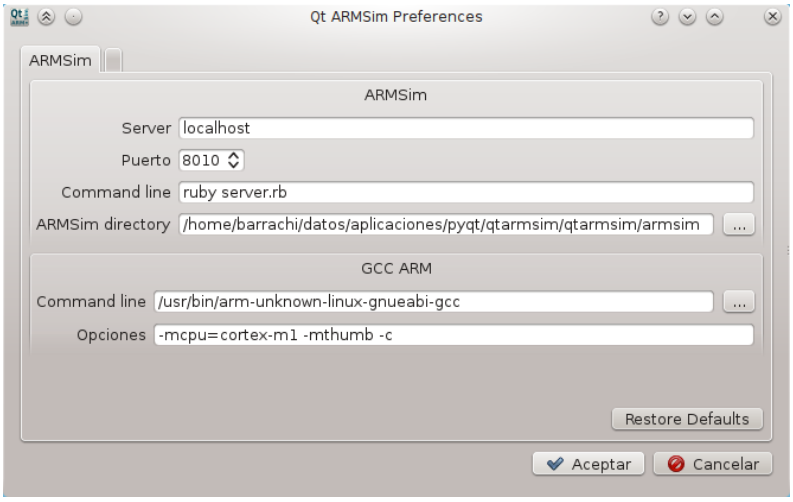


Figura 1.2: Cuadro de diálogo de preferencias de Qt ARMSim

La Figura 1.2 muestra el cuadro de diálogo de preferencias. En dicho cuadro de diálogo se pueden observar dos paneles. El panel superior corresponde al simulador ARMSim y permite configurar el servidor y el puerto en el que debe escuchar el simulador; la línea de comandos para ejecutar el simulador; y el directorio de trabajo del simulador. Generalmente este panel estará bien configurado por defecto y no conviene cambiar nada de dicha configuración.

El panel inferior corresponde al compilador de GCC para ARM. En dicho panel se debe indicar la ruta al ejecutable del compilador de GCC para ARM y las opciones de compilación que se deben pasar al compilador.

Normalmente tan solo será necesario configurar la ruta al ejecutable del compilador de GCC para ARM, y eso en el caso de que Qt ARMSim

no haya podido encontrar el ejecutable en una de las rutas por defecto del sistema.

1.2.2. Modo de edición

Cuando se ejecuta Qt ARMSim se inicia en el modo de edición. En este modo, la parte central de la ventana es un editor de código fuente en ensamblador, que permite escribir el programa en ensamblador que se quiere simular. La Figura 1.3 muestra la ventana de Qt ARMSim en la que se ha introducido el programa en ensamblador visto en el Apartado 1.1 «Introducción al ensamblador Thumb de ARM».

Hay que tener en cuenta que antes de ensamblar y simular el código fuente que se esté editando, primero habrá que guardarlo (menú «`File > Save`»; «`CTRL+s`»), ya que lo que se ensambla es el fichero almacenado en disco. Si se hacen cambios en el editor y no se guardan dichos cambios, la simulación no los tendrá en cuenta.

Como era de esperar, también es posible abrir un fichero en ensamblador guardado previamente. Para ello se puede seleccionar la opción del menú «`File > Open...`» o teclear la combinación de teclas «`CTRL+o`».

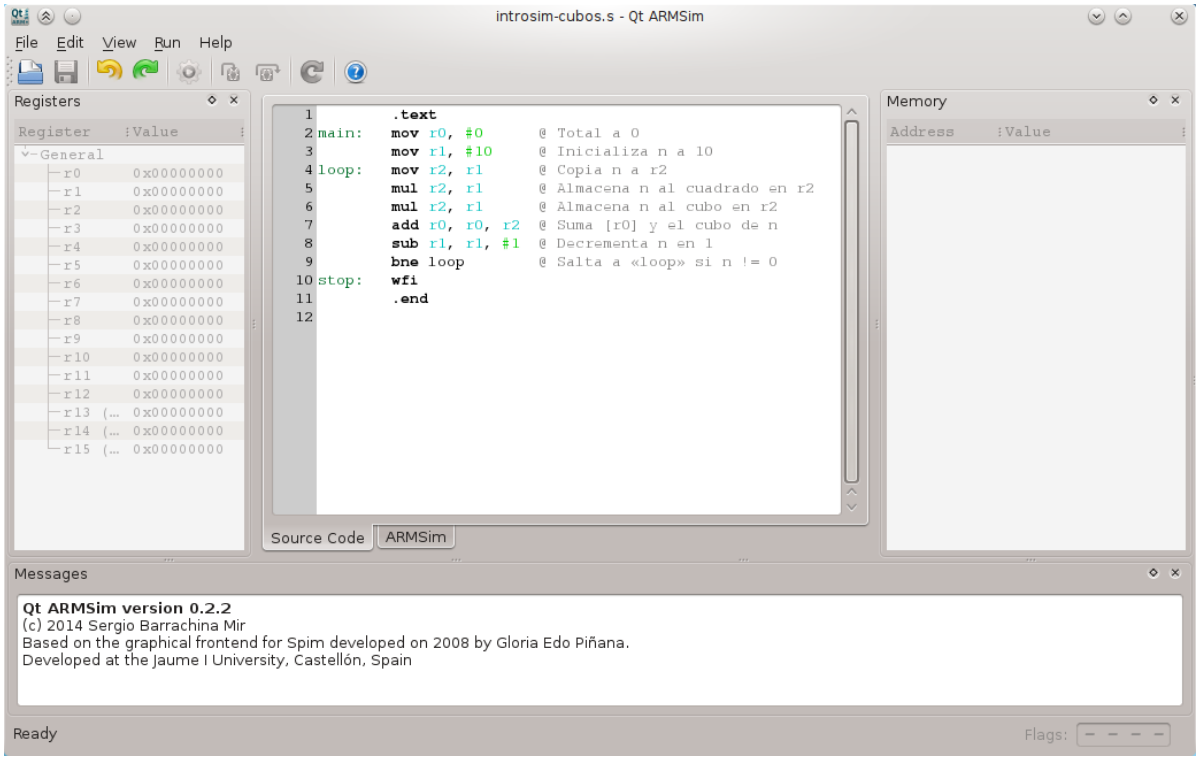


Figura 1.3: Qt ARMSim mostrando el programa «introsim-cubos.s»

1.2.3. Modo de simulación

Una vez se ha escrito un programa en ensamblador, el siguiente paso es ensamblar dicho código y simular su ejecución.

¿Cambiar al modo de simulación?

Para ensamblar el código y pasar al modo de simulación, basta con pulsar sobre la pestaña «`ARMSim`» que se encuentra debajo de la sección central de la ventana principal.

Cuando se pasa al modo de simulación, la interfaz gráfica se conecta con el simulador ARMSim, quien se encarga de realizar las siguientes acciones: I) llamar al ensamblador de GNU para ensamblar el código fuente; II) actualizar el contenido de la memoria ROM con las instrucciones máquina generadas por el ensamblador; III) inicializar, si es el caso, el contenido de la memoria RAM con los datos indicados en el código fuente; y, por último, IV) inicializar los registros del computador simulado.

Si se produjera algún error al intentar pasar al modo de simulación, se mostrará un cuadro de diálogo informando del error, se volverá automáticamente al modo de edición y en el panel de mensajes se mostrarán las causas del error. Es de esperar que la mayor parte de las veces el error sea debido a un error de sintaxis en el código fuente.

La Figura 1.4 muestra la apariencia de Qt ARMSim cuando está en el modo de simulación.

Si se compara la apariencia de Qt ARMSim cuando está en el modo de edición (Figura 1.3) con la de cuando está en el modo de simulación (Figura 1.4), se puede observar que al cambiar al modo de simulación se han habilitado los paneles de registros y memoria que estaban desactivados en el modo de edición. De hecho, si se vuelve al modo de edición pulsando sobre la pestaña «`Source Code`», se podrá ver que dichos paneles se desactivan automáticamente. De igual forma, si se vuelve al modo de simulación, aquéllos volverán a activarse.

¿Volver al modo de edición?

De vuelta en el modo de simulación, el contenido de la memoria del computador simulado se muestra en el panel de memoria. En la Figura 1.4 se puede ver que el computador simulado dispone de dos bloques de memoria: un bloque de memoria ROM que comienza en la dirección `0x0000 1000` y un bloque de memoria RAM que comienza en la dirección `0x2007 0000`. También se puede ver cómo las celdas de la memoria ROM contienen algunos valores distintos de cero (que corresponden a las instrucciones máquina del programa ensamblado) y las celdas de la memoria RAM están todas a cero.

Por otro lado, el contenido de los registros del «`r0`» al «`r15`» se muestra en el panel de registros. El registro «`r15`» merece una mención especial, ya que se trata del contador de programa (PC, por las siglas en inglés de *Program Counter*). Como se puede ver en la Figura 1.4, el «`PC`» está apuntando en este caso a la dirección de memoria `0x0000 1000`.

El PC apunta a la dirección de memoria de la instrucción que va a ser ejecutada.

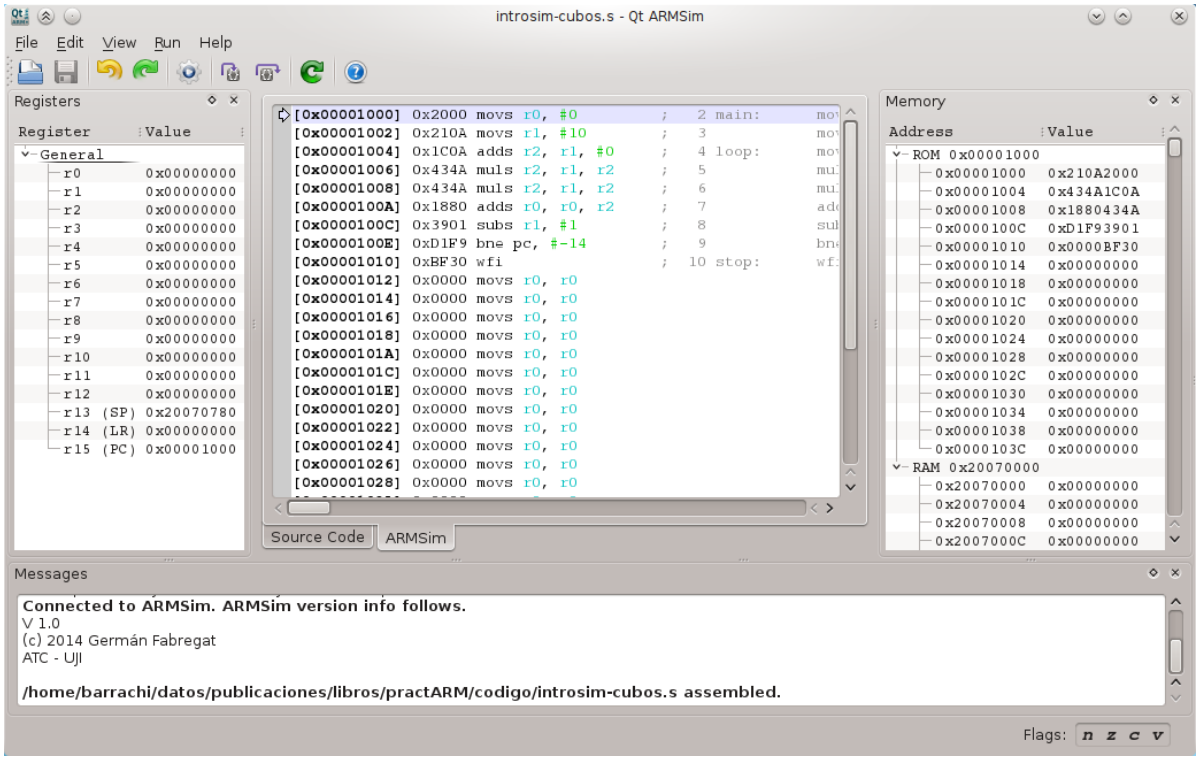


Figura 1.4: Qt ARMSim en el modo de simulación

Como se ha comentado en el párrafo anterior, la dirección `0x00001000` es justamente la dirección de memoria en la que comienza el bloque de memoria ROM del computador simulado, donde se encuentra el programa en código máquina. Así pues, el «`PC`» está apuntando en este caso a la primera dirección de memoria del bloque de la memoria ROM, por lo que la primera instrucción en ejecutarse será la primera del programa.

Puesto que los paneles del simulador son empotrables, es posible cerrarlos de manera individual, reubicarlos en una posición distinta, o desacoplarlos y mostrarlos como ventanas flotantes. La Figura 1.5 muestra la ventana principal del simulador tras cerrar los paneles en los que se muestran los registros y la memoria.

¿Cómo restaurar la disposición por defecto?

Conviene saber que es posible restaurar la disposición por defecto del simulador seleccionando la entrada «`Restore Default Layout`» del menú «`View`» (o pulsando la tecla «`F3`»). También se pueden volver a mostrar los paneles que han sido cerrados previamente, sin necesidad de restaurar la disposición por defecto. Para ello se debe marcar en el menú «`View`» la opción correspondiente al panel que se quiere mostrar.

En el modo de simulación, cada línea de la ventana central muestra la información correspondiente a una instrucción máquina. Esta infor-

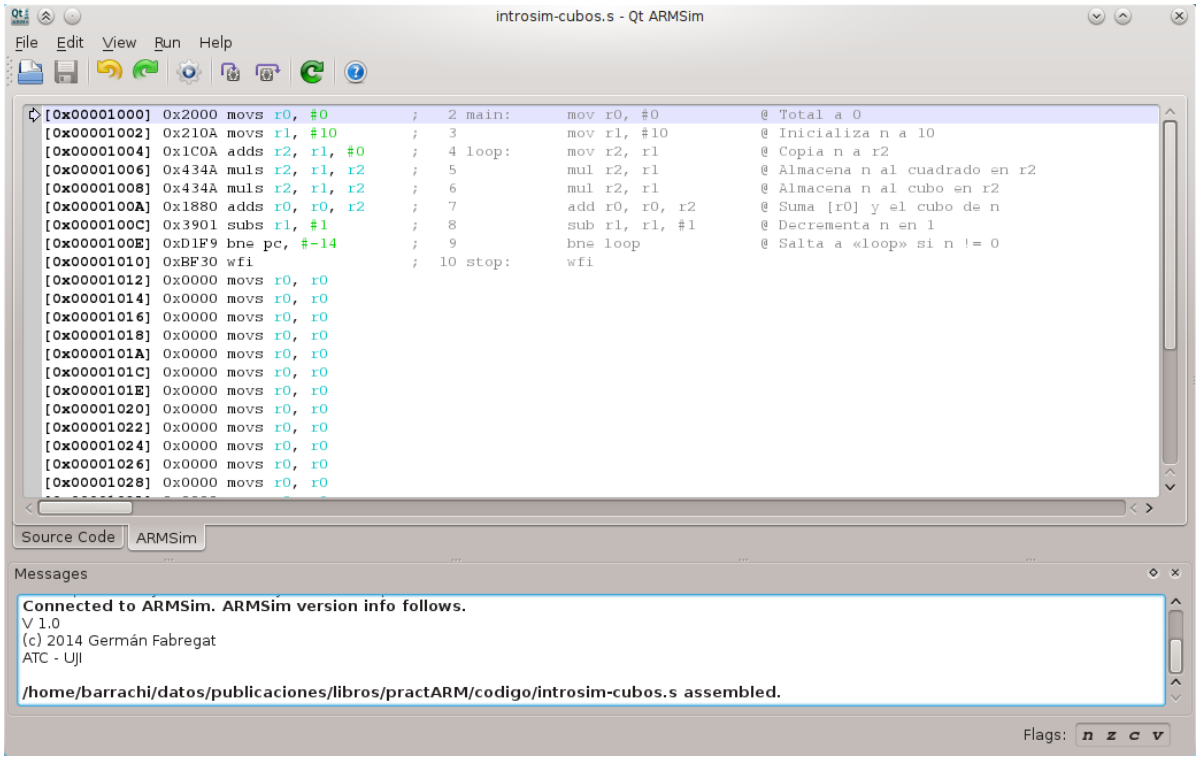


Figura 1.5: Qt ARMSim sin paneles de registros y memoria

mación se obtiene a partir del contenido de la memoria ROM, por medio de un proceso que se denomina *desensamblado*. La información mostrada para cada instrucción máquina es la siguiente:

1º La dirección de memoria en la que está almacenada la instrucción máquina.

2º La instrucción máquina expresada en hexadecimal.

3º La instrucción máquina expresada en ensamblador.

4º La línea original en ensamblador que ha dado lugar a la instrucción máquina.

Tomando como ejemplo la primera línea de la ventana de desensamblado de la Figura 1.5, su información se interpretaría de la siguiente forma:

- La instrucción máquina está almacenada en la dirección de memoria `0x0000 1000`.
- La instrucción máquina expresada en hexadecimal es `0x2000`.

- La instrucción máquina expresada en ensamblador es «`movs r0, #0`».
- La instrucción se ha generado a partir de la línea número 2 del código fuente original cuyo contenido es:

  «`main: mov r0, #0 @ Total a 0`»

Ejecución del programa completo

Una vez ensamblado el código fuente y cargado el código máquina en el simulador, la opción más sencilla de simulación es la de ejecutar el programa completo. Para ejecutar todo el programa, se puede seleccionar la entrada del menú «`Run > Run`» o pulsar la combinación de teclas «`CTRL+F11`».

La Figura 1.6 muestra la ventana de Qt ARMSim después de ejecutar el código máquina generado al ensamblar el fichero «introsim-cubos.s». En dicha figura se puede ver que los registros `r0`, `r1`, `r2` y `r15` tienen ahora fondo azul y están en negrita. Eso es debido a que el simulador resalta aquellos registros y posiciones de memoria que son modificados durante la ejecución del código máquina. En este caso, el código máquina modifica los registros `r0`, `r1` y `r2` durante el cálculo de la suma de los cubos de los números del 10 al 1. El registro `r15`, el contador de programa, también se ha modificado; ahora apunta a la última línea del programa. Por último, y debido a que este programa no escribe en memoria, no se ha resaltado ninguna de las posiciones de memoria.

Una vez realizada una ejecución completa, lo que generalmente se hace es comprobar si el resultado obtenido es realmente el esperado. En este caso, el resultado del programa anterior se almacena en el registro `r0`. Como se puede ver en el panel de registros, el contenido del registro `r0` es `0x0000 0BD1`. Para comprobar si dicho número corresponde realmente a la suma de los cubos de los números del 10 al 1, se puede ejecutar, por ejemplo, el siguiente programa en Python3.

```
1 suma = 0
2 for num in range(1, 11):
3     cubo = num * num * num
4     suma = suma + cubo
5
6 print("El resultado es: {}".format(suma))
7 print("El resultado en hexadecimal es: 0x{:08X}".format(suma))
```

Cuando se ejecuta el programa anterior con Python3, se obtiene el siguiente resultado:

```
$ python3 codigo/introsim-cubos.py
El resultado es: 3025
El resultado en hexadecimal es: 0x00000BD1
```

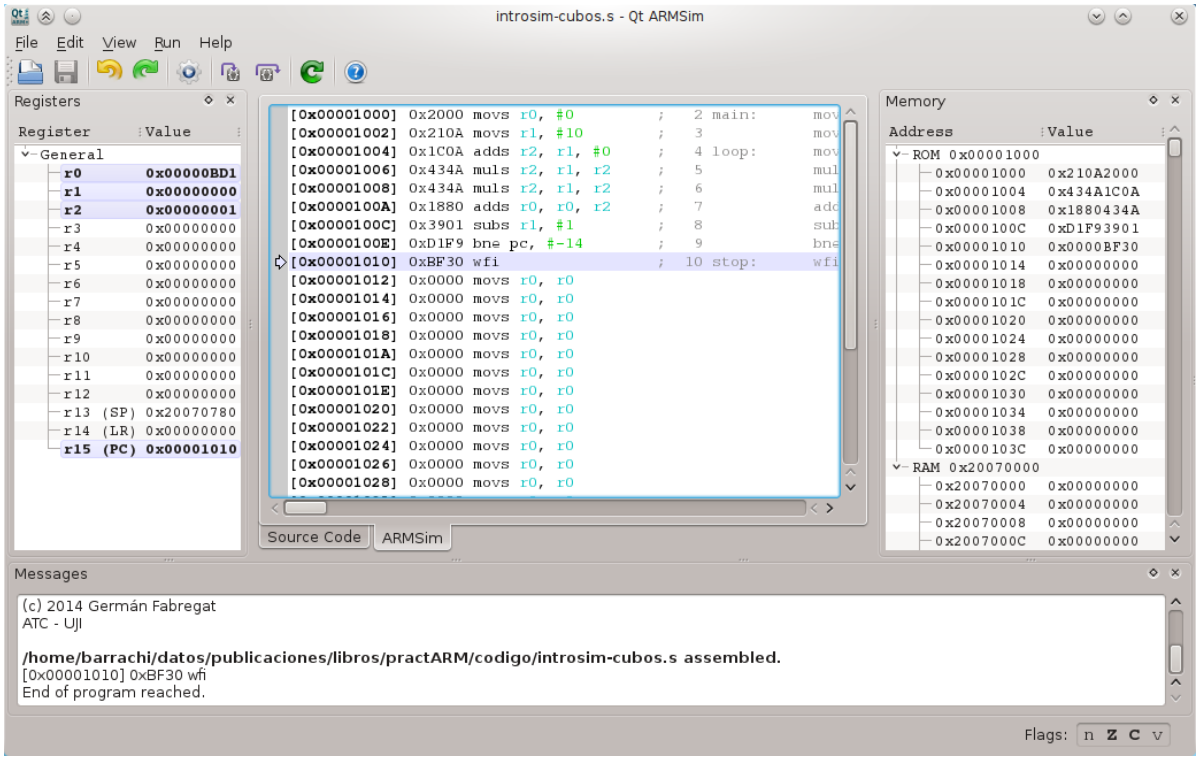


Figura 1.6: Qt ARMSim después de ejecutar el código máquina

El resultado en hexadecimal mostrado por el programa en Python coincide efectivamente con el obtenido en el registro `r0` cuando se ha ejecutado el código máquina generado a partir de «introsim-cubos.s».

Si además de saber qué es lo que hace el programa «introsim-cubos.s», también se tiene claro cómo lo hace, será posible ir un paso más allá y comprobar si los registros `r1` y `r2` tienen los valores esperados tras la ejecución del programa.

El registro `r1` se inicializa con el número 10 y en cada iteración del bucle se va decrementando de 1 en 1. El bucle dejará de repetirse cuando el valor del registro `r1` pasa a valer 0. Por tanto, cuando finalice la ejecución del programa, dicho registro debería valer 0, como así es, tal y como se puede comprobar en la Figura 1.6.

Por otro lado, el registro `r2` se utiliza para almacenar el cubo de cada uno de los números del 10 al 1. Cuando finalice el programa, dicho registro debería tener el cubo del último número evaluado, esto es 1^3, y efectivamente, así es.

Recargar la simulación

Cuando se le pide al simulador que ejecute el programa, en realidad no se le está diciendo que ejecute todo el programa de principio a fin. Se le está diciendo que ejecute el programa a partir de la dirección indicada por el registro PC (`r15`) hasta que encuentre una instrucción de paro («`wfi`»), un error de ejecución, o un punto de ruptura (más adelante se comentará qué son los puntos de ruptura).

Lo más habitual será que la ejecución se detenga por haber alcanzado una instrucción de paro («`wfi`»). Si éste es el caso, el PC se quedará apuntando a dicha instrucción. Por lo tanto, cuando se vuelva a pulsar el botón de ejecución, no sucederá nada, ya que el PC está apuntando a una instrucción de paro, por lo que cuando el simulador ejecute dicha instrucción, se detendrá, y el PC seguirá apuntando a dicha instrucción.

Así que para poder ejecutar de nuevo el código, o para iniciar una ejecución paso a paso, como se verá en el siguiente apartado, es necesario recargar la simulación. Para recargar la simulación se debe seleccionar la entrada de menú «`Run > Refresh`», o pulsar la tecla «`F4`».

Ejecución paso a paso

Aunque la ejecución completa de un programa pueda servir para comprobar si el programa hace lo que se espera de él, no permite ver con detalle cómo se ejecuta el programa. Tan solo se puede observar el estado inicial del computador simulado y el estado al que se llega cuando se termina la ejecución del programa.

Para poder ver qué es lo que ocurre al ejecutar cada instrucción, el simulador proporciona la opción de ejecutar paso a paso. Para ejecutar el programa paso a paso, se puede seleccionar la entrada del menú «`Run > Step Into`» o la tecla «`F5`».

La ejecución paso a paso suele utilizarse para ver por qué un determinado programa o una parte del programa no está haciendo lo que se espera de él. O para evaluar cómo afecta la modificación del contenido de determinados registros o posiciones de memoria al resultado del programa.

Veamos cómo podría hacerse ésto último. La Figura 1.7 muestra el estado del simulador tras ejecutar dos instrucciones (tras pulsar la tecla «`F5`» 2 veces). Como se puede ver en dicha figura, se acaba de ejecutar la instrucción «`movs r1, #10`» y la siguiente instrucción que va a ejecutarse es «`adds r2, r1, #0`». El registro `r1` tiene ahora el número 10 (`0x0000000A` en hexadecimal), por lo que al ejecutarse el resto del programa se calculará la suma de los cubos de los números del 10 al 1, como ya se ha comprobado anteriormente.

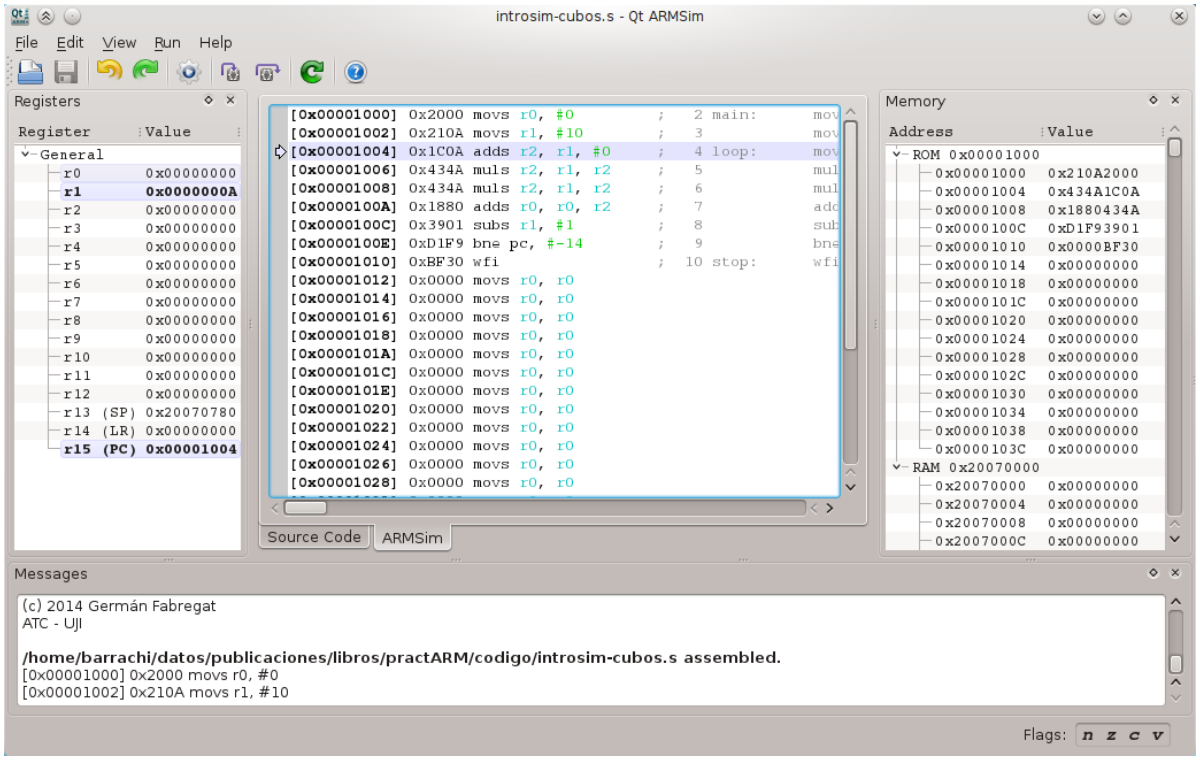


Figura 1.7: Qt ARMSim después de ejecutar dos instrucciones

Si en este momento modificáramos dicho registro para que tuviera el número 3, cuando se ejecute el resto del programa se debería calcular la suma de los cubos del 3 al 1 (en lugar de la suma de los cubos del 10 al 1).

Para modificar el contenido del registro `r1` se debe hacer doble clic sobre la celda en la que está su contenido actual (ver Figura 1.8), teclear el nuevo número y pulsar la tecla «`Retorno`». El nuevo valor numérico[2] puede introducirse en decimal, en hexadecimal (si se precede de «0x», p.e., «0x3»), o en binario (si se precede de «0b», p.e., «0b11»).

Una vez modificado el contenido del registro `r1` para que contenga el valor 3, se puede ejecutar el resto del código de golpe (menú «`Run > Run`»), no hace falta ir paso a paso. Cuando finalice la ejecución, el registro `r0` deberá tener el valor `0x0000 0024`, que en decimal es el número 36, que es $3^3 + 2^3 + 1^3$.

[2]También es posible introducir cadenas de como mucho 4 caracteres. En este caso deberán estar entre comillas simples o dobles, p.e., `"Hola"`. Al convertir los caracteres de la cadena introducida a números, se utiliza la codificación UTF-8 y para ordenar los bytes resultantes dentro del registro se sigue el convenio *Little-Endian*. Si no has entendido nada de lo anterior, no te preocupes... por ahora.

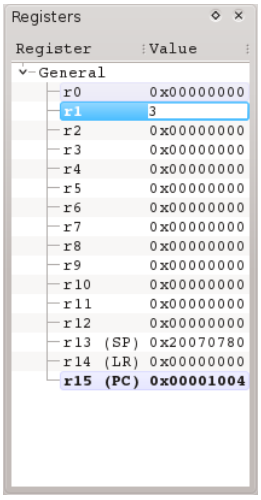


Figura 1.8: Edición del registro `r1`

En realidad, existen dos modalidades de ejecución paso a paso, la primera de ellas, la comentada hasta ahora, menú «`Run > Step Into`», ejecuta siempre una única instrucción, pasando el PC a apuntar a la siguiente instrucción.

La segunda opción tiene en cuenta que los programas suelen estructurarse por medio de rutinas (también llamadas procedimientos, funciones o subrutinas). Una rutina es un fragmento de código que puede ser llamado desde varias partes del programa y que cuando acaba, devuelve el control a la instrucción siguiente a la que le llamó.

Si el código en ensamblador incluye llamadas a rutinas, al utilizar el modo de ejecución paso a paso visto hasta ahora sobre una instrucción de llamada a una rutina, la siguiente instrucción que se ejecutará será la primera instrucción de dicha rutina.

Sin embargo, en ocasiones no interesa tener que ejecutar paso a paso todo el contenido de una determinada rutina, puede ser preferible ejecutar la rutina entera como si de una única instrucción se tratara, y que una vez ejecutada la rutina, el PC pase a apuntar directamente a la siguiente instrucción a la de la llamada a la rutina. De esta forma, sería fácil para el programador ver y comparar el estado del computador simulado antes de llamar a la rutina y justo después de volver de ella.

Para poder hacer lo anterior, se proporciona una opción de ejecución paso a paso llamada «por encima» (*step over*). Para ejecutar paso a paso por encima, se debe seleccionar la entrada del menú «`Run > Step Over`» o pulsar la tecla «`F6`».

La ejecución paso a paso entrando (*step into*) y la ejecución paso a paso por encima (*step over*) se comportarán de forma diferente única-

mente cuando la instrucción que se vaya a ejecutar sea una instrucción de llamada a una rutina. Ante cualquier otra instrucción, las dos ejecuciones paso a paso harán lo mismo.

Puntos de ruptura

La ejecución paso a paso permite ver con detenimiento qué es lo que está ocurriendo en una determinada parte del código. Sin embargo, puede que para llegar a la zona del código que se quiere inspeccionar con detenimiento haya que ejecutar muchas instrucciones. Por ejemplo, podríamos estar interesados en una parte del código al que se llega después de completar un bucle con cientos de iteraciones. No tendría sentido tener que ir paso a paso hasta conseguir salir del bucle y llegar a la parte del código que en realidad queremos ver con más detenimiento.

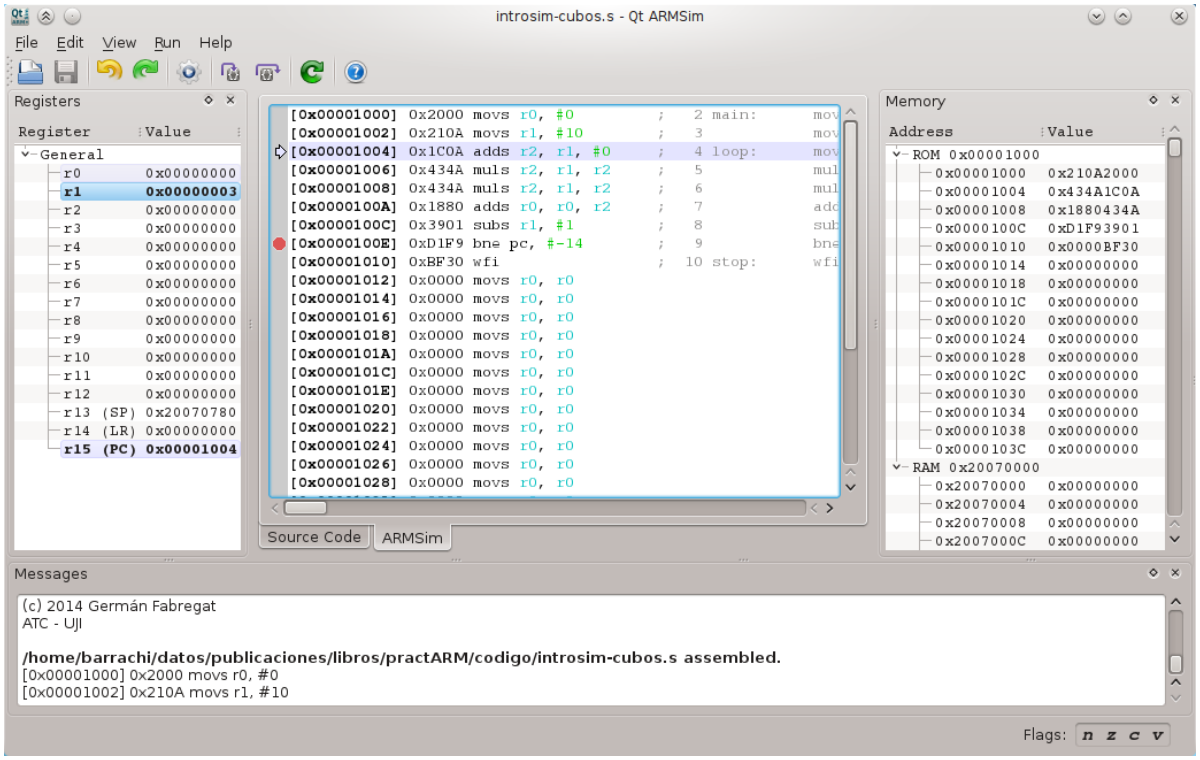


Figura 1.9: Punto de ruptura en la dirección `0x0000100E`

Por tanto, es necesario disponer de una forma de indicarle al simulador que ejecute las partes del código que no nos interesa ver con detenimiento y que solo se detenga cuando llegue a aquella instrucción a partir de la cual queremos realizar una ejecución paso a paso (o en la que queremos poder observar el estado del simulador).

Un punto de ruptura (*breakpoint* en inglés) sirve justamente para eso, para indicarle al simulador que tiene que parar la ejecución cuando se alcance la instrucción en la que se haya definido un punto de ruptura.

Antes de ver cómo definir y eliminar puntos de ruptura, conviene tener en cuenta que los puntos de ruptura solo se muestran y pueden editarse cuando se está en el modo de simulación.

Para definir un punto de ruptura, se debe hacer clic sobre el margen de la ventana de desensamblado, en la línea en la que se quiere definir. Al hacerlo, aparecerá un círculo rojo en el margen, que indica que en esa línea se ha definido un punto de ruptura.

Para desmarcar un punto de ruptura ya definido, se debe proceder de la misma forma, se debe hacer clic sobre la marca del punto de ruptura.

La Figura 1.9 muestra la ventana de Qt ARMSim en la que se han ejecutado 2 instrucciones paso a paso y se ha añadido un punto de ruptura en la instrucción máquina que se encuentra en la dirección de memoria `0x0000100E`. Por su parte, la Figura 1.10 muestra el estado al que se llega después de pulsar la entrada de menú «`Run > Run`». Como se puede ver, el simulador se ha detenido justo en la instrucción marcada con el punto de ruptura (sin ejecutarla).

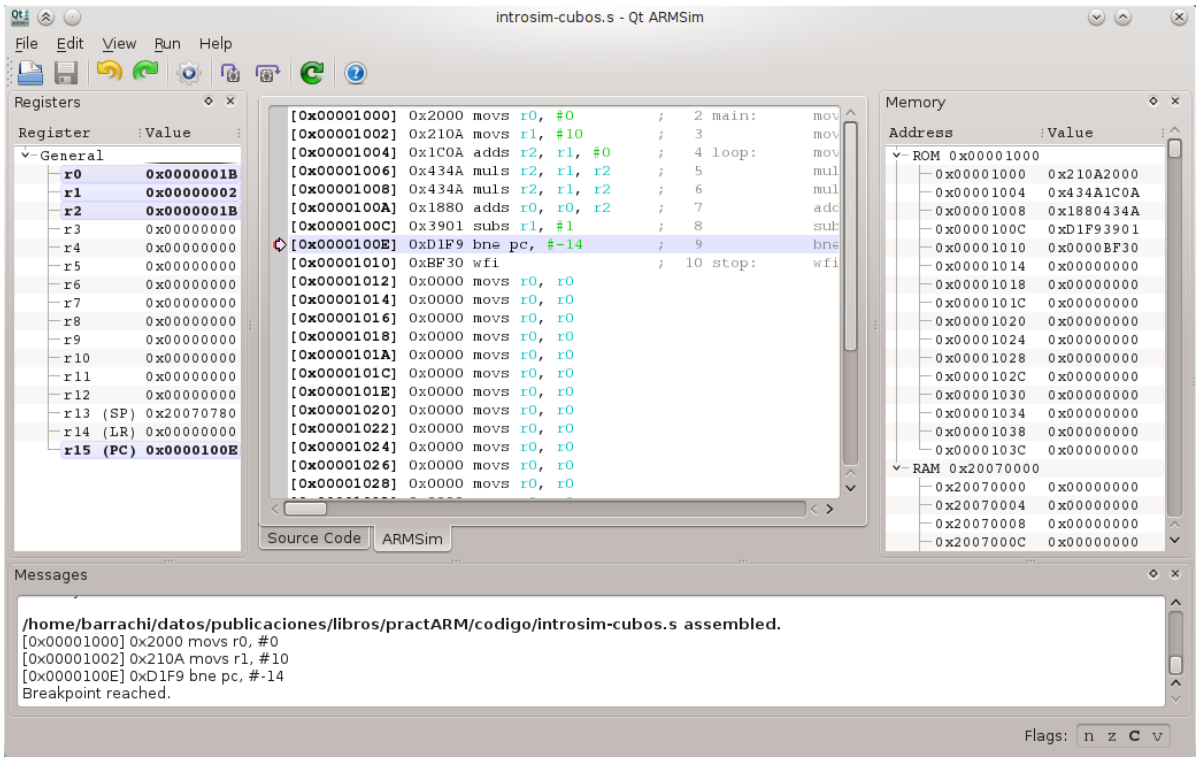


Figura 1.10: Programa detenido al llegar a un punto de ruptura

EJERCICIOS

▶ **1.1** Dado el siguiente ejemplo de programa ensamblador, identifica y señala las etiquetas, directivas y comentarios que aparecen en él.

introsim-cubos.s

```
1         .text
2  main:  mov r0, #0      @ Total a 0
3         mov r1, #10     @ Inicializa n a 10
4  loop:  mov r2, r1      @ Copia n a r2
5         mul r2, r1      @ Almacena n al cuadrado en r2
6         mul r2, r1      @ Almacena n al cubo en r2
7         add r0, r0, r2  @ Suma [r0] y el cubo de n
8         sub r1, r1, #1  @ Decrementa n en 1
9         bne loop        @ Salta a «loop» si n != 0
10 stop:  wfi
11        .end
```

▶ **1.2** Abre el simulador, copia el programa anterior, pasa al modo de simulación y responde a las siguientes preguntas.

1. Localiza la instrucción «`add r0, r0, r2`», ¿en qué dirección de memoria se ha almacenado?
2. ¿A qué instrucción en código máquina (en letra) ha dado lugar la anterior instrucción en ensamblador?
3. ¿Cómo se codifica en hexadecimal dicha instrucción máquina?
4. Localiza en el panel de memoria dicho número.
5. Localiza la instrucción «`sub r1, r1, #1`», ¿en qué dirección de memoria se ha almacenado?
6. ¿A qué instrucción en código máquina (en letra) ha dado lugar la anterior instrucción en ensamblador?
7. ¿Cómo se codifica en hexadecimal dicha instrucción máquina?
8. Localiza en el panel de memoria dicho número.

▶ **1.3** Ejecuta el programa anterior, ¿qué valores toman los siguientes registros?

- `r0`
- `r1`
- `r2`
- `r15`

1.3. Literales y constantes en el ensamblador de ARM

En los apartados anteriores se han visto algunos ejemplos en los que se incluían literales en el código. Por ejemplo, el «`#1`» del final de la instrucción «`sub r1, r1, #1`» indica que queremos restar 1 al contenido del registro `r1`. Ese 1 es un valor literal.

Un literal puede ser un número (expresado en decimal, binario, octal o hexadecimal), un carácter o una cadena de caracteres. La forma de identificar un literal en ensamblador es precediéndolo por el carácter «`#`».

Puesto que el tipo de valor literal que se utiliza más frecuentemente es el de un número en decimal, la forma de indicar un número en decimal es la más sencilla de todas. Simplemente se antepone el carácter «`#`» al número en decimal tal cual (la única precaución que hay que tener al escribirlo es que no comience por 0).

Por ejemplo, como ya se ha visto, la instrucción «`sub r1, r1, #1`» resta 1 (especificado de forma literal) al contenido de `r1` y almacena el resultado de la resta en `r1`. Si en lugar de dicha instrucción, hubiéramos necesitado una instrucción que restara 12 al contenido de `r1`, habríamos escrito «`sub r1, r1, #12`».

En ocasiones es más conveniente especificar un número en hexadecimal, en octal o en binario. Para hacerlo, al igual que antes se debe empezar por el carácter «`#`»; a continuación, uno de los siguientes prefijos: «`0x`» para hexadecimal, «`0`» para octal y «`0b`» para binario[3]; y, por último, el número en la base seleccionada.

En el siguiente código se muestran 4 instrucciones que inicializan los registros `r0` al `r3` con 4 valores numéricos literales en decimal, hexadecimal, octal y binario, respectivamente.

introsim-numeros.s

```
1         .text
2 main:   mov r0, #30             @ 30 en decimal
3         mov r1, #0x1E           @ 30 en hexadecimal
4         mov r2, #036            @ 30 en octal
5         mov r3, #0b00011110     @ 30 en binario
6 stop:   wfi
```

EJERCICIOS

Copia el programa anterior en Qt ARMSim, cambia al modo de simulación y contesta las siguientes preguntas.

▶ **1.4** Cuando el simulador desensambla el código, ¿qué ha pasado con

[3] Si has reparado en ello, los prefijos para el hexadecimal, el octal y el binario comienzan por cero. Pero además, el prefijo del octal es simplemente un cero; por eso cuando el número está en decimal no puede empezar por cero.

kcalc

En GNU/Linux se puede utilizar la calculadora «`kcalc`» para convertir un número entre los distintos sistemas de numeración.

Para poner la calculadora en el modo de conversión entre sistemas de numeración, se debe seleccionar la entrada de menú «`Preferencias > Modo sistema de numeración`».

los números? ¿están en las mismas bases que el código en ensamblador original?, ¿en qué base están ahora?

▶ **1.5** Ejecuta paso a paso el programa, ¿qué números se van almacenando en los registros `r0` al `r3`?

Además de literales numéricos, como se ha comentado anteriormente, también es posible incluir literales alfanuméricos, ya sea caracteres individuales o cadenas de caracteres.

Para especificar un carácter de forma literal se debe entrecomillar entre comillas simples[4]. Por ejemplo:

introsim-letras.s

```
1         .text
2 main:   mov r0, #'H'
3         mov r1, #'o'
4         mov r2, #'l'
5         mov r3, #'a'
6 stop:   wfi
```

EJERCICIOS

Copia el programa anterior en Qt ARMSim, cambia al modo de simulación y contesta las siguientes preguntas.

▶ **1.6** Cuando el simulador ha desensamblado el código máquina, ¿qué ha pasado con las letras «H», «o», «l» y «a»? ¿A qué crees que es debido?

▶ **1.7** Ejecuta paso a paso el programa, ¿qué números se van almacenando en los registros `r0` al `r3`?

Si en lugar de querer especificar un carácter, se quiere especificar una cadena de caracteres, entonces se debe utilizar el prefijo «`#`» y entrecomillar la cadena entre comillas dobles. Por ejemplo, «`#"Hola mundo!"`».

Puesto que la instrucción «`mov rd, #Offset8`» escribe el byte indicado por `Offset8` en el registro `rd`, no tiene sentido utilizar una cadena de caracteres con dicha instrucción. Así que se dejan para más adelante los ejemplos de cómo se suelen utilizar los literales de cadenas.

[4] En realidad basta con poner una comilla simple delante, «`#'A'`» y «`#'A`» son equivalentes.

Otra herramienta que proporciona el lenguaje ensamblador para facilitar la programación y mejorar la lectura del código fuente es la posibilidad de utilizar constantes.

Por ejemplo, supongamos que estamos realizando un código que va a trabajar con los días de la semana. Dicho código utiliza números para representar los números de la semana. El 1 para el lunes, el 2 para el martes y así, sucesivamente. Sin embargo, nuestro código en ensamblador sería mucho más fácil de leer y de depurar si utilizáramos constantes para referirnos a los días de la semana. Por ejemplo, «Monday» para el lunes, «Tuesday» para el martes, etc. De esta forma, podríamos referirnos al lunes con el literal «`#Monday`» en lugar de con «`#1`». Naturalmente, en alguna parte del código tendríamos que especificar que la constante «Monday» debe sustituirse por un 1 en el código, la constante «Tuesday» por un 2, y así sucesivamente.

«`.equ Constant, Value`»

Para declarar una constante se utiliza la directiva «`.equ`»; de la siguiente forma: «`.equ Constant, Value`»[5]. El siguiente programa muestra un ejemplo en el que se declaran y utilizan las constantes «`Monday`» y «`Tuesday`».

introsim-dias.s

```
1         .equ Monday, 1
2         .equ Tuesday, 2
3         @ ...
4
5         .text
6 main:   mov r0, #Monday
7         mov r1, #Tuesday
8         @ ...
9 stop:   wfi
```

EJERCICIOS

▶ **1.8** ¿Dónde se han declarado las constantes en el código anterior? ¿Dónde se han utilizado? ¿Dónde se ha utilizado el el carácter «`#`» y dónde no?

▶ **1.9** Copia el código anterior en Qt ARMSim, ¿qué ocurre al cambiar al modo de simulación? ¿dónde está la declaración de constantes en el código máquina? ¿aparecen las constantes «`Monday`» y «`Tuesday`» en el código máquina?

[5] En lugar de la directiva «`.equ`», se puede utilizar la directiva «`.set`» (ambas directivas son equivalentes). Además, también se pueden utilizar las directivas «`.equiv`» y «`.eqv`», que además de inicializar una constante, permiten evitar errores de programación, ya que comprueban que la constante no se haya definido previamente (en otra parte del código que a lo mejor no hemos escrito nosotros, o que escribimos hace mucho tiempo, lo que viene a ser lo mismo).

▶ **1.10** Modifica el valor de las constantes en el código fuente en ensamblador, guarda el código fuente modificado, y vuelve a ensamblar el código (vuelve al modo de simulación). ¿Cómo se ha modificado el código máquina?

Por último, el ensamblador de ARM también permite personalizar el nombre de los registros. Esto puede ser útil cuando un determinado registro se vaya a utilizar en un momento dado para un determinado propósito. Para asignar un nombre a un registro se puede utilizar la directiva «`.req`» (y para desasociar dicho nombre, la directiva «`.unreq`»). Por ejemplo:

«Name `.req` rd»
«`.unreq` Name»

introsim-diasreq.s

```
1         .equ Monday, 1
2         .equ Tuesday, 2
3         @ ...
4
5         .text
6         day .req r7
7 main:   mov day, #Monday
8         mov day, #Tuesday
9         .unreq day
10        @ ...
11 stop:  wfi
```

EJERCICIOS

▶ **1.11** Copia el código fuente anterior y ensámblalo, ¿cómo se han reescrito las instrucciones «`mov`» en el código máquina?

1.4. Inicialización de datos y reserva de espacio

Prácticamente cualquier programa de computador necesita utilizar datos para llevar a cabo su tarea. Por regla general, estos datos se almacenan en la memoria del computador.

Cuando se programa en un lenguaje de alto nivel se pueden utilizar variables para referenciar a diversos tipo de datos. Será el compilador (o el intérprete, según sea el caso) quien se encargará de decidir en qué posiciones de memoria se almacenarán y cuánto ocuparán los tipos de datos utilizados por cada una de dichas variables. El programador simplemente declara e inicializa dichas variables, pero no se preocupa de indicar cómo ni dónde deben almacenarse.

Bytes, palabras y medias palabras

Los computadores basados en la arquitectura ARM pueden acceder a la memoria a nivel de byte. Esto implica que debe haber una dirección de memoria distinta para cada byte que forme parte de la memoria del computador.

Poder acceder a la memoria a nivel de byte tiene sentido, ya que algunos tipos de datos, por ejemplo los caracteres ASCII, no requieren más que un byte por carácter. Si se utilizara una medida mayor de almacenamiento, se estaría desperdiciando espacio de memoria.

Sin embargo, la capacidad de expresión de un byte es bastante reducida (p.e., si se quisiera trabajar con números enteros habría que contentarse con los números del -128 al 127). Por ello, la mayoría de computadores trabajan de forma habitual con unidades superiores al byte. Esta unidad superior suele recibir el nombre de *palabra* (*word*).

Al contrario de lo que ocurre con un byte que son siempre 8 bits, el tamaño de una palabra depende de la arquitectura. En el caso de la arquitectura ARM, una palabra equivale a 4 bytes. La decisión de que una palabra equivalga a 4 bytes tiene implicaciones en la arquitectura ARM y en la organización de los procesadores basados en dicha arquitectura: registros con un tamaño de 4 bytes, 32 líneas en el bus de datos...

Además de fijar el tamaño de una palabra a 4 bytes, la arquitectura ARM obliga a que las palabras en memoria deban estar alineadas en direcciones de memoria que sean múltiplos de 4.

Por último, además de trabajar con bytes y palabras, también es posible hacerlo con medias palabras (*half-words*). Una media palabra en ARM está formada por 2 bytes y debe estar en una dirección de memoria múltiplo de 2.

Por contra, el programador en ensamblador (o un compilador de un lenguaje de alto nivel) sí debe indicar qué y cuántas posiciones de memoria se deben utilizar para almacenar las variables de un programa, así como indicar sus valores iniciales.

«`.data`»

Los ejemplos que se han visto hasta ahora constaban únicamente de una sección de código (declarada por medio de la directiva «`.text`»). Sin embargo, lo habitual es que un programa en ensamblador tenga dos secciones: una de código y otra de datos. La directiva «`.data`» le indica al ensamblador dónde comienza la sección de datos.

1.4.1. Inicialización de palabras

Para poder ensamblar un código fuente es obligatorio que haya una sección de código con al menos una instrucción.

El siguiente código fuente está formado por dos secciones: una de datos y una de código. En la de datos se inicializan cuatro palabras y en la de código tan solo hay una instrucción de parada («`wfi`»).

datos-palabras.s

```
1       .data       @ Comienzo de la zona de datos
```

```
2 word1:  .word 15   @ Número en decimal
3 word2:  .word 0x15 @ Número en hexadecimal
4 word3:  .word 015  @ Número en octal
5 word4:  .word 0b11 @ Número en binario
6
7         .text
8 stop:   wfi
```

El anterior ejemplo no acaba de ser realmente un programa ya que no contiene instrucciones en lenguaje ensamblador que vayan a realizar alguna tarea. Sin embargo, utiliza una serie de directivas que le indican al ensamblador qué información debe almacenar en memoria y dónde.

La primera de las directivas utilizadas, «`.data`», como se ha comentado hace poco, se utiliza para avisar al ensamblador de que todo lo que aparezca debajo de ella (mientras no se diga lo contrario) debe ser almacenado en la zona de datos.

Las cuatro siguientes líneas utilizan la directiva «`.word`». Esta directiva le indica al ensamblador que se quiere reservar espacio para una palabra e inicializarlo con un determinado valor. La primera de las dos, la «`.word 15`», inicializará[6] una palabra con el número 15 a partir de la primera posición de memoria (por ser la primera directiva de inicialización de memoria después de la directiva «`.data`»). La siguiente, la «`.word 0x15`», inicializará la siguiente posición de memoria disponible con una palabra con el número `0x15`.

«`.word Value32`»

EJERCICIOS

Copia el fichero anterior en Qt ARMSim, ensámblalo y resuelve los siguientes ejercicios.

▶ **1.12** Encuentra los datos almacenados en memoria: localiza dichos datos en el panel de memoria e indica su valor en hexadecimal.

▶ **1.13** ¿En qué direcciones se han almacenado las cuatro palabras? ¿Por qué las direcciones de memoria en lugar de ir de uno en uno van de cuatro a cuatro?

▶ **1.14** Recuerda que las etiquetas sirven para referenciar la posición de memoria de la línea en la que están. Así pues, ¿qué valores toman las etiquetas «`word1`», «`word2`», «`word3`» y «`word4`»?

▶ **1.15** Crea ahora otro programa con el siguiente código:

`datos-palabras2.s`

```
1         .data      @ Comienzo de la zona de datos
2 words:  .word 15, 0x15, 015, 0b11
```

[6] Por regla general, cuando hablemos de directivas que inicializan datos, se sobreentenderá que también reservan el espacio necesario en memoria para dichos datos; por no estar repitiendo siempre reserva e inicialización.

```
3
4         .text
5 stop:   wfi
```

Cambia al modo de simulación. ¿Hay algún cambio en los valores almacenados en memoria con respecto a los almacenados por el programa anterior? ¿Están en el mismo sitio?

▶ **1.16** Teniendo en cuenta el ejercicio anterior, crea un programa en ensamblador que defina un vector[7] de cinco palabras (*words*), asociado a la etiqueta vector, que tenga los siguientes valores: `0x10`, `30`, `0x34`, `0x20` y `60`. Cambia al modo simulador y comprueba que el vector se ha almacenado de forma correcta en memoria.

Big-endian y Litle-endian

Cuando se almacena en memoria una palabra y es posible acceder a posiciones de memoria a nivel de byte, surge la cuestión de en qué orden se deberían almacenar en memoria los bytes que forman una palabra.

Por ejemplo, si se quiere almacenar la palabra `0xAABBCCDD` en la posición de memoria `0x20070000`, la palabra ocupará los 4 bytes: `0x20070000`, `0x20070001`, `0x20070002` y `0x20070003`. Sin embargo, ¿a qué posiciones de memoria irán cada uno de los bytes de la palabra? Las opciones que se utilizan son:

0x20070000	0xAA
0x20070001	0xBB
0x20070002	0xCC
0x20070003	0xDD

a) Big-endian

0x20070000	0xDD
0x20070001	0xCC
0x20070002	0xBB
0x20070003	0xAA

b) Little-endian

En la primera de las dos, la organización *big-endian*, el byte de mayor peso (*big*) de la palabra se almacena en la dirección de memoria más baja (*endian*).

Por el contrario, en la organización *little-endian*, es el byte de menor peso (*little*) de la palabra, el que se almacena en la dirección de memoria más baja (*endian*).

1.4.2. Inicialización de bytes

«`.byte Value8`»

La directiva «`.byte Value8`» sirve para inicializar un byte con el

[7] Un vector es un tipo de datos formado por un conjunto de datos almacenados de forma secuencial. Para poder trabajar con un vector es necesario conocer la dirección de memoria en la que comienza —la dirección del primer elemento— y su tamaño.

contenido Value8.

EJERCICIOS

Teclea el siguiente programa en el editor de Qt ARMSim y ensámblalo.

datos-byte-palabra.s

```
1         .data      @ Comienzo de la zona de datos
2 bytes:  .byte 0x10, 0x20, 0x30, 0x40
3 word:   .word 0x10203040
4
5         .text
6 stop:   wfi
```

▶ **1.17** ¿Qué valores se han almacenado en memoria?

▶ **1.18** Viendo cómo se han almacenado y cómo se muestran en el simulador la secuencia de bytes y la palabra, ¿qué tipo de organización de datos, *big-endian* o *little-endian*, crees que sigue el simulador?

▶ **1.19** ¿Qué valores toman las etiquetas «`bytes`» y «`word`»?

1.4.3. Inicialización de medias palabras y de dobles palabras

Para inicializar medias palabras y dobles palabras se deben utilizar las directivas «`.hword Value16`» y «`.quad Value64`», respectivamente.

«`.hword Value16`»
«`.quad Value64`»

1.4.4. Inicialización de cadenas de caracteres

La directiva «`.ascii "cadena"`» le indica al ensamblador que debe inicializar la memoria con los códigos UTF-8 de los caracteres que componen la cadena entrecomillada. Dichos códigos se almacenan en posiciones consecutivas de memoria.

«`.ascii "cadena"`»

EJERCICIOS

Copia el siguiente código en el simulador y ensámblalo.

datos-cadena.s

```
1         .data      @ Comienzo de la zona de datos
2 str:    .ascii "abcde"
3 byte:   .byte 0xff
4
5         .text
6 stop:   wfi
```

▶ **1.20** ¿Qué rango de posiciones de memoria se han reservado para la variable etiquetada con «`str`»?

▶ **1.21** ¿Cuál es el código UTF-8 de la letra «a»? ¿Y el de la «b»?

▶ **1.22** ¿Qué posición de memoria referencia la etiqueta «`byte`»?

▶ **1.23** ¿Cuántos bytes se han reservado en total para la cadena?

«`.asciz "cadena"`»

▶ **1.24** La directiva «**`.asciz`** `"cadena"`» también sirve para declarar cadenas. Pero hace algo más que tienes que averiguar en este ejercicio.

Sustituye en el programa anterior la directiva «**`.ascii`**» por la directiva «**`.asciz`**» y ensambla de nuevo el código. ¿Hay alguna diferencia en el contenido de la memoria utilizada? ¿Cuál? Describe cuál es la función de esta directiva y cuál crees que puede ser su utilidad con respecto a «**`.ascii`**».

1.4.5. Reserva de espacio

«`.space N`»

La directiva «**`.space`** `N`» se utiliza para reservar N bytes de memoria e inicializarlos a 0.

EJERCICIOS

Dado el siguiente código:

datos-space.s

```
1         .data      @ Comienzo de la zona de datos
2 byte1:  .byte 0x11
3 space:  .space 4
4 byte2:  .byte 0x22
5 word:   .word 0xAABBCCDD
6
7         .text
8 stop:   wfi
```

▶ **1.25** ¿Qué posiciones de memoria se han reservado para almacenar la variable «`space`»?

▶ **1.26** ¿Los cuatro bytes utilizados por la variable «`space`» podrían ser leídos o escritos como si fueran una palabra? ¿Por qué?

▶ **1.27** ¿A partir de qué dirección se ha inicializado «`byte1`»? ¿A partir de cuál «`byte2`»?

▶ **1.28** ¿A partir de qué dirección se ha inicializado «`word`»? ¿La palabra «`word`» podría ser leída o escrita como si fuera una palabra? ¿Por qué?

1.4.6. Alineación de datos en memoria

«`.balign N`» La directiva «`.balign N`» le indica al ensamblador que el siguiente dato que vaya a reservarse o inicializarse, debe comenzar en una dirección de memoria múltiplo de N.

EJERCICIOS

▶ **1.29** Añade en el código anterior dos directivas «`.balign N`» de tal forma que:

- la variable etiquetada con «`space`» comience en una posición de memoria múltiplo de 2, y
- la variable etiquetada con «`word`» esté en un múltiplo de 4.

1.5. Carga y almacenamiento

La arquitectura ARM es una arquitectura del tipo carga/almacenamiento (*load/store*). En este tipo de arquitectura, las únicas instrucciones que acceden a memoria son aquellas encargadas de cargar datos desde la memoria y de almacenar datos en la memoria. El resto de instrucciones requieren que los operandos estén en registros o en la propia instrucción.

Los siguientes subapartados muestran: I) cómo cargar valores constantes en registros, II) cómo cargar de memoria a registros, y III) cómo almacenar en memoria el contenido de los registros.

1.5.1. Carga de datos inmediatos (constantes)

Para cargar un dato inmediato que ocupe un byte se puede utilizar la instrucción «`mov rd, #Inm8`». «`mov rd, #Inm8`»

carga-mov.s

```
1         .text
2 main:   mov r0, #0x12
3         wfi
```

EJERCICIOS

Copia el código anterior y ensámblalo. A continuación, realiza los siguientes ejercicios.

▶ **1.30** Modifica a mano el contenido del registro `r0` para que tenga el valor `0x12345678` (haz doble clic sobre el contenido del registro).

▶ **1.31** Después de modificar a mano el registro `r0`, ejecuta el programa. ¿Qué valor tiene ahora el registro `r0`? ¿Se ha modificado todo el contenido del registro o solo el byte de menor peso del contenido del registro?

. . . . . .

«`ldr rd, =Inm32`»

En el caso de tener que cargar un dato que ocupe más de un byte, no se podría utilizar la instrucción «**`mov`**». Sin embargo, suele ser habitual tener que cargar datos constantes más grandes, por lo que el ensamblador de ARM proporciona una pseudo-instrucción que sí que lo permite: «`ldr rd, =Inm32`». Dicha pseudo-instrucción permite cargar datos inmediatos de hasta 32 bits.

¿Por qué «`ldr rd, =Inm32`» no podría ser una instrucción máquina en lugar de una pseudo-instrucción? Porque puesto que solo el dato inmediato ya ocupa 32 bits, no habría bits suficientes para codificar los otros elementos de la nueva hipotética instrucción máquina. Si recordamos lo comentado anteriormente, las instrucciones máquina de ARM Thumb ocupan generalmente 16 bits (alguna, 32 bits), y puesto que el operando ya estaría ocupando todo el espacio disponible, no sería posible codificar en la instrucción cuál es el registro destino, ni destinar parte de la instrucción a guardar el código de operación (que además de identificar la operación que se debe realizar, permite al procesador distinguir a una instrucción de las restantes de su repertorio).

¿Qué hace el programa ensamblador cuando se encuentra con la pseudo-instrucción «`ldr rd, =Inm32`»? Depende. Si el dato inmediato ocupa un byte, sustituye la pseudo-instrucción por una instrucción «**`mov`**» equivalente. Si por el contrario, el dato inmediato ocupa más de un byte: I) copia el valor del dato inmediato en la memoria ROM, a continuación del código del programa, y II) sustituye la pseudo-instrucción por una instrucción de carga relativa al PC.

El siguiente programa muestra un ejemplo en el que se utiliza la pseudo-instrucción «`ldr rd, =Inm32`». En un primer caso, con un valor que cabe en un byte. En un segundo caso, con un valor que ocupa una palabra entera.

carga-ldr-value.s

```
1         .text
2 main:   ldr r1, =0xFF
3         ldr r2, =0x10203040
4         wfi
```

. . . . . EJERCICIOS . . . . . .

Copia el código anterior, ensámblalo y contesta a las siguientes preguntas.

▶ **1.32** La pseudo-instrucción «`ldr r1, =0xFF`», ¿a qué instrucción ha dado lugar al ser ensamblada?

▶ **1.33** La pseudo-instrucción «`ldr r2, =0x10203040`», ¿a qué instrucción ha dado lugar al ser ensamblada?

▶ **1.34** Localiza el número `0x10203040` en la memoria ROM, ¿dónde está?

▶ **1.35** Ejecuta el programa paso a paso y anota qué valores se almacenan en el registro `r1` y en el registro `r2`.

En lugar de cargar un valor constante puesto a mano, también es posible cargar en un registro la dirección de memoria de una etiqueta. Esto es útil, como se verá en el siguiente apartado, para cargar variables de memoria a un registro.

Para cargar en un registro la dirección de memoria de una etiqueta se utiliza la pseudo-instrucción «`ldr rd, =Label`». El siguiente programa muestra un ejemplo en el que se ha utilizado varias veces dicha pseudo-instrucción.

«`ldr rd, =Label`»

carga-ldr-label.s

```
1         .data
2 word1:  .word 0x10203040
3 word2:  .word 0x11213141
4 word3:  .word 0x12223242
5
6         .text
7 main:   ldr r0, =word1
8         ldr r1, =word2
9         ldr r2, =word3
10        wfi
```

EJERCICIOS

Copia y ensambla el programa anterior. Luego contesta las siguientes preguntas.

▶ **1.36** ¿En qué direcciones de memoria se encuentran las variables etiquetadas con «`word1`», «`word2`» y «`word3`»?

▶ **1.37** ¿Puedes localizar los números que has contestado en la pregunta anterior en la memoria ROM? ¿Dónde?

▶ **1.38** El contenido de la memoria ROM también se muestra en la ventana de desensamblado del simulador, ¿puedes localizar ahí también dichos números? ¿dónde están?

▶ **1.39** Escribe a continuación en qué se han convertido las tres instrucciones «`ldr`».

▶ **1.40** ¿Qué crees que hacen las anteriores instrucciones?

▶ **1.41** Ejecuta el programa, ¿qué se ha almacenado en los registros `r0`, `r1` y `r2`?

▶ **1.42** Anteriormente se ha comentado que las etiquetas se utilizan para hacer referencia a la dirección de memoria en la que se han definido. Sabiendo que en los registros `r0`, `r1` y `r2` se ha almacenado el valor de las etiquetas «`word1`», «`word2`» y «`word3`», respectivamente, ¿se confirma o desmiente dicha afirmación?

▶ **1.43** Repite diez veces:
«Una etiqueta hace referencia a una dirección de memoria, no al contenido de dicha dirección de memoria.»

1.5.2. Carga de palabras (de memoria a registro)

«`ldr rd, [...]`»

Para cargar una palabra de memoria a registro se pueden utilizar las siguientes instrucciones:

- «`ldr rd, [rb]`»,
- «`ldr rd, [rb, #Offset5]`», y
- «`ldr rd, [rb, ro]`».

Las anteriores instrucciones solo se diferencian en la forma en la que indican la dirección de memoria desde la que se quiere cargar una palabra en el registro `rd`.

En la primera variante, «`ldr rd, [rb]`», la dirección de memoria desde la que se quiere cargar una palabra en el registro `rd` es la indicada por el contenido del registro `rb`.

En la segunda variante, «`ldr rd, [rb, #Offset5]`», la dirección de memoria desde la que se quiere cargar una palabra en el registro `rd` se calcula como la suma del contenido del registro `rb` y un desplazamiento inmediato, «`Offset5`». El desplazamiento inmediato, «`Offset5`», debe ser un número múltiplo de 4 entre 0 y 124. Conviene observar que la variante anterior es en realidad una pseudo-instrucción que será sustituida por el ensamblador por una instrucción de este tipo con un desplazamiento de 0, es decir, por «`ldr rd, [rb, #0]`».

En la tercera variante, «`ldr rd, [rb, ro]`», la dirección de memoria desde la que se quiere cargar una palabra en el registro `rd` se calcula como la suma del contenido de los registros `rb` y `ro`.

El siguiente código fuente muestra un ejemplo de cada una de las anteriores variantes de la instrucción «**ldr**».

carga-ldr-rb.s

```
        .data
word1:  .word 0x10203040
word2:  .word 0x11213141
word3:  .word 0x12223242

        .text
main:   ldr r0, =word1   @ r0 <- 0x20070000
        mov r1, #8       @ r1 <- 8
        ldr r2, [r0]
        ldr r3, [r0,#4]
        ldr r4, [r0,r1]
        wfi
```

EJERCICIOS

Copia y ensambla el código anterior. A continuación contesta las siguientes preguntas.

▶ **1.44** La instrucción «**ldr** `r2, [r0]`»:

- ¿En qué instrucción máquina se ha convertido?
- ¿De qué dirección de memoria va a cargar la palabra?
- ¿Qué valor se va a cargar en el registro `r2`?

▶ **1.45** Ejecuta el código paso a paso hasta la instrucción «**ldr** `r2, [r0]`» inclusive y comprueba si es correcto lo que has contestado en el ejercicio anterior.

▶ **1.46** La instrucción «**ldr** `r3, [r0,#4]`»:

- ¿De qué dirección de memoria va a cargar la palabra?
- ¿Qué valor se va a cargar en el registro `r3`?

▶ **1.47** Ejecuta un paso más del programa y comprueba si es correcto lo que has contestado en el ejercicio anterior.

▶ **1.48** La instrucción «**ldr** `r4, [r0,r1]`»:

- ¿De qué dirección de memoria va a cargar la palabra?
- ¿Qué valor se va a cargar en el registro `r4`?

▶ **1.49** Ejecuta un paso más del programa y comprueba si es correcto lo que has contestado en el ejercicio anterior.

1.5.3. Carga de bytes y medias palabras (de memoria a registro)

Cuando se quiere cargar un byte o una media palabra, hay que tener en cuenta que es necesario saber si el valor almacenado en memoria se trata de un número con signo o no. Esto es así ya que en el caso de que se trate de un número con signo, además de cargar el byte o la media palabra, será necesario indicarle al procesador que debe extender su signo al resto de bits del registro.

Las instrucciones para cargar bytes son:

Sin extensión de signo	Con extensión de signo
«`ldrb rd, [rb]`»	«`ldrb rd, [rb]`» «`sxtb rd, rd`»
«`ldrb rd, [rb, #Offset5]`»	«`ldrb rd, [rb, #Offset5]`» «`sxtb rd, rd`»
«`ldrb rd, [rb, ro]`»	«`ldrsb rd, [rb, ro]`»

Como se puede ver en el cuadro anterior, las dos primeras variantes, las que permiten cargar bytes con desplazamiento inmediato, no tienen una variante equivalente que cargue y, a la vez, extienda el signo del byte. Una opción para hacer esto mismo, sin recurrir a la tercera variante, es la de usar la instrucción de carga sin signo y luego utilizar la instrucción «`sxtb rd, rm`», que extiende el signo del byte a la palabra.

Hay que tener en cuenta que el desplazamiento, «`Offset5`», debe ser un número comprendido entre 0 y 31.

Ejemplo con «`ldrb`»:

carga-ldrb.s

```
1         .data
2  byte1:  .byte -15
3  byte2:  .byte 20
4  byte3:  .byte 40
5
6         .text
7  main:   ldr r0, =byte1   @ r0 <- 0x20070000
8          mov r1, #2       @ r1 <- 2
9          @ Sin extensión de signo
10         ldrb r2, [r0]
11         ldrb r3, [r0,#1]
12         ldrb r4, [r0,r1]
13         @ Con extensión de signo
14         ldrb r5, [r0]
15         sxtb r5, r5
16         ldrsb r6, [r0,r1]
17 stop:   wfi
```

En cuanto a la carga de medias palabras, las instrucciones utilizadas son:

Sin extensión de signo	Con extensión de signo
«`ldrh rd, [rb]`»	«`ldrh rd, [rb]`» «`sxth rd, rd`»
«`ldrh rd, [rb, #Offset5]`»	«`ldrh rd, [rb, #Offset5]`» «`sxth rd, rd`»
«`ldrh rd, [rb, ro]`»	«`ldrsh rd, [rb, ro]`»

Como se puede ver en el cuadro anterior, y al igual que ocurría con las instrucciones de carga de bytes, las dos primeras variantes que permiten cargar medias palabras con desplazamiento inmediato, no tienen una variante equivalente que cargue y, a la vez, extienda el signo de la media palabra. Una opción para hacer esto mismo, sin recurrir a la tercera variante, es la de usar la instrucción de carga sin signo y luego utilizando la instrucción «`sxth rd, rm`», que extiende el signo de la media palabra a la palabra.

Hay que tener en cuenta que el desplazamiento, «`Offset5`», debe ser un número múltiplo de 2 comprendido entre 0 y 62.

Ejemplo con «`ldrh`»:

carga-ldrh.s

```
1         .data
2 half1:  .hword -15
3 half2:  .hword 20
4 half3:  .hword 40
5
6         .text
7 main:   ldr r0, =half1   @ r0 <- 0x20070000
8         mov r1, #4       @ r1 <- 4
9         @ Sin extensión de signo
10        ldrh r2, [r0]
11        ldrh r3, [r0,#2]
12        ldrh r4, [r0,r1]
13        @ Con extensión de signo
14        ldrh r5, [r0]
15        sxth r5, r5
16        ldrsh r6, [r0,r1]
17 stop:  wfi
```

1.5.4. Almacenamiento de palabras (de registro a memoria)

Para almacenar una palabra en memoria desde un registro se pueden «`str rd, [...]`»

utilizar las siguientes instrucciones:

- «`str rd, [rb]`»,
- «`str rd, [rb, #0ffset5]`», y
- «`str rd, [rb, ro]`».

Las anteriores instrucciones solo se diferencian en la forma en la que indican la dirección de memoria en la que se quiere almacenar el contenido del registro `rd`.

En la primera variante, «`str rd, [rb]`», la dirección de memoria en la que se quiere almacenar el contenido del registro `rd` es la indicada por el contenido del registro `rb`.

En la segunda variante, «`str rd, [rb, #0ffset5]`», la dirección de memoria en la que se quiere almacenar el contenido del registro `rd` se calcula como la suma del contenido del registro `rb` y un desplazamiento inmediato, «`0ffset5`». El desplazamiento inmediato, «`0ffset5`», debe ser un número múltiplo de 4 entre 0 y 124. Conviene observar que la variante anterior es en realidad una pseudo-instrucción que será sustituida por el ensamblador por una instrucción de este tipo con un desplazamiento de 0, es decir, por «`str rd, [rb, #0]`».

En la tercera variante, «`str rd, [rb, ro]`», la dirección de memoria en la que se quiere almacenar el contenido del registro `rd` se calcula como la suma del contenido de los registros `rb` y `ro`.

El siguiente código fuente muestra un ejemplo con cada una de las anteriores variantes de la instrucción «`str`».

carga-str-rb.s

```
1         .data
2  word1:  .space 4
3  word2:  .space 4
4  word3:  .space 4
5
6         .text
7  main:   ldr r0, =word1   @ r0 <- 0x20070000
8          mov r1, #8       @ r1 <- 8
9          mov r2, #16      @ r2 <- 16
10         str r2, [r0]
11         str r2, [r0,#4]
12         str r2, [r0,r1]
13
14 stop:   wfi
```

EJERCICIOS

Copia y ensambla el código anterior. A continuación contesta las siguientes preguntas.

▶ **1.50** La instrucción «`str r2, [r0]`»:

- ¿En qué instrucción máquina se ha convertido?
- ¿En qué dirección de memoria va a almacenar la palabra?
- ¿Qué valor se va a almacenar en dicha dirección de memoria?

▶ **1.51** Ejecuta el código paso a paso hasta la instrucción «`str r2, [r0]`» inclusive y comprueba si es correcto lo que has contestado en el ejercicio anterior.

▶ **1.52** La instrucción «`str r2, [r0,#4]`»:

- ¿En qué dirección de memoria va a almacenar la palabra?
- ¿Qué valor se va a almacenar en dicha dirección de memoria?

▶ **1.53** Ejecuta un paso más del programa y comprueba si es correcto lo que has contestado en el ejercicio anterior.

▶ **1.54** La instrucción «`str r2, [r0,r1]`»:

- ¿En qué dirección de memoria va a almacenar la palabra?
- ¿Qué valor se va a almacenar en dicha dirección de memoria?

▶ **1.55** Ejecuta un paso más del programa y comprueba si es correcto lo que has contestado en el ejercicio anterior.

1.5.5. Almacenamiento de bytes y medias palabras (de registro a memoria)

Para almacenar bytes o medias palabras se pueden utilizar las mismas variantes que las descritas en el apartado anterior.

Para almacenar bytes se pueden utilizar las siguientes variantes de la instrucción «`strb`»:

- «`strb rd, [rb]`»,
- «`strb rd, [rb, #Offset5]`», y
- «`strb rd, [rb, ro]`».

Como ya se comentó en el caso de la instrucción «`ldrb`», el desplazamiento «`Offset5`» debe ser un número comprendido entre 0 y 31.

Ejemplo con «`strb`»:

carga-strb.s

```
1         .data
2  byte1:  .space 1
3  byte2:  .space 1
4  byte3:  .space 1
5
6         .text
7  main:   ldr r0, =byte1   @ r0 <- 0x20070000
8          mov r1, #2       @ r1 <- 2
9          mov r2, #10      @ r2 <- 10
10         strb r2, [r0]
11         strb r2, [r0,#1]
12         strb r2, [r0,r1]
13 stop:   wfi
```

Para almacenar medias palabras se pueden utilizar las siguientes variantes de la instrucción «**strh**»:

- «**strh** rd, [rb]»,
- «**strh** rd, [rb, #Offset5]», y
- «**strh** rd, [rb, ro]».

Como ya se comentó en el caso de la instrucción «**ldrh**», el desplazamiento «Offset5» debe ser un número múltiplo de 2 comprendido entre 0 y 62.

Ejemplo con «**strh**»:

carga-strh.s

```
1         .data
2  hword1: .space 2
3  hword2: .space 2
4  hword3: .space 2
5
6         .text
7  main:   ldr r0, =hword1  @ r0 <- 0x20070000
8          mov r1, #4       @ r1 <- 4
9          mov r2, #10      @ r2 <- 10
10         strh r2, [r0]
11         strh r2, [r0,#2]
12         strh r2, [r0,r1]
13 stop:   wfi
```

1.6. Problemas del capítulo

EJERCICIOS

▶ **1.56** Desarrolla un programa ensamblador que reserve espacio para dos vectores consecutivos, A y B, de 20 palabras.

▶ **1.57** Desarrolla un programa ensamblador que realice la siguiente reserva de espacio en memoria: una palabra, un byte y otra palabra alineada en una dirección múltiplo de 4.

▶ **1.58** Desarrolla un programa ensamblador que realice la siguiente reserva de espacio e inicialización de memoria: una palabra con el valor `3`, un byte con el valor `0x10`, una reserva de 4 bytes que comience en una dirección múltiplo de 4, y un byte con el valor `20`.

▶ **1.59** Desarrolla un programa ensamblador que inicialice, en el espacio de datos, la cadena de caracteres «Esto es un problema», utilizando:

a) La directiva «**`.ascii`**»

b) La directiva «**`.byte`**»

c) La directiva «**`.word`**»

*(Pista: Comienza utilizando solo la directiva «**`.ascii`**» y visualiza cómo se almacena en memoria la cadena para obtener la secuencia de bytes.)*

▶ **1.60** Sabiendo que un entero ocupa una palabra, desarrolla un programa ensamblador que inicialice en la memoria la matriz A de enteros definida como:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

suponiendo que:

a) La matriz A se almacena por filas (los elementos de una misma fila se almacenan de forma contigua en memoria).

b) La matriz A se almacena por columnas (los elementos de una misma columna se almacenan de forma contigua en memoria).

▶ **1.61** Desarrolla un programa ensamblador que inicialice un vector de enteros, V, definido como $V = (10, 20, 25, 500, 3)$ y cargue los elementos del vector en los registros `r0` al `r4`.

▶ **1.62** Amplía el anterior programa para que además copie a memoria el vector V justo a continuación de éste.

▶ **1.63** Desarrolla un programa ensamblador que dada la siguiente palabra, `0x1020 3040`, almacenada en una determinada posición de memoria, la reorganice en otra posición de memoria invirtiendo el orden de sus bytes.

▶ **1.64** Desarrolla un programa ensamblador que dada la siguiente palabra, `0x1020 3040`, almacenada en una determinada posición de memoria, la reorganice en la misma posición intercambiando el orden de sus medias palabras. (Nota: recuerda que las instrucciones «**`ldrh`**» y «**`strh`**» cargan y almacenan, respectivamente, medias palabras).

▶ **1.65** Desarrolla un programa ensamblador que inicialice cuatro bytes con los valores `0x10`, `0x20`, `0x30`, `0x40`; reserve espacio para una palabra a continuación; y transfiera los cuatro bytes iniciales a la palabra reservada.

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
```

«**add**» *(inm)*

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`.

«**sub**» *(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`.

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 «`0x7FFF FFFE`» en la posición de memoria `0x2007 0000`, 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`.

«**sub**»

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

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 [$res1$] y el número complementado [$res2$]? ¿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:

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

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`.

CAPÍTULO 3

Instrucciones de control de flujo

Índice

La mayor parte de los programas mostrados en los capítulos anteriores constaban de una serie de instrucciones que se debían ejecutar una tras otra, siguiendo el orden en el que se habían escrito, hasta que el programa finalizaba. Este tipo de ejecución, en el que las instrucciones se ejecutan una tras otra, siguiendo el orden en el que están en memoria, recibe el nombre de *ejecución secuencial*.

La ejecución secuencial presenta bastantes limitaciones. Un programa de ejecución secuencial no puede, por ejemplo, tomar diferentes acciones en función de los datos de entrada, o de los resultados obtenidos, o de la interacción con el usuario. Tampoco puede repetir un número de veces ciertas operaciones, a no ser que el programador haya repetido varias veces las mismas operaciones en el programa. Pero incluso en ese

caso, el programa sería incapaz de variar el número de veces que dichas instrucciones deberían ejecutarse.

Debido a la gran ventaja que supone el que un programa pueda tomar diferentes acciones y que pueda repetir un conjunto de instrucciones un determinado número de veces, los lenguajes de programación proporcionan estructuras de control que permiten llevar a cabo dichas acciones: las estructuras de control condicionales y repetitivas. Estas estructuras de control permiten modificar el flujo secuencial de instrucciones. En particular, las estructuras de control condicionales permiten la ejecución de ciertas partes del código en función de una serie de condiciones, mientras que las estructuras de control repetitivas permiten la repetición de cierta parte del código hasta que se cumpla una determinada condición de parada.

Este capítulo está organizado como sigue. El Apartado 3.1 describe el registro `CCR` de ARM, ya que todas las instrucciones de control de flujo condicional de ARM utilizan el contenido de dicho registro para determinar qué acción tomar. El Apartado 3.2 muestra qué son y para qué se utilizan los saltos incondicionales y condicionales. El Apartado 3.3 describe las estructuras de control condicionales *if-then* e *if-then-else*. El Apartado 3.4 presenta las estructuras de control repetitivas *while* y *for*. El último apartado propone una serie de ejercicios más avanzados relacionados con el contenido de este capítulo.

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.6 «*ARM's Flow Control 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.

3.1. El registro `CCR`

Antes de ver cómo se implementan las estructuras de control condicionales y repetitivas, es necesario explicar que cada vez que se ejecuta una instrucción aritmética, se actualiza el *Condition Code Register*[1] (`CCR`). Por ejemplo, al ejecutar una instrucción de suma «**`add`**», se actualiza el indicador de acarreo `C` (que se corresponde con un bit concreto del

[1] Los distintos fabricantes de computadores llaman de forma distinta al registro en el que se almacena el estado del computador después de la ejecución de una operación. Por ejemplo, ARM llama a este registro *Current Processor Status Register* (`CPSR`), mientras que Intel lo denomina *Status Register* (`SR`). Nosotros seguiremos la notación utilizada en [Cle14], que llama a este registro *Condition Code Register* (`CCR`).

registro CCR). Así mismo, al ejecutar la instrucción «**cmp**», que compara dos registros, se actualiza, entre otros, el indicador de cero Z (que se corresponde con otro de los bits del registro CCR). Los valores de los indicadores (*flags*, en inglés) del registro CCR son usados por las instrucciones de salto para decidir qué camino debe tomar el programa.

Los cuatro indicadores del registro CCR que se utilizan por las instrucciones de control de flujo son:

N: Se activa[2] cuando el resultado de la operación es negativo (el 0 se considera positivo).

Z: Se activa cuando el resultado de la operación es 0.

C: Se activa cuando el resultado de la operación produce un acarreo (llamado *carry* en inglés).

V: Se activa cuando el resultado de la operación produce un desbordamiento en operaciones con signo (*overflow*).

La instrucción «**cmp** r1, r2» actualiza el registro CCR en función de los valores almacenados en los registros r1 y r2. Para ello, y como se ha visto en el capítulo anterior, realiza la resta r1 − r2 y, dependiendo del resultado obtenido, activa o no los distintos indicadores. Por ejemplo, si suponemos que los registros r1, r2 y r3 contienen los valores 5, 3 y 5, respectivamente, entonces:

- Tras la operación «**cmp** r1, r2», el indicador N no se activará puesto que el resultado de la comparación no es negativo (r1 − r2 es un número positivo). El indicador Z tampoco se activará puesto que los registros contienen valores diferentes (r1 − r2 es distinto de cero).
- Tras la operación «**cmp** r2, r1», el indicador N se activará puesto que el resultado de la comparación produce un número negativo (r2 − r1 es un número negativo). El indicador Z no se activará puesto que los registros contienen valores diferentes.
- Tras la operación «**cmp** r1, r3», el indicador N no se activará puesto que el resultado de la comparación no es negativo (el cero se considera positivo). El indicador Z si se activará puesto que los registros contienen valores iguales (r1 − r3 es igual a cero).

[2]Como ya se sabe, un bit puede tomar uno de dos valores: 0 o 1. Cuando decimos que un bit está activado (*set*), nos referimos a que dicho bit tiene el valor 1. Cuando decimos que un bit está desactivado (*clear*), nos referimos a que dicho bit tiene el valor 0. Además, cuando decimos que un bit se activa (se pone a 1) bajo determinadas circunstancias, se sobreentiende que si no se dan dichas circunstancias, dicho bit se desactiva (se pone a 0). Lo mismo ocurre, pero al revés, cuando decimos que un bit se desactiva bajo determinadas circunstancias.

Los indicadores del registro `CCR` se muestran en Qt ARMSim en la esquina inferior derecha de la ventana del simulador. La Figura 3.1 muestra un ejemplo donde el indicador `N` está activo y los otros tres no.

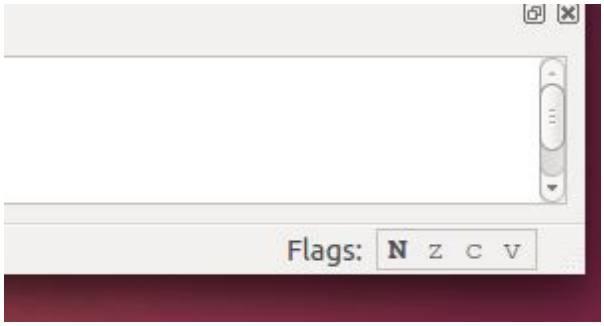


Figura 3.1: Indicadores mostrados por el simulador Qt ARMSim

El siguiente programa muestra un ejemplo en el que se modifican los indicadores del registro `CCR` comparando el contenido de distintos registros.

cap3-E0.s

```
1         .text
2 main:   mov r1, #10
3         mov r2, #5
4         mov r3, #10
5         cmp r1, r2
6         cmp r1, r3
7         cmp r2, r3
8 stop:   wfi
```

EJERCICIOS

Copia y ensambla el programa anterior. A continuación, ejecuta el programa paso a paso conforme vayas respondiendo las siguientes preguntas:

▶ **3.1** Carga y ejecuta el programa anterior. ¿Se activa el indicador `N` tras la ejecución de la instrucción «`cmp r1, r2`»? ¿y el indicador `Z`?

▶ **3.2** ¿Se activa el indicador `N` tras la ejecución de la instrucción «`cmp r1, r3`»? ¿y el indicador `Z`?

▶ **3.3** ¿Se activa el indicador `N` tras la ejecución de la instrucción «`cmp r2, r3`»? ¿y el indicador `Z`?

3.2. Saltos incondicionales y condicionales

En este apartado se describen las instrucciones de salto disponibles en el ensamblador Thumb de ARM. En primer lugar se verá qué son los saltos incondicionales y a continuación, los saltos condicionales.

3.2.1. Saltos incondicionales

Se denominan saltos incondicionales a aquéllos que se realizan siempre, que no dependen de que se cumpla una determinada condición. La instrucción de salto incondicional es «`b etiqueta`», donde «`etiqueta`» referencia la línea del programa a la que se quiere saltar. Al tratarse de una instrucción de salto incondicional, cada vez que se ejecuta la instrucción «`b etiqueta`», el programa saltará a la instrucción etiquetada con «`etiqueta`», independientemente de qué valor tenga el registro `CCR`.

El siguiente programa muestra un ejemplo de salto incondicional.

cap3-E1.s

```
1         .text
2 main:   mov r0, #5
3         mov r1, #10
4         mov r2, #100
5         mov r3, #0
6         b salto
7         add r3, r1, r0
8 salto:  add r3, r3, r2
9 stop:   wfi
```

EJERCICIOS

▶ **3.4** Carga y ejecuta el programa anterior. ¿Qué valor almacena el registro `r3` al finalizar el programa?

▶ **3.5** Comenta completamente la línea «`b salto`» (o bórrala) y vuelve a ejecutar el programa. ¿Qué valor almacena ahora el registro `r3` al finalizar el programa?

▶ **3.6** ¿Qué pasaría si la etiqueta «`salto`» estuviera situada en la línea «`mov r1, #10`», es decir, antes de la instrucción «`b salto`»?

▶ **3.7** Crea un nuevo código basado en el código anterior, pero en el que la línea etiquetada con «`salto`» sea la línea «`mov r1,#10`». Ejecuta el programa paso a paso y comprueba que lo que ocurre coincide con lo que habías deducido en el ejercicio anterior.

3.2.2. Saltos condicionales

Las instrucciones de salto condicional tiene la forma «**bXX** `etiqueta`», donde «`XX`» se sustituye por un nemotécnico que indica el tipo de condición que se debe cumplir para realizar el salto y «`etiqueta`» referencia a la línea del programa a la que se quiere saltar. Las instrucciones de salto condicional comprueban ciertos indicadores del registro `CCR` para decidir si se debe saltar o no. Por ejemplo, la instrucción «**beq** `etiqueta`» (*branch if equal*) comprueba si el indicador `Z` está activo. Si está activo, entonces salta a la instrucción etiquetada con «`etiqueta`». Si no lo está, el programa continúa con la siguiente instrucción. De forma similar, «**bne** `etiqueta`» (*branch if not equal*) saltará a la instrucción etiquetada con «`etiqueta`» si el indicador `Z` no está activo, si esta activo, no saltará.

El Cuadro 3.1 muestra las distintas instrucciones de salto condicional (en el Cuadro 3.2 de [Cle14] se puede consultar también la codificación en binario asociada a cada instrucción de salto).

Cuadro 3.1: Instrucciones de salto condicional

Instrucción	Condición de salto
«**beq**» (*branch if equal*)	Iguales (`Z`)
«**bne**» (*branch if not equal*)	No iguales (`z`)
«**bcs**» (*branch if carry set*)	Mayor o igual sin signo (`C`)
«**bcc**» (*branch if carry clear)*	Menor sin signo (`c`)
«**bmi**» (*branch if minus*)	Negativo (`N`)
«**bpl**» (*branch if plus*)	Positivo o cero (`n`)
«**bvs**» (*branch if overflow set*)	Desbordamiento (`V`)
«**bvc**» (*branch if overflow clear*)	No hay desbordamiento (`v`)
«**bhi**» (*branch if higher*)	Mayor sin signo (`Cz`)
«**bls**» (*branch if lower of same*)	Menor o igual sin signo (`c` o `Z`)
«**bge**» (*branch if greater or equal*)	Mayor o igual (`NV` o `nv`)
«**blt**» (*branch if less than*)	Menor que (`Nv` o `nV`)
«**bgt**» (*branch if greater than*)	Mayor que (`z` y (`NV` o `nv`))
«**ble**» (*branch if less than or equal*)	Menor o igual (`Nv` o `nV` o `Z`)

El siguiente ejemplo muestra un programa en el que se utiliza la instrucción «**beq**» para saltar en función del resultado de la operación anterior.

cap3-E2.s

```
1         .text
2 main:   mov r0, #5
3         mov r1, #10
4         mov r2, #5
5         mov r3, #0
```

```
6         cmp r0, r2
7         beq salto
8         add r3, r0, r1
9   salto:  add r3, r3, r1
10  stop:   wfi
```

EJERCICIOS

▶ **3.8** Carga y ejecuta el programa anterior. ¿Qué valor tiene el registro `r3` cuando finaliza el programa?

▶ **3.9** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r2`»? Para contestar a esta pregunta deberás recargar la simulación y detener la ejecución justo después de ejecutar dicha instrucción.

▶ **3.10** Cambia la línea «**`cmp`** `r0, r2`» por «**`cmp`** `r0, r1`» y vuelve a ejecutar el programa. ¿Qué valor tiene ahora el registro `r3` cuando finaliza el programa?

▶ **3.11** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r1`»?

▶ **3.12** ¿Por qué se produce el salto en el primer caso, «**`cmp`** `r0, r2`», y no en el segundo, «**`cmp`** `r0, r1`»?

Veamos ahora el funcionamiento de la instrucción «**`bne`** `etiqueta`». Para ello, se usará el programa anterior (sin la modificación propuesta en los ejercicios anteriores), pero en el que deberás sustituir la instrucción «**`beq`** `salto`» por «**`bne`** `salto`».

EJERCICIOS

▶ **3.13** Carga y ejecuta el programa. ¿Qué valor tiene el registro `r3` cuando finaliza el programa?

▶ **3.14** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r2`»?

▶ **3.15** Cambia la línea «**`cmp`** `r0, r2`» por «**`cmp`** `r0, r1`» y vuelve a ejecutar el programa. ¿Qué valor tiene ahora el registro `r3` cuando finaliza el programa?

▶ **3.16** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r1`»?

▶ **3.17** ¿Por qué no se produce el salto en el primer caso, «**`cmp`** `r0, r2`», y sí en el segundo, «**`cmp`** `r0, r1`»?

3.3. Estructuras de control condicionales

En este apartado se describen las estructuras de control condicionales *if-then* e *if-then-else*.

3.3.1. Estructura condicional *if-then*

La estructura condicional *if-then* está presente en todos los lenguajes de programación y se usa para realizar o no un conjunto de acciones dependiendo de una condición. A continuación se muestra un programa escrito en Python3 que utiliza la estructura *if-then*.

```
X = 1
Y = 1
Z = 0

if (X == Y):
        Z = X + Y
```

El programa anterior comprueba si el valor de `X` es igual al valor de `Y` y en caso de que así sea, suma los dos valores, almacenando el resultado en `Z`. Si no son iguales, `Z` permanecerá inalterado.

Una posible implementación del programa anterior en ensamblador Thumb de ARM, sería la siguiente:

cap3-E3.s

```
        .data
X:      .word 1
Y:      .word 1
Z:      .word 0

        .text
main:   ldr r0, =X
        ldr r0, [r0]    @ r0 <- [X]
        ldr r1, =Y
        ldr r1, [r1]    @ r1 <- [Y]

        cmp r0, r1
        bne finsi
        add r2, r0, r1  @-
        ldr r3, =Z      @ [Z] <- [X] + [Y]
        str r2, [r3]    @-

finsi:  wfi
```

EJERCICIOS

▶ **3.18** Carga y ejecuta el programa anterior. ¿Qué valor tiene la posición de memoria `Z` cuando finaliza el programa?

▶ **3.19** Modifica el código para que el valor de `Y` sea distinto del de `X` y vuelve a ejecutar el programa. ¿Qué valor hay ahora en la posición de memoria `Z` cuando finaliza el programa?

Como has podido comprobar, la idea fundamental para implementar la instrucción «**`if x==y:`**» ha consistido en utilizar una instrucción de salto condicional que salte si no se cumple dicha condición, «**`bne`**». De esta forma, si los dos valores comparados son iguales, el programa continuará, ejecutando el bloque de instrucciones que deben ejecutarse solo si «`x==y`» (una suma en este ejemplo). En caso contrario, se producirá el salto y dicho bloque no se ejecutará.

EJERCICIOS

▶ **3.20** Cambia el programa anterior para que realice la suma cuando X sea mayor a Y.

3.3.2. Estructura condicional *if-then-else*

Sea el siguiente programa en Python3:

```
1 X = 1
2 Y = 1
3 Z = 0
4
5 if (X == Y):
6         Z = X + Y
7 else:
8         Z = X + 5
```

En este caso, si se cumple la condición (¿son iguales `X` y `Y`?) se realiza una acción, sumar `X` y `Y`, y si no son iguales, se realiza otra acción diferente, sumar el número 5 a `X`.

Una posible implementación del programa anterior en ensamblador Thumb de ARM, sería la siguiente:

cap3-E4.s

```
1         .data
2 X:      .word 1
3 Y:      .word 1
4 Z:      .word 0
5
6         .text
7 main:   ldr r0, =X
8         ldr r0, [r0]            @ r0 <- [X]
9         ldr r1, =Y
10        ldr r1, [r1]            @ r1 <- [Y]
```

```
11
12          cmp r0, r1
13          bne else
14          add r2, r0, r1          @ r2 <- [X] + [Y]
15          b finsi
16
17  else:   add r2, r0, #5          @ r2 <- [X] + 5
18
19  finsi:  ldr r3, =Z
20          str r2, [r3]            @ [Z] <- r2
21
22  stop:   wfi
```

EJERCICIOS

▶ **3.21** Carga y ejecuta el programa anterior. ¿Qué valor hay en la posición de memoria `Z` cuando finaliza el programa?

▶ **3.22** Cambia el valor de `Y` para que sea distinto de `X` y vuelve a ejecutar el programa. ¿Qué valor hay ahora la posición de memoria `Z` al finalizar el programa?

▶ **3.23** Supón que el programa anterior en Python3, en lugar de la línea «`if X == Y:`», tuviera la línea «`if X > Y:`». Cambia el programa en ensamblador para se tenga en cuenta dicho cambio. ¿Qué modificaciones has realizado en el programa en ensamblador?

▶ **3.24** Supón que el programa anterior en Python3, en lugar de la línea «`if X == Y:`», tuviera la línea «`if X <= Y:`». Cambia el programa en ensamblador para se tenga en cuenta dicho cambio. ¿Qué modificaciones has realizado en el programa en ensamblador?

3.4. Estructuras de control repetitivas

Una vez vistas las estructuras de control condicionales, vamos a ver ahora las estructuras de control repetitivas *while* y *for*.

3.4.1. Estructura de control repetitiva *while*

La estructura de control repetitiva *while* permite ejecutar repetidamente un bloque de código mientras se siga cumpliendo una determinada condición. La estructura *while* funciona igual que una estructura *if-then*, en el sentido de que si se cumple la condición evaluada, se ejecuta el código asociado al cumplimiento de dicha condición. Pero a diferencia de la estructura *if-then*, una vez se ha ejecutado la última instrucción del

código asociado a la condición, el control vuelve a la evaluación de la condición y todo el proceso se vuelve a repetir.

El siguiente programa en Python3 muestra el uso de la estructura de control repetitiva *while*.

```
X = 1
E = 1
LIM  = 100

while (X<LIM):
    X = X + 2 * E
    E = E + 1
    print(X)
```

El programa anterior realizará las operaciones $X = X + 2 \cdot E$ y $E = E + 1$ mientras se cumpla que $X < LIM$. Por lo tanto, la variable X irá tomando los siguientes valores con cada iteración del bucle: $3, 7, 13, 21, 31, 43, 57, 73, 91, 111$.

Una posible implementación del programa anterior en ensamblador Thumb de ARM sería la siguiente:

cap3-E6.s

```
        .data
X:      .word 1
E:      .word 1
LIM:    .word 100

        .text
main:   ldr r0, =X
        ldr r0, [r0]    @ r0 <- X
        ldr r1, =E
        ldr r1, [r1]    @ r1 <- E
        ldr r2, =LIM
        ldr r2, [r2]    @ r2 <- LIM

bucle:  cmp r0, r2
        bpl finbuc
        lsl r3, r1, #1  @ r3 <- 2 * [E]
        add r0, r0, r3  @ r0 <- [X] + 2 * [E]
        add r1, r1, #1  @ r1 <- [E] + 1
        ldr r4, =X
        str r0, [r4]    @ [X] <- r0
        ldr r4, =E
        str r1, [r4]    @ [E] <- r1
        b   bucle

finbuc: wfi
```

EJERCICIOS

▶ **3.25** ¿Qué hacen las instrucciones «`cmp r0, r2`», «`bpl finbuc`» y «`b bucle`»?

▶ **3.26** ¿Por qué se ha usado «`bpl`» y no «`bmi`»?

▶ **3.27** ¿Qué indicador del registro `CCR` se está comprobando cuando se ejecuta la instrucción «`bpl`»? ¿Cómo debe estar dicho indicador, activado o desactivado, para que se ejecute el interior del bucle?

▶ **3.28** ¿Qué instrucción se ha utilizado para calcular $2 \cdot E$? ¿Qué hace dicha instrucción?

▶ **3.29** En el código mostrado se actualiza el contenido de las variables X y E en cada iteración del bucle. ¿Se te ocurre algún tipo de optimización que haga que dicho bucle se ejecute mucho más rápido? (El resultado final del programa debe ser el mismo.)

3.4.2. Estructura de control repetitiva *for*

En muchas ocasiones es necesario repetir un conjunto de acciones un número predeterminado de veces. Esto podría conseguirse utilizando un contador que se fuera incrementando de uno en uno y un bucle *while* que comprobara que dicho contador no ha alcanzado un determinado valor. Sin embargo, como dicha estructura se da con bastante frecuencia, los lenguajes de programación de alto nivel suelen proporcionar una forma de llevarla a cabo directamente.

El siguiente programa muestra un ejemplo de uso de la estructura de control repetitiva *for* en Python3.

```
1 V = [2, 4, 6, 8, 10]
2 n = 5
3 suma = 0
4
5 for i in range(n):  # i = [0..N-1]
6   suma = suma + V[i]
```

El programa anterior suma todos los valores de un vector V y almacena el resultado en la variable *suma*. Puesto que en dicho programa se sabe el número de iteraciones que debe realizar el bucle (que es igual al número de elementos del vector), la estructura ideal para resolver dicho problema es el bucle *for*. Se deja como ejercicio para el lector pensar en cómo resolver el mismo problema utilizando la estructura *while*.

Por otro lado, un programa más realista en Python3 no necesitaría la variable n, ya que sería posible obtener la longitud del vector utilizando la función «`len(V)`». Si se ha utilizado la variable n ha sido para acercar

el problema al caso del ensamblador, en el que sí que se va a tener que recurrir a dicha variable.

Una posible implementación del programa anterior en ensamblador Thumb de ARM sería la siguiente:

cap3-E5.s

```
1         .data
2 V:      .word 2, 4, 6, 8, 10
3 n:      .word 5
4 suma:   .word 0
5
6         .text
7 main:   ldr r0, =V      @ r0 <- dirección de V
8         ldr r1, =n
9         ldr r1, [r1]    @ r1 <- n
10        ldr r2, =suma
11        ldr r2, [r2]    @ r2 <- suma
12        mov r3, #0      @ r3 <- 0
13
14 bucle:  cmp r3, r1
15        beq finbuc
16        ldr r4, [r0]
17        add r2, r2, r4  @ r2 <- r2 + V[i]
18        add r0, r0, #4
19        add r3, r3, #1
20        b bucle
21
22 finbuc: ldr r0, =suma
23        str r2, [r0]    @ [suma] <- r2
24
25 stop:   wfi
```

Como se puede comprobar en el código anterior, el índice del bucle está almacenado en el registro `r3` y la longitud del vector en el registro `r1`. En cada iteración se comprueba si el índice es igual a la longitud del vector. En caso positivo, se salta al final del bucle y en caso negativo, se realizan las operaciones indicadas en el bucle.

EJERCICIOS

▶ **3.30** ¿Para qué se utilizan las siguientes instrucciones: «`cmp r3, r1`», «`beq finbuc`», «`add r3, r3, #1`» y «`b bucle`»?

▶ **3.31** ¿Qué indicador del registro `CCR` se está comprobando cuando se ejecuta la instrucción «`beq finbuc`»?

▶ **3.32** ¿Qué contiene el registro `r0`? ¿Para qué sirve la instrucción «`ldr r4, [r0]`»?

▶ **3.33** ¿Para qué sirve la instrucción «`add r0, r0, #4`»?

▶ **3.34** ¿Qué valor tiene la posición de memoria «`suma`» cuando finaliza la ejecución del programa? ¿Y si $V = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ y $n = 10$?

3.5. Problemas del capítulo

EJERCICIOS

▶ **3.35** Implementa un programa que dados dos números almacenados en dos posiciones de memoria A y B, almacene el valor absoluto de la resta de ambos en la posición de memoria RES. Es decir, si A es mayor que B deberá realizar la operación $A - B$ y almacenar el resultado en RES, y si B es mayor que A, entonces deberá almacenar $B - A$.

▶ **3.36** Implementa un programa que dado un vector, calcule el número de elementos de un vector que son menores a un número dado. Por ejemplo, si el vector es $[2, 4, 6, 3, 10, 12, 2]$ y el número es 5, el resultado esperado será 4, puesto que hay 4 elementos del vector menores a 5.

▶ **3.37** Implementa un programa que dada las notas de 2 exámenes parciales almacenadas en memoria (con notas entre 0 y 10), calcule la nota final (sumando las dos notas) y almacene en memoria la cadena de caracteres *APROBADO* si la nota final es mayor o igual a 10 y *SUSPENDIDO* si la nota final es inferior a 10.

▶ **3.38** Modifica el programa anterior para que almacene en memoria la cadena de caracteres *APROBADO* si la nota final es mayor o igual a 10 pero menor a 14, *NOTABLE* si la nota final es mayor o igual a 14 pero menor a 18 y *SOBRESALIENTE* si la nota final es igual o superior a 18.

▶ **3.39** Modifica el programa anterior para que si alguna de las notas de los exámenes parciales es inferior a 5, entonces se almacene *NOSUPERA* independientemente del valor del resto de exámenes parciales.

▶ **3.40** Implementa un programa que dado un vector, sume todos los elementos del mismo mayores a un valor dado. Por ejemplo, si el vector es $[2, 4, 6, 3, 10, 1, 4]$ y el valor 5, el resultado esperado será $6 + 10 = 16$

▶ **3.41** Implementa un programa que dado un vector, sume todos los elementos pares. Por ejemplo, si el vector es $[2, 7, 6, 3, 10]$, el resultado esperado será $2 + 6 + 10 = 18$.

▶ **3.42** Implementa un programa que calcule los N primeros números de la sucesión de Fibonacci. La sucesión comienza con los números 1 y 1 y a partir de estos, cada término es la suma de los dos anteriores. Es decir que el tercer elemento de la sucesión es $1 + 1 = 2$, el cuarto $1 + 2 = 3$, el quinto $2 + 3 = 5$ y así sucesivamente. Por ejemplo, los

$N = 10$ primeros son $[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]$. Para ello deberás usar una estructura de repetición adecuada y almacenar los valores obtenidos en memoria.

▶ **3.43** Modifica el programa anterior para que calcule elementos de la sucesión de Fibonacci hasta que el último elemento calculado sea mayor a un valor concreto.

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-

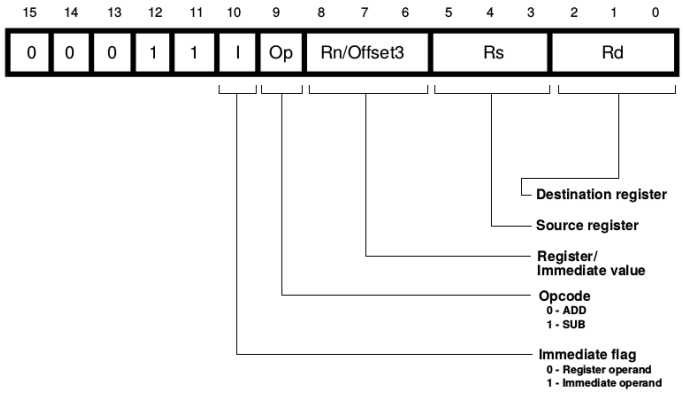


Figura 4.1: Formato de instrucción usado por las instrucciones «`add rd, rs, rn`» y «`sub rd, rs, rn`», entre otras

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.

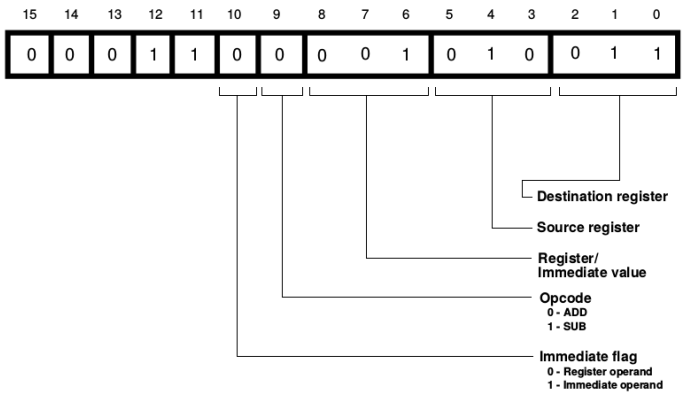


Figura 4.2: Codificación de la instrucción «`add r3, r2, r1`»

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`

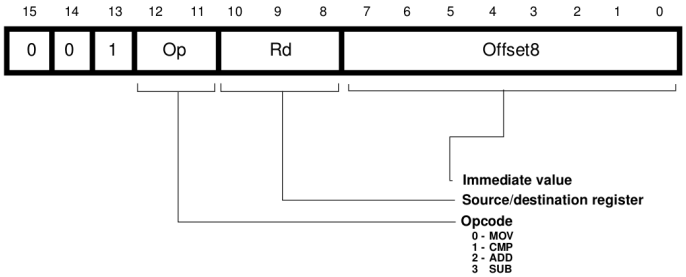


Figura 4.3: Formato de las instrucciones «`mov rd, #Offset8`», «`cmp rd, #Offset8`», «`add rd, #Offset8`» y «`sub rd, #Offset8`»

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).

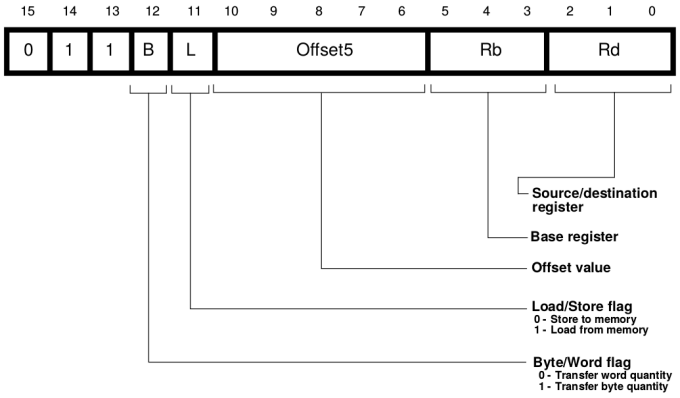


Figura 4.4: Formato de instrucción con direccionamiento relativo a registro con desplazamiento utilizado por las instrucciones «`ldr rd, [rb, #Offset5]`», «`str rd, [rb, #Offset5]`», «`ldrb rd, [rb, #Offset5]`» y «`strb rd, [rb, #Offset5]`»,

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?

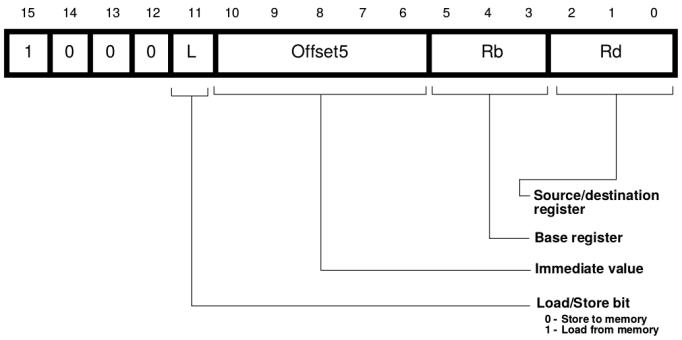


Figura 4.5: Formato de instrucción con direccionamiento relativo a registro con desplazamiento utilizado por las instrucciones «`ldrh rd, [rb, #0ffset5]`» y «`strh rd, [rb, #0ffset5]`»

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 `0ffset5` 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 `0ffset5` 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].

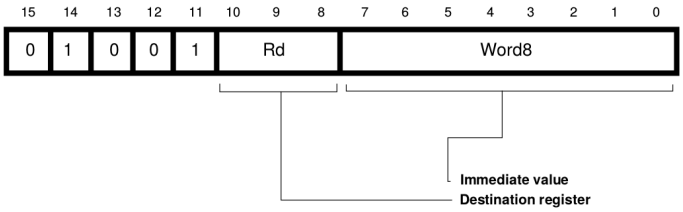


Figura 4.6: Formato de instrucción de carga relativa al contador de programa

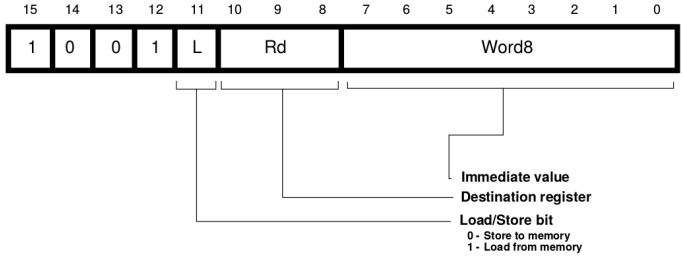


Figura 4.7: Formato de instrucción de carga y almacenamiento relativo al puntero de pila

. . . . . .

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

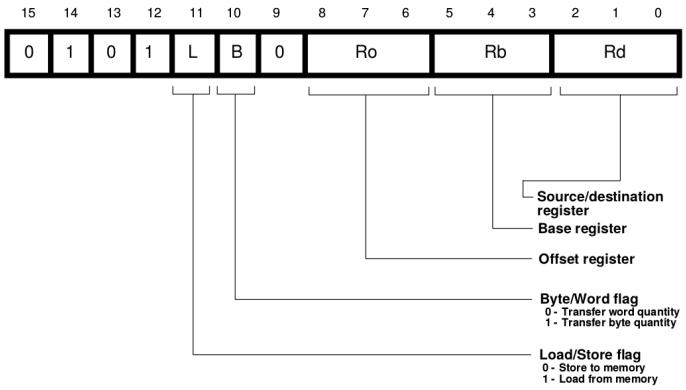


Figura 4.8: Formato de instrucción usado para codificar las instrucciones «**ldr**», «**ldrb**», «**str**» y «**strb**» que utilizan el modo de direccionamiento relativo a registro con registro de desplazamiento

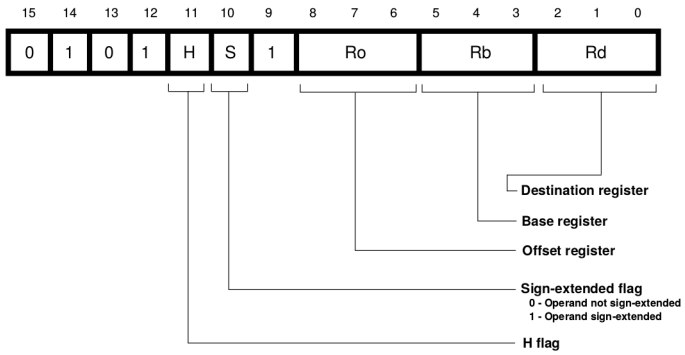


Figura 4.9: Formato de instrucción usado para codificar las instrucciones «**ldrh**», «**ldsh**», «**ldsb**» y «**strh**» que utilizan el modo de direccionamiento relativo a registro con registro de desplazamiento

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`.

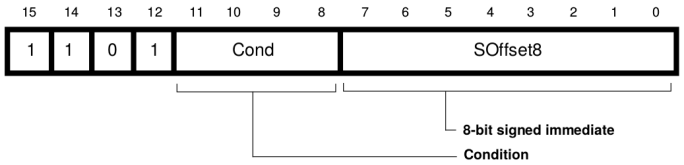


Figura 4.11: Formato de instrucción utilizado para codificar las instrucciones de salto condicional

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.

CAPÍTULO 5

Introducción a la gestión de subrutinas

Índice

Una subrutina es un trozo de código independiente, que normalmente realiza una tarea auxiliar completa, que se puede llamar y ejecutar desde cualquier parte de un programa y que, una vez ejecutado, devuelve el control al programa inmediatamente después de donde se había efectuado la llamada. Siendo habitual que al llamar una subrutina se le pasen parámetros para operar con ellos o para seleccionar distintos comportamientos, y que aquélla, al terminar, devuelva valores al programa que la llamó.

Conviene tener en cuenta que dependiendo del lenguaje de programación y de sus particularidades, una subrutina puede recibir cualquiera de los siguientes nombres: *rutina*, *procedimiento*, *función* o *método*. De hecho, es probable que ya hayas oído hablar de alguno de ellos. Por

ejemplo, en Python3 y en Java, en lugar de hablar de subrutinas se habla de funciones y métodos. Aunque en realidad sí que hay diferencias de significado entre algunos de los términos anteriores, cuando en este texto utilicemos el término subrutina, nos estaremos refiriendo indistintamente a cualquiera de los anteriores.

Para ver con más detalle en qué consiste una subrutina utilizaremos el siguiente programa en Python3, en el que se puede ver un ejemplo de subrutina, llamada «`multadd`».

```
1 def multadd(x, y, a):
2     res = x*y + a
3     return res
4
5 r1 = multadd(5, 3, 2)
6 r2 = multadd(2, 4, 1)
```

La sintaxis de Python3 para declarar el inicio de una subrutina es «`def nombre(param1, param2...):`». En dicha línea se da nombre a la subrutina y se especifica el nombre de cada uno de los parámetros que espera recibir dicha subrutina. En el ejemplo anterior, la subrutina se llama «`multadd`» y espera recibir tres parámetros, a los que nombra `x`, `y` y `z` (nombres que se utilizarán por la subrutina para identificar a los parámetros recibidos). El código que haya a continuación, mientras esté convenientemente indentado con respecto a la línea anterior, se considerará que es parte de la subrutina. Por último, en el caso de que la subrutina devuelva algún resultado, deberá haber al menos una línea con la sintaxis «`return value`». En el ejemplo anterior se puede ver que la subrutina devuelve el valor de la variable `res`.

Una vez que sabemos qué trozo del código corresponde a la subrutina «`multadd`», la siguiente pregunta es qué acciones lleva a cabo dicha subrutina, que son: I) realiza la operación $res \leftarrow x \cdot y + a$ con los parámetros x, y y a que recibe al ser llamada y, una vez completada la operación, II) devuelve el control al punto siguiente desde el que fue llamada, que puede acceder al resultado calculado.

Como se puede observar, la subrutina «`multadd`» es llamada dos veces desde el programa principal. La primera de ellas, los parámetros x, y y a toman los valores 5, 3 y 2, respectivamente. Mientras que la segunda vez, toman los valores 2, 4 y 1.

Cuando se ejecuta la instrucción «`r1 = multadd(5, 3, 2)`», el control se transfiere a la subrutina «`multadd`», que calcula $5 \cdot 3 + 2$ y devuelve el control al programa principal, que, a su vez, almacena el resultado, 17, en la variable `r1`.

A continuación comienza la ejecución de la siguiente instrucción del programa, «`r2 = multadd(2, 4, 1)`». y el control se transfiere de nuevo a la subrutina «`multadd`», quien ahora calcula $2 \cdot 4 + 1$ y devuelve de

nuevo el control al programa principal, pero en esta ocasión al punto del programa en el que se almacena el resultado, 9, en la variable `r2`.

Por tanto, y como se ha podido comprobar, «`multadd`» responde efectivamente a la definición dada de subrutina: es un trozo de código independiente, que realiza una tarea auxiliar completa, que se puede llamar y ejecutar desde cualquier parte de un programa y que, una vez ejecutado, devuelve el control al programa inmediatamente después de donde se había efectuado la llamada. A «`multadd`» se le pasan parámetros para operar con ellos, y, al terminar, devuelve un valor al programa que la llamó.

El uso de subrutinas presenta varias ventajas. La primera de ellas es que permite dividir un problema largo y complejo en subproblemas más sencillos. La ventaja de esta división radica en la mayor facilidad con la que se puede escribir, depurar y probar cada uno de los subproblemas por separado. Esto es, se puede desarrollar y probar una subrutina independientemente del resto del programa y posteriormente, una vez que se ha verificado que su comportamiento es el esperado, se puede integrar dicha subrutina en el programa que la va a utilizar.

Otra ventaja de programar utilizando subrutinas es que si una misma tarea se realiza en varios puntos del programa, no es necesario escribir el mismo código una y otra vez a lo largo del programa. Si no fuera posible utilizar subrutinas se debería repetir la misma fracción de código en todas y cada una de las partes del programa en las que éste fuera necesario. Es más, si en un momento dado se descubre un error en un trozo de código que se ha repetido en varias parte del programa, sería necesario revisar todas las partes del programa en las que éste aparece para rectificar en cada una de ellas el mismo error. De igual forma, cualquier mejora de dicha parte del código implicaría revisar todas las partes del programa en las que aparece. Por el contrario, si se utiliza una subrutina y se detecta un error o se quiere mejorar su implementación, basta con modificar su código; una sola vez.

En los párrafos anteriores se ha mostrado cómo la utilización de subrutinas permite reducir tanto el tiempo de desarrollo del código como el de su depuración. Sin embargo, el uso de subrutinas tiene una ventaja de mayor calado: subproblemas que aparecen con frecuencia en el desarrollo de ciertos programas pueden ser implementados como subrutinas y agruparse en bibliotecas (*libraries* en inglés[1]). Cuando un programador requiere resolver un determinado problema ya implementado, le basta con recurrir a una determinada biblioteca y llamar la subrutina ade-

[1]La palabra inglesa *library* se suele traducir erróneamente en castellano como «librería»; la traducción correcta es «biblioteca».

cuada. Es decir, gracias a la agrupación de subrutinas en bibliotecas, el mismo código puede ser reutilizado por muchos programas.

Desde el punto de vista que nos ocupa, el de los mecanismos que proporciona un procesador para soportar determinadas tareas de programación, el uso de subrutinas implica que se deben facilitar las siguientes acciones: llamar una subrutina; pasar los parámetros con los que debe operar la subrutina; devolver los resultados; y continuar la ejecución del programa a partir de la siguiente instrucción en código máquina a la que invocó a la subrutina. Es por ello que en este capítulo se presentan los aspectos más básicos relacionados con la gestión de subrutinas en el ensamblador Thumb de ARM. Es decir, cómo llamar y retornar de una subrutina y cómo intercambiar información entre el programa invocador y la subrutina utilizando registros.

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.8 «*Subroutine Call and Return*» 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.

5.1. Llamada y retorno de una subrutina

El juego de instrucciones ARM dispone de las siguientes instrucciones para gestionar la llamada y el retorno de una subrutina:

- «`bl etiqueta`» Se utiliza en el programa invocador para **llamar** la subrutina que comienza en la dirección de memoria indicada por la etiqueta. La ejecución de esta instrucción conlleva las siguientes acciones:
  - Se almacena la dirección de memoria de la siguiente instrucción a la que contiene la instrucción «`bl etiqueta`» en el registro `r14` (también llamado `lr`, por *link register*). Es decir, `lr`←`PC`+`4`[2].
  - Se lleva el control del flujo del programa a la dirección indicada en el campo «`etiqueta`». Es decir, se realiza un salto incondicional a la dirección especificada en «`etiqueta`» (`PC`←`etiqueta`).

[2]La instrucción «`bl etiqueta`» se codifica utilizando dos medias palabras, de ahí que al `PC` se le sume `4` para calcular la dirección de retorno.

- «**mov** pc, lr» Se utiliza en la subrutina para **retornar** al programa invocador. Esta instrucción actualiza el contador de programa con el valor del registro `lr`, lo que a efectos reales implica realizar un salto incondicional a la dirección contenida en el registro `lr`. Es decir, `PC←lr`.

La correcta utilización de estas dos instrucciones permite realizar de forma sencilla la llamada y el retorno desde una subrutina. En primer lugar, el programa invocador debe llamar la subrutina utilizando la instrucción «**bl** `etiqueta`». Esta instrucción almacena en `lr` la dirección de vuelta y salta a la dirección indicada por la etiqueta. Por último, cuando finalice la subrutina, para retornar al programa que la llamó, ésta debe ejecutar la instrucción «**mov** `pc, lr`».

Esta forma de proceder será la correcta siempre que el contenido del registro `lr` no se modificado durante la ejecución de la subrutina. Este funcionamiento queda esquematizado en la Figura 5.1.

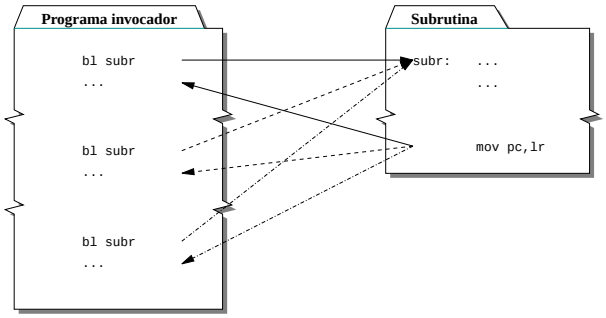


Figura 5.1: Llamada y retorno de una subrutina

Como primer ejemplo de subrutinas en ensamblador de ARM, el siguiente código muestra un programa que utiliza una subrutina llamada «`suma`». Esta subrutina suma los valores almacenados en los registros `r0` y `r1`, y devuelve la suma de ambos en el registro `r2`. Como se puede observar, la subrutina se llama desde dos puntos del programa principal (la primera línea del programa principal se ha etiquetado con «`main`»).

introsub-suma-valor.s

```
1          .data
2  datos:  .word 5, 8, 3, 4
3  suma1:  .space 4
4  suma2:  .space 4
5          .text
6
7  @ Programa invocador
8
9  main:   ldr r4, =datos
```

```
10
11          ldr r0, [r4]
12          ldr r1, [r4, #4]
13  primera: bl suma
14          ldr r5, =suma1
15          str r2, [r5]
16
17          ldr r0, [r4, #8]
18          ldr r1, [r4, #12]
19  segunda: bl suma
20          ldr r5, =suma2
21          str r2, [r5]
22
23  stop:   wfi
24
25  @ Subrutina
26
27  suma:   add r2, r1, r0
28          mov pc, lr
29
30          .end
```

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

EJERCICIOS

▶ **5.1** ¿Cuál es el contenido del `PC` y del registro `lr` antes y después de ejecutar la instrucción «`bl suma`» etiquetada como «`primera`»?

▶ **5.2** ¿Cuál es el contenido de los registros `PC` y `lr` antes y después de ejecutar la instrucción «`mov pc, lr`» la primera vez que se ejecuta la subrutina «`suma`»?

▶ **5.3** ¿Cuál es el contenido del `PC` y del registro `lr` antes y después de ejecutar la instrucción «`bl suma`» etiquetada como «`segunda`»?

▶ **5.4** ¿Cuál es el contenido de los registros `PC` y `lr` antes y después de ejecutar la instrucción «`mov pc, lr`» la segunda vez que se ejecuta la subrutina «`suma`»?

▶ **5.5** Anota el contenido de las variables «`suma1`» y «`suma2`» después de ejecutar el programa anterior.

▶ **5.6** Crea un nuevo programa a partir del anterior en el que la subrutina «`suma`» devuelva en `r2` el doble de la suma de `r1` y `r0`. Ejecuta el programa y anota el contenido de las variables «`suma1`» y «`suma2`».

5.2. Paso de parámetros

Se denomina *paso de parámetros* al mecanismo mediante el cual el programa invocador y la subrutina intercambian datos.

Los parámetros intercambiados entre el programa invocador y la subrutina pueden ser de tres tipos según la dirección en la que se transmita la información: de *entrada*, de *salida* o de *entrada/salida*. Se denominan parámetros de entrada a los que proporcionan información del programa invocador a la subrutina. Se denominan parámetros de salida a los que devuelven información de la subrutina al programa invocador. Por último, los parámetros de *entrada/salida* proporcionan información del programa invocador a la subrutina y devuelven información de la subrutina al programa invocador.

Por otro lado, para realizar el paso de parámetros es necesario disponer de algún lugar físico donde se pueda almacenar y leer la información que se quiere transferir. Las dos opciones más comunes son: utilizar registros o utilizar la pila. Que se utilicen registros o la pila depende de la arquitectura en cuestión y del convenio que se siga para el paso de parámetros en dicha arquitectura.

Este apartado se va a ocupar únicamente del paso de parámetros por medio de registros. De hecho, la arquitectura ARM establece un convenio para el paso de parámetros mediante registros: para los parámetros de entrada y de salida se deben utilizar los registros `r0`, `r1`, `r2` y `r3`.

Hasta aquí se ha visto que hay parámetros de entrada, de salida y de entrada/salida. Además, también se ha visto que los parámetros pueden pasarse por medio de registros o de la pila. El último aspecto a considerar del paso de parámetros es cómo se transfiere cada uno de los parámetros. Hay dos formas de hacerlo: un parámetro puede pasarse por valor o por referencia. Se dice que un parámetro se pasa por valor cuando lo que se transfiere es el dato en sí. Un parámetro se pasa por referencia cuando lo que se transfiere es la dirección de memoria en la que se encuentra dicho dato.

Según todo lo anterior, el desarrollo de una subrutina implica determinar en primer lugar:

- El número de parámetros necesarios.
- Cuáles son de entrada, cuáles de salida y cuáles de entrada/salida.
- Si se van a utilizar registros o la pila para su transferencia.
- Qué parámetros deben pasarse por valor y qué parámetros por referencia.

Naturalmente, el programa invocador deberá ajustarse a los requerimientos que haya fijado el desarrollador de la subrutina en cuanto a cómo se debe realizar el paso de parámetros.

En los siguientes apartados se utilizarán parámetros de entrada, salida o entrada/salida según sea necesario. Además, el paso de parámetros se hará utilizando los registros fijados por el convenio ARM. El primero de los siguientes apartados muestra cómo se realiza el paso de parámetros por valor y el segundo, cómo se realiza el paso de parámetros por referencia.

5.2.1. Paso de parámetros por valor

El paso de parámetros por valor implica la siguiente secuencia de acciones (ver Figura 5.2):

1. Antes de realizar la llamada a la subrutina, el programa invocador carga el valor de los parámetros de entrada en los registros correspondientes.

2. La subrutina, finalizadas las operaciones que deba realizar y antes de devolver el control al programa invocador, carga el valor de los parámetros de salida en los registros correspondientes.

3. El programa invocador recoge los parámetros de salida de los registros correspondientes.

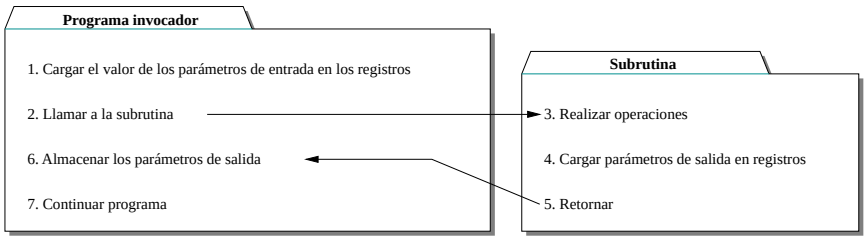


Figura 5.2: Paso de parámetros por valor

Como ejemplo de paso de parámetros por valor se presenta el siguiente código que ya se había visto con anterioridad. Éste muestra un ejemplo de paso de parámetros de entrada y de salida por valor.

introsub-suma-valor.s

```
1           .data
2 datos:    .word 5, 8, 3, 4
3 suma1:    .space 4
4 suma2:    .space 4
5           .text
```

```
6
7  @ Programa invocador
8
9  main:    ldr r4, =datos
10
11         ldr r0, [r4]
12         ldr r1, [r4, #4]
13 primera: bl suma
14         ldr r5, =suma1
15         str r2, [r5]
16
17         ldr r0, [r4, #8]
18         ldr r1, [r4, #12]
19 segunda: bl suma
20         ldr r5, =suma2
21         str r2, [r5]
22
23 stop:    wfi
24
25 @ Subrutina
26
27 suma:    add r2, r1, r0
28         mov pc, lr
29
30         .end
```

EJERCICIOS

▶ **5.7** Enumera los registros que se han utilizado para pasar los parámetros a la subrutina.

▶ **5.8** ¿Los anteriores registros se han utilizado para pasar parámetros de entrada o de salida?

▶ **5.9** Anota qué registro se ha utilizado para devolver el resultado al programa invocador.

▶ **5.10** ¿El anterior registro se ha utilizado para pasar un parámetro de entrada o de salida?

▶ **5.11** Señala qué instrucciones se corresponden a cada uno de las acciones enumeradas en la Figura 5.2. (Hazlo únicamente para la primera de las dos llamadas.)

5.2.2. Paso de parámetros por referencia

En cuanto al paso de parámetros por referencia, éste implica la siguiente secuencia de acciones (ver Figura 5.3):

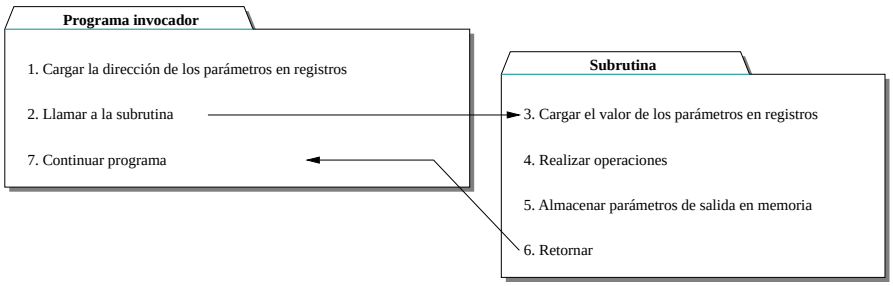


Figura 5.3: Paso de parámetros por referencia

- Antes de realizar la llamada a la subrutina, el programa invocador carga en los registros correspondientes, las direcciones de memoria en las que está almacenada la información que se quiere pasar.
- La subrutina carga en registros el contenido de las direcciones de memoria indicadas por los parámetros de entrada y opera con ellos (recuerda que ARM no puede operar directamente con datos en memoria).
- La subrutina, una vez ha finalizado y antes de devolver el control al programa principal, almacena los resultados en las direcciones de memoria proporcionadas por el programa invocador.

El siguiente programa muestra un ejemplo en el que se llama a una subrutina utilizando el paso de parámetros por referencia tanto para los parámetros de entrada como los de salida.

introsub-suma-referencia.s

```
1          .data
2  datos:  .word 5, 8, 3, 4
3  suma1:  .space 4
4  suma2:  .space 4
5          .text
6
7  @ Programa invocador
8
9  main:   ldr r0, =datos
10         ldr r1, =datos + 4
11         ldr r2, =suma1
12 primera: bl suma
13
14         ldr r0, =datos + 8
15         ldr r1, =datos + 12
16         ldr r2, =suma2
17 segunda: bl suma
18
```

```
19 stop:    wfi
20
21 @ Subrutina
22
23 suma:    ldr r4, [r0]
24          ldr r5, [r1]
25          add r6, r4, r5
26          str r6, [r2]
27          mov pc, lr
28
29          .end
```

EJERCICIOS

▶ **5.12** Enumera los registros que se han utilizado para pasar la dirección de los parámetros de entrada a la subrutina.

▶ **5.13** Anota qué registro se ha utilizado para pasar la dirección del parámetro de salida a la subrutina.

▶ **5.14** Señala qué instrucciones se corresponden con cada una de las acciones enumeradas en la Figura 5.3. (Hazlo únicamente para la primera de las dos llamadas.)

5.2.3. Paso de parámetros, ¿por valor o por referencia?

Una vez descritas las dos formas de paso de parámetros a una subrutina, queda la tarea de decidir cuál de las formas, por referencia o por valor, es más conveniente en cada caso. Los ejemplos anteriores eran un poco artificiales ya que todos los parámetros se pasaban o bien por valor o por referencia. En la práctica, se debe analizar para cada parámetro cuál es la forma idónea de realizar el paso. Es decir, generalmente, no todos los parámetros de una subrutina utilizarán la misma forma de paso.

De hecho, la decisión última de cómo se pasan los parámetros a una subrutina depende en gran medida de la estructura de datos que se quiera pasar. Si se trata de un dato de tipo estructurado (vectores, matrices, cadenas, estructuras...), el paso siempre se hará por referencia. Tanto si el parámetro en cuestión es de entrada, de salida o de entrada/salida.

En cambio, si el dato que se quiere pasar es de tipo escalar (un número entero, un número real, un carácter...), éste se puede pasar por valor o por referencia. Esta última decisión depende de la utilización que se le vaya a dar al parámetro: es decir, si se trata de un parámetro de entrada, de salida o de entrada/salida. La siguiente lista muestra las opciones disponibles según el tipo de parámetro:

- Parámetro de entrada. Un parámetro de este tipo es utilizado por la subrutina pero no debería ser modificado: es preferible pasar este tipo de parámetros por valor.
- Parámetro de salida. Un parámetro de este tipo permite a la subrutina devolver el resultado de las operaciones realizadas. Para este tipo de parámetros se puede optar por cualquiera de las opciones: que el parámetro sea devuelto por valor o por referencia.
  - Por valor: la subrutina devuelve el dato utilizando el registro reservado para ello.
  - Por referencia: la subrutina almacena en memoria el dato. La dirección de memoria la obtiene la subrutina del registro que había sido utilizado para ello.
- Parámetro de entrada/salida. Un parámetro de este tipo proporciona un valor que la subrutina necesita conocer pero en el que posteriormente devolverá el resultado. En este caso, es preferible pasarlo por referencia.

5.2.4. Un ejemplo más elaborado

A continuación se plantea el desarrollo de un programa en lenguaje ensamblador donde se aplican todos los conceptos presentados hasta ahora en este capítulo.

Dicho programa tiene por objeto calcular cuántos elementos de un vector dado tienen el valor `12`. El programa consta de una subrutina que devuelve el número de elementos de un vector que son menores a un número dado. Llamando dos veces a dicha subrutina con los valores `13` y `12` y realizando una simple resta, el programa será capaz de determinar el número de elementos que son iguales a `12`.

Se debe desarrollar dicho programa paso a paso. En primer lugar, se debe desarrollar una subrutina que contabilice cuántos elementos de un vector son menores a un valor dado. Para ello, hay que determinar qué parámetros debe recibir dicha subrutina, así como qué registros se van a utilizar para ello, y cuáles se pasarán por referencia y cuáles por valor.

Una vez desarrollada la subrutina y teniendo en cuenta los parámetros que requiere, se deberá desarrollar el programa que llame a dicha subrutina y obtenga el número de elementos del vector dado que son iguales a `12`.

Una descripción detallada del funcionamiento que debiera tener dicho programa se muestra en el siguiente listado en lenguaje Python3:

```
1 def nummenorque(vector_s, dim_s, dato_s):
2       n = 0
3       for i in range(dim_s):
```

```
4            if vector_s[i] < dato_s:
5               n = n + 1;
6       return n
7
8  vector = [5, 3, 5, 5, 8, 12, 12, 15, 12]
9  dim = 9
10 num = 12
11 res1 = nummenorque(vector, dim, num + 1)
12 res2 = nummenorque(vector, dim, num)
13 res = res1 - res2
```

EJERCICIOS

▶ **5.15** Antes de desarrollar el código de la subrutina, contesta las siguientes preguntas:

a) ¿Qué parámetros debe pasar el programa invocador a la subrutina? ¿Y la subrutina al programa invocador?

b) ¿Cuáles son de entrada y cuáles de salida?

c) ¿Cuáles se pasan por referencia y cuáles por valor?

d) ¿Qué registros se van a utilizar para cada uno de ellos?

▶ **5.16** Completa el desarrollo del fragmento de código correspondiente a la subrutina.

▶ **5.17** Desarrolla el fragmento de código correspondiente al programa invocador.

▶ **5.18** Comprueba que el programa escrito funciona correctamente. ¿Qué valor se almacena en «`res`» cuando se ejecuta? Modifica el contenido del vector «`vector`», ejecuta de nuevo el programa y comprueba que el resultado obtenido es correcto.

5.3. Problemas del capítulo

EJERCICIOS

▶ **5.19** Realiza el siguiente ejercicio:

a) Desarrolla una subrutina que calcule cuántos elementos de un vector de enteros son pares (múltiplos de 2). La subrutina debe recibir como parámetros el vector y su dimensión y devolver el número de elementos pares.

b) Implementa un programa en el que se inicialicen dos vectores de 10 elementos cada uno, «`vector1`» y «`vector2`», y que almacene en sendas variables, «`numpares1`» y «`numpares2`», el número de elementos pares de cada uno de ellos. Naturalmente, el programa deberá utilizar la subrutina desarrollada en el apartado a) de este ejercicio.

▶ **5.20** Realiza el siguiente ejercicio:

a) Implementa una subrutina que calcule la longitud de una cadena de caracteres que finalice con el carácter nulo.

b) Implementa un programa en el que se inicialicen dos cadenas de caracteres y calcule cuál de las dos cadenas es más larga. Utiliza la subrutina desarrollada en el apartado a) de este ejercicio.

▶ **5.21** Realiza el siguiente ejercicio:

a) Desarrolla una subrutina que sume los elementos de un vector de enteros de cualquier dimensión.

b) Desarrolla un programa que sume todos los elementos de una matriz de dimensión $m \times n$. Utiliza la subrutina desarrollada en el apartado a) de este ejercicio para sumar los elementos de cada fila de la matriz.

   En la versión que se implemente de este programa utiliza una matriz con $m = 5$ y $n = 3$, es decir, de dimensión 5×3 con valores aleatorios (los que se te vayan ocurriendo sobre la marcha). Se debe tener en cuenta que la matriz debería poder tener cualquier dimensión, así que se deberá utilizar un bucle para recorrer sus filas.

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:

- 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`.

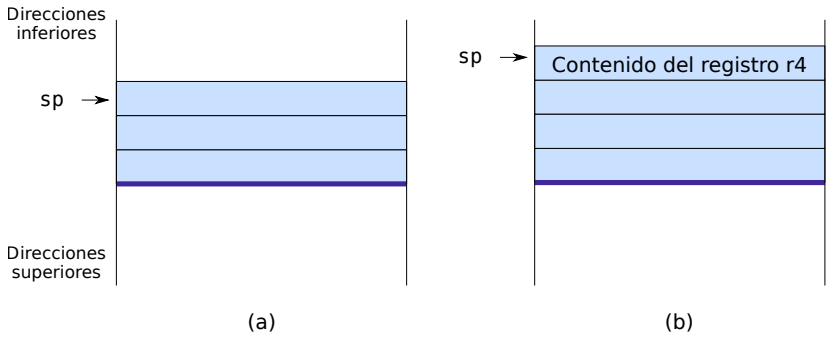


Figura 6.1: La pila (a) antes y (b) después de apilar el registro `r4`

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.

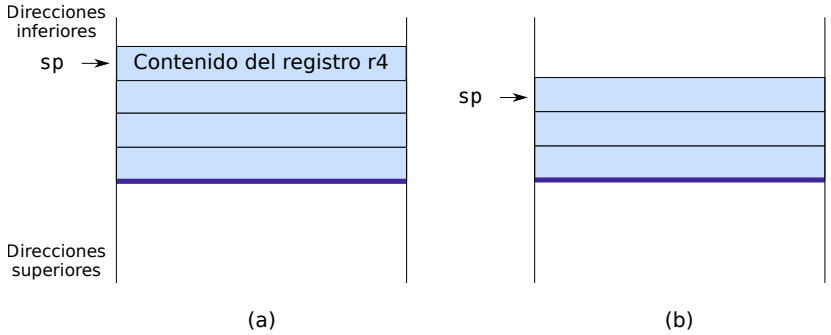


Figura 6.2: La pila (a) antes y (b) después de desapilar el registro `r4`

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.

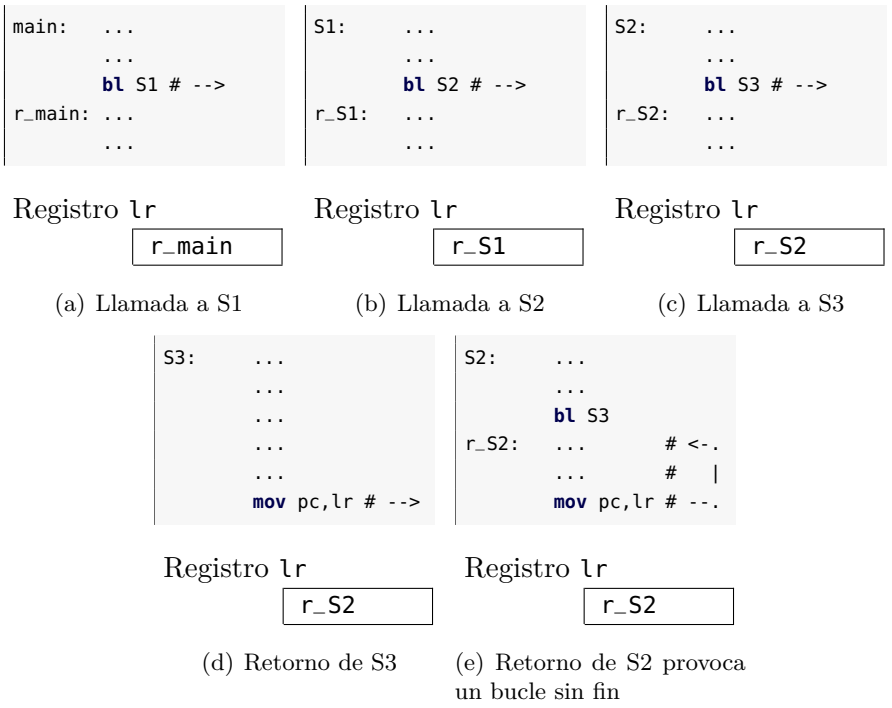
Dicha estructura de datos debe satisfacer dos requisitos. En primer lugar, debe ser capaz de permitir recuperar las direcciones de retorno en orden inverso a su almacenamiento (ya que es el orden en el que se producen los retornos). En segundo lugar, el espacio reservado para este cometido debe poder 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

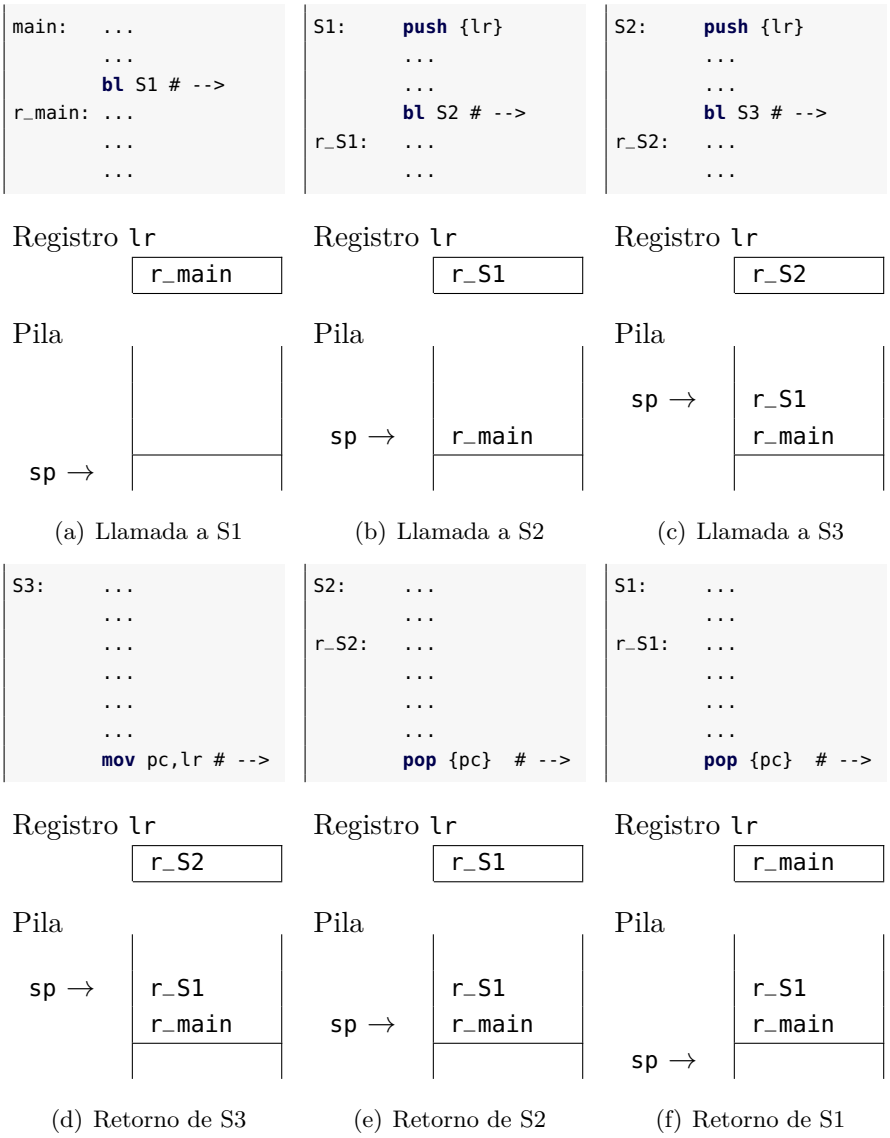
(d) Retorno de S3 (e) Retorno de S2 provoca un bucle sin fin

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

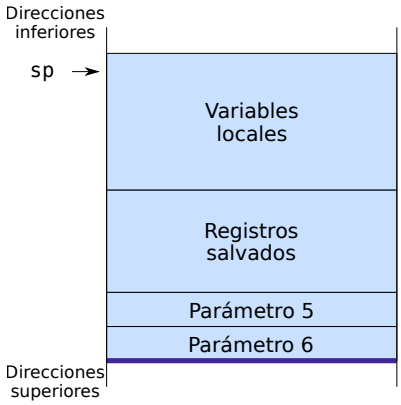


Figura 6.5: Esquema del bloque de activación

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.

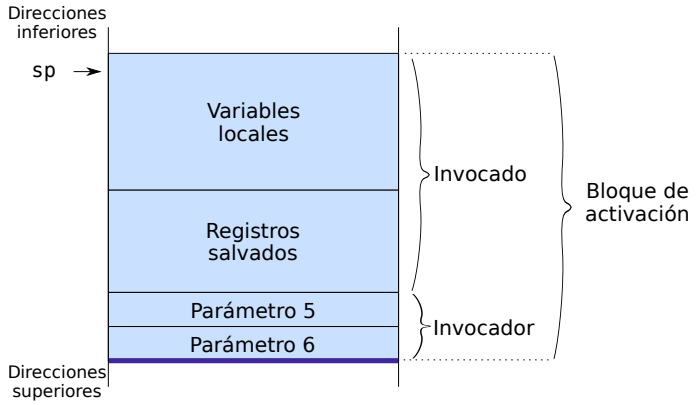


Figura 6.6: Estado de la pila después de una llamada a subrutina

```
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`».

. . . . . .

CAPÍTULO 7

Entrada/Salida: introducción

Índice

La entrada/salida es el componente de un ordenador que se encarga de permitir su interacción con el mundo exterior. Si un ordenador no dispusiera de entrada/salida, sería totalmente inútil, con independencia de la potencia de su procesador y la cantidad de memoria, pues no podría realizar ninguna tarea que debiera manifestarse fuera de sus circuitos electrónicos. De la misma forma que se justifica su necesidad, se explica la variedad de la entrada/salida y de ahí su problemática: el mundo exterior, lejos de ser de la misma naturaleza electrónica que los circuitos del ordenador, se caracteriza por su variedad y su cambio. La entrada/salida debe ser capaz de relacionarse con este mundo diverso y, a la vez, con los dispositivos electrónicos del ordenador.

Los siguientes apartados explican cómo puede gestionarse esta relación haciendo que la entrada/salida sea a la vez, versátil, eficaz y manejable.

7.1. Generalidades y problemática de la entrada/salida

La primera imagen que se nos viene a la cabeza al pensar en un sistema informático es el ordenador personal, con un teclado y un ratón para interactuar con él y un monitor para recibir las respuestas de forma visual. Posiblemente tengamos en el mismo equipo unos altavoces para reproducir audio, una impresora para generar copias de nuestros trabajos, un disco duro externo y, por supuesto, una conexión a internet -aunque pueda no verse al no utilizar cables-. Todos estos elementos enumerados, aunque sean de naturaleza y función completamente distinta, se consideran dispositivos periféricos del ordenador y son una parte -en algunos casos la más alejada del ordenador, como los altavoces- de su entrada/salida.

Considerando la dirección, siempre referida al ordenador, en que fluyen los datos, vemos que unos son de salida, como la impresora o el monitor, otros de entrada, como el teclado y el ratón, mientras que algunos sirven como de entrada y de salida, como el disco duro y la conexión de red. La dirección de los datos se denomina *comportamiento* y es una característica propia de cada dispositivo. Podemos ver además que los dos últimos dispositivos del párrafo anterior no sirven para comunicarse con el usuario -un ser humano- a diferencia del teclado, ratón o monitor. Esto nos permite identificar otra característica de los dispositivos, que es su interlocutor, entendido como el ente que recibe o genera los datos que el dispositivo comunica con el ordenador. Entre los ejemplos anteriores es evidente determinar cuáles tienen un interlocutor humano. En el caso de la conexión de red, el interlocutor, a través de numerosos dispositivos intermedios -que normalmente también son pequeños ordenadores- acaba siendo otro ordenador personal o un servidor. En este ejemplo el interlocutor es una máquina, como ocurre también con otros muchos ordenadores presentes en sistemas empotrados que se comunican con controladores de motores, sistemas de regulación de iluminación u otra maquinaria generalmente electrónica o eléctrica. Pero hoy en día que los ordenadores están extendidos en todos los campos de la actividad humana, podemos tener uno regulando la temperatura de una caldera de vapor -con sensores midiendo la temperatura del aire en su interior-, midiendo la humedad del terreno en un campo de cultivo o el nivel de concentración de cierto soluto en una reacción química. En estos últimos ejemplos el interlocutor no es un ser humano ni una má-

quina, sino un sistema o fenómeno natural -dado que la entrada/salida comunica el ordenador con el mundo exterior-. Esta clasificación de interlocutores no pretende ser un dogma ni está exenta de consideraciones filosóficas. Según ella es evidente que una interfaz de un computador con las terminaciones nerviosas de un ratón en un experimento de bioingeniería no tiene interlocutor humano, pero ¿qué diríamos entonces si las terminaciones nerviosas fueran de una persona?

En el párrafo anterior hemos vuelto a comentar que la entrada/salida pone en contacto el ordenador con el mundo exterior. Esta afirmación no parece confirmarse cuando hablamos de un disco duro. Obviando el ejemplo del que hemos partido, el caso del disco externo, un disco duro es interno al ordenador y parte imprescindible de él, al menos para los ordenadores personales. Sin embargo, dado que el disco duro magnético basa su funcionamiento en piezas mecánicas en movimiento, participa de todas las demás características de los dispositivos de entrada/salida y por ello se considera como tal. Y una de estas características, especialmente importante en los discos duros, es la tasa de trasferencia de datos, es decir, la cantidad de datos por unidad de tiempo que intercambian ordenador y dispositivo. Esta tasa de transferencia influye mucho en la forma de gestionar la entrada/salida, adaptando a ella la forma de tratar cada dispositivo. Un teclado puede comunicar unos pocos bytes por segundo; un disco duro puede alcanzar varios gigabytes por segundo. Aunque todos los periféricos son lentos en relación con la velocidad del procesador -que puede tratar decenas de miles de millones de bytes por segundo- la propia diferencia de velocidades entre dispositivos requiere tratamientos bien diferenciados.

Además, la tasa de transferencia, considerada sin más, no describe correctamente el comportamiento temporal de los dispositivos, pudiendo ser medida de distintas formas, todas ellas correctas aunque no igualmente significativas. Veamos un par de ejemplos que permiten caracterizar mejor la tasa de transferencia. Comparando la reproducción de una película en alta definición almacenada en un disco duro con la pulsación de un teclado, es evidente que la primera actividad requiere mayor tasa de transferencia. Sin embargo, nuestra experiencia al disfrutar de la película no se verá mermada si, desde que ejecutamos el programa de reproducción hasta que aparecen las primeras imágenes transcurren diez o quince segundos. Sería imposible, por otra parte, trabajar con un ordenador si cada vez que pulsamos una tecla transcurrieran varios segundos -no ya diez, simplemente uno o dos- hasta que dicha pulsación se hace evidente en la respuesta del sistema. Estos ejemplos revelan los dos factores que caracterizan el comportamiento temporal de los dispositivos de entrada/salida:

- La *latencia*, que se entiende como el tiempo transcurrido desde

que se inicia una operación de entrada/salida hasta que el primer dato comunicado llega a su destino. En una operación de entrada sería el tiempo transcurrido desde que se inicia la petición de datos hasta que se recibe el primero de ellos. Un teclado, con una baja tasa de transferencia, requiere sin embargo una latencia de decenas de milisegundos para funcionar adecuadamente.

- La *productividad*, que se refiere a la cantidad de datos transferidos por unidad de tiempo, y que coincide con la primera definición que hemos dado de *tasa de transferencia*.

Al indicar un valor para la productividad se debe especificar adecuadamente cómo se ha realizado el cálculo. Teniendo en cuenta estas dos definiciones, la medida correcta de la productividad de una transacción debería incluir el tiempo de latencia, aunque durante él no se transmitan datos. En este caso la productividad vendría dada por la cantidad de datos recibidos dividida entre el tiempo transcurrido desde que se inició la transacción hasta que concluyó la recepción de datos. Si lo que se analiza es un dispositivo y no una transacción en particular, lo más ecuánime es dar una productividad media, teniendo en cuenta los tiempos de latencia y de transacción. Muchas veces se da, sin embargo, sobre todo en información comercial -orientada a demostrar correcta o incorrectamente las bondades de cierto producto-, la productividad máxima, que no tiene en cuenta el tiempo de latencia y considera el mejor caso posible para el funcionamiento del dispositivo.

En este apartado hemos presentado algunas generalidades de los dispositivos y sistemas de entrada/salida y hemos presentado tres propiedades que ayudan a su clasificación: su *comportamiento*, el *interlocutor* al que se aplican y la *tasa de transferencia*, matizada con los conceptos de *latencia* y *productividad*. Los sistemas de entrada/salida actuales son elevadamente complejos e incluso, por decirlo de alguna manera, jerárquicos. Estamos acostumbrados a utilizar dispositivos periféricos USB como los que hemos estado comentando -teclados, ratones, impresoras, etcétera-. Pues bien, el bus de entrada/salida USB -al igual que los SPI, I^2C y CAN, utilizados en sistemas empotrados- es a su vez un dispositivo periférico del ordenador, y debe ser tratado como tal. Una tarjeta de sonido conectada al bus PCI Express de un PC es un dispositivo periférico conectado directamente al sistema; una tarjeta igual -en su mayor parte- conectada a un bus USB es un dispositivo periférico conectado a un dispositivo de entrada/salida conectado al sistema. El tratamiento de ambas es idéntico en muchos aspectos, pero diferente en otros. Afortunadamente los sistemas operativos, a través de sus manejadores de dispositivos, estructurados de forma modular y jerárquica, son capaces de gestionar eficazmente esta complejidad. En este texto, en los siguien-

tes apartados, nos limitaremos a presentar los conceptos básicos de la estructura y la gestión de la entrada/salida, desde el punto de vista de la estructura de los computadores.

7.2. Estructura de los sistemas y dispositivos de entrada/salida

La función de la entrada/salida, como sabemos, es comunicar el ordenador con el mundo exterior. Si a esta afirmación unimos lo tratado en el apartado anterior, especialmente al hablar de los diferentes interlocutores de los dispositivos de entrada/salida, y la propia experiencia acerca de los incontables usos de los ordenadores en la vida actual, es fácil intuir que la estructura física de los elementos de entrada/salida es complicada e incluye diversas tecnologías. Efectivamente, todo dispositivo acaba relacionándose con el ordenador, por lo que dispone de circuitos electrónicos digitales de la misma tecnología. En el otro extremo, el dispositivo es capaz de generar luz, mover una rueda, medir la salinidad del agua o registrar los desplazamientos producidos en una palanca. Buena parte de esta estructura, pero no toda, es electrónica. Es sin embargo posible encontrar una estructura general a la que, como siempre con excepciones, se adaptan de una u otra forma todos los dispositivos de entrada/salida. Esta configuración incluye tres tipos de tecnología, que enumeradas desde el ordenador hacia el exterior serían las siguientes:

- Una parte formada por *circuitos electrónicos digitales* que comunica el dispositivo con el ordenador. Es la parte más genérica, propia del sistema informático y no de los diversos elementos del mundo exterior. Esta parte incluye todo lo necesario para la gestión de la entrada/salida en el ordenador, que iremos describiendo a lo largo de este documento.

- Una parte compuesta por *circuitos electrónicos analógicos*, que suele terminar en uno o varios componentes llamados *transductores* que transforman energía eléctrica en otro tipo de energía, o viceversa. Esta parte se encarga de la adaptación de los niveles eléctricos necesarios para comunicarse con el transductor, y de posibles tratamientos electrónicos de las señales -filtrados, amplificación, etcétera-.

- Una parte con componentes de una o varias *tecnologías no eléctricas* que comienza con los transductores y los adapta, en el ámbito de las magnitudes y características físicas propias del dispositivo.

En un ejemplo tan sencillo como un LED utilizado como dispositivo de salida, tenemos que la parte no eléctrica la constituye el propio encapsulado del diodo, con su color -o capacidad de difusión de la luz para un LED RGB- y su efecto de lente. Como vemos, ambas características son ópticas. La parte eléctrica estaría formada por el propio diodo semiconductor, que es en este caso el transductor, y la resistencia de polarización. La electrónica digital se encontraría en los circuitos de salida de propósito general -GPIO, como veremos más adelante- del microcontrolador al que conectamos el LED.

En el caso de un teclado comenzaríamos con las partes mecánicas de las teclas -incluyendo resortes y membranas, según el tipo- y los contactos eléctricos que completan los transductores. La electrónica analógica estaría formada por resistencias para adaptar los niveles eléctricos y diodos para evitar corrientes inversas. La parte digital, en los teclados más corrientes, la forma un microcontrolador que gestiona el teclado y encapsula la información de las teclas pulsadas en un formato estandarizado que se envía a través de un bus de entrada/salida estándar -USB hoy en día; antes PS/2 o el bus de teclado de los primeros PC-.

Si bien las tres partes mencionadas son necesarias en el funcionamiento del dispositivo, la gestión de la entrada/salida desde el ordenador requiere solo la primera, implementada con electrónica digital y compatible por tanto con el funcionamiento del resto de componentes del computador. Esto permite además que la estructura a grandes rasgos de este bloque sea común a todos los dispositivos y su funcionamiento, el propio de los circuitos digitales. Por ello es posible generalizar la gestión de la entrada/salida, al igual que los comportamientos del procesador y las memorias se adaptan a unas líneas generales bien conocidas. Vamos a ver ahora con detalle la estructura genérica, a nivel lógico o funcional -no a nivel estructural o de componentes físicos- de la parte digital de los dispositivos de entrada/salida. En apartados posteriores describiremos las distintas formas de gestionarla.

La parte de los dispositivos de entrada/salida común a la tecnología electrónica digital del ordenador y que permite relacionarlo con uno o varios periféricos recibe el nombre de *controlador de entrada/salida*. El controlador oculta al procesador las especificidades y la dificultad de tratar con el resto de componentes del periférico y le proporciona una forma de intercambio de información, una interfaz, genérica. Esta generalidad, como se ha dicho, tiene rasgos comunes para todos los tipos de dispositivos pero además tiene rasgos específicos para cada tipo que dependen de sus características. Por ejemplo, un controlador de disco duro se adapta a una especificación común para los controladores -el estándar IDE-ATA, por ejemplo- con independencia de la tecnología de fabricación y aspectos específicos de un modelo de disco en concreto, y estandariza la forma de tratar el disco duro mediante los programas

ejecutados por el procesador.

Para realizar la comunicación entre el procesador y el dispositivo, a través del controlador, existen un conjunto de espacios de almacenamiento, normalmente registros -también conocidos como *puertos*- a los que puede acceder el procesador que se clasifican, según su función, en tres tipos:

- *Registros de control*, que se utilizan para que el procesador configure parámetros en el dispositivo o le indique las operaciones de entrada/salida que debe realizar. Son registros en los que puede escribir el procesador, pero no el dispositivo.
- *Registros de estado*, que permiten al dispositivo mantener información acerca de su estado y del estado de las operaciones de entrada/salida que va realizando. Son registros que escribe el dispositivo y puede leer el procesador.
- *Registros de datos*, que sirven para realizar el intercambio de datos entre el procesador y el dispositivo en las operaciones de entrada/-salida. En el caso de salida, el procesador escribirá los datos que el periférico se encargará de llevar al mundo exterior. En el caso de entrada, el periférico escribirá los datos en estos registros, que de este modo serán accesibles para el procesador mediante lecturas.

Veamos un ejemplo de uso de estos registros a la hora de que el procesador se comunique con una impresora, a través de su controlador, para imprimir cierto documento. Aunque en realidad las cosas no sucedan exactamente de esta manera, por la estandarización de los formatos de documentos y gestión de impresoras, el ejemplo es suficientemente ilustrativo y válido. En primer lugar, el procesador configuraría en la impresora, a través de registros de control, el tamaño de papel, la resolución de la impresión y el uso o no de colores. Una vez realizada la configuración, el procesador iría enviando los datos a imprimir a través de los registros de datos, y al mismo tiempo estaría consultando los registros de estado, ya sea para detectar posibles errores -falta de papel o de tinta, atascos de papel-, ya sea para saber cuándo la impresora no acepta más datos -recordemos que el procesador es mucho más rápido- o ha terminado de imprimir la página en curso. Al acabar todas las páginas del documento, el procesador posiblemente avisaría a la impresora de tal circunstancia, mediante un registro de control, y aquélla podría pasar a un modo de espera con menor consumo.

Aunque la clasificación de los registros y sus características, tal y como se han presentado, son correctas desde un punto de vista teórico, es frecuente que en los controladores reales, para simplificar los circuitos y la gestión, se mezcle información de control y estado en un mismo

registro lógico -es decir, un único registro desde el punto de vista del procesador- e incluso que un bit tenga doble uso, de control y estado, según el momento. Un ejemplo común en los conversores analógico-digitales es disponer de un bit de control que escribe el procesador para iniciar la conversión -poniéndolo a `1`, por ejemplo- y que el dispositivo cambia de valor -a `0` en este ejemplo- cuando ha terminado -típica información de estado- y el resultado está disponible en un registro de datos.

Como se ha visto, cuando el procesador quiere realizar una determinada operación de entrada/salida debe leer o escribir en los registros del controlador. Por lo tanto, estos registros deben ser accesibles por el procesador a través de su conjunto de instrucciones. Este acceso puede realizarse de dos formas:

- Los registros de entrada/salida pueden formar parte del espacio de direcciones de memoria del ordenador. En este caso se dice que el sistema de entrada/salida está *mapeado en memoria*. El procesador lee y escribe en los registros de los controladores de la misma forma y mediante las mismas instrucciones con que lo hace de la memoria. Este esquema es el utilizado por la arquitectura ARM.

- Los registros de entrada/salida se ubican en un mapa de direcciones propio, independiente del mapa de memoria del sistema. En este caso se dice que el mapa de entrada salida es *independiente* o *aislado*. El procesador debe disponer de instrucciones especiales para acceder a los registros de entrada/salida. La ejecución de estas instrucciones se refleja en los circuitos externos del procesador, lo que permite al sistema distinguir estos accesos de los accesos a memoria y usar por tanto mapas distintos. Esta modalidad es utilizada por la arquitectura Intel de 32 y 64 bits, con instrucciones específicas tipo *in* y *out*.

Es necesario indicar que un procesador que dispone de instrucciones especiales de entrada/salida puede sin embargo utilizar un esquema mapeado en memoria, e incluso ambos. No es de extrañar por ello que el mapa del bus PCI Express y los dispositivos en un ordenador tipo PC incluyan regiones en memoria y otras en mapa específico de entrada/salida.

En este apartado hemos visto la estructura de los dispositivos de entrada/salida, que normalmente incluye un bloque de tecnología específica para interactuar con el mundo exterior, otro electrónico analógico que se relaciona con el anterior mediante transductores, y un bloque de electrónica digital, de la misma naturaleza que el resto de circuitos del

ordenador. Este bloque se llama *controlador del dispositivo* y facilita que el procesador se comunique con aquél mediante registros de control para enviar órdenes y configuraciones, registros de estado para comprobar el resultado de las operaciones y los posibles errores, y registros de datos para intercambiar información. Estos registros pueden ser accesibles en el mapa de memoria del procesador, mediante instrucciones de acceso a memoria, o en un mapa específico de entrada/salida que solo puede darse si el procesador incorpora instrucciones especiales. Veamos ahora cómo todos estos registros se usan para relacionar el procesador con los dispositivos.

7.3. Gestión de la entrada/salida

Aunque como hemos visto existen dispositivos con tasas de transferencia muy distintas, en general los periféricos son mucho más lentos que el procesador. Si un ordenador está ejecutando un solo programa y el flujo de ejecución depende de las operaciones de entrada/salida, esto no supondría un gran problema. El procesador puede esperar a que se vayan produciendo cambios en los dispositivos que se relacionan con el exterior, dado que su función consiste en ello. No es éste, sin embargo, el caso general. En el ejemplo que hemos utilizado para mostrar el uso de los distintos registros de los dispositivos, no parece razonable que el ordenador quede bloqueado esperando respuestas -señales de que el trabajo en curso ha terminado o indicaciones de error- de la impresora. Estamos más bien acostumbrados a seguir realizando cualquier otra actividad con el ordenador mientras la impresora va terminando hoja tras hoja, además sin percibir apenas disminución en el rendimiento del sistema.

Así pues, el aspecto fundamental de la gestión de la entrada/salida, que intenta en lo posible evitar que el procesador preste atención al dispositivo mientras no sea necesario, es la *sincronización*. Se pretende que el procesador y los dispositivos se sincronicen de tal modo que aquél solo les preste atención cuando hay alguna actividad que realizar -recoger datos si ya se han obtenido, enviar nuevos datos si se han consumido los anteriores, solucionar algún error o avisar al usuario de ello-. Sabemos que los registros de estado del controlador del dispositivo sirven para este fin, indicar que se ha producido alguna circunstancia que posiblemente requiere de atención. Por lo tanto, la forma más sencilla de sincronización con el dispositivo, llamada *prueba de estado*, *consulta de estado* o *encuesta* -en inglés, *polling*- consiste en que el procesador, durante la ejecución del programa en curso, lea de cuando en cuando los registros de estado necesarios y, si advierte que el dispositivo requiere atención, pase a ejecutar el código necesario para prestarla, posiblemente

contenido en una subrutina de gestión del dispositivo.

El código que aparece a continuación podría ser un ejemplo de consulta de estado.

ej-consulta-estado.s

```
1         ldr     r7, =ST_IMPR    @ Dirección del registro de estado
2         ldr     r6, =0x00000340 @ Máscara para diversos bits
3         ldr     r0, [r7]        @ Leemos el puerto
4         ands    r0, r6          @ y verificamos los bits
5         beq     sigue           @ Seguimos si no hay avisos
6         bl      TRAT_IMPR       @ Llamamos a la subrutina
7 sigue:
8         ...                     @ Continuamos sin prestar atención
```

En este ejemplo se consulta el registro de estado de una impresora y, si alguno de los bits 6, 8 o 9 está a `1`, saltamos a una subrutina de tratamiento para gestionar tal circunstancia. Posiblemente dicha rutina vería cuáles de los tres bits están activos en el registro de estado, y emprendería las acciones correspondientes para gestionar esa circunstancia. Si ninguno de los bits está a `1`, el procesador ignora la impresora y continúa con su programa.

Esta forma de gestionar la entrada/salida es muy sencilla, no requiere mayor complejidad al procesador y puede usarse en todos los sistemas. En muchos de ellos, si están dirigidos por eventos de entrada/salida -es decir, el flujo de ejecución del programa se rige por acciones de entrada/salida y no por condiciones de datos- como ocurre en la mayor parte de los *sistemas empotrados*, es la forma más adecuada de sincronizarse con la entrada/salida.

Sin embargo, para otros casos, sobre todo en los *sistemas de propósito general*, esta forma de gestión presenta serios inconvenientes. Por una parte, el procesador debe incluir instrucciones para verificar cada cierto tiempo el estado del dispositivo. Esta verificación consume tiempo inútilmente si el dispositivo no requiere atención. En un sistema con decenas de dispositivos la cantidad de tiempo perdida podría ser excesiva. Por otra parte, al consultar el estado cada cierto tiempo, la latencia se incrementa y se hace muy variable. Si un dispositivo activa un bit de estado justo antes de que el procesador lea el registro, la latencia será mínima. Sin embargo, si el bit se activa una vez se ha leído el registro, este cambio no será detectado por el procesador hasta que vuelva a realizar una consulta. De esta manera, si se desea una baja latencia se ha de consultar a menudo el registro, lo que consumirá tiempo de forma inútil.

A la vista de los problemas descritos, sería bueno que fuera el propio dispositivo el que avisara al procesador en caso de necesitar su atención, sin que éste tuviera que hacer nada de forma activa. Dado que el pro-

cesador es el encargado de gestionar todo el sistema, en último término sería quien podría decidir qué dispositivos tienen permiso para avisarle y si hacer o no caso a los avisos una vez recibidos. Estas ideas se recogen en el mecanismo de gestión de la entrada/salida mediante interrupciones.

Según esta idea, el mecanismo de gestión de entrada/salida mediante interrupciones permite que cuando un dispositivo, con permisos para ello, activa un aviso en sus registros de estado, esto provoque una señal eléctrica que fuerza al procesador, al terminar de ejecutar la instrucción en curso, a saltar automáticamente al código que permite gestionar el dispositivo. Cuando se completa este tratamiento, el procesador continúa ejecutando la instrucción siguiente a la que estaba ejecutando cuando llegó la interrupción, como si hubiera retornado de una subrutina -pero con más implicaciones que estudiaremos a continuación-. El símil más usado es el de la llamada telefónica, que llega mientras estamos leyendo tranquilamente un libro. Al sonar el teléfono -señal de interrupción- dejamos una marca en la página que estamos leyendo, y vamos a atenderla. Al terminar, continuamos con la lectura donde la habíamos dejado. De esta manera los dispositivos son atendidos con muy poca latencia y los programas de aplicación no necesitan incluir instrucciones de consulta ni preocuparse de la gestión de la entrada/salida.

De las explicaciones anteriores se deduce que el mecanismo de interrupciones se sustenta mediante el hardware del procesador y de la entrada/salida. Efectivamente, no todos los procesadores están diseñados para poder gestionar interrupciones, aunque en realidad hoy en día solo los microcontroladores de muy bajo coste no lo permiten. Veamos los elementos y procedimientos que necesitan incluir el procesador y los dispositivos para que se pueda gestionar la entrada/salida mediante interrupciones:

- El aviso le llega al procesador, como hemos dicho, mediante una señal eléctrica. Esto requiere que el procesador -o su núcleo, en los procesadores y microcontroladores con dispositivos integrados- disponga de una o varias líneas -pines o contactos eléctricos- para recibir interrupciones. Estas líneas se suelen denominar *líneas de interrupción* o *IRQn* -de *Interrupt Request*, en inglés- donde la *n* indica el número en caso de haber varias. Los controladores de los dispositivos capaces de generar interrupciones han de poder a su vez generar estas señales eléctricas, por lo que también disponen de una -o varias en algunos casos- señales eléctricas de salida para enviarlas al procesador.

- El procesador, guiado por el código con el que ha sido programado, es el gestor de todo el sistema. Así pues, debe poder seleccionar qué dispositivos tienen permiso para interrumpirlo y cuáles no. Es-

to se consigue mediante los bits de *habilitación de interrupciones*. Normalmente residen en registros de control de los controladores de dispositivos, que tendrán uno o más según los tipos de interrupciones que puedan generar. Además el procesador dispone de uno o varios bits de control propios para deshabilitar totalmente las interrupciones, o hacerlo por grupos, según prioridades, etcétera. Más adelante explicaremos este aspecto con más detalle.

- La arquitectura de un procesador especifica cómo debe responder a la señalización de una interrupción habilitada. La organización del procesador y sus circuitos deben permitir que, al acabar la ejecución de una instrucción, se verifique si hay alguna interrupción pendiente y, en caso afirmativo, se cargue en el contador de programa la dirección de la primera instrucción del código que debe atenderla, lo que se conoce como *rutina de tratamiento* o *rutina de servicio de la interrupción*, *RTI*. El procesador suele realizar más acciones en respuesta, como el cambio de estado a modo privilegiado o supervisor, el uso de otra pila u otro conjunto de registros, la deshabilitación automática de las interrupciones, etcétera. Todos estos cambios deben deshacerse al volver, para recuperar el estado en que se encontraba el procesador al producirse la interrupción y poder continuar con el código de aplicación. Más adelante se analizará con más detalle el comportamiento del procesador para tratar una interrupción.

- El último mecanismo que debe proveer el hardware del procesador, según lo especificado en su arquitectura, tiene que ver con la obtención de la dirección de inicio de la RTI, la dirección a la que saltar cuando se recibe una interrupción. Esta dirección, que puede ser única o dependiente de la interrupción en particular, recibe el nombre de *vector de interrupción*. En el caso más sencillo hay una única línea IRQ y un solo vector de interrupción. La RTI debe entonces consultar todos los dispositivos habilitados para determinar cuál o cuáles de ellos activaron sus bits de estado y deben por tanto ser atendidos. En los casos más complejos existen distintas interrupciones, identificadas con un número de interrupción diferente y asociadas a un vector diferente. Para ello, el procesador debe tener diversas líneas de interrupción independientes o implementar un protocolo especial de interrupción en que, además de una señal eléctrica, el dispositivo indica al procesador el número de interrupción. En este caso, cada causa de interrupción puede tener su propia RTI, o bien unos pocos dispositivos se agrupan en la misma, haciendo más rápida la consulta por parte de la RTI.

El mecanismo de interrupciones ha demostrado ser tan eficaz que se

ha generalizado más allá de la entrada/salida en lo que se llama *excepciones -exceptions* o *traps* en inglés-. Una excepción sirve para señalar cualquier circunstancia fuera de lo habitual -por lo tanto, excepcional- durante el funcionamiento de un procesador en su relación con el resto de componentes del ordenador. En este marco más amplio, las interrupciones son excepciones generadas por los dispositivos de entrada/salida. Otras excepciones se utilizan para señalar errores -accesos a memoria inválidos, violaciones de privilegio, división por cero, etcétera- que en muchos casos pueden ser tratados por el sistema y dar lugar a extensiones útiles. Por ejemplo, el uso del disco duro como extensión de la memoria principal se implementa mediante la excepción de fallo de página; las excepciones de coprocesador no presente permiten incorporar emuladores al sistema, como la unidad en coma flotante emulada en los antiguos PC. Entre las excepciones se incluyen también las generadas voluntariamente por software mediante instrucciones específicas o registros a tal efecto de la arquitectura, que al provocar un cambio al modo de ejecución privilegiado, permiten implementar las llamadas al sistema como se verá al estudiar sistemas operativos.

Así pues, las interrupciones son, en la mayoría de los procesadores, un tipo de excepciones asociadas a los dispositivos de entrada/salida y su gestión. Una diferencia fundamental con el resto de excepciones radica en el hecho de que las interrupciones no se generan con la ejecución de ninguna instrucción -un acceso a memoria incorrecto se genera ejecutando una instrucción de acceso a memoria; una división por cero al ejecutar una instrucción de división- por lo que son totalmente asíncronas con la ejecución de los programas y no tienen ninguna relación temporal con ellos que pueda ser conocida a priori.

Vamos a describir con más detalle todos los conceptos sobre interrupciones expuestos más arriba. Para ello recorreremos las sucesivas fases relacionadas con las interrupciones en la implementación y ejecución de un sistema, comentando exclusivamente los aspectos pertinentes a este tema.

El uso de interrupciones para la gestión de la entrada/salida se debe tener en cuenta desde el diseño del hardware del sistema. El procesador seleccionado deberá ser capaz de gestionarlas y contar, por tanto, con una o varias líneas externas de interrupción. Es posible además que utilice interrupciones vectorizadas y el número de interrupción -asociado al vector- se entregue al señalarla mediante cierto protocolo de comunicación específico. En este caso habrá que añadir al sistema algún controlador de interrupciones. Estos dispositivos suelen ampliar el número de líneas de interrupción del sistema para conectar diversos dispositivos y se encargan de propagar al procesador las señales en tales líneas, enviando además el número de interrupción correspondiente. Desde el punto de vista del procesador, el controlador de interrupciones funciona como un

sencillo dispositivo de entrada/salida. Ejemplos de estos dispositivos son el *NVIC* utilizado en la arquitectura ARM y que se verá más adelante o el *8259A* asociado a los procesadores Intel x86. En estos controladores se puede programar el número de interrupción asociado a cada línea física, la prioridad, las máscaras de habilitación y algún otro aspecto relacionado con el comportamiento eléctrico de las interrupciones.

Es interesante comentar que una interrupción puede señalarse eléctricamente mediante un flanco o mediante nivel. En el primer caso, una transición de estado bajo a alto o viceversa en la señal eléctrica produce que se detecte la activación de una interrupción. En el segundo caso, es el propio nivel lógico presente en la línea, `1` o `0`, el que provoca la detección. Existe una diferencia fundamental en el comportamiento de ambas opciones, que tiene repercusión en la forma de tratar las interrupciones. Un flanco es un evento único y discreto, no tiene duración real, mientras que un nivel eléctrico se puede mantener todo el tiempo que sea necesario. Así pues, un dispositivo que genera un flanco, ha mandado un aviso al procesador y posiblemente no esté en disposición de mandar otro mientras no sea atendido. Un dispositivo que genera una interrupción por nivel, continúa generándola hasta que no sea atendido y se elimine la causa que provocó la interrupción. Así, una rutina de tratamiento de interrupciones debe verificar que atiende todas las interrupciones pendientes, tratando adecuadamente los dispositivos que las señalaron. De esta manera se evita perder las que se señalan por flanco y se dejan de señalar las que lo hacen por nivel. Más adelante, al comentar las RTI, volveremos a tratar esta circunstancia. Los dispositivos capaces de generar interrupciones deberán tener sus líneas de salida de interrupción conectadas a la entrada correspondiente del procesador o del controlador de interrupciones, y por supuesto, tener sus registros accesibles en el mapa de memoria o de entrada/salida mediante la lógica de decodificación del sistema.

Una vez diseñado el hardware hay que programar el código de las diversas RTI y configurar los vectores de interrupción para que se produzcan las llamadas adecuadamente. Las direcciones de los vectores suelen estar fijas en el mapa de memoria del procesador, es decir en su arquitectura, de forma que a cada número de interrupción le corresponde una dirección de vector determinada. Algunos sistemas permiten configurar el origen de la tabla de vectores de interrupción, entonces la afirmación expuesta antes hace referencia, no al valor absoluto del vector, sino a su desplazamiento con respecto al origen. Sea como sea, los vectores de interrupción suelen reservar una zona de memoria de tamaño fijo -y reducido- en la que no hay espacio para el código de las RTI. Para poder ubicar entonces el código de tratamiento con libertad en la zona de memoria que se decida al diseñar el software del sistema, y con espacio suficiente, se utilizan dos técnicas. La más sencilla consiste en

dejar entre cada dos vectores de interrupción el espacio suficiente para una instrucción de salto absoluto. Así pues, al llegar la interrupción se cargará el PC con la dirección del vector y se ejecutará la instrucción de salto que nos llevará efectivamente al código de la RTI en la dirección deseada. La segunda opción requiere más complejidad para el hardware del procesador. En este caso, en la dirección del vector se guarda la dirección de inicio de la RTI, usando una especie de direccionamiento indirecto. Cuando se produce la interrupción, lo que copiamos en el PC no es la dirección del vector, sino la dirección contenida en el vector, lo que otorga al sistema flexibilidad para ubicar las RTI libremente en memoria. Esta última opción es la utilizada en las arquitecturas ARM e Intel de 32 y 64 bits.

Las dos etapas explicadas tienen lugar durante el diseño del sistema y están ya realizadas antes de que éste funcione. El hardware del sistema y el código en ROM ya están dispuestos cuando ponemos el ordenador en funcionamiento. La siguiente etapa es la configuración del sistema, que se realiza una vez al arrancar, antes de que comience el funcionamiento normal de las aplicaciones. Esta fase es sencilla y conlleva únicamente la asignación de prioridades a las interrupciones y la habilitación de aquéllas que deban ser tratadas. Por supuesto, en sistemas complejos pueden cambiarse de forma dinámica ambas cosas, según las circunstancias del uso o de la ejecución, aunque es normal que se realice al menos una configuración básica inicial. Es conveniente aprovechar esta descripción para comentar algo más sobre la prioridad de las interrupciones. En un sistema pueden existir numerosas causas de interrupción y es normal que algunas requieran un tratamiento mucho más inmediato que otras. En un teléfono móvil inteligente es mucho más prioritario decodificar un paquete de voz incorporado en un mensaje de radiofrecuencia durante una conversación telefónica que responder a un cambio en la orientación del aparato. Como las interrupciones pueden coincidir en lapsos temporales pequeños, es necesario aportar mecanismos que establezcan prioridades entre ellas. De esta manera, si llegan varias interrupciones al mismo tiempo el sistema atenderá exclusivamente a la más prioritaria. Además, durante una RTI es normal que se mantengan deshabilitadas las interrupciones de prioridad inferior a la que se está tratando. Para ello el sistema debe ser capaz de asignar una prioridad a cada interrupción o grupo de ellas; esto se asocia normalmente, igual que el vector, al número de interrupción o a la línea física. Una consecuencia de la priorización de interrupciones es que las rutinas de tratamiento suelen concluir revisando las posibles interrupciones pendientes, que se hayan producido mientras se trataba la primera. De esta manera no se pierden interrupciones por una parte y se evita por otra que nada más volver de una RTI se señale otra que hubiera quedado pendiente, con la consiguiente pérdida de tiempo. Cuando varios dispositivos comparten el

mismo número y vector de interrupción, entonces la priorización entre ellas se realiza en el software de tratamiento, mediante el orden en que verifica los bits de estado de los distintos dispositivos.

Una vez el sistema en funcionamiento, ya configurado, las interrupciones pueden llegar de forma asíncrona con la ejecución de instrucciones. En este caso el procesador, al terminar la instrucción en curso, de alguna de las formas vistas, carga en el contador de programa la dirección de la RTI. Previamente debe haber guardado el valor que contenía el contador de programa para poder retornar a la instrucción siguiente a la que fue interrumpida. Al mismo tiempo se produce un cambio a modo de funcionamiento privilegiado -por supuesto, solo en aquellos procesadores que disponen de varios modos de ejecución- y se deshabilitan las interrupciones. Bajo estas circunstancias comienza la ejecución de la RTI.

Dado que una interrupción puede ocurrir en cualquier momento, es necesario guardar el estado del procesador -registros- que vaya a ser modificado por la RTI, lo que suele hacerse en la pila. Si la rutina habilita las interrupciones para poder dar paso a otras más prioritarias, deberá preservar también la copia del contador de programa guardada por el sistema -que frecuentemente se almacena en un registro especial del sistema-. Posteriormente, la rutina de servicio tratará el dispositivo de la forma adecuada, normalmente intentando invertir el menor tiempo posible. Una vez terminado el tratamiento, la rutina recuperará el valor de los registros modificados y recuperará el contador de programa de la instrucción de retorno mediante una instrucción especial -normalmente de retorno de excepción- que devuelva al procesador al modo de ejecución original.

En algunos sistemas, diseñados para tratar las excepciones de forma especialmente eficiente, todos los cambios de estado y de preservación de los registros comentados se realizan de forma automática por el hardware del sistema, que dispone de un banco de registros de respaldo para no modificar los de usuario durante la RTI. Este es el caso de la arquitectura ARM, en que una rutina de tratamiento de interrupción se comporta a nivel de programa prácticamente igual que una subrutina.

En este apartado hemos visto que la sincronización entre el procesador y los dispositivos es el punto clave de la gestión de la entrada/salida. De forma sencilla, el procesador puede consultar los bits de estado de un dispositivo para ver si necesita atención, con lo que se gestiona mediante consulta de estado o encuesta. Si el procesador incorpora el hardware necesario, puede ser el dispositivo el que le avise de que necesita su atención, mediante interrupciones. En este mecanismo, el procesador debe incorporar más complejidad en sus circuitos, pero los programas pueden diseñarse de forma independiente de la entrada/salida. Para la gestión de excepciones, que generaliza e incluye la de interrupciones, el proce-

sador debe ser capaz de interrumpir la ejecución de la instrucción en curso y de volver a la siguiente una vez terminado el tratamiento; de saber dónde comienza la rutina de servicio RTI mediante vectores; de establecer prioridades entre aquéllas y de preservar el estado para poder continuar la ejecución donde se quedó al llegar la interrupción.

7.4. Transferencia de datos y DMA

La transferencia de datos entre el procesador y los dispositivos se realiza, como se ha visto, a través de los registros de datos. En el caso de un teclado, un ratón u otros dispositivos con una productividad de pocos bytes por segundo, no es ningún problema que el procesador transfiera los datos del dispositivo a la memoria, para procesarlos más adelante. De esta forma se liberan los registros de datos del dispositivo para que pueda registrar nuevas pulsaciones de teclas o movimientos y clicks del ratón por parte del usuario. Esta forma sencilla de movimiento de datos, que se adapta a la estructura de un computador vista hasta ahora, se denomina *transferencia de datos por programa*. En ella el procesador ejecuta instrucciones de lectura del dispositivo y escritura en memoria para ir transfiriendo uno a uno los datos disponibles. Análogamente, si se quisiera enviar datos a un dispositivo, se realizarían lecturas de la memoria y escrituras en sus registros de datos para cada uno de los datos a enviar.

Consideremos ahora que la aplicación que se quiere ejecutar consiste en la reproducción de una película almacenada en el disco duro. En este caso, el procesador debe ir leyendo bloques de datos del disco, decodificándolos de la forma adecuada y enviando por separado -en general- información a la tarjeta gráfica y a la tarjeta de sonido. Todos los bloques tratados son ahora de gran tamaño, y el procesador no debe únicamente copiarlos del dispositivo a la memoria o viceversa, sino también decodificarlos, extrayendo por separado el audio y el vídeo, descomprimiéndolos y enviándolos a los dispositivos adecuados. Y todo ello en tiempo real, generando y enviando no menos de 24 imágenes de unos 6 megabytes y unos 25 kilobytes de audio por segundo. Un procesador actual de potencia media o alta es capaz de realizar estas acciones sin problemas, en el tiempo adecuado, pero buena parte del tiempo lo invertiría en mover datos desde el disco a la memoria y de ésta a los dispositivos de salida, sin poder realizar trabajo más productivo. En un sistema multitarea, con muchas aplicaciones compitiendo por el uso del procesador y posiblemente también leyendo y escribiendo archivos en los discos, parece una pérdida de tiempo de la CPU dedicarla a estas transferencias de bloques de datos. Para solucionar este problema y liberar al procesador de la copia de grandes bloques de datos, se ideó una técnica llamada

acceso directo a memoria, o *DMA -Direct Memory Access* en inglés- en la que un dispositivo especializado del sistema, llamado *controlador de DMA*, se encarga de realizar dichos movimientos de datos.

El controlador de DMA es capaz de copiar datos desde un dispositivo de entrada/salida a la memoria o de la memoria a los dispositivos de entrada/salida. Hoy en día es frecuente que también sean capaces de transferir datos entre distintas zonas de memoria o de unos dispositivos de entrada/salida a otros. Recordemos que el procesador sigue siendo el gestor de todo el sistema, en particular de los buses a través de los cuales se transfieren datos entre los distintos componentes: memoria y entrada/salida. Por ello, los controladores de DMA deben ser capaces de actuar como maestros de los buses necesarios, para solicitar su uso y participar activamente en los procesos de arbitraje necesarios. Por otra parte, para que el uso del DMA sea realmente eficaz, el procesador debe tener acceso a los recursos necesarios para poder seguir ejecutando instrucciones y no quedarse en espera al no poder acceder a los buses. Aunque no vamos a entrar en detalle, existen técnicas tradicionales como el *robo de ciclos*, que permiten compaginar sin demasiada sobrecarga los accesos al bus del procesador y del controlador de DMA. Hoy en día, sin embargo, la presencia de memorias caché y el uso de sistemas complejos de buses independientes, hace que el procesador y los controladores de DMA puedan realizar accesos simultáneos a los recursos sin demasiados conflictos utilizando técnicas más directas de acceso a los buses.

Desde el punto de vista del procesador, el controlador de DMA es un dispositivo más de entrada/salida. Cada controlador dispone de varios canales de DMA, que se encargan de realizar copias simultáneas entre parejas de dispositivos. Cada canal se describe mediante un conjunto de registros de control que indican los dispositivos de origen y de destino, las direcciones de inicio de los bloques de datos correspondientes y la cantidad de datos a copiar. Tras la realización de las copias de datos el controlador indica el resultado -erróneo o correcto- mediante registros de estado, siendo también capaz de generar interrupciones.

Los controladores de DMA actuales son muy potentes y versátiles. Es normal que puedan comunicarse mediante los protocolos adecuados con ciertos dispositivos de entrada/salida para esperar a que haya datos disponibles y copiarlos luego en memoria hasta llenar el número de datos programado. Esto por ejemplo liberaría al procesador de tener que leer los datos uno a uno de un conversor analógico-digital, ahorrándose los tiempos de espera inherentes a la conversión. También es frecuente que puedan encadenar múltiples transferencias separadas en una sola orden del procesador, mediante estructuras de datos enlazadas que residen en memoria. Cada una de estas estructuras contiene la dirección de origen, de destino, el tamaño a copiar y la dirección de la siguiente estructura. De esta manera se pueden leer datos de diversos buffers de un disposi-

tivo y copiarlos en otros tantos pertenecientes a distintas aplicaciones, respondiendo así a una serie de llamadas al sistema desde diferentes clientes.

En este apartado se ha descrito la *transferencia de datos por programa*, que es la forma más sencilla de copiar datos entre los dispositivos de entrada/salida y la memoria, ejecutando las correspondientes instrucciones de lectura y escritura por parte del procesador. Se ha descrito el *acceso directo a memoria* como otra forma de transferir datos que libera de esa tarea al procesador, y se han comentado las características de los dispositivos necesarios para tal técnica, los controladores de DMA.

CAPÍTULO 8

Entrada/Salida: dispositivos

Índice

8.1. Entrada/salida de propósito general (GPIO - General Purpose Input Output)

La forma más sencilla de entrada/salida que podemos encontrar en un procesador son sus propios pines de conexión eléctrica con el exterior. Si la organización del procesador permite relacionar direcciones del mapa de memoria o de entrada salida con algunos pines, la escritura de un `1` o `0` lógicos por parte de un programa —arquitectura— en esas direcciones se reflejará en cierta tensión eléctrica en el pin, normalmente 0V para el nivel bajo y 5 o 3,3V para el alto, que puede ser utilizada para activar

o desactivar algún dispositivo externo. Por ejemplo, esto nos permitiría encender o apagar un LED mediante instrucciones de nuestro programa. De modo análogo, en el caso de las entradas, si el valor eléctrico presente en el pin se ve traducido por el diseño eléctrico del circuito en un `1` o `0` lógico que se puede leer en una dirección del sistema, podremos detectar cambios en el exterior de nuestro procesador. De esta manera, por ejemplo, nuestro programa podrá consultar si un pulsador está libre u oprimido, y tomar decisiones en función de su estado.

Veámoslo en un sencillo ejemplo:

ej-entrada-salida.s

```
1        ldr     r0, [r7, #PULSADOR]     @ Leemos el nivel
2        cmp     r0, #1                  @ Si no está pulsado
3        bne     sigue                   @ seguimos
4        mov     r0, #1                  @ Escribimos 1 para
5        str     r0, [r7, #LED]          @ encender el LED
```

El fragmento de código anterior supuestamente enciende un LED escribiendo un `1` en la dirección «`r7 + LED`» si el pulsador está presionado, es decir, cuando lee un `1` en la dirección «`r7 + PULSADOR`». Es un ejemplo figurado que simplifica el caso real. El apartado siguiente profundiza en la descripción de la *GPIO* (*General Purpose Input/Ouput* en inglés) y describe con más detalle sus características en los sistemas reales.

8.1.1. La GPIO en la E/S de los sistemas

La GPIO (*General Purpose Input Output*) es tan útil y necesaria que está presente en todos los sistemas informáticos. Los PC actuales la utilizan para leer pulsadores o encender algún LED del chasis. Por otra parte, en los microcontroladores, sistemas completos en un chip, la GPIO tiene más importancia y muestra su mayor complejidad y potencia. Vamos a analizar a continuación los aspectos e implicaciones de la GPIO y su uso en estos sistemas.

Aspectos lógicos y físicos de la GPIO o Programación y electrónica de la GPIO

En el ejemplo que hemos visto antes se trabajaba exclusivamente con un bit, que se corresponde con un pin del circuito integrado, tanto en entrada como en salida, utilizando instrucciones de acceso a una palabra de memoria, 32 bits en la arquitectura ARM. En la mayor parte de los sistemas, los diversos pines de entrada/salida se agrupan en *palabras*, de tal forma que cada acceso como los del ejemplo afectaría a todos los pines asociados a la palabra a cuya dirección se accede. De esta manera, se habla de *puertos* refiriéndose a cada una de las direcciones

asociadas a conjuntos de pines en el exterior del circuito, y cada pin individual es un bit del puerto. Así por ejemplo, si hablamos de *PB12* —en el microcontrolador ATSAM3X8E— nos estamos refiriendo al bit 12 del puerto de salida llamado PB —que físicamente se corresponde con el pin 86 del encapsulado LQFP del microcontrolador, algo que es necesario saber para diseñar el hardware del sistema—. En este caso, para actuar —modificar o comprobar su valor— sobre bits individuales o sobre conjuntos de bits es necesario utilizar máscaras y operaciones lógicas para no afectar a otros pines del mismo puerto. Suponiendo que el LED se encuentra en el bit 12 y el pulsador en el bit 20 del citado puerto PB, una versión más verosímil del ejemplo propuesto sería:

ej-acceso-es.s

```
1        ldr     r7, =PB          @ Dirección del puerto
2        ldr     r6, =0x00100000 @ Máscara para el bit 20
3        ldr     r0, [r7]         @ Leemos el puerto
4        ands    r0, r6           @ y verificamos el bit
5        beq     sigue            @ Seguimos si está a 0
6        ldr     r6, =0x00001000 @ Máscara para el bit 12
7        ldr     r0, [r7]         @ Leemos el puerto
8        orr     r0, r6           @ ponemos a 1 el bit
9        str     r0, [r7]         @ y lo escribimos en el puerto
```

En este caso, en primer lugar se accede a la dirección del puerto PB para leer el estado de todos los bits y, mediante una máscara y la operación lógica *AND*, se verifica si el bit correspondiente al pulsador —bit 20— está a `1`. En caso afirmativo —cuando el resultado de AND no es cero— se lee de nuevo PB y mediante una operación OR y la máscara correspondiente se pone a `1` el bit 12, correspondiente al LED para encenderlo. La operación OR permite, en este caso, poner a `1` un bit sin modificar los demás. Aunque este ejemplo es más cercano a la realidad y sería válido para muchos microcontroladores, el caso del ATSAM3X8E es algo más complejo, como se verá en su momento.

Obviando esta complejidad, el ejemplo que se acaba de presentar es válido para mostrar la gestión por programa de la entrada y salida tipo GPIO. Sin embargo, es necesario que nos surja alguna duda al considerar —no lo olvidemos— que los pines se relacionan realmente con el exterior mediante magnitudes eléctricas. Efectivamente, el comportamiento eléctrico de un pin que funciona como entrada es totalmente distinto al de otro que se utiliza como salida, como veremos más adelante. Para resolver esta paradoja volvemos a hacer hincapié en que el ejemplo que se ha comentado es de gestión de la entrada/salida durante el funcionamiento del sistema, pero no se ha querido comentar, hasta ahora, que previamente hace falta una configuración de los puertos de la GPIO en que se indique qué pines van a actuar como entrada y cuáles como salida. Así

pues, asociado a la dirección en la que se leen o escriben los datos y que hemos llamado *PB* en el ejemplo, habrá al menos otra que corresponda a un registro de control de la GPIO en la que se indique qué pines se comportan como entradas y cuáles como salidas, lo que se conoce como *dirección de los pines*. Consideremos de nuevo el hecho diferencial de utilizar un pin —y su correspondiente bit en un puerto— como entrada o como salida. En el primer caso son los circuitos exteriores al procesador los que determinan la tensión presente en el pin, y la variación de ésta no depende del programa, ni en valor ni en tiempo. Sin embargo, cuando el pin se usa como salida, es el procesador ejecutando instrucciones de un programa el que modifica la tensión presente en el pin al escribir en su bit asociado. Se espera además —pensemos en el LED encendido— que el valor se mantenga en el pin hasta que el programa decida cambiarlo escribiendo en él otro valor. Si analizamos ambos casos desde el punto de vista de necesidad de almacenamiento de información, veremos que en el caso de la entrada nos limitamos a leer un valor eléctrico en cierto instante, valor que además viene establecido desde fuera y no por el programa ni el procesador, mientras que en la salida es necesario asociar a cada pin un espacio de almacenamiento para que el `1` o `0` escrito por el programa se mantenga hasta que decidamos cambiarlo, de nuevo de acuerdo con el programa. Esto muestra por qué la GPIO a veces utiliza dos puertos, con direcciones distintas, para leer o escribir en los pines del sistema. El registro —o *latch*— que se asocia a las salidas suele tener una dirección y las entradas —que no requieren registro pues leen el valor lógico fijado externamente en el pin— otra. Así, en el caso más común, un puerto GPIO ocupa al menos tres direcciones en el mapa: una para el registro de control que configura la dirección de los pines, otra para el registro de datos de salida y otra para leer directamente los pines a través del registro de datos de entrada. En la Figura 8.1 (obtenida del manual [Atm11]) se muestra la estructura interna de un pin de E/S de un microcontrolador de la familia Atmel AVR.

En este caso, que como hemos dicho es muy común, ¿qué ocurre si escribimos en un pin configurado como entrada, o si leemos un pin configurado como salida? Esto depende en gran medida del diseño electrónico de los circuitos de E/S del microcontrolador, pero en muchos casos el comportamiento es el siguiente: si leemos un pin configurado como salida podemos leer bien el valor almacenado en el registro, bien el valor presente en el pin. Ambos deberían coincidir a nivel lógico, salvo que algún error en el diseño del circuito o alguna avería produjeran lo contrario. Por ejemplo, en un pin conectado a masa es imposible que se mantenga un `1` lógico. Por otra parte, si escribimos en un pin configurado como entrada es común que, en realidad, se escriba en el registro de salida, sin modificar el valor en el pin. Este comportamiento es útil, dado que permite fijar un valor lógico conocido en un pin, antes de configurarlo

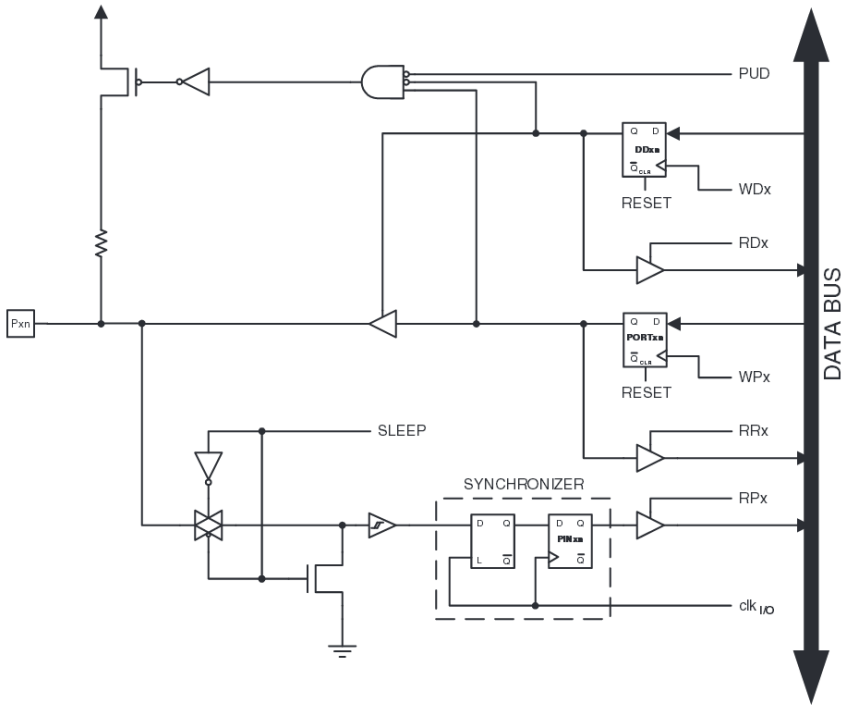


Figura 8.1: Estructura interna de un pin de E/S de un microcontrolador de la familia Atmel AVR

como salida. Dado que las entradas son eléctricamente más seguras, los pines suelen estar configurados como tales tras el reset del sistema. Así, el procedimiento normal para inicializar un pin de salida es escribir su valor mientras está configurado como entrada, y luego configurarlo como salida. Esto permite además fijar valores lógicos en el exterior mediante resistencias, que si son de valor elevado permitirán posteriormente el funcionamiento normal del pin como salida.

Para comprender adecuadamente esta última afirmación, vamos a estudiar brevemente las características eléctricas de los pines de entrada/salida. Como hemos dicho, la forma de interactuar con el exterior de un pin de E/S es típicamente mediante una tensión eléctrica presente en él. En el caso de las salidas, cuando escribimos un `1` lógico en el registro se tendrá un cierto voltaje en el pin correspondiente, y otro distinto cuando escribimos un `0`. En el de las entradas, al leer el pin obtendremos un `1` o un `0` según la tensión fijada en él por la circuitería externa.

Tratemos en primer lugar las salidas, y consideremos el caso más común hoy en día de lógica positiva —las tensiones de los `1` lógicos son mayores que las de los `0`—. Las especificaciones eléctricas de los

circuitos indican típicamente un valor mínimo, `VOHMIN`, que especifica la mínima tensión que vamos a tener en dicho pin cuando escribimos en él un `1` lógico. Se especifica solo el valor mínimo porque se supone que el máximo es el de alimentación del circuito. Por ejemplo 5V de alimentación y 4,2V como `VOHMIN` serían valores razonables. Estos valores nos garantizan que la tensión en el pin estará comprendida entre 4,2 y 5V cuando en él tenemos un `1` lógico. De manera análoga se especifica `VOLMAX` como la mayor tensión que podemos tener en un pin cuando en él escribimos un `0` lógico. En este caso, la tensión mínima es 0 voltios y el rango garantizado está entre `VOLMAX`, por ejemplo 0,8V, y 0V. En las especificaciones de valores anteriores, `V` indica voltaje, `O` salida (*output*), `H` y `L` se refieren a nivel alto (*high*) y bajo (*low*) respectivamente, mientras que `MAX` y `MIN` indican si se trata, como se ha dicho, de un valor máximo o mínimo. Inmediatamente veremos cómo estas siglas se combinan para especificar otros parámetros.

El mundo real tiene, sin embargo, sus límites, de tal modo que los niveles de tensión eléctrica especificados requieren, para ser válidos, que se cumpla una restricción adicional. Pensemos en el caso comentado antes en que una salida se conecta directamente a masa —es decir, 0V—. La especificación garantiza, según el ejemplo, una tensión mínima de 4,2V, pero sabemos que el pin está a un potencial de 0V por estar conectado a masa. Como la resistividad de las conexiones internas del circuito es despreciable, la intensidad suministrada por el pin, y con ella la potencia disipada, debería ser muy elevada para poder satisfacer ambas tensiones. Sabemos que esto no es posible, que un pin normal de un circuito integrado puede suministrar como mucho algunos centenares de miliamperios —y estos valores tan altos solo se alcanzan en circuitos especializados de potencia—. Por esta razón, la especificación de los niveles de tensión en las salidas viene acompañada de una segunda especificación, la de la intensidad máxima que se puede suministrar —en el nivel alto— o aceptar —en el nivel bajo— para que los citados valores de tensión se cumplan. Estas intensidades, `IOHMAX` e `IOLMAX` o simplemente `IOMAX` cuando es la misma en ambas direcciones, suelen ser del orden de pocas decenas de miliamperios —normalmente algo más de 20mA, lo requerido para encender con brillo suficiente un LED—. Así pues la especificación de los valores de tensión de las salidas se garantiza siempre y cuando la corriente que circule por el pin no supere el valor máximo correspondiente.

La naturaleza y el comportamiento de las entradas son radicalmente distintos, aunque se defina para ellas un conjunto similar de parámetros. Mediante un puerto de entrada queremos leer un valor lógico que se relacione con la tensión presente en el pin, fijada por algún sistema eléctrico exterior. Así pues, la misión de la circuitería del pin configurado como entrada es detectar niveles de tensión del exterior, con la menor

influencia en ellos que sea posible. De este modo, una entrada aparece para un circuito externo como una resistencia muy elevada, lo que se llama una *alta impedancia*. Dado que es un circuito activo y no una resistencia, el valor que se especifica es la intensidad máxima que circula entre el exterior y el circuito integrado, que suele ser despreciable en la mayor parte de los casos —del orden de pocos microamperios o menor— . Los valores especificados son `IIHMAX` e `IILMAX` o simplemente `IIMAX` si son iguales. En este caso la `I` significa entrada (*input*). Según esto, el circuito externo puede ser diseñado sabiendo la máxima corriente que va a disiparse hacia el pin, para generar las tensiones adecuadas para ser entendidas como `0` o `1` al leer el pin de entrada. Para ello se especifican `VIHMIN` y `VILMAX` como la mínima tensión de entrada que se lee como un `1` lógico y la máxima que se lee como un `0`, respectivamente. En ambos casos, por diseño del microcontrolador, se sabe que la corriente de entrada está limitada, independientemente del circuito externo.

¿Qué ocurre con una tensión en el pin comprendida entre `VIHMIN` y `VILMAX`? La lectura de un puerto de entrada siempre devuelve un valor lógico, por lo tanto cuando la tensión en el pin se encuentra fuera de los límites especificados, se lee también un valor lógico `1` o `0` que no se puede predecir según el diseño del circuito —una misma tensión podría ser leída como nivel alto en un pin y bajo en otro—. Visto de otra forma, un circuito —y un sistema en general— se debe diseñar para que fije una tensión superior a `VIHMIN` cuando queremos señalar un nivel alto, e inferior a `VILMAX` cuando queremos leer un nivel bajo. Otra precaución a tener en cuenta con las entradas es la de los valores máximos. En este caso el peligro no es que se lea un valor lógico distinto del esperado o impredecible, sino que se dañe el chip. Efectivamente, una tensión superior a la de alimentación o inferior a la de masa puede dañar definitivamente el pin de entrada e incluso todo el circuito integrado.

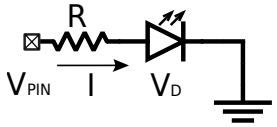


Figura 8.2: Conexión de un LED a un pin de E/S de un microcontrolador

Hagamos un pequeño estudio de los circuitos eléctricos relacionados con los dispositivos de nuestro ejemplo, el LED y el pulsador. Comencemos como viene siendo habitual por el circuito de salida. En la Figura 8.2 se muestra esquemáticamente la conexión de un LED a un pin de E/S de un microcontrolador. Un LED, por ser un diodo, tiene una tensión de conducción más o menos fija —en realidad en un LED depende más de la corriente que en un diodo de uso general— que en uno de color

rojo está entorno a los 1,2V. Por otra parte, a partir de 10mA el brillo del diodo es adecuado, pudiendo conducir sin deteriorarse hasta 30mA o más. Supongamos en nuestro microcontrolador los valores indicados más arriba para `VOHMIN` y `VOLMAX`, y una corriente de salida superior a los 20mA. Si queremos garantizar 10mA al escribir un `1` lógico en el pin, nos bastará con polarizar el LED con una resistencia que limite la corriente a este valor en el peor caso, es decir cuando la tensión de salida sea `VOHMIN`, es decir 4,2V. Mediante la ley de Ohm tenemos:

$$I = \frac{V}{R} \rightarrow 10mA = \frac{4{,}2V - 1{,}2V}{R} = \frac{3V}{R} \rightarrow R = 300\Omega$$

Una vez fijada esta resistencia, podemos calcular el brillo máximo del led, que se daría cuando la tensión de salida es de 5V, y entonces la corriente de 12,7mA aproximadamente.

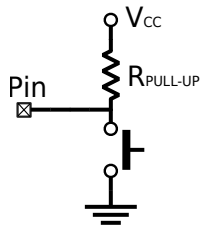


Figura 8.3: Conexión de un pulsador a un pin de E/S de un microcontrolador

Veamos ahora cómo conectar un pulsador a una entrada del circuito. En la Figura 8.3 se muestra el circuito esquemático de un pulsador conectado a un pin de E/S de un microcontrolador. Un pulsador no es más que una placa de metal que se apoya o se separa de dos conectores, permitiendo o no el contacto eléctrico entre ellos. Es, pues, un dispositivo electromecánico que no genera de por sí ninguna magnitud eléctrica. Para ello, hay que conectarlo en un circuito y, en nuestro caso, en uno que genere las tensiones adecuadas. Para seguir estrictamente los ejemplos, podemos pensar que el pulsador hace contacto entre los 5V de la alimentación y el pin. De este modo, al pulsarlo, conectaremos el pin a la alimentación y leeremos en él un `1`, tal y como se ha considerado en el código de ejemplo. Sin embargo, si no está pulsado, el pin no está conectado a nada por lo que el valor presente en él sería, en general, indefinido. Por ello el montaje correcto requiere que el pin se conecte a otro nivel de tensión, masa —0V— en este caso, a través de una resistencia para limitar la corriente al pulsar. Como la corriente de entrada en el pin es despreciable, el valor de la resistencia no es crítico, siendo lo habitual usar decenas o cientos de KΩ. Según este circuito y siguiendo

el ejemplo, al pulsar leeríamos un nivel alto y en caso de no pulsar, un `0` lógico.

La configuración más habitual es, sin embargo la contraria: conectar el pulsador a masa con uno de sus contactos y al pin y a la alimentación, a través de una resistencia, con el otro. De esta forma los niveles lógicos se invierten y se lee un `1` lógico en caso de no pulsar y un nivel bajo al hacerlo. Esto tiene implicaciones en el diseño de los microcontroladores y en la gestión de la GPIO, que nos ocupa. Es tan habitual el uso de resistencias conectadas a la alimentación —llamadas *resistencias de pull-up* o simplemente *pull-ups*— que muchos circuitos las llevan integradas en la circuitería del pin y no es necesario añadirlas externamente. Estas resistencias pueden activarse o no en las entradas, por lo que suele existir alguna forma de hacerlo, un nuevo registro de control del GPIO en la mayor parte de los casos.

8.1.2. Interrupciones asociadas a la GPIO

Como sabemos, las interrupciones son una forma de sincronizar el procesador con los dispositivos de entrada/salida para que éstos puedan avisar de forma asíncrona al procesador de que requieren su atención, sin necesidad de que aquél se preocupe periódicamente de atenderlos —lo que sería encuesta o prueba de estado. Los sistemas avanzados de GPIO incorporan la posibilidad de avisar al procesador de ciertos cambios mediante interrupciones, para poder realizar su gestión de forma más eficaz. Existen dos tipos de interrupciones que se pueden asociar a la GPIO, por supuesto siempre utilizada como entrada, como veremos a continuación.

En primer lugar se pueden utilizar los pines como líneas de interrupción, bien para señalar un cambio relativo al circuito conectado al pin, como oprimir un pulsador, bien para conectar una señal de un circuito externo y que así el circuito sea capaz de generar interrupciones. En este último caso el pin de la GPIO haría el papel de una línea de interrupción externa de un procesador. En ambos casos suele poder configurarse si la interrupción se señala por nivel o por flanco y su polaridad. En segundo lugar, y asociado a las características de bajo consumo de los microcontroladores, se tiene la interrupción por cambio de valor. Esta interrupción puede estar asociada a un pin o un conjunto de ellos, y se activa cada vez que alguno de los pines de entrada del grupo cambia su valor, desde la última vez que se leyó. Esta interrupción, además, suele usarse para sacar al procesador de un modo de bajo consumo y activarlo otra vez para reaccionar frente al cambio indicado.

El uso de interrupciones asociadas a la GPIO requiere añadir nuevos registros de control y estado, para configurar las interrupciones y sus

características —control— y para almacenar los indicadores —*flags*— que informen sobre las circunstancias de la interrupción.

8.1.3. Aspectos avanzados de la GPIO

Además de las interrupciones y la relación con los modos de bajo consumo, los microcontroladores avanzados añaden características y, por lo tanto, complejidad, a sus bloques de GPIO. Aunque estas características dependen bastante de la familia de microcontroladores, se pueden encontrar algunas tendencias generales que se comentan a continuación.

En primer lugar tenemos las modificaciones eléctricas de los bloques asociados a los pines. Estas modificaciones afectan solo a subconjuntos de estos pines y en algunos casos no son configurables, por lo que se deben tener en cuenta fundamentalmente en el diseño electrónico del sistema. Por una parte tenemos pines de entrada que soportan varios umbrales lógicos —lo más normal es 5V y 3,3V para el nivel alto—. También es frecuente encontrar entradas con disparador de Schmitt para generar flancos más rápidos en las señales eléctricas en el interior del circuito, por ejemplo en entradas que generen interrupciones, lo que produce que los valores `VIHMIN` y `VILMAX` en estos pines estén más próximos, reduciendo el rango de tensiones indeterminadas —a nivel lógico— entre ellos. Tenemos también salidas que pueden configurarse como colector abierto —*open drain*, en nomenclatura CMOS— lo que permite utilizarlas en sistemas AND cableados, muy utilizados en buses.

Otra tendencia actual, de la que participa el ATSAM3X8E, es utilizar un muestreo periódico de los pines de entrada, lo que requiere almacenar su valor en un registro, en lugar de utilizar el valor presente en el pin en el momento de la lectura. De esta manera es posible añadir filtros que permitan tratar ruido eléctrico en las entradas o eliminar los rebotes típicos en los pulsadores e interruptores. En este caso, se incorporan a la GPIO registros para activar o configurar estos métodos de filtrado. Esta forma de tratar las entradas requiere de un reloj para muestrearlas y almacenar su valor en el registro, lo que a su vez requiere poder parar este reloj para reducir el consumo eléctrico.

La última característica asociada a la GPIO que vamos a tratar surge de la necesidad de versatilidad de los microcontroladores. Los dispositivos actuales, además de gran número de pines en su GPIO, incorporan muchos otros dispositivos —conversores ADC y DAC, buses e interfaces estándar, etcétera— que también necesitan de pines específicos para relacionarse con el exterior. Para dejar libertad al diseñador de seleccionar la configuración del sistema adecuada para su aplicación, muchos pines pueden usarse como parte de la GPIO o con alguna de estas funciones específicas. Esto hace que exista un complejo subsistema de encaminado de señales entre los dispositivos internos y los pines, que afecta directa-

mente a la GPIO y cuyos registros de configuración suele considerarse como parte de aquélla.

8.1.4. GPIO en el Atmel ATSAM3X8E

El microcontrolador ATSAM3X8E dispone de bloques de GPIO muy versátiles y potentes. Dichos bloques, llamados *Parallel Input/Output Controller* (*PIO* en inglés) se describen en detalle a partir de la página 641 del manual. Vamos a resumir en este apartado los aspectos más importantes, centrándonos en la versión ATSAM3X8E del microcontrolador, presente en la tarjeta *Arduino Due*. En la Figura 8.4 (obtenida del manual [Atm12]) se muestra la estructura interna de un pin de E/S del microcontrolador ATSAM3X8E.

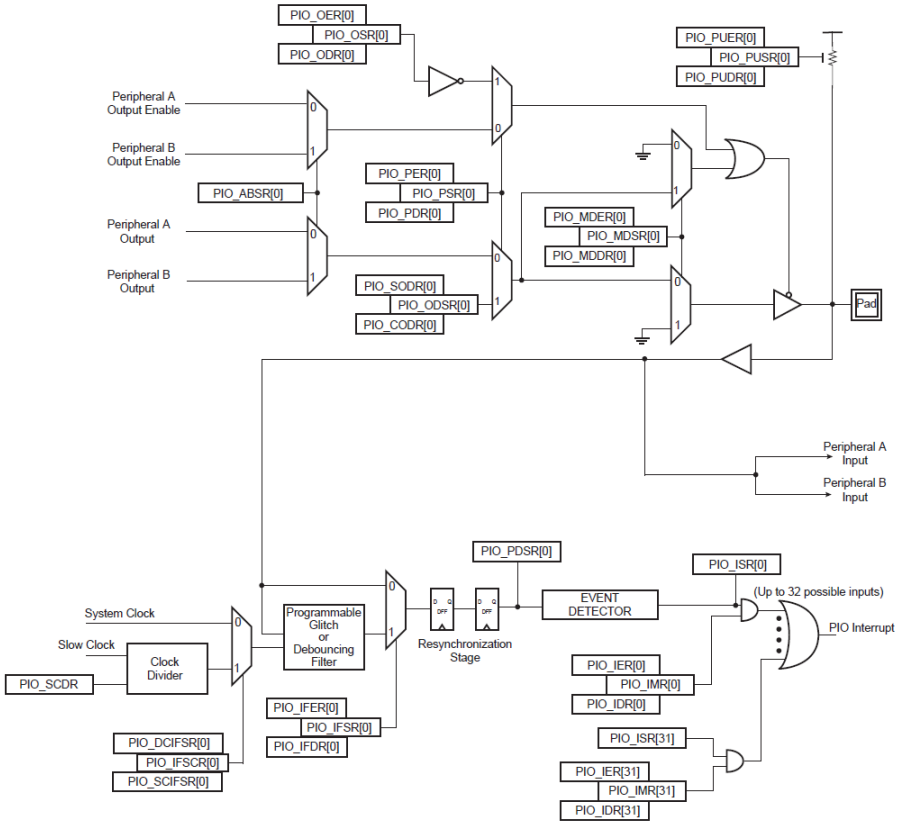


Figura 8.4: Estructura interna de un pin de E/S del microcontrolador ATSAM3X8E

Con un encapsulado LQFP de 144 pines, el microcontrolador dispone de 4 controladores PIO capaces de gestionar hasta 32 pines cada uno

de ellos, para un total de 103 líneas de GPIO. Cada una de estas líneas de entrada/salida es capaz de generar interrupciones por cambio de valor o como línea dedicada de interrupción; de configurarse para filtrar o eliminar rebotes de la entrada; de actuar con o sin *pull-up* o en colector abierto. Además, los pines de salida pueden ponerse a uno o a cero individualmente, mediante registros dedicados de *set* o *clear*, o conjuntamente a cualquier valor escribiendo en un tercer registro. Todas estas características hacen que cada bloque PIO tenga una gran complejidad, ocupando un espacio de 324 registros de 32 bits —1.296 direcciones— en el mapa de memoria. Veamos a continuación los registros más importantes y su uso.

Configuración como GPIO o E/S específica de otro periférico

La mayor parte de los pines del encapsulado pueden utilizarse como parte de la GPIO o con una función específica, seleccionable de entre dos dispositivos de E/S del microcontrolador. Así pues, para destinar un pin a la GPIO deberemos habilitarlo para tal fin. Si posteriormente queremos utilizar alguna de las funciones específicas, podremos volver a deshabilitarlo como E/S genérica. Como en muchos otros casos, y por motivos de seguridad y velocidad —ahorra tener que leer los registros para preservar sus valores— el controlador PIO dedica tres registros a este fin: uno para habilitar los pines, otro para deshabilitarlos y un tercero para leer el estado de los pines en un momento dado. De esta manera, dado que al habilitar y deshabilitar se escriben `1` en los bits afectados, sin modificar el resto, no es necesario preservar ningún estado al escribir. Veamos los registros asociados a esta funcionalidad:

- *PIO Enable Register* (`PIO_PER`): escribiendo un `1` en cualquier bit de este registro habilita el pin correspondiente para uso como GPIO, inhibiendo su uso asociado a otro dispositivo de E/S.
- *PIO Disable Register* (`PIO_PDR`): al revés que el anterior, escribiendo un `1` se deshabilita el uso del pin como parte de la GPIO y se asocia a uno de los dispositivos periféricos asociados.
- *PIO Status Register* (`PIO_PSR`): este registro de solo lectura permite conocer en cualquier momento el estado de los pines asociados al PIO. Un `1` en el bit correspondiente indica que son parte de la GPIO mientras que un `0` significa que están dedicados a la función del dispositivo asociado.
- *PIO Peripheral AB Select Register* (`PIO_ABSR`): permite seleccionar a cuál de los dos posibles dispositivos periféricos está asociado el pin en caso de no estarlo a la GPIO. Un `0` selecciona el `A` y un `1` el

`B`. Los dispositivos identificados como `A` y `B` dependen de cada pin en particular.

Configuración y uso genéricos como GPIO

Una vez los pines se han asignado a la GPIO, es necesario realizar su configuración específica. Esto incluye indicar si su dirección es entrada o salida, y activar o no las resistencias de *pull-up* o la configuración en colector abierto. Veamos los registros del PIO que se utilizan:

- *Output Enable Register* (`PIO_OER`): escribiendo un `1` en cualquier bit de este registro configura el pin correspondiente como salida.
- *Output Disable Register* (`PIO_ODR`): escribiendo un `1` en cualquier bit de este registro deshabilita el pin correspondiente como salida, quedando entonces como pin de entrada.
- *Output Status Register* (`PIO_OSR`): este registro de solo lectura permite conocer en cualquier momento la dirección de los pines. Un `1` en el bit correspondiente indica que el pin está configurado como salida mientras que un `0` significa que el pin es una entrada.

Se dispone además del trío de registros *Pull-up Enable*, *Pull-up Disable* y *Pull-up Status* que permiten respectivamente activar, desactivar y leer el estado de configuración de las resistencias de *pull-up*. En este último caso, un `1` indica deshabilitada y un `0` habilitada. Por último, los tres registros *Multi-driver Enable*, *Multi-driver Disable* y *Multi-driver Status* permiten configurar eléctricamente el pin en colector abierto —o deshacer esta configuración— y leer el estado de los pines a este respecto —de la forma habitual, no invertida como en el caso anterior—.

Una vez configurada la GPIO, nuestro programa debe únicamente trabajar con los pines, escribiendo y leyendo valores según la tarea a realizar. De nuevo el bloque GPIO ofrece una gran versatilidad, a costa de cierta complejidad, como veremos a continuación. Comencemos con la lectura de los valores de entrada, algo sencillo dado que basta con leer el registro *Pin Data Status Register* (`PIO_PDSR`) para obtener los valores lógicos presentes en ellos. Es conveniente indicar que para poder leer los pines de entrada —igual que para muchas otras funciones del PIO— el reloj que lo sincroniza debe estar activado. La gestión de las salidas presenta algo más de complejidad dado que existen dos modos de actuar sobre cada una de ellas. Por una parte, tenemos la forma común en este microcontrolador, disponiendo de un registro para escribir unos y otro para escribir ceros. Este modo, que en muchas ocasiones simplifica la escritura en los pines, presenta el problema de que no se pueden escribir de forma simultánea unos y ceros. Por ello existe un segundo modo, en

que se escribe en una sola escritura el valor, con los unos y ceros deseado, sobre el registro de salida. Para que este modo no afecte a todos los pines gestionados por el PIO —hasta 32— existe un registro adicional para seleccionar aquéllos a los que va a afectar esta escritura. Para poder implementar todos estos modos, el conjunto de registros relacionados con la escritura de valores en las salidas, es el siguiente:

- *Set Output Data Register* (`PIO_SODR`): escribiendo un `1` en cualquier bit de este registro se escribe un uno en la salida correspondiente.
- *Clear Output Data Register* (`PIO_CODR`): escribiendo un `1` en cualquier bit de este registro se escribe un cero en la salida correspondiente.
- *Output Data Status Register* (`PIO_ODSR`): al leer este registro obtenemos en cualquier momento el valor lógico que hay en las salidas cuando se lee. Al escribir en él, modificamos con el valor escrito los valores de aquellas salidas habilitadas para escritura directa en `PIO_OWSR`.
- *Output Write Enable Register* (`PIO_OWER`): escribiendo un `1` en cualquier bit de este registro habilita tal pin para escritura directa.
- *Output Write Disable Register* (`PIO_OWDR`): escribiendo un `1` en cualquier bit de este registro deshabilita tal pin para escritura directa.
- *Output Write Status Register* (`PIO_OWSR`): este registro de solo lectura permite conocer en cualquier momento las salidas habilitadas para escritura directa.

Gestión de interrupciones asociadas a la GPIO

El controlador PIO es capaz de generar diversas interrupciones asociadas a los pines de la GPIO a él asociados. Para que dichas interrupciones se puedan propagar al sistema, la interrupción generada por el PIO debe estar convenientemente programada en el controlador de interrupciones del sistema, llamado *NVIC*. Además el reloj de sincronización del PIO debe estar activado. Dándose estas circunstancias, el PIO gestiona diferentes fuentes de interrupción que disponen de un registro de señalización y otro de máscara. La activación de cualquier causa de interrupción se reflejará siempre en el registro de señalización y se generará o no la interrupción en función del valor de la máscara correspondiente, que estará a `1` si la interrupción está habilitada. La causa básica de interrupción es el cambio de valor en un pin. Sin embargo, esta causa

puede modificarse para que la interrupción se genere cuando se detecta un flanco de subida o de bajada, o un nivel determinado en el pin. Veamos el conjunto de registros que permiten esta funcionalidad, y su uso:

- *Interrupt Enable Register* (`PIO_IER`): escribiendo un `1` en cualquier bit de este registro se habilita la interrupción correspondiente.
- *Interrupt Disable Register* (`PIO_IDR`): escribiendo un `1` en cualquier bit de este registro se deshabilita la interrupción correspondiente.
- *Interrupt Mask Register* (`PIO_IMR`): al leer este registro obtenemos el valor de la máscara de interrupciones, que se corresponde con las interrupciones habilitadas.
- *Interrupt Status Register* (`PIO_ISR`): en este registro se señalan con un `1` las causas de interrupción pendientes, es decir aquéllas que se han dado, sea cual sea su tipo, desde la última vez que se leyó este registro. Se pone a `0` automáticamente al ser leído.
- *Additional Interrupt Modes Enable Register* (`PIO_AIMER`): escribiendo un `1` en cualquier bit de este registro se selecciona la causa de interrupción adicional, por flanco o por nivel.
- *Additional Interrupt Modes Disable Register* (`PIO_AIMDR`): escribiendo un `1` en cualquier bit de este registro se selecciona la causa básica de interrupción, cambio de valor en el pin.
- *Additional Interrupt Modes Mask Register* (`PIO_AIMMR`): este registro de solo lectura permite saber si la causa de interrupción configurada es cambio de valor, lo que se indica con un `0`, o modo adicional, con un `1`.
- *Edge Select Register* (`PIO_ESR`): escribiendo un `1` en cualquier bit de este registro se selecciona el flanco como la causa de interrupción adicional.
- *Level Select Register* (`PIO_LSR`): escribiendo un `1` en cualquier bit de este registro se selecciona el nivel como la causa de interrupción adicional.
- *Edge/Level Status Register* (`PIO_ELSR`): este registro de solo lectura permite saber si la causa de interrupción adicional configurada es flanco, lo que se indica con un `0`, o nivel, con un `1`.
- *Falling Edge/Low Level Select Register* (`PIO_FELLSR`): escribiendo un `1` en cualquier bit de este registro se selecciona el flanco de

bajada o el nivel bajo, según `PIO_ELSR`, como la polaridad de interrupción.

- *Rising Edge/High Level Select Register* (`PIO_REHLSR`): escribiendo un `1` en cualquier bit de este registro se selecciona el flanco de subida o el nivel alto, según `PIO_ELSR`, como la polaridad de interrupción.

- *Fall/Rise Low/High Status Register* (`PIO_FRLHSR`): este registro de solo lectura permite saber la polaridad de la interrupción.

Registros adicionales y funciones avanzadas del PIO

Una de las funciones avanzadas del controlador PIO es la eliminación de ruidos en las entradas. Aunque no se va a ver en detalle —recordemos que siempre se puede acceder a la especificación completa en el manual— conviene comentar que se tiene la posibilidad de activar filtros para las señales de entrada, configurables como filtros de ruido —traducción aproximada de *glitches*— o para la eliminación de rebotes si a la entrada se conecta un pulsador —*debouncing*—. Como estos filtros están basados en el sobremuestreo de la señal presente en el pin —recordemos que las entradas no leen directamente el pin sino que muestrean su valor y lo almacenan en un registro— es posible además variar el reloj asociado a este sobremuestreo.

La última posibilidad que ofrece el controlador PIO es la de bloquear o proteger contra escritura parte de los registros de configuración que se han descrito, para prevenir que errores en la ejecución del programa produzcan cambios indeseados en la configuración.

8.1.5. Controladores PIO en el ATSAM3X8E

Conocida la información que aparece en el texto anterior, para hacer programas que interactúen con la GPIO del microcontrolador solo falta conocer las direcciones del mapa de memoria en que se sitúan los registros de los controladores PIO del ATSAM3X8E y la dirección —más bien desplazamiento u *offset*— de cada registro dentro del bloque. El Cuadro 8.1 muestra las direcciones base de los controladores PIO de que dispone el sistema.

Así mismo, en los Cuadross 8.2, 8.3 y 8.4 se muestran los desplazamientos (*offsets*) de los registros de E/S de cada uno de los controladores PIO, de forma que para obtener la dirección de memoria efectiva de uno de los registros hay que sumar a la dirección base del controlador PIO al que pertenece, el desplazamiento indicado para el propio registro:

PIO	Pines de E/S disponibles	Dirección base
PIOA	30	0x400E 0E00
PIOB	32	0x400E 1000
PIOC	31	0x400E 1200
PIOD	10	0x400E 1400

Cuadro 8.1: Direcciones base de los controladores PIO del ATSAM3X8E

Registro	Alias	Desplazamiento
PIO Enable Register	PIO_PER	0x0000
PIO Disable Register	PIO_PDR	0x0004
PIO Status Register	PIO_PSR	0x0008
Output Enable Register	PIO_OER	0x0010
Output Disable Register	PIO_ODR	0x0014
Output Status Register	PIO_OSR	0x0018
Glitch Input Filter Enable Register	PIO_IFER	0x0020
Glitch Input Filter Disable Register	PIO_IFDR	0x0024
Glitch Input Filter Status Register	PIO_PIFSR	0x0028
Set Output Data Register	PIO_SODR	0x0030
Clear Output Data Register	PIO_CODR	0x0034
Output Data Status Register	PIO_ODSR	0x0038
Pin Data Status Register	PIO_PDSR	0x003C

Cuadro 8.2: Registros de E/S de cada controlador PIO y sus desplazamientos. Parte I

8.1.6. La tarjeta de entrada/salida

Para poder practicar con la GPIO se ha diseñado una pequeña tarjeta que se inserta en los conectores de expansión de la Arduino Due. La tarjeta dispone de un LED RGB —rojo *Red* verde *Green* azul *Blue*— conectado a tres pines que se usarán como salidas, y un pulsador conectado a un pin de entrada. Los tres diodos del LED están configurados

Registro	Alias	Desplazamiento
Interrupt Enable Register	`PIO_IER`	`0x0040`
Interrupt Disable Register	`PIO_IDR`	`0x0044`
Interrupt Mask Register	`PIO_IMR`	`0x0048`
Interrupt Status Register	`PIO_ISR`	`0x004C`
Multi-driver Enable Register	`PIO_MDER`	`0x0050`
Multi-driver Disable Register	`PIO_MDDR`	`0x0054`
Multi-driver Status Register	`PIO_MDSR`	`0x0058`
Pull-up Disable Register	`PIO_PUDR`	`0x0060`
Pull-up Enable Register	`PIO_PUER`	`0x0064`
Pad Pull-up Status Register	`PIO_PUSR`	`0x0068`
Peripheral AB Select Register	`PIO_ABSR`	`0x0070`
System Clock Glitch Input Filter Select Register	`PIO_SCIFSR`	`0x0080`
Debouncing Input Filter Select Register	`PIO_DIFSR`	`0x0084`
Glitch or Debouncing Input Filter Clock Selection Status Register	`PIO_IFDGSR`	`0x0088`
Slow Clock Divider Debouncing Register	`PIO_SCDR`	`0x008C`

Cuadro 8.3: Registros de E/S de cada controlador PIO y sus desplazamientos. Parte II

en ánodo común, por lo que se encienden al escribir un `0` en la salida correspondiente. Cada canal del LED lleva su correspondiente resistencia para limitar la corriente; el terminal común se debe conectar, a través del cable soldado a la tarjeta, a la salida de `3.3V` de la Arduino Due. El pulsador se conecta a un pin de entrada a través de una resistencia de protección, y a masa. Activando la resistencia de *pull-up* asociada al pin, se leerá un `1` lógico si el interruptor no está pulsado, y un `0` al pulsarlo.

El Cuadro 8.5 y las Figuras 9.3 y 9.4 completan la información técnica acerca de la tarjeta.

Registro	Alias	Desplazamiento
Output Write Enable	PIO_OWER	0x00A0
Output Write Disable	PIO_OWDR	0x00A4
Output Write Status Register	PIO_OWSR	0x00A8
Additional Interrupt Modes Enable Register	PIO_AIMER	0x00B0
Additional Interrupt Modes Disable Register	PIO_AIMDR	0x00B4
Additional Interrupt Modes Mask Register	PIO_AIMMR	0x00B8
Edge Select Register	PIO_ESR	0x00C0
Level Select Register	PIO_LSR	0x00C4
Edge/Level Status Register	PIO_ELSR	0x00C8
Falling Edge/Low Level Select Register	PIO_FELLSR	0x00D0
Rising Edge/ High Level Select Register	PIO_REHLSR	0x00D4
Fall/Rise - Low/High Status Register	PIO_FRLHSR	0x00D8
Lock Status	PIO_LOCKSR	0x00E0
Write Protect Mode Register	PIO_WPMR	0x00E4
Write Protect Status Register	PIO_WPSR	0x00E8

Cuadro 8.4: Registros de E/S de cada controlador PIO y sus desplazamientos. Parte III

8.2. Gestión del tiempo

La medida del tiempo es fundamental en la mayoría de las actividades humanas y por ello, lógicamente, se incluye entre las características principales de los ordenadores, en los que se implementa habitualmente mediante dispositivos de entrada/salida. Anotar correctamente la fecha y hora de modificación de un archivo, arrancar automáticamente tareas con cierta periodicidad, determinar si una tecla se ha pulsado duran-

PIN	Función	Puerto	Bit
6	LED azul	PIOC	24
7	LED verde	PIOC	23
8	LED rojo	PIOC	22
13	Pulsador	PIOB	27

Cuadro 8.5: Pines y bits de los dispositivos de la tarjeta de E/S en la tarjeta Arduino Due

te más de medio segundo, son actividades comunes en los ordenadores que requieren de una correcta medida y gestión del tiempo. En estos ejemplos se puede ver además las distintas escalas y formas de tratar el tiempo. Desde expresar una fecha y hora de la forma habitual para las personas —donde además se deben tener en cuenta las diferencias horarias entre distintos países— hasta medir lapsos de varias horas o pocos milisegundos, los ordenadores son capaces de realizar una determinación adecuada de tiempos absolutos o retardos entre sucesos. Esto se consigue mediante un completo y elaborado sistema de tratamiento del tiempo, que tiene gran importancia dentro del conjunto de dispositivos y procedimientos relacionados con la entrada/salida de los ordenadores.

8.2.1. El tiempo en la E/S de los sistemas

Un sistema de tiempo real se define como aquél capaz de generar resultados correctos y a tiempo. Los ordenadores de propósito general pueden ejecutar aplicaciones de tiempo real, como reproducir una película o ejecutar un videojuego, de la misma forma en que mantienen la fecha y la hora del sistema, disparan alarmas periódicas, etcétera. Para ser capaces de ello, además de contar con la velocidad de proceso suficiente, disponen de un conjunto de dispositivos asociados a la entrada/salida que facilitan tal gestión del tiempo liberando al procesador de buena parte de ella. En los microcontroladores, dispositivos especialmente diseñados para interactuar con el entorno y adaptarse temporalmente a él, normalmente mediante proceso de tiempo real, el conjunto de dispositivos y mecanismos relacionados con el tiempo es mucho más variado e importante.

En todos los ordenadores se encuentra, al menos, un dispositivo tipo contador que se incrementa de forma periódica y permite medir intervalos de tiempo de corta duración —milisegundos o menos—. A partir de esta base de tiempos se puede organizar toda la gestión temporal del sistema, sin más que incluir los programas necesarios. Sin embargo se suele disponer de otro dispositivo que gestiona el tiempo en formato

humano —formato de fecha y hora— que permite liberar de esta tarea al software del sistema y además, utilizando alimentación adicional, mantener esta información aún con el sistema apagado. Por último, para medir eventos externos muy cortos, para generar señales eléctricas con temporización precisa y elevadas frecuencias, se suelen añadir otros dispositivos que permiten generar pulsos periódicos o aislados o medir por hardware cambios eléctricos en los pines de entrada/salida.

Todos estos dispositivos asociados a la medida de tiempo pueden avisar al sistema de eventos temporales tales como desbordamiento en los contadores o coincidencias de valores de tiempo —alarmas— mediante los correspondientes bits de estado y generación de interrupciones. Este variado conjunto de dispositivos se puede clasificar en ciertos grupos que se encuentran, de forma más o menos similar, en la mayoría de los sistemas. En los siguientes apartados se describen estos grupos y se indican sus características más comunes.

El temporizador del sistema

El temporizador —*timer*— del sistema es el dispositivo más común y sencillo. Constituye la base de medida y gestión de tiempos del sistema. Se trata de un registro contador que se incrementa de forma periódica a partir de cierta señal de reloj generada por el hardware del sistema. Para que su resolución y tiempo máximo puedan configurarse según las necesidades, es habitual encontrar un divisor de frecuencia o *prescaler* que permite disminuir con un margen bastante amplio la frecuencia de incremento del contador. De esta manera, si la frecuencia final de incremento es f, se tiene que el tiempo mínimo que se puede medir es el periodo, $T = 1/f$, y el tiempo que transcurre hasta que se desborde el contador $2^n \cdot T$, siendo n el número de bits del registro temporizador. Para una frecuencia de 10KHz y un contador de 32 bits, el tiempo mínimo sería 100μs y transcurrirían unos 429.496s —casi cinco días— hasta que se desbordara el temporizador. El temporizador se utiliza, en su forma más simple, para medir tiempos entre dos eventos —aunque uno de ellos pueda ser el inicio del programa—. Para ello, se guarda el valor del contador al producirse el primer evento y se resta del valor que tiene al producirse el segundo. Esta diferencia multiplicada por el periodo nos da el tiempo transcurrido entre ambos eventos.

El ejemplo comentado daría un valor incorrecto si entre las dos lecturas se ha producido más de un desbordamientos del reloj. Por ello, el temporizador activa una señal de estado que generalmente puede causar una interrupción cada vez que se desborda, volviendo a `0`. El sistema puede tratar esta información, sobre todo la interrupción generada, de distintas formas. Por una parte se puede extender el contador con variables en memoria para tener mayor rango en la cuenta de tiempos. Es

habitual también utilizarla como interrupción periódica para gestión del sistema —por ejemplo, medir los tiempos de ejecución de los procesos en sistemas multitarea—. En este último caso es habitual poder recargar el contador con un valor distinto de `0` para tener un control más fino de la periodicidad de la interrupción, por ello suele ser posible escribir sobre el registro que hace de contador.

Además de este funcionamiento genérico del contador, existen algunas características adicionales bastante extendidas en muchos sistemas. Por una parte, no es extraño que la recarga del temporizador después de un desbordamiento se realice de forma automática, utilizando un valor almacenado en otro registro del dispositivo. De esta forma, el software de gestión se libera de esta tarea. En sistemas cuyo temporizador ofrece una medida de tiempo de larga duración, a costa de una resolución poco fina, de centenares de ms, se suele generar una interrupción con cada incremento del contador. La capacidad de configuración de la frecuencia de tal interrupción es a costa del *prescaler*. Es conveniente comentar que, en arquitecturas de pocos bits que requieren contadores con más resolución, la lectura de la cuenta de tiempo requiere varios accesos —por ejemplo, un contador de 16 bits requeriría dos en una arquitectura de 8 bits—. En este caso pueden leerse valores erróneos si el temporizador se incrementa entre ambos accesos, de forma que la parte baja se desborde. Por ejemplo, si el contador almacena el valor `0x3AFF` al leer la parte baja y se incrementa a `0x3B00` antes de leer la alta, el valor leído será `0x3BFF`, mucho mayor que el real. En estos sistemas el registro suele constar de una copia de respaldo que se bloquea al leer una de las dos partes, con el valor de todo el temporizador en ese instante. De esta manera, aunque el temporizador real siga funcionando, las lecturas se harán de esta copia bloqueada, evitando los errores.

Más adelante se describen las particularidades del temporizador *RTT* (*Real-time Timer*) del ATSAM3X8E.

Otros dispositivos temporales

Si solo se dispone de un dispositivo temporizador se debe elegir entre tener una medida de tiempos de larga duración —hasta de varios años en muchos casos— para hacer una buena gestión del tiempo a lo largo de la vida del sistema o tener una buena resolución —pocos ms o menos— para medir tiempos con precisión. Por eso es común que los sistemas dispongan de varios temporizadores que, compartan o no la misma base de tiempos, pueden configurar sus periodos mediante *prescaleres* individuales. Estos sistemas, con varios temporizadores, añaden otras características que permiten una gestión mucho más completa del tiempo. Las características más comunes de estas extensiones del temporizador básico se analizan a continuación.

Algún registro temporizador puede utilizar como base de tiempos una entrada externa —un pin del microcontrolador, normalmente—. Esto permite por una parte tener una fuente de tiempo con las características que se deseen o utilizar el registro como contador de eventos, no de tiempos, dado que las señales en el pin no tienen por qué cambiar de forma periódica.

Se utilizan registros de comparación, con el mismo número de bits del temporizador, que desencadenan un evento cuando el valor del temporizador coincide con el de alguno de aquéllos. Estos eventos pueden ser internos, normalmente la generación de alguna interrupción, o externos, cambiando el nivel eléctrico de algún pin y pudiendo generar salidas dependientes del tiempo.

Se añaden registros de copia que guardan el valor del temporizador cuando se produce algún evento externo, además de poder generar una interrupción. Esto permite medir con precisión el tiempo en que ocurre algo en el exterior, con poca carga para el software del sistema.

Están muy extendidas como salidas analógicas aquéllas que permiten modulación de anchura de pulsos, *PWM —Pulse Width Modulation—*. Se dispone de una base de tiempos asociada a un temporizador que marca la frecuencia del canal PWM, y de un registro que indica el porcentaje de nivel alto o bajo de la señal de salida. De esa forma se genera una señal periódica que se mantiene a nivel alto durante un cierto tiempo y a nivel bajo el resto del ciclo. Como el ciclo de trabajo depende del valor almacenado en el registro, la cantidad de potencia —nivel alto— entregada de forma analógica en cada ciclo se relaciona directamente con su valor —magnitud digital—. Si la salida PWM ataca un dispositivo que se comporta como un filtro pasa-baja, lo que es muy frecuente en dispositivos reales —bombillas y LEDs, calefactores, motores, etcétera— se tiene una conversión digital-analógica muy efectiva, basada en el tiempo.

El reloj en tiempo real

En un computador, el reloj en tiempo real o *RTC* (*Real-time Clock*) es un circuito específico encargado de mantener la fecha y hora actuales incluso cuando el computador está desconectado de la alimentación eléctrica. Es por este motivo que suele estar asociado a una batería o a un condensador que le proporciona la energía necesaria para seguir funcionando cuando se interrumpe la alimentación.

Habitualmente este periférico emplea como frecuencia base una señal de 32.768 Hz, es decir, una señal cuadrada que completa 32.768 veces un ciclo OFF-ON cada segundo. Esta frecuencia es la empleada habitualmente por los relojes de cuarzo, dado que coincide con 2^{15} ciclos por segundo, con lo cual el bit de peso 15 del contador de ciclos cambia de valor exactamente una vez por segundo y puede usarse como señal de

activación del segundero en el caso de un reloj analógico o del contador de segundos en uno digital.

El módulo RTC se suele presentar como un dispositivo independiente conteniendo el circuito oscilador, el contador, la batería y una pequeña cantidad de memoria RAM que se usa para almacenar la configuración de la *BIOS* del computador. Este módulo se incorpora en la placa base del computador presentando, respecto de la opción de implementarlo por software, las siguientes ventajas:

- El procesador queda liberado de la tarea de contabilizar el tiempo. El RTC dispone de algunos registros de E/S mediante los cuales se pueden configurar y consultar la fecha y hora actuales,
- Suele presentar mayor precisión, dado que está diseñado específicamente.
- La presencia de la batería permite mantener el reloj en funcionamiento cuando el computador se apaga.

8.2.2. El temporizador del Atmel ATSAM3X8E y del sistema Arduino

La arquitectura ARM especifica un temporizador llamado *System Timer* como la base principal de tiempos del sistema, con una frecuencia de incremento similar a la del procesador lo que le permite medir intervalos de tiempo muy pequeños —del orden de microsegundos—. Para este temporizador, que no es otra cosa que un dispositivo de entrada/salida, aunque especial en el sistema, reserva sin embargo una excepción del sistema, llamada *SysTick*, con un número de interrupción fijo en el sistema, a diferencia del resto de dispositivos, cuyos números de interrupción no están fijados por la arquitectura.

El microcontrolador ATSAM3X8E implementa el *System Timer* especificado en la arquitectura. Se trata de un dispositivo de entrada/salida que se comporta como un temporizador convencional, que se decrementa con cada pulso de su reloj. Dispone de cuatro registros, que se describen a continuación:

- *Control and Status Register* (`CTRL`): de los 32 bits que contiene este registro solo 4 son útiles, tres de control y uno de estado. De los primeros, el bit 2 —`CLKSOURCE`— indica la frecuencia del temporizador, que puede ser la misma del sistema o un octavo de ésta; el bit 1 —`TICKINT`— es la habilitación de interrupción y el bit 0 —`ENABLE`— la habilitación del funcionamiento del temporizador. El bit 16 —`COUNTFLAG`— es el único de estado, e indica si el contador ha llegado a `0` desde la última vez que se leyó el registro.

- *Reload Value Register* (`LOAD`): cuando el temporizador llega a `0` recarga automáticamente el valor de 24 bits —los 8 más altos no se usan— presente en este registro, comenzando a decrementarse desde tal valor. De esta manera se puede ajustar con más precisión el tiempo transcurrido hasta que se llega a cero y con ello, si están habilitadas, el tiempo entre interrupciones.

- *Current Value Register* (`VAL`): este registro guarda el valor actual del contador decreciente, de 24 bits —los 8 más altos no se usan—.

- *Calibration Value Register* (`CALIB`): contiene valores relacionados con la calibración de la frecuencia de actualización.

En el Cuadro 8.6 aparecen las direcciones de los registros citados.

Registro	Alias	Dirección
Control and Status Register	`CTRL`	`0xE000E010`
Reload Value Register	`LOAD`	`0xE000E014`
Current Value Register	`VAL`	`0xE000E018`
Calibration Value Register	`CALIB`	`0xE000E01C`

Cuadro 8.6: Registros del temporizador del ATSAM3X8E y sus direcciones de E/S

El entorno Arduino añade a cada programa el código necesario para la configuración del sistema y las rutinas de soporte necesarias. Entre ellas se tiene la configuración del *System Timer* y la rutina de tratamiento de la excepción *SysTick*. Este código configura el reloj del sistema, de 84MHz, como frecuencia de actualización del temporizador, y escribe el valor `0x01481F`, 83.999 en decimal, en el registro de recarga. De esta manera se tiene un cambio en el contador cada 12 nanosegundos más o menos y una interrupción cada milisegundo, lo que sirve de base para las funciones «`delay()`» y «`millis()`» del entorno. Ambas utilizan el contador de milisegundos del sistema «`_dwTickCount`», que es una variable en memoria que se incrementa en la rutina de tratamiento de *SysTick*. Las funciones de mayor precisión «`delayMicroseconds()`» y «`micros()`» se implementan leyendo directamente el valor del registro `VAL`.

8.2.3. El reloj en tiempo real del Atmel ATSAM3X8E

Algunos microcontroladores incorporan un RTC entre sus periféricos integrados, lo cual les permite disponer de fecha y hora actualizadas. En este caso, sin embargo, no suele existir alimentación específica para el

módulo RTC, con lo cual, al desaparecer la alimentación externa, se pierde la información de fecha y hora actuales.

El microcontrolador ATSAM3X8E posee un RTC cuya estructura se muestra en la Figura 8.5. Como puede apreciarse, recibe una señal de reloj `SCLK` (*Slow Clock*) generada internamente por el microcontrolador que presenta la ya mencionada frecuencia de 32.768 Hz. Esta señal se hace pasar por un divisor por 32768 para obtener una señal de reloj de exactamente 1 Hz, que se encargará de activar las actualizaciones de los contenidos de los registros que mantienen la hora y la fecha actuales, en ese orden.

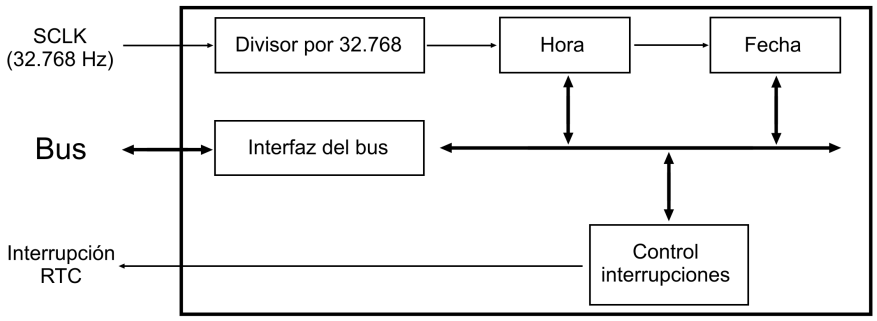


Figura 8.5: Estructura interna del RTC del ATSAM3X8E

Por otro lado, el RTC está conectado al bus interno del ATSAM3X8E para que se pueda acceder a los contenidos de sus registros. De esta forma es posible tanto leer la fecha y hora actuales, modificarlas y configurar las alarmas. Para ello, el RTC dispone de algunos registros de control encargados de gestionar las funciones de consulta, modificación y configuración del módulo.

Hora actual

La hora actual se almacena en un registro de 32 bits denominado `RTC_TIMR` (*RTC Time Register*), cuyo contenido se muestra en la Figura 8.6, donde la hora puede estar expresada en formato de 12 horas más indicador AM/PM —bit 22— o en formato de 24 horas. Todos los valores numéricos están codificados en *BCD* (*Binary Coded Decimal*, decimal codificado en binario), cuya equivalencia se muestra en el Cuadro 8.7.

Los segundos —`SEC`— se almacenan en los bits 0 al 6, conteniendo los bits del 0 al 3 el valor de las unidades. Dado que las decenas adoptan como máximo el valor 5, para este dígito solamente son necesarios tres

Decimal	BCD	Decimal	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Cuadro 8.7: Equivalencia entre decimal y BCD

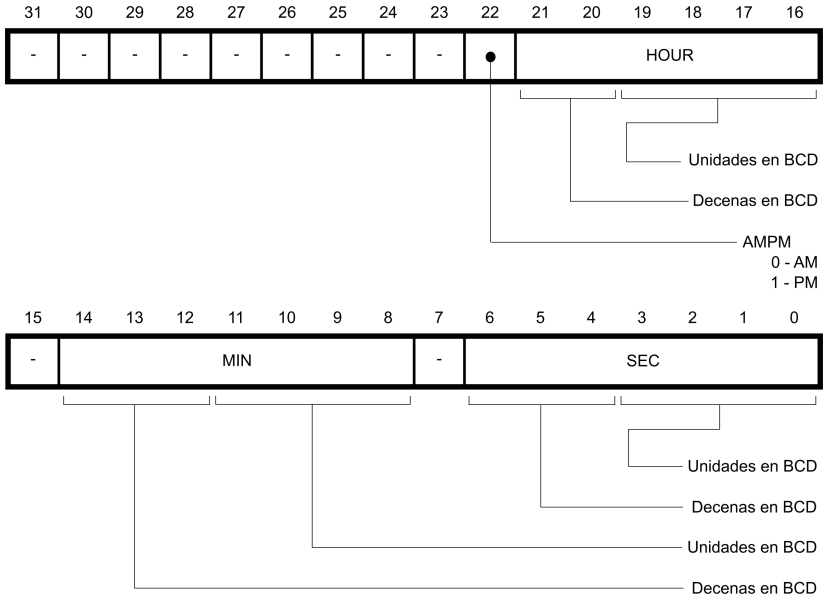


Figura 8.6: Contenido del registro RTC Time Register

bits —del 4 al 6—, por lo cual el bit 7 no se usa nunca y siempre debe valer cero.

Los minutos —`MIN`— se almacenan en los bits del 8 al 14. Las unidades se almacenan en los bits del 8 al 11 y con las decenas ocurre lo mismo que con los segundos: solamente hacen falta tres bits, del 12 al 14.

En cuanto a las horas —`HOUR`—, las decenas solamente pueden tomar los valores 0, 1 y 2, con lo cual es suficiente con dos bits —el 20 y el 21— y así el bit 22 queda para expresar la mañana y la tarde en el formato de 12 horas y el 23 no se usa.

El resto de bits —del 24 al 31— no se usan. El valor de la hora actual se lee y se escribe como un valor de 32 bits accediendo a este registro

con una sola operación de lectura o escritura.

Fecha actual

La fecha actual se almacena en el registro `RTC_CALR` (*RTC Calendar Register*) organizado como se muestra en la Figura 8.7:

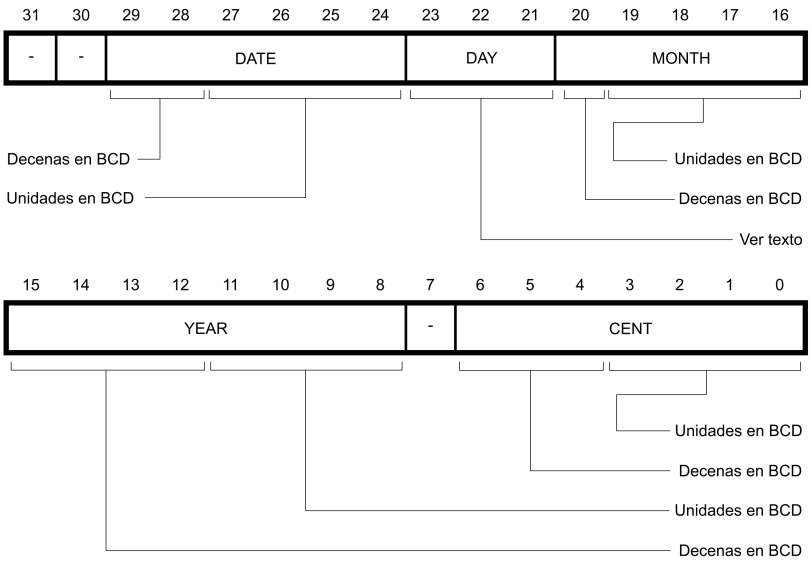


Figura 8.7: Contenido del registro RTC Calendar Register

Los bits del 0 al 6 contienen el valor del siglo —`CENTURY`—, pudiendo tomar solamente los valores 19 y 20 —refiriéndose, respectivamente, a los siglos 20 y 21—. Los bits del 0 al 3 almacenan las unidades de este valor (9 ó 0) y los bits del 4 al 6 las decenas (0 ó 2).

El año actual —`YEAR`— se almacena en los bits del 8 al 15, conteniendo los bits del 8 al 11 el valor BCD correspondiente a las unidades y los bits del 12 al 15 el valor BCD correspondiente a las decenas.

Esto confiere al ATSAM3X8E la capacidad de expresar la fecha actual en un rango de 200 años, desde el 1 de enero de 1900 hasta el 31 de diciembre de 2099.

El mes del año —`MONTH`— se almacena en los bits del 16 al 20, conteniendo el valor BCD de las unidades los bits del 16 al 19 y el de las decenas —`0` ó `1`— el bit 20.

El día de la semana —`DAY`— es almacenado en los bits del 21 al 24, pudiendo tomar valores comprendidos entre `0` y `7` cuyo significado es asignado por el usuario.

La fecha del mes —DATE— se almacena en los bits del 24 al 29, de forma que los bits del 24 al 27 contienen el valor en BCD de las unidades y los bits 28 y 29 el valor en BCD de las decenas (0, 1, 2 ó 3).

Lectura de la fecha y hora actuales

Para poder acceder a los registros del RTC se debe conocer tanto la dirección base que ocupa el periférico en el mapa de memoria como el desplazamiento del registro al que se desea acceder. En este caso, el RTC del ATSAM3X8E abarca 256 direcciones —desplazamientos comprendidos entre 0x00 y 0xFF— a partir de la dirección 0x400E 1A60. Los desplazamientos de los diferentes registros del RTC pueden consultarse en el Cuadro 8.8, donde puede verse que el correspondiente al registro RTC_TIMR es 0x08 y el del registro RTC_CALR es 0x0C. Así pues, para leer la fecha actual será necesario realizar una operación de lectura sobre la dirección 0x400E 1A6C y para obtener la hora actual será necesario leer el contenido de la dirección 0x400E 1A68.

Registro	Alias	Desplazamiento
Control Register	RTC_CR	0x00
Mode Register	RTC_MR	0x04
Time Registert	RTC_TIMR	0x08
Calendar Register	RTC_CALR	0x0C
Time Alarm Register	RTC_TIMALR	0x10
Calendar Alarm Register	RTC_CALALR	0x14
Status Register	RTC_SR	0x18
Status Clear Command Register	RTC_SCCR	0x1C
Interrupt Enable Register	RTC_IER	0x20
Interrupt Disable Register	RTC_IDR	0x24
Interrupt Mask Register	RTC_IMR	0x28
Valid Entry Register	RTC_VER	0x2C
Reserved Register	—	0x30–0xE0
Write Protect Mode Register	RTC_WPMR	0xE4
Reserved Register	—	0xE8–0xFC

Cuadro 8.8: Desplazamientos de los registros del RTC

Debido a que el RTC es independiente del resto del sistema y funciona de forma asíncrona respecto del mismo, para asegurar que la lectura de sus contenidos es correcta, es necesario realizarla por duplicado y comparar ambos resultados. Si son idénticos, es correcto. De lo contrario hay que repetir el proceso, requiriéndose un mínimo de dos lecturas y un máximo de tres para obtener el valor correcto.

Actualización de la fecha y hora actuales

La configuración de la fecha y hora actuales en el RTC requiere de un procedimiento a que se describe a continuación.

1. Inhibir la actualización del RTC. Esto se consigue mediante los bits `UPDCAL` para la fecha y `UPDTIM` para la hora. Ambos se encuentran, como muestra la Figura 8.8, en el registro de control `RTC_CR`. Cada uno de estos bits detiene la actualización del contador correspondiente cuando toma al valor `1`, permitiendo el funcionamiento normal del RTC cuando vale `0`. Así pues, si deseamos establecer la fecha actual, deberemos escribir un `1` en el bit de peso 1 del registro `RTC_CR` antes de modificar el registro `RTC_CALR`. Este registro, junto con los que sirven para configurar las alarmas, dispone de una protección contra escritura que se puede habilitar en el registro `RTC_WPMR` (*RTC Write Protect Mode Register*) introduciendo la clave correcta en el registro, cuyo contenido se muestra en la Figura 8.9. Para que el cambio de modo de protección contra escritura de los registros protegidos —`RTC_CR`, `RTC_CALALR` y `RTC_TIMALR`— se produzca, la clave introducida en el campo `WPKEY` debe ser `0x525443` —'RTC' en ASCII—, mientras el byte de menor peso de la palabra de 32 bits debe tomar el valor `0x00` para permitir la escritura y el valor `0x01` para impedirla. Por defecto, la protección contra escritura está deshabilitada.

2. Esperar la activación de `ACKUPD`, que es el bit de peso 0 del registro de estado `RTC_SR` —mostrado en la Figura 8.10—. En caso de que la generación de interrupciones esté activada, no será necesario consultar el registro, dado que se producirá una interrupción.

3. Una vez se haya detectado que el bit `ACKUPD` ha tomado el valor `1`, es necesario restablecer este indicador escribiendo un `1` en el bit de peso 0, denominado `ACKCLR`, del registro `RTC_SCCR` (*RTC Status Clear Command Register*) cuyo contenido se muestra en la Figura 8.11.

4. Ahora se puede escribir el nuevo valor de la hora y/o fecha actuales en sus correspondientes registros —`RTC_TIMR` y `RTC_CALR`

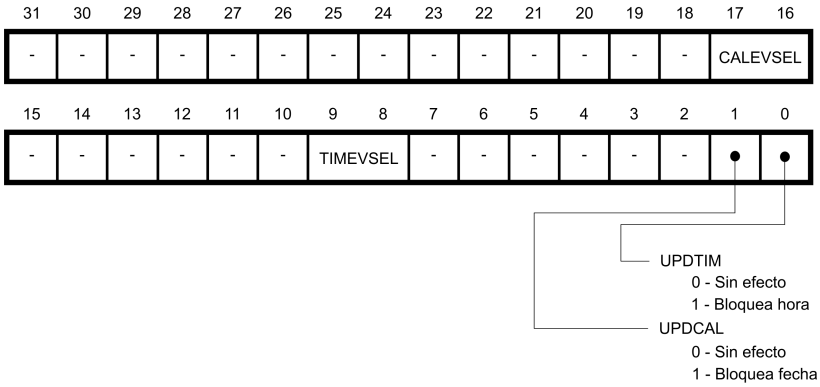


Figura 8.8: Contenido del registro RTC Control Register

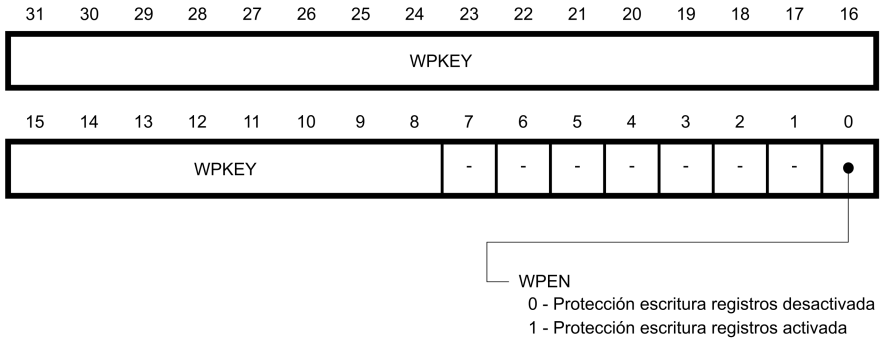


Figura 8.9: Contenido del registro RTC Write Protect Mode Register

respectivamente—. El RTC comprueba que los valores que se escriben sean correctos. De no ser así, se activa el indicador correspondiente en el `RTC_VER` (*RTC Valid Entry Register*) cuyo contenido se muestra en la Figura 8.12.

De esta forma, si alguno de los campos de la hora indicada no es correcto, se activará (presentando un valor `1`) el indicador `NVTIM` (*Non-valid Time*) y si uno o más de los campos de la fecha no es correcto, se activará el indicador `NVCAL` (*Non-valid Calendar*). El RTC quedará bloqueado mientras se mantenga esta situación y los indicadores solamente volverán a la normalidad cuando se introduzca un valor correcto.

5. Restablecer el valor de los bits de inhibición de la actualización

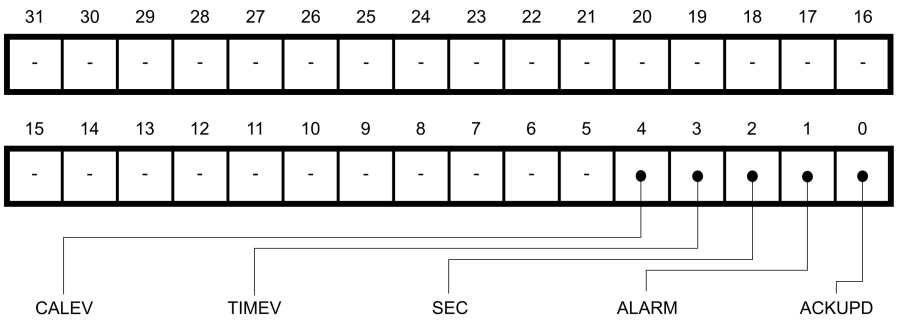


Figura 8.10: Contenido del registro RTC Status Register

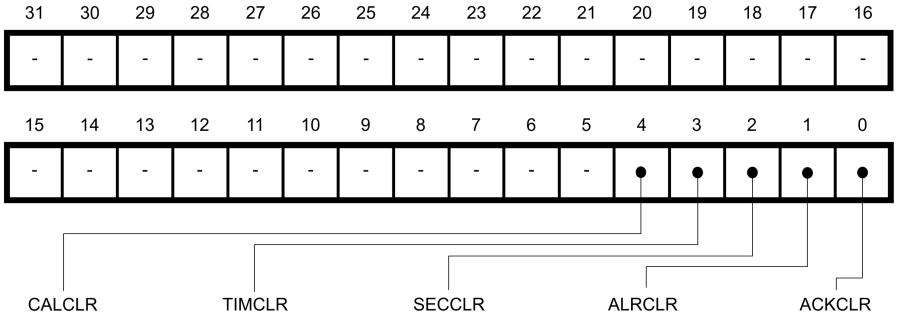


Figura 8.11: Contenido del registro RTC Status Clear Command Register

—`UPDCAL` y/o `UPDTIM`— para permitir la reanudación del funcionamiento del RTC. Si solamente se modifica el valor de la fecha actual, la porción del RTC dedicada al cálculo de la hora actual sigue en funcionamiento, mientras que si solo se modifica la hora, el calendario también es detenido. La modalidad de 12/24 horas se puede seleccionar mediante `HRMOD`, bit de peso 0 del registro `RTC_MR` (*Mode Register*) cuyo contenido se muestra en la Figura 8.13. Escribiendo en `HRMOD` el valor `1` se configura el RTC en modo 24 horas, mientras que el valor `0` establece la configuración en el modo de 12 horas.

Alarmas

El RTC posee la capacidad de establecer valores de alarma para cinco campos: mes, día del mes, hora, minuto, segundo. Estos valores están re-

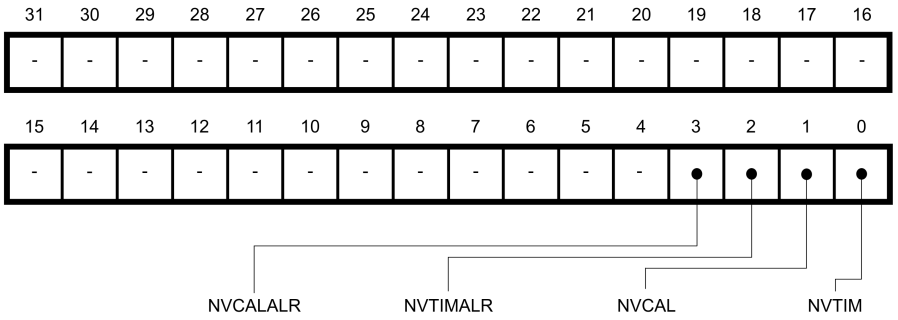


Figura 8.12: Contenido del registro RTC Valid Entry Register

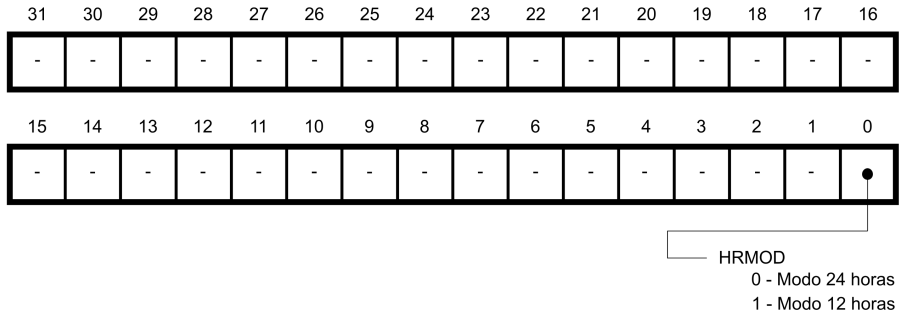


Figura 8.13: Contenido del registro RTC Mode Register

partidos en dos registros: `RTC_TIMALR` (*RTC Time Alarm Register*) cuyo contenido es mostrado en la Figura 8.14, y `RTC_CALALR` (*RTC Calendar Alarm Register*) cuyo contenido es mostrado en la figura 8.15.

Cada uno de los campos configurables posee un bit de activación asociado, de forma que su valor puede ser considerado o ignorado en la activación de la alarma. Así pues, si por ejemplo escribimos un `1` en `DATEEN` —bit 23 del registro `RTC_CALALR`— y el valor '18' en BCD en `DATE` —bits 16 a 20 del mismo registro— generaremos una alarma el día 18 de cada mes.

Los valores introducidos en los campos configurables se comprueban al igual que los de fecha y hora anteriormente comentados y, si se detecta un error, se activan los indicadores correspondientes del registro `RTC_VER` (*RTC Valid Entry Register*) mostrado en la Figura 8.12. Si se activan todos los campos configurables y se establece un valor válido para cada uno de ellos, se configura una alarma para un instante determinado, llegado el cual se activará el bit `ALARM` (bit 1 del registro `RTC_SR` (*RTC Status Register*) cuyo contenido se muestra en la Figura 8.10) y, en ca-

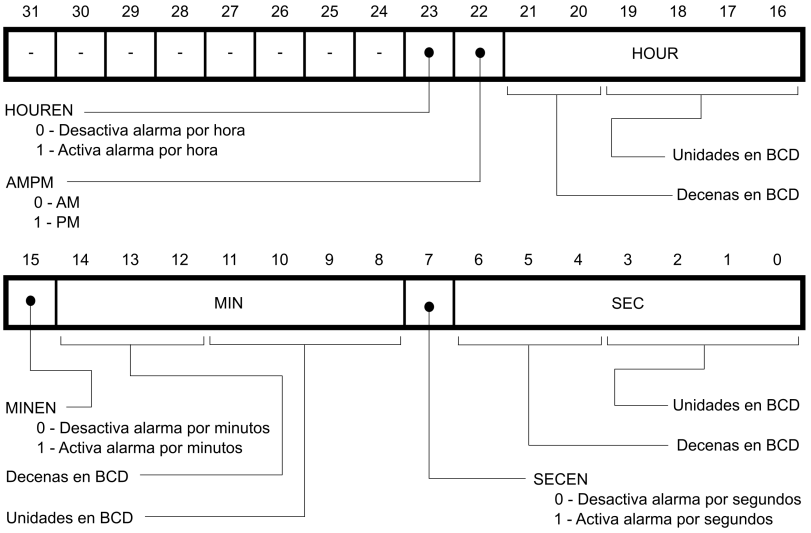


Figura 8.14: Contenido del registro RTC Time Alarm Register

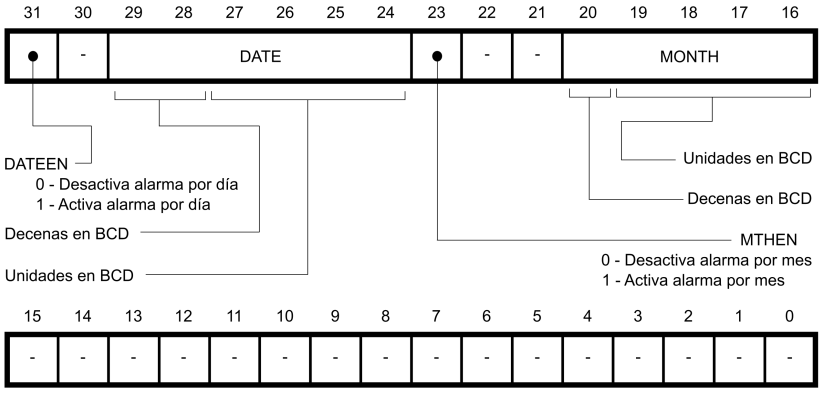


Figura 8.15: Contenido del registro RTC Calendar Alarm Register

so de estar activada la generación de interrupciones, se producirá una interrupción. Para restablecer los indicadores del registro `RTC_SR` (*RTC Status Register*) hay que escribir un `1` en cada uno de los bits correspondientes del registro `RTC_SCCR` (*RTC Status Clear Command Register*).

Si se produce una segunda alarma antes de que se haya leído el registro `RTC_SR` (*RTC Status Register*), mostrado en la Figura 8.10, tras una alarma, se activará `SEC` —bit de peso 2 del registro `RTC_SR`— que indica

que al menos dos alarmas se han producido desde que se restableció el valor del indicador por última vez.

Eventos periódicos

Además de las alarmas en instantes programados, como se ha visto en el apartado anterior, el RTC también posee la capacidad de producir alarmas periódicas con diferentes cadencias configurables a través del registro `RTC_CR` (*RTC Control Register*), cuyo contenido se muestra en la Figura 8.8. En sus bits 8 y 9 se encuentra el valor del campo `TIMEVSEL` que activa/desactiva la generación de eventos periódicos de hora y en los bits 16 y 17 el campo `CALEVSEL` que activa/desactiva la generación de eventos periódicos de calendario.

Un evento de hora puede ser a su vez de cuatro tipos diferentes, mostrados en el Cuadro 8.9, dependiendo del valor que tome el campo `TIMEVSEL`.

Valor	Nombre	Evento
0	`MINUTE`	Cada cambio de minuto
1	`HOUR`	Cada cambio de hora
2	`MIDNIGHT`	Cada día a medianoche
3	`NOON`	Cada día a mediodía

Cuadro 8.9: Tipos de eventos periódicos de hora

De la misma forma, un evento de fecha puede ser a su vez de tres tipos diferentes, mostrados en el Cuadro 8.10, dependiendo del valor que tome el campo `CALEVSEL`.

Valor	Nombre	Evento
0	`WEEK`	Cada lunes a las 0:00:00
1	`MONTH`	El día 1 de cada mes a las 0:00:00
2	`YEAR`	Cada 1 de enero a las 0:00:00
3	—	Valor no permitido

Cuadro 8.10: Tipos de eventos periódicos de fecha

Al igual que ocurre con las alarmas de tiempo concreto, la notificación de que se ha producido un evento periódico se produce a través del registro `RTC_SR` (*RTC Status Register*) mostrado en la Figura 8.10, donde, en caso de que se haya producido un evento periódico de hora, se activará `TIMEV` —bit de peso 3— y en caso de que se haya detectado

un evento periódico de fecha de acuerdo con lo configurado, se activará `CALEV` —bit de peso 4—.

Al leer este registro, el hecho de que uno o más de estos bits estén activos, es decir, que presenten el valor `1`, nos indicará que la condición de evento periódico se ha producido al menos en una ocasión desde la última vez que se leyó el contenido del registro. La lectura del registro restablece el valor de todos sus indicadores a `0`.

Interrupciones en el RTC

El RTC posee la capacidad de generar interrupciones cuando se producen una serie de circunstancias:

- Actualización de fecha/hora.
- Evento de tiempo.
- Evento de calendario.
- Alarma.
- Segundo evento de alarma periódica.

Para gestionar la generación de estas interrupciones y la atención de las mismas, existen los registros de interrupción del RTC, que a continuación se describen.

La activación de la generación de interrupciones se consigue a través del registro `RTC_IER` (*RTC Interrupt Enable Register*), cuyo contenido se muestra en la Figura 8.16, donde puede apreciarse que se dispone de cinco bits de configuración para activar la generación de interrupciones:

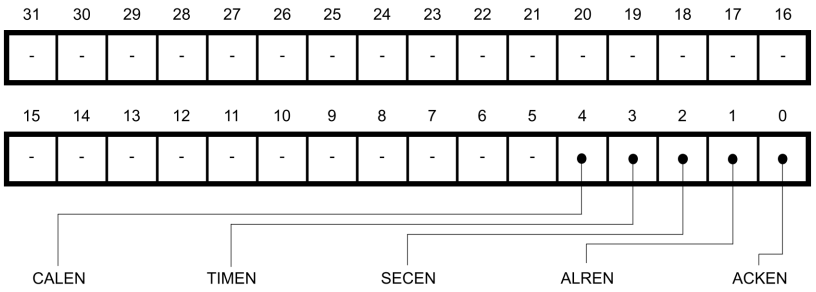


Figura 8.16: Contenido del registro RTC Interrupt Enable Register

- *Inhibición de la actualización del RTC*: escribiendo un `1` en `ACKEN` —bit de peso 0— activamos la generación de una interrupción cuando se active el bit `ACKUPD` del registro `RTC_SR` (*RTC Status Register*), mostrado en la Figura 8.10, como consecuencia de haber inhibido la actualización del RTC mediante uno de los bits `UPDCAL` o `UPDTIM` del registro `RTC_CR` —véase la Figura 8.8— o ambos.

- *Condición de alarma*: escribiendo un `1` en `ALREN` —bit de peso 1— activamos la generación de una interrupción al activarse el bit `ALARM` del registro `RTC_SR` (*RTC Status Register*). Esto ocurre cuando se cumple la condición de generación de alarma especificada en uno de los registros `RTC_TIMALR` (*RTC Time Alarm Register*) o `RTC_CALALR` (*RTC Calendar Alarm Register*).

- *Segunda alarma*: escribiendo un `1` en `SECEN` —bit de peso 2—, activamos la generación de una interrupción al activarse el bit `SEC` del registro `RTC_SR` (*RTC Status Register*), lo cual indica que se ha cumplido la condición de alarma en una segunda ocasión sin que el registro *RTC Status Register* haya sido leído.

- *Evento periódico de hora*: escribiendo un `1` en `TIMEN` —bit de peso 3— se activa la generación de una interrupción cuando se activa el bit `TIMEV` del registro `RTC_SR`, lo cual ocurrirá cuando se cumpla la condición periódica configurada en el campo `TIMEVSEL` del registro `RTC_CR` ((*RTC Control Register*) —véanse la Figura 8.8 y el Cuadro 8.9—.

- *Evento periódico de fecha*: escribiendo un `1` en `CALEN` —bit de peso 4— se activa la generación de una interrupción cuando se activa el bit `CALEV` del registro `RTC_SR` lo cual ocurrirá cuando se cumpla la condición periódica configurada en el campo `CALEVSEL` del registro `RTC_CR` (*RTC Control Register*) —véanse la Figura 8.8 y el Cuadro 8.10—.

Cada vez que se produzca una interrupción, en la correspondiente rutina de servicio se deberá acceder al registro `RTC_SR` (*RTC Status Register*) para averiguar cuál es la causa de la misma. Es posible que varias circunstancias hayan concurrido para la generación de la interrupción, con lo cual es aconsejable comprobar todos y cada uno de los bits del registro de estado. Una vez averiguadas las causas de la interrupción y tomadas las acciones pertinentes, antes de regresar de la rutina de servicio, se deben restablecer los indicadores escribiendo ceros en el registro `RTC_SCCR` (*RTC Status Clear Command Register*), cuyo contenido se muestra en la Figura 8.11, para dejar el sistema en disposición de que se produzcan nuevas interrupciones.

8.2.4. El Real-Time Timer (RTT) del Atmel ATSAM3X8E

El *Real-Time Timer* (RTT) es un temporizador del ATSAM3X8E simple y versátil, por lo que se puede utilizar de forma sencilla —y con plena libertad dado que no es utilizado por el sistema Arduino—. Se trata básicamente de un registro contador de 32 bits, que tiene una frecuencia base de actualización de 32.768Hz, como el RTC. Esta frecuencia se puede dividir gracias a un prescaler de 16 bits. El dispositivo dispone además de un registro de alarma para generar interrupciones cuando la cuenta del temporizador alcanza el valor almacenado en él, y es capaz además de generar interrupciones periódicas cada vez que se incrementa el valor del temporizador. Utiliza los cuatro registros que se describen a continuación:

- *Mode Register* (`RTT_MR`): es el registro de control del dispositivo. Los 16 bits más bajos almacenan el prescaler. Con un valor de `0x8000` se tiene una frecuencia de actualización de un segundo, lo que indica que está pensado para temporizaciones relacionadas con tiempos de usuario más que de sistema. No obstante, se puede poner en estos bits cualquier valor superior a 2. El bit 18 —`RTTRST`— sirve para reiniciar el sistema —escribiendo un `1`—, poniendo el contador a `0` y actualizando el valor del prescaler. El bit 17 —`RTTINCIEN`— es la habilitación de interrupción por incremento, y el bit 16 —`ALMIEN`— la de interrupción por alarma. Ambas se habilitan con un `1`.

- *Alarm Register* (`RTT_AR`): almacena el valor de la alarma, de 32 bits. Cuando el contador alcance este valor, se producirá una interrupción en caso de estar habilitada.

- *Value Register* (`RTT_VR`): guarda el valor del contador, que se va incrementando con cada pulso —según la frecuencia base dividida por el prescaler—, de forma cíclica.

- *Status Register* (`RTT_SR`): es el registro de estado. Su bit 1 —`RTTINC`— indica que se ha producido un incremento del valor, y su bit 0 —`ALMS`— que ha ocurrido una alarma. Ambas circunstancias se señalan con un `1`, que se pone a `0` al leer el registro.

En el Cuadro 8.11 aparecen las direcciones de los registros citados.

Es interesante realizar un comentario acerca del uso del RTT y de la forma en que se generan las interrupciones. La interrupción por incremento está pensada para generar una interrupción periódica; según el valor del prescaler, el periodo puede ser desde inferior a una décima de milisegundo hasta casi dos segundos. Eléctricamente la interrupción

Registro	Alias	Dirección
Mode Register	RTT_MR	0x400E 1A30
Alarm Register	RTT_AR	0x400E 1A34
Value Register	RTT_VR	0x400E 1A38
Status Register	RTT_SR	0x400E 1A3C

Cuadro 8.11: Registros del temporizador en tiempo real del ATSAM3X8E y sus direcciones de E/S

se genera por flanco, por lo que al producirse basta con leer el RTT_SR para evitar que se genere hasta el próximo incremento, comportamiento típico de una interrupción periódica.

La interrupción de alarma, sin embargo, se produce por nivel mientras el valor del contador sea igual al de la alarma. Leer en este caso el RTT_SR no hace que deje de señalarse la interrupción, que se seguirá disparando hasta que llegue otro pulso de la frecuencia de actualización. Este comportamiento no se evita cambiando el valor del RTT_AR pues la condición de interrupción se actualiza con el mismo reloj que incrementa el temporizador. Esto quiere decir que la interrupción por alarma está pensada para producirse una vez y deshabilitarse, hasta que el programa decida configurar una nueva alarma por algún motivo. Su uso como interrupción periódica no es, por tanto, recomendable.

8.3. Gestión de excepciones e interrupciones en el ATSAM3X8E

La arquitectura ARM especifica un modelo de excepciones que lógicamente incluye el tratamiento de interrupciones. Es un modelo elaborado y versátil, pero a la vez sencillo de usar, dado que la mayor parte de los mecanismos son llevados a cabo de forma automática por el hardware del procesador.

Se trata de un modelo vectorizado, con expulsión —*preemption*— en base a prioridades. Cada causa de excepción, de las que las interrupciones son un subconjunto, tiene asignado un número de orden que identifica un vector de excepción en una zona determinada de la memoria. Cuando se produce una excepción, el procesador carga el valor almacenado en ese vector y lo utiliza como dirección de inicio de la rutina de tratamiento. Al mismo tiempo, cada excepción tiene una prioridad que determina cuál de ellas será atendida en caso de detectarse varias a la vez, y permite que una rutina de tratamiento sea interrumpida —expulsada— si se detecta una excepción con mayor prioridad, volviendo

a aquélla al terminar de tratar la más prioritaria. El orden de la prioridad es inverso a su valor numérico, así una prioridad de 0 es mayor que una de 7, por ejemplo.

De acuerdo con este modelo, una excepción es marcada como *pendiente —pending—* cuando ha sido detectada, pero no ha sido tratada todavía, y como *activa —active—* cuando su rutina de tratamiento ya ha comenzado a ejecutarse. Debido a la posibilidad de expulsión, en un momento puede haber más de una excepción activa en el sistema. Cuando una excepción es a la vez pendiente y activa, ello significa que se ha vuelto a detectar la excepción mientras se trataba la anterior.

Cuando se detecta una excepción, si no se está atendiendo a otra de mayor prioridad, el procesador guarda automáticamente en la pila los registros `r0` a `r3` y `r12`; la dirección de retorno, el registro de estado y el registro `lr`. Entonces realiza el salto a la dirección guardada en el vector de interrupción y pasa la excepción de pendiente a activa. En el registro `lr` se escribe entonces un valor especial, llamado `EXC_RETURN`, que indica que se está tratando una excepción, y que será utilizado para volver de la rutina de tratamiento. La rutina de tratamiento tiene la estructura de una rutina normal, dado que los registros ya han sido salvados adecuadamente. Cuando se termina el tratamiento se retorna a la ejecución normal con una instrucción «`pop`» que incluya el registro `pc`. De esta manera, el procesador recupera de forma automática los valores de los registros que había guardado previamente, continuando con la ejecución de forma normal.

En el Cuadro 8.12 aparecen las excepciones del sistema, con su número, su prioridad y el número de interrupción asociado. Por conveniencia, la arquitectura asigna a las excepciones un número de interrupción negativo, quedando el resto para las interrupciones de dispositivos.

8.3.1. El Nested Vectored Interrupt Controller NVIC

Como hemos visto en el apartado anterior, la arquitectura ARM especifica el modelo de tratamiento de las excepciones. Del mismo modo, incluye entre los dispositivos periféricos del núcleo —*Core Peripherals*— de la versión *Cortex-M3* de dicha arquitectura, un controlador de interrupciones llamado *Nested Vectored Interrupt Controller*, *NVIC*. Cada implementación distinta de la arquitectura conecta al NVIC las interrupciones generadas por sus distintos dispositivos periféricos. En el caso del ATSAM3X8E, dichas conexiones son fijas, de manera que cada dispositivo tiene un número de interrupción —y por tanto, un vector— fijo y conocido de antemano.

El NVIC, además de implementar el protocolo adecuado para señalar las interrupciones al núcleo, contiene una serie de registros que permiten al software del sistema configurar y tratar las interrupciones según las

Excepción	IRQ	Tipo	Prioridad	Vector (offset)
1	—	Reset	−3	`0x0000 0004`
2	−14	NMI	−2	`0x0000 0008`
3	−13	Hard fault	−1	`0x0000 000C`
4	−12	Memory management fault	Configurable	`0x0000 0010`
5	−11	Bus fault	Configurable	`0x0000 0014`
6	−10	Usage fault	Configurable	`0x0000 0018`
11	−5	SVCall	Configurable	`0x0000 002C`
14	−2	PendSV	Configurable	`0x0000 0038`
15	−1	SysTick	Configurable	`0x0000 003C`
16	0	IRQ0	Configurable	`0x0000 0040`
17	1	IRQ1	Configurable	`0x0000 0044`
...	...	...	...	...

Cuadro 8.12: Algunas de las excepciones del ATSAM3X8E y sus vectores de interrupción

necesidades de la aplicación. Para ello, dispone de una serie de registros especificados en la arquitectura, que en el caso del ATSAM3X8E permiten gestionar las interrupciones de los periféricos del microcontrolador. Veamos cuáles son esos registros y su función.

Para la habilitación individual de las interrupciones se dispone de los conjuntos de registros:

- *Interrupt Set Enable Registers* (`ISERx`): escribiendo un `1` en cualquier bit de uno de estos registros se habilita la interrupción asociada.
- *Interrupt Clear Enable Registers* (`ICERx`): escribiendo un `1` en cualquier bit de uno de estos registros se deshabilita la interrupción asociada.

Al leer cualquiera de los registros anteriores se obtiene la máscara de interrupciones habilitadas, indicadas con `1` en los bits correspondientes.

Para gestionar las interrupciones pendientes se tienen:

- *Interrupt Set Pending Registers* (`ISPRx`): escribiendo un `1` en cualquier bit de uno de estos registros se marca la interrupción aso-

ciada como pendiente. Esto permite forzar el tratamiento de una interrupción aunque no se haya señalado físicamente.

- *Interrupt Clear Pending Registers* (`ICPRx`): escribiendo un `1` en cualquier bit de uno de estos registros se elimina la interrupción asociada del estado pendiente. Esto permite evitar que se produzca el salto por hardware a la rutina de tratamiento.

Al leer cualquiera de los registros anteriores se obtiene la lista de interrupciones pendientes, indicadas con `1` en los bits correspondientes. Una interrupción puede estar en estado pendiente aunque no esté habilitada en la máscara vista más arriba.

La lista de interrupciones activas se gestiona por hardware, pero se puede consultar en los registros de solo lectura *Interrupt Active Bit Registers* (`ICPRx`). En ellos, un `1` indica que la interrupción correspondiente está siendo tratada en el momento de la lectura. Las prioridades asociadas a las interrupciones se configuran en los registros *Interrupt Priority Registers* (`IPRx`). Cada interrupción tiene asociado un campo de 8 bits en estos registros, aunque en la implementación actual solo los 4 de mayor peso almacenan el valor de la prioridad, entre 0 y 15.

Por último, existe un registro que permite generar interrupciones por software, llamado *Software Trigger Interrupt Register* (`STIR`). Escribiendo un valor en sus 9 bits menos significativos se genera la interrupción con dicho número.

Para dar cabida a los bits asociados a todas las posibles interrupciones —que pueden llegar a ser hasta 64 en la serie de procesadores SAM3X—, el NVIC implementado en el ATSAM3X8E dispone de 2 registros para cada uno de los conjuntos, excepto para el de prioridades que comprende 8 registros. Dado un número de IRQ es sencillo calcular cómo encontrar los registros y bits que almacenan la información que se desea leer o modificar, sin embargo ARM recomienda utilizar ciertas funciones en lenguaje C que suministra —en el modelo de código del sistema llamado *CMSIS*— y que son las que utilizaremos en nuestros programas. A continuación se describen las primitivas más usadas.

- «`void NVIC_EnableIRQ(IRQn_t IRQn)`». Habilita la interrupción cuyo número se le pasa como parámetro.

- «`void NVIC_DisableIRQ(IRQn_t IRQn)`». Deshabilita la interrupción cuyo número se le pasa como parámetro.

- «`uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)`». Devuelve un valor distinto de `0` si la interrupción cuyo número se pasa está pendiente; `0` en caso contrario.

- «`void NVIC_SetPendingIRQ (IRQn_t IRQn)`». Marca como pendiente la interrupción cuyo número se pasa como parámetro.
- «`void NVIC_ClearPendingIRQ (IRQn_t IRQn)`». Elimina la marca de pendiente de la interrupción cuyo número se pasa como parámetro.
- «`void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)`». Asigna la prioridad indicada a la interrupción cuyo número se pasa como parámetro.
- «`uint32_t NVIC_GetPriority (IRQn_t IRQn)`». Devuelve la prioridad de la interrupción cuyo número se pasa como parámetro.

8.3.2. Interrupciones del ATSAM3X8E en el entorno Arduino

Como se ha visto, buena parte de las tareas de salvaguarda y recuperación del estado en la gestión de interrupciones son realizadas por el hardware en las arquitecturas ARM. Gracias a esto el diseño de rutinas de tratamiento queda muy simplificado.

La estructura de una RTI es idéntica a la de una rutina normal, con la única restricción de que no admite ni devuelve parámetros. El hecho de que los registros que normalmente no se guardan —`r0 ... r3`— sean preservados automáticamente hace que el código de entrada y salida de una RTI no difiera del de una rutina, y por ello, las únicas diferencias entre ambas estriban en su propio código.

Teniendo en cuenta esta estructura, y teniendo en cuenta las funciones de gestión del NVIC que se han descrito más arriba, lo único que se necesita saber para implementar una RTI es el número de interrupción. Además, para configurarla habría que modificar el vector correspondiente para que apunte a nuestra función. De nuevo, en el estándar CMSIS se dan las soluciones para estos requisitos. Todos los dispositivos tienen definido su número de IRQ —que recordemos es fijo— de forma regular. Del mismo modo, los nombres de las funciones de tratamiento están igualmente predefinidos, de manera que para crear una RTI para un cierto dispositivo simplemente hemos de programar una función con el nombre adecuado. El proceso de compilación de nuestro programa se encarga de configurar el vector de manera transparente para el programador.

Los Cuadros 8.13, 8.14 y 8.15 muestran los símbolos que se deben usar para referirse a las distintas IRQ según el dispositivo, la descripción del dispositivo y el nombre de la rutina de tratamiento.

Una vez creada la RTI solo es necesario configurar el NVIC adecuadamente. El proceso suele consistir en deshabilitar primero la interrupción

Símbolo	Núm	Dispositivo	RTI
SUPC_IRQn	0	Supply Controller (SUPC)	«void SUPC_Handler()»
RSTC_IRQn	1	Reset Controller (RSTC)	«void RSTC_Handler()»
RTC_IRQn	2	Real Time Clock (RTC)	«void RTC_Handler()»
RTT_IRQn	3	Real Time Timer (RTT)	«void RTT_Handler()»
WDT_IRQn	4	Watchdog Timer (WDT)	«void WDT_Handler()»
PMC_IRQn	5	Power Management Controller (PMC)	«void PMC_Handler()»
EFC0_IRQn	6	Enhanced Flash Controller 0 (EFC0)	«void EFC0_Handler()»
EFC1_IRQn	7	Enhanced Flash Controller 1 (EFC1)	«void EFC1_Handler()»
UART_IRQn	8	Universal Asynchronous Receiver Transmitter	«void UART_Handler()»
SMC_IRQn	9	Static Memory Controller (SMC)	«void SMC_Handler()»
PIOA_IRQn	11	Parallel I/O Controller A, (PIOA)	«void PIOA_Handler()»
PIOB_IRQn	12	Parallel I/O Controller B (PIOB)	«void PIOB_Handler()»
PIOC_IRQn	13	Parallel I/O Controller C (PIOC)	«void PIOC_Handler()»
PIOD_IRQn	14	Parallel I/O Controller D (PIOD)	«void PIOD_Handler()»

Cuadro 8.13: IRQs del ATSAM3X8E y sus rutinas de tratamiento asociadas (Parte I)

correspondiente, limpiar el estado pendiente por si se produjo alguna falsa interrupción durante el arranque del sistema y establecer la prioridad —que por defecto suele ser `0` en el entorno Arduino—. Una vez hecho esto, hemos de configurar el dispositivo de la manera que se desee y habilitar la interrupción correspondiente.

A continuación se muestran unos fragmentos de código que permitirían establecer una RTI para el dispositivo `PIOB`.

RTI-PIOB.c

```
1 void setup() {
2         /* Otra configuración del sistema*/
```

Símbolo	Núm	Dispositivo	RTI
USART0_IRQn	17	USART 0 (USART0)	«**void** USART0_Handler()»
USART1_IRQn	18	USART 1 (USART1)	«**void** USART1_Handler()»
USART2_IRQn	19	USART 2 (USART2)	«**void** USART2_Handler()»
USART3_IRQn	20	USART 3 (USART3)	«**void** USART3_Handler()»
HSMCI_IRQn	21	Multimedia Card Interface (HSMCI)	«**void** HSMCI_Handler()»
TWI0_IRQn	22	Two-Wire Interface 0 (TWI0)	«**void** TWI0_Handler()»
TWI1_IRQn	23	Two-Wire Interface 1 (TWI1)	«**void** TWI1_Handler()»
SPI0_IRQn	24	Serial Peripheral Interface (SPI0)	«**void** SPI0_Handler()»
SSC_IRQn	26	Synchronous Serial Controller (SSC)	«**void** SSC_Handler()»
TC0_IRQn	27	Timer Counter 0 (TC0)	«**void** TC0_Handler()»
TC1_IRQn	28	Timer Counter 1 (TC1)	«**void** TC1_Handler()»
TC2_IRQn	29	Timer Counter 2 (TC2)	«**void** TC2_Handler()»
TC3_IRQn	30	Timer Counter 3 (TC3)	«**void** TC3_Handler()»
TC4_IRQn	31	Timer Counter 4 (TC4)	«**void** TC4_Handler()»
TC5_IRQn	32	Timer Counter 5 (TC5)	«**void** TC5_Handler()»
TC6_IRQn	33	Timer Counter 6 (TC6)	«**void** TC6_Handler()»
TC7_IRQn	34	Timer Counter 7 (TC7)	«**void** TC7_Handler()»
TC8_IRQn	35	Timer Counter 8 (TC8)	«**void** TC8_Handler()»

Cuadro 8.14: IRQs del ATSAM3X8E y sus rutinas de tratamiento asociadas(Parte II)

```
3         NVIC_DisableIRQ(PIOB_IRQn);
4         NVIC_ClearPendingIRQ(PIOB_IRQn);
5         NVIC_SetPriority(PIOB_IRQn, 0);
6         PIOsetupInt(EDGE, 1);
7         NVIC_EnableIRQ(PIOB_IRQn);
8 }
9
10 // RTI en C
11 void PIOB_Handler() {
12         /* Código específico del tratamiento */
```

Símbolo	Núm	Dispositivo	RTI
PWM_IRQn	36	Pulse Width Modulation Controller (PWM)	«void PWM_Handler()»
ADC_IRQn	37	ADC Controller (ADC)	«void ADC_Handler()»
DACC_IRQn	38	DAC Controller (DACC)	«void DACC_Handler()»
DMAC_IRQn	39	DMA Controller (DMAC)	«void DMAC_Handler()»
UOTGHS_IRQn	40	USB OTG High Speed (UOTGHS)	«void UOTGHS_Handler()»
TRNG_IRQn	41	True Random Number Generator (TRNG)	«void TRNG_Handler()»
EMAC_IRQn	42	Ethernet MAC (EMAC)	«void EMAC_Handler()»
CAN0_IRQn	43	CAN Controller 0 (CAN0)	«void CAN0_Handler()»
CAN1_IRQn	44	CAN Controller 1 (CAN1)	«void CAN1_Handler()»

Cuadro 8.15: IRQs del ATSAM3X8E y sus rutinas de tratamiento asociadas (Parte III)

```
13 }
```

RTI-PIOB.s

```
1 @ RTI en ensamblador
2 PIOB_Handler:
3         push {lr}
4         /* Código específico del tratamiento */
5         pop {pc}
```

8.4. El controlador de DMA del ATSAM3X8E

El *AHB DMA Controller* (*DMAC*) es el dispositivo controlador de acceso directo a memoria en el ATSAM3X8E. Se trata de un dispositivo con seis canales, con capacidad para almacenamiento intermedio de 8 o 32 bytes —en los canales 3 y 5— y que permite transferencias entre dispositivos y memoria, en cualquier configuración. Cada movimiento de información requiere de la lectura de datos de la fuente a través de los buses correspondientes, su almacenamiento intermedio y su posterior escritura en el destino, lo que requiere siempre dos accesos de transferencia de datos.

Además de un conjunto de registros de configuración globales, cada canal dispone de seis registros asociados que caracterizan la transacción.

Mediante estos registros, además de indicar el dispositivo fuente y destino y las direcciones de datos correspondientes, se pueden programar transacciones múltiples de bloques de datos contiguos o dispersos, tanto en la fuente como en el destino.

El dispositivo, además de gestionar adecuadamente los accesos a los distintos buses y dispositivos, es capaz de generar interrupciones para indicar posibles errores o la finalización de las transacciones de DMA programadas.

Para una información más detallada, fuera del objetivo de esta breve introducción, se puede consultar el *apartado 24. AHB DMA Controller (DMAC)* en la página 349 del manual de Atmel.

CAPÍTULO 9

Entrada/Salida: ejercicios prácticos con Arduino Due

Índice

9.1. El entorno Arduino

Arduino de Ivrea (955–1015) fue Rey de Italia entre 1002 y 1014. Massimo Banzi y un grupo de docentes del Interaction Design Institute en Ivrea, en Italia, desarrollaron una plataforma de hardware libre basada en un microcontrolador y un entorno de desarrollo diseñados para facilitar la realización de proyectos de electrónica. Banzi y su grupo se reunían habitualmente en el Bar del Rey Arduino, en la localidad de Ivrea, de ahí el nombre del sistema.

Arduino está compuesto por una plataforma de hardware libre y un entorno de desarrollo. A grandes rasgos, esto significa que el diseño está

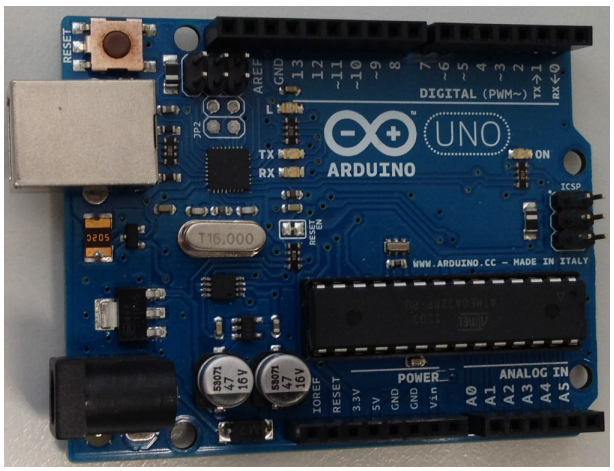


Figura 9.1: Tarjeta Arduino Uno

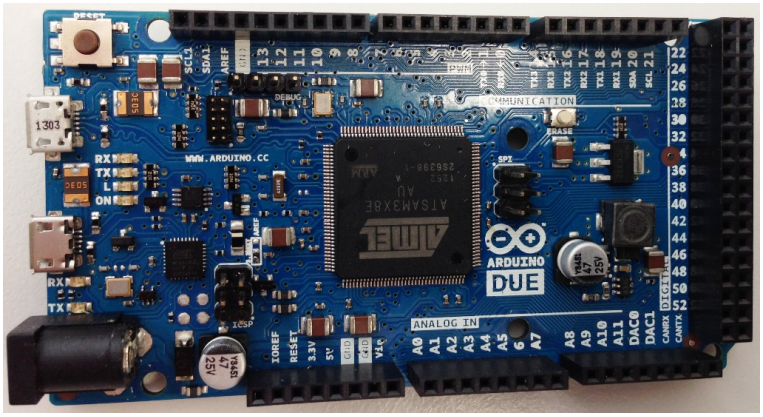


Figura 9.2: Tarjeta Arduino Due

a disposición de quien lo quiera emplear y modificar, dentro de unos límites de beneficio económico y siempre publicando las modificaciones introducidas.

Existen diferentes versiones de la arquitectura Arduino que emplean diversos microcontroladores respetando las dimensiones físicas de los conectores de ampliación y la ubicación de las señales en los mismos. El entorno de desarrollo introduce una capa de abstracción mediante la cual un conjunto de comandos y funciones puede ser empleado en cualquier plataforma Arduino.

En nuestro caso, usaremos la versión *Arduino Due* —véase la Figura 9.2— que, respecto de la tarjeta Arduino Uno original —mostrada

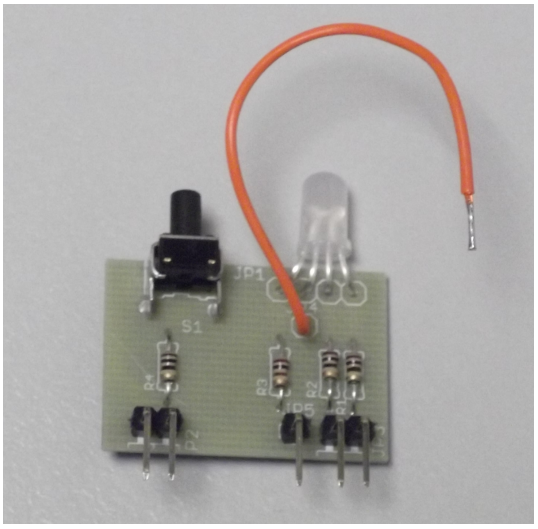


Figura 9.3: Tarjeta de E/S de prácticas de laboratorio

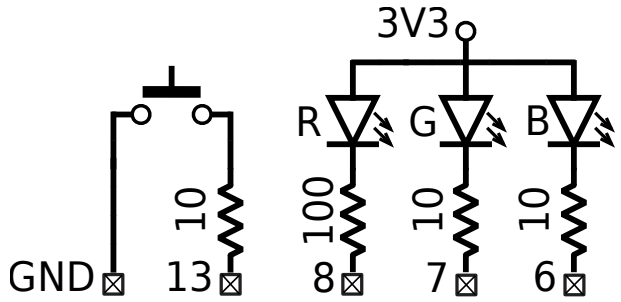


Figura 9.4: Esquema de la tarjeta de E/S de prácticas de laboratorio

en la Figura 9.1—, entre otras diferencias, presenta un microprocesador ATSAM3X8E, más potente que el ATmega328 de aquélla y con mayor número de entradas/salidas.

En la primera parte de las prácticas se ha utilizado el conjunto de instrucciones *Thumb* correspondiente a la versión Cortex-M0 de la arquitectura ARM. El microcontrolador ATMSAM3X8E de la tarjeta AruinoDue implementa la versión Cortex-M3 que utiliza un conjunto de instrucciones mayor, el *Thumb II*. Aunque todas las instrucciones *Thumb* están incluidas en *Thumb II* existe una diferencia crítica en el lenguaje ensamblador, que se señala aquí porque puede dar lugar a errores.

Las instrucciones aritméticas y lógicas del conjunto *Thumb II* permiten no modificar los indicadores de estado —*flags*— mediante cierto bit en su formato. Esta circunstancia se expresa en lenguaje ensambla-

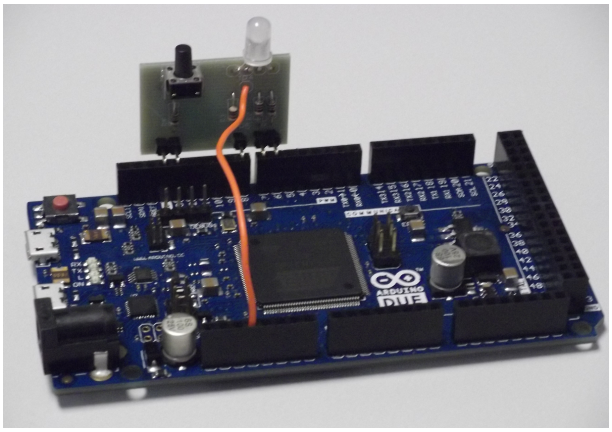


Figura 9.5: Tarjeta de E/S insertada en la Arduino Due

dor añadiendo una `s` en el código de instrucción cuando sí se vayan a modificar, de manera que se tiene:

```
and     r0, r1, r3   @ No modifica los flags
ands    r0, r1, r3   @ Sí los modifica
```

Si bien esta característica añade potencia al conjunto de instrucciones, es fácil confundirse cuando se está acostumbrado a programar con instrucciones *Thumb*, dado que todas afectan a los flags en general.

En el documento `CortexM3 Instruction Set` —disponible en el Aula Virtual de la asignatura EI1004/MT1004, sección *Teoría*, apartado *Tema 2 La Arquitectura ARM*, enlace *Cortex^{TM}-M3 Instruction Set*—, se describe el conjunto de instrucciones *Thumb II* completamente. Dado que es más potente, se recomienda consultarlo para conocer todas las posibilidades de este conjunto.

Para el desarrollo de las prácticas, se ha confeccionado una tarjeta específica —mostrada en la Figura 9.3— que contiene un pulsador y un LED RGB conectados como muestra la Figura 9.4 que se emplearán como dispositivos de E/S. La Figura 9.5 muestra la tarjeta instalada sobre la Arduino Due, donde puede apreciarse que, además de insertar la tarjeta correctamente, hay que conectar un cable de alimentación al pin de la Arduino Due rotulado con el texto `3.3V`. Como se puede ver en la Figura 9.4, el LED RGB es del tipo ánodo común, por lo que será necesario escribir un `0` en cada salida conectada a un LED para encenderlo y un `1` para mantenerlo apagado. Igualmente se puede ver cómo el pulsador se conecta a un pin y a masa. Se infiere, pues, que será necesario activar el `pull-up` de la salida 13 para leer y que en el estado no pulsado se obtendrá un `1` en el pin. Al pulsarlo, lógicamente,

cambiará a `0`.

El entorno de programación de Arduino —véase la Figura 9.6— se presenta como un editor de código en el cual podemos escribir nuestro programa, guardarlo, compilarlo y subirlo al dispositivo Arduino que tengamos conectado.

Figura 9.6: Entorno de programación Arduino

La estructura de un programa típico de Arduino consta de dos funciones:

- «`void setup()`»: Contiene el conjunto de acciones que se realizarán al inicio de nuestro programa. Aquí habitualmente configuraremos las entradas/salidas que vayamos emplear y daremos valores iniciales a las variables.

- «`void loop()`»: Contiene el código que se ejecutará indefinidamente.

Es necesario especificar al entorno el modelo de sistema Arduino que tenemos conectado para que se usen las bibliotecas adecuadas y se asignen correctamente las señales de E/S, se acceda correctamente a los registros del procesador, etcétera. Para ello emplearemos la entrada

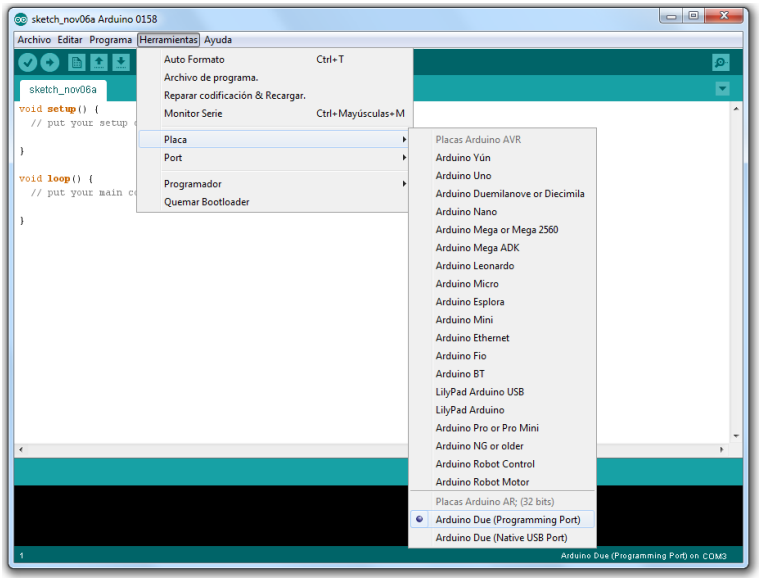


Figura 9.7: Selección del sistema Arduino a emplear en entorno Windows

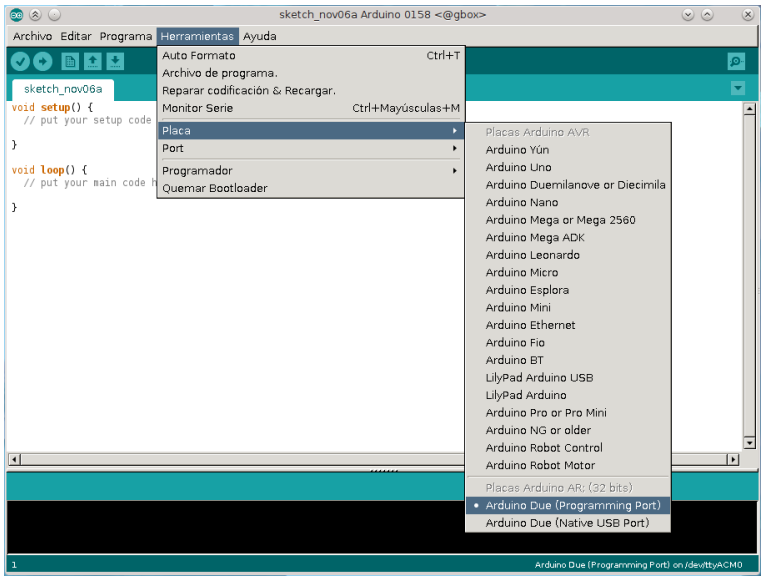


Figura 9.8: Selección del sistema Arduino a emplear en entorno Linux

«`Placa`» dentro del menú «`Herramientas`». En nuestro caso seleccionaremos la opción «`Arduino Due (Programming Port)`» —véase la Figura 9.7 para el entorno Windows y la Figura 9.8 para el entorno Linux—. Los

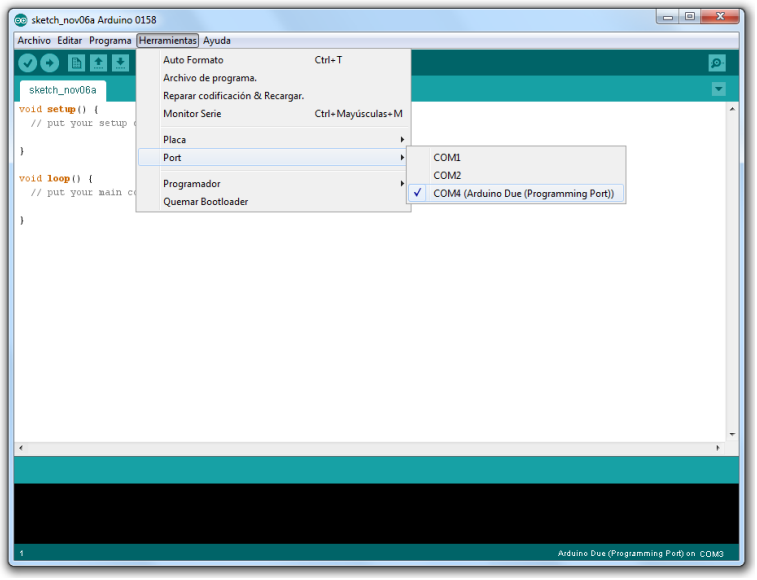


Figura 9.9: Selección del puerto de comunicaciones en entorno Windows

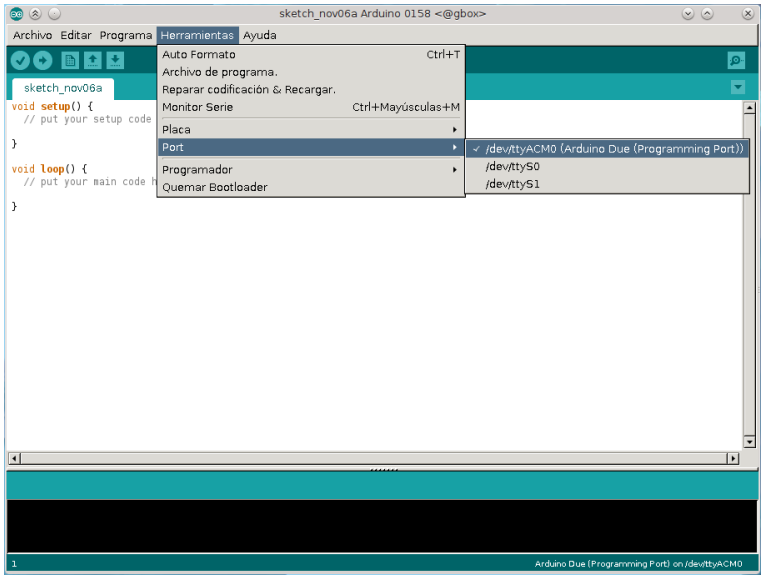


Figura 9.10: Selección del puerto de comunicaciones en entorno Linux

dos puertos USB de la tarjeta Arduino Due están identificados en la cara de soldaduras de la misma.

De la misma forma, hay que indicar el puerto serie de comunica-

ciones en que está conectado el sistema Arduino, lo cual especificaremos mediante la entrada «`Port`» dentro del menú «`Herramientas`», como muestran la Figura 9.9 para el entorno Windows y la Figura 9.10 para el entorno Linux. Como opciones se nos ofrecerán los puertos serie —`/dev/ttyx` o `COMx` en Linux o Windows respectivamente— de que disponga el sistema. Tanto en el entorno Windows como en el entorno Linux aparece como opción un puerto —que es donde se encontrará conectada nuestra tarjeta Arduino Due— junto a cuyo nombre aparecerá el texto *(Arduino Due (Programming Port))* y es, por tanto, el que debemos seleccionar.

9.2. Creación de proyectos

El entorno Arduino posee una estructura denominada *proyecto — sketch* en la bibliografía Arduino— que contiene los archivos correspondientes a cada programa. Cuando se inicia el entorno, se nos muestra un proyecto vacío con un nombre del tipo «`sketch_mmmdda`» —donde «`mmm`» es la abreviatura del nombre del mes actual y «`dd`» es el día del mes— que podemos usar como base para desarrollar un nuevo programa. De la misma forma, mediante la entrada «`Abrir...`» dentro del menú «`Archivo`» podemos abrir un proyecto existente.

Los archivos que componen un proyecto se guardan en una carpeta cuyo nombre coincide con el del proyecto. Generalmente un proyecto consta de un solo archivo de extensión «`.ino`» —con el mismo nombre que la carpeta— que contiene el código en lenguaje «`C / C++`» del programa principal.

Un programa puede, sin embargo, constar de más de un archivo. En tal caso, para añadir archivos al proyecto emplearemos el botón del entorno marcado con un recuadro en la esquina superior derecha en la Figura 9.6, que desplegará un menú del cual elegiremos la opción `Nueva Pestaña`. Al hacerlo, en la parte inferior del entorno aparecerá una barra en la que se nos solicitará el nombre de la nueva pestaña —y por tanto del archivo en que se guardará su contenido— donde deberemos especificar tanto el nombre como la extensión de dicho archivo y crear tanto la pestaña como el archivo pinchando en el botón `Ok`.

La versión del entorno que se va a usar en las prácticas ha sido modificada por el profesorado de la asignatura EI1004 para permitir el uso de archivos fuente en ensamblador. Las instrucciones para instalar esta versión se encuentran en el Aula Virtual de dicha asignatura, sección *Laboratorio*, apartado *Material adicional prácticas con Arduino*, enlace *Guías de instalación de Arduino (versión UJI).*

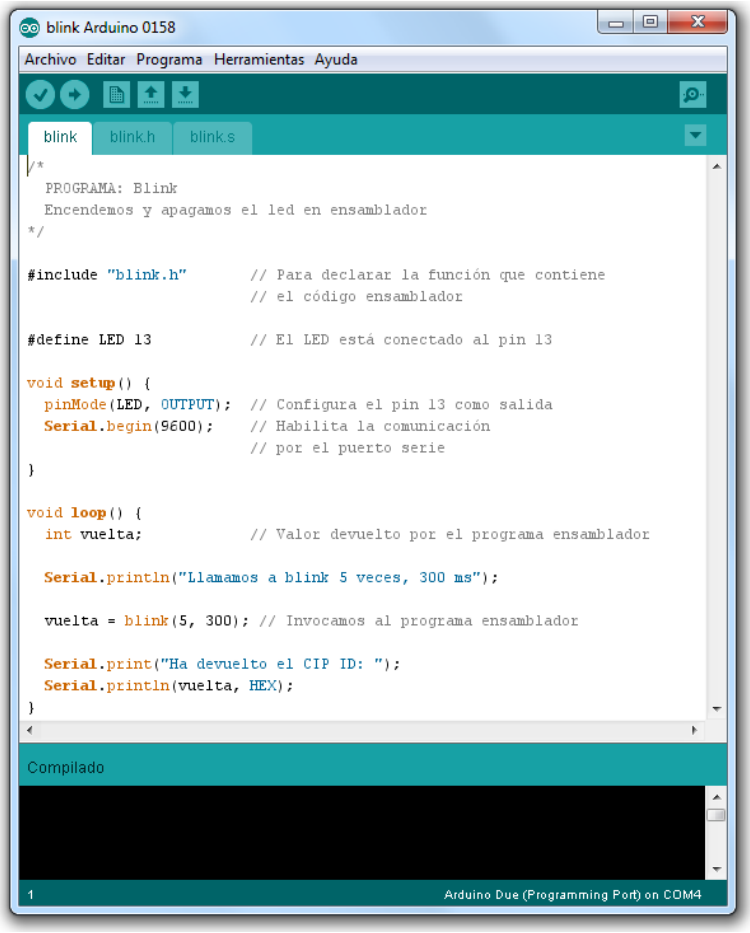


Figura 9.11: Entorno Arduino con el programa «blink» cargado

9.2.1. Ejemplo

Como ejemplo básico vamos a mostrar un programa que hace parpadear el LED incorporado en la tarjeta Arduino. De momento no es necesario que la tarjeta esté conectada al computador, con lo que esta fase se puede llevar a cabo sin necesidad de poseer una tarjeta Arduino. El código es el mostrado en la Figura 9.11:

Una vez introducido el código como texto, podemos comprobar que es correcto —lo cual implica *compilarlo*— mediante el botón de la parte izquierda de la barra de botones —marcado con un círculo en la Figura 9.6 y que hace aparecer el texto `Verificar` (tanto en Linux como en Windows) cuando situamos el cursor sobre él—. Tras el proceso de compilación se mostrarán los posibles errores detectados indicando el número de línea en que se encuentra cada uno de ellos o, si no hay errores,

Figura 9.12: Resultado de la compilación del programa «blink»

la cantidad de memoria ocupada por el código generado y el máximo de que dispone la tarjeta Arduino seleccionada actualmente, como puede apreciarse en la Figura 9.12.

Mediante el segundo botón de la barra de botones —marcado con un cuadrado en la Figura 9.6 y que hace aparecer el texto `Subir` (en Windows) o `Cargar` (en Linux) cuando situamos el cursor sobre él— se desencadena el mismo proceso pero, si en la compilación no se han producido errores, el código generado es enviado a la tarjeta Arduino —que ahora sí debe estar conectada al computador— y ejecutado de inmediato. De hecho, la programación se verifica comunicando el ordenador con la tarjeta y forzando a que se ejecute un programa especial llamado *bootloader*. Este programa lee del puerto USB-serie las instrucciones a cargar en la ROM. Una vez terminada la comunicación, el ordenador

fuerza un RESET y el microcontrolador comienza a ejecutar el programa descargado.

Siguiendo el procedimiento descrito se puede programar el microcontrolador empleando el lenguaje de alto nivel `C / C++` y las funciones específicas de Arduino. Nuestro objetivo, sin embargo, es programar el microcontrolador empleando directamente su lenguaje ensamblador y para conseguirlo vamos a introducir unas ligeras modificaciones en el código.

En primer lugar, escribiremos nuestro programa ensamblador en un archivo con la extensión `.s` para identificarlo como código ensamblador. En el caso del programa `blink` que hace parpadear el LED de Arduino, el código ensamblador correspondiente será:

blink.s

```
# blink.s - Parpadeo en ensamblador
# Acceso al controlador del PIO B

.syntax unified
.cpu cortex-m3
.text
.align  2
.thumb
.thumb_func
.extern delay                   @ No es necesario
.global blink                   @ Función externa
.type blink, %function

.equ PIOB,      0x400E1000      @ Dir. base del puerto B
.equ SODR,      0x030           @ OFFSET Set Output Data Reg
.equ CODR,      0x034           @ OFFSET Clear Output Data Reg
.equ CHIPID,    0x400E0940      @ Registro CHIP ID
.equ LEDMSK,    0x08000000      @ El LED está en el pin 27

/* int blink(int times, int delay)
     r0 = times. Número de veces que parpadea
     r1 = delay. Retardo del parpadeo
     Devuelve el CHIP_ID, porque sí
     Los parámetros se pasan en r0-r3
     El valor devuelto en r0 ó r0-r1 si ocupa 8 bytes
     Cualquier función puede modificar r0-r3
     El resto se han de preservar */

blink:
   push         {r4-r7, lr}     @ Vamos a usar de r4 a r7
                                @ porque llamamos a delay
   mov          r4, r0          @ r4 contiene el número de veces
   mov          r5, r1          @ r5 contiene el retardo a pasar a delay
```

```
34    ldr         r6, =PIOB       @ Dirección base del Controlador PIO B
35    ldr         r7, =LEDMSK     @ Máscara con el bit 27 a 1 (pin del LED)
36 principio:
37    str         r7, [r6, #SODR] @ Encendemos el LED escribiendo en SET
38    mov         r0, r5          @ Preparamos el parámetro de delay en r0
39    bl          delay           @ Invocamos a la función delay
40    str         r7, [r6, #CODR] @ Apagamos el LED escribiendo en CLEAR
41    mov         r0, r5          @ Volvemos a llamar a delay como antes
42    bl          delay
43    subs        r4, r4, #1      @ Decrementamos el número de veces
44    bne         principio       @ y si no es cero seguimos. ¡Ojo a la s!
45    ldr         r6, =CHIPID     @ Leemos CHIPID_CIDR
46    ldr         r0, [r6]        @ y devolvemos el valor en r0
47    pop         {r4-r7, pc}     @ ret con pop al pc.
48 .end
```

Para poder ejecutar este código desde el entorno de programación de Arduino es necesario indicar durante el proceso de compilación que se utilizan funciones en otro módulo, y que siguen el convenio de llamada de funciones de lenguaje `C`, ligeramente distinto del de `C++`. Para ello emplearemos un fichero de cabecera con extensión `.h` que llamaremos `blink.h` y cuyo contenido será:

blink.h ✍

```
1 // Declaración de las funciones externas
2
3 extern "C" {
4   int blink(int times, int del);
5   }
```

Este código define una función llamada `blink` que acepta dos argumentos. El primer argumento indica el número de veces que se desea que el LED parpadee y el segundo argumento el periodo de tiempo en milisegundos que el LED permanecerá encendido y apagado en cada ciclo de parpadeo, es decir, el ciclo completo tendrá una duración del doble de milisegundos que el valor de este argumento.

Finalmente, el programa principal se encarga de definir los valores iniciales de las variables y de invocar el código en ensamblador que efectivamente hará parpadear el LED.

blink.ino ✍

```
1 /*
2   PROGRAMA: Blink
3   Encendemos y apagamos el led en ensamblador
4 */
5
6 #include "blink.h"       // Para declarar la función que
```

```
7                          // contiene el código ensamblador
8
9  #define LED 13          // El LED está conectado al pin 13
10
11 void setup() {
12   pinMode(LED, OUTPUT);  // Configura el pin 13 como salida
13   Serial.begin(9600);    // Habilita la comunicación por el puerto serie
14   int vuelta;            // Valor devuelto por el programa ensamblador
15 }
16
17 void loop() {
18
19   Serial.println("Llamamos␣a␣blink␣5␣veces,␣300␣ms");
20
21   vuelta = blink(5, 300); // Invocamos el programa ensamblador
22
23   Serial.print("Ha␣devuelto␣el␣CIP␣ID:␣");
24   Serial.println(vuelta, HEX);
25 }
```

En este programa podemos, además, señalar que se ha hecho uso de la comunicación serie incorporada en la plataforma Arduino para obtener mensajes durante la ejecución del programa. Para ello hay que activar esta funcionalidad dentro de «`setup()`» mediante la llamada a la función «`Serial.begin(9600)`», donde el argumento indica la velocidad de comunicación en baudios. Posteriormente, ya dentro de la función «`loop`», se pueden enviar mensajes a través del puerto serie —asociado al USB— empleando las funciones «`Serial.print`» —muestra el texto que se le pasa como argumento y permite seguir escribiendo en la misma línea— y «`Serial.println`» —muestra el texto y pasa a la línea siguiente—. El argumento de estas funciones puede ser una cadena de caracteres entre comillas —que se mostrará textualmente— o una variable, en cuyo caso se mostrará su valor. En la página *www.arduino.cc* —o desde el propio entorno— se puede acceder a la referencia para obtener información sobre las funciones de Arduino. Para visualizar los mensajes recibidos hay que iniciar el *Monitor Serie*, lo cual se consigue pinchando sobre el botón del extremo derecho de la barra de botones —que contiene el icono de una lupa y que hace aparecer el texto `Monitor Serie` (en Windows) o `Monitor Serial` (en Linux) cuando situamos el cursor sobre él—. Hay que tener en cuenta que al iniciar el *Monitor Serie* se envía a la tarjeta Arduino una señal de *RESET*.

Identificación de las entradas/salidas

El estándar Arduino otorga a cada entrada/salida un número de identificación que es independiente del modelo de tarjeta Arduino em-

pleada. Así pues, la salida número 13 está conectada a un diodo LED incorporado en la tarjeta Arduino y es la salida que usa el programa mostrado. En nuestro caso, el diodo LED RGB de la tarjeta de prácticas está conectado a los pines de E/S números 6 —azul—, 7 —verde— y 8 —rojo—, mientras que el pin 13 está conectado al pulsador, como se muestra en el Cuadro 8.5.

9.3. Problemas del capítulo

9.3.1. Introducción a la E/S

EJERCICIOS

▶ **9.1** Conecta a la tarjeta Arduino la tarjeta de prácticas de forma que los tres pines bajo el LED se correspondan con los pines `6`, `7` y `8` y los otros dos pines estén conectados al pin `13` y al pin `GND` que hay junto a él. Recuerda conectar el cable al pin `3.3V` de la tarjeta Arduino. Inicia el entorno Arduino y abre el proyecto `blink` mediante la opción `Archivo - Ejemplos - 01.Basics - Blink` del menú. Compílalo y súbelo a la tarjeta. Comprueba que el LED incorporado en la Arduino Due parpadea —de color amarillo, situado aproximadamente entre los dos conectores USB de la misma e identificado con la letra `L`—.

Sustituye en `blink.c` las tres apariciones del número 13 (como argumento de la función «`pinMode`» y de las dos llamadas a la función «`digitalWrite`») por el número 6. Compila y sube a la tarjeta el nuevo programa. ¿Cómo ha cambiado el comportamiento del programa?

▶ **9.2** Modifica el programa `blink.c` para que haga parpadear el LED de color rojo.

▶ **9.3** Descarga del Aula Virtual, sección *Laboratorio*, apartado *Programas para Arduino*, los archivos del programa `blink_asm`, compílalo y ejecútalo. Comprueba que el LED de la tarjeta Arduino Due parpadea. Recordemos que el microcontrolador ATSAM3X8E, incorporado en la tarjeta Arduino Due que estamos usando, posee varios `PIOs` (*Parallel Input Output*) con varios pines de E/S cada uno de ellos, de forma que los pines `6`, `7` y `8` de la tarjeta Arduino están físicamente conectados a los pines `24`, `23` y `22` del `PIO C` del ATSAM3X8E respectivamente, como se muestra en el Cuadro 8.5.

Consulta el Cuadro 8.1 para determinar la dirección del registro base del `PIO C` y realiza las modificaciones necesarias en `blink_asm.c` y en `blink.s` para que haga parpadear el LED de color rojo. Ten en cuenta que, mientras que el LED incorporado en la Arduino Due se enciende escribiendo un `1` y se apaga escribiendo un `0` en el puerto correspondiente, cada componente del LED RGB de la tarjeta de prácticas se encien-

de escribiendo un `0` y se apaga escribiendo un `1` en su puerto de E/S. Comenta qué modificaciones has tenido que introducir en el programa.

▶ **9.4** Tal como vimos al estudiar las funciones, hay dos formas de pasar parámetros: por valor y por referencia. En el ejemplo propuesto se muestra la técnica de paso de parámetros por valor a un programa en ensamblador —a través de los registros `r0` y `r1`—. Alguna vez, sin embargo, será necesario poder acceder desde el programa en ensamblador a una variable del programa principal. Debemos establecer un mecanismo para poder transferir información entre el programa en `C` y el código ensamblador. Para ello declaramos un vector en `C` y declaramos su nombre como `.extern` en el programa ensamblador para acceder a él usando su propio nombre como etiqueta.

Descarga del Aula Virtual el programa «`blink_cadena`» y observa los siguientes cambios respecto del programa «`blink_asm`»:

- Se ha modificado la declaración de la función `blink` en el fichero «`blink.h`» para que no acepte parámetros pero siga devolviendo un resultado. Para ello se han eliminado las declaraciones de los dos parámetros: «`int time`» e «`int del`». También se han eliminado los parámetros entre paréntesis en la invocación a la función `blink` en el ficheo «`blink_asm.ino`».

- Se ha declarado una cadena de caracteres al principio del programa en `C`, con el contenido «`mensaje`» donde es importante que el último elemento de la cadena sea el carácter `0`.

- Se ha declarado el nombre de la cadena en el programa ensamblador como «`.extern`» para que dicho nombre pueda utilizarse como una etiqueta dentro del program ensamblador.

- Al eliminar los paŕametros de la función `blink`, en el programa en ensamblador hemos asignado al retardo una cantidad fija —#300— y el número de veces que parpadea el LED será el número de caracteres de que consta la cadena. Para ello se ha confeccionado un bucle que recorre la cadena y realiza un parpadeo por cada carácter de la misma hasta encontrar el `0` del final.

Completa el programa ensamblador para que realice la función descrita. Compílalo y súbelo a la tarjeta Ardunio Due. Modifica la longitud de la cadena para comprobar que realmente el programa hace lo que se espera.

▶ **9.5** Modifica el programa ensamblador para que devuelva el número de caracteres de la cadena simplemente copiándolo en el registro `r0`.

▶ **9.6** Modifica el programa en `C` para que muestre en pantalla el número de caracteres de la cadena. Recuerda que las funciones que muestran información en pantalla son «`Serial.print`» y «`Serial.println`».

▶ **9.7** La técnica de compartición de variables se puede emplear también para devolver información desde el programa en ensamblador al programa invocador. En lenguaje `C` se puede reservar espacio para un vector de enteros llamado «`vector`», de forma equivalente a como haríamos con «`.space m`» en ensamblador, de la siguiente forma:

```
«int vector[n];»
```

Hay que tener en cuenta, sin embargo, que el parámetro «`m`» indica número de bytes a reservar, mientras que cada elemento del vector ocupa 4 bytes —1 word— en memoria. Así pues, para realizar la reserva en memoria de un vector de «`n`» elementos, en ensamblador debemos usar $m = 4 * n$.

Así mismo, desde el programa en `C` podemos acceder al elemento i-ésimo del vector mediante «`vector[i]`», pudiendo «`i`» tomar valores entre «`0`» y «`n-1`».

Considerando lo expuesto y empleándolo de forma adecuada, descarga el programa «`blink_vect`» del Aula Virtual y complétalo para que devuelva los contenidos de los registros `r0` al `r7` en el vector llamado «`retorno`» y los muestre en pantalla empleando «`Serial.print`» y «`Serial.println`». Ten en cuenta que estas funciones pueden mostrar valores numéricos en hexadecimal si se les añade el modificador «`HEX`», como por ejemplo en «`Serial.println(n, HEX);`»

▶ **9.8** Consulta el Cuadro 8.8 y, a partir de su contenido, completa el programa «`leefecha`» del Aula Virtual para que acceda a los registros `RTC_CALR` y `RTC_TIMR` y lea la configuración de fecha y hora actuales del ATSAM3X8E. Comparte esa información con el programa principal mediante los vectores `fecha` y `hora` y muestra en pantalla la fecha y hora actuales usando adecuadamente las funciones «`Serial.print`» y «`Serial.println`». ¿Cuál es la fecha con que se configura el RTC por defecto?

▶ **9.9** En el campo `DAY` del registro `RTC_CALR` se almacena el día de la semana dejando la codificación al criterio del usuario. Atendiendo al contenido de este campo cuando se inicializa el sistema, ¿qué codificación se emplea por defecto para el día de la semana?

. . . . . .

9.3.2. E/S por consulta de estado

EJERCICIOS

▶ **9.10** Sabiendo que el pulsador incorporado en la tarjeta de prácticas de laboratorio está conectado a un pin del PIOB, ¿qué registro deberíamos leer para detectar que se ha presionado el pulsador?

▶ **9.11** El pin al que está conectado el pulsador es el correspondiente al bit 27 del PIOB. ¿Qué máscara tenemos que aplicar al valor leído del registro para determinar la posición del pulsador?

▶ **9.12** De acuerdo con el esquema de conexión del pulsador de la tarjeta de prácticas mostrado en la Figura 9.4 y si la resistencia de pull-up de la entrada donde está conectado el pulsador está activada, ¿qué valor leído del bit 27 del registro de E/S del PIOB nos indicará que el pulsador está presionado?

▶ **9.13** Descarga el programa «`pulsa`» del Aula Virtual y complétalo para que, mediante consulta de estado, espere la pulsación del pulsador de la tarjeta de prácticas para regresar devolviendo el `CHIPID` del procesador.

▶ **9.14** Modifica el programa anterior para que regrese cuando se suelte el pulsador tras haberlo pulsado en lugar de cuando se pulse.

▶ **9.15** Completa el programa «`cambia`», disponible en el Aula Virtual, para que con cada pulsación del pulsador encienda cíclicamente el LED RGB con cada uno de sus tres colores básicos.

▶ **9.16** Para poder modificar los contenidos de los registros de fecha —`RTC_CALR`— y hora —`RTC_TIMR`— es preciso detener su actualización escribiendo un `1` en los bits `UPDCAL` y `UPDTIM` del registro de control `RTC_CR` y esperando la confirmación de que se ha detenido la actualización en el bit `ACKUPD` del registro de estado `RTC_SR`. Puedes consultar los detalles de este procedimiento en el apartado *Actualización de la fecha y hora actuales*. Completa el programa «`cambiahora`» para configurar el RTC en el modo de 12 horas con la fecha y hora actuales. Haz que el programa compruebe los valores de los bits `NVCAL` y `NVTIM` para confirmar que la configuración ha sido correcta.

▶ **9.17** Partiendo del programa «`alblink`» disponible en el Aula Virtual, configura una alarma que se active dentro de 8 segundos. Completa el código proporcionado para que consulte el estado del bit de alarma y espere a que ésta se produzca para regresar al programa en `C`, el cual mostrará el instante en que se detectó la activación de la alarma. Ten en cuenta que tienes que copiar la función `cambiahora` confeccionada en el apartado anterior en la zona indicada del código fuente proporcionado.

▶ **9.18** Modifica el periodo del parpadeo para que sea de 6 segundos (3.000 ms encendido y 3.000 ms apagado). ¿Coincide el instante de detección de la alarma con el configurado?¿A qué crees que es debido?

9.3.3. E/S por interrupciones

EJERCICIOS

▶ **9.19** En el programa «`PIOint`», disponible en el Aula Virtual, completa la función «`PIOsetupInt`» para que configure adecuadamente el modo de interrupción seleccionado por los parámetros que se le suministran. Una vez completada, comprueba que funciona correctamente compilando y subiendo el programa a la tarjeta Arduino Due.

▶ **9.20** Modifica el código anterior para ir probando todas las posibles combinaciones de los parámetros «`mode`» y «`polarity`». Completa la siguiente tabla, comentando en cada caso cuál es el comportamiento del programa.

Mode	Polarity	Efecto
FLANCO	BAJO	
	ALTO	
NIVEL	BAJO	
	ALTO	
CAMBIO	BAJO	
	ALTO	

▶ **9.21** Completa en el programa «`RTCint`», disponible en el Aula Virtual, los valores de las etiquetas «`HOY`» y «`AHORA`» para configurar el RTC con la fecha y hora indicadas en los comentarios de las líneas de código que asignan valor a dichas etiquetas.

▶ **9.22** Completa asimismo los valores de las etiquetas «`ALR_FECHA`» y «`ALR_HORA`» para configurar la alarma del RTC de forma que se active a la fecha y hora indicadas en los comentarios de las líneas de código que asignan valor a dichas etiquetas.

▶ **9.23** Completa la función «`RTCsetAlarm`» de forma que configure adecuadamente la fecha y hora de activación de la alarma contenidas en las etiquetas «`ALR_FECHA`» y «`ALR_HORA`» y que active la generación de interrupciones de alarma por parte del `RTC`.

▶ **9.24** Completa en el programa anterior la función «`RTC_Handler`» para que atienda la interrupción de alarma del RTC realizando correctamente las acciones que se describen en los comentarios del código proporcionado. Una vez completada la función, compila y sube el programa a la tarjeta Arduino Due. Explica qué acciones realiza.

▶ **9.25** Prueba diferentes valores de la etiqueta «`RETARDO`», comprueba su efecto en el comportamiento del programa y compáralo con el obtenido con el programa «`alblink`» de la sesión anterior. Comenta las diferencias observadas y relaciónalas con los métodos de consulta de estado e interrupciones.

9.3.4. Pulse Width Modulation (PWM), Direct Memory Access (DMA) y Universal Serial Bus (USB)

EJERCICIOS

▶ **9.26** Consulta en la ayuda de Arduino el funcionamiento de la función «`analogWrite()`» y, al final de la misma, el enlace «`Tutorial:PWM`». ¿Cómo se consigue variar la intensidad luminosa del LED mediante la técnica PWM

▶ **9.27** Descarga el programa «`pwm`» del Aula Virtual, compílalo y súbelo a la tarjeta Arduino Due.

▶ **9.28** Observa que con cada pulsación cambia la intensidad del LED rojo mientras se indica el valor del ciclo de trabajo (*Duty Cycle*) que provoca la intensidad observada. Explica qué relación hay entre el valor del registro `PWM Channel Duty Cycle Register` y la intensidad del LED.

▶ **9.29** Desplaza la tarjeta Arduino Due con suavidad describiendo un arco de unos 60 centímetros a una velocidad moderada y observa el patrón que se visualiza para los diferentes valores del contenido del registro `PWM Channel Duty Cycle Register`. Explica cómo estos patrones obedecen al principio de funcionamiento del PWM visto en el tutorial de Arduino.

▶ **9.30** Cambia el valor actual de la etiqueta «`CLK`» (`0x0000 0680`) por `0x0000 0180`. Repite el experimento anterior y comenta las diferencias apreciadas. ¿Qué cres que ha cambiado?

▶ **9.31** Descarga el programa «`EjemploPWM`» del Aula Virtual, compílalo y súbelo a la tarjeta Arduino Due. ¿Cómo se consiguen los diferentes colores en el LED RGB?

▶ **9.32** Prueba diferentes valores de «`FACTRED`», «`FACTGRN`» y «`FACTBLU`». ¿Qué efecto tienen estos parámetros en el comportamiento del programa?

▶ **9.33** Descarga el programa «`testDMA`» del Aula Virtual, compílalo y súbelo a la tarjeta Arduino Due. Observa en el funcionamiento del programa que el contador de iteraciones se incrementa al mismo tiempo que el DMA realiza las transferencias de datos.

▶ **9.34** Busca en el código del programa el grupo de líneas encerrado en el comentario «`TAMAÑO DE LOS BLOQUES A TRANSFERIR`» y prueba diferentes valores de los parámetros «`SIZE0`», «`SIZE1`», «`SIZE2`» y «`SIZE3`». Ten en cuenta que deben tener valores comprendidos entre 100 y 2048. Explica cómo cambia el tiempo de ejecución en función de dichos valores.

▶ **9.35** ¿Para qué valores de los parámetros se obtiene el tiempo máximo de realización de las transferencias?¿De qué tiempo se trata?¿Qué valor ha alcanzado el contador de iteraciones para dicho tiempo máximo?

▶ **9.36** Descarga el programa «`kbmouse`» del Aula Virtual, compílalo y súbelo a la tarjeta Arduino Due. Abre un editor de textos. Desconecta el cable USB del puerto de programación de la tarjeta Arduino Due y conéctalo al puerto de comunicaciones. Observa el funcionamiento del programa.

▶ **9.37** Consulta la ayuda de Arduino y modifica el programa para que simule un doble click en el instante en que se presiona el pulsador de la tarjeta Arduino Due. Presiona el pulsador cuando el puntero se encuentre sobre un icono del escritorio y comprueba que se inicia la aplicación correspondiente.

▶ **9.38** Modifica el programa para que, en lugar de una elipse, el puntero del ratón describa un rectángulo.

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