Capítulo 1

Primeros pasos con ARM y Qt ARMSim

Índice

1.1. Introducción al ensamblador Thumb de ARM . . . **3**
1.2. Introducción al simulador Qt ARMSim **8**
1.3. Literales y constantes en el ensamblador de ARM . **22**
1.4. Inicialización de datos y reserva de espacio **25**
1.5. Carga y almacenamiento **31**
1.6. Problemas del capítulo **41**

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-

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

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.

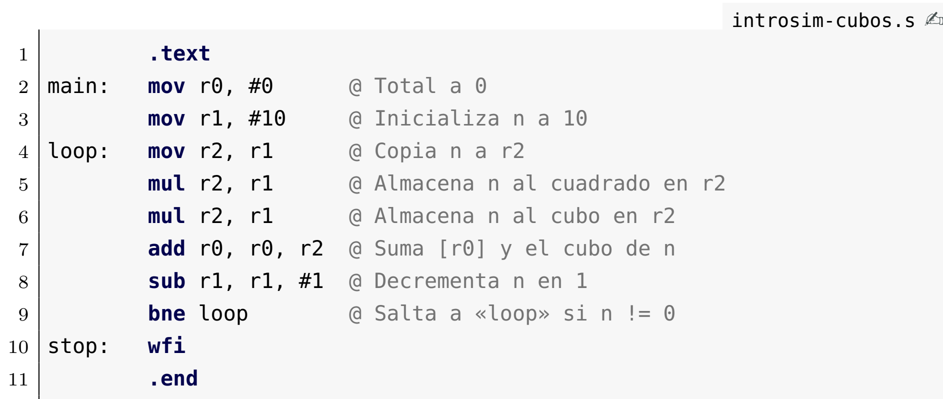
```
2 main:   mov r0, #0      @ Total a 0
3         mov r1, #10     @ Inicializa n a 10
4 loop:   mov r2, r1      @ Copia n a r2
5         mul r2, r1      @ Almacena n al cuadrado en r2
6         mul r2, r1      @ Almacena n al cubo en r2
7         add r0, r0, r2  @ Suma [r0] y el cubo de n
8         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.

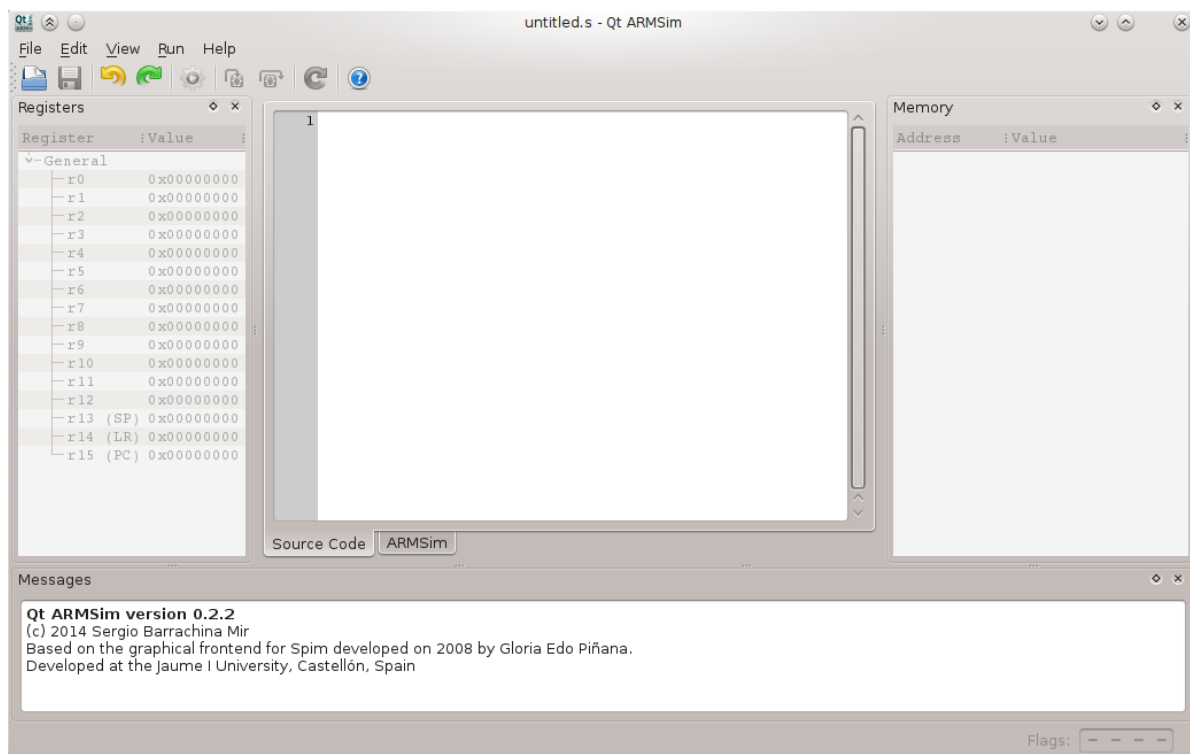


Figura 1.1: Ventana principal de Qt ARMSim

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?

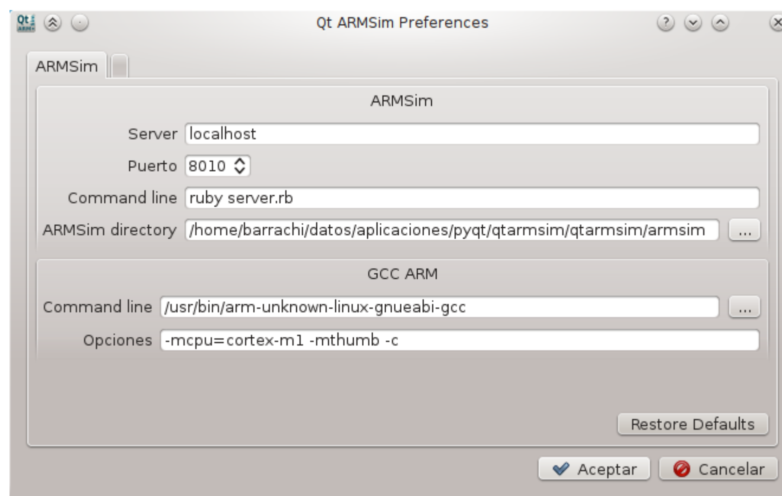


Figura 1.2: Cuadro de diálogo de preferencias de Qt ARMSim

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`».

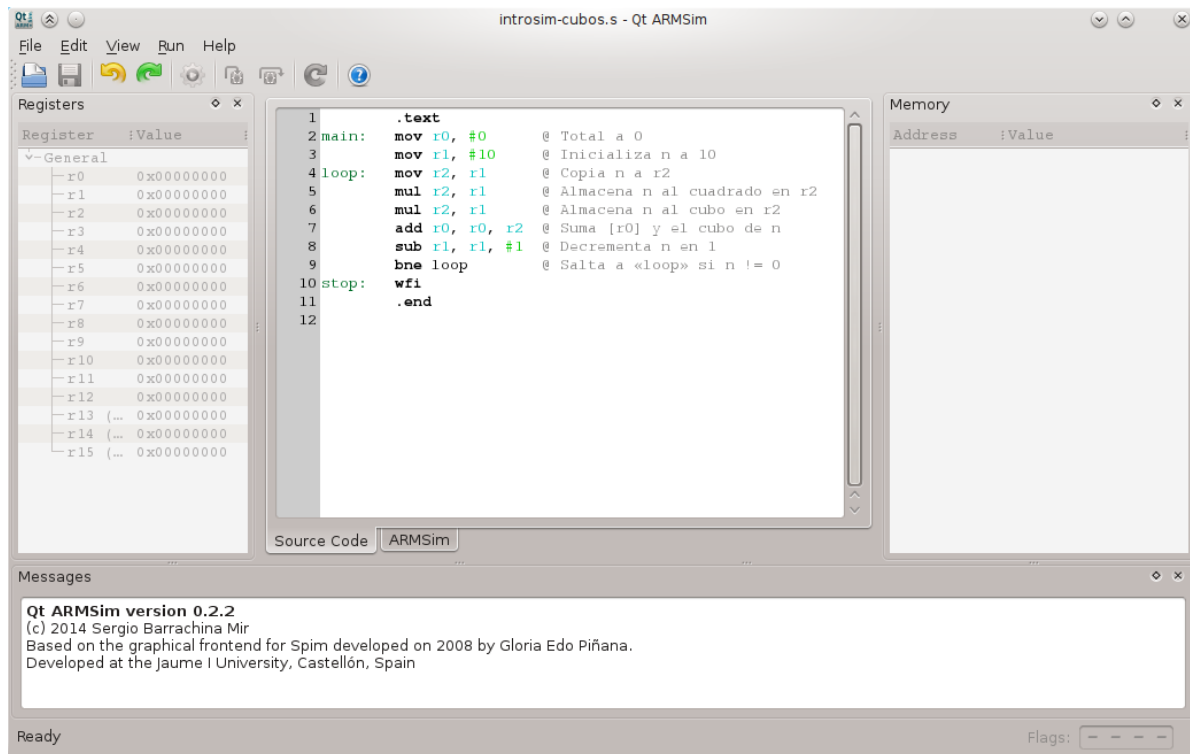


Figura 1.3: Qt ARMSim mostrando el programa «introsim-cubos.s»

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.

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.

¿Cambiar al modo de simulación?

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.

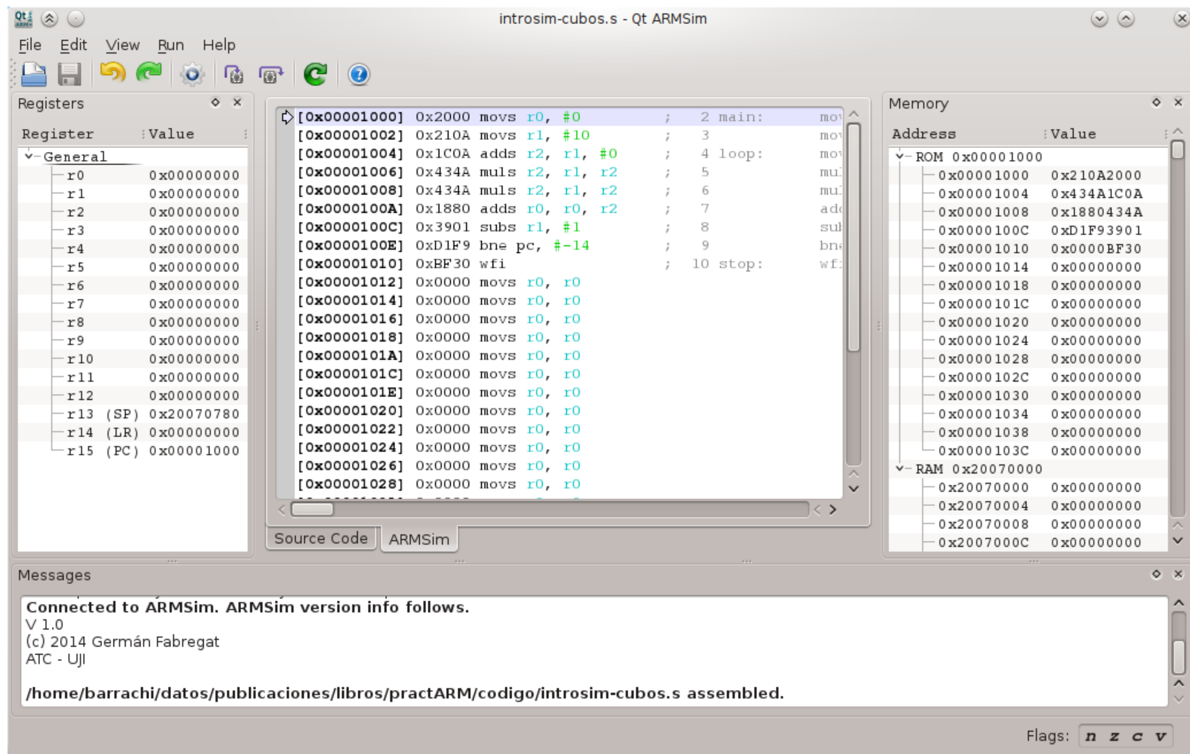


Figura 1.4: Qt ARMSim en el modo de simulación

Como se ha comentado en el párrafo anterior, la dirección `0x0000 1000` 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-

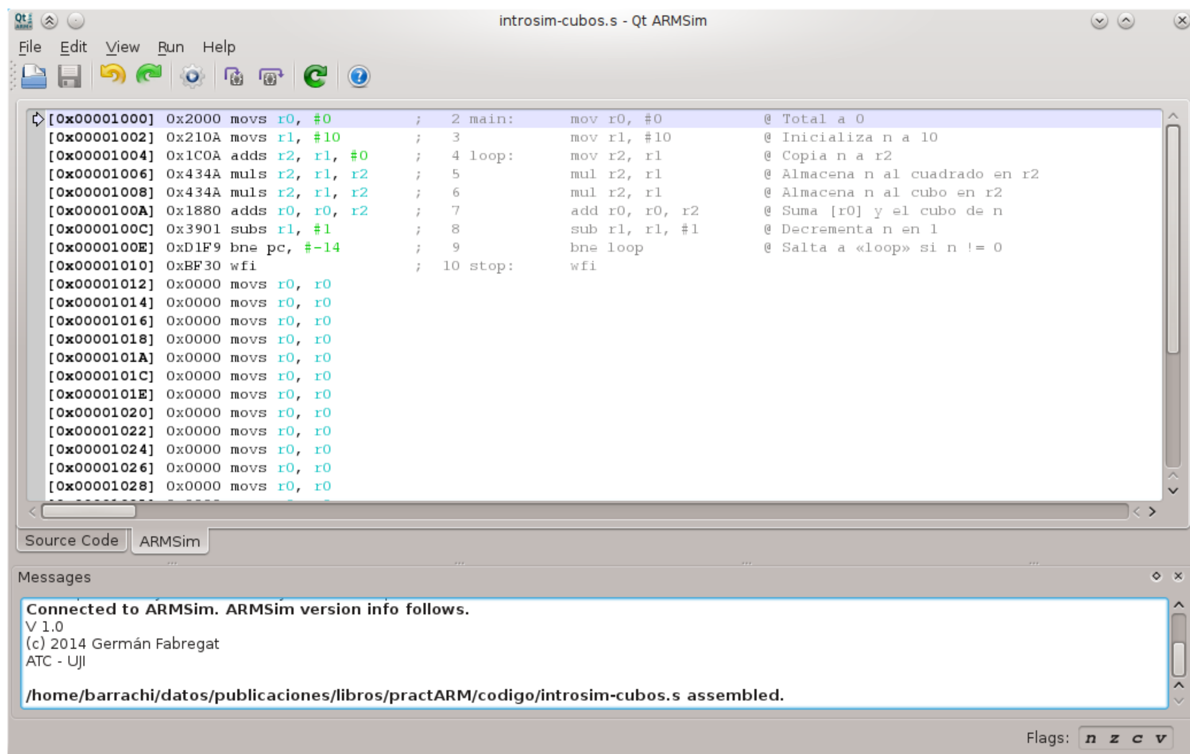


Figura 1.5: Qt ARMSim sin paneles de registros y memoria

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.

```python
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
```

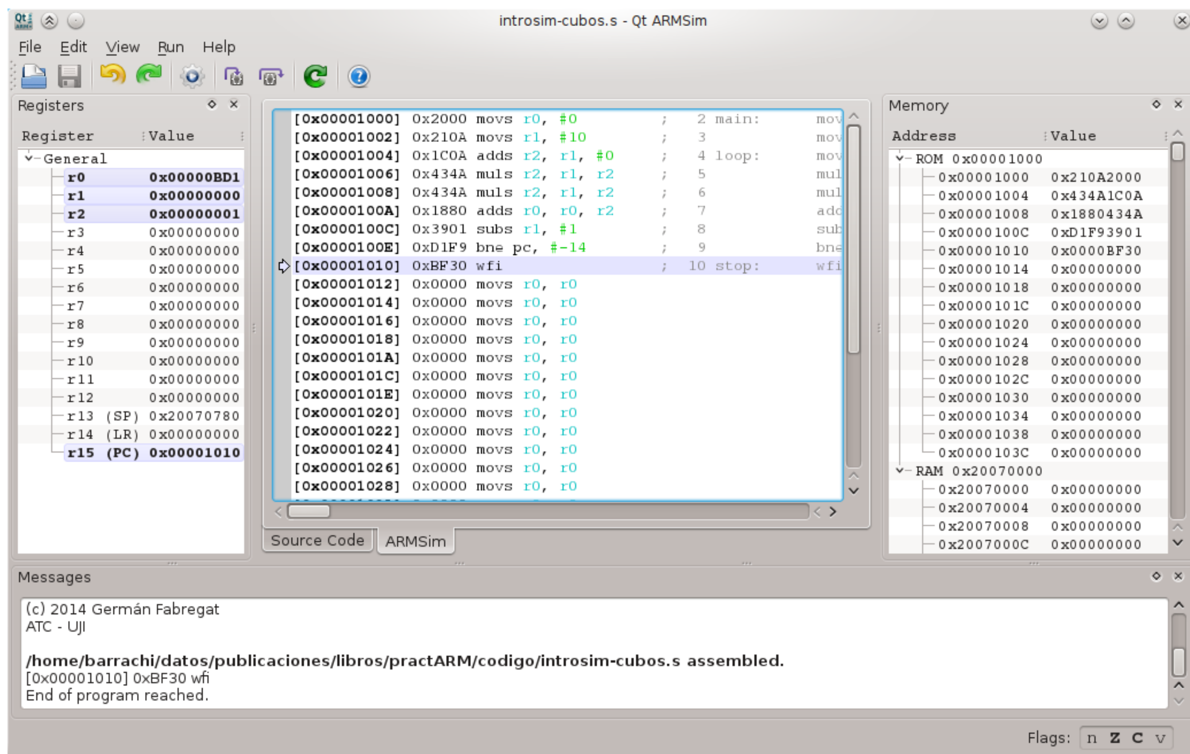


Figura 1.6: Qt ARMSim después de ejecutar el código máquina

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.

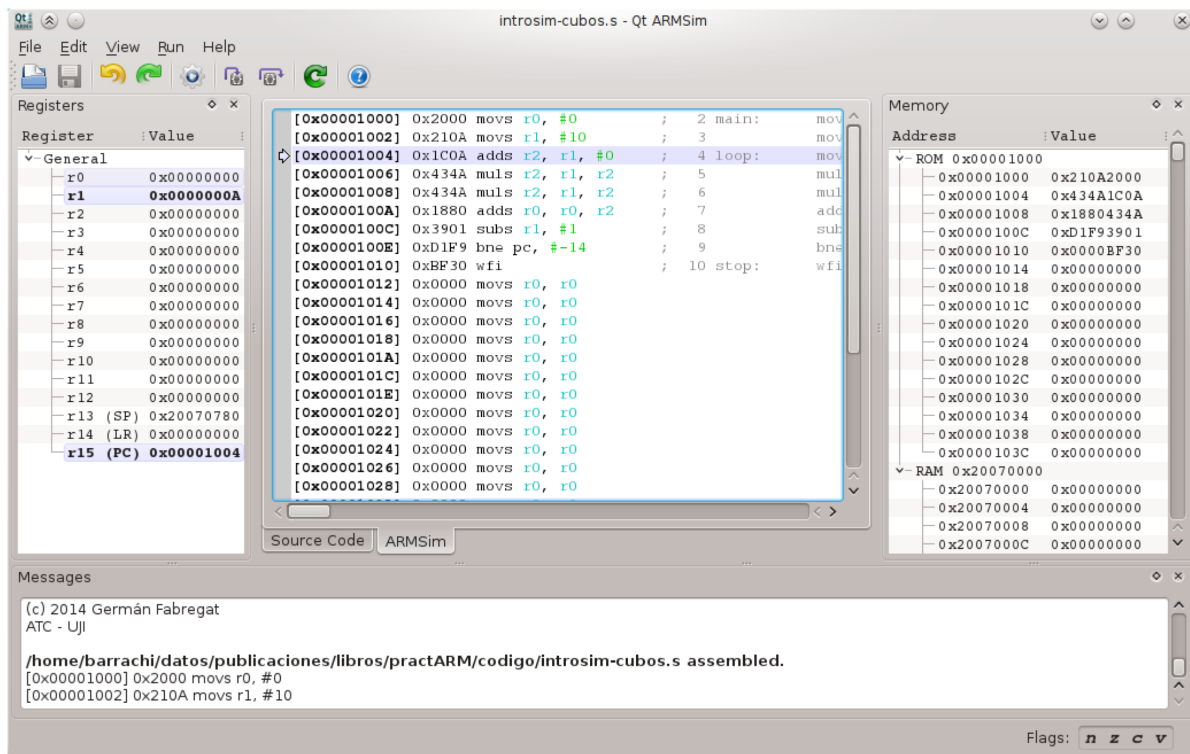


Figura 1.7: Qt ARMSim después de ejecutar dos instrucciones

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.

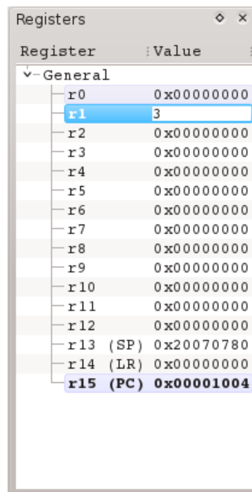


Figura 1.8: Edición del registro `r1`

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.

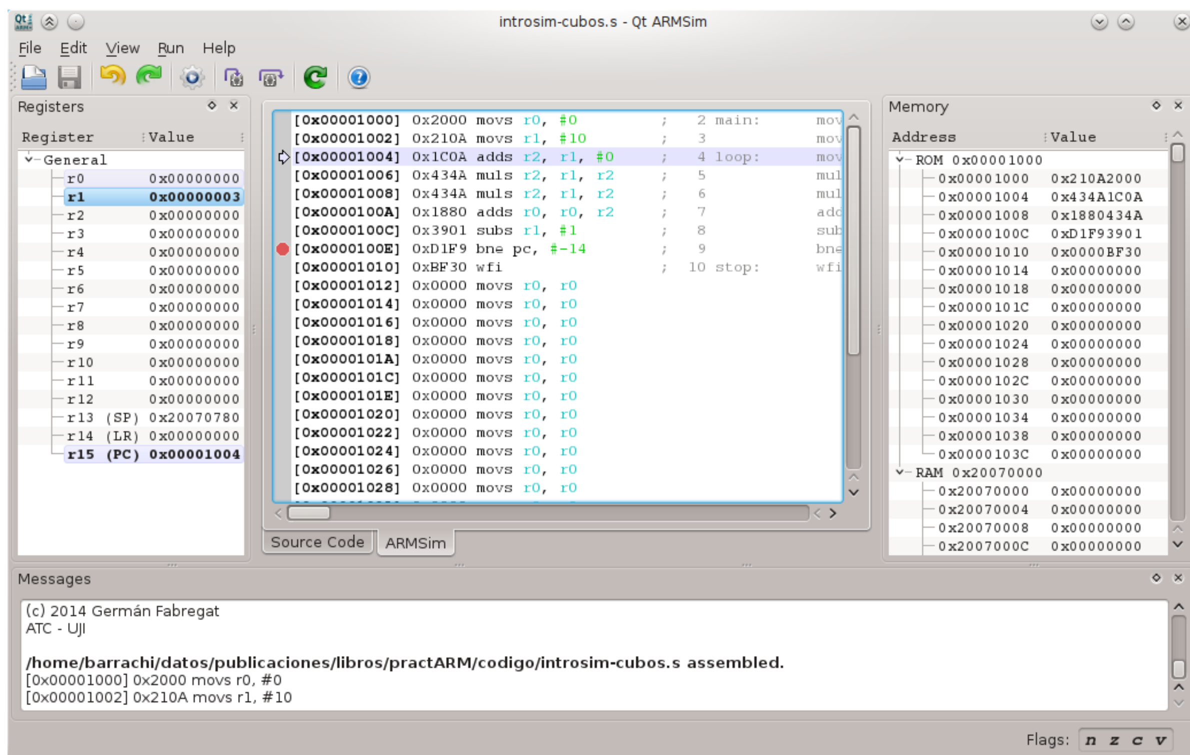


Figura 1.9: Punto de ruptura en la dirección `0x0000 100E`

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).

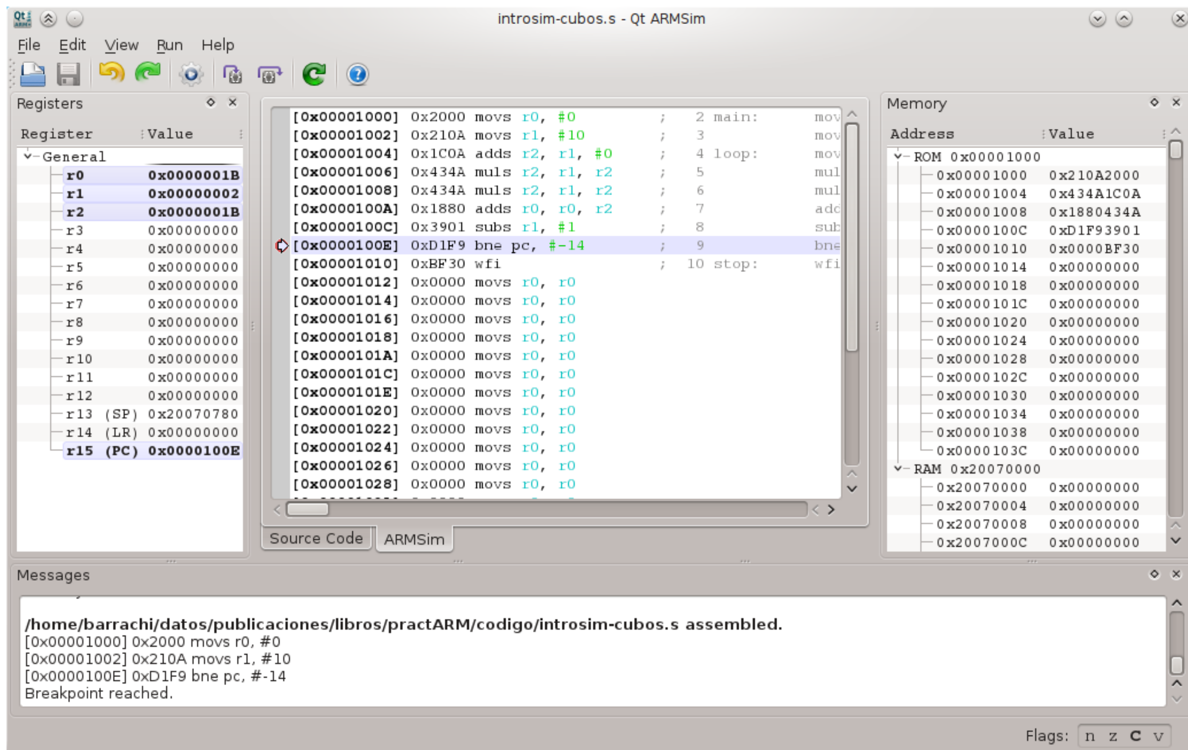


Figura 1.10: Programa detenido al llegar a un punto de ruptura

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

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

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`».

«`.equ Constant, Value`»

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?

. . . .

«Name .req rd»
«.unreq Name»

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:

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

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**»).

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

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.

«`.word Value32`»

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`.

. . . . 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 `0xAABB CCDD` en la posición de memoria `0x2007 0000`, la palabra ocupará los 4 bytes: `0x2007 0000`, `0x2007 0001`, `0x2007 0002` y `0x2007 0003`. Sin embargo, ¿a qué posiciones de memoria irán cada uno de los bytes de la palabra? Las opciones que se utilizan son:

0x2007 0000	0xAA
0x2007 0001	0xBB
0x2007 0002	0xCC
0x2007 0003	0xDD

a) Big-endian

0x2007 0000	0xDD
0x2007 0001	0xCC
0x2007 0002	0xBB
0x2007 0003	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

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

«**.byte** Value8»

[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?

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

«`.asciz "cadena"`»

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

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

«`.space N`»

.... 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

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.

«`.balign 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.

```
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 `0x1020 3040` 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.

«`ldr rd, =Label`»

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.

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, #Offset5]», 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, #Offset5]», 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, «`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 «**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.

. . . .

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