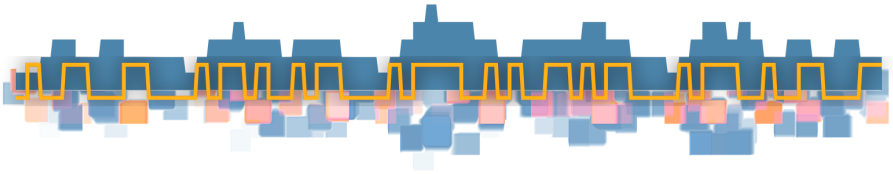
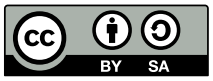
Introducción a la arquitectura de computadores con QtARMSim y Arduino

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





A Laia y a Marc, ¡bienvenidos!

Índice general

Prefacio

Los cursos de Arquitectura de Computadores se han visto obligados históricamente a seguir el frenético ritmo marcado inicialmente por los avances tecnológicos y arquitectónicos en el diseño de grandes computadores, y, posteriormente, a partir de los ochenta, por la evolución en el diseño de microprocesadores. De hecho, se puede decir que la docencia en Arquitectura de Computadores ha pasado por seis grandes eras: *mainframes*, *minicomputadores*, primeros microprocesadores, microprocesadores, RISC y post-RISC [11].

Por otro lado, conforme las universidades han podido acceder a hardware específico a un coste razonable, este ha pasado a utilizarse ampliamente como material de referencia. En concreto, los computadores, procesadores o arquitecturas que han disfrutado de una mayor popularidad en la docencia de Arquitectura de Computadores han sido: el computador PDP-11, el procesador 68000 de Motorola, el procesador 80x86 de Intel, la arquitectura MIPS y el procesador SPARC. Aunque en ocasiones también se ha recurrido a computadores hipotéticos dedicados en exclusiva a la enseñanza de los conceptos arquitectónicos.

En la Universitat Jaume I también habíamos optado por algunas de las alternativas ya comentadas. Comenzamos con el 68000 de Motorola, más adelante utilizamos brevemente un computador hipotético y posteriormente, cambiamos a la arquitectura MIPS, que se utilizó como arquitectura de referencia hasta el curso 2013/14.

A principios de 2013, los profesores de la unidad docente de Arquitectura de Computadores nos plantemos migrar la arquitectura de referencia a ARM por los siguientes motivos. En primer lugar, la arquitectura ARM, al estar basada en RISC, es relativamente sencilla. En segundo lugar, pese a su sencillez, presenta muchas características que la distinguen de otras arquitecturas contemporáneas [10]. Por último, pero no menos importante, el hecho de que ARM sea una arquitectura actual y ampliamente difundida, especialmente en dispositivos móviles, teléfonos inteligentes y tabletas, es un factor especialmente motivador [12]. Cabe destacar que la popularidad de la arquitectura ARM ha explotado en las dos últimas décadas debido a su eficiencia y a la riqueza de su ecosistema: se han fabricado más de 50 000 millones de procesadores ARM;

más del 75 % de la población mundial utiliza productos con procesadores ARM [14].

Una vez tomada la decisión de realizar dicho cambio, comenzamos a replantearnos las guías docentes y los materiales que se deberían utilizar en la enseñanza tanto teórica como práctica de las distintas asignaturas relacionadas con la materia de Arquitectura de Computadores.

En el caso de la asignatura Estructura de Computadores, de primer curso, primer semestre, partíamos del siguiente material: el libro *Estructura y diseño de computadores: la interfaz software/hardware*, de David A. Patterson y Jonh L. Hennessy [17], como referencia para la parte teórica de la asignatura; el libro *Prácticas de introducción a la arquitectura de computadores con el simulador SPIM*, de Sergio Barrachina, Maribel Castillo, José Manuel Claver y Juan Carlos Fernández [5], como libro de prácticas; y el simulador de MIPS `xspim` (actualmente `QtSpim`[1]), como material de laboratorio.

En un primer momento nos planteamos utilizar como documentación para la parte de teoría el libro *Computer Organization and Architecture: Themes and Variations. International Edition*, de Alan Clements, y para la parte de prácticas, el libro de prácticas indicado anteriormente pero adaptado a un simulador de ARM que fuera lo suficientemente sencillo como para permitir centrarse en los contenidos de la asignatura, más que en el manejo del propio simulador.

Desde un primer momento consideramos que la utilización de un simulador era adecuada para los conceptos básicos de los fundamentos de la arquitectura de computadores. Sin embargo, y aprovechando que también debíamos adaptar las prácticas relacionadas con la parte de la entrada/salida a la arquitectura ARM, nos planteamos si queríamos continuar también en esta parte con una experiencia docente basada en el uso de un simulador o si, por el contrario, apostábamos por un enfoque cercano a lo que se ha dado en llamar *computación física* [16]. En el primer caso, las prácticas consistirían en programar en ensamblador el código necesario para interactuar con los dispositivos de entrada/salida proporcionados por el simulador que fuera a utilizarse. En el segundo caso, se interactuaría con dispositivos físicos, lo que permitiría ser consciente de qué es lo que se quiere que pase en la realidad y observar cómo la aplicación que se desarrolle es capaz, o no, de reaccionar adecuadamente ante eventos externos. Siguiendo este enfoque más aplicado, se podría relacionar fácilmente la secuencia de acciones a las que un dispositivo tiene que dar respuesta, con la programación que se haya realizado de dicho dispositivo. Así pues, consideramos que esta segunda opción sería mucho más enriquecedora que simplemente limitarse a observar en la pantalla de un simulador si un código de gestión de la entrada/salida se

[1] QtSpim: <`http://spimsimulator.sourceforge.net/`>

comporta como teóricamente debería. Es más, puesto que mucha de la problemática de la entrada/salida está directamente relacionada con la interacción hombre-máquina, esta segunda opción ofrece la oportunidad de enfrentarse a un escenario real, en el que si no se toman las debidas precauciones, no todo funcionará como se espera.

Sobre la base de las anteriores decisiones, quedaba por concretar qué simulador de ARM se utilizaría para las prácticas no relacionadas con la entrada/salida y qué componente hardware utilizar para dicha parte. Tras evaluar varios entornos de desarrollo y simuladores de ARM, optamos por diseñar nuestro propio simulador, QtARMSim [7], para aquellas prácticas no relacionadas con la entrada/salida. Por otro lado, y tras valorar varias alternativas, optamos por la tarjeta Arduino Due [8] para las prácticas de entrada/salida.

La versión anterior de este libro, *Prácticas de introducción a la arquitectura de computadores con QtARMSim y Arduino* [6], que escribimos para el curso 2014/15 como manual de prácticas, proponía un conjunto de prácticas sobre QtARMSim y Arduino y referenciaba al libro de Alan Clements para los aspectos teóricos de la asignatura.

Este libro es una versión ampliada y reestructurada del anterior. Incorpora aquellos aspectos teóricos que no se abordaban en la versión previa, por lo que ahora debiera ser posible utilizarlo como material de referencia tanto de teoría como de laboratorio. Además, propone una secuenciación distinta de algunos de sus capítulos, principalmente para que el primer contacto con el ensamblador sea con aquellos tipos de instrucciones conceptualmente más sencillos. Adicionalmente, los formatos de instrucción que antes se abordaban en un capítulo propio, ahora se han incluido en aquellos capítulos en los que se presentan las instrucciones correspondientes. Por último, se han reorganizado las colecciones de ejercicios en cada capítulo con el objetivo de que el estudiante explore todo el contenido de un capítulo antes de enfrentarse a ejercicios de mayor complejidad.

El libro está estructurado en tres partes. La primera parte aborda los aspectos teóricos básicos de la arquitectura de computadores. La segunda parte presenta aspectos más avanzados de los fundamentos de la arquitectura de computadores tomando como referencia la arquitectura ARM y proponiendo ejercicios sobre el simulador QtARMSim. La última parte describe la problemática de la entrada/salida proponiendo una serie de prácticas con la tarjeta Arduino Due. En concreto, este manual pretende cubrir los siguientes temas de la unidad docente Fundamentos de Arquitectura de Computadores definida en el *Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering* [15]:

- Organización de la máquina de von Neumann.
- Formatos de instrucción.

- El ciclo de ejecución; decodificación de instrucciones y su ejecución.
- Registros.
- Tipos de instrucciones y modos de direccionamiento.
- Mecanismos de llamada y retorno de una subrutina.
- Programación en lenguaje ensamblador.
- Técnicas de entrada/salida e interrupciones.

Además, como complemento a este libro se ha creado el siguiente sitio web: «http://lorca.act.uji.es/libro/introARM/», en el que se puede consultar y obtener material adicional relacionado con este libro. Entre otros: el simulador QtARMSim, el entorno de desarrollo de Arduino modificado para la ejecución de programas en ensamblador, las guías para la instalación de dicho entorno en GNU/Linux y en Windows y la colección de ejercicios utilizados en la tercera parte del libro.

Deseamos que este libro te sea útil y esperamos poder contar con tus sugerencias para mejorarlo.

Reconocimientos

Algunas de las figuras de este libro se han elaborado a partir de imágenes de OpenClipart,[2] que están licenciadas bajo la «Creative Commons Zero 1.0 Public Domain License». Por otra parte, el logo de la Wikipedia,[3] que aparece debajo de las definiciones extraídas de esta enciclopedia, pertenece a la Wikimedia Foundation, Inc., y está licenciado bajo la «Creative Commons Attribution-ShareAlike 3.0 License».

[2] OpenClipart: <https://openclipart.org/>
[3] Wikipedia: <https://wikipedia.org/>

Parte I

Introducción

Capítulo 1

Introducción a la Arquitectura de Computadores

Índice

Los primeros procesadores que aparecieron en el mercado se componían de muy pocos transistores —decenas de miles— y tenían un campo muy reducido de aplicaciones. Se trataba de sencillos microcontroladores destinados a usos muy específicos y que básicamente eran empleados en sistemas de control. Han pasado más de cuarenta años desde entonces y los avances tecnológicos han provocado notables cambios tanto en el campo de los procesadores como en el de sus aplicaciones. Los procesadores cada vez se componen de más transistores —actualmente del orden de miles de millones—, lo que ha permitido mejorar notablemente su arquitectura e incorporar técnicas que los hacen más rápidos, complejos y económicos, lo que a su vez ha propiciado que su campo de aplicación sea cada vez más extenso.

Actualmente, el procesador es el elemento principal de los ordenadores de sobremesa y portátiles, y de muchos dispositivos electrónicos de gran uso, como agendas, móviles, dispositivos de uso doméstico, etcétera. No obstante, los principios básicos de un ordenador, o de cualquier dispositivo que incluya un ordenador, son muy sencillos. En este capítulo se describen los elementos básicos que componen un ordenador y sus principios de funcionamiento.

1.1. Componentes de un ordenador

El modelo de funcionamiento de los ordenadores actuales continúa siendo, con variaciones poco significativas, el establecido por John von Neumann en 1945, que a su vez se basó en las ideas de la máquina analítica de Charles Babbage, de 1816. Estas ideas, con casi doscientos años de antigüedad, materializadas en circuitos muy rápidos, con miles de millones de transistores, hacen que la informática haya llegado a ser lo que conocemos hoy en día.

El principio de funcionamiento de los ordenadores es sencillo. El núcleo del ordenador transforma y modifica datos que tiene almacenados, dirigido por una sucesión de órdenes que es capaz de interpretar, y que también están almacenadas en él. Este conjunto de órdenes y datos constituye lo que se conoce como **programa**. Siguiendo un programa, un ordenador es capaz de modificar y transformar datos para, por ejemplo, hacer cálculos matemáticos o buscar palabras en un texto. Además de lo anterior, el ordenador también dispone de un conjunto de elementos que hacen posible su interacción con el mundo exterior, lo que le permite recibir los datos de partida y las órdenes, y comunicar los resultados.

De esta descripción, simple pero fiel, del funcionamiento de un ordenador se deduce que contiene las siguientes tres clases de elementos (véase la Figura 1.1), con funciones claramente diferenciadas.

El núcleo del ordenador, que recibe el nombre de **procesador**, es capaz de encontrar, entender y mandar realizar las órdenes, también llamadas *instrucciones*. Se puede decir que el procesador es el elemento del ordenador capaz de: I) ejecutar las instrucciones codificadas en un programa, encontrando los datos que se van a transformar y almacenando el resultado de dicha transformación; y II) generar las señales eléctricas necesarias para coordinar el funcionamiento de todo el sistema.

Por otro lado, el elemento que almacena los datos y las instrucciones de un programa recibe el nombre de **memoria**. Esta se compone de una colección ordenada de recursos de almacenamiento de manera que cada uno de ellos se identifica por una dirección. Cuando la memoria de un ordenador almacena de forma indistinta datos e instrucciones, tal y como se propuso originalmente, se dice que dicho ordenador tiene una

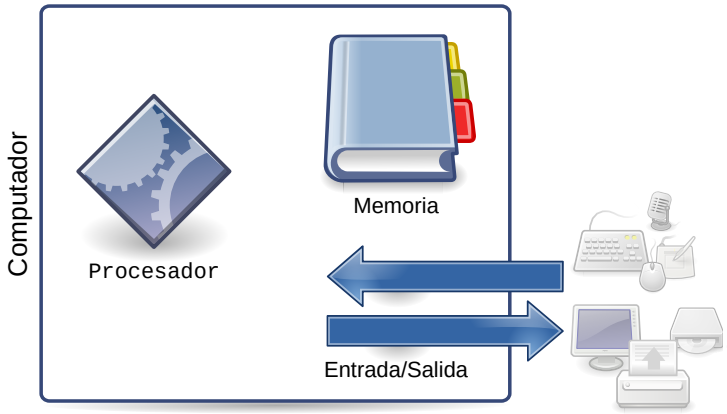


Figura 1.1: Componentes de un computador

arquitectura von Neumann. Por contra, si existe una memoria específica para almacenar los datos y otra distinta para las instrucciones, se dice que el ordenador en cuestión presenta una **arquitectura Harvard**. Posteriormente se profundizará más acerca de esta diferencia. En cualquier caso, y salvo por este detalle, el funcionamiento de la memoria de los ordenadores es el mismo independientemente de la arquitectura elegida.

Por último, la **entrada/salida** está formada por el conjunto de componentes que permiten relacionar un ordenador con el mundo exterior. La comunicación del ordenador con el exterior es compleja y admite tanta diversidad como las aplicaciones actuales de la informática, que como sabemos son muchas y no todas implican la comunicación de datos entre el ordenador y usuarios humanos. Por eso, la entrada/salida de un ordenador es igualmente compleja y variada, tal y como se verá en su momento.

Una vez vistas las clases de elementos que constituyen un ordenador, es posible reformular su funcionamiento diciendo que el procesador ejecuta un programa, o secuencia de instrucciones, almacenado en memoria para realizar, guiado por aquel, las transformaciones adecuadas de un conjunto de datos, que también está almacenado en la memoria. La ejecución de determinadas instrucciones permite la lectura de datos, la presentación de resultados y en general, la comunicación con el exterior, que se realiza a través de la entrada/salida.

En los siguientes apartados se describe con más detalle el funcionamiento de cada uno de dichos componentes con el objetivo de profundizar en todos los aspectos del funcionamiento del ordenador. Aunque en general se seguirá el modelo de la arquitectura von Neumann, con una única memoria principal almacenando tanto datos como instrucciones,

en el apartado dedicado a la memoria se explicarán las diferencias, ventajas e inconvenientes de esta arquitectura con respecto a la Harvard, y se describirá el modelo más común de los ordenadores de propósito general actuales.

1.2. El procesador, el núcleo del ordenador

Aunque los tres componentes que se han mencionado son necesarios para que un ordenador funcione, el procesador es el elemento principal del ordenador. Las razones son evidentes. Por una parte, es capaz de interpretar órdenes y generar las señales de control que, con más o menos intermediaciones posteriores, rigen el funcionamiento de todo el sistema. Por otra parte, el conjunto de todas las órdenes que es capaz de ejecutar, lo que se llama el **conjunto de instrucciones**[1] (*instruction set*, en inglés) del procesador, determina las características del sistema y la estructura de los programas. El tamaño y la forma de organizar la memoria, la manera de interactuar con la entrada/salida, vienen también determinadas por el procesador. De esta manera, el procesador establece las características propias diferenciales de cada ordenador, lo que se denomina *arquitectura*, y que definiremos con rigor más adelante.

Es posible clasificar los procesadores dependiendo de las características del ordenador en el que se van a utilizar en: I) procesadores de altas prestaciones, y II) procesadores de alta eficiencia energética. Los primeros se utilizan tanto para ordenadores personales o de sobremesa como para superordenadores destinados a cálculo masivo, como por ejemplo cualquiera de los que conforman la Red Española de Supercomputación[2] o de los que aparecen en la lista TOP 500.[3] La prioridad en el diseño de estos procesadores es la ejecución del mayor número posible de instrucciones u operaciones por segundo. El juego de instrucciones de este tipo de procesadores suele ser muy amplio y frecuentemente proporcionan recursos que facilitan al sistema operativo la gestión avanzada de la memoria y la gestión de la multitarea. Los procesadores Xeon de Intel son actualmente los líderes en este mercado.

El segundo tipo de procesadores, los de alta eficiencia energética, está destinado a dispositivos alimentados mediante baterías, como son, por ejemplo, los teléfonos inteligentes y las tabletas, y que tienen el ahorro energético como uno de sus objetivos de diseño. Los principales representantes de este grupo son los procesadores basados en la arquitectura ARM. Algunas versiones de esta arquitectura están específicamente diseñadas para trabajar con pocos recursos, optimizar el tamaño de las

[1] También llamado **juego de instrucciones** o **repertorio de instrucciones**.

[2] <https://es.wikipedia.org/wiki/Red_Española_de_Supercomputación>

[3] <https://www.top500.org/resources/top-systems/>

instrucciones para ahorrar espacio en memoria y disponer de un juego de instrucciones simple y sencillo. Intel también tiene una familia de procesadores pugnando por este mercado: los Atom.

Conviene aclarar que aunque los términos *procesador* y *microprocesador* se suelen utilizar indistintamente, el término **microprocesador** se refiere concretamente a un procesador implementado por medio de un circuito integrado.

Por otro lado, también conviene saber que el término **microcontrolador** hace referencia a un circuito integrado que incorpora no solo un procesador —generalmente muy sencillo—, sino también las restantes partes del ordenador, todo en el mismo circuito integrado. Los microcontroladores están orientados a realizar una o algunas tareas muy concretas y específicas, como por ejemplo el control de frenado ABS de un coche o las tareas de un electrodoméstico cualquiera. Puesto que las aplicaciones en las que se utilizan los microcontroladores son muy variables en cuanto a requisitos de ejecución, los fabricantes de microcontroladores suelen ofrecer una amplia gama de modelos con diferentes prestaciones. De esta forma, es posible seleccionar un modelo que se ajuste a las necesidades de la aplicación en la que va a ser utilizado, abaratando así el coste del producto final. Como ejemplos de microcontroladores están los PIC de la empresa Microchip o el SAM3X8E, utilizado en las tarjetas Arduino Due y que incorpora: un procesador ARM Cortex M3 de 32 bits, memoria (512 KiB Flash más 96 KiB SRAM) y un conjunto de dispositivos de E/S.

1.2.1. Partes del procesador

Para explicar cómo el procesador lleva a cabo sus funciones, se puede considerar que este está compuesto de dos partes, con cometidos bien diferenciados: la unidad de control y el camino de datos (véase la Figura 1.2). La **unidad de control** es la encargada de generar y secuenciar las señales eléctricas que sincronizan el funcionamiento tanto del propio procesador, mediante señales internas del circuito integrado, como del resto del ordenador, con líneas eléctricas que se propagan al exterior a través de sus pines de conexión. El **camino de datos**, por su parte, está formado por las unidades de transformación y de transporte de datos. Es el responsable de almacenar, transportar y realizar operaciones —sumas, restas, operaciones lógicas, etcétera— con los datos, siguiendo la coordinación de la unidad de control.

Aunque la división en unidad de control y camino de datos es más conceptual que física, es fácil identificar los bloques estructurales dentro del procesador que pertenecen a cada una de estas partes. Sin preocuparnos especialmente de esta circunstancia, vamos a describir a conti-

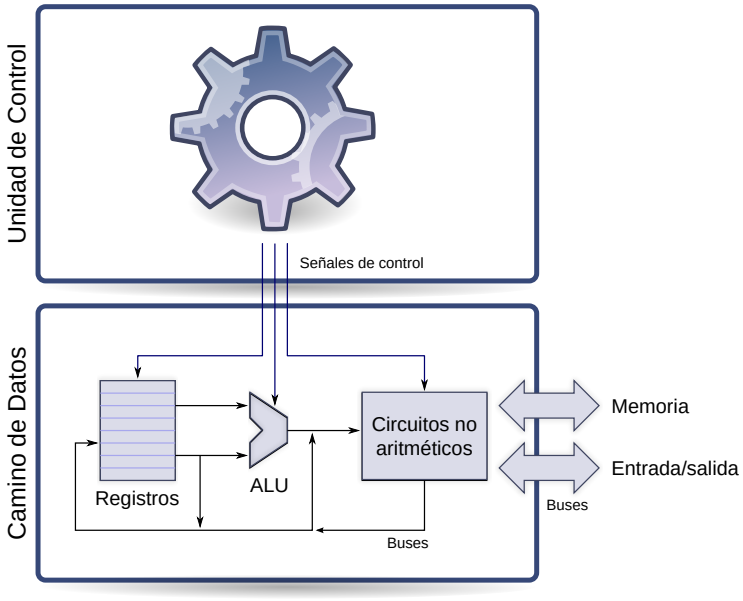


Figura 1.2: Componentes de un procesador

nuación los elementos estructurales más importantes que se encuentran en todos los procesadores:

Registros. Son elementos de almacenamiento propios del procesador. Así como la memoria es un elemento externo al procesador que permite almacenar gran cantidad de datos e instrucciones, los registros constituyen un elemento interno de almacenamiento que permiten almacenar una pequeña cantidad de información. La limitación en la cantidad de información que pueden almacenar los registros es debida a que el número de registros disponibles en un procesador es muy pequeño. Eso sí, gracias a que son un elemento interno del procesador y a que el número de registros es limitado, el acceso a la información almacenada en ellos es mucho más sencillo y rápido que el acceso a la memoria.

Los registros pueden clasificarse según su visibilidad en:

Registros de uso interno. Son registros usados internamente por el procesador, son necesarios para su funcionamiento, pero no son visibles por el programador.

Registros visibles por el programador. Son aquellos registros que pueden ser utilizados explícitamente por las instrucciones máquina.

Por otro lado, ya sean registros internos o visibles al programador, los registros se pueden clasificar en cuanto a su funcionalidad como:

Registros de propósito específico. Tienen asignada una función específica. Entre los registros de propósito específico destacan: el contador de programa, el registro de instrucción y el registro de estado.

Registros de propósito general. Son registros que no tienen un cometido predeterminado y que se utilizan como almacenamiento temporal de los datos que se están procesando en un momento dado.

Más adelante se comentará con más profundidad la función y las características de los registros y su relación con la arquitectura.

Unidades de transformación. Permiten realizar operaciones con los datos. Son circuitos electrónicos que generan un resultado en función de uno o varios datos iniciales. Todos los procesadores cuentan entre ellas con una **unidad aritmético-lógica** (o ALU), que suele operar con dos datos de entrada y que es capaz de realizar sumas, restas y las operaciones lógicas bit a bit más comunes. Tampoco suele faltar en los procesadores una unidad de desplazamiento que entrega a su salida rotaciones o desplazamientos de un número de bits dado, del valor presente a su entrada. Además de estas, es frecuente encontrar unidades que realicen la multiplicación o división de datos enteros, todo tipo de operaciones sobre datos en coma flotante, u operaciones más especializadas sobre vectores de datos para tratamiento de señal, multimedia, gráficos, etcétera.

Circuitos digitales y secuenciadores. Se encargan de generar, transformar y distribuir las señales de control que sincronizan la ejecución de las instrucciones y, en general, el funcionamiento del sistema. Estos circuitos no son importantes para saber qué conjunto de instrucciones es capaz de interpretar el procesador, pero sí absolutamente necesarios para que sea capaz de hacerlo correctamente.

Buses. Están compuestos por un conjunto de líneas conductoras que conectan los distintos componentes del procesador por los que fluye la información, esto es, los registros y las unidades de transformación. Tanto las señales de control como los datos que transforma un procesador son considerados, lógicamente, valores binarios de varios bits —entre 8 y 64 según el tipo de procesador—, que se almacenan y distribuyen también como señales eléctricas a través de estas líneas. Dado el número y el tamaño de los registros y la gran cantidad de unidades funcionales de los procesadores de hoy

en día, los buses constituyen una parte importante del procesador —y también del resto del ordenador, en el que desempeñan una función similar—.

1.2.2. Ejecución de instrucciones

Al inicio del capítulo hemos descrito, de forma simple y poco detallada, el funcionamiento de un ordenador. Una vez descrito el procesador y sus partes de forma más pormenorizada, es posible explicar con más detenimiento su funcionamiento y, con ello, el del ordenador casi en su totalidad —salvo lo relacionado con los elementos de entrada/salida, que se verá explícitamente más adelante—. Cuando se enciende un ordenador, sus circuitos activan de forma adecuada la señal de inicio del procesador, lo que se conoce como **reset**. Entonces el procesador comienza a funcionar y, tras un proceso de configuración interna, ejecuta[4] la primera instrucción, ubicada en una dirección predeterminada de la memoria —lo que se suele conocer como **dirección o vector de reset**—. A partir de ese momento, y hasta que se detiene su funcionamiento, el procesador no hace otra cosa que ejecutar una instrucción tras otra y, como se ha comentado, ir transformando y moviendo los datos de acuerdo con las instrucciones ejecutadas. De esta manera, para entender cómo funciona un procesador, basta con saber cómo se ejecutan las instrucciones y el resultado que produce, sobre los datos o el sistema, la ejecución de cada una de ellas.

Independientemente de la operación que realicen, todas las instrucciones se ejecutan siguiendo una serie de pasos que se conocen como fases de ejecución. Estas fases son comunes a todos los procesadores, aunque puedan llevarse a cabo con diferencias en cada uno de ellos, sobre todo en lo que afecta a su temporización. Las fases de ejecución de una instrucción reciben el nombre de **ciclo de instrucción** y son:

1. **Lectura de la instrucción.** El procesador mantiene en uno de sus registros, llamado generalmente contador de programa —abreviado como PC, de *Program Counter* en inglés—, la dirección de memoria de la siguiente instrucción que va a ejecutar. En esta fase, el procesador envía a la memoria, mediante los buses de interconexión externos al procesador, la dirección almacenada en el PC y la memoria responde devolviendo la instrucción a ejecutar. Esta fase también se suele designar en la literatura como **búsqueda de la instrucción**.

[4] Una instrucción especifica ciertas acciones que debe llevar a cabo el procesador. Ejecutarla implica realizar efectivamente estas acciones. Como la propia instrucción incluye información acerca de sus operandos, transformación, etcétera, es frecuente utilizar también la expresión interpretar una instrucción.

2. **Decodificación de la instrucción.** El procesador almacena la instrucción recibida de la memoria en uno de sus registros internos, el registro de instrucciones. La decodificación consiste en que los circuitos de control que forman parte de la unidad de control interpreten dicha instrucción y generen la secuencia de señales eléctricas que permiten ejecutarla específicamente. En muchos procesadores esta fase no consume tiempo pues la circuitería ya está preparada para funcionar adecuadamente guiada por los bits de la propia instrucción.

3. **Incremento del contador de programa.** Como se ha dicho, el procesador ejecuta una instrucción tras otra. Para que al terminar la ejecución de la instrucción en curso se pueda comenzar con la siguiente, el PC debe incrementarse según el tamaño de la instrucción leída, que es lo que se hace en esta fase.

4. **Ejecución de la instrucción.** Las fases anteriores son comunes a todas las instrucciones; las diferencias entre unas instrucciones y otras, se manifiestan únicamente en esta fase de ejecución, que se puede descomponer en tres subetapas que se dan en la mayor parte de las instrucciones:

   4.1 Lectura de los operandos. Casi todas las instrucciones operan con datos o los copian de unos recursos de almacenamiento a otros. En esta fase se leen los datos que se van a tratar, llamados normalmente operandos fuente, desde su localización.

   4.2 Ejecución. En esta fase se realiza la transformación de los datos leídos en una de las unidades del procesador, por ejemplo, una operación aritmética entre dos operandos fuente obtenidos en la subetapa anterior.

   4.3 Escritura de resultados. En esta parte de la ejecución, el resultado generado en la fase anterior se almacena en algún recurso de almacenamiento, que recibe el nombre de operando destino.

Una vez completadas las fases anteriores, el procesador ha finalizado la ejecución de una instrucción y vuelve a la primera fase. Como durante la ejecución de la instrucción que acaba de completar, el PC se había incrementado para contener la dirección de la siguiente instrucción que se debía ejecutar, el procesador repetirá de nuevo las mismas fases, pero esta vez con la siguiente instrucción del programa, por tanto, ejecutará una nueva instrucción. Y así una vez tras otra hasta que se apague la fuente de alimentación —o se ejecute alguna instrucción especial que detenga el funcionamiento del procesador y, con ello, su consumo de energía—.

Conviene tener en cuenta que estas fases de ejecución, que se han presentado secuencialmente, pueden en la práctica ejecutarse en un orden diferente, sobre todo en lo que respecta a la ejecución simultánea de distintas fases de varias instrucciones. Sin embargo, para los objetivos de este texto, quedémonos con la secuencia de fases tal y como se ha explicado, y con que las instrucciones se ejecutan secuencialmente, una tras otra, a medida que se incrementa el contador de programa, ya que es la forma más adecuada para entender el funcionamiento del ordenador. En textos más avanzados se puede encontrar información sobre aspectos más complejos como son la ejecución simultánea, ejecución fuera de orden, etcétera.

1.2.3. Tipos de instrucciones

Las instrucciones que ejecuta un procesador se pueden clasificar en un conjunto reducido de tipos de instrucciones. De hecho, como los tipos de instrucciones soportados suelen ser los mismos, uno de los aspectos que diferencia a las distintas arquitecturas de procesador es el número y la forma de las instrucciones de cada tipo soportadas por cada una de ellas. A continuación se describen los distintos tipos de instrucciones que puede ejecutar un procesador.

Las **instrucciones de transformación de datos** son las que realizan operaciones sobre los datos en alguna de las unidades de transformación del procesador. Como estas operaciones requieren operandos de entrada y generan un resultado, siguen fielmente las tres subfases de ejecución descritas en el apartado anterior. El número y tipo de instrucciones de esta clase que implemente un procesador dependerá de las unidades de transformación de datos de que disponga.

Las **instrucciones de transferencia de datos** son las encargadas de copiar los datos de unos recursos de almacenamiento a otros. Lo más común es que se transfieran datos entre los registros y la memoria, y viceversa, pero también pueden moverse datos entre registros o, con menos frecuencia, entre posiciones de memoria. Las instrucciones específicas que intercambian datos con la entrada/salida, si existen en una arquitectura, también se clasifican dentro de este tipo de instrucciones.

Las **instrucciones de control del flujo del programa** son las que permiten alterar el orden de ejecución de las instrucciones de un programa. Según lo descrito en el apartado anterior, el procesador ejecuta una instrucción, luego la siguiente, luego otra, etcétera. De esta forma, el procesador ejecutaría todas las instrucciones que se encuentran en la memoria al ir recorriendo las respectivas direcciones secuencialmente. Sin embargo, con algo tan simple como que haya instrucciones que puedan modificar el contenido del PC, siendo este por tanto el operando destino de dichas instrucciones, se conseguiría alterar el flujo de ejecu-

ción de un programa para que este no fuera el estrictamente secuencial. Estas son las llamadas instrucciones de control de flujo de programa o instrucciones de salto, cuya finalidad es modificar el contenido del PC con la dirección efectiva de memoria hacia donde se quiere desviar la ejecución de un programa (dirección del salto). En muchos casos, estas instrucciones verifican una condición de datos en la subfase de ejecución, de esta manera el programa puede decidir qué instrucciones ejecuta en función de los datos con que está trabajando. Este tipo de instrucciones permiten implementar estructuras de programación condicionales —*if*, *else*, *switch*, etcétera— o iterativas —*for*, *while*, etcétera— .

Por último, muchos procesadores disponen de distintos modos de funcionamiento o de configuraciones que influyen en su relación con el sistema o son propios del procesador. Las **instrucciones de control del procesador** sirven para cambiar de modo de funcionamiento del procesador, por ejemplo, entre modo de bajo consumo y funcionamiento normal, configurar alguna característica como la habilitación de interrupciones o la forma de gestionar la memoria, etcétera. Son instrucciones relacionadas con la arquitectura de cada procesador y se utilizan normalmente en el código del sistema operativo, rara vez en programas de aplicación.

1.2.4. Codificación de instrucciones y formatos de instrucción

Tal y como se ha visto, el procesador ejecuta una instrucción llevando a cabo una secuencia de fases. En la primera fase del ciclo de instrucción, lee la instrucción desde la memoria. En la segunda fase, al decodificar la instrucción, obtiene información acerca de los operandos fuente, la operación a realizar con ellos y el lugar en el que se deberá almacenar el resultado. En la cuarta fase, la de ejecución de la instrucción, se realiza la ejecución propiamente dicha de la instrucción utilizando la información obtenida gracias a las dos primeras fases. Si analizamos lo anterior se puede deducir, por una parte, que la instrucción en sí es un tipo de información que puede almacenarse en memoria y, por otra, que la instrucción debe contener indicaciones acerca de: I) la operación que debe realizar el procesador, II) sus operandos fuente, y III) el destino del resultado.

Efectivamente, como se verá en un apartado posterior, la memoria almacena dígitos binarios —unos y ceros lógicos— llamados **bits**, agrupados en conjuntos de 8 que se denominan **bytes**. Cada una de las instrucciones que un procesador es capaz de interpretar se codifica utilizando un cierto número de bytes,[5] pudiendo haber instrucciones de la

[5] En algunos casos raros, siempre con arquitectura Harvard, el tamaño en bits de la instrucción puede no ser múltiplo de 8.

misma arquitectura que requieran de un número distinto de bytes. El grupo de bytes que constituye una determinada instrucción de una arquitectura dada se codifica siguiendo un formato concreto que se define durante el diseño de dicha arquitectura. Un **formato de instrucción** determina cómo codificar la información que contiene una instrucción, especificando los campos en los que se divide el conjunto de bits que forman dicha instrucción y el tamaño —número de bits— y contenido de cada campo. Cada uno de estos campos codifica una información diferente: I) lo que hace la instrucción, lo que se conoce como código de operación —abreviado generalmente como *opcode*, por *operation code* en inglés—, y II) los operandos fuente y destino, que se especifican mediante lo que se conoce como modos de direccionamiento.

Las instrucciones, vistas como valores o conjuntos de bytes en memoria, son el nivel de abstracción más bajo de los programas, lo que se conoce como **código o lenguaje máquina**. Sin embargo, tanto el procesador como sus instrucciones son diseñados por seres humanos, quienes también han diseñado una traducción del lenguaje máquina comprensible por los programadores humanos, llamada **lenguaje ensamblador**. En este lenguaje, los códigos de operación, que son números, se traducen por palabras o apócopes llamados mnemónicos que recuerdan, en inglés, la operación que realiza cada instrucción. Los datos, tanto fuente como destino, se incorporan separados por comas y con un formato previamente especificado y fácil de comprender. Cada instrucción en ensamblador se corresponde con una en código máquina,[6] si bien aquellas son texto legible en lugar de bytes en memoria.

El Cuadro 1.1 muestra, para tres arquitecturas de procesador representativas de los distintos tipos de arquitecturas existentes, algunas instrucciones expresadas en lenguaje ensamblador y en código máquina. El código máquina se ha representado utilizando la notación hexadecimal, donde cada byte se codifica utilizando dos cifras.

El Cuadro 1.2 muestra las instrucciones del Cuadro 1.1, pero identificando mediante distintos colores los diferentes campos de los que consta cada instrucción. También se muestra el código máquina de cada instrucción representado en binario, marcando qué bits de cada instrucción en código máquina codifican qué campo de la instrucción utilizando la misma codificación de colores que la usada en la instrucción en ensamblador. El formato de instrucción, establecido *a priori*, es quien determina esta división de cada instrucción en campos. El procesador y su conjunto de instrucciones se diseñan a la vez, de manera que los circuitos del procesador, al recibir los distintos campos a su entrada, son capaces de

[6] En realidad, el lenguaje ensamblador suele proporcionar más instrucciones que las estrictamente soportadas por la arquitectura. Estas instrucciones adicionales, llamadas **seudoinstrucciones**, y que facilitan la labor del programador, pueden dar lugar a una o más instrucciones máquina.

Ensamblador	Máquina	Operación
`addwf 0xD9, f, c`	`26D9`	Suma al acumulador el contenido de la dirección de memoria `0xD9` del banco común.
`movf 0x17, w, c`	`5017`	Copia en el acumulador el contenido de la dirección de memoria `0x17` del banco común.
`bz +8`	`E004`	Salta a la instrucción 8 posiciones después en memoria si el bit de estado Z es 1.

(a) Familia de procesadores PIC18 de 8 bits de microchip

Ensamblador	Máquina	Operación
`add r4, r5, r7`	`19EC`	Guarda en el registro `r4` la suma de los contenidos de los registros `r5` y `r7`.
`ldr r5, [r0, #44]`	`6AC5`	Copia en el registro `r5` el contenido de la dirección de memoria formada sumando 44 al contenido del registro `r0`.
`beq #-12`	`D0FA`	Salta a la instrucción 12 posiciones antes en memoria si el bit de estado Z es 1.

(b) Subconjunto Thumb de la arquitectura ARM

Ensamblador	Máquina	Operación
`addl $0x4000000,%eax`	`0504000000`	Suma al contenido del registro `eax` el valor constante `0x400 0000`.
`movl -4(%ebp),%edx`	`8B55FC`	Copia en el registro `edx` el contenido de la dirección de memoria formada restando 4 al contenido del registro `ebp`.
`je +91`	`745B`	Salta a la instrucción 91 posiciones después en memoria si el bit de estado Z es 1.

(c) Arquitectura Intel de 32 bits

Cuadro 1.1: Instrucciones de diferentes arquitecturas expresadas en ensamblador y en código máquina (representado en hexadecimal), junto con una descripción de la operación realizada por dichas instrucciones

realizar de manera electrónica las operaciones requeridas sobre los datos dados, para ejecutar correctamente la instrucción.

Además de los bits que forman el código de operación, que se muestran en azul, al igual que el mnemónico, en el resto de los campos se codifica cómo obtener los operandos fuente, o dónde almacenar el resultado de la acción, en el caso de las instrucciones que tienen operando destino.

Ensamblador	Máquina	Máquina en binario
`addwf 0xD9, f, c`	26D9	0010 0110 1101 1001
`movf 0x17, w, c`	5017	0101 0000 0001 0111
`bz +8`	E004	1110 0000 0000 0100

(a) Familia de procesadores PIC18 de 8 bits de microchip

Ensamblador	Máquina	Máquina en binario
`add r4, r5, r7`	19EC	0001 1001 1110 1100
`ldr r5, [r0, #44]`	6AC5	0110 1010 1100 0101
`beq #-12`	D0FA	1101 0000 1111 1010

(b) Subconjunto Thumb de la arquitectura ARM

Ensamblador	Máquina	Máquina en binario
`addl $0x4000000, %eax`	0504000000	0000 0101 0000 0100 0000 0000 0000 0000...
`movl -4(%ebp), %edx`	8B55FC	1000 1011 0101 0101 1111 1100
`je +91`	745B	0111 0100 0101 1011

(c) Arquitectura Intel de 32 bits

Cuadro 1.2: Instrucciones de diferentes arquitecturas en ensamblador y en código máquina (representado en hexadecimal y en binario). Los colores identifican los distintos campos de cada instrucción y los bits utilizados para codificar dichos campos en el código máquina

Como se puede deducir de los Cuadros 1.1 y 1.2, algunos procesadores tienen tamaños de instrucción fijos y formatos más o menos regulares, mientras que otros tienen una gran variedad de formatos y tamaños de instrucción variables. Esto tiene, evidentemente, implicaciones sobre la complejidad del procesador e, indirectamente, sobre su velocidad, aunque extenderse más sobre esto está fuera del alcance del presente texto.

1.2.5. Modos de direccionamiento

Si estudiamos lo que se ha comentado hasta ahora acerca del funcionamiento de un ordenador, se puede deducir que los operandos con que va a trabajar una instrucción pueden residir en tres lugares: I) en la propia instrucción,[7] II) en registros del procesador, y III) en memoria.[8] Por tanto, además de conocer en qué campo se encuentra cada operando, también es necesario saber cómo se codifica en dicho campo la dirección efectiva en la que se encuentra el operando. Así, el formato de instrucción, además de especificar, como se ha visto previamente, los campos en los que se divide el conjunto de bits que forman la instrucción y el tamaño y contenido de cada campo, también indica cómo codificar la dirección efectiva de cada uno de los operandos de la instrucción. Se denomina **dirección efectiva** a la dirección que acaba calculando el procesador y que indica la ubicación del operando.

Las distintas formas en las que se puede indicar la dirección efectiva de un operando reciben el nombre de **modos de direccionamiento**. Los modos de direccionamiento que referencian datos constantes o contenidos en registros son sencillos. Por otro lado, los que se refieren a datos almacenados en memoria son muy variados y pueden ser de gran complejidad. A continuación se describen los modos de direccionamiento más comunes, con las formas más habituales, si bien no únicas, de identificarlos.

El modo de direccionamiento **inmediato** —o literal, traduciendo la nomenclatura utilizada en inglés—, es aquel en el que el operando está codificado en la propia instrucción. Puesto que una instrucción no es un recurso de almacenamiento cuyo valor se pueda cambiar, sino un dato inmutable, si un operando utiliza este modo de direccionamiento, se tratará siempre de un operando fuente. Por otro lado, el rango de datos que se pueden especificar de esta forma depende del tamaño de las instrucciones. En arquitecturas con instrucciones de tamaño variable, como la Intel de 32 bits, el rango de valores puede ser muy grande, como en la primera instrucción de esta arquitectura que aparece en los Cuadros 1.1 y 1.2, en que la constante ocupa 4 bytes. En aquellas arquitecturas en las que el tamaño de instrucción es fijo, el rango puede estar limitado por un campo de 16, 8, 5 o incluso menos bits. Las ventajas de este modo de direccionamiento son que el operando está disponible desde el mismo momento en el que se lee la instrucción y que no es

[7] En el caso de que el operando esté en la propia instrucción, el dato referenciado será una constante y, por tanto, solo podrá actuar como operando fuente

[8] Como en muchos procesadores las instrucciones acceden de la misma forma a la entrada/salida que a la memoria, los datos podrían ubicarse igualmente en aquella. De nuevo, esta posibilidad no se comenta explícitamente por simplicidad.

necesario dedicar un registro o una posición de memoria para albergar dicho operando.

El modo **directo a registro** (véase la Figura 1.3) indica que el operando se encuentra en un registro de la arquitectura, pudiendo de este modo usarse tanto para operandos fuente como destino. La primera instrucción Thumb de los Cuadros 1.1 y 1.2 utiliza los registros `r5` y `r7` como operandos fuente, y el registro `r4`, como destino; en la segunda instrucción de las mostradas de Intel se utiliza el registro `edx` como destino y en la primera, el registro `eax` a la vez como fuente y destino, lo que es una característica de esta arquitectura. Los registros de un procesador suelen estar numerados internamente, lo que es evidente en el lenguaje ensamblador de ARM y no tanto en el de Intel, aunque realmente sí lo estén —p.e., el registro `edx` es el registro 2, tal y como se puede comprobar en la codificación en binario de la instrucción—. Una ventaja de usar registros como operandos es que la referencia a un registro consume muy pocos bits del código de instrucción comparado con el tamaño del operando contenido en el registro. Por ejemplo, dada una arquitectura con 16 registros de 32 bits, referenciar a un operando de 32 bits almacenado en uno de los 16 registros, consumiría únicamente 4 bits de la instrucción.

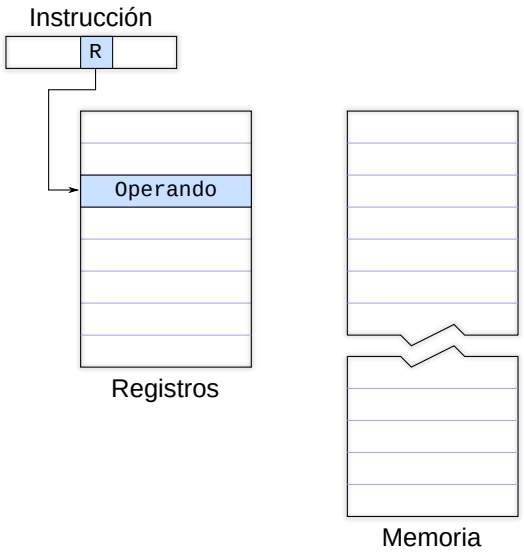


Figura 1.3: Modo de direccionamiento directo a registro. En este modo, la dirección efectiva del operando es uno de los registros de la arquitectura. En la instrucción se codifica el número de registro en el que se encuentra el operando

El modo más evidente de referirse a datos en memoria es el **directo a memoria** o **absoluto** (véase la Figura 1.4). La instrucción incorpo-

ra en el campo correspondiente la dirección de memoria del operando, que puede ser fuente o destino. El consumo de bits de la instrucción de este modo es muy elevado, por lo que es más común que se proporcione en arquitecturas con tamaño de instrucción variable —la arquitectura Intel lo incluye entre sus modos de direccionamiento—. No obstante, también puede encontrarse en arquitecturas con tamaño de instrucción fijo, aunque en estos casos se aplican restricciones a la zona de memoria accesible —como es el caso en la arquitectura PIC18—. A modo de ejemplo, las dos primeras instrucciones de los Cuadros 1.1 y 1.2 tienen sendos operandos que utilizan este modo. Los campos asociados a dichos operandos utilizan 8 bits para codificar una dirección de memoria, si bien el procesador puede acceder a 4 096 bytes, lo que requeriría en realidad 12 bits. La arquitectura PIC18 divide lógicamente la memoria en bancos, y los 4 bits de mayor peso, que faltan para formar la dirección, salen de un registro especial que identifica el banco activo. Para proporcionar más flexibilidad al programador a la hora de acceder a memoria, un bit adicional de la instrucción permite indicar si el acceso debe efectuarse en el banco activo o en un banco global común —las dos instrucciones de los Cuadros 1.1 y 1.2 utilizan el banco común, lo que se indica en lenguaje ensamblador mediante la c coloreada en naranja en ambas instrucciones—.

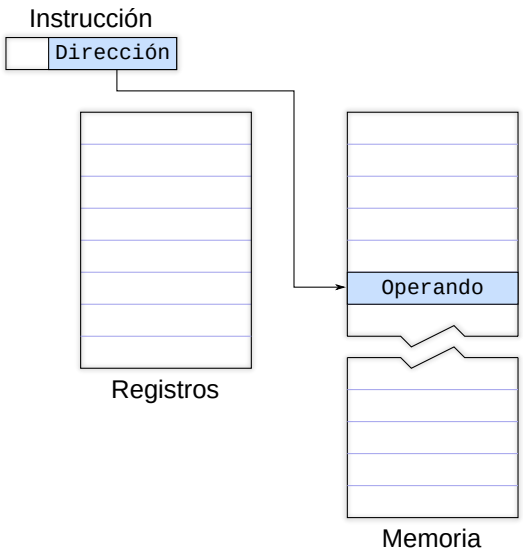


Figura 1.4: Modo de direccionamiento directo a memoria. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica la dirección de memoria en la que se encuentra el operando

El modo **indirecto con registro** (véase la Figura 1.5) permite referirse a datos en memoria consumiendo tan solo los bits necesarios para identificar un registro. En este modo, los bits de la instrucción indican el número de un registro, cuyo contenido es la dirección de memoria en la que se encuentra el operando, que puede ser fuente o destino. Debido a lo compacto de este modo, que utiliza muy pocos bits para referirse a datos en memoria, existen otros modos derivados de él más versátiles y que se adaptan a ciertas estructuras de datos en memoria habituales en los lenguajes de alto nivel.

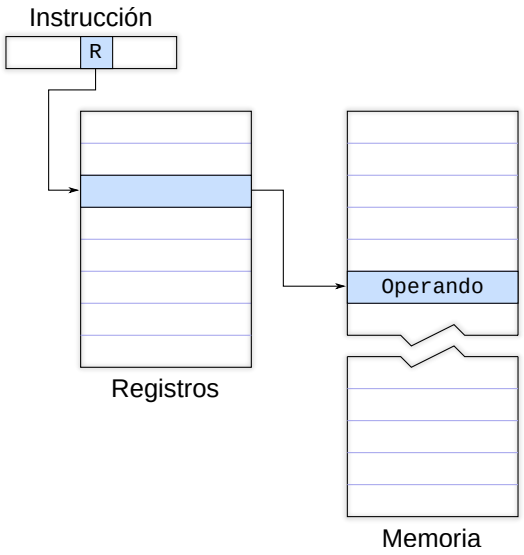


Figura 1.5: Modo de direccionamiento indirecto con registro. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica un número de registro, el contenido del cual indica la dirección de memoria en la que se encuentra el operando

El más común de los modos de direccionamiento derivados del indirecto con registro es el modo **indirecto con desplazamiento** (véase la Figura 1.6). En este modo, en la instrucción se especifica, además de un registro —como en el caso del indirecto con registro—, una constante que se suma al contenido de aquel para formar la dirección de memoria en la que se encuentra el operando. Las segundas instrucciones de las arquitecturas ARM e Intel presentes en los Cuadros 1.1 y 1.2 utilizan este modo para sus respectivos operandos fuente, sumando respectivamente 44 y -4 al contenido de sendos registros para obtener la dirección de sus operandos fuente. Un caso especial de este modo es aquel en el que el registro es el contador de programa. En este caso, a este modo se le llama **relativo al Contador de Programa** y es muy utilizado en las instrucciones de salto para indicar la dirección destino del salto.

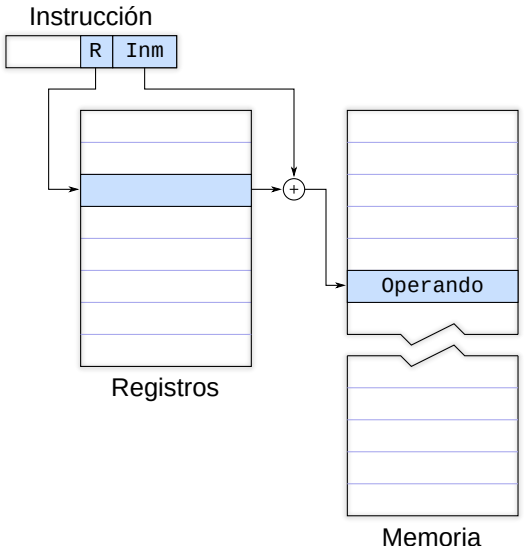


Figura 1.6: Modo de direccionamiento indirecto con desplazamiento. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica el número de un registro y un dato inmediato, la suma del contenido de dicho registro y el dato inmediato proporciona la dirección de memoria en la que está el operando

Otra variación del modo indirecto con registro es el modo **indirecto con registro de desplazamiento**, en el que en la instrucción se especifican dos registros, la suma de los contenidos de los cuales da la dirección de memoria en la que está el operando. En algunos casos, el contenido de uno de estos dos registros se multiplica[9] por el número de bytes que ocupa el operando. Si este es el caso, el registro cuyo contenido no se modifica se denomina **base**, y el otro, **índice**, por lo que este modo se puede llamar también **base más índice** o **indexado**.

Finalmente, cuando se tienen ambas cosas, dos registros y una constante, se tiene el modo **base más índice con desplazamiento**. En este último, la dirección de memoria del operando se calcula sumando el contenido de los dos registros, posiblemente multiplicando el índice como se ha comentado anteriormente, más la constante.

Por último, algunas instrucciones se refieren a algunos de sus operandos sin necesidad de indicar dónde se encuentran estos en ninguno de sus campos, dado que están implícitos en la propia instrucción. Por ello, este modo se direccionamiento se denomina **implícito**. Por ejemplo, las dos primeras instrucciones de la arquitectura PIC18 de los Cuadros 1.1

[9]Puesto que la multiplicación que se requiere en este caso siempre es por una potencia de dos, en lugar de una multiplicación se realiza un desplazamiento a la izquierda.

y 1.2 utilizan el acumulador `w` sin que ninguno de los bits del código máquina de dichas instrucciones se refiera explícitamente a él, al igual que ocurre en la primera instrucción de la arquitectura Intel, en la que el registro `eax` no es necesario codificarlo ya que esa instrucción utiliza dicho registro de forma implícita.

La lista anterior, lejos de ser exhaustiva, presenta únicamente los modos de direccionamiento más importantes. Además de los vistos, existen otros menos usados, algunos tan extraños como los modos doblemente —e incluso infinitamente— indirectos de arquitecturas ya en desuso, y otros más comunes como los que incluyen incrementos y decrementos automáticos de los registros que contienen direcciones de memoria, etcétera. También existen toda una serie de modos relativos al contador de programa, utilizados sobre todo en instrucciones de control de flujo del programa, pero también para acceder a datos en memoria. Concluyendo, los modos de direccionamiento, del tipo que sean, indican las formas mediante las cuales se emplean ciertos bits de las instrucciones para que el procesador pueda obtener los datos fuente o saber dónde almacenar el destino de las operaciones.

1.2.6. Arquitectura y organización del procesador

El término *arquitectura*, que hemos utilizado con cierta frecuencia a lo largo de este capítulo, cuando se emplea en el contexto de los ordenadores, hace referencia a su modelo de funcionamiento. Más precisamente, la **arquitectura de un ordenador** especifica su modo de comportarse y funcionar de tal manera que sea posible realizar programas correctos para ese ordenador. De manera análoga, **la arquitectura de un procesador** especifica cómo funciona un procesador incluyendo todos los aspectos necesarios para poder realizar programas correctos en lenguaje ensamblador. Aunque la arquitectura de un ordenador depende en algunos aspectos de la del procesador —o procesadores— que incluya, existen muchas diferencias entre ambas. Por ejemplo, así como el espacio de memoria direccionable viene fijado por la arquitectura del procesador, el mapa de memoria, que refleja la implementación del subsistema de memoria en un ordenador, se especifica en la de este.

Siguiendo la definición que se ha dado, centrada en el programador, se suele denominar también la arquitectura de un procesador como la **arquitectura del conjunto de instrucciones** o **ISA** —del inglés *Instruction Set Architecture*—. Exceptuando otras características del procesador tales como los modos de funcionamiento, la gestión de errores, excepciones e interrupciones, etcétera, que tienen que ver con el diseño y programación de sistemas, el conjunto de instrucciones es la manera más completa y objetiva de especificar cómo se comporta el procesador a la hora de ejecutar los programas y, por tanto, qué características de

él hay que tener en cuenta al programarlo. A continuación se presentan con más detalle estas características fundamentales que definen la arquitectura del conjunto de instrucciones. Como se verá, están íntimamente relacionadas entre sí, pues entre todas definen el funcionamiento del procesador.

En primer lugar se tienen los distintos **tipos de instrucciones** que conforman el conjunto de operaciones que es capaz de realizar un procesador y que define en buena medida su arquitectura. Además de las operaciones de transformación, que vienen dadas por las unidades funcionales, se tienen también los tipos y forma de los saltos, las maneras de acceder a memoria y a la entrada/salida, etcétera.

Los **tipos de datos** con que trabaja el procesador son también una parte importante de su arquitectura. Además de los distintos tamaños de enteros con que es capaz de operar, determina si trabaja con datos en coma flotante o si es capaz de interpretar, mediante sus instrucciones, otros formatos de datos.

Los **registros de la arquitectura**, su tamaño y su número, son otro parámetro fundamental. En particular, el tamaño de los registros define el de la arquitectura —de esta manera podemos hablar de una arquitectura de 32 o 64 bits, por ejemplo— lo que además da una aproximación cualitativa de la potencia del procesador. El tamaño de los registros también marca, de una u otra forma, el espacio de direcciones del procesador, pues las direcciones de memoria se almacenan en registros durante la ejecución de los programas. Por otro lado, un gran número de registros permite realizar menos accesos a memoria; pero como contrapartida, consume más bits en las instrucciones que los utilizan en sus modos de direccionamiento.

El **formato de las instrucciones** determina cómo se codifican las instrucciones e indica, además de otras circunstancias, el número y tipo de operandos con que trabajan las arquitecturas. Las de acumulador —por ejemplo, la PIC18— suelen especificar solo un operando en sus instrucciones, dado que el otro es implícitamente el acumulador. Además, uno de estos dos operandos es a la vez fuente y destino. De esta manera, el código puede ser muy compacto puesto que las instrucciones son pequeñas. Otras arquitecturas —como la Intel, que también mantiene ciertas características de arquitectura de acumulador— especifican solo dos operandos siendo también uno de ellos fuente y destino a la vez. Esto hace que las instrucciones sean pequeñas, pero también conlleva que siempre se modifique uno de sus operandos fuente. Las arquitecturas RISC —es el caso de la ARM— suelen tener instrucciones de tres operandos, siempre en registros —salvo en las instrucciones de acceso a memoria, obviamente—. El código resultante es de mayor tamaño, pero a cambio, no obliga a sobrescribir el operando fuente en cada instrucción.

Por último, los **modos de direccionamiento** indican la flexibilidad con que la arquitectura accede a sus operandos, sobre todo en memoria. Tener una gran variedad de modos de direccionamiento da mucha versatilidad, pero a costa de una mayor complejidad en el procesador y en los formatos de instrucción, que en muchos casos deben ser de tamaño variable. Por contra, la restricción a pocos modos de direccionamiento requerirá utilizar más instrucciones cuando sea necesario efectuar los tipos de accesos no directamente soportados —por ejemplo, cargar una constante en un registro para acceder a una dirección de memoria requerirá dos o tres instrucciones—, sin embargo, esta restricción simplifica el diseño del procesador y facilita el diseño de formatos de instrucción simples, de tamaño fijo.

Como se ha visto, el conocimiento para programar en lenguaje ensamblador un procesador, lo que define su arquitectura, se puede desglosar en cinco características fundamentales. La arquitectura constituye una especificación del procesador necesaria para generar el código de los programas y realizar aplicaciones útiles con ellos. Pero esta especificación también sirve para diseñar los circuitos electrónicos digitales que acaben implementando un procesador acorde con dicha arquitectura. El tipo de estos circuitos y la forma en que se conectan e interactúan entre sí para adaptarse a la arquitectura, se conoce como **organización del procesador**. Una misma arquitectura puede implementarse mediante distintas organizaciones, que darán lugar a procesadores más rápidos, más económicos..., es decir, a distintas implementaciones de la misma arquitectura.

1.2.7. Instrucciones y programas

La función de un ordenador y, por tanto, la de su procesador es la ejecución de programas de aplicación que lo doten de utilidad. En la definición de arquitectura se ha hablado explícitamente del lenguaje ensamblador, sin embargo, hoy en día es poco frecuente utilizar este lenguaje para el desarrollo de aplicaciones, ya que se suelen utilizar los llamados lenguajes de alto nivel. Las razones de este cambio son en parte históricas. En los orígenes de la informática se diseñaba el ordenador, incluyendo su unidad central de proceso, y se utilizaba el lenguaje ensamblador propio de dicho hardware para programarlo. Esto implicaba que un mismo programa debía desarrollarse de nuevo para cada nueva arquitectura en la que quisiera utilizarse. A medida que fueron apareciendo más sistemas, comenzaron a diseñarse lenguajes de alto nivel con los que era posible desarrollar programas independientes del hardware en el que fueran a utilizarse. Gracias a estos lenguajes, era posible desarrollar un programa una vez y que este pudiera ejecutarse en distintas arquitecturas. Tan solo era necesario disponer de un compilador del

lenguaje de programación de alto nivel utilizado para cada una de las arquitecturas en las que se quisiera ejecutar dicho programa. Conforme la fiabilidad de los compiladores iba mejorando, y gracias a su mayor comodidad e independencia del hardware, más populares se iban haciendo los lenguajes de programación de alto nivel. De hecho, la tendencia en el desarrollo de lenguajes de programación de alto nivel continuó de tal manera que llegó a influir en el diseño de las siguientes generaciones de procesadores. En particular, los conjuntos de instrucciones de los procesadores se diseñan desde hace tiempo para adaptarse a las estructuras utilizadas habitualmente en los lenguajes de alto nivel.

Es especialmente interesante a este respecto la división, a partir de los años ochenta del siglo pasado, entre las arquitecturas CISC, *Complex Instruction Set Computer*, y las más sencillas RISC, *Reduced Instruction Set Computer*. En un momento de la historia, los procesadores se complicaron tanto para adaptarse a los lenguajes de programación de alto nivel, que dejó de ser posible reducir más su ciclo de reloj. Entonces surgieron los procesadores RISC, que podían diseñarse con circuitos más simples y ciclos de reloj menores, ejecutando sus instrucciones en menos tiempo. Como contrapartida, necesitaban ejecutar más instrucciones para realizar las mismas tareas. A partir de ese momento dio comienzo una disputa entre cuál de las dos arquitecturas, CISC o RISC, constituía la mejor opción para diseñar un procesador. Sea cual sea el resultado de esta disputa, importe o no que exista un ganador, los conjuntos de instrucciones de los procesadores basados en una u otra arquitectura se han pensado para adaptarse, lo mejor posible dadas las restricciones de diseño, a los lenguajes de alto nivel.

En la actualidad, el conocimiento de la arquitectura del conjunto de instrucciones reside, más que en un programador humano, en el compilador de los lenguajes de alto nivel. Participando de este conocimiento, veamos cómo los conjuntos de instrucciones se relacionan con los lenguajes de programación y con los programas.

Las instrucciones de transformación determinan el conjunto de operaciones que el procesador puede realizar directamente sobre los datos propios de su arquitectura. Cualquier operación que no se encuentre en este conjunto, sea por la operación en sí, sea por los datos con que trabaja, deberá efectuarse por programa mediante un conjunto de instrucciones, con el consiguiente incremento en su tiempo de cálculo. Los lenguajes de alto nivel suelen incluir las operaciones básicas de suma, resta, multiplicación y división, sobre datos de tipo entero y real. Los enteros se codifican utilizando el complemento a dos y su rango se adapta al tamaño propio de la arquitectura. Los procesadores de gama baja, que no disponen de unidades de multiplicación o división incorporan, además de la ALU, una unidad de desplazamiento de bits. De esta manera, las multiplicaciones y divisiones pueden implementarse fácilmente

utilizando instrucciones de suma, resta y desplazamiento. Las operaciones en coma flotante entre números reales son más costosas en tiempo de ejecución, pues requieren operar independientemente con los exponentes y las mantisas. Por eso, la mayoría de los procesadores de propósito general incluyen unidades de operación en coma flotante, que se adaptan a los estándares establecidos tanto en el formato de los datos como en la precisión de las operaciones. Estas unidades, que no solo realizan las operaciones básicas sino también algunas funciones trascendentales, se utilizan mediante un subconjunto bien diferenciado de instrucciones del procesador que los compiladores utilizan de forma eficaz. Algunos procesadores incluyen otras unidades funcionales especializadas y, por lo tanto, otros subconjuntos de instrucciones. Es más complicado que los compiladores genéricos puedan aprovechar estas instrucciones, que suelen utilizarse mediante funciones de biblioteca que sí se han programado en lenguaje ensamblador.

La evolución de las unidades funcionales en los procesadores es más una consecuencia de los avances en las tecnologías de integración que de una adaptación a los lenguajes de alto nivel. Aún así, donde más se ha reflejado este seguimiento es en las formas de acceder a memoria que, aunque pueden ser parte de todas las instrucciones con operandos en memoria, se comenta a continuación para el caso de las instrucciones de transferencia de datos, junto con algunas características de dichas instrucciones. Las arquitecturas RISC suelen operar siempre con datos en registros, siendo las instrucciones de transferencia las únicas que acceden a datos en memoria, para llevarlos o traerlos a los registros. Este tipo de arquitecturas se llaman de carga/almacenamiento. Como se ha visto al inicio del capítulo, por regla general los datos residen en memoria y se llevan a los registros para operar con ellos y almacenarlos temporalmente. Los registros tienen un tamaño propio de la arquitectura que suele coincidir con el de los enteros en los lenguajes de alto nivel. Sin embargo, los procesadores permiten trabajar con datos de otros tamaños y, del mismo modo, las funciones de transferencia también permiten intercambiar al menos datos de tamaño byte —que se utilizan entre otras cosas para representar caracteres de texto—. Cuando un dato de menor tamaño se lleva desde la memoria a un registro, las arquitecturas suelen dar la posibilidad, mediante instrucciones distintas, de transferir rellenando el resto con ceros —lo que respondería a datos sin signo— o mantener el signo del dato de menor tamaño —poniendo unos o ceros, en función del signo, lo que se conoce como extensión de signo—. De esta manera, los conjuntos de instrucciones permiten a los lenguajes de programación trabajar con enteros de distintos tamaños.

Centrándonos por fin en los modos de direccionamiento, estos se han diversificado para adaptarse a las estructuras de datos de los programas. Vimos cómo el direccionamiento indirecto con registro permite acceder a

cualquier posición de la memoria especificando simplemente un registro. Al añadir un desplazamiento se permite que toda una región de datos de un programa, por ejemplo el conjunto de variables locales y parámetros de una función, pueda ser accesible sin necesidad de cambiar el valor del registro base, accediendo a cada variable individual mediante su propio desplazamiento. Añadir un segundo registro, base más índice, da un direccionamiento óptimo para trabajar con vectores. El registro base mantiene la dirección del inicio del vector y el índice selecciona el elemento de forma similar a como se hace en un lenguaje de alto nivel. Si la arquitectura permite además que el índice se multiplique según el tamaño en bytes de los datos, un acceso a un vector requiere una única instrucción en el lenguaje máquina. Añadir un desplazamiento a este modo permite intercambiar elementos dentro de un mismo vector aplicando un desplazamiento fijo, acceder a vectores de datos estructurados, donde el desplazamiento selecciona un campo particular dentro de cada elemento, etcétera.

Las instrucciones de control del flujo del programa o de salto son las que permiten implementar las estructuras de control en los lenguajes de alto nivel. Existen diversos criterios para clasificar las instrucciones de salto. Uno de ellos las divide en relativas o absolutas según la dirección de destino del salto sea relativo al contador de programa —es decir, un operando de la instrucción es un valor que se suma o resta al contador de programa—, o sea una dirección absoluta independiente de aquel. Otro las diferencia en incondicionales o condicionales, según el salto se verifique siempre que se ejecute la instrucción de salto o solo cuando se satisfaga cierta condición. Esta condición puede ser una operación entre registros que realiza la propia instrucción o una evaluación de uno o varios bits de estado del procesador —que a su vez se ven modificados por los resultados de las operaciones ejecutadas previamente—. Las estructuras de control suelen implementarse mediante saltos relativos al contador de programa, tanto condicionales como incondicionales. Para implementar una estructura iterativa —bucle *for*, *while*, etcétera— se utiliza una instrucción de salto condicional que resta al contador de programa. De esta manera, si la condición de permanencia en la estructura de control es cierta, la ejecución vuelve a una instrucción anterior, al inicio del bucle. Las estructuras condicionales *if-else* se implementan con saltos condicionales e incondicionales que suman al contador de programa. Un primer salto condicional dirige la ejecución al código de una de las dos alternativas, y un segundo salto incondicional evita que se ejecute el código de una alternativa en caso de que se haya ejecutado el correspondiente a la otra. Por último, existe en todas las arquitecturas un tipo especializado de instrucciones de salto que permiten que los programas se estructuren en subrutinas o funciones. Son instrucciones de salto que suelen utilizarse emparejadas. Una de ellas, denominada *llamada —call*,

en inglés— salta a un dirección, normalmente absoluta, y además guarda en algún registro o en memoria, la dirección de la siguiente instrucción —es decir, la que se habría ejecutado de no producirse el salto— llamada dirección de retorno. La segunda instrucción, llamada precisamente *de retorno*, permite modificar el contador de programa y, por lo tanto, saltar a una dirección almacenada en memoria o en un registro. De esta manera, mediante la primera instrucción podemos llamar a una subrutina desde cualquier parte de un programa y, mediante la segunda, volver a la instrucción siguiente al salto, puesto que la dirección de retorno depende de la dirección de la instrucción que realiza la llamada. De esta forma, la arquitectura de los procesadores provee instrucciones que permiten ejecutar las estructuras más utilizadas en los lenguajes de alto nivel.

1.3. Introducción a los buses

Un bus, en una primera aproximación muy simple, es un conjunto de conductores eléctricos por el que se intercambia información entre dos o más dispositivos electrónicos digitales. La información que circula a través del bus necesita de la participación de todas las líneas conductoras que lo conforman que, por ello, se consideran lógicamente agrupadas en un único bus.

Si profundizamos y extendemos un poco más esta definición, dado que la finalidad de los buses es comunicar dispositivos, todos los que se conecten al mismo bus para intercambiar información deben adaptarse a la forma en que dicha información se descompone entre las distintas líneas conductoras y, además, en las distintas etapas temporales de sincronización, transmisión, recepción, etcétera. Dicho de otro modo, para poder intercambiar información a través de un bus, los dispositivos conectados a dicho bus deben adaptarse a un conjunto de especificaciones que rigen el funcionamiento del bus y reciben el nombre de **protocolo de bus**. De esta manera, podemos completar la definición de bus diciendo que es un conjunto de conductores eléctricos por el que se intercambia información, mediante un protocolo adecuadamente especificado.

Como veremos en capítulos posteriores, los protocolos y otros aspectos que se tienen en cuenta en la especificación de un bus —número de conductores, magnitudes eléctricas empleadas, temporizaciones, tipos de información, etcétera— se extienden a varios niveles de abstracción, llegando, por ejemplo en el caso de los servicios de Internet[10] al formato de los mensajes que permiten la navegación web. En esta introducción vamos a limitarnos al nivel más sencillo de los buses que interconectan

[10]Internet especifica, entre otras cosas, los protocolos de una red de comunicaciones. Y una red es un tipo de bus.

el procesador con el resto del sistema, es decir, con la memoria y con los dispositivos de entrada/salida.

El bus principal del sistema es el bus que utiliza el procesador para interactuar con el resto de elementos principales del ordenador. En los ordenadores tipo PC actuales, cuyo bus principal es el PCI Express, este se gestiona a través de un dispositivo puente conectado directamente al bus del procesador, que se conoce como FSB —*Front Side Bus*—. Como hemos comentado previamente, el procesador se encarga de generar las señales eléctricas que sincronizan el funcionamiento del ordenador. Así pues, el procesador es el que inicia todas las transacciones del bus, a las que los demás dispositivos responden. Esta situación, utilizando la terminología propia de los buses, se define diciendo que el procesador es el único maestro del bus del sistema, del que todos los demás dispositivos son esclavos.[11]

El procesador, guiado por la ejecución de instrucciones, debe indicar al exterior: I) la dirección a la que dirige el acceso, II) si se trata de una lectura o una escritura y, posiblemente, III) el tamaño en bytes de los datos a los que quiere acceder. Además, en algunos sistemas también deberá indicar si el acceso es a una dirección de memoria o del subsistema de entrada/salida. Por tanto, el bus del procesador necesita tres tipos de información para realizar un acceso: I) la dirección, generada y puesta en el bus por el procesador; II) los datos, que los pondrá el procesador o la memoria —o algún dispositivo de entrada/salida— en el bus, según se trate de un acceso de escritura o de lectura; y III) algunas señales de control para indicar las características del acceso y para pautar la sincronización de acuerdo con el protocolo del bus. Según esto, en los buses se diferenciarán tres tipos de líneas: de direcciones, de datos y de control. Las de dirección permiten la selección de los dispositivos sobre los que se va a realizar el acceso; las de datos transfieren la información que se va a intercambiar entre los distintos componentes; y las de control indican cómo se lleva a cabo la transferencia. Todas las transacciones comienzan con el envío de la dirección a las líneas del bus, así como la activación de las señales de control y sincronización necesarias para que se lleve a cabo la operación. De esta manera, los dispositivos, junto con la circuitería de decodificación, tienen tiempo de que las señales eléctricas los activen y se configuren para enviar o recibir datos, según el tipo de acceso.

[11] Esto ya no es cierto en la mayor parte de sistemas. Para gestionar más eficazmente la entrada/salida, algunos dispositivos, y en particular los controladores de DMA, también pueden actuar como maestros del bus.

1.4. La memoria

La memoria principal es el dispositivo que, en la arquitectura von Neumann, almacena las instrucciones y los datos de los programas en ejecución. Esta visión tan clara de la memoria es susceptible sin embargo de muchas confusiones debido, fundamentalmente, a dos razones. Por una parte, porque la memoria es un dispositivo de almacenamiento al igual que los discos duros, aunque estos sean almacenamiento secundario. Por otra parte, porque la memoria ha sido, desde el origen de los ordenadores, un recurso escaso por su precio y lento en comparación con la velocidad del procesador, lo que ha llevado a concebir sistemas complejos de uso de la memoria que añaden confusión a la terminología y a los conceptos relacionados con esta.

Retomando los conceptos básicos, la memoria principal es el dispositivo que almacena las instrucciones y los datos de los programas en ejecución, es decir, aquellos con los que está trabajando el procesador. A través de su bus, el procesador genera accesos de lectura o escritura a los que la memoria —olvidamos la entrada/salida en esta descripción— responde entregando o recibiendo datos. Siguiendo este modelo se puede ver que la estructura lógica de la memoria es muy sencilla. La memoria es una colección ordenada de recursos de almacenamiento, de manera que cada uno de ellos está identificado por su dirección. Cuando se realiza una lectura, la memoria entrega el dato que tiene almacenado en la dirección que se le haya indicado. En caso de una escritura, la memoria guarda el dato que se le suministra en la dirección que se le haya proporcionado. Para aproximar este modelo sencillo de la memoria a la realidad, basta con definir qué datos almacena la memoria. En los sistemas actuales, y no es previsible que cambie en años, el elemento básico de almacenamiento en memoria es el byte. De esta manera, cada dirección de memoria se asocia con un byte y la unidad mínima de lectura o escritura en memoria es el byte. Sin embargo, el tamaño de las instrucciones y de los registros de la mayor parte de las arquitecturas es de varios —dos, cuatro u ocho— bytes, lo que hace que la mayor parte de los accesos a memoria sean a conjuntos de bytes, de un tamaño u otro según la arquitectura. Manteniendo la visión lógica de la memoria direccionable por bytes, los buses de los procesadores suelen tener suficientes líneas de datos para intercambiar a la vez tantos bits como tienen los registros de la arquitectura. De esta manera, un acceso indica en el bus de direcciones la dirección del byte más bajo en memoria y, mediante otras señales de control, cuántos bytes de datos van a intervenir en el acceso. Esto obliga a que las direcciones que aparecen en el bus sean siempre múltiplos del número máximo de bytes que se pueden transferir en paralelo. Así, si se quiere transferir el número máximo de bytes en un solo acceso, deberá hacerse desde una dirección que sea múltiplo de este

número. Si no se hace así, será necesario efectuar una transacción para la primera parte del dato y otra, para la parte restante. Este concepto recibe el nombre de **alineamiento de datos**. Se dice que un dato está alineado en memoria si este comienza en una dirección múltiplo de su tamaño en bytes. Muchas arquitecturas solo permiten accesos a datos alineados en memoria; otras sí que permiten acceder a datos desalineados, pero los accesos en este caso serán necesariamente más lentos.

Por otra parte, cuando se almacena en memoria un dato del procesador que ocupa varios bytes, además de los problemas de alineamiento que ya se han comentado, se tiene la posibilidad de hacerlo de dos maneras distintas. Si imaginamos un entero de n bytes como una cantidad expresada en base 256 —recordemos que un byte, conjunto de 8 bits, al interpretarlo como un número entero sin signo puede tener un valor entre 0 y 255— donde cada byte es un dígito, podemos decidir escribir el de mayor peso —el más a la izquierda siguiendo con el símil de la cantidad— en la dirección menor de las n que ocupa el entero, o en la mayor. Esta decisión depende del procesador, no de la memoria, pero afecta a cómo se almacenan en ella los datos que se extienden por múltiples bytes. Las arquitecturas que se han ido desarrollando a lo largo de la historia no se han decidido por ninguna de estas opciones y las han usado a su albedrío, hasta tal punto que hoy en día las arquitecturas no suelen especificar ninguna en particular, y los procesadores pueden configurar mediante algún bit si usan una u otra. Volviendo a ambas opciones, son tan importantes en arquitectura de computadores que reciben cada una su nombre particular. De este modo, si el byte de menor peso es el que ocupa la dirección más baja de memoria, la forma de almacenar los datos se dice que es **little endian**, pues se accede a la cantidad por su extremo menor. Si por el contrario, se sitúa el byte de mayor peso en la dirección más baja, la forma de almacenamiento se denomina, consecuentemente, **big endian**. A modo de ejemplo, la Figura 1.7 muestra cómo se almacenaría una palabra de 4 bytes en la dirección de memoria `0x2007 0004` dependiendo de si se sigue la organización *big endian* o la *little endian*.

Volviendo a la visión lógica de la memoria, esta se considera un único bloque de bytes desde la dirección mínima hasta la máxima, donde el rango de direcciones depende del procesador. Este rango es función de la cantidad de bits que componen la dirección que es capaz de emitir el procesador cuando va a realizar un acceso a memoria. Por otro lado, se sabe que en los sistemas hay distintos tipos de memoria, con diferentes tecnologías y usos. En un ordenador podemos encontrar memoria no volátil, normalmente llamada ROM, aunque hoy en día suele ser de tecnología Flash, que almacena el código de arranque y otras rutinas básicas del sistema; una buena cantidad de memoria RAM para los datos, el sistema operativo y los programas de aplicación; direcciones

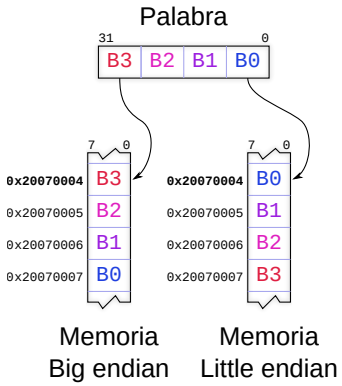


Figura 1.7: Los bytes de una palabra almacenada en la dirección de memoria `0x2007 0004` se dispondrán de una forma u otra dependiendo de si se sigue la organización *big endian* o la *little endian*

ocupadas por dispositivos de entrada/salida e incluso zonas de memoria no utilizadas. Así pues, la memoria direccionable de un ordenador está poblada por dispositivos de memoria de diferentes tecnologías e incluso en algunos casos, por dispositivos de entrada/salida. Los sistemas reales incorporan circuitos lógicos de decodificación que a partir de las líneas de mayor peso del bus de direcciones generan señales de activación que se conectan a las entradas de selección —*chip select*— de los distintos dispositivos para que respondan a los accesos del procesador. Esto configura lo que se conoce como mapa de memoria del ordenador, que configura el espacio de direcciones del procesador con los dispositivos presentes en el sistema real.

1.4.1. Arquitecturas von Neumann y Harvard

Al principio del capítulo se describió el modelo de funcionamiento de un ordenador según la arquitectura von Neumann. Se indicó así mismo que muchos ordenadores siguen hoy en día la arquitectura Harvard, que solo se distingue de la anterior en que mantiene dos memorias separadas, una para instrucciones y otra para datos. La ventaja de esta arquitectura se hace patente principalmente cuando las instrucciones pueden ejecutarse de forma solapada —es decir, cuando el procesador es capaz de comenzar la ejecución de una instrucción antes de haber terminado la ejecución de la instrucción anterior—. Con una memoria de instrucciones separada de la de datos, y por supuesto con buses de acceso también separados, el procesador puede estar accediendo a la memoria de datos, debido a la ejecución de la instrucción en curso, a la vez que adquiere la siguiente instrucción que va a ejecutar de la memoria de instrucciones. Por supuesto, la arquitectura Harvard también tiene inconvenientes con

respecto a la von Neumann. Si la memoria de datos es independiente de la de instrucciones, ¿cómo puede el procesador llevar los programas a la memoria, como ocurre cuando un sistema operativo quiere ejecutar una nueva aplicación que inicialmente reside en el disco?

Vistas estas disyuntivas, la realidad es que la mayor parte de los procesadores de propósito general, y buena parte de los microcontroladores —por ejemplo todos los de las familias PIC— implementan una arquitectura Harvard y de esta manera pueden ejecutar las instrucciones de forma solapada, y con mayor velocidad. La forma de llevar los programas a memoria es bien distinta en ambos casos. La solución de los procesadores potentes de propósito general es que las memorias se mantienen separadas en la caché —que es una memoria de acceso muy rápido que comentaremos brevemente en el siguiente apartado—. De esta manera, la memoria principal es única, para datos e instrucciones, y a este nivel se sigue la arquitectura von Neumann. Cuando el sistema operativo va a ejecutar una nueva aplicación lleva sus instrucciones a la memoria principal común. Cuando vaya a ejecutarse, el hardware del sistema se encargará de copiar trozos de este programa en la caché de instrucciones, antes de ejecutarlos siguiendo la arquitectura Harvard.

El caso de los microcontroladores es más sencillo: las memorias de datos e instrucciones están totalmente separadas. Es más, utilizan una tecnología distinta para cada una de ellas: RAM para la memoria de datos y Flash para la de instrucciones. En el caso de los microcontroladores, los programas se llevan a la memoria de instrucciones mediante un proceso que es externo al procesador. No obstante, si por alguna razón el procesador tuviera que modificar el código, cosa que normalmente no hará, podría hacerlo modificando bloques completos de la memoria de instrucciones, accediendo a esta como si fuera un dispositivo de entrada/salida.

1.4.2. Jerarquía de memoria

Además de la memoria caché que hemos introducido en el apartado anterior, al sistema de memoria se asocia a menudo el disco o almacenamiento secundario. Estos dos tipos de almacenamiento, junto con la memoria principal, constituyen lo que se llama la jerarquía de memoria de algunos sistemas.

La necesidad de organizar de esta manera un sistema de memoria se ha insinuado anteriormente. Los procesadores son muy rápidos, y llenar el mapa de memoria con grandes cantidades de memoria que permita accesos a esas velocidades no es práctico económicamente. De hecho, la memoria principal de los ordenadores de propósito general suele ser RAM dinámica, varias veces más lenta que el procesador. Para solucionar este problema, que se conoce como el cuello de botella de la memoria,

se añadió la memoria caché. Esta memoria, que funciona a la misma velocidad que el procesador, mantiene copias de los datos e instrucciones de memoria principal próximos a los está accediendo el procesador con mayor frecuencia. Cuando el procesador debe acceder a un dato o instrucción que no se encuentra en la caché, la memoria caché copia de la memoria principal no solo lo que necesita, sino toda la información cercana, ya que previsiblemente el procesador la necesitará en breve. Mediante esta técnica, y con tasas de acierto en la caché superiores al 90 %, se consigue una máxima del diseño de la jerarquía de memoria: que la velocidad de todo el sistema de memoria sea comparable a la del elemento más rápido, en este caso la memoria caché, y el tamaño, al del elemento más abundante, en este caso la memoria principal.

La gestión de copia y reemplazo de elementos de memoria en la caché se realiza por el hardware de la propia memoria caché. De forma similar conceptualmente, pero gestionada por el sistema operativo, se tienen los mecanismos de memoria virtual, que expanden el espacio de almacenamiento de la memoria principal recurriendo al almacenamiento secundario. Cuando una aplicación que ocupa memoria lleva tiempo sin utilizar ciertos bloques de memoria, estos son llevados al disco, con lo que se libera espacio en memoria para otros programas. De esta manera, la jerarquía de memoria se completa con el último elemento citado anteriormente: el almacenamiento secundario.

En la actualidad, los procesadores con varios núcleos pueden incorporar dos o más niveles de caché. El primero, el más rápido, es propio de cada núcleo y separa datos e instrucciones implementando la arquitectura Harvard. Los demás, más lentos, suelen ser unificados y compartidos por varios o todos los núcleos. A partir de ahí, el procesador se comunica a través de su bus con una única memoria principal para datos e instrucciones, según la arquitectura von Neumann. En los ordenadores basados en multiprocesadores suele ser posible instalar un sistema operativo con soporte para la gestión de memoria virtual que utilice el disco, completando así la jerarquía de memoria.

Parte II

Arquitectura ARM con QtARMSim

Capítulo 2

Primeros pasos con ARM y QtARMSim

Índice

En este capítulo se introducen el lenguaje ensamblador de la arquitectura ARM y la aplicación QtARMSim.

En cuanto 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. Utilizar el juego de instrucciones Thumb permite reducir el tamaño de los programas a prácticamente la mitad. Esto presenta dos ventajas. La primera y más evidente es que hace posible seleccionar dispositivos con menores requisitos de memoria y, por tanto, con un

menor coste de fabricación. La segunda es que puesto que el programa ocupa menos, el número de accesos a memoria también es menor, por lo que se disminuye el tiempo de ejecución, lo que de nuevo permite ajustar a la baja las características del dispositivo necesario para ejecutar dicho código, reduciendo aún más el coste de fabricación. Se entiende así que el haber proporcionado el juego de instrucciones Thumb haya sido uno de los motivos por los que la arquitectura ARM ha acaparado el mercado de los dispositivos empotrados.

Por otro lado, para escribir programas en ensamblador de ARM, ya sea con el juego de instrucciones de 32 bits o con el Thumb, se puede optar por dos convenios: el propio de ARM y el de GNU.[1] Aunque la sintaxis de las instrucciones será similar independientemente de qué convenio se siga, la sintaxis de la parte del código fuente que describe el entorno del programa —directivas, comentarios, etcétera— es totalmente diferente.

Por tanto, al programar en ensamblador para ARM es necesario elegir en qué juego de instrucciones —estándar o Thumb— se quiere programar, y qué convenio de ensamblador se quiere utilizar —ARM o GNU—. En este libro se utiliza el juego de instrucciones Thumb y el convenio del ensamblador de GNU, puesto que son los utilizados por QtARMSim.

En cuanto a QtARMSim, es un simulador formado por una interfaz gráfica multiplataforma y el motor de simulación ARMSim.[2] Proporciona un entorno de simulación de ARM multiplataforma, fácil de usar y que ha sido diseñado con el objetivo de ser utilizado en cursos de introducción a la arquitectura de computadores. QtARMSim y ARMSim se distribuyen bajo la licencia libre GNU GPL v3+ y pueden descargarse gratuitamente desde la siguiente página web: <`http://lorca.act.uji.es/project/qtarmsim`>.

Una vez identificada la versión del juego de instrucciones de ARM que se va a utilizar y descrito brevemente el simulador QtARMSim, el resto de este capítulo se ha organizado como sigue. El primer apartado realiza una breve introducción al ensamblador Thumb de ARM. El segundo describe la aplicación QtARMSim. Los siguientes dos apartados proporcionan información sobre ciertas características y directivas del ensamblador de ARM que serán utilizadas frecuentemente a lo largo del libro. En concreto, el Apartado 2.3 muestra cómo utilizar literales y constantes; y el Apartado 2.4 cómo inicializar datos y reservar espacio de memoria. Más adelante, el Apartado 2.5 introduce el firmware incor-

[1] GCC, the GNU Compiler Collection. <`https://gcc.gnu.org/`>

[2] ARMSim es un motor de simulación de ARM desarrollado por Germán Fabregat Llueca, se distribuye conjuntamente con QtARMSim.

porado en el simulador ARMSim. Finalmente, se proponen una serie de ejercicios.

2.1. Introducción al ensamblador Thumb de ARM

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

De hecho, antes de comenzar con el lenguaje ensamblador propiamente dicho, conviene recordar las diferencias 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 constituyen un código que el procesador —la máquina— es capaz de reconocer como una instrucción y, por tanto, ejecutar. Por ejemplo, un procesador basado en la arquitectura ARM reconocería el código dado por la secuencia de bits $\texttt{0001100010001011}_2$ 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 —veáse el cuadro sobre «Notación RTL»—).

Como se vio en el capítulo anterior, cada instrucción máquina codifica, por medio de los distintos bits que forman la instrucción, la siguiente información: I) la operación que se quiere realizar, II) los operandos con los que se ha de realizar la operación, y III) el operando en el que se ha de guardar el resultado. Es fácil deducir pues, que la secuencia de bits del ejemplo anterior, $\texttt{0001100010001011}_2$, sería distinta en al menos uno de sus bits en cualquiera de los siguientes casos: I) si se quisiera realizar una operación que no fuera la suma, II) si los registros fuente no fueran los registros `r1` y `r2`, o III) si el operando destino no fuera el registro `r3`.

Si tenemos en cuenta lo anterior, un **programa en código máquina** no es más que un conjunto de instrucciones máquina, y de datos, que cuando son ejecutadas por el procesador 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 para codificarlas en los unos y ceros que constituyen las correspondientes instrucciones máquina. Así, el lenguaje de programación que representa el lenguaje de la máquina, pero de una forma más cercana al lenguaje humano, recibe el nombre

Notación RTL

La notación RTL –del inglés *Register Transfer Language*— permite describir las operaciones llevadas a cabo por las instrucciones máquina de forma genérica, evitando la sintaxis propia de una arquitectura particular. Los principales aspectos de la notación RTL utilizada en este libro se muestran a continuación.

Para referirse al contenido de un registro, se utilizará el nombre de dicho registro. Así, cuando se describa una operación, `r4` hará referencia en realidad al contenido del registro `r4`, no al registro en sí.

Para indicar el contenido de una posición de memoria, se utilizarán corchetes. Así, [`0x2000 0004`] hará referencia al contenido de la dirección de memoria `0x2000 0004`. De igual forma, [`r4`] también hará referencia al contenido de una dirección de memoria, la indicada por el contenido de `r4`.

Para mostrar el contenido actual de un registro o de una posición de memoria, se utilizará el símbolo «=». Así por ejemplo, se utilizará «`r4` = 20» para indicar que el registro `r4` contiene el número 20.

Por último, para indicar una transferencia de datos, se utilizará el símbolo «←». Por ejemplo, para indicar que la dirección de memoria `0x2000 0004` se debe sobrescribir con la suma del contenido del registro `r4` más el número 1, se utilizará la siguiente expresión: «[`0x2000 004`] ← `r4` + 1».

de **lenguaje ensamblador**. Este lenguaje permite escribir las instrucciones máquina en forma de texto. Así pues, la instrucción máquina del ejemplo anterior, 0001100010001011_2, se escribiría en el lenguaje ensamblador Thumb de ARM como «`add r3, r1, r2`». Lo que obviamente es más fácil de entender que 0001100010001011_2, por poco inglés que sepamos.

Aunque el lenguaje ensamblador es más asequible para nosotros que las secuencias de ceros y unos, sigue estando estrechamente ligado al *hardware* en el que va a ser utilizado. 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 anterior instrucción se debe escribir como «`add $3, $1, $2`» en el lenguaje ensamblador de la arquitectura MIPS. Así pues, el lenguaje ensamblador entra dentro de la categoría de los **lenguajes de programación de**

bajo nivel, ya que está fuertemente relacionado con el *hardware* en el que se va a utilizar.

No obstante lo anterior, podemos considerar que los lenguajes ensambladores de las diferentes arquitecturas son más bien dialectos, en lugar de 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 en realidad tan radicalmente distintas desde el punto de vista de su programación en ensamblador.

Como ya se ha comentado, uno de los hitos en la evolución 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, 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 fuente en ensamblador y genera un fichero objeto con instrucciones en código máquina que el procesador es capaz de entender. Es fácil intuir 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 propio lenguaje ensamblador con el objetivo de hacer más fácil la programación. Así pues, el lenguaje ensamblador no se limita a reproducir el juego de instrucciones de una arquitectura en concreto en un lenguaje más cercano al humano, si no que también proporciona una serie de recursos adicionales destinados a facilitar la programación en dicho lenguaje. Algunos de estos recursos se muestran a continuación, particularizados para el caso del lenguaje ensamblador de GNU para ARM:

El término *ensamblador* se refiere a un tipo de programa que se encarga de traducir un fichero fuente escrito en un lenguaje ensamblador, a un fichero objeto que contiene código máquina, ejecutable directamente por el procesador.

Comentarios. Permiten dejar por escrito qué es lo que está haciendo alguna parte del programa y mejorar su legibilidad señalando las distintas 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 mucho más conveniente comentar el código para poder reconocer 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, ignora dicho carácter y el resto de la línea en la que está. Aunque también es posible utilizar el carácter «#» para indicar el comienzo de un comentario, el carácter «#» debe estar al principio de la línea —o precedido únicamente de espa-

cios—. Así que para evitar problemas, es mejor utilizar siempre «@» para escribir dichos comentarios. Por otro lado, en el caso de querer escribir un comentario que englobe varias líneas, es posible utilizar los delimitadores «/*» y «*/» para marcar dónde empieza y acaba el comentario, respectivamente.

Seudoinstrucciones. Extienden el conjunto de instrucciones disponibles para el programador. Las seudoinstrucciones no pueden codificarse en lenguaje máquina ya que no forman parte del repertorio de instrucciones de la arquitectura en cuestión. Son instrucciones proporcionadas por el lenguaje ensamblador para facilitar un poco la vida del programador. El programa ensamblador se encargará de traducir automáticamente cada seudoinstrucción por aquella instrucción máquina o secuencia de instrucciones máquina que realicen la operación correspondiente.

Etiquetas. Sirven para referirse a la dirección de memoria del elemento contenido en la línea en la que se encuentran. Para declarar una etiqueta, esta debe aparecer al comienzo de una línea, estar formada por letras y números y terminar con el carácter dos puntos, «:», no pudiendo empezar con 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. De esta forma, cuando más adelante encuentre una instrucción o directiva en la que aparezca dicha etiqueta, sustituirá la etiqueta por un valor numérico, que puede ser directamente la dirección de memoria asociada a dicha etiqueta o un desplazamiento relativo a la dirección de memoria de la instrucción actual.

La etiqueta «`main`» tiene un significado especial para QtARMSim: sirve para indicar al motor de simulación cuál es la primera instrucción a ejecutar, que puede no ser la primera del código.

Directivas. Sirven para informar al ensamblador sobre cómo interpretar el código fuente. Son palabras reservadas que el ensamblador reconoce. Se identifican fácilmente ya que comienzan con un punto.

Teniendo en cuenta lo anterior, una instrucción en lenguaje ensamblador tiene la forma general[3]:

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

El siguiente ejemplo de programa en ensamblador está formado por tres líneas. Cada una contiene una instrucción —que indica el nombre de la operación a realizar y sus argumentos— y un comentario —que comienza con el carácter «@»—. En la primer línea, además, se declara la etiqueta «`Bucle`», para que pueda ser utilizada por otras instrucciones para referirse a dicha línea. En el ejemplo, dicha etiqueta es utilizada

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

en la instrucción que hay en la tercera línea. Cuando se ensamble dicho programa, el ensamblador traducirá la instrucción «`bne Bucle`» por la instrucción máquina correspondiente a «`bne pc, #-8`». Es decir, sustituirá, sin entrar por el momento en más detalles, la etiqueta «`Bucle`» por un número, el «`-8`».

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

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 se pueda ensamblar y ejecutar. Por ejemplo, aún no se ha indicado dónde comienza el código. Un programa en ensamblador está compuesto en realidad por dos tipos de sentencias: **instrucciones**, que darán lugar a instrucciones máquina, y **directivas**, que informan al programa ensamblador cómo interpretar el código fuente. Las directivas, que ya habíamos introducido previamente entre los recursos adicionales del lenguaje ensamblador, se utilizan entre otras cosas para: I) informar al programa ensamblador de dónde se debe colocar el código en memoria, II) reservar espacio de memoria para el almacenamiento de las variables del programa, y III) inicializar los datos que pueda necesitar el programa.

Para que el programa anterior pudiera ser ensamblado y ejecutado en el simulador QtARMSim, sería necesario añadir la primera y la penúltima de las líneas mostradas a continuación (la última línea no es estrictamente necesaria).

02_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**». Esta directiva indica al ensamblador que lo que viene a continuación es el programa en ensamblador y que las instrucciones que lo forman deberán cargarse en la zona de memoria asignada al código ejecutable. En el caso del simulador QtARMSim, esto implica que el código que vaya a continuación de la directiva «**.text**» se cargue en la memoria ROM, concretamente a partir de la dirección de memoria `0x0018 0000`.

«**.text**»

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

«**wfi**»

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. En la práctica, tan solo tiene sentido hacerlo en el caso de que se quieran escribir comentarios sobre el código a continuación de dicha línea, ya que el ensamblador ignorará todo lo que aparezca después de esta directiva.

«**.end**»

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

02_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
```

2.2. Introducción al simulador QtARMSim

Como se ha comentado en la introducción de este capítulo, QtARMSim es un simulador compuesto de una interfaz gráfica multiplataforma y del motor de simulación ARMSim, que proporciona un entorno de simulación basado en ARM. QtARMSim 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/project/qtarmsim`>.

2.2.1. Ejecución, descripción y configuración

Para ejecutar QtARMSim, basta con pulsar sobre el icono correspondiente o ejecutar la orden «`qtarmsim`».

La Figura 2.1 muestra la ventana principal de QtARMSim cuando acaba de iniciarse. La parte central de la ventana principal 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 —estos paneles se describen más adelante—. El panel de mensajes muestra 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, etcétera. Ocultos tras el panel de mensajes, en sendas pestañas, se encuentran los paneles de volcado de memoria y del visualizador LCD —que se describirán más adelante—.

Si se acaba de instalar QtARMSim, 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 QtARMSim (véase la Figura 2.2) se debe seleccionar en el menú «`Edit`», la opción «`Preferences...`». En el cuadro de diálogo de preferencias se pueden observar dos paneles. El panel superior corresponde al motor de simulación ARMSim y permite configurar el nombre del servidor y el puerto que se utilizarán para conectar con él, la línea de comandos para ejecutarlo y su directorio de trabajo. Generalmente no será necesario cambiar la configuración por defecto de este panel. Por otro lado, el panel inferior corresponde al compilador cruzado de GCC para ARM. En dicho panel se puede indicar la ruta al ejecutable del compilador cruzado de GCC para ARM y las opciones de compilación que se deben pasar al compilador. En el caso de que QtARMSim no haya podido encontrar el ejecutable en una de las rutas por defecto del sistema, será necesario indicarla en el campo correspondiente.

¿Mostrar el cuadro de diálogo de preferencias?

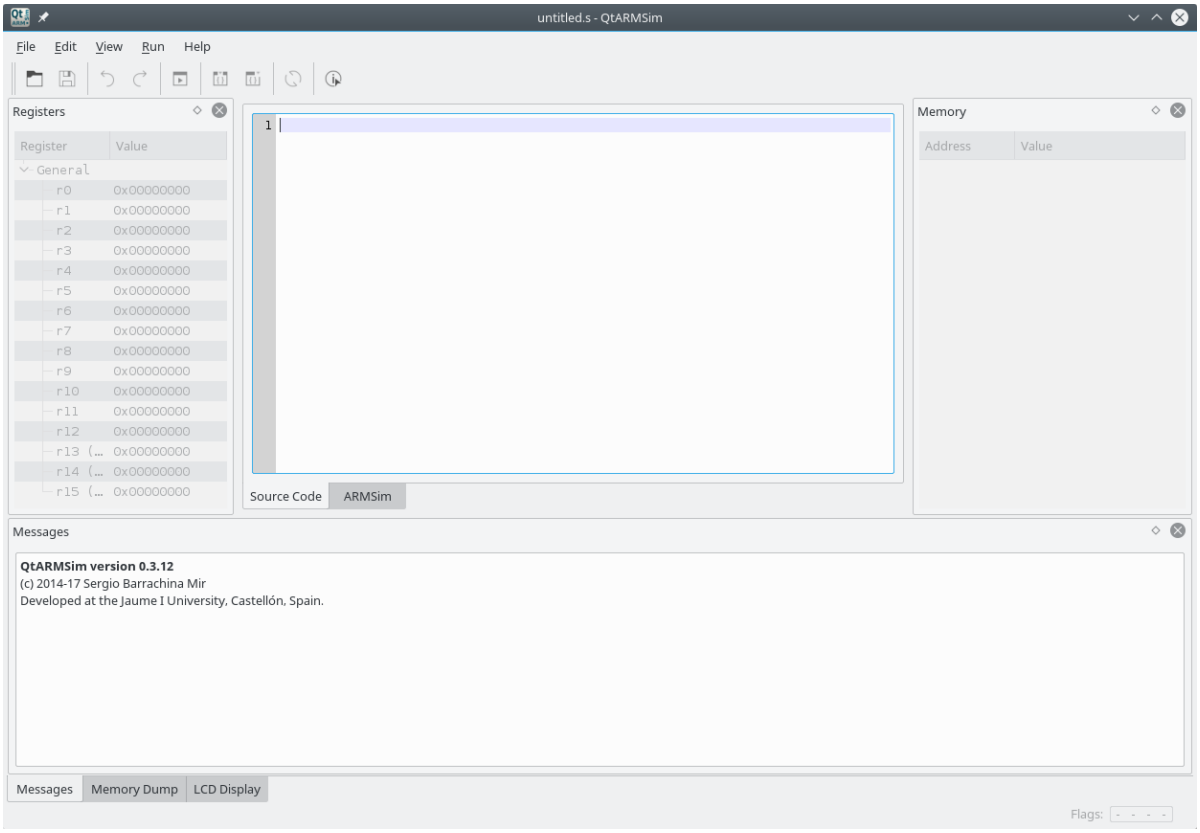


Figura 2.1: Ventana principal de QtARMSim

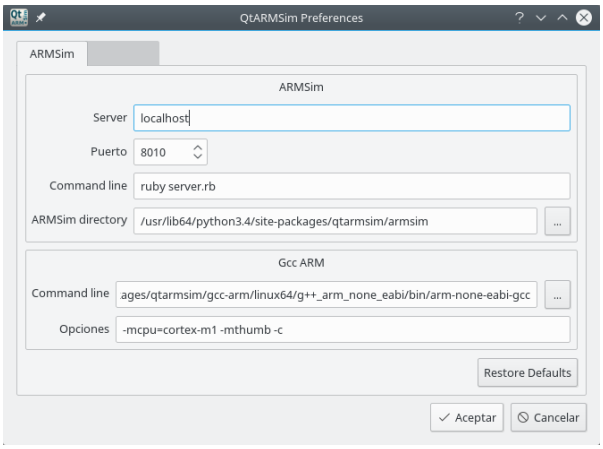


Figura 2.2: Cuadro de diálogo de preferencias de QtARMSim

2.2.2. Modo de edición

Cuando se ejecuta QtARMSim, este se inicia en el modo de edición. En este modo, como ya se ha comentado, 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 2.3 muestra la ventana de QtARMSim en la que se ha introducido el programa en ensamblador visto en el Apartado 2.1. Para abrir un fichero en ensamblador guardado previamente se puede seleccionar la opción del menú «`File > Open...`» o teclear la combinación de teclas «`CTRL+o`». Conviene tener en cuenta que antes de ensamblar y simular el código fuente editado, primero habrá que guardarlo (menú «`File > Save`», o «`CTRL+s`»), ya que lo que se ensambla es el fichero almacenado en disco. Los cambios que se hagan en el editor no afectarán a la simulación hasta que se guarden en disco.

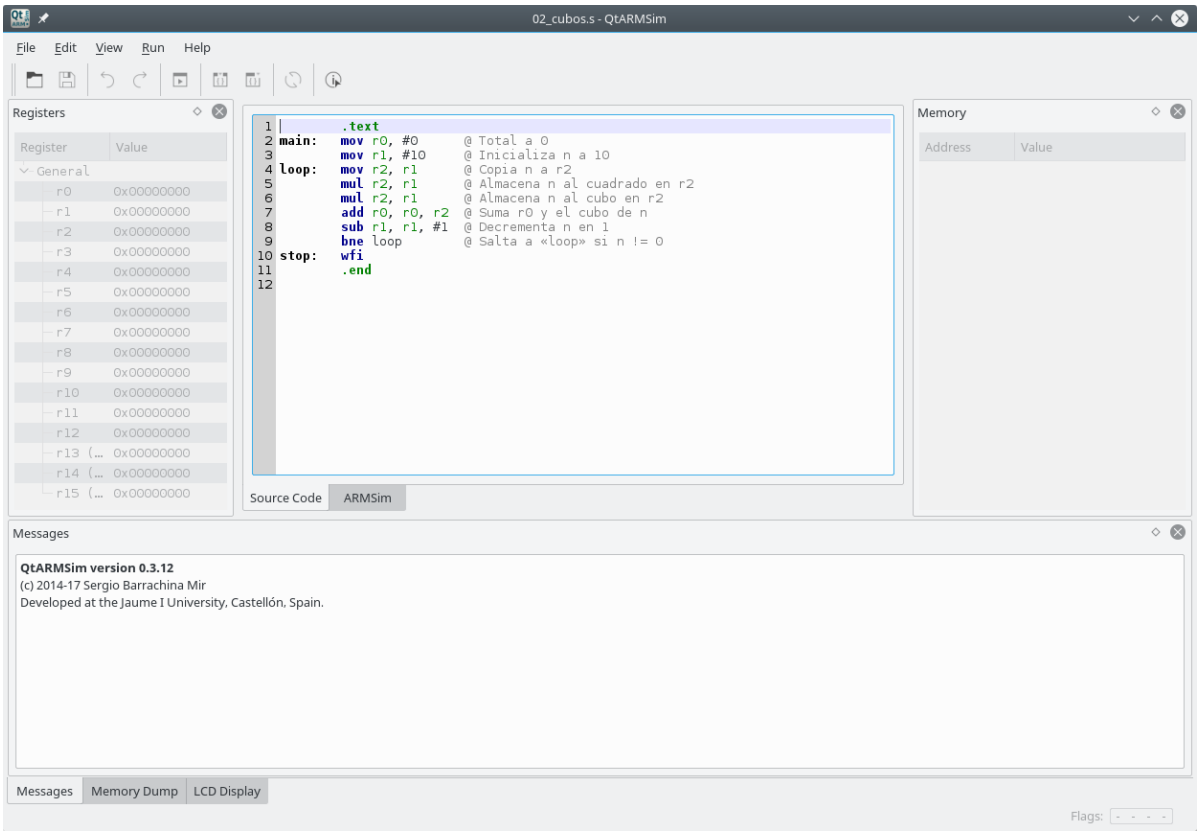


Figura 2.3: QtARMSim mostrando el programa «`02_cubos.s`»

2.2.3. Modo de simulación

¿Cambiar al modo de simulación?

Una vez se ha escrito y guardado en disco un programa en ensamblador, el siguiente paso es ensamblar dicho código para poder 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. Cuando se pasa al modo de simulación, la interfaz gráfica se conecta con el motor de simulación ARMSim, quien se encarga de realizar las siguientes acciones: I) llama al ensamblador de GNU para ensamblar el código fuente; II) actualiza el contenido de la memoria ROM con las instrucciones máquina generadas por el ensamblador; III) inicializa, si es el caso, el contenido de la memoria RAM con los datos indicados en el código fuente; y, por último, IV) inicializa 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 simplemente a un error de sintaxis en el código fuente. La Figura 2.4 muestra la apariencia de QtARMSim cuando está en el modo de simulación.

Si se compara la apariencia de QtARMSim cuando está en el modo de edición (Figura 2.3) con la de cuando está en el modo de simulación (Figura 2.4), 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, volverán a activarse.

¿Volver al modo de edición?

De vuelta en el modo de simulación, es posible consultar el contenido de la memoria del computador simulado en los paneles de memoria y de volcado de memoria. Tal y como se puede ver en la Figura 2.4, el computador simulado dispone de cuatro bloques de memoria: I) un bloque de memoria ROM que comienza en la dirección `0x0018 0000`, en el que se encuentra el código máquina correspondiente al programa ensamblado; II) otro bloque de memoria ROM que comienza en la dirección de memoria `0x0019 0000`, que contiene el código máquina del firmware de ARMSim (véase el Apartado 2.5); III) un bloque de memoria RAM que comienza en la dirección `0x2007 0000`, que se utiliza para almacenar los datos requeridos por el programa; y, por último, IV) otro bloque de memoria RAM, que comienza en la dirección `0x2008 0000` y que está mapeado con el visualizador LCD. Por otra parte, en la Figura 2.6 se puede observar que es posible consultar el código máquina de un programa en tres sitios: en el panel de memoria —a la derecha—, en el panel de volcado de la memoria —abajo— y, como se verá más adelante, en la ventana

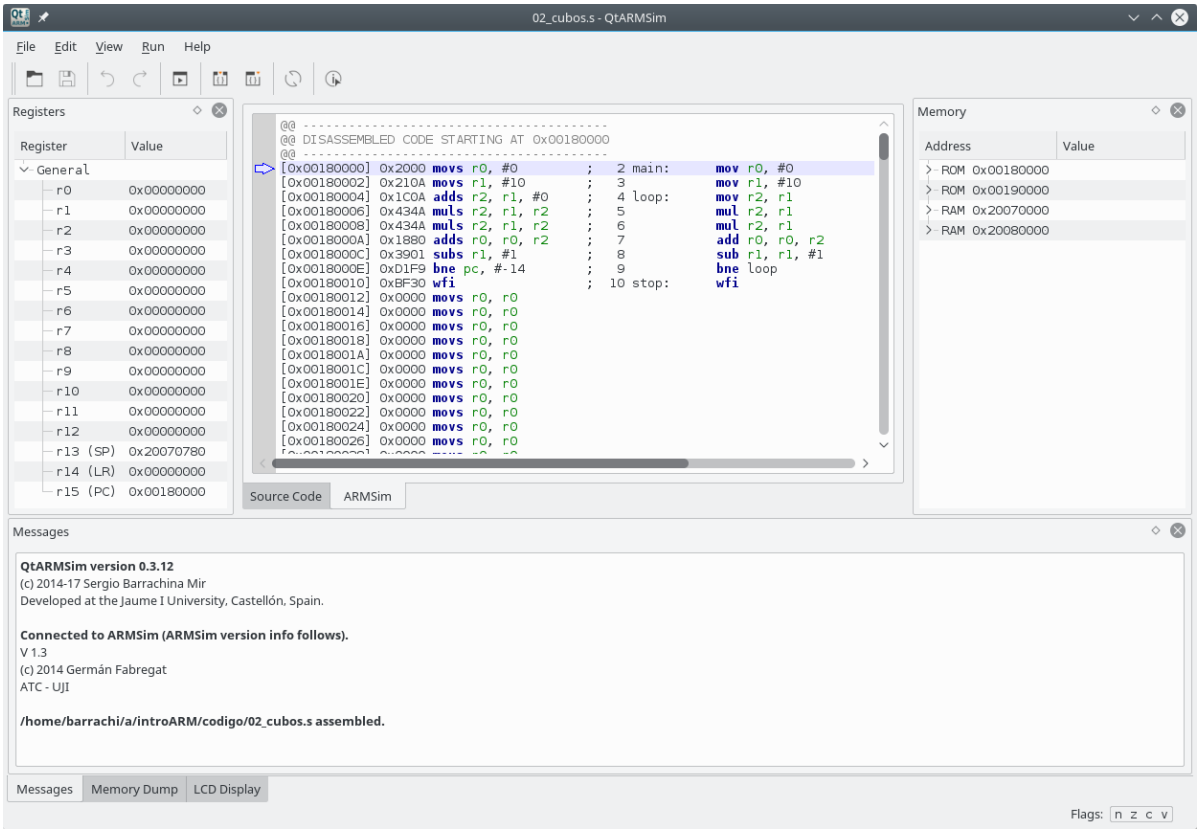


Figura 2.4: QtARMSim en el modo de simulación

central. Así pues, el contenido de la memoria ROM puede visualizarse de tres formas distintas: I) a nivel de palabra, en el panel de memoria; II) a nivel de byte, en el panel de volcado de memoria; y III) a nivel de instrucción —media palabra o una palabra—, en la ventana central.

Además, el simulador también permite visualizar el contenido de los registros, del `r0` al `r15`, 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 2.4, el `PC` contiene en este caso la dirección de memoria `0x0018 0000`, que, como se ha comentado en el párrafo anterior, es justamente la dirección de memoria en la que comienza el bloque de memoria ROM en el que se encuentra almacenado el programa en código máquina que se quiere simular. Por tanto, en este caso, el `PC` está apuntando a la primera instrucción presente en la memoria ROM, por lo que la primera instrucción en ejecutarse en este ejemplo será justamente la primera del programa.

El contador de programa, `PC`, es un registro del procesador que contiene la dirección de memoria de la siguiente instrucción a ejecutar.

Conviene destacar que 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. A modo de ejemplo, la Figura 2.5 muestra la ventana principal del simulador tras cerrar los paneles en los que se muestran los registros y la memoria. También es posible restaurar rápidamente la disposición por defecto del simulador seleccionando la entrada «`Restore Default Layout`» del menú «`View`» (o pulsando la tecla «`F3`»). Naturalmente, se pueden volver a mostrar los paneles que han sido cerrados previamente, sin necesidad de restaurar la disposición por defecto. Para hacerlo, se debe marcar en el menú «`View`», la entrada correspondiente al panel que se quiere volver a mostrar.

¿Cómo restaurar la disposición por defecto?

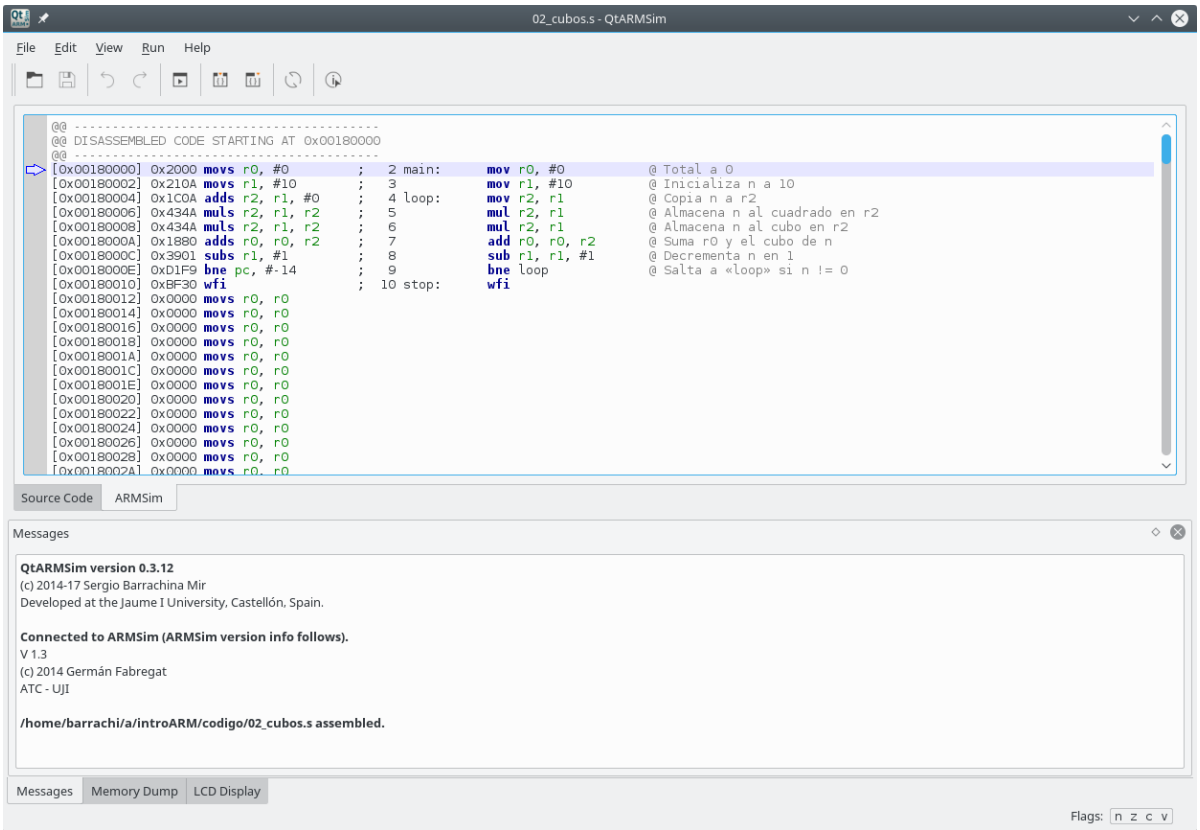


Figura 2.5: QtARMSim sin paneles de registros y memoria

En el modo de simulación, cada línea de la ventana central muestra la información correspondiente a una instrucción máquina. Esta información se obtiene por medio de un proceso que se denomina **desensamblado**, y que se realiza a partir del contenido de la memoria ROM.

La información mostrada para cada instrucción máquina, de izquierda a derecha, es la siguiente:

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

2.º La instrucción máquina expresada en hexadecimal.

3.º La instrucción máquina desensamblada.

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

Si tomamos como ejemplo la primera línea de la ventana de desensamblado de la Figura 2.5, su información se interpretaría de la siguiente forma:

- La instrucción máquina está almacenada en la dirección de memoria `0x0018 0000`.
- La instrucción máquina expresada en hexadecimal es `0x2000`.
- La instrucción máquina desensamblada es «`movs r0, #0`».
- La instrucción máquina se ha generado a partir de la línea número 2 del código fuente original, cuyo contenido es:
  «`main: mov r0, #0 @ Total a 0`»

▶ **2.2** Abre el simulador, copia el siguiente programa, pasa al modo de simulación y responde a las siguientes preguntas:

02_suma.s

```
1         .text
2 main:   mov r0, #2      @ r0 <- 2
3         mov r1, #3      @ r1 <- 3
4         add r2, r0, r1  @ r2 <- r0 + r1
5 stop:   wfi
```

2.2.1 Localiza la instrucción «`mov r0, #2`», ¿en qué dirección de memoria se ha almacenado?

2.2.2 ¿Cómo se codifica dicha instrucción máquina (en hexadecimal)? (Consulta para ello el segundo campo de la línea mostrada en la ventana central correspondiente a dicha instrucción.)

2.2.3 Localiza el número anterior en los paneles de memoria y de volcado de memoria.

2.2.4 Localiza la instrucción «`mov r1, #3`», ¿en qué dirección de memoria se ha almacenado?

2.2.5 ¿Cómo se codifica la anterior instrucción en código máquina (en hexadecimal)?

2.2.6 Localiza el número anterior en los paneles de memoria y de volcado de memoria.

. . . . . .

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 2.6 muestra la ventana de QtARMSim después de ejecutar el código máquina generado al ensamblar el fichero «`02_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 se han sobrescrito durante la ejecución del código máquina —no se ha resaltado ninguna de las posiciones de memoria debido a que el programa de ejemplo no accede a la memoria de datos—. En este caso, la ejecución del código máquina ha modificado los registros `r0`, `r1` y `r2` durante el cálculo de la suma de los cubos de los números del 10 al 1. También ha modificado el registro `r15`, el contador de programa, que ahora apunta a la última instrucción máquina del programa.

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:

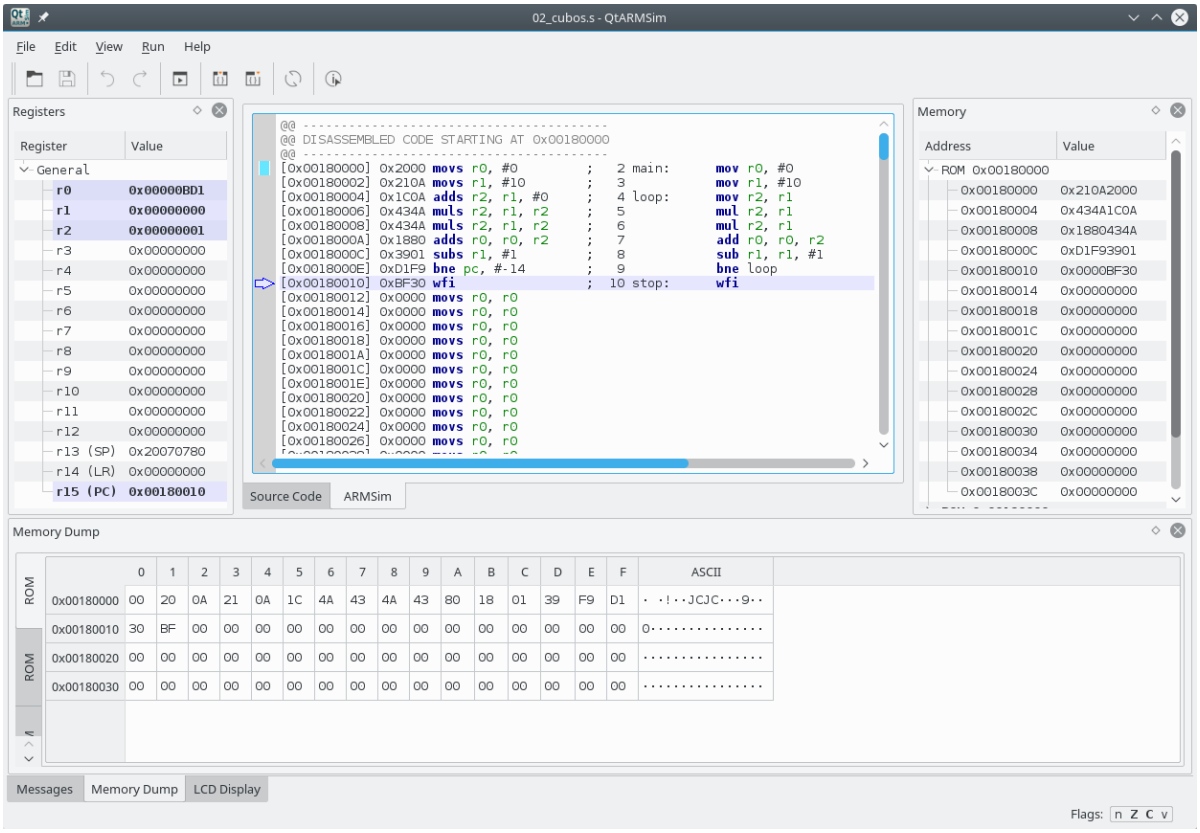


Figura 2.6: QtARMSim después de ejecutar el código máquina

```
$ python3 codigo/02_cubos.py
El resultado es: 3025
El resultado en hexadecimal es: 0x00000BD1
```

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 «`02_cubos.s`». Si además de saber qué es lo que hace el programa «`02_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 2.6. 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.

▶ **2.3** Ejecuta el programa «`02_suma.s`» del ejercicio 2.2, ¿qué valores acaban teniendo los siguientes registros?

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

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 este 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 al estar apuntando el PC a una instrucción de paro, el simulador ejecutará esa instrucción y, al hacerlo, 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, será necesario recargar previamente la simulación, lo que restaurará el sistema al estado inicial. 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 es muy útil si se quiere 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 el programa 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. A continuación se muestra cómo podría hacerse esto último. La Figura 2.7 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. 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—.

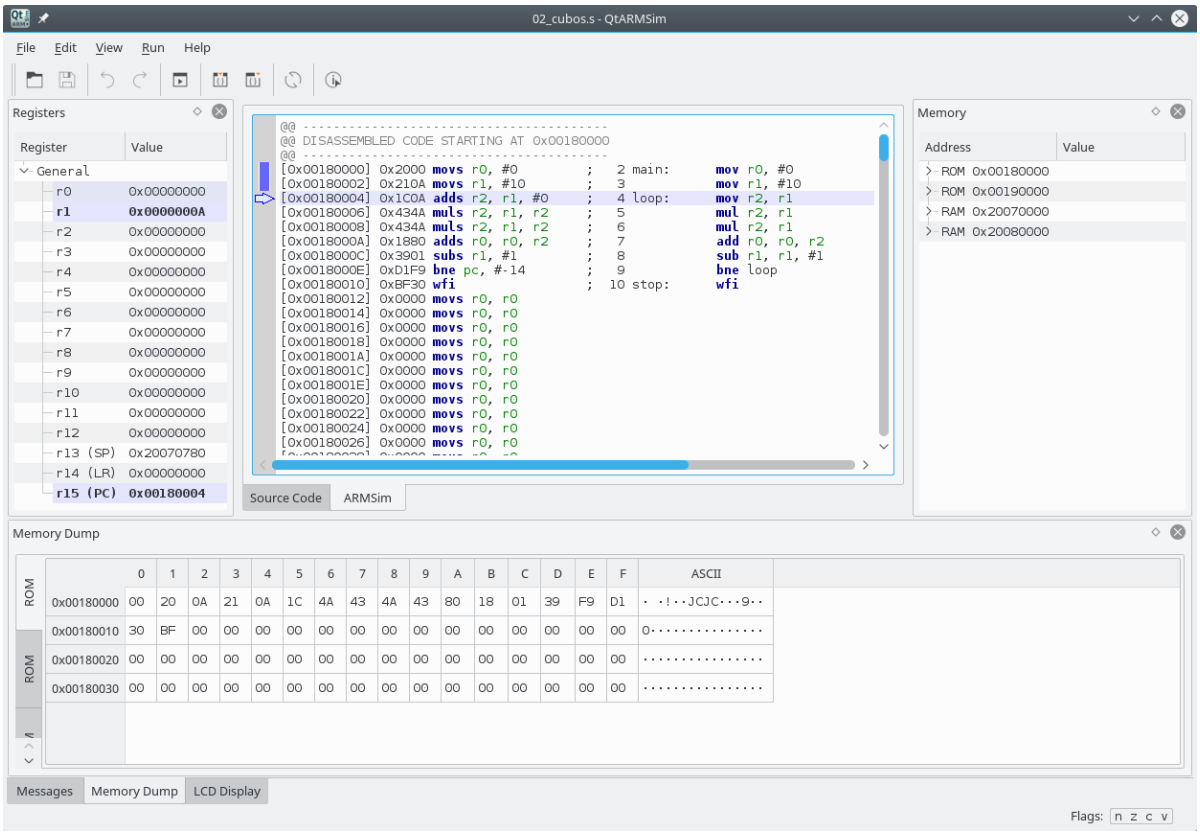


Figura 2.7: QtARMSim después de ejecutar dos instrucciones

¿Cómo modificar el contenido de un registro con el simulador?

Para modificar el contenido del registro `r1` con el simulador, se debe hacer doble clic sobre la celda en la que está su contenido actual (véase la Figura 2.8), teclear el nuevo número y pulsar la tecla «`Retorno`». El

nuevo valor numérico[4] puede introducirse en decimal (p.e., «3»), en octal (si se precede de «0», p.e., «03»), 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 para que contenga el valor 3, recuerda que es necesario pulsar la tecla «`Retorno`» para actualizar su valor. 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, y que efectivamente es $3^3 + 2^3 + 1^3$.

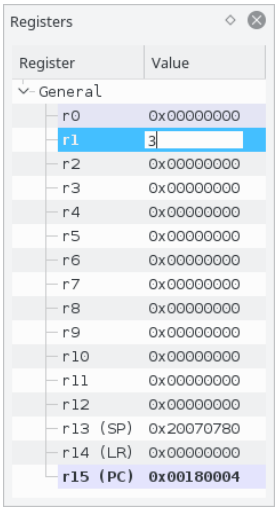


Figura 2.8: Edición del registro `r1`

Conviene comentar que en realidad existen dos modalidades de ejecución paso a paso. La primera de ellas, la ya comentada, menú «`Run > Step Into`», ejecuta siempre una única instrucción, pasando el PC a apuntar a la siguiente. La segunda modalidad tiene en cuenta que los programas suelen estructurarse por medio de subrutinas (también llamadas procedimientos, funciones o rutinas). Si el código en ensamblador incluye llamadas a subrutinas, al utilizar el modo de ejecución paso a paso visto hasta ahora sobre una instrucción de llamada a una subrutina, la siguiente instrucción que se ejecutará será la primera instrucción de la subrutina. Sin embargo, en ocasiones no interesa tener que ejecutar paso a paso todo el contenido de una determinada subrutina, siendo preferible ejecutar la subrutina entera como si de una única instrucción

Una *subrutina* 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.

[4] 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.

se tratara, y que una vez ejecutada la subrutina, el PC pase a apuntar directamente a la instrucción siguiente a la de la llamada a la subrutina. De esta forma, sería fácil para el programador ver y comparar el estado del computador simulado antes de llamar a la subrutina y justo después de volver de ella. Para poder hacer lo anterior, se proporciona una modalidad 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 únicamente cuando la instrucción que se vaya a ejecutar sea una instrucción de llamada a una subrutina. 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, se podría estar interesado 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 se quiere inspeccionar con más detenimiento. Por tanto, sería conveniente disponer de una forma de indicarle al simulador que ejecute las partes del código que no interesa ver con detenimiento y que solo se detenga cuando llegue a aquella instrucción a partir de la cual se quiere realizar una ejecución paso a paso —o en la que se quiere 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.

Un punto de ruptura define un lugar en el que se desea parar la ejecución de un programa, con la intención de depurar dicho programa.

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 eliminar un punto de ruptura ya definido, se debe proceder de la misma forma, se debe hacer clic sobre la marca del punto de ruptura. A modo de ejemplo, la Figura 2.9 muestra la ventana de QtARMSim 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 `0x0018000E`. Por su parte, la Fi-

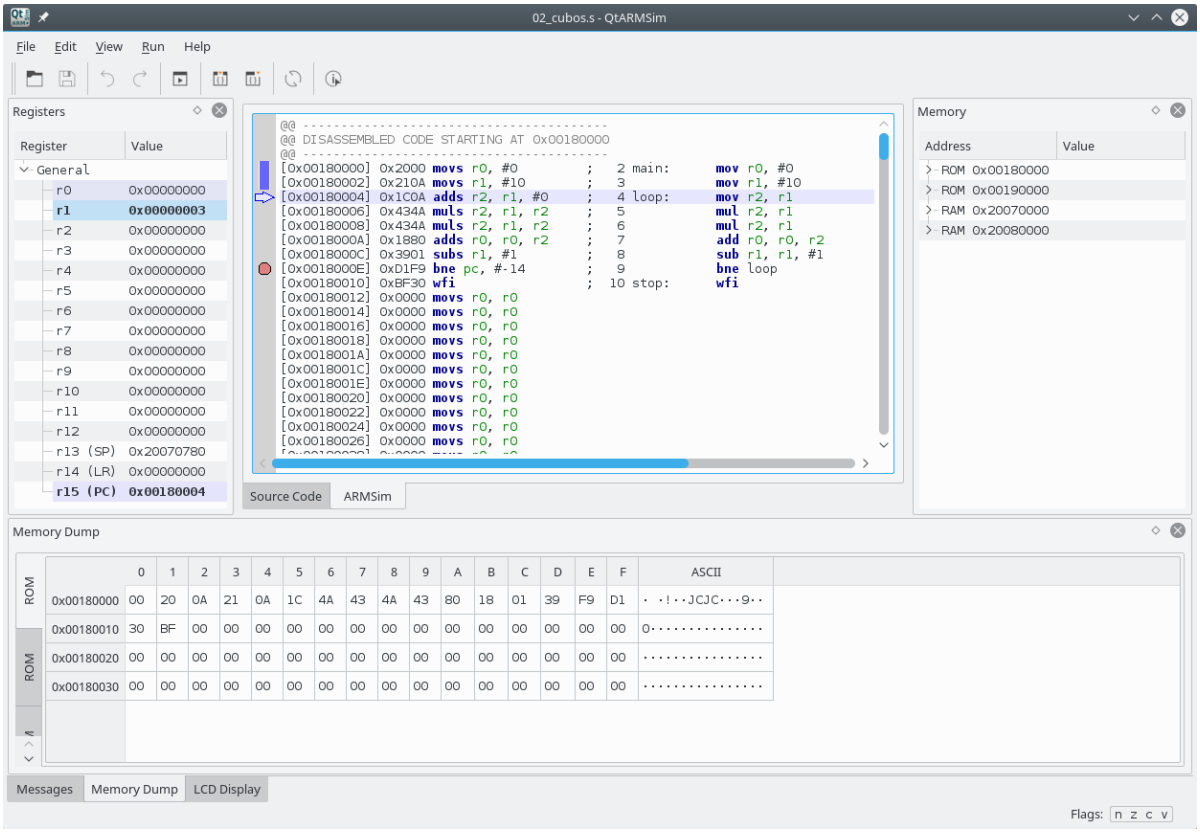


Figura 2.9: Punto de ruptura en la dirección `0x0018000E`

gura 2.10 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).

2.3. Literales y constantes en el ensamblador de ARM

Un **literal** es un número —expresado en decimal, binario, octal o hexadecimal—, un carácter o una cadena de caracteres que se indican tal cual en el programa en ensamblador. En el ensamblador de ARM, cuando se utiliza un literal como parte de una instrucción, este debe estar precedido del carácter «`#`», tal y como se ha visto en algunos de los ejemplos de instrucciones presentados en los apartados anteriores. Por ejemplo, el «`10`» de la instrucción «`mov r1, #10`» —que se codifica como `0x210A`— indica que se quiere guardar un 10 en el registro destino

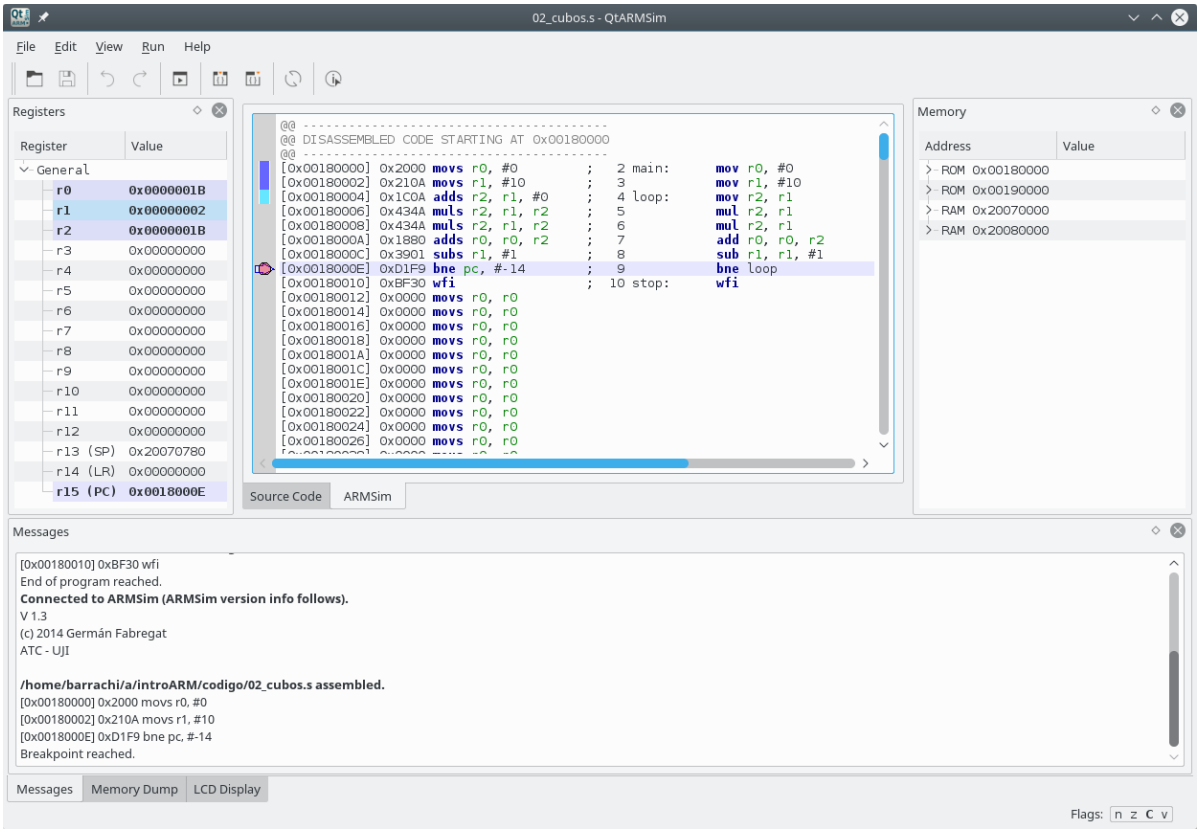


Figura 2.10: Programa detenido al llegar a un punto de ruptura

(en notación RTL: $r1 \leftarrow 10$). El «`10`» de la instrucción «**`mov r1, #10`**» es un valor literal.

Como la representación de un literal que se utiliza más frecuentemente es la de un número en decimal, esta forma de indicar un número es la más sencilla de todas. La única precaución que hay que tener es que el número no puede comenzar por 0, por el motivo que se verá en el siguiente párrafo. Por ejemplo, como ya se ha visto, la instrucción «**`mov r1, #10`**» guarda un 10, especificado de forma literal y en decimal, en el registro `r1`.

Aunque especificar un dato en decimal es lo más habitual, en ocasiones es más conveniente especificar dicho número en hexadecimal, en octal o en binario. Para hacerlo, se debe emplear uno de los siguientes

prefijos: «`0x`» para hexadecimal, «`0`» para octal y «`0b`» para binario;[5] y, a continuación, el número en la base seleccionada.

▶ **2.4** El siguiente código muestra cuatro instrucciones que inicializan los registros `r0` al `r3` con cuatro valores numéricos literales en decimal, hexadecimal, octal y binario, respectivamente. Copia dicho programa en QtARMSim, cambia al modo de simulación y contesta las siguientes preguntas.

02_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
```

2.4.1 Cuando el simulador desensambla el código, ¿qué ha pasado con los números? ¿están en los mismos sistemas de numeración que el código en ensamblador original?, ¿en qué sistema de numeración están ahora?

2.4.2 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, este se debe entrecomillar entre comillas simples,[6] p.e., «`mov r0, #'L'`». 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, «`msg: .asciz "Hola mundo!"`».

Aunque ya se ha comentado, conviene hacer hincapié en que cuando se utiliza un literal en una instrucción, este debe estar precedido por el símbolo «`#`», tal y como se puede ver en muchos de los ejemplos precedentes.

[5] 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; es por dicho motivo que cuando el número literal esté en decimal, este no puede empezar por cero.

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

Cambio de sistemas de numeración con calculadora

Las calculadoras proporcionadas con la mayoría de sistemas operativos suelen proporcionar alguna forma de trabajar con distintos sistemas de numeración. Por ejemplo, en Windows se puede seleccionar la opción del menú «`Ver > Programador`» de la calculadora del sistema para realizar los cambios de base más frecuentes. De forma similar, en GNU/Linux se puede seleccionar la opción del menú «`Preferencias > Modo sistema de numeración`» de la calculadora «`kcalc`» para convertir un número entre los distintos sistemas de numeración. En ambos casos, basta con indicar el sistema de numeración en el que se va a introducir un número —entre hexadecimal, decimal, octal o binario—, introducirlo y cambiar el sistema de numeración. Al hacerlo, se mostrará dicho número en la nueva base.

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 se está 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, el código en ensamblador sería mucho más fácil de leer y de depurar si en lugar de números se utilizaran constantes para referirse a los días de la semana. Por ejemplo, «Monday» para el lunes, «Tuesday» para el martes, etcétera. De esta forma, podríamos referirnos al lunes con el literal «`#Monday`» en lugar de con «`#1`». Naturalmente, en alguna parte del código se tendría 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`».[7] El siguiente programa muestra un ejemplo en el que se declaran y utilizan las constantes «`Monday`» y «`Tuesday`».

«`.equ Constant, Value`»

02_dias.s

```
1       .equ Monday, 1
2       .equ Tuesday, 2
3       @ ...
```

[7] 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 —por ejemplo, en otra parte del código escrita por otro programador, o por nosotros hace mucho tiempo, lo que viene a ser lo mismo—.

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

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»

02_diasreg.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
```

2.4. Inicialización de datos y reserva de espacio

Como se ha explicado en el Capítulo 1, prácticamente cualquier programa de computador necesita utilizar datos para llevar a cabo su tarea. Por regla general, estos datos se almacenan en la memoria principal del computador. Cuando se programa en un lenguaje de alto nivel se pueden utilizar variables para referirse a diversos tipos 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 dichas variables. El programador simplemente declara e inicializa las variables, pero no se preocupa de indicar cómo ni dónde deben almacenarse. Por contra, el programador en ensamblador —o un compilador de un lenguaje de alto nivel— sí debe indicar cuántas posiciones de memoria se deben utilizar para almacenar las variables de un programa, así como proporcionar sus valores iniciales, si es el caso.

«**.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. Para indicarle al ensamblador en qué parte del código fuente comienza la sección de datos, se utiliza la directiva «`.data`». En el caso del simulador QtARMSim, las reservas e inicializaciones de datos indicadas a continuación de esta directiva se realizarán en la memoria RAM a partir de la dirección `0x2007 0000`. En los siguientes subapartados se muestra cómo inicializar diversos tipos de datos y cómo reservar espacio para variables que no tienen un valor inicial definido.

Bytes, palabras, medias palabras y dobles palabras

Recordando lo que se vio en el Capítulo 1, todos los computadores y, por tanto, también los computadores basados en la arquitectura ARM, acceden a la memoria a nivel de **byte**, siendo cada byte un conjunto de 8 bits. Esto implica que hay una dirección de memoria distinta para cada byte que forma 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. Pero por otro lado, 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 los bytes, 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, etcétera.

La arquitectura ARM no solo fija el tamaño de una palabra a 4 bytes, si no que también obliga a que las palabras en memoria deban estar alineadas en direcciones de memoria que sean múltiplos de 4.

Además de trabajar con bytes y palabras, también es posible hacerlo con **medias palabras** *(half-words)* y con dobles palabras *(doubles)*. Una media palabra en ARM está formada por 2 bytes y debe estar en una dirección de memoria múltiplo de 2. Una doble palabra está formada por 8 bytes y se alinea igual que una palabra —en múltiplos de 4—.

2.4.1. Inicialización de palabras

El código fuente que se muestra después de este párrafo 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`». Dicho código fuente no acaba de ser realmente un programa ya que no contiene instrucciones en lenguaje ensamblador que vayan a realizar alguna tarea. Simplemente está formado por 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 inicializar dicho espacio con un determinado valor. La primera de las dos, «`.word 15`», inicializará[8] una palabra con el número 15 a partir de la primera dirección de memoria (por ser la primera directiva de inicialización de memoria después de la directiva «`.data`»). La siguiente, «`.word 0x15`», inicializará la siguiente posición de memoria disponible con una palabra con el número `0x15`.

«`.word Value32`»

02_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
```

▶ **2.5** Copia el código anterior en QtARMSim, ensámblalo y resuelve los siguientes ejercicios.

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

2.5.2 ¿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 en cuatro?

[8] Por regla general, cuando se hable 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.

Big-endian y *Little-endian*

Como ya se vio en el Capítulo 1, 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 dirección 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? Dependerá de la organización utilizada.

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

a) *Big-endian*

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

b) *Little-endian*

En 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)*.

2.5.3 Recuerda que las etiquetas sirven para hacer referencia a la dirección de memoria de la línea en la que están. Así pues, ¿qué valores toman las etiquetas «`word1`», «`word2`», «`word3`» y «`word4`»?

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

02_palabras2.s

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

2.4.2. Inicialización de bytes

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

«**.byte** Value8»

▶ **2.7** Teclea el siguiente programa en el editor de QtARMSim y ensámblalo.

02_byte.s

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

2.7.1 ¿Qué valores se han almacenado en memoria? ¿En qué posiciones de memoria?

2.7.2 ¿Qué valor toma la etiqueta «bytes»?

2.4.3. 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»

2.4.4. 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. La directiva «**.asciz** "cadena"» también sirve para declarar cadenas, la diferencia entre «**.ascii**» y «**.asciz**» radica en que la última añade un byte a 0 después del último carácter de la cadena. De esta forma y asegurándose de que todos los caracteres que pueden formar parte de una cadena sean siempre distintos de cero, un programa podría recorrer la cadena hasta encontrar el byte a cero añadido por «**.asciz**», lo que le serviría para saber que ha llegado al final de la cadena. Muchos lenguajes de programación, entre ellos Java, C y C++, utilizan este método, conocido como estándar ASCIIZ, para el almacenamiento de cadenas.

«**.ascii** "cadena"»

«**.asciz** "cadena"»

▶ **2.8** Copia el siguiente código en el simulador y ensámblalo.

02_cadena.s

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

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

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

2.8.3 ¿Cuántos bytes se han reservado en total para la cadena?

2.8.4 ¿A qué dirección de memoria hace referencia la etiqueta «byte»?

2.4.5. Reserva de espacio

«**.space** N»

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

▶ **2.9** Dado el siguiente código:

02_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
```

2.9.1 ¿Qué posiciones de memoria se han reservado para almacenar la variable «space»?

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

2.9.3 ¿A partir de qué dirección se ha inicializado «byte1»? ¿A partir de cuál «byte2»?

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

2.4.6. 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. Por otro lado, la directiva «**`.align N`**» le indica al ensamblador que el siguiente dato que vaya a reservarse o inicializarse, deberá comenzar en una dirección de memoria múltiplo de 2^N.

«**`.balign N`**»

«**`.align N`**»

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

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

2.5. Firmware incluido en ARMSim

El motor de simulación ARMSim incorpora un firmware propio que proporciona varias subrutinas aritméticas y de visualización de datos. Las subrutinas proporcionadas, así como los parámetros requeridas por estas se pueden consultar en el Apéndice C. Este firmware se instala automáticamente en el simulador a partir de la posición de memoria ROM `0x00190000` y puede ser utilizado directamente desde el código fuente. A modo de ejemplo, el siguiente código llama a las subrutinas del firmware «`sdivide`», «`printString`» y «`printInt`», para realizar una división entera y mostrar en el visualizador LCD el resultado del programa (véase la Figura 2.11).

02_firmware.s

```
1         .data
2 str:    .asciz "50 dividido entre 2 es igual a: "
3
4         .text
5         mov r0, #0
6         mov r1, #0
7         ldr r2, =str
```

```
8         bl  printString @ Imprime str en pantalla
9         mov r4, r0
10        mov r0, #50
11        mov r1, #2
12        bl  sdivide     @ Divide 50 entre 2
13        mov r5, r0
14        mov r0, r4
15        mov r1, #0
16        mov r2, r5
17        bl  printInt    @ Imprime el resultado
18 stop:  wfi
```

Figura 2.11: Visualizador LCD tras la ejecución de «`02_firmware.s`»

▶ **2.11** Carga el código «`02_firmware.s`» en el simulador, ejecútalo y realiza los siguientes ejercicios:

2.11.1 Comprueba que la salida del visualizador LCD coincide con la mostrada en la Figura 2.11.

2.11.2 Modifica el código para que se muestre el resultado de dividir 40 entre 3. ¿Qué partes del código has tenido que modificar para que todo lo mostrado en pantalla sea correcto? El resultado de dividir 40 entre 3 es $13.\widehat{3}$, ¿aparece la parte decimal en el resultado mostrado en el visualizador? ¿a qué crees que es debido? (hay tres motivos).

2.6. Ejercicios

Ejercicios de nivel medio

▶ **2.12** El siguiente programa carga en los registros `r0` al `r3` los caracteres 'H', 'o', 'l' y 'a', respectivamente. Copia dicho programa en QtARMSim, cambia al modo de simulación y contesta las siguientes preguntas.

02_letras.s

```
        .text
main:   mov r0, #'H'
        mov r1, #'o'
        mov r2, #'l'
        mov r3, #'a'
stop:   wfi
```

2.12.1 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?

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

▶ **2.13** Dado el siguiente código, contesta a las preguntas que aparecen a continuación:

02_dias.s

```
        .equ Monday, 1
        .equ Tuesday, 2
        @ ...

        .text
main:   mov r0, #Monday
        mov r1, #Tuesday
        @ ...
stop:   wfi
```

2.13.1 ¿Dónde se han declarado las constantes en dicho código?, ¿dónde se han utilizado?, ¿en cuál de los dos casos se ha utilizado el carácter «`#`» para referirse a un literal y en cuál no?

2.13.2 Copia el código anterior en QtARMSim, ¿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?

2.13.3 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). ¿Qué ha cambiado en el código máquina?

▶ **2.14** El siguiente código fuente utiliza constantes y personaliza el nombre de un registro. Cópialo en el simulador y ensámblalo, ¿cómo se han reescrito las instrucciones «**`mov`**» en el código máquina?

02_diasreg.s

```
        .equ Monday, 1
        .equ Tuesday, 2
        @ ...

        .text
        day .req r7
main:   mov day, #Monday
        mov day, #Tuesday
        .unreq day
        @ ...
stop:   wfi
```

Ejercicios avanzados

▶ **2.15** Sea el siguiente programa

02_byte-palabra.s

```
        .data      @ Comienzo de la zona de datos
bytes:  .byte 0x10, 0x20, 0x30, 0x40
word:   .word 0x10203040

        .text
stop:   wfi
```

2.15.1 ¿Qué valores se han almacenado en memoria?

2.15.2 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?

2.15.3 ¿Qué valores toman las etiquetas «`bytes`» y «`word`»?

▶ **2.16** Teniendo en cuenta que es posible inicializar varias palabras en una sola línea, separándolas por comas, crea un programa en ensamblador que defina un vector[9] de cinco palabras *(words)*, asociado a la etiqueta «`vector`», con 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.

[9] Un vector es un tipo de datos formado por un conjunto de datos del mismo tipo 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.

▶ **2.17** El siguiente programa utiliza las siguientes directivas: «**.ascii**» y «**.asciz**», para inicializar sendas cadenas. ¿Hay alguna diferencia en el contenido de la memoria utilizada por ambas cadenas? ¿Cuál?

02_cadena2.s

```
1         .data      @ Comienzo de la zona de datos
2  str:   .ascii "abcde"
3  byte:  .byte 0xff
4         .balign 4
5  str2:  .asciz "abcde"
6  byte2: .byte 0xff
7
8         .text
9  stop:  wfi
```

Ejercicios adicionales

▶ **2.18** 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.

▶ **2.19** 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`.

▶ **2.20** Modifica el siguiente código sustituyendo la primera directiva «**.word**» por una directiva «**.byte**» y la segunda directiva «**.word**» por una directiva «**.hword**», y modificando los valores numéricos dados para que al ensamblar el nuevo código se realice la misma inicialización de memoria que la realizada por el código original.

02_palabras3.s

```
1         .data      @ Comienzo de la zona de datos
2  a:     .word 0x10203040
3  b:     .word 0x50607080
4
5         .text
6  stop:  wfi
```

▶ **2.21** Desarrolla un programa que reserve espacio para una variable de cada uno de los tipos de datos soportados por ARM. Cada variable debe estar convenientemente alineada en memoria y etiquetada.

▶ **2.22** Desarrolla un programa que en lugar de reservar espacio, inicialice una variable de cada uno de los tipos de datos soportados

por ARM. Cada variable debe estar convenientemente alineada en memoria y etiquetada.

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

▶ **2.24** 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.)*

El programa debe mostrar cada una de dichas cadenas en el visualizador LCD.

▶ **2.25** 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).

Capítulo 3

Instrucciones de transformación de datos

Índice

En el capítulo anterior se ha visto una breve introducción al ensamblador de ARM y al simulador QtARMSim, además de cómo es posible en ensamblador declarar y utilizar literales y constantes e inicializar y reservar posiciones de memoria para distintos tipos de datos.

En este capítulo se verán las instrucciones de transformación de datos que, como se vio en el Capítulo 1, son las que realizan algún tipo de operación sobre los datos utilizando las unidades de transformación del procesador. Concretamente, se presentarán las instrucciones proporcionadas por ARM para la realización de operaciones aritméticas, lógicas

y de desplazamiento de bits. Puesto que estas instrucciones operan con números almacenados en registros, antes de describirlas, se introducirá brevemente cómo se representan los números en el computador y el banco de registros de ARM. A continuación, una vez vistas las instrucciones de transformación de datos, se describirán los modos de direccionamiento utilizados para codificar sus operandos y su formato.

3.1. Los números en el computador

Como se ha comentado previamente, los computadores utilizan como unidad de información el bit —*binary information unit*—, que puede tomar uno de dos valores: 0 o 1. Sin embargo, esta elección no es debida a que el sistema de numeración binario proporcione alguna ventaja frente al sistema decimal; los computadores trabajan con bits simplemente porque se fabrican con circuitos electrónicos que solo pueden distinguir y operar con dos niveles de tensión diferenciados. Si estos circuitos pudieran distinguir y operar con diez niveles, al mismo precio y a la misma velocidad que con dos, los computadores probablemente utilizarían el sistema decimal en lugar del binario.

Algunos de los circuitos que forman un computador están diseñados específicamente para ser capaces de almacenar un bit. Es decir, están diseñados para proporcionar y mantener durante cierto tiempo un nivel bajo o alto de tensión —un 0 o un 1—, y para que se les pueda indicar que proporcionen y mantengan un nuevo nivel a partir de un determinado momento. Puesto que cada uno de estos circuitos almacena un bit, para construir un circuito que almacene una cantidad de información mayor: un byte, una palabra, etcétera, simplemente hay que juntar varios de ellos. De esta forma, por ejemplo, se puede construir un registro de 32 bits juntando 32 circuitos de este tipo, uno para cada uno de los bits del registro —véase la Figura 3.1—. De igual forma, una posición de memoria de 8 bits se construiría con 8 circuitos capaces de almacenar un bit cada uno, eso sí, más sencillos y económicos que los utilizados en los registros. Así pues, el número de bits que se puede almacenar en un registro o en una posición de memoria está físicamente determinado por su circuitería. De hecho, todos los espacios de almacenamiento —contenedores— que proporcione un computador tendrán un número de bits prefijado.

Teniendo en cuenta los dos párrafos anteriores, es fácil ver que por un lado, cualquier tipo de información que se quiera procesar en un computador deberá codificarse forzosamente en binario y que, por otro, para almacenar los elementos de dicha información será necesario utilizar uno o más contenedores, cada uno con un tamaño fijo. Así pues, una vez se haya decidido cómo codificar un tipo de información, se deberá seleccio-

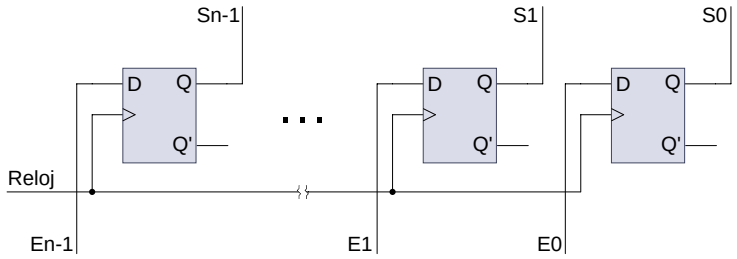


Figura 3.1: Registro de n bits. Cada uno de los biestables tipo D almacena el valor de un bit, que proporciona en su salida Q_i. Las entradas E_i junto con la señal de reloj, sirven para actualizar el contenido del registro

nar el tamaño de contenedor más adecuado para dicho tipo. Para ello, se deberá averiguar en primer lugar el máximo número de bits requeridos para codificar cualquiera de los posibles valores que pueda tomar dicho tipo de información. A continuación, simplemente se seleccionará el contenedor con un tamaño superior o igual más cercano al máximo número de bits requerido. A modo de ejemplo, se describen a continuación tres tipos de información y para cada uno de ellos: una posible codificación y el máximo número de bits requeridos. Una vez descrito lo anterior, se indica el tamaño del contenedor más adecuado para cada uno de ellos. El primer tipo de información a considerar podría ser el de si algo es cierto o falso, que se podría codificar en binario como 1 o 0, y que, por tanto, bastará con un bit. El segundo tipo de información podría referirse al estado de un semáforo —verde, ámbar o rojo—, que podría codificarse como: verde (00), ámbar (10) y rojo (11), en cuyo caso será suficiente con dos bits. El tercer tipo de información a considerar sería el formado por los números entre 0 y 31, que si se codifican en binario natural, bastará con cinco bits. Una vez descritos los ejemplos anteriores y teniendo en cuenta que la unidad mínima de almacenamiento de la mayoría de computadores es de tamaño byte, el tamaño de contenedor más adecuado para estos ejemplos será el byte, a pesar de que tan solo uno, dos y cinco bits de cada byte, respectivamente, estarán codificando información.[1]

Entre la información que un computador procesa en contenedores de tamaño fijo está, cómo no, la información numérica. Aunque la mayoría de computadores son capaces de operar con números enteros y reales, en este apartado, y, de hecho, en el resto del libro, tan solo nos ocuparemos de los números enteros. En cuanto a estos, los computadores permiten

[1] Aunque no es el objetivo de este apartado, a lo largo de este libro y ya en este mismo capítulo, se verá cómo es posible agrupar diversos tipos de información en un mismo contenedor.

diferenciar entre: *enteros sin signo*, cuando únicamente trabajan con números positivos y el 0, y *enteros*, cuando trabajan con números enteros, y por tanto, con signo.

¿Cómo se codifican los números enteros sin signo en un computador?

Los números enteros sin signo se codifican en binario natural con un determinado número de bits, el del contenedor en el que vayan a almacenarse. Este tamaño, por su parte, se escogerá en función de los valores posibles que pueda tomar el tipo de información representada. Si estos valores están entre 0 y 255, un byte será suficiente. Si están entre 0 y 4 294 967 295, será necesaria una palabra.

¿Cuándo se produce un acarreo?

El tamaño del contenedor en el que se almacene el resultado de una operación aritmética también limita los posibles valores que este puede tomar. Así, si se suman dos números sin signo contenidos en sendos registros de 32 bits y el resultado de la suma se debe almacenar también en un registro de 32 bits, el máximo resultado que podrá almacenarse correctamente será 4 294 967 295. Si el resultado de la suma es mayor, harían falta 33 bits para almacenar el resultado correcto, ya que en este caso, al realizar la suma bit a bit empezando por los bits de menor peso, al llegar a los dos bits de mayor peso, su suma más el posible acarreo de la suma anterior habrían dado un resultado, 0 o 1, más un acarreo. Este último acarreo —el «me llevo una» de cuando se suma a mano— no podrá almacenarse en el registro en el que se va a escribir el resultado y, por tanto, el número finalmente almacenado no representará correctamente el resultado de la suma.

¿Qué es el bit de acarreo?

Para poder detectar que se ha producido un acarreo —bien para avisar de que el resultado almacenado no es correcto o para permitir sumas o restas que involucren a varias palabras—, los circuitos electrónicos encargados de realizar las operaciones aritméticas suelen proporcionar, además de los bits correspondientes al resultado de la operación, un bit con el valor del acarreo, que recibe el nombre de **bit de acarreo**. El valor de este bit se puede tener en cuenta para saber si el resultado de una suma o una resta puede expresarse en el número de bits disponible. En el caso de la suma, el resultado será correcto si el bit de acarreo vale 0. En el caso de la resta, por el contrario, el resultado será correcto si el bit de acarreo vale 1. —Esto último es debido a que la resta se implementa en realidad como la suma de un número y el complemento en base 2 del otro, tal y como se describe en el Apartado D.5.

¿Cómo se codifican los números enteros en un computador?

En cuanto a los número enteros, estos se codifican en un sistema de representación en base a 2 llamado complemento a 2 (Ca2) con un determinado número de bits, el del contenedor en el que vayan a almacenarse. De nuevo, el tamaño del contenedor se escogerá en función de los valores posibles que pueda tomar el tipo de información representada. Si estos valores están entre -128 y 127, un byte será suficiente. Si están entre $-2\,147\,483\,648$ y $2\,147\,483\,647$, será necesario recurrir a una palabra.

¿Por qué se utiliza el complemento a 2 en la mayoría de los computadores?

El complemento a 2 es el sistema de representación de números en-

teros utilizado en la mayoría de los computadores. Esta elección está motivada principalmente porque los circuitos electrónicos capaces de realizar operaciones aritméticas con números sin signo, cuando operan con números codificados en complemento a 2, obtienen directamente el resultado correcto, también codificado en complemento a 2. Esto no ocurre con otros sistemas de representación de números con signo, que requerirán: I) hardware adicional para realizar operaciones con signo, II) un procesador con instrucciones máquina distintas para operar con números con y sin signo, y III) que el programador utilice unas u otras en función de si los operandos son con signo o no.

¿Cómo se codifica un número en complemento a 2?

Sin entrar en demasiados detalles, un número $n \in [-2^{k-1}, 2^{k-1} - 1]$ se representa con k bits de la siguiente forma. Si $n >= 0$, como el número positivo n en binario con k bits —y el bit de mayor peso valdrá 0—. En caso contrario, si $n < 0$, como el número positivo $2^k + n$ en binario con k bits —y el bit de mayor peso valdrá 1—. Así pues, el bit de mayor peso de un número codificado en Ca2 indicará su signo. Será 0 si es un número positivo y 1 si es negativo. Para saber más sobre los sistemas de numeración en general y el complemento a 2 en particular, se puede consultar el Apéndice D.

¿Cuándo se produce un desbordamiento?

Cuando se opera con números enteros codificados en complemento a 2 hay que tener en cuenta que el bit de mayor peso indica el signo del número —es decir, no implica que sea un número muy grande— y que además, el rango de valores que pueden codificarse se ha dividido en dos partes: una para los números positivos y otra para los negativos. Así pues, el bit de acarreo ya no servirá para indicar si el resultado ha podido codificarse correctamente en el número de bits disponibles. Bastan dos ejemplos para verlo: si se suman dos números positivos en Ca2, el bit de acarreo siempre valdrá 0 y si se suman dos números negativos en Ca2, siempre valdrá 1, independientemente de cuán grandes o pequeños sean los números sumados. Lo que sí ocurrirá cuando el resultado no pueda codificarse correctamente en Ca2 con el número de bits disponibles es que el bit de mayor peso, el de signo, tendrá un valor distinto al del signo esperado del resultado. Entonces se dice que se ha producido un desbordamiento.

¿Qué es el bit de desbordamiento?

Para poder detectar que se ha producido un desbordamiento cuando se opera con números codificados en Ca2, los circuitos electrónicos encargados de realizar las operaciones aritméticas suelen proporcionar, además de los bits correspondientes al resultado de la operación, un bit que, en función de la operación realizada —suma o resta—, de los bits de signo de los operandos y del signo esperado del resultado, indica si se ha producido o no un desbordamiento. Este bit recibe el nombre de **bit de desbordamiento**.

3.2. Banco de registros de ARM

El banco de registros de ARM está formado por 16 registros visibles por el programador (véase la Figura 3.2) y por un registro de estado, todos ellos de 32 bits. De los 16 registros visibles por el programador, los 13 primeros —del `r0` al `r12`— son de propósito general. Por contra, los registros `r13`, `r14` y `r15` son registros de propósito específico. El registro `r13` almacena el puntero de pila —o SP, por *Stack Pointer*—; el registro `r14` recibe el nombre de registro enlace —o LR, por *Link Register*— y almacena la dirección de vuelta de una subrutina; y, por último, el registro `r15`, que es el contador de programa —o PC, por *Program Counter*—, almacena, como ya se ha visto en los capítulos anteriores, la dirección de memoria de la siguiente instrucción a ejecutar. Los registros `r13` (`SP`) y `r14` (LR) se utilizan profusamente en la gestión de subrutinas, por lo que se tratarán con más detalle en los Capítulos 6 y 7, que abordan dicha temática. El registro `r15` (`PC`) es especialmente importante para las instrucciones de control de flujo, por lo que se abordará más detenidamente en el Capítulo 5, dedicado a dichas instrucciones. El último de los registros de ARM, el registro de estado, se describe un poco más adelante.

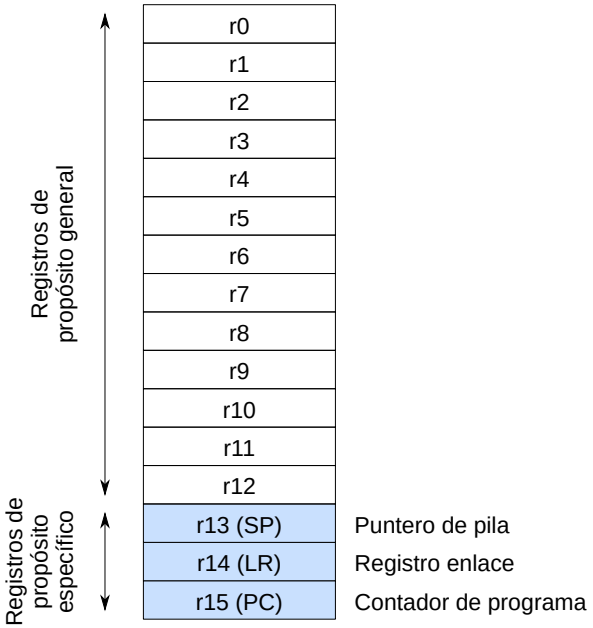


Figura 3.2: Registros visibles de ARM

En vista de lo anterior, conviene saber que la arquitectura ARM es inusual dentro de las arquitecturas RISC por el hecho de tener únicamente 16 registros de propósito general —lo habitual es disponer de 32—.

Cuando se codifica una instrucción, poder seleccionar uno de entre 16 registros requiere 4 bits de la instrucción, en lugar de los 5 bits necesarios en las arquitecturas RISC con 32 registros. Este menor número de bits, 4 frente a 5 por operando, conlleva un ahorro de hasta 3 bits por instrucción —con tres operandos en registros—. Gracias a este ahorro en el número de bits que hacen falta para codificar los operandos de una instrucción, la arquitectura ARM dispone de más bits en la instrucción que puede utilizar para distinguir entre un mayor número de instrucciones diferentes o para incorporar variaciones a una misma instrucción. En definitiva, este ahorro ha permitido que la arquitectura ARM proporcione un juego de instrucciones más rico que el presente en otras arquitecturas RISC [13].

Por su parte, la versión Thumb de ARM, donde la mayor parte de las instrucciones ocupan únicamente 16 bits, va un paso más allá haciendo que la mayoría de las instrucciones solo puedan operar con los ocho primeros registros, del `r0` al `r7`; proporcionando, por otro lado, y para aquellos casos en los que sea conveniente disponer de un mayor número de registros o se requiera acceder a uno de los registros de propósito específico, un pequeño conjunto de instrucciones que sí que pueden acceder y operar con los registros del `r8` al `r15`. Esta distinción entre registros bajos —los ocho primeros— y registros altos —los ocho siguientes—, y la limitación en el uso de los registros altos a unas pocas instrucciones, ha permitido a la arquitectura Thumb de ARM codificar un amplia variedad de instrucciones en tan solo 16 bits por instrucción.

Thumb distingue entre registros bajos *(low registers)* y registros altos *(high registers)*.

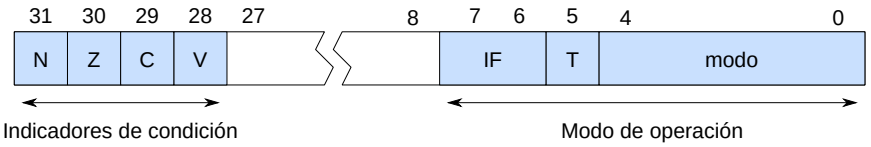


Figura 3.3: Registro de estado —*current processor status register*—

En cuanto al registro de estado (véase la Figura 3.3), los cuatro bits de mayor peso almacenan los indicadores de condición *(condition flags)*, que se describen a continuación, y los ocho bits de menor peso contienen información del sistema, tales como el estado del procesador y los mecanismos de tratamiento de las interrupciones, que se abordarán en los últimos capítulos de este libro. Los indicadores de condición, que son `N`, `Z`, `C` y `V`, sirven para indicar distintas propiedades del resultado obtenido tras la ejecución de ciertas instrucciones. A continuación se muestra cuándo se activa[2] o desactiva cada uno de los indicadores:

El registro de estado es un registro que contiene información sobre el estado actual del procesador.

[2]Como ya se sabe, un bit puede tomar uno de dos valores: 0 o 1. Se dice que un bit se activa *(set)*, cuando toma el valor 1. Por el contrario, se dice que un bit se desactiva *(clear)*, cuando toma el valor 0. Además, si se indica que un bit se activa bajo

N: Se activa cuando el resultado de la operación realizada es negativo (si el resultado de la operación es 0, este se considera positivo).

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

C: Se activa cuando el resultado de la operación produce un acarreo (llamado *carry* en inglés). —Véase el Apartado 3.1.

V: Se activa cuando el resultado de la operación produce un desbordamiento en operaciones con signo *(overflow)*. —Véase el Apartado 3.1.

Los indicadores de condición del registro de estado se muestran en QtARMSim en la esquina inferior derecha de la ventana del simulador. La Figura 3.4 muestra un ejemplo donde los indicadores `Z` y `C` se han activado y los otros dos no.

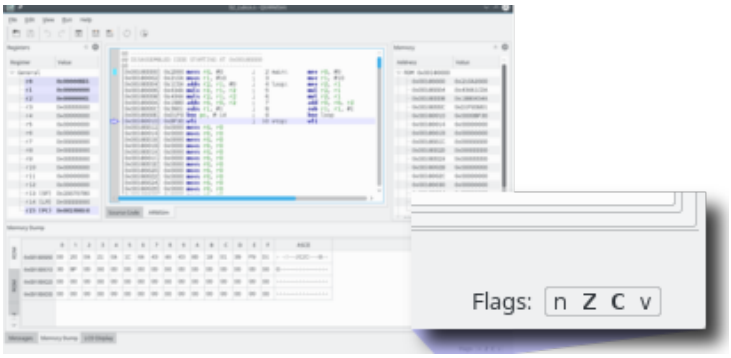


Figura 3.4: Visualización de los indicadores de condición

3.3. Operaciones aritméticas

De las instrucciones de ARM que realizan operaciones aritméticas, se van a ver en primer lugar las instrucciones de suma y resta. En concreto, la instrucción «`add rd, rs, rn`», suma el contenido de los registros `rs` y `rn`, y almacena el resultado de la suma en `rd` ($rd \leftarrow rs + rn$). Por su parte, la instrucción «`sub rd, rs, rn`», resta el contenido de `rn` del contenido de `rs` y almacena el resultado de la resta en `rd` ($rd \leftarrow rs - rn$). En el siguiente programa de ejemplo se puede ver una instrucción que suma el contenido de los registros `r0` y `r1` y almacena el resultado de la suma en el registro `r2`.

«add»

«sub»

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 se dice que un bit se desactiva bajo determinadas circunstancias.

Instrucción «`mov rd, #Inm8`»

Aunque «**`mov`** `rd, #Inm8`» es una instrucción de transferencia de datos, por lo que se verá con más detalle en el Capítulo 4, se utiliza a menudo en este capítulo, sobre todo para simplificar la realización de los ejercicios. Por el momento, basta con saber que la instrucción «**`mov`** `rd, #Inm8`», carga en el registro `rd` el dato inmediato de 8 bits, `Inm8`, especificado en la propia instrucción, es decir, `rd` $\leftarrow Inm8$.

03_add.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #2      @ r0 <- 2
3         mov r1, #3      @ r1 <- 3
4         add r2, r0, r1  @ r2 <- r0 + r1
5 stop:   wfi
```

▶ **3.1** Copia el programa anterior en el simulador y contesta a las siguientes preguntas ejecutándolo paso a paso:

3.1.1 ¿Qué hace la primera instrucción, «**`mov`** `r0, #2`»?

3.1.2 ¿Qué hace la segunda instrucción, «**`mov`** `r1, #3`»?

3.1.3 ¿Qué hace la tercera instrucción, «**`add`** `r2, r1, r0`»?

3.1.4 Modifica el código para que realice la suma de los números 10 y 6. Ejecuta el programa y comprueba que el contenido del registro `r2` es el esperado.

3.1.5 Modifica el último código para que en lugar de realizar la operación $10 + 6$, calcule $10 - 6$. Ejecuta el nuevo código y comprueba que el contenido del registro `r2` es el esperado.

Puesto que por simplificar, la mayoría de los ejemplos propuestos en este capítulo inicializan los registros con valores constantes antes de realizar las operaciones correspondientes, podría dar la impresión de que ese es el procedimiento habitual en un programa, cuando no lo es. En un caso real, los registros contienen el valor actual de las variables con las que se está operando en un momento dado, y el valor de dichas variables puede cambiar entre una ejecución y la siguiente de la misma instrucción. El siguiente ejercicio pretende ilustrar la naturaleza variable del contenido de los registros que intervienen en una determinada operación. Para ello, se propone que se ejecute varias veces un mismo

En programación, se denomina *variable* a un espacio en memoria asociado a un identificador.

programa, que consta de una única instrucción de suma, modificando a mano el contenido de los registros antes de cada ejecución.

▶ **3.2** Copia el siguiente código en el simulador, pasa al modo de simulación y completa la tabla que viene después, donde cada fila corresponde a una nueva ejecución en la que se han cargado previamente los valores indicados en los registros `r0` y `r1`. En la última columna deberás anotar el contenido de `r2` en hexadecimal (utilizando la notación `0x`) tras la ejecución correspondiente. Recuerda que para modificar el contenido de un registro hay que hacer doble-clic sobre el registro que se quiere modificar, introducir el nuevo valor y pulsar retorno. Y que para recargar una simulación, tal y como se vio en el Capítulo 2, se debe seleccionar la entrada de menú «`Run > Refresh`», o pulsar la tecla «`F4`».

03_add_sin_datos.s

```
1         .text           @ Zona de instrucciones
2 main:   add r2, r0, r1  @ r2 <- r0 + r1
3 stop:   wfi
```

r0	r1	r2
4	5	
10	6	
0x23	0x12	
0x23	0x27	

Como se ha visto, antes de realizar una operación con dos registros como operandos fuente, ha sido necesario cargar previamente en dichos registros los datos que se quería sumar o restar. Sin embargo, es muy frecuente encontrar en los programas sumas y restas en las que uno de los operandos, ahora sí, es un valor constante (p.e., cuando se quiere decrementar en uno un determinado contador: «`nvidas = nvidas - 1`»). Así que para evitar tener que cargar un valor constante en un registro antes de realizar la suma o resta correspondiente, ARM proporciona varias instrucciones que suman o restan el contenido de un registro y un valor constante especificado en la propia instrucción. Dos de estas instrucciones son: «**`add`** `rd, rs, #Inm3`», que suma el dato inmediato «`Inm3`» al contenido del registro `rs` y almacena el resultado en `rd`; y «**`sub`** `rd, rs, #Inm3`», que resta el dato inmediato «`Inm3`» del contenido del registro `rs` y almacena el resultado en `rd`. Conviene tener en cuenta que en estas instrucciones el campo destinado al dato inmediato es de solo 3 bits, por lo que solo se podrá recurrir a estas instrucciones en el caso de que el dato inmediato sea un número entre 0 y 7.

«**`add`**» *(Inm3)*

«**`sub`**» *(Inm3)*

▶ **3.3** Copia el siguiente programa en el simulador y contesta a las preguntas ejecutándolo paso a paso:

03_add_inm.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #10     @ r0 <- 10
3         sub r1, r0, #1  @ r1 <- r0 - 1
4 stop:   wfi
```

3.3.1 ¿Qué hace la primera instrucción, «**mov** r0, #10»?

3.3.2 ¿Qué hace la segunda instrucción, «**sub** r1, r0, #1»?

3.3.3 ¿Qué valor hay al final de la ejecución del programa en los registros r0 y r1?

3.3.4 Sustituye la instrucción «**sub** r1, r0, #1» del código anterior por una en la que el dato inmediato sea mayor a 7. Pasa al modo de simulación, ¿qué mensaje de error se ha mostrado en el panel de mensajes?, ¿muestra el mensaje de error en qué línea del código se ha producido el error?, ¿en cuál?

El ensamblador ARM también proporciona otras dos instrucciones de suma y resta, «**add** rd, #Inm8» y «**sub** rd, #Inm8», que permiten que uno de los operandos sea un dato inmediato, pero esta vez, de 8 bits —frente a las anteriores instrucciones en las que el dato inmediato era de tan solo 3 bits—. Utilizando estas instrucciones se puede especificar un valor en el rango $[0, 255]$ —en lugar de en el rango $[0, 7]$, como ocurría en las anteriores—. Sin embargo, la posibilidad de utilizar un dato inmediato de mayor tamaño tiene su coste: solo queda espacio en la instrucción para especificar un registro que, por tanto, deberá actuar a la vez como operando fuente y destino. Así, a diferencia de las instrucciones «**add** rd, rs, #Inm3» y «**sub** rd, rs, #Inm3», en las que se sumaban o restaban dos operandos, uno de ellos en un registro, y el resultado se guardaba sobre un segundo registro, posiblemente distinto del otro, estas instrucciones tan solo permiten incrementar o decrementar el contenido de un determinado registro en un valor determinado. A modo de ejemplo, el siguiente código decrementa en 50 el contenido del registro r0.

«**add**» *(Inm8)*
«**sub**» *(Inm8)*

03_add_inm8.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #200    @ r0 <- 200
3         sub r0, #50     @ r0 <- r0 - 50
4 stop:   wfi
```

Otra operación aritmética que se utiliza con frecuencia es la de comparación, «`cmp rs, rn`», sobre todo, y tal como se verá en el Capítulo 5, para determinar el flujo del programa en función del resultado de la última comparación realizada. Sin entrar por el momento en detalles, al comparar dos registros se activan o desactivan una serie de indicadores del registro de estado, y en el caso de las instrucciones de control de flujo, el estado de algunos de dichos indicadores se utilizará para fijar el camino a seguir. En realidad, cuando el procesador ejecuta la instrucción de comparación, lo único que hace es restar sus operandos fuente para que los indicadores de condición se actualicen en función del resultado obtenido, pero no almacena dicho resultado. Así por ejemplo, cuando el procesador ejecuta la instrucción de comparación «`cmp r0, r1`» resta el contenido del registro `r1` del contenido del registro `r0`, lo que activa los indicadores correspondientes en función del resultado obtenido, que no se almacena en ningún sitio.

▶ **3.4** Copia el siguiente programa en el simulador y realiza los ejercicios propuestos a continuación.

03_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
```

3.4.1 Ejecuta paso a paso el programa hasta ejecutar la instrucción «`cmp r0, r1`» inclusive. ¿Se ha activado el indicador `C`? *(Nota: En el caso de la resta, el indicador `C` estará activo cuando el resultado quepa en una palabra e inactivo cuando no.)*

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

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

Otra de las operaciones aritméticas que puede realizar un procesador es la de cambio de signo. La instrucción que permite cambiar el signo de un número es «`neg rd, rs`» ($rd \leftarrow -rs$). El siguiente ejercicio muestra un ejemplo en el que se usa dicha instrucción.

«neg»

▶ **3.5** Copia el siguiente programa en el simulador, ejecútalo y completa la tabla que le sigue. Ten en cuenta que los números enteros se representan en complemento a 2.

03_neg.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #64
3         neg r1, r0      @ r1 <- -r0
4         neg r2, r1      @ r2 <- -r1
5 stop:   wfi
```

r0	r1	r2

La última de las operaciones aritméticas que se va 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` —donde `rd` forzosamente tiene que ser uno de los dos registros `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»

▶ **3.6** Ejecuta el siguiente programa y resuelve las siguientes cuestiones.

03_mul.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #10     @ r0 <- 10
3         mov r1, #6      @ r1 <- 6
4         mul r1, r0, r1  @ r1 <- r0 * r1
5 stop:   wfi
```

3.6.1 ¿Qué valor se ha almacenado en `r1`? ¿Corresponde al resultado de 10×6? ¿Cómo lo has comprobado?

3.6.2 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 mensaje de error se genera?

3.6.3 Modifica el programa original sustituyendo la instrucción «**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.

3.4. Operaciones lógicas

Una vez vistas las instrucciones aritméticas soportadas por ARM, en este apartado se presentan las operaciones lógicas que dicha arquitectura puede realizar. En concreto, la arquitectura ARM proporciona las siguientes instrucciones que permiten realizar las operaciones lógicas «y» *(and)*, «o» *(or)*, «o exclusiva» (*eor* o *xor*) y «complementario» *(not)*, respectivamente:

- «**and** rd, rs», $rd \leftarrow rd\ AND\ rs$ (operación lógica «y»).
- «**tst** rn, rm», realiza $rn\ AND\ rm$ (operación lógica «y»), actualiza los indicadores correspondientes, pero no guarda su resultado.
- «**orr** rd, rs», $rd \leftarrow rd\ OR\ rs$ (operación lógica «o»).
- «**eor** rd, rs»: $rd \leftarrow rd\ EOR\ rs$ (operación lógica «o exclusiva»).
- «**mvn** rd, rs»: $rd \leftarrow NOT\ rs$ (operación lógica «complementario»).

Las operaciones lógicas «y», «o» y «o exclusiva» realizan bit a bit la operación lógica correspondiente sobre los dos operandos fuente y almacenan el resultado en el primero de dichos operandos. 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. Es decir, $\texttt{r0}_i \leftarrow \texttt{r0}_i \wedge \texttt{r1}_i, \forall i \in [0, 31]$. Así pues, y suponiendo que los registros r0 y r1 tuvieran los valores 0x0000 00D7 y 0x0000 00E0, la instrucción «**and** r0, r1» realizaría la operación que se describe a continuación, almacenando el resultado en el registro r0.

$$\begin{array}{rr} & \texttt{0000 0000 0000 0000 0000 0000 1101 0111}_2 \\ \text{y} & \texttt{0000 0000 0000 0000 0000 0000 1110 0000}_2 \\ \hline & \texttt{0000 0000 0000 0000 0000 0000 1100 0000}_2 \end{array}$$

Si se describiera la operación anterior en función del segundo operando, se podría decir que puesto que dicho operando tiene todos sus bits a 0 excepto los bits 5, 6 y 7, todos los bits del resultado serán 0

Operaciones lógicas «y», «o» y «o exclusiva»

A continuación se muestra para cada operación lógica su correspondiente tabla de verdad en función del valor de los dos operandos (izquierda) y en función del valor de uno de ellos (derecha).

Operación lógica «y»:

ab	$a \wedge b$
00	0
01	0
10	0
11	1

a	$a \wedge b$
0	0
1	b

Operación lógica «o»:

ab	$a \vee b$
00	0
01	1
10	1
11	1

a	$a \vee b$
0	b
1	1

Operación lógica «o exclusiva»:

ab	$a \oplus b$
00	0
01	1
10	1
11	0

a	$a \oplus b$
0	b
1	$\bar{b}$

salvo los bits 5, 6 y 7, que tomarán el valor que esos bits tengan en el primer operando. De hecho, lo anterior es cierto independientemente del valor que tenga el primer operando: si el registro `r1` contiene el valor `0x0000 00E0`, el resultado de ejecutar la instrucción «**`and`** `r0, r1`» será:

	$b_{31}b_{30}b_{29}b_{28}$	$\cdots$	$b_{11}b_{10}b_9b_8$	$b_7b_6b_5b_4$	$b_3b_2b_1b_0$
y	0 0 0 0	$\cdots$	0 0 0 0	1 1 1 0	0 0 0 0
	0 0 0 0	$\cdots$	0 0 0 0	$b_7b_6b_5$ 0	0 0 0 0

Una máscara de bits es una secuencia de bits que permite poner a 0, a 1 o invertir múltiples bits de un número en una única operación.

Cuando se utiliza una secuencia de bits con este fin, esta suele recibir el nombre de **máscara de bits**, ya que «oculta» (pone a cero en el ejemplo) determinados bits del otro operando, a la vez que permite «ver» los bits restantes. Teniendo en cuenta las tablas de verdad de las operaciones lógicas «y», «o» y «o exclusiva», es posible crear máscaras de bits que, usadas en conjunción con la correspondiente operación lógica, pongan determinados bits a 0, a 1, o los inviertan, respectivamente. Como se verá en los Capítulos 8, 9 y 10, el uso de máscaras de bits es muy frecuente en la gestión de la entrada/salida.

▶ **3.7** El siguiente programa implementa una operación similar a la del ejemplo anterior. Cópialo en el simulador y ejecútalo. ¿Qué valor, expresado en hexadecimal, se almacena en el registro `r0`?

¿Coincide el resultado calculado con el comportamiento esperado según la explicación anterior?

03_and.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #0xD7   @ r0 <- 0b0000 00...00 1101 0111
3         mov r1, #0xE0   @ r1 <- 0b0000 00...00 1110 0000
4         and r0, r1      @ r0 <- r0 AND r1
5 stop:   wfi
```

La última de las operaciones lógicas que se describen en este apartado, el complementario bit a bit de un número, opera sobre un único operando. La instrucción «**mvn** rd, rs» permite obtener el complemento bit a bit de un número. Es decir, $\texttt{rd}_i \leftarrow \neg\texttt{r0}_i, \forall i \in [0, 31]$. Esta operación también se denomina *complemento a 1* (abreviado como Ca1).

«**mvn**»

▶ **3.8** Copia y ejecuta el siguiente programa. ¿Qué valor se almacena en `r1` tras su ejecución? ¿Es el complemento bit a bit de `0xF0`?

03_mvn.s

```
1         .text           @ Zona de instrucciones
2 main:   mov r0, #0xF0   @ r0 <- 0b0000 00...00 1111 0000
3         mvn r1, r0      @ r1 <- NOT r0
4 stop:   wfi
```

3.5. Operaciones de desplazamiento

Además de operaciones aritméticas y lógicas, la arquitectura ARM también proporciona instrucciones que permiten desplazar los bits almacenados en un registro un determinado número de posiciones a la derecha o a la izquierda. Las instrucciones de desplazamiento son:

«**asr**», «**lsr**» y «**lsl**»

- Desplazamiento aritmético a la derecha —*arithmetic shift right*—:
  - «**asr** rd, rm, #Shift» desplaza el contenido de `rm` hacia la derecha conservando su signo, tantas veces como las indicadas por el dato inmediato «`Shift`», y almacena el resultado en rd (es decir, $rd \leftarrow rm >>_a Shift$).
  - «**asr** rd, rs» desplaza el contenido de `rd` hacia la derecha conservando su signo, tantas veces como las indicadas por el byte más bajo de `rs` (es decir, $rd \leftarrow rd >>_a rs$).

- Desplazamiento lógico a la derecha —*logical shift right*—:
  - «**lsr** rd, rm, #Shift» desplaza el contenido de rm hacia la derecha rellenando con ceros por la izquierda, tantas veces como las indicadas por el dato inmediato «Shift», y almacena el resultado en rd (es decir, $rd \leftarrow Rm >>_l Shift$).
  - «**lsr** rd, rs» desplaza el contenido de rd hacia la derecha rellenando con ceros por la izquierda, tantas veces como las indicadas por el byte más bajo de rs (es decir, $rd \leftarrow rd >>_l rs$).
- Desplazamiento lógico a la izquierda —*logical shift left*—:
  - «**lsl** rd, rm, #Shift» desplaza el contenido de rm hacia la izquierda rellenando con ceros por la derecha, tantas veces como las indicadas por el dato inmediato «Shift», y almacena el resultado en rd (es decir, $rd \leftarrow rm << Shift$).
  - «**lsl** rd, rs» desplaza el contenido de rd hacia la izquierda rellenando con ceros por la derecha, tantas veces como las indicadas por el byte más bajo de rs (es decir, $rd \leftarrow rd << rs$).

▶ **3.9** Dado el siguiente programa:

03_desp.s

```
1         .text           @ Zona de instrucciones
2  main:  mov r0, #32
3         mov r1, #1
4         lsr r0, r1      @ r0 <- r0 >>l 1
5         lsr r0, r1      @ r0 <- r0 >>l 1
6         lsr r0, r1      @ r0 <- r0 >>l 1
7         lsl r0, r1      @ r0 <- r0 << 1
8         lsl r0, r1      @ r0 <- r0 << 1
9         lsl r0, r1      @ r0 <- r0 << 1
10 stop:  wfi
```

Cópialo en el simulador y completa la siguiente tabla indicando el contenido del registro r0 —en decimal, hexadecimal y en binario— tras la ejecución de cada una de las instrucciones del programa.

		Contenido de `r0`		
	Instrucción	decimal	hexadecimal	binario
2:	«`mov r0, #32`»	32	`0x00000020`	`0...010 0000`$_2$
3:	«`mov r1, #1`»	32	`0x00000020`	`0...010 0000`$_2$
4:	«`lsr r0, r1`»			
5:	«`lsr r0, r1`»			
6:	«`lsr r0, r1`»			
7:	«`lsl r0, r1`»			
8:	«`lsl r0, r1`»			
9:	«`lsl r0, r1`»			

Al realizar el ejercicio anterior tal vez te hayas dado cuenta de que cada uno de los valores obtenidos para `r0` al desplazarlo 1 bit a la derecha es la mitad del anterior. Esto ocurre porque desplazar cierto valor un bit hacia la derecha es equivalente a dividir entre dos dicho valor. Es fácil ver que si desplazamos dos bits estamos dividiendo entre 4, si son 3, entre 8 y así, en general, se divide entre 2 elevado al número de bits del desplazamiento. Es lo mismo que ocurre, en base diez, al dividir entre la unidad seguida de 0. De manera análoga, si desplazamos hacia la izquierda, lo que hacemos es multiplicar. Si es un bit, por 2, si son 2, por 4, etcétera.

3.6. Modos de direccionamiento y formatos de instrucción de ARM

Como se ha visto en el Capítulo 1, una instrucción en ensamblador codifica qué operación se debe realizar, con qué operandos fuente y dónde se debe guardar el resultado. También se ha visto que los operandos fuente pueden estar: I) en la propia instrucción, II) en un registro, o III) en la memoria principal. Con respecto al operando destino, se ha visto que se puede almacenar: I) en un registro, o II) en la memoria principal.

Puesto que los operandos fuente de la instrucción deben codificarse en la instrucción, 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 efectiva** de los operandos, principalmente por los siguientes motivos [9]:

- 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 operando, 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, etcétera) 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, este 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 estos 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 todo lo anterior, 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**. Algunos de los principales modos de direccionamiento se vieron de forma genérica en el Apartado 1.2.5.

Por otro lado, tal y como se ha comentado al principio, una instrucción en ensamblador codifica qué operación se debe realizar, con qué operandos fuente y dónde se debe guardar el resultado. Queda por

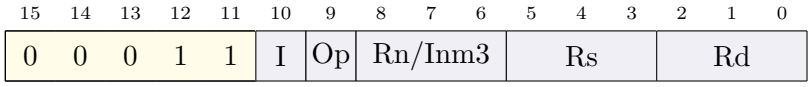
resolver cómo se codifica toda esa información en la secuencia de bits que conforman la instrucción. Una primera idea podría ser la de definir una forma de codificación única y general que pudiera ser utilizada por todas las instrucciones. Sin embargo, como ya se ha visto, el número de operandos puede 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 instrucciones reciben el nombre de **formatos de instrucción** (tal y como se detalló en el Apartado 1.2.4). Cada formato de instrucción define su tamaño y los campos que lo forman —cuánto ocupan, su orden y su significado—. Un mismo formato de instrucción puede ser utilizado para codificar uno o varios tipos de instrucciones.

En el Capítulo 1 se vieron de forma genérica los modos de direccionamiento y los formatos de instrucción —presentando ejemplos de la codificación de varias instrucciones de distintas arquitecturas—. En este capítulo y en los siguientes, se describirán dentro del correspondiente apartado «Modos de direccionamiento y formatos de instrucción de ARM», los formatos de las instrucciones vistas en el capítulo y los nuevos modos de direccionamiento utilizados. Si se desea consultar una referencia completa del juego de instrucciones Thumb de ARM y de sus formatos de instrucción, se puede recurrir al Capítulo 5 «*THUMB Instruction Set*» de [1].

3.6.1. Direccionamiento directo a registro

El direccionamiento directo a registro es el más simple de los modos de direccionamiento ya que el operando se encuentra en un registro y en la instrucción simplemente se debe codificar en cuál. Este modo de direccionamiento se utiliza en la mayor parte de las instrucciones, tanto de transferencia, como de transformación de datos, para algunos o todos sus operandos. En ARM se utiliza este modo, por ejemplo, para especificar los dos operandos fuente y el destino de las instrucciones «`add rd, rs, rn`» y «`sub rd, rs, rn`». En el caso de la variante Thumb de ARM, y que como se ha visto distingue entre ocho registros bajos y



I Inmediato: 1, inmediato; 0, registro.

Op Tipo de operación: 1, resta; 0, suma.

Rn/Inm3 Registro o dato inmediato.

Rs Registro fuente.

Rd Registro destino.

Figura 3.5: Formato de instrucción usado por las instrucciones de suma y resta con tres registros o con dos registros y un dato inmediato de 3 bits

ocho registros altos, tan solo se utilizarán 3 bits de la instrucción para codificar cada uno de los registros en los que se encuentran los distintos operandos.

3.6.2. 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, #Inm3`», «`sub rd, rs, #Inm3`», «`add rd, #Inm8`» y «`sub rd, #Inm8`», que utilizan el modo de direccionamiento inmediato en su segundo operando fuente. Las dos primeras instrucciones codifican el dato inmediato en binario natural utilizando 3 bits de la instrucción, lo que les proporciona un rango de posibles valores del 0 al 7. Las otras dos instrucciones, también codifican el dato inmediato en binario natural, pero utilizando 8 bits de la instrucción, lo que les proporciona un rango de posibles valores del 0 al 255.

3.6.3. Formato de las instrucciones aritméticas con tres operandos

El formato de instrucción utilizado para codificar las instrucciones de suma y resta con tres operandos, ya sea con tres registros o con dos registros y un dato inmediato de 3 bits, ocupa 16 bits y está formado, de izquierda a derecha, por los siguientes campos (véase la Figura 3.5):

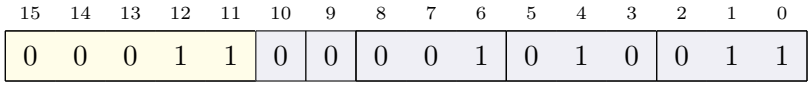


Figura 3.6: Codificación de la instrucción «**add** r3, r2, r1»

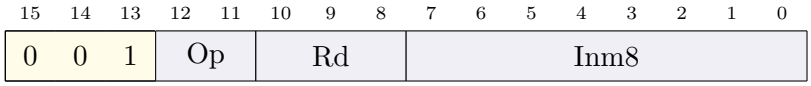
- OpCode: campo de 5 bits con el valor 00011_2, que permitirá a la unidad de control saber que la instrucción es una de las soportadas por este formato de instrucción.
- I: campo de 1 bit para indicar si el segundo operando fuente viene dado por un dato inmediato o por el contenido de un registro.
- Op: campo de 1 bit para indicar si se trata de una instrucción de resta o de suma.
- Rn/Inm3: campo de 3 bits correspondiente al segundo operando fuente.
- Rs: campo de 3 bits correspondiente al primer operando fuente.
- Rd: campo de 3 bits correspondiente al operando destino.

Siguiendo dicho formato, la instrucción «**add** r3, r2, r1» se codifica como la siguiente secuencia de bits: «00011 0 0 001 010 011», tal y como se desglosa en la Figura 3.6.

▶ **3.10** ¿Qué instrucción se codifica como «00011 1 0 001 010 011»? Compruébalo con el simulador.

3.6.4. Formato de las instrucciones con registro y dato inmediato de 8 bits

Las instrucciones que utilizan un registro como operando fuente y destino y un dato inmediato de 8 bits como segundo operando fuente, «**mov** rd, #Inm8», «**cmp** rd, #Inm8», «**add** rd, #Inm8» y, finalmente, «**sub** rd, #Inm8», se codifican utilizando el formato de instrucción mostrado en la Figura 3.7. Como se ha comentado y se puede observar en dicha figura, el campo destinado en este caso para el dato inmediato, Inm8, ocupa 8 bits. Por tanto, y puesto que el dato inmediato se codifica en binario natural, el rango de números posibles es de $[0, 255]$. A modo de ejemplo, la instrucción «**add** r4, #45», se codifica como: «001 10 100 00101101».



Op Tipo de operación: 0, mov; 1, cmp; 2, add; 3, sub.

Rd Registro fuente/destino.

Inm8 Dato inmediato.

Figura 3.7: Formato de las instrucciones «`mov rd, #Inm8`», «`cmp rd, #Inm8`», «`add rd, #Inm8`» y «`sub rd, #Inm8`»

▶ **3.11** Codifica las siguientes instrucciones a mano y comprueba con el simulador que las has codificado correctamente:

3.11.1 «`sub r2, #200`»

3.11.2 «`cmp r4, #42`»

3.7. Ejercicios

Ejercicios de nivel medio

▶ **3.12** Intenta ensamblar la instrucción: «`add r3, r4, #8`». ¿Qué mensaje muestra el ensamblador? ¿A qué es debido?

▶ **3.13** Copia el siguiente programa en el simulador, ensámblalo y completa la tabla que lo sigue. Para ello, I) recarga el simulador cada vez; II) modifica a mano el contenido del registro `r0` con el valor indicado en la primera columna de la tabla; y III) ejecuta el programa. Para anotar los resultados, sigue el ejemplo de la primera línea y una vez completada la tabla, contesta la pregunta que aparece después.

03_neg_mvn.s

```
1         .text           @ Zona de instrucciones
2 main:   neg r1, r0      @ r1 <- -r0
3         mvn r2, r0      @ r2 <- NOT r0
4 stop:   wfi
```

Valor	r0	r1	r2
10	0x0000000A	0xFFFFFFF6	0xFFFFFFF5
-10			
0			
−252 645 136			

3.13.1 Observando los resultados de la tabla anterior, ¿hay alguna relación entre el número con el signo cambiado, `r1`, y el número complementado, `r2`? ¿Cuál?

▶ **3.14** Codifica a mano las siguientes instrucciones y comprueba con el simulador que las has codificado correctamente:

3.14.1 «**add** `r3, r4, r4`».

3.14.2 «**add** `r3, r4, #5`»

Ejercicios avanzados

▶ **3.15** Se quiere guardar en el registro `r1` el resultado de sumar 100 a una variable. El siguiente código, que no puede compilarse, inicializa el registro `r0` a 250 para después sumar el valor constante 100 al contenido de `r0`. Comprueba qué problema presenta el siguiente código y corrígelo para que en `r1` se almacene la suma de un valor variable y el dato inmediato 100.

03_add_inm_error.s

```
1        .text            @ Zona de instrucciones
2 main:  mov r0, #250     @ r0 <- 250
3        add r1, r0, #100 @ r1 <- r0 + 100
4 stop:  wfi
```

▶ **3.16** El siguiente programa aplica una máscara de 32 bits sobre el contenido de `r0` para poner todos sus bits a 0 salvo los bits 8, 9, 10 y 11, que mantienen su valor original. Cópialo en el simulador y realiza los siguientes ejercicios.

03_and_32bits.s

```
1        .text                  @ Zona de instrucciones
2 main:  ldr r0, =0x12345678    @ r0 <- 0x1234 5678
3        ldr r1, =0x00000F00    @ r1 <- 0x0000 0F00
```

```
4         and r0, r1              @ r0 <- r0 AND r1
5 stop:   wfi
```

3.16.1 ¿Qué (seudo-)instrucciones se han utilizado en el código fuente en ensamblador para cargar sendos valores de 32 bits en los registros `r0` y `r1`?

3.16.2 Ejecuta el programa. ¿Qué valor, expresado en hexadecimal, se almacena en el registro `r0`? ¿Coincide con lo descrito al principio de este ejercicio?

3.16.3 Modifica el código anterior para que en lugar de aplicar una máscara que mantenga los bits 8, 9, 10 y 11 del contenido del registro `r0` y ponga a 0 el resto, mantenga los mismos bits, pero ponga a 1 los restantes. ¿Qué mascara de bits has cargado en el registro `r1`? ¿Qué operación lógica has realizado?

3.16.4 Modifica el código original para que en lugar de aplicar una máscara que mantenga los bits 8, 9, 10 y 11 del contenido del registro `r0` y ponga a 0 el resto, mantenga los mismos bits, pero invierta los bits restantes. ¿Qué mascara de bits has cargado en el registro `r1`? ¿Qué operación lógica has realizado?

Ejercicios adicionales

▶ **3.17** Copia el siguiente programa en el simulador, ejecútalo y realiza los ejercicios propuestos.

03_asr.s

```
1         .text                   @ Zona de instrucciones
2 main:   ldr r0, =0xffffff41     @ r0 <- 0xffffff41
3         mov r1, #4              @ r1 <- 4
4         asr r0, r1              @ r0 <- r0 >>a 4
5 stop:   wfi
```

3.17.1 ¿Qué valor acaba teniendo el registro `r0`? ¿Se ha conservado el signo del número cargado inicialmente en `r0`?

3.17.2 Modifica el programa para comprobar su funcionamiento cuando el número que se desplaza es positivo.

3.17.3 Modifica el programa propuesto originalmente para que realice un desplazamiento de 3 bits, en lugar de 4. 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.

3.17.4 Como se ha visto, la instrucción «**`lsr`**», desplazamiento lógico a derechas *(logical 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 `r0`?

3.17.5 Modifica el código anterior para desplazar el contenido de `r0` 2 bits a la izquierda. ¿Qué valor acaba teniendo ahora el registro `r0` tras ejecutar el programa?

3.17.6 Siempre que no se produzca un desbordamiento, desplazar n bits a la izquierda equivale a una determinada operación aritmética. ¿A qué operación aritmética equivale? ¿A qué equivale desplazar 1 bit a la izquierda? ¿Y desplazar 2 bits? ¿Y n bits?

3.17.7 Desplazar n bits a la derecha conservando el signo también equivale a una determinada operación aritmética. ¿A qué operación aritmética equivale? ¿A qué equivale desplazar 1 bit a la derecha? ¿Y desplazar 2 bits? ¿Y n bits?
(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.)

▶ **3.18** Desarrolla un programa en ensamblador que multiplique por 5 dos números almacenados en los registros `r0` y `r1`.
Para probar el programa, inicializa dichos registros con los números 18 y $-1\,215$.

▶ **3.19** Desarrolla un programa que multiplique por 32 el número almacenado en el registro `r0` sin utilizar operaciones aritméticas (es decir, no puedes utilizar la instrucción de multiplicación, ni la de suma).
Para probar el programa, inicializa el registro `r0` con la palabra `0x0000 0001`.

▶ **3.20** Desarrolla un programa que modifique el valor de la palabra almacenada en el registro `r0` 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 el registro `r0` con `0x00FF F0F0`.

▶ **3.21** Modifica alguno de los ejercicios anteriores para que muestre los resultados en el visualizador LCD del simulador.

Capítulo 4

Instrucciones de transferencia de datos

Índice

En el Capítulo 2 se vio, además de una introducción al ensamblador de ARM, al simulador QtARMSim y al uso de constantes, cómo reservar espacio en memoria para las variables que se fueran a utilizar en un programa y cómo inicializar dicho espacio. Por otra parte, el Capítulo 3 abordó las instrucciones de transformación de datos —aritméticas, lógicas y de desplazamiento—. Tal como se vio en ese capítulo, las instrucciones de transformación requerían que los datos se hubieran cargado previamente en registros o formaran parte de la propia instrucción. De hecho, en ese capítulo se utilizó frecuentemente la instrucción de transferencia: «`mov rd, #Inm8`», que carga un dato inmediato de 8 bits en el registro indicado.

Este capítulo se centra justamente en las instrucciones de transferencia de datos, como la «`mov rd, #Inm8`» vista en el capítulo anterior. Estas instrucciones permiten transferir información entre la memoria y los registros del procesador, y viceversa, distinguiéndose, según el destino de la transferencia, entre **instrucciones de carga**, las que transfieren información a los registros, e **instrucciones de almacenamiento**, las que transfieren información a la memoria.

En la arquitectura ARM, las instrucciones de transferencia de datos son las únicas que pueden acceder a la memoria. Esta decisión de diseño, común a muchas arquitecturas RISC, implica que para poder realizar operaciones con datos almacenados en memoria, primero será necesario cargarlos en registros y, una vez realizadas las operaciones, se deberá almacenar su resultado en memoria. Puede parecer contraproducente el que en lugar de disponer de instrucciones máquina que operen directamente con memoria, sea necesario que el procesador, antes de de poder realizar la operación, tenga que ejecutar instrucciones previas para cargar los operandos requeridos en registros y, además, una vez realizadas las operaciones correspondientes, deba ejecutar más instrucciones para almacenar los resultados en memoria. Sin embargo, esta decisión permite que la arquitectura disponga de un juego de instrucciones más simple, por un lado, y que las organizaciones de dicha arquitectura puedan optimizar fácilmente el cauce de ejecución *(instruction pipeline)* de las instrucciones de transformación —puesto que siempre van a operar con registros o con información que está en la propia instrucción—. Las arquitecturas que utilizan este enfoque reciben el nombre de **arquitecturas de carga/almacenamiento**.

Este capítulo comienza mostrando las instrucciones de carga proporcionadas por la arquitectura Thumb de ARM, continúa con las instrucciones de almacenamiento y termina describiendo los formatos de instrucción y los modos de direccionamiento utilizados por estas instrucciones.

4.1. Instrucciones de carga

Como se ha comentado, las instrucciones de carga son las que transfieren información a los registros. Esta información puede estar en la propia instrucción o en memoria. El próximo subapartado muestra cómo cargar datos constantes en un registro —tanto de la propia instrucción como de memoria—, los siguientes muestran cómo cargar datos variables de distintos tamaños almacenados en memoria.

4.1.1. Carga de datos constantes

La instrucción «`mov rd, #Inm8`», utilizada profusamente en el capí- «`mov rd, #Inm8`»

tulo anterior, permite cargar un dato inmediato de tamaño byte en el registro `rd` (es decir, `rd` ← $\#Inm8$).

▶ **4.1** Copia el siguiente código, ensámblalo —no lo ejecutes— y realiza los ejercicios mostrados a continuación.

04_mov.s

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

4.1.1 Modifica a mano el contenido del registro `r0` para que tenga el valor `0x12345678` (haz doble clic sobre el contenido del registro e introduce dicho número).

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

«**ldr** rd, =Inm32»

Si el dato inmediato que se quiere cargar en un registro cabe en un byte, «**mov**» es la instrucción idónea para hacerlo. Sin embargo, en el caso de tener que cargar un dato que ocupe más de un byte, no es posible utilizar dicha instrucción. Como suele ser habitual tener que cargar datos constantes más grandes en registros, el ensamblador de ARM proporciona una seudoinstrucción que sí puede hacerlo: «**ldr** `rd, =Inm32`». Dicha seudoinstrucción permite cargar datos inmediatos de hasta 32 bits.

Recordando lo comentado en el Capítulo 2, las instrucciones máquina de ARM Thumb ocupan generalmente 16 bits (y alguna, 32 bits). Sabiendo lo anterior, ¿por qué «**ldr** `rd, =Inm32`» no podría ser directamente una instrucción máquina y la tiene que proporcionar el ensamblador como seudoinstrucción? Porque puesto que el dato inmediato ya ocupa 32 bits, no quedarían bits disponibles para codificar los restantes elementos de dicha instrucción máquina. Así pues, como el operando inmediato de 32 bits ya estaría ocupando todo el espacio disponible, no sería posible codificar en la instrucción cuál es el registro destino, ni dedicar parte de la instrucción para guardar el código de operación —que además de identificar la operación que se debe realizar, debe permitir 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 puede codificarse en un byte, sustituye la seudoinstrucción por una instrucción

«**mov**» equivalente. Si por el contrario, el dato inmediato necesita más de un byte para codificarse: I) copia el valor del dato inmediato en la memoria, a continuación del código del programa, y II) sustituye la seudoinstrucción por una instrucción de carga relativa al `PC`.

El siguiente programa muestra un ejemplo en el que se utiliza la seudoinstrucción «**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.

▶ **4.2** Copia el siguiente código, ensámblalo —no lo ejecutes— y contesta a las preguntas que se muestran a continuación.

04_ldr_value.s

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

4.2.1 La seudoinstrucción «**ldr** `r1, =0xFF`», ¿a qué instrucción ha dado lugar al ser ensamblada?

4.2.2 La seudoinstrucción «**ldr** `r2, =0x10203040`», ¿a qué instrucción ha dado lugar al ser ensamblada?

4.2.3 Durante la ejecución de la instrucción anterior, tras la fase de actualización del PC, este pasará a valer `0x0018 0004`. Por otro lado, como has podido ver en la pregunta anterior, la instrucción anterior tiene un dato inmediato con valor 4. ¿Cuánto es `0x0018 0004` + 4?

4.2.4 Localiza el número `0x1020 3040` en la memoria ROM, ¿dónde está? ¿Coincide con el número que has calculado en la pregunta anterior?

4.2.5 Por último, ejecuta el programa paso a paso y anota qué valores se almacenan en el registro `r1` y en el registro `r2`.

La seudoinstrucción «**ldr** `rd, =Inm32`», tal y como se ha visto, permite cargar en un registro un valor constante escrito en el programa, pero en ocasiones, lo que se quiere cargar en un registro es la dirección de memoria de una variable utilizando directamente su etiqueta. Para cargar en un registro la dirección de memoria dada por una etiqueta, se puede utilizar la misma seudoinstrucción pero indicando la etiqueta en cuestión, en lugar de una constante numérica, «**ldr** `rd, =Label`».

«**ldr** `rd, =Label`»

▶ **4.3** Copia el siguiente programa, ensámblalo —no lo ejecutes— y contesta a las preguntas que se muestran a continuación.

04_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
```

4.3.1 ¿Qué hacen las anteriores instrucciones?

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

4.3.3 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?

4.1.2. Carga de palabras

Tras ver en el Subapartado 4.1.1 cómo se pueden cargar datos constantes, en este subapartado y siguientes se verá cómo cargar datos variables de distintos tamaños, empezando por cómo cargar palabras. Para cargar una palabra de la memoria a un registro se puede utilizar una de las siguientes instrucciones:

«`ldr rd, [...]`»

- «`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`, es decir, en sus modos de direccionamiento. 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` (es decir, `rd` ← [`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`» (es decir, `rd` ← [`rb` + `Offset5`]). El desplazamiento inmediato, «`Offset5`», puesto que los datos deben estar alineados en memoria, debe ser un número múltiplo de 4 entre 0 y 124. Conviene observar que la variante anterior, «**`ldr rd, [rb]`**», es en realidad una seudoinstrucción que será sustituida por el ensamblador por una instrucción igual pero con un desplazamiento de valor 0, es decir, por «**`ldr rd, [rb, #0]`**». Por último, 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` (es decir, `rd` ← [`rb` + `ro`]). En el siguiente ejercicio se muestra un programa de ejemplo en el que se utilizan estas tres instrucciones.

▶ **4.4** Copia el siguiente programa, ensámblalo —no lo ejecutes— y contesta a las preguntas que se muestran a continuación.

04_ldr_rb.s

```
1         .data
2  word1: .word 0x10203040
3  word2: .word 0x11213141
4  word3: .word 0x12223242
5
6         .text
7  main:  ldr r0, =word1   @ r0 <- 0x20070000
8         mov r1, #8       @ r1 <- 8
9         ldr r2, [r0]
10        ldr r3, [r0,#4]
11        ldr r4, [r0,r1]
12        wfi
```

4.4.1 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`?

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

4.4.3 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`?

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

4.4.5 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`?

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

. . . . . .

4.1.3. Carga de bytes y medias palabras

La instrucción «**`ldr`**», vista en los apartados anteriores, permite cargar una palabra en un registro. Sin embargo, hay datos que caben perfectamente en unidades de información más pequeñas que una palabra: ya sea en un byte o en una media palabra. Para evitar que dichos datos malgasten espacio de memoria, suelen almacenarse utilizando el espacio que realmente requieren. Así que para poder operar con dichos datos, la arquitectura ARM proporciona instrucciones capaces de cargar un byte o una media palabra de memoria en un registro.

Cuando se carga una palabra de memoria a un registro, simplemente se copia el contenido de los 32 bits de memoria a los 32 bits del registro. De esta forma, el registro pasa a tener el mismo contenido que la palabra de memoria, independientemente del significado o del uso que se vaya a dar a esos 32 bits. Sin embargo, cuando se carga un byte de memoria a un registro, los 8 bits de memoria se copian en los 8 bits de menor peso del registro, pero, ¿qué se hace con los 24 bits de mayor peso del registro? Una posible solución sería simplemente la de poner los 24 bits de mayor peso a 0. Si se opta por esta solución, todo irá bien mientras el dato cargado no sea un número negativo, ya que en dicho caso, el contenido del registro sería un número positivo, cuando no debería serlo. Así, si el dato almacenado en memoria no tiene signo —porque no es un número o porque es un número natural—, es suficiente con poner a 0 los 24 bits de mayor peso del registro; pero si el dato almacenado tiene signo —es decir, puede ser positivo o negativo—, los 24 bits de mayor peso deberán rellenarse con 0 si el número es positivo, o con 1 si el número es negativo —esta operación recibe el nombre de **extensión de signo**—. Por lo tanto, cuando se vaya a cargar un byte, el ensamblador deberá permitir especificar si se quiere extender o no su signo —será el programador, dependiendo de si la variable en cuestión corresponde a un número con signo o no, quien active la extensión de signo o no—. El razonamiento

La extensión de signo es la operación que incrementa la cantidad de bits de un número preservando el signo y el valor del número original.

anterior también se aplica cuando en lugar de un byte se quiere cargar una media palabra.

En el cuadro siguiente se muestran las instrucciones proporcionadas por ARM para cargar bytes sin y con extensión de signo. Hay que tener en cuenta que el desplazamiento «`Offset5`», utilizado en la segunda variante debe ser un número comprendido entre 0 y 31. Además, las dos primeras variantes no tienen una instrucción equivalente que cargue y, a la vez, extienda el signo. Para extender el signo en estos dos casos es necesario, después de ejecutar la instrucción «**`ldrb`**» correspondiente, ejecutar la instrucción «**`sxtb`** `rd, rm`», que extiende el signo del byte de menor peso del registro `rm`, almacenando el resultado en el registro `rd`.

«**`sxtb`** `rd, rm`»

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]`»

El siguiente programa muestra varios ejemplos de carga de bytes.

04_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 el cuadro siguiente se muestran las instrucciones proporcionadas por ARM para cargar medias palabras sin y con extensión de signo. Hay que tener en cuenta que el desplazamiento «`Offset5`», utilizado en la segunda variante debe ser un número múltiplo de 2 comprendido entre 0 y 62. Además, y tal y como ocurría con las instrucciones de carga de bytes, las dos primeras variantes no tienen una instrucción equivalente

que cargue y, a la vez, extienda el signo de la media palabra. Para extender el signo en estos dos casos es necesario, después de ejecutar la instrucción «`ldrh`» correspondiente, ejecutar la instrucción «`sxth rd, rm`», que extiende el signo de la media palabra de menor peso del registro `rm`, almacenando el resultado en el registro `rd`.

«`sxth rd, rm`»

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]`»

El siguiente programa muestra varios ejemplos de carga de medias palabras.

04_ldrh.s

```
1         .data
2 half1:  .hword -1500
3 half2:  .hword 2000
4 half3:  .hword 4000
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
```

4.2. Instrucciones de almacenamiento

Como se ha comentado en la introducción de este capítulo, las instrucciones de almacenamiento son las que transfieren información a la memoria. En este caso, la información que se transfiere parte siempre de un registro. Los siguientes subapartados muestran cómo almacenar datos variables de distintos tamaños en memoria.

4.2.1. Almacenamiento de palabras

Para almacenar una palabra en memoria desde un registro se puede

«`str rd, [...]`»

utilizar una de 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` (es decir, [`rb`] ← `rd`). 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`» (es decir, [`rb` + `Offset5`] ← `rd`). El desplazamiento inmediato, «`Offset5`», debe ser un número múltiplo de 4 entre 0 y 124. Conviene observar que la variante anterior, «`str rd, [rb]`» es en realidad una seudoinstrucción que será sustituida por el ensamblador por una instrucción de este tipo con un desplazamiento de 0, es decir, por «`str rd, [rb, #0]`». Por último, 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` (es decir, [`rb` + `ro`] ← `rd`). En el siguiente ejercicio se muestra un programa de ejemplo en el que se utilizan estas tres instrucciones.

▶ **4.5** Copia el siguiente programa, ensámblalo —no lo ejecutes— y contesta a las preguntas que se muestran a continuación.

04_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
```

4.5.1 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?

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

4.5.3 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?

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

4.5.5 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?

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

4.2.2. Almacenamiento de bytes y medias palabras

Cuando se almacenan bytes o medias palabras, no ocurre como en la carga de bytes o medias palabras, donde era necesario extender el signo en el caso de que los datos tuvieran signo. En este caso simplemente se va a copiar el contenido del byte o de la media palabra de menor peso de un registro a una posición de memoria del mismo tamaño.

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`» (teniendo en cuenta que, como ya se comentó en el caso de la instrucción «`ldrb`», el desplazamiento «`Offset5`» debe ser un número comprendido entre 0 y 31):

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

- «**strb** rd, [rb, ro]».

El siguiente programa muestra el uso de dichas instrucciones:

04_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
```

Por su parte, para almacenar medias palabras se pueden utilizar las siguientes variantes de la instrucción «**strh**» (teniendo en cuenta que, 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):

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

El siguiente programa muestra el uso de dichas instrucciones:

04_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
```

4.3. Modos de direccionamiento y formatos de instrucción de ARM

En este capítulo se han visto varias instrucciones de ARM que utilizan los siguientes modos de direccionamiento: I) el direccionamiento indirecto con desplazamiento, utilizado, por ejemplo, por el operando fuente de «`ldr rd, [rb, #Offset5]`», II) el direccionamiento relativo al contador de programa, utilizado, por ejemplo, por el operando fuente de la instrucción «`ldr rd, [PC, #Offset8]`»,[1] y III) el direccionamiento indirecto con registro de desplazamiento, utilizado, por ejemplo, por el operando fuente de «`ldr rd, [rb, ro]`». Estos modos, que ya se vieron de forma genérica en el Apartado 1.2.5, se describen con más detalle y particularizados a la arquitectura ARM en los Apartados 4.3.1, 4.3.2 y 4.3.3, respectivamente.

4.3.1. Direccionamiento indirecto con desplazamiento

En el modo de direccionamiento indirecto 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 (véase la Figura 4.1). Este modo de direccionamiento se utiliza en las instrucciones de carga y almacenamiento para el operando fuente y destino, respectivamente. La instrucción de carga, «`ldr rd, [rb, #Offset5]`»,[2] 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 del registro `rb` y un desplazamiento inmediato, `Offset5`. El modo de direccionamiento utilizado para este segundo operando es el indirecto con desplazamiento. Por otro lado, la instrucción «`str rd, [rb, #Offset5]`» funciona de forma parecida.

[1] La instrucción «`ldr rd, [PC, #Offset8]`» no se ha visto directamente en este capítulo, se genera al ensamblar las seudoinstrucciones «`ldr rd, =Inm32`» y «`ldr rd, =Label`»

[2] La codificación del desplazamiento, como se verá más adelante, y debido a que los accesos a medias palabras y palabras están alineados a múltiplos de 2 y de 4, respectivamente, no incluirá el bit o los 2 bits de menor peso, ya que estos valdrán siempre 0.

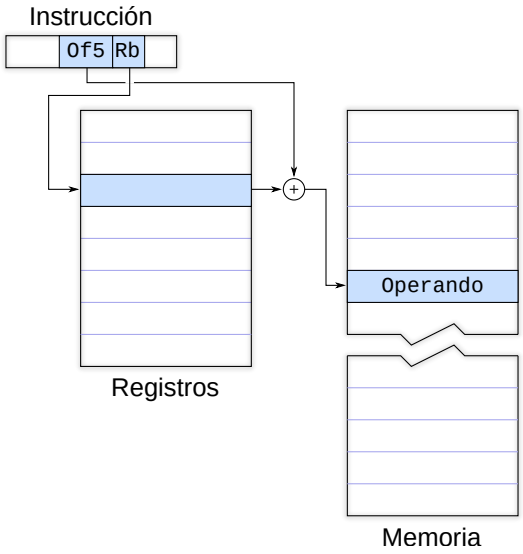


Figura 4.1: Modo de direccionamiento indirecto con desplazamiento. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica el número del registro `rb` y un dato inmediato, `Offset5`, la suma del contenido de dicho registro y el dato inmediato proporciona la dirección de memoria en la que está el operando

▶ **4.6** Contesta las preguntas que aparecen a continuación con respecto a la instrucción «`str rd, [rb, #Offset5]`».

4.6.1 ¿Cuántos operandos tiene dicha instrucción?

4.6.2 ¿Cuál es el operando fuente? ¿Cuál es el operando destino?

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

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

4.3.2. Direccionamiento relativo al contador de programa

El direccionamiento relativo al contador de programa es una variante del direccionamiento indirecto con desplazamiento, visto en el apartado anterior, en el que el registro base es el contador de programa (PC).

Este modo especifica la dirección efectiva del operando como la suma del contenido del contador de programa y un desplazamiento codificado en la propia instrucción. Es especialmente útil para acceder a operandos que se encuentran en las inmediaciones de la instrucción que se está ejecutando. Una ventaja de este modo, que se verá también más adelante cuando se comente el formato de instrucción en el que se utiliza, es que tan solo se debe proporcionar un campo para especificar el desplazamiento, puesto que el registro base siempre es el `PC`, lo que permite dedicar más bits de la instrucción para codificar el desplazamiento. Este modo de direccionamiento lo utiliza la instrucción «`ldr rd, [PC, #Offset8]`» que carga en el registro `rd` el contenido de la dirección de memoria dada por la suma del registro `PC` y el desplazamiento `#Offset8`. Esta instrucción, que no se ha visto tal cual en este capítulo, la genera el ensamblador al procesar las seudoinstrucciones «`ldr rd, =Inm32`» y «`ldr rd, =Label`».

4.3.3. Direccionamiento indirecto con registro de desplazamiento

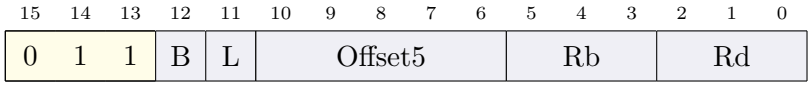
En el modo de direccionamiento relativo a registro con registro de desplazamiento, la dirección efectiva del operando es una dirección de memoria que se obtiene sumando el contenido de dos registros. Por tanto, si un operando utiliza este modo de direccionamiento, el formato de instrucción deberá proporcionar 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.

4.3.4. Formato de las instrucciones de carga/almacenamiento de bytes y palabras con direccionamiento indirecto con desplazamiento

Este formato de instrucción lo utilizan las siguientes instrucciones de carga y almacenamiento: «`ldr`», «`str`», «`ldrb`» y «`strb`». Como se puede observar en la Figura 4.2, está formado por los siguientes campos:

- OpCode: campo de los 3 bits de mayor peso con el valor 011_2, que permitirá a la unidad de control saber que la instrucción es una de las soportadas por este formato de instrucción.
- B: se utiliza para indicar si se debe transferir un byte ($B = 1$) o una palabra ($B = 0$).
- L: se utiliza para indicar si se trata de una instrucción de carga ($L = 1$) o de almacenamiento ($L = 0$).

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* (OpCode).



B Byte/Word: 1, transferir byte; 0, palabra.

L Load/Store: 1, cargar; 0, almacenar.

Offset5 Dato inmediato.

Rb Registro base.

Rd Registro fuente/destino.

Figura 4.2: Formato de las instrucciones de carga/almacenamiento de bytes y palabras con direccionamiento indirecto con desplazamiento: «`ldr rd, [rb, #0ffset5]`», «`str rd, [rb, #0ffset5]`», «`ldrb rd, [rb, #0ffset5]`» y «`strb rd, [rb, #0ffset5]`»

- Offset5: se utiliza para codificar el desplazamiento que junto con el contenido del registro `rb` proporciona la dirección de memoria de uno de los operandos.
- Rb: se utiliza para codificar el registro base, cuyo contenido se usa junto con el valor de `Offset5` para calcular la dirección de memoria del operando indicado en el campo anterior.
- Rd: se utiliza para codificar el registro destino o fuente en el que se encuentra el otro operando (dependiendo de si la instrucción es de carga o de almacenamiento, respectivamente).

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».

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. Como 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 ($0 = 00000_2$, $4 = 000100_2$, $8 = 001000_2$, $12 = 001100_2 \ldots$). Por último, el desplazamiento posible, si se cuenta en número de palabras, sería únicamente de $[0, 7]$, lo que limitaría bastante la utilidad de este modo de

Conviene tener en cuenta que la optimización comentada en este párrafo es posible porque la arquitectura ARM fuerza a que los datos estén alineados en memoria.

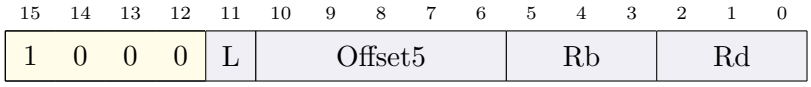
direccionamiento. Teniendo en cuenta todo lo anterior, ¿cómo se podrían aprovechar mejor los 5 bits del campo `Offset5` en el caso de la carga y almacenamiento de palabras? Simplemente no malgastando los 2 bits de menor peso del campo `Offset5` para almacenar los 2 bits de menor peso del desplazamiento —que se sabe 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, se estarían codificando 7 bits de desplazamiento utilizando únicamente 5 bits —ya que los 2 bits de menor peso se sabe que son 0—. Como 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—. ¿Cómo se codificaría un desplazamiento de, por ejemplo, 20 bytes en una instrucción de carga de bytes?, ¿y en una 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 directamente el número 20 con 5 bits: 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` únicamente los bits del 2 al 6. Puesto que $20 = 0010100_2$, en el campo `Offset5` se almacenaría el valor 00101_2 —o lo que es lo mismo, $20/4 = 5$—.

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

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

4.3.5. Formato de las instrucciones de carga/almacenamiento de medias palabras con direccionamiento indirecto con desplazamiento

Las instrucciones de carga y almacenamiento de medias palabras se codifican con un formato de instrucción (véase la Figura 4.3) ligeramente distinto al de las instrucciones de carga y almacenamiento de bytes y palabras (visto en la Figura 4.2). Aunque como se puede observar, ambos formatos de instrucción tan solo se diferencian en los 4 bits de mayor peso. 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



L Load/Store: 1, cargar; 0, almacenar.

Offset5 Dato inmediato.

Rb Registro base.

Rd Registro fuente/destino.

Figura 4.3: Formato de las instrucciones de carga/almacenamiento de medias palabras con direccionamiento indirecto con desplazamiento: «`ldrh rd, [rb, #Offset5]`» y «`strh rd, [rb, #Offset5]`»

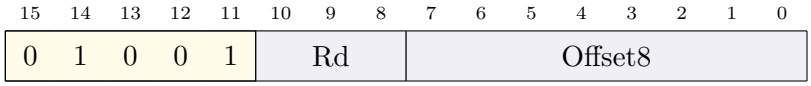
cuanto al cuarto bit de mayor peso, el bit 12, en el caso del formato de instrucción para bytes y palabras, este podía tomar como valor un 0 o un 1, mientras que en el formato de instrucción para medias palabras siempre vale 0. Así pues, el código de operación de este formato está formado por 4 bits con el valor $\texttt{1000}_2$. Los campos restantes, L, Offset5, Rb y Rd son los mismos que los del formato de instrucción anterior. Además, y siguiendo un razonamiento similar al del caso de carga o almacenamiento de palabras, el campo Offset5 permite codificar un dato inmediato de 6 bits almacenando en él únicamente los 5 bits de mayor peso, ya que el bit de menor peso no es necesario almacenarlo puesto que siempre vale 0 —para poder leer o escribir una media palabra, esta debe estar almacenada en una dirección múltiplo de 2, y un múltiplo de 2 en binario siempre tiene el bit de menor peso a 0—. 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—.

4.3.6. Formato de la instrucción de carga con direccionamiento relativo al `PC`

La instrucción «`ldr rd, [PC, #Offset8]`» carga en el registro `rd` la palabra leída de la dirección de memoria especificada por la suma del $\texttt{PC}_{inst}$+4 alineado a 4, donde $\texttt{PC}_{inst}$ es la dirección de la instrucción, más el dato inmediato codificado en `Offset8`. Es decir:

$$rd \leftarrow [(PC_{inst} + 4) \text{ AND } 0\text{xFFFFFFC} + Offset8]$$

¿Por qué $\texttt{PC}_{inst}$+4 en lugar de $\texttt{PC}_{inst}$ o $\texttt{PC}_{inst}$+2? La respuesta a esta pregunta se verá más adelante, concretamente en el Apartado 5.4.2, por el momento, nos lo creemos. ¿Por qué se alinea a 4 el resultado de $\texttt{PC}_{inst}$+4? Puesto que en ARM la memoria está alineada, la dirección de la palabra que se va a cargar será múltiplo de 4. Si $\texttt{PC}_{inst}$+4 se alinea a



Rd Registro destino.

Offset8 Dato inmediato.

Figura 4.4: Formato de la instrucción de carga con direccionamiento relativo al contador de programa: «`ldr rd, [PC, #Offset8]`»

4, entonces el dato inmediato codificado en `Offset8` también deberá ser múltiplo de 4 [1]. Y esto tiene sus ventajas, como se verá en breve.

El formato de esta instrucción, «`ldr rd, [PC, #Offset8]`», está formado por los campos que aparecen a continuación y que se representan en la Figura 4.4:

- OpCode: campo de 5 bits con el valor $\texttt{01001}_2$, que permitirá a la unidad de control saber que se trata de esta instrucción.
- Rd: se utiliza para codificar el registro destino.
- Offset8: se utiliza para codificar el desplazamiento que junto con el contenido del registro `PC` proporciona la dirección de memoria del operando fuente.

Es interesante observar que el registro que se utiliza como registro base, el `PC`, está implícito en la instrucción. Es decir, no se requiere un campo adicional para codificar dicho registro, basta con saber que se trata de esta instrucción para que el procesador utilice dicho registro. Por otro lado, puesto que la instrucción «`ldr rd, [PC, #Offset8]`» carga una palabra cuya dirección de memoria tiene que estar alineada a 4 y el campo `Offset` codifica un número múltiplo de 4, por lo que se acaba de comentar, en el campo Offset8, de 8 bits, se codifica en realidad un dato inmediato de 10 bits —se guardan solo los 8 bits de mayor peso, ya que los dos de menor peso no es necesario guardarlos ya que siempre serán 0—. De esta forma, el rango del desplazamiento es en realidad de $[0, 1020]$ bytes[3] —o lo que es lo mismo, de $[0, 255]$ palabras—.

▶ **4.9** Con ayuda del simulador, obtén el valor de los campos `rd` y `Offset8` de la codificación de la instrucción «`ldr r3, [pc, #844]`».

[3] Si los bits 1 y 0 no se consideraran como bits implícitos del campo `Offset8`, el rango del desplazamiento correspondería al rango con 8 bits para números múltiplos de 4, es decir, $[0, 252]$ bytes, o $[0, 63]$ palabras, con respecto al `PC`.

4.3.7. Formatos de las instrucciones de carga/almacenamiento con direccionamiento indirecto con registro de desplazamiento

Para codificar las instrucciones de carga/almacenamiento que utilizan el modo de direccionamiento indirecto con registro de desplazamiento se utilizan dos formatos de instrucción. El primero de estos formatos de instrucción se muestra en la Figura 4.5 y se utiliza para codificar las instrucciones:

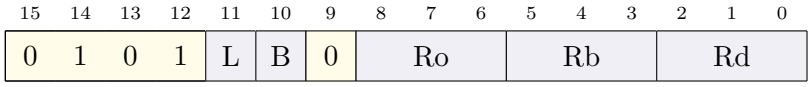
- «`ldr rd, [rb, ro]`»,
- «`ldrb rd, [rb, ro]`»,
- «`str rd, [rb, ro]`» y
- «`strb rd, [rb, ro]`».

El segundo de los formatos de instrucción se muestra en la Figura 4.6 y codifica las restantes instrucciones de este tipo:

- «`strh rd, [rb, ro]`»,
- «`ldrh rd, [rb, ro]`»,
- «`ldrsh rd, [rb, ro]`» y
- «`ldrsb rd, [rb, ro]`».

Ambos formatos de instrucción comparten los primeros 4 bits del código de operación, $\texttt{0101}_2$, y los campos Rd, Rb y Ro, utilizados para codificar el registro destino/fuente, el registro base y el registro desplazamiento, respectivamente. Se diferencian en el quinto bit del código de operación —situado en el bit 9 de la instrucción—, que está a 0 en el primer caso y a 1 en el segundo, y en que los bits 11 y 10, que ambos utilizan para identificar la operación en concreto, reciben nombres diferentes.

▶ **4.10** Cuando el procesador lee una instrucción de uno de los formatos descritos en las Figuras 4.5 y 4.6, ¿cómo distingue de cuál de los dos formatos de instrucción se trata?



L Load/Store: 1, cargar; 0, almacenar.

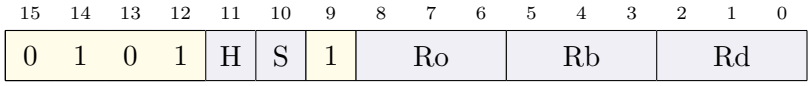
B Byte/Word: 1, transferir byte; 0, palabra.

Ro Registro desplazamiento.

Rb Registro base.

Rd Registro fuente/destino.

Figura 4.5: Formato A de las instrucciones de carga/almacenamiento con direccionamiento indirecto con registro de desplazamiento



H y S Identifican la instrucción:

H	S	Instrucción
0	0	«`strh rd, [rb, ro]`»
0	1	«`ldrsb rd, [rb, ro]`»
1	0	«`ldrh rd, [rb, ro]`»
1	1	«`ldrsh rd, [rb, ro]`»

Ro Registro desplazamiento.

Rb Registro base.

Rd Registro fuente/destino.

Figura 4.6: Formato B de las instrucciones de carga/almacenamiento con direccionamiento indirecto con registro de desplazamiento

4.4. Ejercicios

Ejercicios de nivel medio

▶ **4.11** Dado el siguiente programa (ya visto en el ejercicio 4.3), ensámblalo —no lo ejecutes— y resuelve los ejercicios que se muestran a continuación.

04_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
```

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

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

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

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

▶ **4.12** 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?

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

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

Ejercicios avanzados

▶ **4.15** 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 $[10, 20]$.

▶ **4.16** Con ayuda del simulador, obtén el valor de los campos `rd` y `Offset8` de la codificación de «`ldr r4, [pc, #492]`». ¿Cuántos bits se necesitan para codificar en binario el número 492? ¿Se han guardado todos los bits del número 492 en el campo `Offset8`? ¿Cuáles no y por qué no hace falta hacerlo?

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

▶ **4.18** Con ayuda del simulador comprueba cuál es la diferencia entre la codificación de la instrucción «`ldrsh r5, [r1, r3]`», realizada en el ejercicio anterior, y la de «`ldrsh r3, [r1, r2]`».

Ejercicios adicionales

▶ **4.19** 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`.

▶ **4.20** Amplía el anterior programa para que sume 5 a cada elemento del vector V y almacene el nuevo vector justo después del vector V. Dicho de otra forma, el programa deberá realizar la siguiente operación: $W_i = V_i + 5, \forall i \in [0, 4]$.

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

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

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

Capítulo 5

Instrucciones de control de flujo

Índice

Los programas mostrados en los anteriores capítulos constan de una serie de instrucciones que se ejecutan una tras otra, siguiendo el orden en el que estas se habían puesto, 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**. Este tipo de ejecución 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.

El registro de estado (recordatorio)

Tal y como se vio en el capítulo anterior (§ 3.2), el registro de estado mantiene información sobre el estado actual del procesador. Parte de dicha información consiste en 4 bits que indican:

N: si el resultado ha sido negativo.

Z: si el resultado ha sido 0.

C: si se ha producido un acarreo.

V: si se ha producido un desbordamiento en operaciones con signo.

Cuando el procesador ejecuta una instrucción de transformación, como «**add**» o «**cmp**», actualiza los indicadores en función del resultado obtenido. De esta forma, cuando posteriormente ejecute una instrucción de control de flujo, podrá realizar una acción u otra dependiendo del valor de dichos indicadores.

Pero incluso en ese caso, el programa sería incapaz de variar el número de veces que dichas instrucciones se ejecutarían.

Debido a la gran ventaja que supone el que un programa pueda tomar diferentes acciones y que pueda repetir un conjunto de instrucciones un número de veces variable, los lenguajes de programación proporcionan dichas funcionalidades por medio de 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.

Las estructuras de control permiten que un programa pueda tomar diferentes acciones y que pueda repetir un conjunto de instrucciones un determinado número de veces.

Este capítulo se centra en las instrucciones y recursos proporcionados por la arquitectura ARM para la implementación de las estructuras de control de flujo. El Apartado 5.1 muestra qué son y para qué se utilizan los saltos incondicionales y condicionales. El Apartado 5.2 describe las estructuras de control condicionales *if-then* e *if-then-else*. El Apartado 5.3 presenta las estructuras de control repetitivas *while* y *for*. Una vez vistas las instrucciones que hacen posibles las estructuras de control, se ven con detalle los modos de direccionamiento utilizados para codificar sus operandos y el formato de estas instrucciones. Por último, se proponen una serie de ejercicios.

5.1. Saltos incondicionales y condicionales

Las estructuras de control, tanto las condicionales como las repetitivas, se implementan por medio de saltos incondicionales y condicionales. Así pues, antes de ver con detalle las estructuras de control, es necesario conocer los tipos de salto soportados por ARM y las instrucciones que los implementan. En primer lugar se verán los saltos incondicionales, y a continuación, los condicionales.

5.1.1. Saltos incondicionales

Los saltos incondicionales son aquellos que se realizan siempre, es decir, que no dependen de que se cumpla una determinada condición para realizar o no el salto. La instrucción de ARM que realiza un salto incondicional es «`b etiqueta`», donde «`etiqueta`» indica la dirección de memoria de la instrucción 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 valores tengan los indicadores del registro de estado.

«`b etiqueta`»

▶ **5.1** El siguiente programa muestra un ejemplo de salto incondicional.

05_branch.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
```

5.1.1 Carga y ejecuta el programa anterior. ¿Qué valor contiene el registro `r3` al finalizar el programa?

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

5.1.3 Volviendo al código original, ¿qué crees que pasaría si la etiqueta «`salto`» estuviera antes de la instrucción «`b salto`», por ejemplo, en la línea «`mov r1, #10`»?

5.1.4 Crea un nuevo código basado en el código anterior, pero en el que la línea etiquetada con «`salto`» sea la línea en la que

está la instrucción «`mov r1,#10`». Ejecuta el programa paso a paso y comprueba si lo que ocurre coincide con lo que habías deducido en el ejercicio anterior.

5.1.2. Saltos condicionales

Las instrucciones de salto condicional presentan la siguiente forma: «`bXX etiqueta`», donde «`XX`» se sustituye por un nemotécnico que indica la condición que se debe cumplir para realizar el salto y «`etiqueta`» indica la dirección de memoria a la que se quiere saltar en el caso de que se cumpla dicha condición. Cuando el procesador ejecuta una instrucción de salto condicional, comprueba los indicadores del registro de estado para decidir si realizar el salto o no. Por ejemplo, cuando se ejecuta la instrucción «`beq etiqueta`» *(branch if equal)*, el procesador 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, cuando el procesador ejecuta «`bne etiqueta`» *(branch if not equal)*, saltará a la instrucción etiquetada con «`etiqueta`» si el indicador `Z` no está activo. Si está activo, no saltará. En el Cuadro 5.1 se recogen las instrucciones de salto condicional disponibles en ARM.

«`beq etiqueta`»

«`bne etiqueta`»

▶ **5.2** El siguiente ejemplo muestra un programa en el que se utiliza la instrucción «`beq`» para saltar en función de si los valores contenidos en los registros comparados en la instrucción previa eran iguales o no.

05_beq.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
```

5.2.1 Carga y ejecuta el programa anterior. ¿Qué valor contiene el registro `r3` cuando finaliza el programa?

Cuadro 5.1: Instrucciones de salto condicional. Se muestra el nombre de la instrucción, el código asociado a dicha variante —que se utilizará al codificar la instrucción— y la condición de salto —detrás de la cual y entre paréntesis se muestran los indicadores relacionados y el estado en el que deben estar, activo en mayúsculas e inactivo en minúsculas, para que se cumpla la condición—

Instrucción	Código	Condición de salto
«**beq**» *(branch if equal)*	0000	Igual (`Z`)
«**bne**» *(branch if not equal)*	0001	Distinto (`z`)
«**bcs**» *(branch if carry set)*	0010	Mayor o igual sin signo (`C`)
«**bcc**» *(branch if carry clear)*	0011	Menor sin signo (`c`)
«**bmi**» *(branch if minus)*	0100	Negativo (`N`)
«**bpl**» *(branch if plus)*	0101	Positivo o cero (`n`)
«**bvs**» *(branch if overflow set)*	0110	Desbordamiento (`V`)
«**bvc**» *(branch if overflow clear)*	0111	No hay desbordamiento (`v`)
«**bhi**» *(branch if higher)*	1000	Mayor sin signo (`Cz`)
«**bls**» *(branch if lower or same)*	1001	Menor o igual sin signo (`c` o `Z`)
«**bge**» *(branch if greater or equal)*	1010	Mayor o igual (`NV` o `nv`)
«**blt**» *(branch if less than)*	1011	Menor que (`Nv` o `nV`)
«**bgt**» *(branch if greater than)*	1100	Mayor que (`z` y (`NV` o `nv`))
«**ble**» *(branch if less than or equal)*	1101	Menor o igual (`Nv` o `nV` o `Z`)

5.2.2 ¿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.

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

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

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

▶ **5.3** Se verá ahora el funcionamiento de la instrucción «**bne** `etiqueta`». Para ello, el punto de partida será el programa anterior (sin la modificación propuesta en el ejercicio previo), y en el que se deberá sustituir la instrucción «**beq** `salto`» por «**bne** `salto`».

5.3.1 Carga y ejecuta el programa. ¿Qué valor contiene el registro `r3` cuando finaliza el programa?

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

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

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

5.3.5 ¿Por qué no se produce el salto en el primer caso, cuando se había utilizado «`cmp r0, r2`», y sí en el segundo, con «`cmp r0, r1`»?

5.2. Estructuras de control condicionales

Los saltos incondicionales y condicionales vistos en el apartado anterior se utilizan para construir las estructuras de control condicionales y repetitivas. En este apartado se verá cómo utilizar dichos saltos para la construcción de las siguientes estructuras de control condicionales: *if-then* e *if-then-else*.

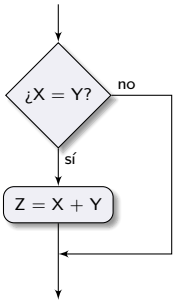
5.2.1. Estructura condicional 'if-then'

if-then es un tipo de estructura condicional que ejecuta o no un bloque de código en función de una determinada condición.

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*. El programa comprueba si el valor de la variable `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.

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

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



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

05_if.s

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

        .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 finsi
14           add r2, r0, r1  @-
15           ldr r3, =Z      @ [Z] <- [X] + [Y]
16           str r2, [r3]    @-
17
18 finsi:  wfi
```

▶ **5.4** Carga el programa anterior, ejecútalo y realiza los siguientes ejercicios:

5.4.1 ¿Qué valor contiene la dirección de memoria `Z` cuando finaliza el programa?

5.4.2 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 dirección de memoria `Z` cuando finaliza el programa?

Como se puede observar en el programa anterior, la idea fundamental para implementar la instrucción «**`if x==y:`**» ha consistido en utilizar una instrucción de salto condicional que salte cuando no se cumpla dicha condición, esto es, «**`bne`**». Es decir, se ha puesto como condición de salto, la condición contraria a la expresada en el «**`if`**». Al hacerlo así, si los dos valores comparados son iguales, la condición de salto no se dará y 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.

Además de implementar la estructura condicional *if-then*, el programa anterior constituye el primer ejemplo en el que se han combinado las instrucciones de transformación, de transferencia y de control de flujo, por lo que conviene detenerse un poco en su estructura, ya que es la que seguirán la mayor parte de programas a partir de ahora. Como se puede ver, en la zona de datos se han inicializado tres variables. Al comienzo del programa, dos de dichas variables, X e Y, se han cargado en sendos registros. Una vez hecho lo anterior, el resto del programa opera exclusivamente con registros, salvo cuando es necesario actualizar el valor de la variable «`Z`», en cuyo caso, se almacena la suma de $X + Y$, que está en el registro `r2`, en la dirección de memoria etiquetada con «`Z`». La Figura 5.1 muestra el esquema general que se seguirá a partir de ahora.

Zona de datos

- Inicialización de variables de entrada.
- Reserva de espacio para resultados.

Zona de código

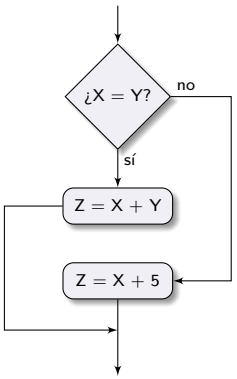
- Carga de variables de entrada en registros.
- Operaciones.
- Almacenamiento de los resultados en memoria.

Figura 5.1: Esquema general de un programa en ensamblador de ARM

5.2.2. Estructura condicional 'if-then-else'

Otra estructura condicional que suele utilizarse habitualmente es *if-then-else*. Se trata de una extensión de la estructura *if-then* vista en el apartado anterior, que permitir indicar qué se quiere hacer cuando se cumpla o no una determinada condición. Así por ejemplo, en el siguiente programa en Python3, si se cumple la condición —¿son iguales `X` e `Y`?— se realiza una acción, la misma que en el ejemplo anterior, sumar `X` e `Y`, y si no son iguales, se realiza otra acción diferente, sumar el número 5 a `X`.

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



Cuando se escriba un programa equivalente en código máquina, las instrucciones de salto que se utilicen determinarán, en función de la condición evaluada y del tipo de salto, qué instrucciones del programa se ejecutan y cuáles no. ¿Qué instrucciones de salto utilizarías? ¿Dónde las pondrías? Una vez que hayas esbozado una solución, compara tu respuesta con la siguiente implementación:

05_if_else.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
```

▶ **5.5** Carga el programa anterior, ejecútalo y realiza los siguientes ejercicios:

5.5.1 ¿Qué valor hay en la dirección de memoria `Z` cuando finaliza el programa?

5.5.2 Cambia el valor de `Y` para que sea distinto de `X` y vuelve a ejecutar el programa. ¿Qué valor hay ahora en la dirección de memoria `Z` al finalizar el programa?

5.5.3 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 que se tenga en cuenta dicho cambio. ¿Qué modificaciones has realizado en el programa en ensamblador?

5.3. Estructuras de control repetitivas

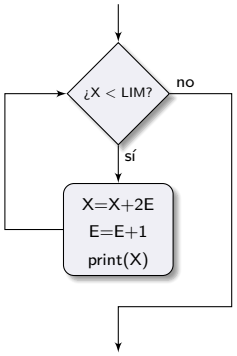
En el anterior apartado se ha visto cómo se pueden utilizar las instrucciones de salto para implementar las estructuras condicionales *if-then* e *if-then-else*. En este apartado se verán las dos estructuras de control repetitivas, también llamadas iterativas, que se utilizan con más frecuencia, *while* y *for*, y cómo se implementan al nivel de la máquina.

5.3.1. 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 a 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 flujo del programa vuelve a la evaluación de la condición y todo el proceso se vuelve a repetir mientras se cumpla la condición. Así por ejemplo, el siguiente programa en Python3, realizará las operaciones $X = X + 2 \cdot E$ y $E = E + 1$ mientras se cumpla que $X < LIM$. Por lo tanto, y dados los valores iniciales de X, E y LIM, la variable X irá tomando los siguientes valores con cada iteración del bucle *while*: 3, 7, 13, 21, 31, 43, 57, 73, 91 y 111.

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

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



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

05_while.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
        bge 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
```

```
22          str r1, [r4]          @ [E] <- r1
23        b   bucle
24
25  finbuc: wfi
```

▶ **5.6** Carga el programa anterior, ejecútalo y realiza los siguientes ejercicios:

5.6.1 ¿Qué proceso realizan de forma conjunta las instrucciones «`cmp r0, r2`», «`bge finbuc`» y «`b bucle`»?

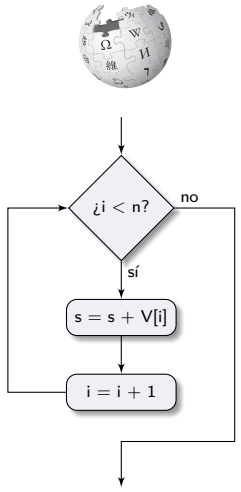
5.6.2 ¿Por qué se ha usado la instrucción de salto condicional «**bge**» y no «**blt**»? ¿Por qué «**bge**» y no «**bgt**»?

5.6.3 ¿Qué indicadores del registro de estado se comprueban cuando se ejecuta la instrucción «**bge**»? ¿Cómo deben estar dichos indicadores, activados o desactivados, para que se ejecute el interior del bucle?

5.6.4 ¿Qué instrucción se ha utilizado para calcular $2 \cdot E$? ¿Qué operación realiza dicha instrucción?

5.3.2. Estructura de control repetitiva 'for'

Un bucle *for* o *para* es una estructura de control que permite que un determinado código se ejecute repetidas veces. Se distingue de otros tipos de bucles, como el *while*, por el uso explícito de un contador.



Con frecuencia se suele utilizar como condición para finalizar un bucle, el número de veces que se ha iterado sobre él. En estos casos, se utiliza una variable para llevar la cuenta del número de iteraciones realizadas, que recibe el nombre de **contador**. Así pues, para implementar un bucle con dichas características, bastaría con inicializar un contador —p.e. a 0— y utilizar la estructura *while* vista en el apartado anterior, de tal forma que mientras dicho contador no alcance un determinado valor, se llevaría a cabo una iteración de las acciones que forman parte del bucle y se incrementaría en 1 el valor almacenado en el contador. Como este caso se da con bastante frecuencia, los lenguajes de programación de alto nivel suelen proporcionar una forma de llevarla a cabo directamente: el bucle *for*. El siguiente programa muestra un ejemplo de uso de la estructura de control repetitiva *for* en Python3, en el que se suman 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*.

```python
V = [2, 4, 6, 8, 10]
n = 5
suma = 0

for i in range(n):  # i = [0..N-1]
  suma = suma + V[i]
```

Conviene hacer notar que un programa en Python3 no necesitaría la variable n, ya que en Python3 un vector es una estructura de datos de la que es posible obtener su longitud utilizando la función «`len()`». Así que si se ha utilizado la variable n, ha sido simplemente 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:

05_for.s

```
        .data
V:      .word 2, 4, 6, 8, 10
n:      .word 5
suma:   .word 0

        .text
main:   ldr r0, =V      @ r0 <- dirección de V
        ldr r1, =n
        ldr r1, [r1]    @ r1 <- [n]
        ldr r2, =suma
        ldr r2, [r2]    @ r2 <- [suma]
        mov r3, #0      @ r3 <- 0

bucle:  cmp r3, r1
        beq finbuc
          ldr r4, [r0]
          add r2, r2, r4        @ r2 <- r2 + V[i]
          add r0, r0, #4
          add r3, r3, #1
        b bucle

finbuc: ldr r0, =suma
        str r2, [r0]    @ [suma] <- r2

stop:   wfi
```

En el código anterior se utiliza el registro `r3` para almacenar el contador del bucle, que se inicializa a 0, y el registro `r1` para almacenar la longitud del vector. Al principio del bucle se comprueba si el contador es igual a la longitud del vector (`r3` == `r1`). Si son iguales, se salta fuera del bucle. Si no, se realizan las operaciones indicadas en el interior del bucle, entre ellas, la de incrementar en uno el contador (`r3` $\leftarrow$ `r3` $+ 1$) y

se vuelve de nuevo al principio del bucle, donde se vuelve a comprobar si el contador es igual a la longitud del vector...

▶ **5.7** Carga el programa anterior, ejecútalo y realiza los siguientes ejercicios:

5.7.1 ¿Qué proceso realizan de forma conjunta las siguientes instrucciones «`cmp r3, r1`», «`beq finbuc`», «`add r3, r3, #1`» y «`b bucle`»?

5.7.2 ¿Qué indicador del registro de estado se está comprobando cuando se ejecuta la instrucción «`beq finbuc`»?

5.7.3 ¿En qué dirección de memoria se almacena cada uno de los elementos del vector V?

5.7.4 ¿Qué valores va tomando el registro `r0` durante la ejecución del código?

5.7.5 ¿Para qué se utiliza la instrucción «`ldr r4, [r0]`»?

5.7.6 ¿Para qué se utiliza la instrucción «`add r0, r0, #4`»?

5.7.7 ¿Qué valor contiene la dirección de memoria «`suma`» cuando finaliza la ejecución del programa?

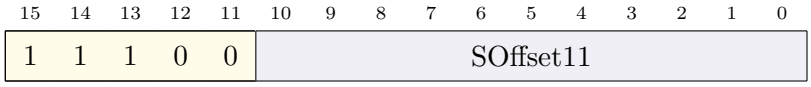
5.4. Modos de direccionamiento y formatos de instrucción de ARM

En este apartado se describen los modos de direccionamiento empleados en las instrucciones de salto y sus formatos de instrucción.

5.4.1. Direccionamiento de las instrucciones de salto

Uno de los operandos de las instrucciones de salto (§ 5.1) —tanto en las de salto incondicional, «`b etiqueta`», como en las de salto condicional, «`bXX etiqueta`»— es justamente la dirección de salto, que se indica por medio de una etiqueta, que a su vez referencia la dirección de memoria a la que se quiere saltar. ¿Cómo se codifica dicha dirección de memoria en una instrucción de salto?

Una primera opción sería la de codificar la dirección de salto en la propia instrucción. Pero esto plantea dos problemas. El primero de ellos es que 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



SOffset11 Dato inmediato con signo.

Figura 5.2: Formato de la instrucción de salto incondicional «`b etiqueta`»

las instrucciones de salto, puesto que no todas las instrucciones tienen que ser del mismo tamaño. El segundo de los problemas si se utiliza esta aproximación, la de codificar la dirección completa del salto como un valor absoluto, es que se forzaría a que el código se tuviera que ejecutar siempre en las mismas direcciones de memoria. No obstante, esta limitación se podría evitar durante la fase de carga del programa: bastaría con que el programa cargador, cuando cargara un código para su ejecución, sustituyera las direcciones de salto absolutas por nuevas direcciones teniendo en cuenta la dirección de memoria a partir de la cual se esté cargando el código [18]. Aunque de esta forma se solventaría el hecho de que el programa se deba ejecutar siempre en las mismas posiciones de memoria, esta opción implicaría que el programa cargador debería: I) reconocer los saltos absolutos en el código, II) calcular la nueva dirección de salto, y III) sustituir las direcciones de salto originales por las nuevas.

Por lo tanto, 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. Así pues, 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 utilizan el modo de direccionamiento relativo al PC para codificar la dirección de salto.

Por otro lado, el operando destino de las instrucciones de salto es siempre el registro `PC`, por lo que el modo de direccionamiento utilizado para indicar el operando destino es el implícito.

5.4.2. Formato de la instrucción de salto incondicional

El formato de instrucción utilizado para codificar la instrucción de salto incondicional, «`b etiqueta`», (véase la Figura 5.2) está formado

por dos campos. El primero de ellos corresponde al código de operación ($\texttt{11100}_2$). El segundo campo, `SOffset11`, que consta de 11 bits, se destina a codificar el desplazamiento. Para poder aprovechar mejor dicho espacio, se utiliza la misma técnica ya comentada anteriormente. Puesto que las instrucciones Thumb ocupan 16 o 32 bits, 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 `SOffset11`, lo que permite guardar los bits 1 al 11 del desplazamiento —en lugar de los bits del 0 al 10—. Pudiendo codificar, por tanto, el desplazamiento con 12 bits —en lugar de con 11—, lo que proporciona un rango de salto de $[-2\,048, 2\,046]$ bytes[1] con respecto al `PC`.

Hasta ahora sabemos que la dirección de memoria a la que se quiere saltar se calcula como un desplazamiento de 12 bits con respecto al contenido del registro `PC`. Sin embargo, no hemos dicho nada de cuál es el contenido del registro `PC` en el momento que se va a ejecutar el salto. Teóricamente, si el procesador ejecutara una instrucción tras otra, que es como lo hace el simulador, el contenido del registro `PC` en la fase de ejecución de la instrucción de salto sería $\texttt{PC}_{inst}+2$, siendo $\texttt{PC}_{inst}$ la dirección en la que se encuentra la instrucción de salto y $\texttt{PC}_{inst}+2$ la dirección de la siguiente instrucción. Por tanto, el valor del desplazamiento debería ser tal que al sumarse a $\texttt{PC}_{inst}+2$ diera la dirección de salto. Lo que no es cierto.

En realidad, para acelerar la ejecución de instrucciones, la arquitectura ARM Thumb utiliza un cauce de ejecución de instrucciones con 3 etapas: lectura, decodificación y ejecución. Sin entrar en detalles, esto implica que cuando la instrucción de salto está en la etapa de ejecución (3.ª etapa), se está decodificando la instrucción siguiente a la de salto (2.ª etapa) y se está leyendo una tercera instrucción (1.ª etapa). Por lo tanto, el registro `PC` tendrá en dicho momento el valor $\texttt{PC}_{inst}+4$, ya que se está leyendo la segunda instrucción después de la actual. Así pues, el desplazamiento codificado como dato inmediato deberá ser tal que al sumarse a $\texttt{PC}_{inst}+4$ se consiga la dirección de memoria a la que se quiere saltar. Para ilustrar esto, se utilizará el siguiente código formado por varias instrucciones que saltan a la misma dirección:

05_mod_dir_b.s

```
1         .text
2 main:   b  salto
3         b  salto
4         b  salto
5         b  salto
```

[1] 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, $[-1\,024, 1\,022]$ bytes con respecto al `PC`.

```
6  salto:  mov r0, r0
7          mov r1, r1
8          b   salto
9          b   salto
10
11 stop:   wfi
```

▶ **5.8** Copia el programa anterior y ensámblalo, pero no lo ejecutes (el programa no tiene ningún propósito como tal y si se ejecutara, entraría en un bucle sin fin).

5.8.1 ¿Cuál es la dirección de memoria de la instrucción etiquetada con «`salto`»?

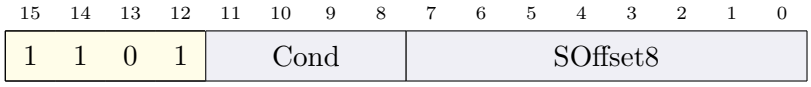
5.8.2 Según la explicación anterior, cuando se va a realizar el salto desde la primera instrucción, ¿qué valor tendrá el registro `PC`?, ¿qué número se ha puesto como desplazamiento?, ¿cuánto suman?

5.8.3 ¿Cuánto vale el campo `SOffset11` de la primera instrucción? ¿Qué relación hay entre el desplazamiento y el valor codificado de dicho desplazamiento?

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

5.4.3. Formato de las instrucciones de salto condicional

Las instrucciones de salto condicional se codifican de forma similar a como se codifica la de salto incondicional (véase la Figura 5.3). A diferencia del formato de la de salto incondicional, el campo utilizado para codificar el salto, `SOffset8`, es tan solo de 8 bits —frente a los 11 en la de salto incondicional—. Esto es debido a que este formato de instrucción debe proporcionar un campo adicional, `Cond`, para codificar la condición del salto (véase el Cuadro 5.1), por lo que quedan menos bits disponibles para codificar el resto de la instrucción. Puesto que el campo `SOffset8` solo dispone de 8 bits, el desplazamiento que se puede codificar en dicho campo corresponderá al de un número par en complemento a 2 con 9 bits —ganando un bit por el mismo razonamiento que el indicado para las instrucciones de salto incondicional—. Por tanto, el rango del desplazamiento de los saltos condicionales está limitado a $[-256, 254]$ con respecto al `PC+4`.



Cond Condición.

SOffset8 Dato inmediato con signo.

Figura 5.3: Formato de las instrucciones de salto condicional

5.5. Ejercicios

Ejercicios de nivel medio

▶ **5.9** 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 dirección 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 $B - A$.

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

Ejercicios avanzados

▶ **5.11** 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 $6 + 10 = 16$

▶ **5.12** Implementa un programa que dadas las notas de 2 exámenes parciales almacenadas en memoria (con notas entre 0 y 10), calcule la nota final (haciendo la media de las dos notas —recuerda que puedes utilizar la instrucción «`lsr`» para dividir entre dos—) y almacene en memoria la cadena de caracteres *APROBADO* si la nota final es mayor o igual a 5 y *SUSPENSO* si la nota final es inferior a 5.

Ejercicios adicionales

▶ **5.13** El siguiente programa muestra un ejemplo en el que se modifican los indicadores del registro de estado comparando el contenido

de distintos registros. Copia y ensambla el programa. A continuación, ejecuta el programa paso a paso y responde a las siguientes preguntas.

05_cambia_indicadores.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
```

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

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

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

▶ **5.14** Modifica el programa del Ejercicio 5.12 para que almacene en memoria la cadena de caracteres *SUSPENSO* si la nota final es menor a 5, *APROBADO* si la nota final es mayor o igual a 5 pero menor a 7, *NOTABLE* si la nota final es mayor o igual a 7 pero menor a 9 y *SOBRESALIENTE* si la nota final es igual o superior a 9.

▶ **5.15** Modifica el programa del ejercicio 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.

▶ **5.16** 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 $2 + 6 + 10 = 18$.

▶ **5.17** 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.

▶ **5.18** 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 6

Introducción a la gestión de subrutinas

Índice

En los capítulos anteriores se ha visto una gran parte del juego de instrucciones de la arquitectura ARM: las instrucciones de transformación —que permiten realizar operaciones aritméticas, lógicas y de desplazamiento—, las de carga y almacenamiento —necesarias para cargar las variables y almacenar los resultados— y las de control de flujo —que permiten condicionar la ejecución de determinadas instrucciones o el número de veces que se ejecuta un bloque de instrucciones—. De hecho, los lenguajes de programación pueden realizar transformaciones sobre datos, trabajar con variables en memoria e implementar las estructuras de programación condicionales y repetitivas, gracias a que el procesador es capaz de ejecutar las instrucciones vistas hasta ahora. Este capítulo introduce otra herramienta fundamental para la realización de programas en cualquier lenguaje: la subrutina. En este capítulo se explicará

qué son las subrutinas, cómo se gestionan a bajo nivel y el soporte que proporciona la arquitectura ARM para su gestión.

Una subrutina es un fragmento de código independiente, que normalmente realiza una tarea auxiliar completa, a la que se puede llamar mediante una instrucción específica, desde cualquier parte de un programa y que, una vez ha completado su cometido, devuelve el control a la instrucción siguiente a la que había efectuado la llamada. Es habitual que cuando se llame a una subrutina se le pasen uno o varios datos, llamados parámetros, para que opere con ellos, para que realice unas acciones u otras, etcétera. Asimismo, también es frecuente que la subrutina, al terminar, devuelva uno o varios resultados a la parte del programa que la llamó. Por otro lado, 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*, *método* o *subprograma*. De hecho, es probable que ya hayas oído alguno de dichos nombres. 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 ellos, ya que en ensamblador no se hace dicha distinción. 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`».

Una subrutina es una parte del programa a la que se puede llamar para resolver una tarea específica.

En este libro se utiliza el término *subrutina* para referirse de forma indistinta a rutinas, procedimientos, funciones, métodos o subprogramas.

```
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 se le van a pasar. En el ejemplo anterior, la subrutina se llama «`multadd`» y espera recibir tres parámetros, nombrados como «`x`», «`y`» y «`a`» —estos nombres se utilizarán en el código de la subrutina para hacer referencia a los parámetros recibidos—. El código indentado que aparece a continuación es el código de la subrutina. En este ejemplo, consiste en dos instrucciones. La primera, «`res = x*y +a`», realiza una operación con los parámetros de entrada y la segunda, «`return res`», sigue la sintaxis de Python3 para la devolución de resultados. Así pues, las acciones que lleva a cabo dicha subrutina, 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 com-

pletada la operación, II) devuelve el control al punto siguiente desde el que fue llamada, que puede acceder al resultado proporcionado.

Como se puede observar, la subrutina «`multadd`» es llamada dos veces desde el programa principal. En 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`», que 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`.

Como se ha podido comprobar, «`multadd`» responde efectivamente a la definición dada de subrutina: es un fragmento de código independiente, que realiza una tarea auxiliar completa, que se puede llamar desde cualquier parte de un programa y que, una vez ha completado su cometido, devuelve el control al punto del programa inmediatamente después de donde se había efectuado la llamada. Además, se le pasan varios parámetros para que opere con ellos, y, al terminar, devuelve un resultado al programa que la llamó.

¿Por qué utilizar subrutinas? 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 el mismo fragmento de código en todas y en cada una de las partes del programa en las que este fuera necesario. Es más, si en un momento dado se descubre un error en un trozo de código que se ha repetido en varias partes del programa, sería necesario revisarlas todas 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 se ha copiado. 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.

Esta división de los programas en subrutinas es importante porque permite estructurarlos y desarrollarlos de forma modular: cada subrutina es un trozo de código independiente que realiza una tarea, y el resto del programa puede hacerse sabiendo únicamente qué hace la subrutina y cuál es su interfaz, es decir, con qué parámetros debe comunicarse con ella. No es necesario saber cómo está programado el código de la subrutina para poder utilizarla en un programa. Por esta misma razón, la mayor parte de depuradores de código y simuladores —incluyendo QtARMSim—, incluyen la opción de depurar paso a paso pasando por encima —*step over*—. Esta opción de depuración, cuando se encuentra con una llamada a una subrutina, ejecuta la subrutina como si fuera una única instrucción, en lugar de entrar en su código. Sabiendo qué pasar a una subrutina y qué valores devuelve, es decir conociendo su interfaz, y habiéndola probado con anterioridad, es posible centrarse en la depuración del resto del programa de forma independiente, aprovechando la modularidad comentada anteriormente.

Así pues, 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). Cuando un programador requiere resolver un determinado problema ya resuelto por otro, le basta con recurrir a una determinada biblioteca y llamar a la subrutina adecuada. Es decir, gracias a la agrupación de subrutinas en bibliotecas, el mismo código puede ser reutilizado por muchos programas.

Library se suele traducir erróneamente en castellano como 'librería'; la traducción correcta es 'biblioteca'.

Desde el punto de vista de los mecanismos proporcionados por un procesador para soportar el uso de subrutinas, el diseño de una arquitectura debe hacerse de tal forma que facilite la realización de las siguientes acciones: I) la llamada a una subrutina, II) el paso de los parámetros con los que debe operar la subrutina, III) la devolución de los resultados, y IV) la continuación de la ejecución del programa a partir de la siguiente instrucción en código máquina a la que invocó a la subrutina.

En este capítulo se presentan los aspectos básicos de la gestión de subrutinas en ensamblador Thumb de ARM: cómo llamar y retornar de una subrutina y cómo intercambiar información entre el programa que llama a la subrutina y esta por medio de registros. El siguiente capítulo mostrará aspectos más avanzados de dicha gestión.

6.1. Llamada y retorno de una subrutina

ARM Thumb proporciona las siguientes instrucciones para gestionar la llamada y el retorno de una subrutina: «`bl etiqueta`» y «`mov pc, lr`».

La instrucción «`bl etiqueta`» se utiliza para **llamar** a una subrutina que comienza en la dirección de memoria indicada por dicha etiqueta. Cuando el procesador ejecuta esta instrucción, lleva a cabo las siguientes acciones:

- 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*, registro enlace). Es decir,[1] `LR` ← `PC` + `4`.
- Transfiere 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`).

La instrucción «`mov pc, lr`» se utiliza al final de la subrutina para **retornar** a la instrucción siguiente a la que la había llamado. Cuando el procesador ejecuta 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`.

Así pues, las instrucciones «`bl etiqueta`» y «`mov pc, lr`» permiten programar de forma sencilla la llamada y el retorno desde una subrutina. En primer lugar, basta con utilizar «`bl etiqueta`» para llamar a la subrutina. Cuando el procesador ejecute dicha instrucción, almacenará en `LR` la dirección de vuelta y saltará a la dirección indicada por la etiqueta. Al final de la subrutina, para retornar a la siguiente instrucción a la que la llamó, es suficiente con utilizar la instrucción «`mov pc, lr`». Cuando el procesador ejecute dicha instrucción, sobrescribirá el `PC` con el contenido del registro `LR`, que enlaza a la instrucción siguiente a la que llamó a la subrutina. La única precaución que conviene tomar es que el contenido del registro `LR` no se modifique durante la ejecución de la subrutina. Este funcionamiento queda esquematizado en la Figura 6.1, en la que se puede ver como desde una parte del programa, a la que se suele llamar *programa invocador*, se realizan tres llamadas a la parte del programa correspondiente a la subrutina.

El siguiente código muestra un programa de ejemplo en ensamblador de ARM 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 `r0`. Como se puede observar, la subrutina se llama desde dos puntos del programa principal —la primera línea del programa principal es la etiquetada con «`main`»—.

[1] Puesto que la instrucción «`bl etiqueta`» se codifica utilizando dos medias palabras, la dirección de la siguiente instrucción, la dirección de retorno, se calculará como `PC` + 4.

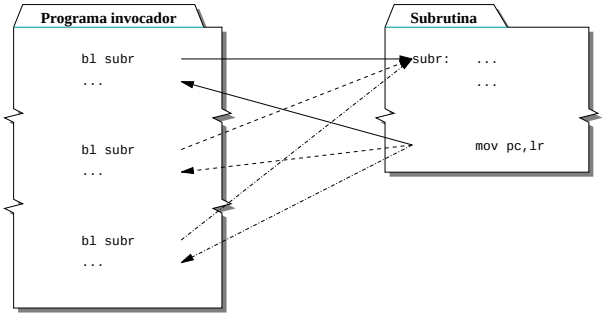


Figura 6.1: Llamada y retorno de una subrutina

06_suma_valor.s

```
1           .data
2  datos:   .word 5, 8, 3, 4
3  suma1:   .space 4
4  suma2:   .space 4
5
6           .text
7           @ --------------------
8           @ Programa invocador
9           @ --------------------
10 main:    ldr r4, =datos
11
12          ldr r0, [r4]
13          ldr r1, [r4, #4]
14 primera: bl suma
15          ldr r5, =suma1
16          str r0, [r5]
17
18          ldr r0, [r4, #8]
19          ldr r1, [r4, #12]
20 segunda: bl suma
21          ldr r5, =suma2
22          str r0, [r5]
23
24 stop:    wfi
25
26          @ --------------------
27          @ Subrutina
28          @ --------------------
29 suma:    add r0, r0, r1
30          mov pc, lr
31
32          .end
```

▶ **6.1** Copia el programa anterior en el simulador y mientras realizas una ejecución paso a paso contesta a las siguientes preguntas. Utiliza la versión de paso a paso entrando *(step into)*, ya que la versión paso a paso por encima *(step over)* en lugar de entrar en la subrutina, que es lo que se quiere en este caso, la ejecutaría como si se tratara de una única instrucción.

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

	Antes	Después
`PC`		
`LR`		

6.1.2 ¿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`»?

	Antes	Después
`PC`		
`LR`		

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

	Antes	Después
`PC`		
`LR`		

6.1.4 ¿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`»?

	Antes	Después
`PC`		
`LR`		

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

6.1.6 Crea un nuevo programa a partir del anterior en el que la subrutina «`suma`» devuelva en `r0` el doble de la suma de `r1` y

`r0`. Ejecuta el programa y anota el contenido de las variables «`suma1`» y «`suma2`».

. . . . . .

6.2. 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*. Los parámetros de entrada proporcionan información del programa invocador a la subrutina. Los de salida devuelven información de la subrutina al programa invocador. Por último, los 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 recurso físico donde se pueda almacenar y leer la información que se quiere transferir. Las dos opciones más comunes son los registros o la memoria —mediante una estructura de datos llamada pila, que se describirá en el siguiente capítulo—. El que se utilicen registros, la pila o ambos, depende de la arquitectura en cuestión y del convenio adoptado en dicha arquitectura para el paso de parámetros.

Para poder utilizar las subrutinas conociendo únicamente su interfaz, pudiendo de esta manera emplear funciones de bibliotecas, es necesario establecer un convenio acerca de cómo pasar y devolver los parámetros para que cualquier programa pueda utilizar cualquier subrutina aunque estén programados de forma independiente. Este capítulo se va a ocupar únicamente del paso de parámetros por medio de registros y, en este caso, la arquitectura ARM adopta como convenio el paso mediante los registros `r0`, `r1`, `r2` y `r3`, ya sea para parámetros de entrada, de salida o de entrada/salida. Para aquellos casos en los que se tengan que pasar más de 4 parámetros, el convenio define cómo pasar el resto de parámetros mediante la pila, tal y como se verá en el siguiente capítulo.

El último aspecto a tener en cuenta del paso de parámetros es cómo se transfiere cada uno de los parámetros. Hay dos formas de hacerlo: por valor o por referencia. Se dice que un parámetro se pasa por valor cuando lo que se transfiere es el dato en sí. Por otra parte, un parámetro se pasa por referencia cuando lo que se transfiere es la dirección de memoria en la que se encuentra dicho dato.

Sobre la base de 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.

Los siguientes subapartados muestran cómo pasar parámetros por valor y por referencia, cómo decidir qué tipo es el más adecuado para cada parámetro y, por último, cómo planificar el desarrollo de una subrutina por medio de un ejemplo más elaborado. En dichos subapartados se muestran ejemplos de parámetros de entrada, de salida y de entrada/salida y cómo seguir el convenio de ARM para pasar los parámetros mediante registros.

6.2.1. Paso de parámetros por valor

Como se ha comentado, un parámetro se pasa por valor cuando únicamente se transfiere su valor. El paso de parámetros por valor implica la siguiente secuencia de acciones (véase la Figura 6.2):

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.

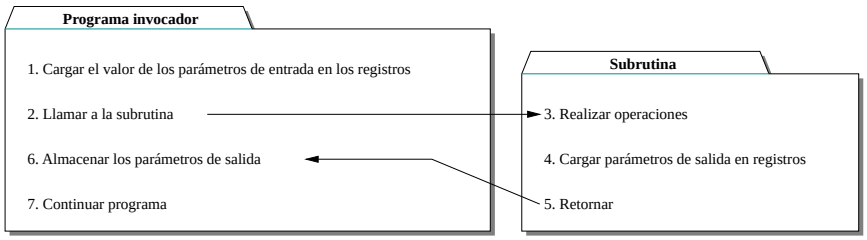


Figura 6.2: Paso de parámetros por valor

En el siguiente código, ya visto previamente, se puede observar cómo se pasan por valor dos parámetros de entrada y uno de salida.

06_suma_valor.s

```
1         .data
2 datos:  .word 5, 8, 3, 4
3 suma1:  .space 4
4 suma2:  .space 4
5
6         .text
7         @ --------------------
8         @ Programa invocador
9         @ --------------------
10 main:  ldr r4, =datos
11
12        ldr r0, [r4]
13        ldr r1, [r4, #4]
14 primera: bl suma
15        ldr r5, =suma1
16        str r0, [r5]
17
18        ldr r0, [r4, #8]
19        ldr r1, [r4, #12]
20 segunda: bl suma
21        ldr r5, =suma2
22        str r0, [r5]
23
24 stop:  wfi
25
26        @ --------------------
27        @ Subrutina
28        @ --------------------
29 suma:  add r0, r0, r1
30        mov pc, lr
31
32        .end
```

▶ **6.2** Contesta las siguientes preguntas con respecto al código anterior:

6.2.1 Enumera los registros que se han utilizado para pasar los parámetros de entrada a la subrutina.

6.2.2 Anota qué registro se ha utilizado para devolver el resultado al programa invocador.

6.2.3 Señala qué instrucciones se corresponden a cada una de las acciones enumeradas en la Figura 6.2. (Hazlo únicamente para la primera de las dos llamadas.)

6.2.2. Paso de parámetros por referencia

Como ya se ha comentado, el paso de parámetros por referencia consiste en pasar las direcciones de memoria en las que se encuentran los parámetros. Para pasar los parámetros por referencia se debe realizar la siguiente secuencia de acciones (véase la Figura 6.3):

- 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.

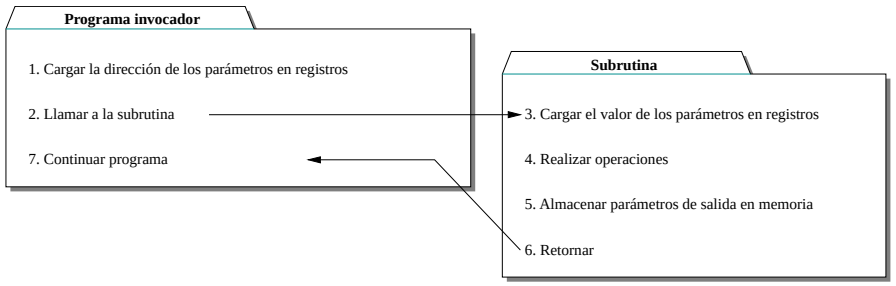


Figura 6.3: Paso de parámetros por referencia

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.

06_suma_referencia.s

```
1           .data
2   datos:  .word 5, 8, 3, 4
3   suma1:  .space 4
4   suma2:  .space 4
5
6           .text
7           @ --------------------
8           @ Programa invocador
9           @ --------------------
10  main:   ldr r0, =datos
11          ldr r1, =datos + 4
```

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

▶ **6.3** Contesta las siguientes preguntas con respecto al código anterior:

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

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

6.3.3 Señala qué instrucciones se corresponden con cada una de las acciones enumeradas en la Figura 6.3. (Hazlo únicamente para la primera de las dos llamadas.)

6.2.3. 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 dos formas, por referencia o por valor, es más conveniente para cada parámetro. 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. Esto implica que por regla general no todos los parámetros de una subrutina utilizarán la misma forma de paso. De hecho, la decisión de cómo pasar los parámetros a una subrutina viene condicionada en gran medida por la estructura de datos de los parámetros que se quieren pasar. Si se trata de un dato de tipo estructurado —vectores, matrices, cadenas, estructuras...—, que tienen tamaño indeterminado y solo pueden residir

completos en memoria, no hay alternativa posible, el paso siempre se hace por referencia. Independientemente de 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...—, que puede estar contenido temporalmente en un registro, entonces sí se puede decidir si se pasa por valor o por referencia. Hacerlo de una forma u otra depende entonces del sentido en el que se va a transferir 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, por lo que 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. Si se hace por valor, la subrutina devuelve el dato utilizando el registro reservado para ello. Si se hace por referencia, la subrutina almacenará en memoria el dato, pero para ello el programa invocador deberá haberle pasado previamente en qué dirección de memoria deberá almacenarlo.
- Parámetro de entrada/salida. Un parámetro de este tipo proporciona un valor que la subrutina necesita conocer y en el que posteriormente devolverá el resultado. En este caso, se debe pasar por referencia.

6.2.4. Un ejemplo más elaborado

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

El programa que se propone desarrollar debe obtener cuántos elementos de un vector dado tienen unos valores determinados, p.e., cuántos elementos valen `7` o `8`. La solución que se propone incluye el desarrollo de una subrutina que devuelva el número de elementos de un vector que sean iguales a uno dado y el de un programa que llamará a esa subrutina tantas veces como valores distintos se quieran contabilizar y que acabe sumando los resultados obtenidos en cada llamada. Siguiendo con el ejemplo anterior, el programa debería llamar una vez a la subrutina para obtener el número de elementos iguales a `7` y otra, para obtener

los elementos iguales a 8, y finalmente deberá sumar ambos resultados y obtener el número de elementos del vector que son iguales a 7 o a 8.

A continuación se muestra una posible implementación en Python3 de dicho programa. Como se puede ver, el programa consta de una subrutina, «numigualque», que recibe los parámetros «vector_s», «dim_s» y «dato_s». Esta subrutina recorre el vector «vector_s», cuyo tamaño viene dado por «dim_s», contabilizando el número de elementos que son iguales a «dato_s»; y devuelve dicho número como resultado. Por su parte, el programa principal comienza inicializando un vector y su tamaño; a continuación, llama dos veces a la anterior subrutina, pasándole el mismo vector y dimensión, pero con los números a comparar con «7» y «8», respectivamente, y almacena el resultado de ambas llamadas en sendas variables «res1» y «res2»; finalmente, suma ambos resultados para obtener el número de elementos del vector que son iguales a 7 o a 8.

```
1  def numigualque(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, 8, 5, 6, 8, 10, 4, 8, 7, 5, 7]
9  dim = 12
10 res1 = numigualque(vector, dim, 7)
11 res2 = numigualque(vector, dim, 8)
12 res = res1 + res2
```

En los siguientes ejercicios se propondrá desarrollar paso a paso una versión en ensamblador de dicho programa. Así pues, en primer lugar, se deberá desarrollar una subrutina que contabilice cuántos elementos de un vector son iguales 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 la parte del programa que llama dos veces a dicha subrutina y calcula el número de elementos del vector dado que son iguales a 7 o a 8 sumando ambos resultados.

▶ **6.4** 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 deben pasar por referencia y cuáles por valor?

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

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

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

▶ **6.7** Comprueba que el programa escrito funciona correctamente. ¿Qué valor se almacena en «`res`» cuando se ejecuta? Modifica el contenido del vector «`vector`» variando el número de `7` o de `8`, ejecuta de nuevo el programa y comprueba que el nuevo resultado obtenido también es correcto.

6.3. Ejercicios

Ejercicios de nivel medio

▶ **6.8** Modifica la subrutina y el programa principal desarrollados en los Ejercicios 6.5 y 6.6, tal y como se describe a continuación:

a) Modifica la subrutina del Ejercicio 6.5 para que dado un vector, su dimensión y un número n, devuelva el número de elementos del vector que son menores o iguales al número n. Cambia el nombre de la subrutina a «`nummenoroigualque`».

b) Modifica el programa invocador desarrollado en el Ejercicio 6.6 para que almacene en memoria el número de elementos de un vector —p.e., el vector que aparecía en el ejercicio original, $[5, 3, 5, 5, 8, 12, 12, 15, 12]$— que son menores o iguales a 5 —utilizando la subrutina desarrollada en el apartado a) de este ejercicio—.

Ejercicios avanzados

▶ **6.9** Realiza los siguientes desarrollos:

a) Implementa una subrutina a la que se le pase la dirección de comienzo de una cadena de caracteres que finalice con el carácter nulo y devuelva su longitud. Es recomendable que antes de empezar a escribir la subrutina, respondas las preguntas planteadas en el Ejercicio 6.4.

b) Implementa un programa en el que se inicialicen dos cadenas de caracteres y calcule cuál de las dos cadenas es más larga. Dicho programa deberá escribir un `1` en una posición de memoria etiquetada como «`res`» si la primera cadena es más larga que la segunda y un `2` en caso contrario. Para obtener el tamaño de cada cadena se deberá llamar a la subrutina desarrollada en el apartado a) de este ejercicio.

Ejercicios adicionales

▶ **6.10** 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. Además, el programa deberá mostrar en el visualizador LCD el número de elementos pares de ambos vectores. Para ello, el programa deberá utilizar la subrutina desarrollada en el apartado a) de este ejercicio.

▶ **6.11** 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). No obstante, se debe tener en cuenta que la matriz debería poder tener cualquier dimensión, por lo que se deberá utilizar un bucle para recorrer sus filas.

Capítulo 7

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 que 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 llamada a la subrutina, 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. En cualquier caso, se omitieron en dicho capítulo una serie de aspectos como son: I) ¿qué registros puede modificar una subrutina?, II) ¿cómo preservar el valor de los registros que no pueden modificarse al llamar a una subrutina?, III) ¿cómo pasar más de cuatro parámetros?, y IV) ¿cómo preservar el contenido del registro `LR` si la subrutina tiene que llamar a su vez a otra subrutina?

Los aspectos que, por simplicidad, no se trataron en el capítulo anterior se corresponden con la gestión de la información que deben realizar las subrutinas. 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.

Una subrutina solo puede modificar los registros del `r0` al `r3`, los utilizados para el paso de parámetros.

Con respecto al primero de los aspectos no vistos en el capítulo anterior, ¿qué registros puede modificar una subrutina? El convenio de ARM establece que una subrutina solo puede modificar el contenido de los registros `r0` al `r3`. ¿Qué implicaciones tiene esto? Desde el punto de vista del desarrollo del programa invocador, se ha de tener en cuenta que cada vez que se llama a una subrutina, los registros `r0` al `r3` han podido ser modificados, por lo que una vez realizada la llamada, se debe suponer que no mantienen el valor que tuvieran antes de la llamada. Una consideración adicional es que para almacenar un dato que se quiera utilizar antes y después de la llamada, será conveniente utilizar un registro del `r4` en adelante, ya que la subrutina tiene la obligación de preservar su contenido. Desde el otro punto de vista, el del desarrollo de la subrutina, el que solo se puedan modificar los registros del `r0` al `r3` implica que, salvo que la subrutina sea muy sencilla y, por tanto, sea posible desarrollarla contando únicamente con dichos registros, será necesario crear y gestionar un espacio de memoria donde la subrutina pueda almacenar la información adicional que vaya a necesitar durante su ejecución. A este espacio de memoria, cuya definición se matizará más adelante, se le denomina bloque de activación de la subrutina y se implementa por medio de una estructura de datos conocida como pila. La gestión del bloque de activación de una subrutina constituye un tema central en la gestión de subrutinas.

El resto de este capítulo está organizado como sigue. El primer apartado describe la estructura de datos conocida como pila y cómo se utiliza en la arquitectura ARM. El segundo apartado describe cómo se construye y gestiona el bloque de activación de una subrutina. Por último, se propone una serie de ejercicios relacionados con la gestión de subrutinas.

7.1. 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 orden 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, este 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 de propósito general no dispone de un dispositivo específico que implemente una pila en la que se puedan introducir y extraer datos. En realidad, la pila se implementa por medio de dos de los elementos ya conocidos de un computador: la memoria y un registro. La memoria sirve para almacenar los elementos que se van introduciendo en la pila y el registro para apuntar a la dirección del último elemento introducido en la pila (que recibe el nombre de **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 almacenarla. 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, estos 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 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: I) 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 7.1 muestra el contenido de la pila y el valor del registro `SP` antes y después de apilar el contenido del registro `r4`.

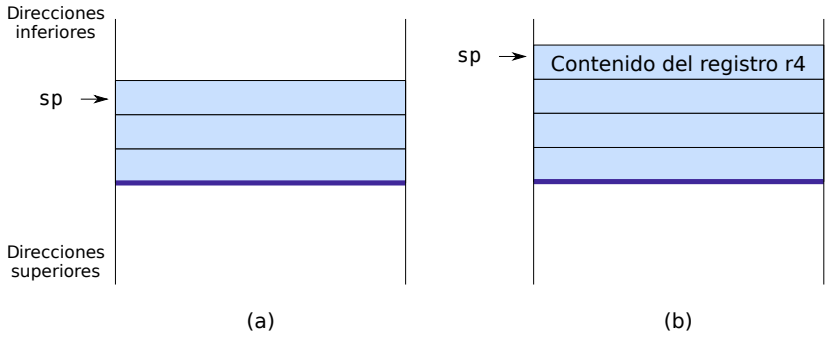


Figura 7.1: La pila antes y después de apilar el registro `r4`

La operación 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 alineado a 4 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 dirección indicada por el registro `SP` en el registro `r4`, «`ldr r4, [sp]`», y II) incrementar en 4 posiciones el puntero de pila, «`add sp, sp, #4`». La Figura 7.2 muestra el contenido de la pila y el valor del registro `SP` antes y después de desapilar una palabra.

▶ **7.1** Contesta las siguientes preguntas relacionadas con la operación apilar.

7.1.1 Suponiendo que el puntero de pila contiene `0x2007 06FC` 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?

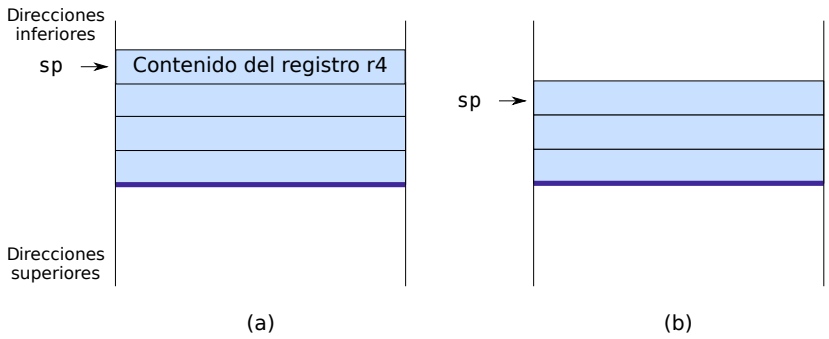


Figura 7.2: La pila antes y después de desapilar el registro `r4`

7.1.2 ¿Qué instrucción se utilizará para almacenar el contenido del registro `r5` en la dirección apuntada por el puntero de pila?

7.1.3 A partir de las dos preguntas anteriores, indica qué dos instrucciones permiten apilar en la pila el registro `r5`.

▶ **7.2** El siguiente fragmento de código apila, uno detrás de otro, el contenido de los registros `r4` y `r5`. Copia dicho programa en el simulador, cambia al modo de simulación y realiza los ejercicios que se muestran a continuación.

07_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      [sp] <- r4
7         sub sp, sp, #4  @ Actualiza sp   sp  <- sp - 4
8         str r5, [sp]    @ Apila r5      [sp] <- r5
9
10 stop:  wfi
11        .end
```

7.2.1 Ejecuta el programa paso a paso y comprueba en qué posiciones de memoria, pertenecientes a la pila, se almacenan los contenidos de los registros `r4` y `r5`.

7.2.2 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.

▶ **7.3** Contesta las siguientes preguntas relacionadas con la operación desapilar.

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

7.3.2 Suponiendo que el puntero de pila contiene `0x200706F8`, ¿qué valor deberá pasar a tener el puntero de pila después de desapilar una palabra (4 bytes) de la pila? ¿Qué instrucción en ensamblador ARM se utilizará para actualizar el puntero de pila?

7.3.3 A partir de las dos preguntas anteriores, indica qué dos instrucciones permiten desapilar de la pila el registro `r5`.

. . . . . .

7.1.1. 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 dichas acciones proporcionando dos instrucciones que se encargan de realizar automáticamente todos los pasos vistos en el apartado anterior: «**push**» y «**pop**». Como estas instrucciones permiten apilar o desapilar varios registros, su sintaxis es específica y está pensada para tal fin. Así, entre las llaves que encierran el operando se pueden incluir: varios registros separados por comas (ej. «`{r1, r3, r6, lr}`»), un rango completo de registros indicando el primero y el último y separándolos con un guion (ej. «`{r3-r7}`») o una combinación de ambos (ej. «`{r1-r5, r7, pc}`»). Es importante indicar que además de los registros de propósito general se puede incluir el registro `LR` en una instrucción «**push**» y el registro `PC` en una «**pop**». De esta manera, se puede guardar en la pila la dirección de retorno mediante la instrucción «**push**» y copiarla en el contador de programa mediante la instrucción «**pop**». A lo largo de este capítulo se verá la conveniencia de ello. A modo de ejemplo de utilización de las instrucciones «**push**» y «**pop**», 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**».

«**push**»
«**pop**»

07_apilar_r4r5_v2.s

```
1         .text
2 main:
3         mov r4, #4
4         mov r5, #5
5         push {r4, r5}
```

```
6           mov r4, #40
7           mov r5, #50
8           pop {r4, r5}
9
10  stop:   wfi
11          .end
```

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

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

7.4.2 ¿En qué posiciones de memoria, pertenecientes a la pila, se almacenan los contenidos de los registros `r4` y `r5`?

7.4.3 ¿Qué valores toman los registros `r4` y `r5` una vez realizadas las dos operaciones «**mov**» que están detrás del primer «**push**»?

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

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

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

7.2. Bloque de activación de una subrutina

Aunque la pila se utiliza 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 una subrutina. Puesto que en el apartado anterior se ha explicado qué es la pila, ahora se puede particularizar la definición previa de bloque de activación, para decir que el **bloque de activación de una subrutina** es la parte de la pila que contiene la información requerida por una subrutina. El bloque de activación de una subrutina tiene los siguientes cometidos:

- Almacenar la dirección de retorno original, en el caso de que la subrutina llame a otras subrutinas.
- 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 espera que no sean modificados.
- Mantener los valores que se han pasado como argumentos a la subrutina.

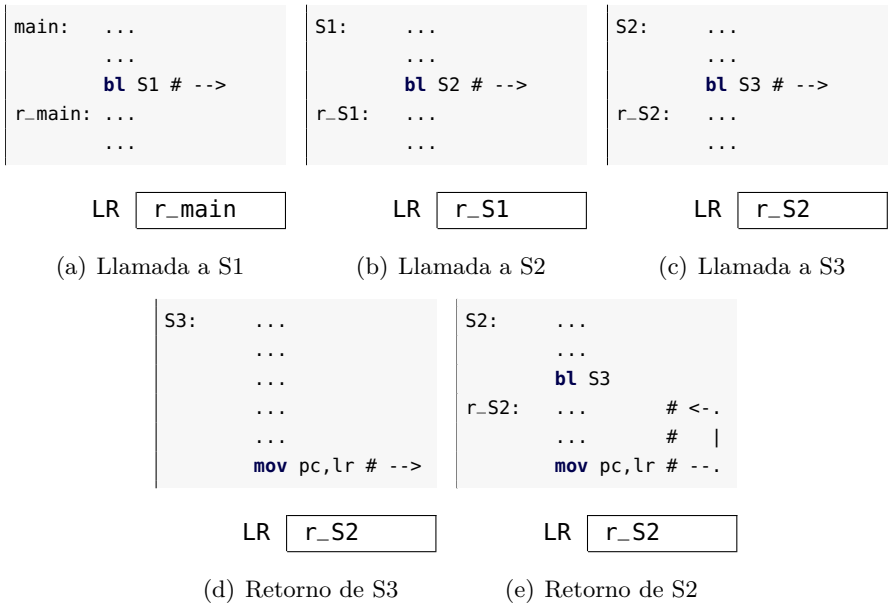
Los siguientes subapartados describen los distintos aspectos de la gestión de las subrutinas en los que se hace uso del bloque de activación de una subrutina. El Subapartado 7.2.1 trata el problema del anidamiento de subrutinas —subrutinas que llaman a otras o a sí mismas—. El Subapartado 7.2.2, sobre cómo utilizar el bloque de activación para almacenar las variables locales de la subrutina. El Subapartado 7.2.3, sobre cómo preservar el valor de aquellos registros que la subrutina necesita utilizar pero cuyo valor se debe restaurar antes de devolver el control al programa invocador. El Subapartado 7.2.4 muestra la estructura y gestión del bloque de activación. El Subapartado 7.2.5 describe el convenio completo que se ha de seguir en las llamadas a subrutinas para la creación y gestión del bloque de activación. Por último, el Subapartado 7.2.6 muestra un ejemplo en el que se utiliza el bloque de activación para gestionar la información requerida por una subrutina.

7.2.1. Anidamiento de subrutinas

Hasta este momento se han visto ejemplos relativamente sencillos en los que un programa invocador llamaba una o más veces a una subrutina. Sin embargo, conforme la tarea a realizar se hace más compleja, suele ser habitual que una subrutina llame a su vez a otras subrutinas, que se hacen cargo de determinadas subtareas. Esta situación en la que una subrutina llama a su vez a otras subrutinas, recibe el nombre de anidamiento de subrutinas. Conviene destacar que cuando se implementa un algoritmo recursivo, se da un caso particular de anidamiento de subrutinas, en el que una subrutina se llama a sí misma para resolver de forma recursiva un determinado problema.

Cuando se anidan subrutinas, una subrutina que llame a otra, una vez finalizada su ejecución, deberá devolver el control del programa a la instrucción siguiente a la que la llamó. Por su parte, una subrutina invocada por otra deberá devolver el control del programa a la instrucción siguiente a la que realizó la llamada, que estará en la subrutina que la llamó. Como se ha visto en el capítulo anterior, cuando se ejecuta «`bl etiqueta`» para llamar a una subrutina, antes de realizar el salto

propiamente dicho, se almacena la dirección de retorno en el registro `LR`. También se ha visto que cuando finaliza la subrutina, la forma de devolver el control al programa invocador consiste en sobrescribir el registro `PC` con la dirección de vuelta, p.e., utilizando la instrucción «**`mov pc, lr`**». Si durante la ejecución de la subrutina, esta llama a otra, o a sí misma, al ejecutarse la instrucción «**`bl`**», se almacenará en el registro `LR` la nueva dirección de retorno, sobrescribiendo su contenido original. Por lo tanto, la dirección de retorno almacenada en el registro `LR` tras ejecutar el «**`bl`**» que llamó a la subrutina inicialmente, se perderá. Si no se hiciera nada al respecto, cuando se ejecutaran las correspondientes instrucciones de vuelta, se retornaría siempre a la misma dirección de memoria, a la almacenada en el registro `LR` por la última instrucción «**`bl`**». Este error se ilustra en la Figura 7.3, que muestra de forma esquemática qué ocurre si no se gestionan correctamente las direcciones de retorno.



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

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

Figura 7.3: Llamadas anidadas a subrutinas cuando no se gestionan las direcciones de retorno. Las distintas subfiguras muestran distintos momentos en la ejecución de un programa en ensamblador en el que desde el programa principal se llama a una subrutina «`S1`», que a su vez llama a una subrutina «`S2`», que llama a una subrutina «`S3`». En cada subfigura se indica el contenido del registro `LR` tras ejecutar la última instrucción mostrada. Como se puede ver en la subfigura (e), al ejecutar la instrucción «**`mov pc,lr`**», se entra en un bucle sin fin.

▶ **7.5** En el siguiente programa se llama a subrutinas de forma anidada sin gestionar las direcciones de vuelta. Edita el programa en el simulador, ejecútalo paso a paso (utilizando la opción *step into*) y contesta a las siguientes cuestiones.

07_llamada.s

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

7.5.1 ¿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`»?

7.5.2 ¿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`»?

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

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

7.5.5 Explica qué ocurre cuando el procesador ejecuta dicho programa.

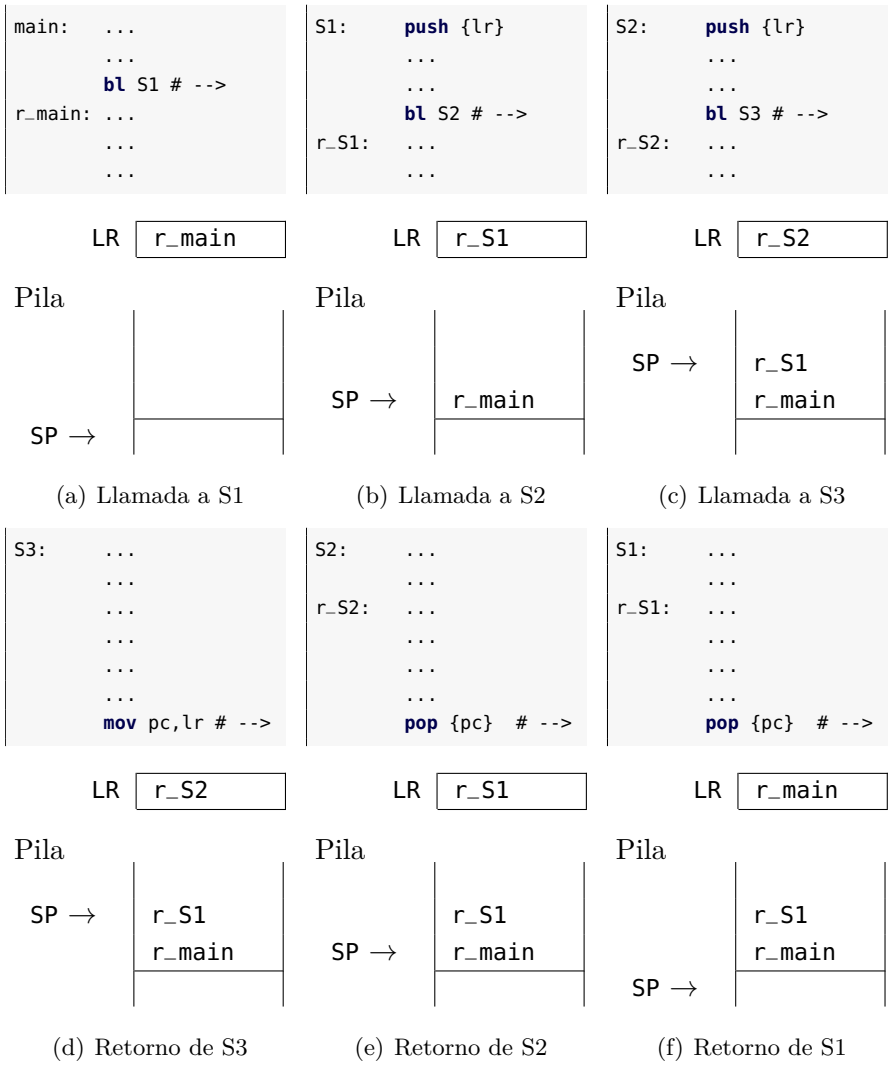
Como ha quedado patente en la Figura 7.3 y en el Ejercicio 7.5, cuando se realizan llamadas anidadas, es necesario almacenar de alguna forma las distintas direcciones de retorno. Dicho almacenamiento debe satisfacer dos requisitos. En primer lugar, debe ser capaz de permitir recuperar las direcciones de retorno en orden inverso a su almacenamiento —ya que es el orden en el que se van a producir los retornos—. En segundo lugar, el espacio reservado para este cometido debe poder crecer de forma dinámica —ya que la mayoría de las veces no se conoce cuántas llamadas se van a producir, puesto que dicho número puede depender de cuáles sean los datos del problema—. Como ya se habrá podido intuir, la estructura de datos que mejor se adapta a los anteriores requisitos es la pila. Si se utiliza la pila para almacenar y recuperar las direcciones de retorno, bastará con proceder de la siguiente forma: I) antes de realizar una llamada a otra subrutina (o a sí misma), la subrutina deberá apilar la dirección de retorno actual, y II) antes de retornar, deberá desapilar la última dirección de retorno apilada. Es decir, la subrutina deberá apilar el registro `LR` antes de llamar a otra subrutina y desapilar dicho registro antes de retornar. La Figura 7.4 muestra de forma esquemática qué ocurre cuando sí se gestiona de forma correcta la dirección de retorno.

▶ **7.6** Modifica el código del ejercicio anterior para que la dirección de retorno se apile y desapile de forma adecuada.

Como se ha visto en este apartado, el contenido del registro `LR` formará parte de la información que se tiene que apilar en el bloque de activación de la subrutina en el caso de que se realicen llamadas anidadas.

7.2.2. Variables locales de la subrutina

Para que una subrutina pueda realizar su cometido suele ser necesario utilizar variables propias a la subrutina. Dichas variables reciben el nombre de variables locales puesto que solo existen en el contexto de la subrutina. Por ello, suelen almacenarse bien en registros, bien en una zona de memoria privada de la propia subrutina, en el bloque de activación de la subrutina. Para almacenar las variables locales de una



(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 7.4: Llamadas anidadas a subrutinas apilando las direcciones de retorno. Las distintas subfiguras muestran distintos momentos en la ejecución de un programa en ensamblador en el que desde el programa principal se llama a una subrutina «`S1`», que a su vez llama a una subrutina «`S2`», que llama a una subrutina «`S3`». En cada subfigura se indica el contenido del registro `LR` y el estado de la pila tras ejecutar la última instrucción mostrada. Como se puede ver, gracias al apilado y desapilado de la dirección de retorno, es posible retornar al programa principal correctamente.

subrutina en el bloque de activación se debe: I) reservar espacio en el bloque de activación para almacenar dichas variables, y, antes de finalizar, II) liberar el espacio ocupado por dichas variables.

7.2.3. Almacenamiento de los registros utilizados por la subrutina

Como se ha visto en el apartado anterior, la subrutina puede utilizar registros como variables locales. En este caso, el contenido original de dichos registros se sobrescribirá durante la ejecución de la subrutina. Así que 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. Conviene tener en cuenta, además, que el convenio de paso de parámetros de ARM obliga a la subrutina a preservar todos aquellos registros de propósito general salvo los registros del `r0` al `r3`. Por lo tanto, si una subrutina va a modificar el contenido de algún registro distinto a los anteriores, deberá forzosamente preservar su valor.

La forma en la que se almacena y restaura el contenido de los registros cuyo contenido original debe preservarse es la siguiente: I) la subrutina, antes de modificar el contenido original de dichos registros, los apila en el bloque de activación; y II) una vez finalizada la ejecución de la subrutina, y justo antes del retorno, recupera el contenido original. 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.

7.2.4. 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 7.5. ¿Cómo se accede a los datos contenidos en el bloque de activación? La forma más sencilla y eficiente para acceder a un dato que se encuentra en el bloque de activación es utilizando el modo indirecto con desplazamiento. Como ya se sabe, en dicho modo de direccionamiento, la dirección del operando se obtiene mediante la suma de una dirección base y un desplazamiento. Como dirección base se podría utilizar el contenido del puntero de pila, que apunta a la dirección de memoria más baja del bloque de activación (véase de nuevo la Figura 7.5). El desplazamiento sería entonces la dirección 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, cargándolo sobre el registro `r4`, utilizando la instrucción «`ldr r4, [sp, #8]`».

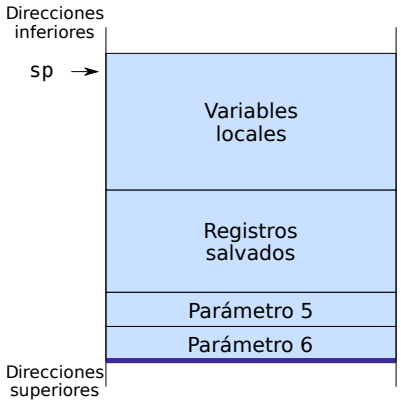


Figura 7.5: Esquema del bloque de activación

7.2.5. 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 7.5).

En el momento en que la subrutina toma el control:

1. Apilar en el bloque de activación aquellos registros que vaya a modificar la subrutina (incluido el registro `LR` en su caso).

2. Reservar memoria en la pila para las variables locales de la subrutina. El tamaño se calcula en función del espacio en bytes que ocupan las variables locales que se vayan a almacenar en el bloque de activación. Conviene tener en cuenta que el espacio reservado deberá estar alineado a 4.

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. Liberar el espacio reservado para las variables locales.

3. 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 7.6 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 se indica qué parte del bloque de activación se ha creado por el programa invocador y cuál por el invocado.

7.2.6. Ejemplo de uso del bloque de activación

Como ejemplo de cómo se utiliza el bloque de activación se propone el siguiente programa en Python3. Dicho programa, dado un vector A de dimensión dim, sustituye cada elemento del vector por el sumatorio de todos los elementos del vector a partir de dicho elemento inclusive, es decir, $a'_i = \sum_{j=i}^{j=dim-1} a_j, \forall i \in [0, dim]$.

```
1 def sumatorios(A, dim):
2       B = [0]*dim
3       for i in range(dim):
4            B[i] = sumatorio(A[i:], dim-i)
5
```

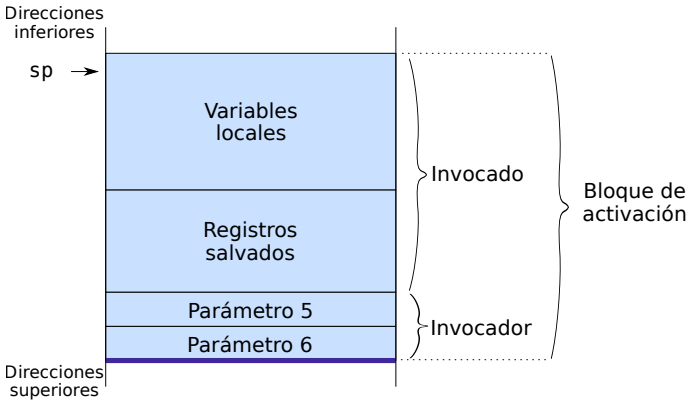


Figura 7.6: Estado de la pila después de una llamada a subrutina

```
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.

07_varlocal.s

```
1               .data
2   A:          .word 7, 6, 5, 4, 3, 2
3   dim:        .word 6
4
5               .text
6   @---------------------------------------
7   @ Programa invocador
8   @---------------------------------------
9   main:       ldr r0, =A
10              ldr r1, =dim
11              ldr r1, [r1]
12              bl sumatorios
13
14  fin:        wfi
15
16  @---------------------------------------
```

```
17 @ Subrutina sumatorios
18 @---------------------------------------
19 sumatorios: @ --- 1 ---
20             push {r4, r5, r6, lr}
21             sub sp, sp, #32
22             add r4, sp, #0
23             str r0, [sp, #24]
24             str r1, [sp, #28]
25             mov r5, r0
26             mov r6, r1
27
28 for1:        cmp r6, #0
29             beq finfor1
30               @ --- 2 ---
31               bl sumatorio
32               str r2, [r4]
33               @ --- 3 ---
34               add r4, r4, #4
35               add r5, r5, #4
36               sub r6, r6, #1
37               mov r0, r5
38               mov r1, r6
39             b for1
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 finfor2:    @ --- 5 ---
54             add sp, sp, #32
55             pop {r4, r5, r6, pc}
56
57 @---------------------------------------
58 @ subrutina sumatorio
59 @---------------------------------------
60 sumatorio:  push {r5, r6, r7, lr}
61             mov r2, #0
62             mov r6, r1
63             mov r5, r0
64 for3:        cmp r6, #0
65             beq finfor3
66               ldr r7, [r5]
67               add r5, r5, #4
68               add r2, r2, r7
```

```
69              sub r6, r6, #1
70            b for3
71  finfor3:  pop {r5, r6, r7, pc}
72
73            .end
```

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

7.7.1 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.

7.7.2 Indica el contenido del registro `LR` una vez ejecutada la instrucción «**`bl`** `sumatorios`».

7.7.3 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.

7.7.4 Indica el contenido del registro `LR` una vez ejecutada la instrucción «**`bl`** `sumatorio`».

7.7.5 Dibuja el bloque de activación de la subrutina «`sumatorio`».

7.7.6 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`»?

7.3. Ejercicios

Ejercicios de nivel medio

▶ **7.8** Dado el código del programa anterior, `07_varlocal.s`, resuelve las siguientes cuestiones:

7.8.1 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.

7.8.2 Localiza el fragmento de código donde se desapila el bloque de activación de la subrutina «`sumatorios`». Explica qué hacen las instrucciones que forman dicho fragmento.

7.8.3 ¿Dónde se recupera el valor que permite retornar al programa principal?

Ejercicios avanzados

▶ **7.9** 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 pasarán como parámetros la matriz, su dimensión y devolverá el máximo buscado.
- «`subr2`» calculará la suma de los elementos de un vector. Se le pasarán 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 pasarán 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`», quien devolverá el máximo buscado. El programa principal deberá almacenar este dato en una dirección etiquetada con «`max`».

Ejercicios adicionales

▶ **7.10** 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.)

▶ **7.11** 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 parte, la subrutina «`subr3`» devolverá un 1 si el contenido de los dos vectores que se le pasan como parámetros son iguales.

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

▶ **7.12** 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)
```

Parte III

Entrada/salida con Arduino

Capítulo 8

Introducción a la entrada/salida

Índice

La entrada/salida es el componente de un ordenador encargado de su interacción con el mundo exterior. Si un ordenador no dispusiera de entrada/salida, con independencia de cuál fuera la potencia de su procesador y de la cantidad de memoria que tuviera, sería totalmente inútil, pues no podría realizar ninguna tarea que debiera manifestarse fuera de sus circuitos electrónicos. Pero por otro lado, el mismo motivo que justifica la existencia de la entrada/salida, la necesidad del ordenador de comunicarse con el mundo exterior, explica también su gran variedad y, de ahí, su problemática. El mundo exterior del ordenador, lejos de ser de la misma naturaleza electrónica y previsible que la de sus circuitos, se caracteriza por su enorme variedad y rápida evolución. La entrada/salida

debe ser capaz de relacionar los dispositivos electrónicos del ordenador con el diverso mundo exterior.

Este capítulo muestra cómo se gestiona esta relación para que la entrada/salida de un computador pueda ser versátil y eficaz, a la vez que manejable.

8.1. 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 imaginemos 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 si es sin cables—. Todos estos elementos, aunque sean de naturaleza diferente y tengan una función completamente distinta, son dispositivos periféricos del ordenador y forman parte de su entrada/salida.

Además, los sistemas de entrada/salida actuales son elevadamente complejos e incluso, por decirlo de alguna manera, jerárquicos. Por ejemplo, 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 y en entornos industriales— 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. Sin embargo, una tarjeta igual —en su mayor parte— conectada a un bus USB es un dispositivo periférico conectado a un dispositivo de entrada/salida —el bus USB—, que es el que está conectado al sistema. El tratamiento de ambas tarjetas de sonido es idéntico en muchos aspectos, pero diferente en otros. Afortunadamente, los sistemas operativos, a través de sus manejadores de dispositivos —*drivers*—, estructurados de forma modular y jerárquica, son capaces de gestionar eficazmente esta complejidad. En este libro obviaremos esta complejidad añadida y 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.

Así pues, la problemática de la entrada/salida reside en que el computador debe ser capaz de interactuar adecuadamente con una inmensa variedad de dispositivos de entrada/salida, algunos elevadamente complejos, y que presentan características muy diversas.

Para poder comprender mejor esta diversidad, comenzaremos viendo cómo caracterizar adecuadamente los dispositivos de entrada/salida, tanto desde un punto de vista cualitativo como temporal.

8.1.1. Características cualitativas de los dispositivos de entrada/salida

Considerando el sentido, siempre referido al ordenador, en que fluyen los datos, podemos observar que unos dispositivos son de salida, como la impresora o el monitor, otros de entrada, como el teclado y el ratón, mientras que algunos son de entrada y de salida, como el disco duro y la conexión de red. El sentido en el que fluyen los datos se denomina **comportamiento** y es una característica propia de cada dispositivo de entrada/salida.

Por otro lado, también podemos percatarnos de que los dos últimos dispositivos del párrafo anterior, el disco duro y la conexión de red, no se comunican con un usuario —un ser humano—, a diferencia de otros que sí lo hacen, como el teclado, el ratón o el monitor. Esto nos permite identificar otra característica cualitativa de los dispositivos de entrada/salida, 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 otros dispositivos podría no ser tan evidente cuál es su interlocutor. Por ejemplo, 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. Por tanto, en este caso el interlocutor es una máquina, no un humano, conclusión errónea a la que se podría llegar si en lugar de pensar en el funcionamiento de la conexión de red, se hubiera pensado en alguno de sus usos en el que intervienen personas como, por ejemplo, el envío y recepción de mensajes de texto.[1] También tienen como interlocutor una máquina 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. Sin embargo, hoy en día los ordenadores están extendidos en todos los campos de la actividad humana, por lo que un computador podría estar 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 midiendo el nivel de concentración de cierto soluto en una reacción química. En estos tres ejemplos, el interlocutor no es un ser humano ni una máquina,

[1] En la mensajería instantánea la conexión de red se limita a interconectar un ordenador con un servidor y a este con otro ordenador. La interacción con los usuarios se realiza a través de otros dispositivos de entrada/salida, p.e.: teclado y pantalla.

sino un sistema o fenómeno natural. Esta clasificación de interlocutores en humano, máquina u otros, 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 si las terminaciones nerviosas fueran las de una persona?

En resumen, las características cualitativas —no medibles— de los dispositivos de entrada/salida son:

- Su comportamiento, que indica si el dispositivo es de entrada, de salida o bidireccional.
- Su interlocutor, que hace referencia al ente —ser humano, máquina u otros— con el que interactúa el dispositivo para intercambiar información entre él y el ordenador.

8.1.2. Características temporales de los dispositivos de entrada/salida

En cuanto a las características temporales de los dispositivos de entrada/salida, la primera que vamos a considerar está relacionada con el tiempo que transcurre entre que se inicia una comunicación y se comienza a recibir la respuesta. Esta característica recibe el nombre de latencia inicial, o simplemente, **latencia**, y se define más formalmente como el tiempo que transcurre desde que se inicia una operación de entrada/salida hasta que los datos comienzan a llegar a su destino. En una operación de entrada, la latencia sería el tiempo transcurrido desde que se inicia la petición de datos al dispositivo hasta que el computador comienza a recibir el primero de ellos. En una operación de salida, el tiempo transcurrido desde que el computador comienza a enviar datos al dispositivo hasta que el dispositivo comienza a recibirlos. Cuando se diseña un sistema de entrada/salida, el valor máximo de su latencia vendrá fijado por el uso que se le vaya a dar. Así por ejemplo, nuestra experiencia al disfrutar de una película no se verá mermada si, desde que ejecutamos el programa de reproducción hasta que aparecen las primeras imágenes transcurren diez o quince segundos, por lo que la latencia de un sistema orientado a esta función podría alcanzar dichos valores. 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 hiciera evidente en la respuesta del sistema, por lo que la latencia de un teclado deberá ser lo suficientemente baja como para no exasperar al usuario.

La otra característica temporal de los dispositivos de entrada/salida que nos incumbe es la **productividad** *(throughput)*, también llamada **tasa de transferencia**, que viene determinada por la cantidad de datos

por unidad de tiempo que pueden intercambiar ordenador y dispositivo. Esta característica es especialmente importante en algunos dispositivos de entrada/salida, como por ejemplo, los discos duros[2], y tiene una gran influencia en la forma en que se gestiona la entrada/salida, puesto que en función de cuál sea la productividad de un dispositivo de entrada/salida, este será tratado de una forma u otra. Aunque todos los periféricos son lentos en comparación con la velocidad del procesador —que puede tratar decenas de miles de millones de bytes por segundo—, la diferencia de velocidades entre los distintos dispositivos de entrada/salida es lo suficientemente alta como para requerir tratamientos bien diferenciados. Así, por ejemplo, un disco duro puede alcanzar varios gigabytes por segundo, mientras que un teclado tan solo puede comunicar unos pocos bytes por segundo. Por tanto, el intercambio de información con un disco duro requerirá utilizar estrategias eficientes de entrada/salida que permitan aprovechar su alta productividad, mientras que la comunicación con un teclado será poco exigente desde este punto de vista.

La productividad puede ser medida de distintas formas, todas ellas válidas, aunque no igualmente significativas. La primera forma de medirla consiste en limitarse a una transacción de entrada/salida determinada. La **productividad de una transacción** de entrada/salida se calculará dividiendo la cantidad total de datos transmitidos en la transacción entre el tiempo transcurrido desde que se inició la transacción hasta que concluyó completamente, incluyendo el tiempo de latencia. La segunda forma se centra en la productividad de un dispositivo, en lugar de en una transacción en particular, y es la **productividad media**, que se calcula como la media de las productividades de un conjunto representativo de transacciones de entrada/salida, en las que se habrá tenido en cuenta tanto los tiempos de transmisión de información como los de latencia. La tercera forma, que es la que se suele dar sobre todo en información comercial —orientada a demostrar correcta o incorrectamente las bondades de cierto producto—, es la **productividad máxima**, que se calcula sin tener en cuenta el tiempo de latencia y considerando el mejor caso posible para el funcionamiento del dispositivo. De aquí se puede concluir que cuando un fabricante indica un valor para la productividad, debería especificar adecuadamente cómo ha realizado su cálculo.

[2] A primera vista podría parecer que un disco duro interno —situado físicamente en el interior del ordenador— no es un dispositivo de entrada/salida, puesto que aparentemente no cumple la definición de que la entrada/salida pone en contacto el ordenador con el mundo exterior. Sin embargo, si se piensa en su funcionamiento, un disco duro es capaz de leer y escribir información de un medio determinado, en algunos casos gracias a piezas mecánicas en movimiento, y de transmitir dicha información al computador. Así que realmente, un disco duro sí es un dispositivo de E/S, siendo su comportamiento bidireccional y su interlocutor, el medio —que en el caso de los discos duros magnéticos, es la polaridad magnética de un conjunto de platos ferromagnéticos—.

Por otra parte, y retomando la comparación anterior entre el visionado de una película y el uso de un teclado, conviene observar que los requisitos de productividad y de latencia varían en función del uso que se vaya dar a un dispositivo, no siendo siempre exigible una latencia baja y una productividad alta. Así, un disco duro, que intervendría en el caso del visionado de una película proporcionándola en un determinado formato, deberá tener una productividad lo suficientemente alta como para transmitir la película al computador sin que se produzcan cortes en su reproducción, pero sería admisible que tuviera una latencia que no fuera especialmente baja. Por otro lado, un teclado, que requiere una latencia baja, de decenas de milisegundos, para funcionar adecuadamente, se ha visto que tiene una productividad bastante baja —que está relacionada con la velocidad máxima a la que podemos teclear—.

A modo de ejemplo de cómo se puede obtener la latencia y la productividad a partir de la descripción temporal de una transacción de entrada/salida, se muestra a continuación el enunciado de un problema de este tipo, así como su solución.

▶ Un procesador realiza una petición a un dispositivo de entrada/salida de un bloque de datos de 448 bytes. Transcurridos 20 ms, el procesador comienza a recibir un primer bloque de 64 bytes, que tarda 2 ms en recibirse completamente. Después de este bloque se van recibiendo otros, con idénticas características, con un lapso de 5 ms desde el final de un bloque hasta el principio del siguiente, hasta completar la recepción de todos los datos. ¿Cuáles serían, para estos datos, la latencia del acceso, la productividad máxima y la productividad de la transacción?

La *latencia*, medida como el tiempo transcurrido desde que se inicia la operación hasta que se comienzan a recibir los datos, viene especificada en el propio enunciado del problema y es de 20 ms.

La *productividad máxima* se puede calcular sabiendo que cada bloque de 64 bytes se recibe en 2 ms, por lo que es de:

$$\frac{64\,\text{bytes}}{2\,\text{ms}} = 32\,000\,\text{B/s, o } 256\,000\,\text{b/s.}$$

La *productividad de la transacción* se calcula dividiendo el total de datos transferidos entre la duración total de la transacción, latencia incluida. Para calcular el tiempo total de la transacción hemos de tener en cuenta que:

- el primer bloque tarda 20 ms en llegar;
- se reciben 7 bloques ($\frac{448}{64} = 7$), que tardan 2 ms cada uno; y

- entre bloque y bloque transcurren 5 ms —seis veces—.

Así pues, el tiempo total es de:

$$20\,\text{ms} + 7 * 2\,\text{ms} + 6 * 5\,\text{ms} = 64\,\text{ms}$$

Como se reciben un total de 448 bytes en 64 ms, la *productividad de la transacción* es por tanto de:

$$\frac{448\,\text{bytes}}{64\,\text{ms}} = 7\,000\,\text{B/s, o } 56\,000\,\text{b/s}$$

Recapitulando, en este apartado se han comentado algunas generalidades de los dispositivos y sistemas de entrada/salida, su problemática, y se han presentado cuatro características que ayudan a su clasificación: su comportamiento, el interlocutor al que se aplican, su latencia y su productividad. En el siguiente apartado se describe la estructura de los sistemas y dispositivos de entrada/salida.

8.2. 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 debe ser compleja e involucrar a diversas tecnologías. Por otra parte, según el elemento del mundo al que esté conectado un dispositivo, su velocidad de funcionamiento puede ser tan lenta como la conmutación de las luces de un semáforo o tan rápida como para enviar 60 imágenes de alta resolución por segundo a un monitor. Pese a esta diversidad extrema, se pueden extraer generalizaciones comunes a todos los dispositivos, que comprenden tanto su estructura física como la forma de relacionarse con el resto del ordenador. En los siguientes subapartados se va a describir la estructura física de los dispositivos de entrada/salida, la arquitectura de sus controladores y el acceso a los registros de los controladores.

8.2.1. Estructura física de los dispositivos de entrada/salida

¿Cuál es la estructura física de los dispositivos de entrada/salida? ¿De qué partes están compuestos? Para responder a estas preguntas

conviene plantearse de nuevo cuál es el uso que se les da. Como ya se ha visto, un dispositivo de entrada/salida sirve para relacionar con el mundo exterior a un computador. Así que, por un lado, estos deberán interactuar con el mundo exterior, lo que pueden hacer de una increíble variedad de formas: generando luz, moviendo una rueda, midiendo la salinidad del agua, registrando los desplazamientos producidos en una palanca... Gran gran parte de esta funcionalidad, aunque no toda, puede conseguirse por medio de tecnologías electrónicas. Por otro lado, el dispositivo de entrada/salida será una parte constituyente del computador, por lo que deberá contar con circuitos electrónicos digitales de la misma tecnología que aquel. Así pues, y pese a la gran diversidad de dispositivos de entrada/salida existentes, es posible describir una estructura general a la que, como siempre con excepciones, se pueden adaptar de una u otra forma estos dispositivos. Esta configuración incluye tres tipos de tecnología, que enumeradas desde el mundo exterior hacia el ordenador serían las siguientes (véase la Figura 8.1):

- Una parte con componentes de una o varias **tecnologías no eléctricas**, que proporcionan las características físicas —mecánicas, ópticas, etcétera— propias del dispositivo.
- Una parte compuesta por **circuitos electrónicos analógicos**, que suele comenzar con uno o varios **transductores** —elementos capaces de transformar energía no eléctrica en energía eléctrica, o viceversa— y que se encarga de adaptar los niveles eléctricos a los requeridos por los transductores, y de los posibles tratamientos electrónicos de las señales —filtrado, amplificación, etcétera—.
- Una parte formada por **circuitos electrónicos digitales**, que comunica el dispositivo de entrada/salida con el ordenador. Esta parte incluye todo lo necesario para que el ordenador pueda gestionar la entrada/salida, tal y como se irá viendo en el resto del libro.

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 su capacidad de difusión de la luz, en el caso de 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 esté conectado el led.

En el caso de un teclado, la parte no eléctrica estaría formada por la mecánica de sus teclas —incluyendo resortes o membranas, según el tipo

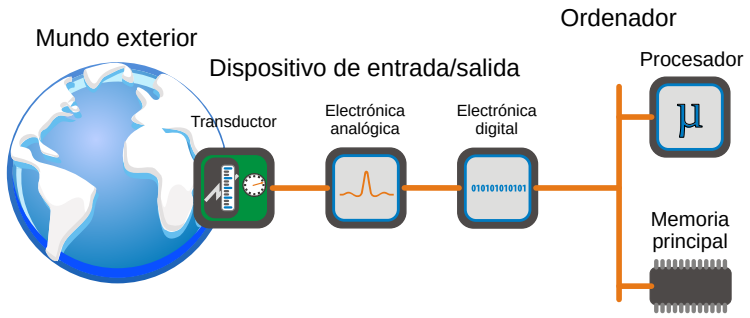


Figura 8.1: Estructura de un dispositivo de entrada/salida

de teclado—. La electrónica analógica estaría formada por los contactos eléctricos que se cierran al pulsar las teclas, que harían las veces de transductores, por las resistencias que adaptan los niveles eléctricos y por los diodos utilizados 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 —hoy en día, USB; antes, PS/2; y aún antes, el bus de teclado de los primeros PC—.

8.2.2. Arquitectura de los controladores de entrada/salida

Como se ha visto en el apartado anterior, los dispositivos de entrada/salida se puede considerar que están formados por tres partes diferenciadas. Aunque las tres son necesarias para su correcto funcionamiento, solo una de ellas, la formada por circuitos digitales, incluye los elementos necesarios para que el computador gestione la entrada/salida. El conjunto de los componentes digitales de un dispositivo de entrada/salida que permiten que el computador gestione su entrada/salida recibe el nombre de **controlador de entrada/salida**. El controlador proporciona al procesador una forma genérica de intercambiar información con el dispositivo, ocultándole sus características específicas, así como la dificultad de tratar con el resto de sus componentes.

La estructura de los controladores de entrada/salida es, a grandes rasgos, común a todos los dispositivos. Gracias a esto, es posible gestionar la entrada/salida de una forma genérica, a pesar de la gran diversidad de dispositivos existente. Además, los controladores de entrada/salida de determinados tipos de dispositivos suelen compartir rasgos comunes más específicos. Así, los controladores de entrada/salida de los discos duros suelen compartir una determinada interfaz de comunicación —por ejemplo, SATA—, independientemente de cuál sea la tecnología

de fabricación y los aspectos específicos de los discos duros de los que formen parte.

Para permitir la comunicación entre el procesador y el dispositivo, el controlador de entrada/salida dispone de un conjunto de espacios de almacenamiento, normalmente **registros** —también conocidos como **puertos**—, a los que el procesador puede acceder. Los registros del controlador de entrada/salida se clasifican según su función en:

Registros de control Se utilizan para configurar los parámetros del dispositivo o indicarle las operaciones de entrada/salida que debe realizar. Son registros en los que el procesador puede escribir, pero no el dispositivo.

Registros de estado Sirven para consultar el estado del dispositivo y el de las operaciones de entrada/salida que este va realizando. Son registros en los que escribe el dispositivo y que pueden ser leídos por el procesador.

Registros de datos Se usan en las operaciones de entrada/salida para intercambiar datos entre el procesador y el dispositivo. En el caso de una operación de salida, el procesador escribirá en estos registros aquellos datos que el periférico deberá trasladar al mundo exterior. En el caso de una operación de entrada, cuando el procesador lea estos registros, el dispositivo deberá proporcionarle, o haber almacenado en ellos previamente, los datos recogidos del mundo exterior.

Veamos cómo se podrían emplear estos registros para el caso de querer utilizar una impresora para imprimir un documento. Aunque en realidad las cosas no sucedan exactamente de la siguiente manera, debido a la estandarización de los formatos de documentos y de la gestión de impresoras, el ejemplo es suficientemente ilustrativo y válido. En primer lugar, el procesador configuraría, escribiendo en los registros de control de la impresora, el tamaño del 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 mediante sucesivas escrituras en los registros de datos de la impresora. Al mismo tiempo, el procesador leería periódicamente los registros de estado de la impresora, ya sea para detectar posibles errores —falta de papel o de tinta, atascos de papel—, ya sea para saber si la impresora acepta nuevos datos —recordemos que el procesador es mucho más rápido que la impresora— o si ha terminado de imprimir la página en curso. Al acabar todas las páginas del documento, el procesador avisaría a la impresora de esta circunstancia escribiendo en uno de sus registros de control, y aquella podría pasar entonces a un modo de espera, con menor consumo.

A pesar de que la clasificación y descripción que se ha realizado de los registros de entrada/salida son correctas desde un punto de vista teórico, es frecuente que los controladores de entrada/salida reales, para simplificar los circuitos y su gestión, mezclen información de control y estado en un mismo registro lógico —es decir, en un único registro desde el punto de vista del procesador— e incluso que algún bit tenga un doble uso, de control y estado, según el momento. Un ejemplo habitual de este último caso es de los conversores analógico-digitales, que disponen de un bit de control en el que el procesador debe escribir para iniciar la conversión —poniéndolo a `1`, por ejemplo— y que el dispositivo cambiará de valor —a `0` en este ejemplo— cuando haya terminado la conversión —típica información de estado— y el resultado esté disponible en un registro de datos.

8.2.3. Acceso a los registros de entrada/salida

Como se ha visto, el procesador se comunica con un dispositivo de entrada/salida leyendo o escribiendo en los registros del controlador del dispositivo. Por lo tanto, para poder comunicarse con el dispositivo, el procesador deberá poder acceder a estos registros. Esto puede conseguirse utilizando cualquiera de las siguientes opciones:

Sistema de entrada/salida mapeado en memoria Se basa en la asignación de una parte del espacio de direcciones de memoria a los registros de entrada/salida. De esta forma, el procesador podrá leer y escribir en los registros de los controladores como si estuviera leyendo o escribiendo en memoria: exactamente igual y ejecutando las mismas instrucciones. Lo que ocurrirá en realidad es que cuando el procesador lea o escriba en determinadas direcciones del mapa de memoria, en lugar de acceder a memoria, lo hará a los registros de un controlador de entrada/salida determinado. Esta estrategia es la utilizada por la arquitectura ARM.

Sistema de entrada/salida independiente —o aislado— Consiste en la ubicación de los registros de entrada/salida en un mapa de direcciones propio, de entrada/salida, independiente del mapa de memoria del sistema. En este caso, el procesador deberá contar con instrucciones específicas de entrada/salida para poder acceder al mapa de direcciones de entrada/salida y, por tanto, a los registros de los controladores —ya que en este caso, las instrucciones de acceso a memoria no podrán utilizarse para dicho fin—. Esta estrategia se utiliza por ejemplo en la arquitectura Intel, que cuenta con instrucciones específicas de entrada/salida llamadas *in* y *out*. Cuando el procesador ejecuta estas instrucciones, activa uno de

```
00:11.4 SATA controller: Intel Corporation C610/X99
        I/O ports at 3078 [size=8]
        I/O ports at 308c [size=4]
        I/O ports at 3070 [size=8]
        I/O ports at 3088 [size=4]
        I/O ports at 3040 [size=32]
        Memory at 91e01000 (32-bit) [size=2K]
```

Figura 8.2: Acceso a la memoria de un contralor SATA. La salida del comando «`lspci -v`» muestra que el acceso a la memoria de un controlador SATA se realiza por medio de las dos estrategias comentadas: se accede a parte de su memoria por medio de un mapa independiente de entrada/salida y a la restante, como parte del espacio de direcciones de memoria del sistema

sus pines para indicar que la dirección especificada en el bus de direcciones pertenece al mapa de entrada/salida.

Las dos opciones anteriores no son excluyentes. Un computador podría utilizar ambas estrategias para acceder al espacio de memoria de los controladores de entrada/salida, siempre y cuando su procesador dispusiera de instrucciones específicas para acceder a un mapa de direcciones independiente de entrada/salida. Este caso se da en los computadores personales basados en la arquitectura Intel. A modo de ejemplo, la Figura 8.2 muestra un controlador SATA, instalado en un PC, que expone parte de su memoria en un mapa de direcciones independiente —las líneas que empiezan por `I/O ports at...`— y otra parte de su memoria mapeada en el mapa de direcciones de la memoria del sistema —la línea que empieza por `Memory at...`—.

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 último bloque recibe el nombre de controlador del dispositivo y facilita que el procesador se comunique con aquel 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. El procesador puede acceder a estos registros como parte del mapa de memoria del sistema, mediante instrucciones de acceso a memoria, o en un mapa específico de entrada/salida, situación que solo puede darse si el procesa-

dor incorpora instrucciones especiales. En los siguientes capítulos se verá cómo usar estos registros para interactuar con diferentes dispositivos.

8.3. Ejercicios

▶ **8.1** Indica las características de los dispositivos de la siguiente lista. Para la latencia puedes dar un valor cualitativo como alta, media o baja; para la productividad basta con una estimación razonable. Busca información sobre aquellos que no conozcas.

8.1.1 Una pantalla táctil de 1024x768 píxeles.

8.1.2 Un mando *(gamepad)* de videojuegos.

8.1.3 Un sensor de posición de 3 ejes.

8.1.4 Un controlador de bus SMB.

8.1.5 Una tableta digitalizadora.

▶ **8.2** Busca información acerca de los dispositivos que aparecen en la siguiente lista e indica para cada uno de ellos la estructura y la función de su parte electrónica analógica y el transductor que utiliza.

8.2.1 El bloque dispensador de plástico de una impresora 3D.

8.2.2 Una impresora de chorro de tinta.

8.2.3 Un lector de huellas dactilares.

8.2.4 Un monitor TFT.

8.2.5 Un mono para captura de movimiento.

▶ **8.3** En el Apéndice A se encuentra la información técnica de algunos dispositivos del microcontrolador ATSAM3X8E de la tarjeta Arduino Due. Consúltalo e indica, para los siguientes dispositivos, los bits o grupos de bits de control, estado o datos de sus registros.

8.3.1 Temporizador *(System Timer)*.

8.3.2 Reloj en tiempo real *(RTC)*.

8.3.3 Temporizador de tiempo real *(RTT)*.

▶ **8.4** El conversor digital analógico MCP4822 convierte un valor digital de 12 bits en un voltaje analógico entre 0 y 2 048 mV o 4 096 mV, según se haya configurado. Los valores a convertir se le envían a través de un bus SPI con una velocidad máxima de 20 Mbps, mediante dos envíos de 8 bits, de los que solo se utilizan los 12 menos significativos. Indica la productividad máxima, referida a los datos útiles, de este dispositivo.

▶ **8.5** Se ha diseñado un sistema de vigilancia de bajo consumo eléctrico para enviar imágenes de 1024x768 píxeles y 24 bpp. Dicho sistema se encuentra normalmente en modo de bajo consumo, pasando cada 20 segundos a modo activo. En este modo, si hay alguna petición pendiente, adquiere una imagen, lo que le cuesta 25 ms, y la envía a razón de 200 kbps. Indica las latencias máxima, promedio y mínima del sistema, así como su productividad máxima y promedio. Considera despreciable el tiempo de envío de las peticiones.

Capítulo 9

Dispositivos de entrada/salida y el entorno Arduino

Índice

En el capítulo anterior se presentó la problemática y las características de la entrada/salida (E/S) en los ordenadores, así como la estructura tanto física como lógica de los dispositivos de E/S. Los siguientes pasos en el estudio de los sistemas de E/S son: i) conocer con cierto detalle algunos de estos dispositivos de E/S, y ii) realizar programas que interactúen con ellos. En este capítulo se describen de forma genérica los dos grupos de dispositivos de E/S más comúnmente usados: la E/S de propósito general y los dispositivos de gestión del tiempo. Todos los sistemas incluyen y utilizan estos tipos de dispositivos, pero son especialmente relevantes en los sistemas empotrados y en los microcontroladores. La última parte de este capítulo introduce el entorno de desarrollo que se va a utilizar para realizar programas de E/S: la tarjeta Arduino Due,

su entorno de desarrollo y una sencilla tarjeta de expansión que incorpora un led RGB y un pulsador. Finalmente, se proponen una serie de ejercicios que permiten interactuar con los dispositivos de E/S descritos.

9.1. Entrada/salida de propósito general (GPIO)

Una E/S de propósito general (GPIO, de *General Purpose Input/Output*) es un pin genérico en un circuito integrado o placa de computador cuyo comportamiento —incluyendo si se trata de un pin de entrada o de salida—, se puede controlar desde un programa.

Una E/S de propósito general es un pin de E/S controlable por programa.

Es la forma más sencilla de E/S. Tan solo requiere añadir el hardware necesario para convertir los valores físicos presentes en determinados pines a valores lógicos, y viceversa, y que una parte del mapa de memoria o de la E/S se asigne a esos pines. Así, si el valor eléctrico presente en un pin determinado se ve traducido por la circuitería interna en un `0` o `1` lógico que se puede leer en una dirección del sistema, entonces un programa podría detectar cambios en dicho pin simplemente consultando esa dirección. Por ejemplo, para consultar si un pulsador conectado a determinado pin está libre u oprimido. De forma análoga, si la escritura de un `0` o `1` lógico por parte de un programa en alguna de esas direcciones se reflejara en cierta tensión eléctrica en un determinado pin (normalmente 0 V —nivel bajo—, si se escribe un `0`, o 3,3 V o 5 V —nivel alto—, si se escribe un `1`), un programa podría desactivar o activar algún dispositivo externo, por ejemplo, un led, escribiendo un `0` o un `1` en la dirección adecuada. El siguiente ejemplo utiliza entradas/salidas de propósito general para consultar el estado de un pulsador y en el caso de que esté oprimido, encender un led:

09_entrada_salida.s

```
1         ldr     r0, [r7, #PULSADOR]     @ Leemos el pulsador
2         cmp     r0, #1                  @ Si no está oprimido,
3         bne     sigue                   @  seguimos, si no,
4         mov     r0, #1                  @ Escribimos 1 para
5         str     r0, [r7, #LED]          @  encender el LED
6 sigue:  ...
```

El fragmento de código anterior consulta el estado del pulsador leyendo de la dirección «`r7 + PULSADOR`» y si está presionado, es decir, si lee un `1`, enciende el led escribiendo un `1` en la dirección «`r7 + led`». El apartado siguiente profundiza en la descripción de la E/S de propósito general y describe con más detalle sus características en los sistemas reales.

9.1.1. La GPIO como E/S de los sistemas

La GPIO 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. Algunos fabricantes de portátiles utilizan una GPIO para encender el amplificador para los altavoces internos y el conector de auriculares. Sin embargo, es en los microcontroladores, sistemas completos en un chip, donde la GPIO tiene más importancia y muestra su mayor complejidad y potencia. Vamos a analizar a continuación cómo se utiliza la GPIO en estos sistemas y sus aspectos e implicaciones.

Aspectos lógicos (programación) de la GPIO

El ejemplo visto previamente, consulta y modifica la información de un pin de entrada y uno de salida, respectivamente, utilizando instrucciones de carga y almacenamiento a nivel de palabra sobre sendas direcciones de memoria, «`r7 + #PULSADOR`» y «`r7 + #LED`», respectivamente. Recordando que en la arquitectura ARM cada palabra está formada por 32 bits, y sabiendo que cada pin solo puede tomar uno de entre dos valores, para consultar o modificar un único bit, se han consultado o modificado 32 bits de dos direcciones distintas. Esta forma de proceder presenta dos desventajas. Por un lado, desperdicia recursos, puesto que cada pin de E/S consume una dirección de memoria. Por otro, impide que varios pines sean consultados o modificados de forma simultánea, ya que solo se puede leer, o escribir, una palabra (pin) cada vez. Así pues, en realidad, la mayor parte de los sistemas en lugar de asociar una dirección de memoria a cada pin individual, lo que hacen es agrupar diversos pines de E/S en una misma palabra. Estas palabras, cada una asociada a una dirección de memoria distinta y que agrupan a varios pines de E/S, reciben el nombre de **puertos de E/S**. Así por ejemplo, el microcontrolador ATSAM3X8E dispone de cuatro puertos de E/S llamados PIOA, PIOB, PIOC y PIOD (PIO por *Parallel Input Output*). Para referirse a un bit concreto de un puerto se suele utilizar la nomenclatura «NombreDelPuertoNúmeroBit». Así, PIOB12 haría referencia al bit 12 del puerto PIOB. Este bit, tal y como se puede ver en la página 18 de [4], se corresponde físicamente con el pin 86 de uno de los posibles encapsulados del microcontrolador, el LQFP. Esta última correspondencia es necesario conocerla para poder diseñar hardware en el que se use este microcontrolador y han debido tenerla en cuenta, p.e., para diseñar la tarjeta Arduino DUE.

Cuando los pines de E/S están agrupados en puertos, para actuar —modificar o comprobar su valor— sobre bits individuales o sobre conjuntos de bits, será necesario utilizar máscaras y operaciones lógicas para

ignorar, cuando se lea, o no modificar, cuando se escriba, los otros pines representados en el mismo puerto. Por tanto, una versión más verosímil del ejemplo propuesto, suponiendo que el pulsador se encuentre en el bit 27 del puerto PIOB y el led, en el bit 12, sería:

09_acceso_es.s

```
1         ldr     r7, =PIOB       @ Dirección del puerto PIOB
2         ldr     r6, =0x08000000 @ Máscara para el bit 27
3         ldr     r0, [r7]        @ Leemos el puerto
4         ands    r0, r6          @ y verificamos el bit 27
5         beq     sigue           @ Seguimos si está a 0, si no,
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 12
9         str     r0, [r7]        @ y lo escribimos en el puerto
10    sigue:  ...
```

En esta nueva versión, en primer lugar se accede a la dirección del puerto PIOB 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 27— está a `1`. La operación AND permite, en este caso, quedarse con el valor de un único bit, el 27:

	`r6` =	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$\cdots$	$\cdots$	$\cdots$	b_3	b_2	b_1 b_0
AND	`r0` =	0	0	0	0	1	0	0	0	$\cdots$	$\cdots$	$\cdots$	0	0	0 0
	`r0` ←	0	0	0	0	b_{27}	0	0	0	$\cdots$	$\cdots$	$\cdots$	0	0	0 0

Si el bit 27 está a 1 —esto es, cuando el resultado de la AND no sea cero—, se lee de nuevo el puerto PIOB y mediante una operación OR y la máscara correspondiente, se pone el bit 12 a 1 para encender el led. La operación OR permite, en este caso, poner a `1` el bit 12 sin modificar el resto de bits:

	`r6` =	b_{31}	b_{30}	b_{29}	b_{28}	$\cdots$	$\cdots$	b_{15}	b_{14}	b_{13}	b_{12}	$\cdots$	b_3	b_2	b_1 b_0
OR	`r0` =	0	0	0	0	$\cdots$	$\cdots$	0	0	0	1	$\cdots$	0	0	0 0
	`r0` ←	b_{31}	b_{30}	b_{29}	b_{28}	$\cdots$	$\cdots$	b_{15}	b_{14}	b_{13}	1	$\cdots$	b_3	b_2	b_1 b_0

Este nuevo ejemplo, obviando que aún no es del todo cierto para el ATSAM3X8E, como se verá más adelante, es válido para mostrar la gestión por programa de la GPIO. Sin embargo, llegados a este punto debería surgirnos la duda de cómo es que un pin de E/S de propósito general puede actuar a la vez como entrada y como salida. No olvidemos que los pines se relacionan realmente con el exterior mediante magnitudes eléctricas y que el comportamiento eléctrico de un pin que funciona como entrada es totalmente distinto al de otro que se utiliza como salida, como se concretará más adelante. La solución consiste en que en realidad, antes de utilizar un pin como entrada o salida, como

en el ejemplo anterior, es necesario indicar el **sentido del pin**, es decir, si se va a utilizar como entrada o como salida. Para ello, asociado a la dirección del puerto en la que se leen o escriben los valores presentes en los pines de E/S, y que hemos llamado PIOB en el ejemplo, habrá al menos otra que corresponderá a un registro de control de la GPIO en la que se indique el sentido de sus 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 esta no depende del programa, ni en valor, ni en tiempo. Sin embargo, cuando el pin se usa como salida, es el procesador ejecutando las 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 la 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 `0` o `1` 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 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 el sentido de los pines, introducido en el párrafo anterior; otra para el registro de datos de salida, y otra para leer directamente los pines a través del registro de datos de entrada.

Aspectos físicos (electrónica) de la GPIO

En este apartado se van a estudiar brevemente las características físicas de los pines de E/S de propósito general. Como hemos dicho, la forma en la que un pin de E/S interactúa con el exterior es típicamente mediante su tensión eléctrica. En el caso de estar configurado como salida, cuando escribamos un `0` lógico, se tendrá un cierto voltaje en el pin correspondiente, y cuando escribamos un `1`, otro distinto. En el caso de estar configurado como entrada, al leer el pin obtendremos un `0` o un `1`, según la tensión fijada en él por la circuitería externa.

A modo de ejemplo, la Figura 9.1 muestra la estructura interna de un pin de E/S de un microcontrolador de la familia Atmel AVR. A la

izquierda está el pin propiamente dicho, al que se conectará el dispositivo externo, y a la derecha, el bus de datos del microcontrolador. El biestable D[1] superior, «`DDxn`», almacena el sentido del pin (esto es, si es el pin es de entrada o de salida). El biestable D central, «`PORTxn`», que se utilizará en el caso de que el pin se haya configurado como salida, almacena el valor lógico de la salida. Este biestable también tiene otra función cuando el pin actúa como entrada, pero se verá más adelante. Por último, el biestable D inferior, «`PINxn`», que se utilizará en el caso de que el pin se haya configurado como entrada, servirá para leer el valor lógico asociado a la tensión presente en el pin.

En cuanto a las características eléctricas de los pines, las especificaciones eléctricas de los circuitos proporcionan típicamente los siguientes parámetros para cuando actúen como salidas: `VOHMIN` y `VOLMAX`. El primero de ellos, `VOHMIN`, indica la tensión mínima que habrá en el pin cuando se escriba en él un `1` lógico. No se suele proporcionar la tensión máxima, ya que se sobrentiende que es la de la alimentación del circuito. El fabricante, al especificar una `VOHMIN`, garantiza que cuando se escriba un `1` lógico, la tensión en el pin estará comprendida entre `VOHMIN` y la de alimentación. Un valor razonable de `VOHMIN` para una alimentación de 5 V sería 4,2 V. De manera análoga, `VOLMAX` indica la tensión máxima que habrá en el pin cuando se escriba un `0` lógico. En este caso, la tensión mínima se sobreentiende que es 0 voltios y el rango garantizado está entre 0 V y `VOLMAX`. Un valor típico de `VOLMAX` es 0,8 V. Como se habrá podido observar, el nombre de estos parámetros sigue la siguiente nomenclatura: la primera letra, `V`, indica voltaje; la siguiente, `O`, salida *(output)*; la tercera letra, `H` o `L`, indica si se refiera al nivel alto *(high)* o al bajo *(low)*, respectivamente; por último, `MAX` o `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 de que una salida se conecte directamente a masa —es decir, a 0 V—. La especificación garantiza, según el ejemplo, una tensión mínima de 4,2 V, pero sabemos que el pin está a un potencial de 0 V 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, de-

[1] Un biestable D es un circuito secuencial capaz de almacenar un bit. Tiene una entrada D, que le indica el valor que deberá almacenar y una entrada de reloj, CLK, que marca en qué momento debe actualizar su valor con el indicado en su entrada D. La salida Q muestra el valor almacenado —y la salida $\bar{Q}$, ese mismo valor negado—. También puede tener una entrada asíncrona $RESET$ para fijar su valor a 0 sin tener que esperar a la señal de reloj. Se suele representar por medio de un rectángulo en el que se pueden identificar sus entradas y salidas características.

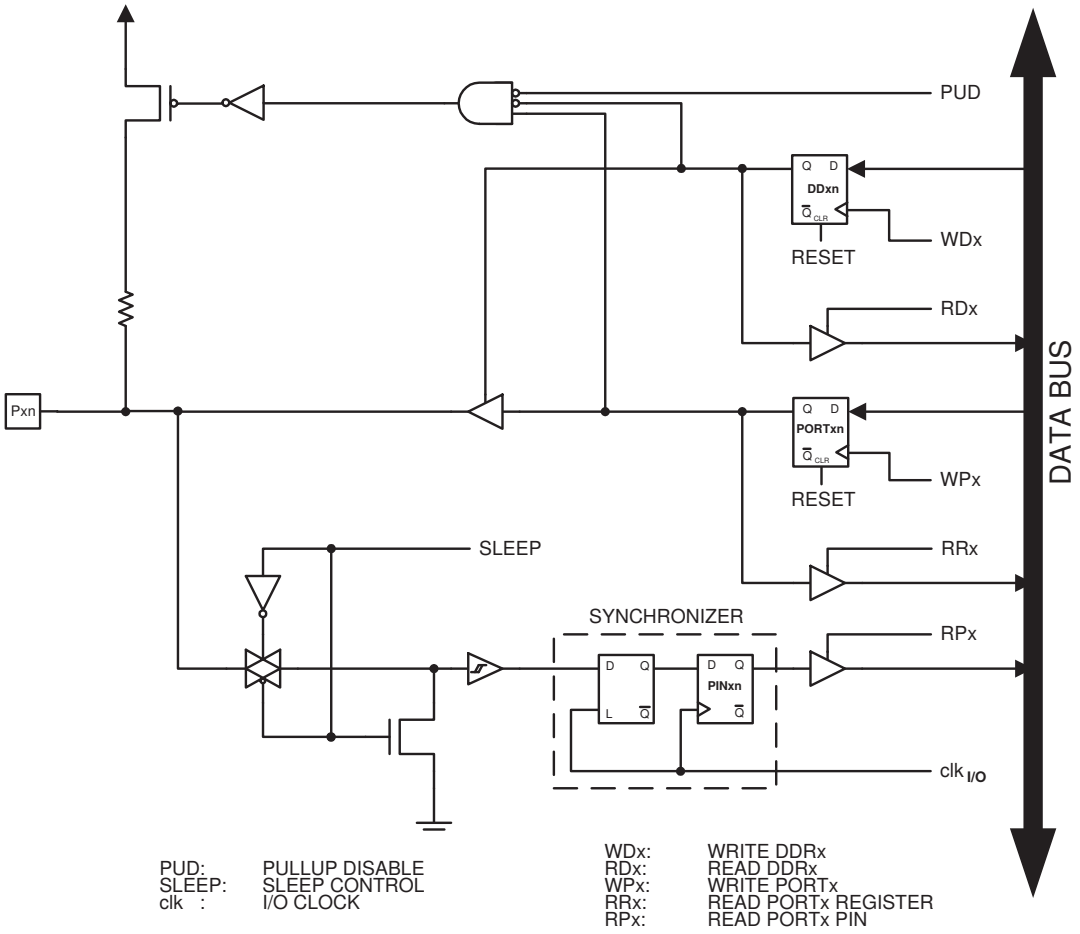


Figura 9.1: Estructura interna de un pin de E/S de un microcontrolador de la familia Atmel AVR *(fuente: [3])*

bería ser muy elevada para poder satisfacer ambas tensiones. Sabemos que esto no es posible, puesto que un pin normal de un circuito integrado puede suministrar como mucho algunos centenares de miliamperios —que además solo se consiguen 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 sea la misma en ambas direcciones, suelen ser del orden de pocas decenas de miliamperios —normalmente algo más de 20 mA, 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 la de detectar niveles de tensión del exterior, con la menor influencia en ellos que sea posible. De este modo, para un circuito externo una entrada aparece como si se tratara de una resistencia muy elevada, lo que se llama una **alta impedancia**. Dado que en realidad se trata de un circuito activo y no de una resistencia, el valor que se especifica en las características eléctricas es la intensidad máxima que circulará 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 segunda `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 que sean 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 estará limitada, independientemente de cuál sea el 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.

Ejemplos de circuitos conectados a la GPIO: led y pulsador

En este apartado se realiza un pequeño estudio de los circuitos eléctricos relacionados con los dispositivos de nuestro ejemplo, el led y el

pulsador. Comenzamos, como viene siendo habitual, por el circuito de salida. En la Figura 9.2 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, que en uno de color rojo está en torno a los 1,2 V. Por otra parte, a partir de 10 mA el brillo del diodo es adecuado, pudiendo conducir sin deteriorarse hasta 30 mA o más. Supongamos en nuestro microcontrolador los valores indicados previamente para `VOHMIN` y `VOLMAX`, y una corriente de salida superior a los 20 mA. Si queremos garantizar 10 mA 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 4,2 V. Mediante la ley de Ohm tenemos:

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

Una vez fijada esta resistencia, el brillo máximo del led se daría cuando la tensión de salida fuera de 5 V, y la corriente, por tanto, de 12,7 mA, aproximadamente.

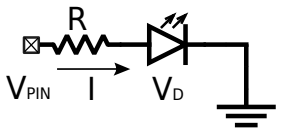


Figura 9.2: Conexión de un led a un pin de E/S de un microcontrolador

Veamos ahora cómo conectar un pulsador a una entrada del circuito. 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 que lo haga, hay que conectarlo en un circuito y, en nuestro caso, en uno que genere las tensiones adecuadas. Para seguir estrictamente los ejemplos vistos hasta ahora, podríamos plantearnos que el pulsador hiciera contacto entre los 5 V de la alimentación y el pin. De este modo, al pulsarlo, el pin se conectaría a la alimentación y se leería un `1`, tal y como se espera en el ejemplo de código fuente mostrado anteriormente. Sin embargo, si el pulsador 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, a masa —0 V— en este caso, a través de una resistencia para limitar la corriente generada cuando se oprima el pulsador. 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 de acuerdo con el ejemplo de código fuente, al pulsar leeríamos un `1` lógico y, mientras no se pulse, un `0` lógico.

Sin embargo, la configuración más habitual es la contraria: conectar el pulsador entre el pin y masa, y conectar el pin a la alimentación a través de una resistencia (véase el esquemático mostrado en la Figura 9.3), pese a que al hacerlo así, los niveles lógicos se invierten y se lee un `0` lógico al pulsarlo y un `1` lógico cuando no. El que esta sea la configuración más habitual tiene implicaciones en el diseño de los microcontroladores y en la gestión de la GPIO. Por ejemplo, es tan habitual el uso de resistencias conectadas a la alimentación —llamadas **resistencias de pull-up** o simplemente **pull-ups**—, que muchos circuitos la integran en la circuitería del pin (como se puede ver por ejemplo en la Figura 9.1), por lo que 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 de la GPIO en la mayor parte de los casos. Además, la configuración seleccionada deberá almacenarse en la lógica del pin de E/S (p.e., en el esquema mostrado en la Figura 9.1, el biestable D central se encarga de mantener la configuración del *pull-up* cuando el pin está configurado como entrada).

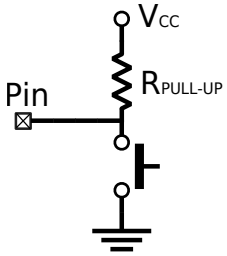


Figura 9.3: Conexión de un pulsador a un pin de E/S de un microcontrolador

9.1.2. Interrupciones asociadas a la GPIO

Como se verá con más detalle en el Capítulo 10, existen dos formas de sincronizar los dispositivos de E/S. Una de ellas consiste en preguntar periódicamente al dispositivo si requiere atención. Recibe el nombre de *sincronización mediante encuesta* o *consulta de estado*, no requiere hardware adicional y es la que se ha utilizado en los ejemplos anteriores cuando se comprobaba el estado del pulsador en un momento dado. La segunda forma de sincronización consiste en que el propio dispositivo avise al procesador de que requiere atención. Recibe el nombre de *sincronización mediante interrupciones*, requiere hardware específico y en la mayor parte de los casos, es más eficaz que la sincronización por consulta de estado. Los sistemas avanzados de GPIO incorporan la posibilidad de avisar al procesador de ciertos cambios en sus pines mediante interrupciones.

Se suele habilitar la generación de interrupciones de la GPIO para conseguir alguna de las siguientes funcionalidades:

- Para utilizar algunos pines de la GPIO como líneas de interrupción. Bien para señalar un cambio relativo al circuito conectado al pin, como oprimir un pulsador, bien para que un circuito externo sea capaz de generar interrupciones.
- Para sacar al procesador del modo de bajo consumo ante un cambio en determinadas entradas.

El uso de interrupciones asociadas a la GPIO requiere añadir nuevos registros de control y estado. Los primeros para configurar qué pines van a generar interrupciones y en qué circunstancias; los segundos, para almacenar los indicadores que permitan conocer qué interrupciones se han generado.

9.1.3. 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 5 V y 3,3 V 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 de 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 directamente a la GPIO y cuyos registros de configuración suelen considerarse como parte de aquella.

9.2. 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 E/S. Anotar correctamente la fecha y hora de modificación de un archivo, arrancar automáticamente tareas con cierta periodicidad, determinar si una tecla se ha pulsado durante más de medio segundo, son actividades comunes en los ordenadores que requieren de una correcta medida y gestión del tiempo. En estos ejemplos se pueden 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 E/S de los ordenadores.

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

Un sistema de tiempo real se define como aquel 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 E/S que facilitan la 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 procesos 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, el reloj en tiempo real (RTC), que gestiona el tiempo en formato humano —es decir, en forma de fecha y hora—, con lo que libera al software del sistema de esta tarea y además, es capaz de actualizar esta información incluso cuando el sistema esté apagado, en el caso de disponer de alimentación propia —generalmente una pila—-. 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 E/S.

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 viene dado por el periodo, $T = 1/f$, y que el tiempo que transcurre hasta que se desborde el contador es $2^n \cdot T$, siendo n el número de bits del registro

temporizador. Para una frecuencia de 10 kHz y un contador de 32 bits, el tiempo mínimo sería 100 μs y transcurrirían unos 429 496 s —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, da el tiempo que ha transcurrido entre ambos eventos. No obstante, hay que tener en cuenta que este procedimiento daría un valor incorrecto si entre las dos lecturas se hubiera producido un desbordamiento del temporizador. Por ello, el temporizador cada vez que se desborda, activa una señal de estado que generalmente puede causar una interrupción. El sistema puede entonces tener en cuenta esta circunstancia. Por ejemplo, para extender el tamaño del contador utilizando una variable —cada vez que se genere la interrupción se incrementaría la variable—.

Por otro lado, la interrupción generada por el temporizador al desbordarse se puede utilizar como una interrupción periódica en la gestión del sistema. Por ejemplo, en los sistemas multitarea se puede utilizar dicha interrupción para cambiar la tarea activa. En este último caso, para tener un control más fino de la periodicidad de la interrupción es habitual poder recargar el contador con un valor distinto de `0`. Por ello suele ser posible escribir sobre el registro que hace de contador.

Además de este funcionamiento genérico del temporizado, 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 milisegundos, 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 accesos 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`, que es 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 estos errores.

En el Apéndice A se describen las particularidades del temporizador en tiempo real RTT *(Real-time Timer)* del ATSAM3X8E.

Otros dispositivos temporales

Si solo se dispone de un dispositivo temporizador, se deberá elegir entre tener una medida de tiempos de larga duración —hasta de varios años en muchos casos— para gestionar correctamente el tiempo a lo largo de la vida del sistema, o por el contrario, tener una buena resolución —pocos milisegundos o menos— para medir tiempos con precisión. Por eso es común que los sistemas dispongan de varios temporizadores que, compartiendo o no la misma base de tiempos, puedan configurar sus periodos mediante *prescalers* individuales. Estos sistemas, con varios temporizadores, suelen añadir además otras características que permiten una gestión mucho más completa del tiempo. A continuación se analizan las extensiones más comunes del temporizador básico.

Algunos temporizadores pueden utilizar una entrada externa —un pin del microcontrolador, normalmente— como base de tiempos. Esto permite utilizar una fuente de tiempo con las características que se deseen o hacer que el temporizador sea en realidad un contador de eventos, en lugar de tiempos —ya que no es necesario que la señal en esa entrada cambie de forma periódica—.

También es habitual contar con registros de comparación, con el mismo número de bits que el del temporizador, que desencadenen un evento cuando el valor del temporizador coincida con el de alguno de aquellos. Estos eventos pueden ser internos, normalmente la generación de alguna interrupción, o externos, cambiando el nivel eléctrico de algún pin y pudiendo generar así salidas dependientes del tiempo.

Otra opción común es la de disponer de registros de copia que guarden el valor del temporizador cuando se produzca 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.

Por último, también está muy extendido el poder modular el ancho de los pulsos de una salida digital para generar así una señal analógica. Esta técnica, también llamada PWM, por *Pulse Width Modulation*, consiste en el caso que nos ocupa, en generar una señal periódica que alterne entre un nivel alto y uno bajo en cada periodo, pudiendo variar el porcentaje de tiempo que se está en cada nivel. Como el porcentaje de tiempo que la señal esté en nivel alto, especificado de forma digital, variará la cantidad de potencia entregada, que es un valor analógico, al atacar a un dispositivo que se comporte como un filtro paso-bajo, lo que

La modulación por ancho de pulsos de una señal o fuente de energía es una técnica en la que se modifica el ciclo de trabajo de una señal periódica (una senoidal o una cuadrada, por ejemplo), ya sea para transmitir información a través de un canal de comunicaciones o para controlar la cantidad de energía que se envía a una carga.

es muy frecuente en dispositivos reales —bombillas, ledes, calefactores, motores, etcétera—, se consigue una conversión digital-analógica muy efectiva, basada en el tiempo. Un circuito capaz de generar una señal PWM, requerirá de un temporizador, para marcar el periodo de la señal PWM, y de un registro, que marcará el porcentaje de tiempo que la señal deberá estar en el nivel alto —o en el bajo—.

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. Por este motivo, el RTC suele llevar aparejada una batería o un condensador que le proporcione la energía necesaria para seguir funcionando cuando se interrumpa la alimentación del computador.

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 apagado-encendido 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 para mantener la hora correcta.
- La presencia de la batería permite que el reloj siga en funcionamiento cuando el computador se apaga.

9.3. 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, Italia, diseñaron una plataforma de hardware libre basada en un

microcontrolador y un entorno de desarrollo con el objetivo de facilitar la realización de proyectos de electrónica. Banzi y su grupo se reunían habitualmente en el bar «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 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, pero que respetan tanto las dimensiones físicas de los conectores de ampliación como su cometido. Además, el entorno de desarrollo proporciona una capa de abstracción, un conjunto de funciones, que puede ser empleada en cualquier modelo de tarjeta de Arduino.

En este libro se propone usar la tarjeta Arduino Due —véase la Figura 9.4— que, con respecto a la tarjeta Arduino Uno original —mostrada en la Figura 9.5—, entre otras diferencias, dispone de un microcontrolador ATSAM3X8E, más potente que el ATmega328 de aquella, y un mayor número de entradas/salidas. Esta tarjeta puede alimentarse mediante una fuente de alimentación externa conectada a su entrada de alimentación o por medio de uno cualquiera de sus dos puertos USB. Para programar la tarjeta Arduino, esta se deberá conectar al computador utilizando el puerto etiquetado como *«Programming Port»*, el que está al lado de la entrada de alimentación —en la Figura 9.4, el de arriba—. El otro puerto USB, llamado *«Native USB»*, permite que la tarjeta Arduino Due pueda programarse para actuar como un dispositivo USB —p.e., un teclado o un ratón—.

¿Cómo conectar la tarjeta Arduino Due al computador para programarla?

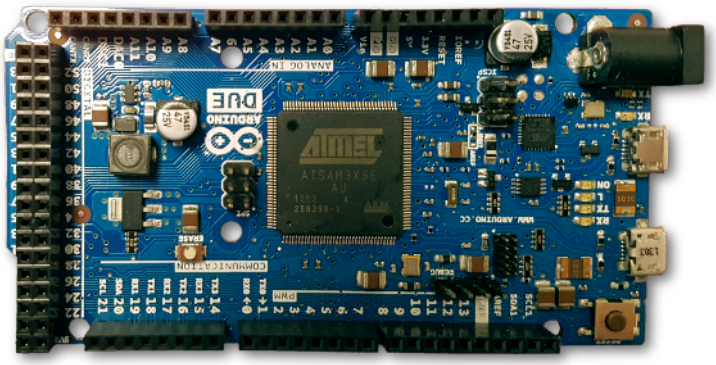


Figura 9.4: Tarjeta Arduino Due

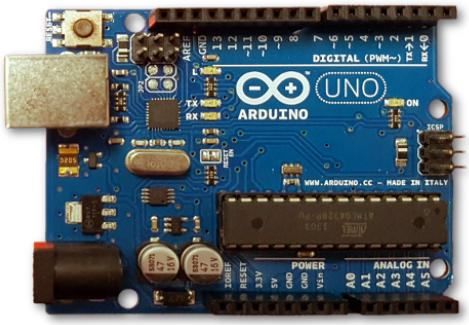


Figura 9.5: Tarjeta Arduino Uno

Conviene tener en cuenta que en la segunda parte de este libro, *Arquitectura ARM con QtARMSim*, se ha utilizado el conjunto de instrucciones Thumb correspondiente a la versión Cortex-M0 de la arquitectura ARM. Sin embargo, el microcontrolador ATSAM3X8E de la tarjeta Arduino Due implementa la versión Cortex-M3 de ARM, que utiliza un conjunto de instrucciones mayor, llamado Thumb II. Aunque todas las instrucciones Thumb están incluidas en Thumb II, existe una diferencia crítica en el lenguaje ensamblador que puede dar lugar a errores al programar en ensamblador la tarjeta Arduino Due y que se señala a continuación. Las instrucciones aritméticas y lógicas del conjunto Thumb siempre modifican los indicadores de estado. Sin embargo, el juego de instrucciones Thumb II proporciona dos variantes de las instrucciones aritméticas y lógicas: una de ellas sí que modifica los indicadores de estado y la otra no. Esta circunstancia se expresa en lenguaje ensamblador añadiendo una `s` al nombre de la instrucción cuando se quiera que dicha instrucción los modifique, de manera que se tiene que:

```
1 ands r0, r1, r2        @ Sí modifica los indicadores
2 and  r0, r1, r2        @ No los modifica
```

Si bien esta característica añade potencia al conjunto de instrucciones, es fácil confundirse cuando se está acostumbrado a programar con instrucciones Thumb, ya que en este juego de instrucciones, como ya se ha comentado, todas las instrucciones actualizan los indicadores de estado.

En cualquier caso, puesto que el juego de instrucciones Thumb II es más potente que el visto hasta ahora, es recomendable consultar el manual *Cortex-M3 Instruction Set* [19] para conocer todas las posibilidades de este conjunto de instrucciones.

Además de la tarjeta Arduino Due, se propone utilizar una tarjeta de E/S —mostrada en la Figura 9.6—, que incorpora un pulsador y un

led RGB, para poder realizar ejercicios de E/S con dichos dispositivos. De esta forma se dispondrá de un dispositivo de entrada, el pulsador, y tres de salida, los ledes rojo, verde y azul del led RGB. La tarjeta de E/S se debe insertar en la tarjeta Arduino Due tal y como se muestra en la Figura 9.7. Por un lado, los pines de la tarjeta de E/S se deben conectar a los siguientes pines de la tarjeta Arduino Due: `GND`, `13`, `8`, `7` y `6`. Por otro, el cable de la tarjeta de E/S se debe conectar al pin `3.3V` de la Arduino Due. En cuanto a la configuración de dichos dispositivos, la Figura 9.8 muestra el esquemático de la tarjeta de E/S. Como se puede ver en esa figura, el led RGB es del tipo ánodo común, por lo que para encender cualquiera de sus led, será necesario escribir un `0` en el pin al que esté conectado y para apagarlo será necesario escribir un `1`. Igualmente, también se puede observar que el pulsador está conectado entre el pin 13 y masa, por lo que se puede inferir que será necesario activar el `pull-up` de la salida 13 y que cuando el pulsador esté oprimido se leerá un `0` y en el caso contrario, un `1`.

¿Cómo conectar la tarjeta de E/S a la placa Arduino Due?

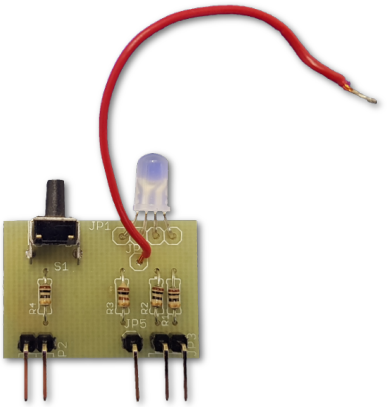


Figura 9.6: Tarjeta de E/S

9.3.1. El entorno de programación de Arduino

El entorno de programación de Arduino —véase la Figura 9.9— permite escribir un programa, guardarlo, compilarlo y subirlo al dispositivo Arduino que esté conectado al computador en el que este se esté ejecutando. Conviene tener en cuenta que para realizar los ejercicios propuestos, se deberá usar una variante del entorno de programación original que permite utilizar archivos fuente en ensamblador. Las instrucciones para instalar esta versión se pueden consultar en el sitio web de este libro.

¿Qué entorno de programación se debe instalar?

Puesto que el entorno Arduino está diseñado para poder desarrollar programas para varios modelos de tarjetas Arduino (22 en total), y que

¿Cómo configurar el entorno Arduino para la tarjeta Arduino Due?

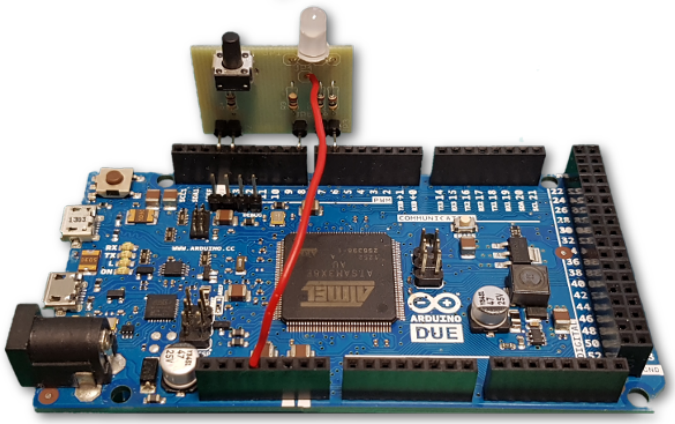


Figura 9.7: Tarjeta de E/S insertada en la Arduino Due

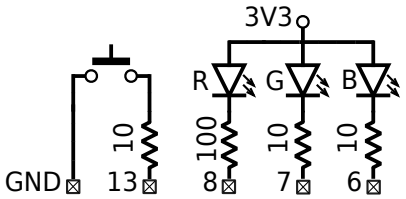


Figura 9.8: Esquema de la tarjeta de E/S

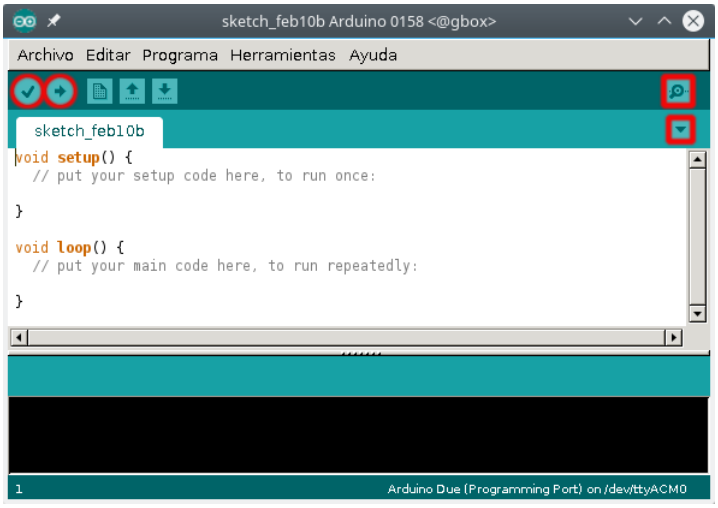


Figura 9.9: Entorno de programación Arduino. Se han marcado en rojo los botones que se utilizan más frecuentemente

varias de estas podrían estar conectadas a la vez al computador, será necesario indicar al entorno que se va a utilizar la tarjeta Arduino Due y el puerto USB al que está conectada. De esta forma, el entorno utilizará las herramientas de compilación correspondientes a la arquitectura ARM, las bibliotecas para dicha tarjeta, el protocolo adecuado para subir el código máquina y el puerto USB que se haya especificado para comunicarse con ella. Esta configuración consta de dos pasos. En primer lugar, se deberá seleccionar en el menú «`Herramientas ▷ Placa`» la opción «`Arduino Due (Programming Port)`». En segundo lugar, con la tarjeta Arduino Due conectada al computador por el puerto USB *Programming Port*, se deberá seleccionar en el menú «`Herramientas ▷ Port`» la opción «`/dev/ttyACMn`» en GNU/Linux, o «`COMn`» en Windows, que esté seguida del texto «`(Arduino Due (Programming Port))`».

Como se comenta más adelante, es posible desarrollar código y verificarlo —comprobar que se puede compilar sin errores— sin necesidad de disponer de una tarjeta Arduino. En caso de querer hacer esto, simplemente será necesario indicar el modelo de tarjeta antes de seleccionar la opción de verificar el código.

9.3.2. Proyectos Arduino

El entorno Arduino agrupa los ficheros fuente en una estructura denominada **proyecto**, que consiste en realidad en una carpeta, con el mismo nombre que el proyecto, que contiene los archivos fuente del proyecto. Cada proyecto tendrá al menos un archivo con su mismo nombre y con la extensión «`.ino`», que será el programa principal del proyecto. Este programa principal deberá definir las siguientes funciones:

- «`void setup()`»: Contiene el conjunto de acciones que se realizarán al inicio del programa. Aquí habitualmente se configuran las entradas/salidas que se vayan a emplear y se inicializan las variables necesarias para la ejecución del programa.

- «`void loop()`»: Contiene la parte del código que se ejecutará indefinidamente.

Cuando se ejecuta el entorno Arduino, este crea automáticamente un proyecto nuevo llamado «`sketch_MMMDDX`» —donde «`MMM`» es la abreviatura del nombre del mes actual, «`DD`», el día del mes y «`X`» es una letra, comenzando en «a», que permite diferenciar entre varios proyectos creados el mismo día—. Este proyecto creado de forma automática puede utilizarse directamente para desarrollar un proyecto nuevo.

Otra opción es partir de un proyecto de ejemplo de los proporcionados por el propio entorno de Arduino. Estos proyectos están organizados en categorías y se pueden abrir desde la entrada «`Ejemplos`» del

menú «`Archivo`». La Figura 9.10 muestra el entorno Arduino tras abrir el proyecto de ejemplo «`Blink`», que se encuentra dentro de la categoría «`01.Basics`». El significado del código de dicho proyecto se comenta más adelante, en el Apartado 9.3.3.

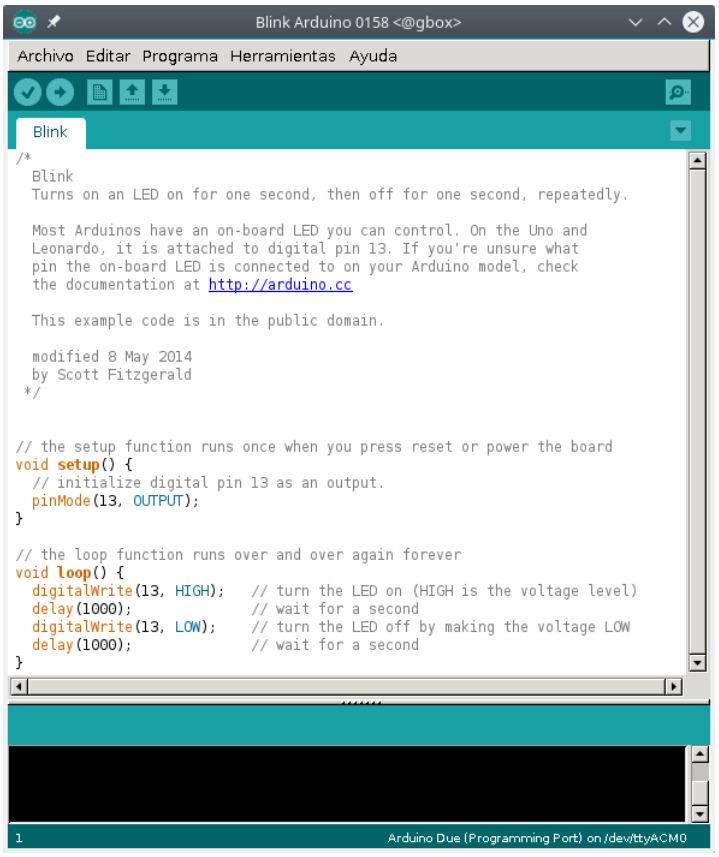


Figura 9.10: Entorno Arduino con el proyecto de ejemplo «Blink»

Por último, para abrir un proyecto ya creado, se puede utilizar la entrada «`Abrir...`» del menú «`Archivo`».

Conviene señalar que aunque inicialmente un proyecto conste de un único fichero es posible añadir nuevos ficheros a dicho proyecto. Para hacerlo desde el propio entorno, se puede emplear el botón con una flecha hacia abajo —de los botones resaltados en la Figura 9.9, el de abajo de los dos de la derecha—. Al hacer clic sobre dicho botón, se desplegará un menú del cual se deberá elegir la opción «`Nueva Pestaña`». Al hacerlo, en la parte inferior del entorno aparecerá una cuadro de texto en el que se solicitará el nombre del nuevo fichero. Una vez especificado el nombre y extensión del nuevo fichero, se deberá hacer clic sobre el botón `Ok` para

que se cree el archivo correspondiente y se muestre en forma de pestaña en el proyecto.

¿Cómo se compila un proyecto Arduino?

Una vez desarrollado el código fuente de un proyecto, este se puede compilar para comprobar si es correcto. Para ello, se puede seleccionar la entrada «`Verificar/Compilar`» del menú «`Programa`» o bien hacer clic sobre el botón `Verificar`, el que está más a la izquierda en la barra de botones y tiene forma de marca de aprobación —de los botones resaltados en la Figura 9.9, el que está más a la izquierda—. Tras completarse el proceso de compilación, si no hay errores, se mostrará en el panel inferior del entorno: la cantidad de memoria ocupada por el código máquina generado y el máximo de memoria disponible en la tarjeta Arduino seleccionada actualmente, como puede apreciarse en la Figura 9.11. En el caso de que haya errores, se mostrarán en este mismo panel, los errores de compilación encontrados, indicando para cada uno de ellos el tipo de error y el número de línea en que se encuentra. El entorno Arduino muestra cuál es el número de la línea en la que está actualmente el cursor en su barra de estado, a la izquierda. También es posible indicarle que numere todas las líneas abriendo el cuadro de diálogo «`Preferencias`» (menú «`Archivo`») y seleccionando la opción «`Display line numbers`».

¿Cómo mostrar los números de línea en el entorno Arduino?

Es importante recordar que aunque no se disponga en un momento dado de una tarjeta Arduino, siempre es posible compilar un proyecto en la forma descrita en el párrafo anterior para comprobar si es sintácticamente correcto.

¿Cómo se sube un proyecto Arduino a la tarjeta?

Una vez que un proyecto se compila correctamente, el siguiente paso consiste en subir el código máquina a la memoria ROM de la tarjeta Arduino para que esta pueda ejecutarlo. Para ello se debe hacer clic sobre el botón `Subir`, el que tiene forma de flecha hacia la derecha —de los botones resaltados en la Figura 9.9, el segundo por la izquierda—. Cuando se pulse dicho botón: I) el entorno compilará el proyecto, y en el caso de que se compile correctamente, II) subirá el código máquina a la tarjeta Arduino, y III) reiniciará la tarjeta Arduino, lo que hará que la tarjeta comience a ejecutar el código máquina subido. Como se puede observar, al hacer clic sobre el botón `Subir`, la primera acción que se realiza es la de compilar el proyecto. Por tanto, en el caso de modificar el código y querer subir la nueva versión a la tarjeta, no será necesario hacer clic primero sobre el botón `Verificar`, para luego hacerlo sobre el botón `Subir`. Bastará con hacer clic directamente sobre el botón `Subir`.

9.3.3. Ejemplo de proyecto Arduino con código en ensamblador

El ejemplo «`Blink`» (en `Archivo` ▷ `Ejemplos` ▷ `01.Basics`) muestra cómo configurar y utilizar la E/S de propósito general para hacer que el led integrado en la propia placa (y etiquetado con la letra «`L`») se en-

Figura 9.11: Resultado de la compilación del proyecto «Blink»

cienda y apague consecutivamente. Como se puede ver en la Figura 9.10 o 9.11, el proyecto «Blink» define las funciones «`setup()`» y «`loop()`» de la siguiente forma. La función «`setup()`», que se ejecutará una única vez en cuanto se inicie la placa Arduino, solo realiza una acción, que es llamar a la función de Arduino «`pinMode()`» con los parámetros «`13`» y «`OUTPUT`». Dicha llamada configura el pin 13 de la GPIO como salida. Por su parte, la función «`loop()`», que como sabemos se ejecutará de forma indefinida, llama a las funciones de Arduino, «`digitalWrite()`» y «`delay()`». La primera escribe un valor alto o bajo en el pin indicado. Así por ejemplo, «`digitalWrite(13, HIGH)`» escribirá un valor alto en el pin 13 —que recordemos que en la función «`setup()`» se había configurado como salida—. La segunda función, «`delay()`», simplemente espera a que transcurra el tiempo que se le haya indicado (en milisegundos). Así pues, la función «`loop()`» de «`Blink`»: I) pone un «`1`» en el pin 13, lo que enciende el led etiquetado con la letra «`L`» en la placa Arduino, puesto que dicho led está conectado al pin 13 de la placa; II) espera un

segundo, durante el cual el led permanece encendido; III) pone un «`0`» en el pin 13, lo que apaga el led de la placa Arduino; IV) y espera otro segundo, durante el cual el led permanece apagado. Puesto que la función «`loop()`» se ejecuta de forma indefinida, y cada vez que se llama a «`loop()`», se enciende y se apaga el led, el resultado será que el led de la placa Arduino parpadeará de forma indefinida.

La mayoría de proyectos Arduino, como es el caso del proyecto de ejemplo «`Blink`», se programan en un lenguaje de alto nivel, en `C` o en `C++`, y gestionan la E/S llamando a funciones proporcionadas por el entorno de programación de Arduino. Algunas de estas funciones de E/S, las más genéricas, se pueden usar directamente, como es el caso de las ya vistas: «`pinMode()`», «`digitalWrite()`» y «`delay()`». En cuanto a las funciones más específicas, basta con importar previamente la biblioteca de Arduino correspondiente —p.e., «`Ethernet`» para conectarse a Internet, «`SD`», para escribir y leer de tarjetas SD, etcétera—. Se puede consultar qué funciones y bibliotecas proporciona Arduino, abriendo la ayuda del propio entorno (para consultar las funciones de E/S genéricas, seleccionando el menú: `Ayuda` ▷ `Referencia`; y para ver el catálogo de bibliotecas, pulsando en el enlace `Libraries` en la anterior página de ayuda).

Así pues, la E/S en Arduino suele gestionarse a alto nivel por medio de llamadas a las funciones proporcionadas por el propio entorno de Arduino. Una de las ventajas de desarrollar proyectos de Arduino de esta forma es que los proyectos podrán ejecutarse prácticamente en cualquier modelo de Arduino, independientemente de cuál sea su hardware/arquitectura, y sin que sea necesario realizar ningún cambio en el código fuente. Aparentemente, el mismo software puede ejecutarse en tarjetas con hardware que en algunos casos es radicalmente distinto. ¿Dónde está el truco? Pues en que para compilar un proyecto Arduino, se ha tenido que indicar previamente para qué modelo de tarjeta se quería compilar y, en función de dicha elección, el entorno Arduino habrá seleccionado las bibliotecas específicas para dicho modelo de tarjeta, que sí dependen del hardware/arquitectura de dicho modelo.

Puesto que nuestro objetivo es ver cómo interactúa el procesador con la E/S, no es suficiente con utilizar las funciones de E/S de Arduino, que, además, se han diseñado para esconder la complejidad de la E/S. Deberemos programar el microcontrolador a bajo nivel, empleando directamente su lenguaje ensamblador. Que es justamente lo que hacen los desarrolladores de las bibliotecas de Arduino para las distintos modelos de tarjetas o, de forma más genérica, los desarrolladores de *drivers* de sistemas operativos.

¿Cómo incluir código ensamblador en un proyecto Arduino?

A continuación, y a modo de ejemplo de cómo programar en ensamblador utilizando el entorno Arduino, se va a desarrollar paso a paso una variante del proyecto de ejemplo «`Blink`» comentado previamente. En

esta nueva variante, la configuración de la E/S se hará desde C, pero el encendido y apagado del led se hará desde código en ensamblador. Así pues, comenzaremos creando un proyecto nuevo, llamado «blink_asm», cuyo programa principal, «blink_asm.ino», será el siguiente:

```
1  /*
2    blink_asm
3    Enciende y apaga el LED en ensamblador
4  */
5
6  #include "blink_asm.h"    // Para declarar la función externa blink
7  #define LED 13            // El LED está conectado al pin 13
8
9  void setup() {
10   pinMode(LED, OUTPUT);   // Configura el pin del LED como salida
11   Serial.begin(9600);     // Habilita la comunicación del puerto serie
12 }
13
14 void loop() {
15   int chip_id;            // Para almacenar el resultado de blink()
16
17   Serial.println("Llamamos a blink(5 parpadeos, 500 ms enc/apag)...");
18   chip_id = blink(5, 500);
19
20   Serial.print("El identificador de este Arduino es: ");
21   Serial.println(chip_id, HEX);
22   Serial.println("");
23
24   delay(5000);            // Espera 5 s antes de volver a empezar
25 }
```

Como se puede ver, al igual que el código del proyecto «Blink» mostrado en la Figura 9.10, «blink_asm.ino» define las funciones «setup()» y «loop()». La función «setup()»: I) configura el pin 13 como salida —aunque en lugar de poner el número 13, utiliza la constante «LED», definida en la línea 7 mediante «#define LED 13»—, y II) activa la comunicación serie de la tarjeta Arduino con el computador, lo que permitirá ver desde el entorno qué es lo que está haciendo la tarjeta Arduino cuando esté ejecutando el código.

En cuanto a la función «loop()», que es la que se ejecutará de forma indefinida, esta realiza varias llamadas a funciones de la biblioteca «Serial» —«Serial.print()» y «Serial.println()»— que envían información por el puerto serie al ordenador al que esté conectada la tarjeta Arduino. También, al igual que en el caso del proyecto ejemplo «Blink», se llama a la función «delay()», aunque en este caso para que entre una iteración de la función «loop()» y la siguiente, transcurran 5 segundos. Por último, la diferencia más importante con el proyecto de ejemplo «Blink», está realmente en las instrucciones «int chip_id;» y

«`chip_id = blink(5, 500);`». La primera de ellas declara una variable llamada «`chip_id`» del tipo entero, que sería equivalente[2] al ensamblador «`chip_id: .space 4`». La segunda llama a la función «`blink()`» pasándole como parámetros los números 5 y 500 y almacenando el resultado de dicha función en la variable «`chip_id`». Cuando se llame a la función «`blink()`» con los parámetros 5 y 500, el led de la tarjeta Arduino deberá parpadear 5 veces y cada encendido y apagado deberá durar 0,5 segundos. La función «`blink()`» es la que queremos programar en ensamblador.

El primer paso para poder llamar a una función en ensamblador desde una función en `C`, consiste en decirle al código en `C` que dicha función es una función externa. Para ello, desde el código en `C` se debe incluir un fichero que declare dicha función como externa. Esto se hace en la línea 6 del código de «`blink_asm.ino`». La línea «`#include "blink_asm.h"`» le indica al compilador que para compilar el código «`blink_asm.ino`», deberá tener en cuenta el fichero «`blink_asm.h`». Por su parte, el fichero «`blink_asm.h`» declara la función «`blink`» como una función externa que sigue el convenio `C` de paso de parámetros, que acepta dos parámetros de tipo entero y que devuelve como resultado un entero:

```
1  // Declaraciones de funciones externas
2  extern "C" {
3    /*
4      Función externa 'blink'. Hace parpadear el LED.
5      Parámetros de entrada:
6        int times: Número de veces que parpadeará el LED.
7        int delay: Milisegundos que el LED permanecerá encendido/apagado.
8      Devuelve:
9        int: Identificador del chip.
10   */
11   int blink(int times, int delay);
12 }
```

Ahora que ya tenemos el código «`blink_asm.ino`» que va a llamar a nuestra función en ensamblador y el fichero «`blink_asm.h`» que declara cuántos parámetros y de qué tipo acepta y devuelve nuestra función, ¡ya podemos programar en ensamblador! El programa en ensamblador, «`blink_asm.s`», que implementa la subrutina «`blink`» es el siguiente:

09_blink_asm.s

```
1  // blink_asm.s - Parpadeo en ensamblador
2  // Acceso al controlador del puerto PIOB
```

[2] Para el propósito del ejemplo, podemos considerar que «`int chip_id;`» es equivalente al ensamblador «`chip_id: .space 4`». Sin embargo, esto es cierto solo en el caso de que la variable «`chip_id`» se hubiera declarado de forma global en `C`. Si la variable se declara, como ocurre en este caso, en el ámbito de una función, en realidad lo que se hace es reservar espacio en la pila para dicha variable local.

```
3
4  // Directivas para el ensamblador
5  // ------------------------------
6  .syntax unified
7  .cpu cortex-m3
8
9  // Declaración de funciones externas y exportadas
10 // ----------------------------------------------
11 .extern  delay              @ delay es una función externa
12 .global blink              @ Para que la función 'blink' sea
13 .type blink, %function     @   accesible desde otros módulos
14
15
16 // Declaración de constantes
17 // -------------------------
18 .equ PIOB,     0x400E1000   @ Dirección base del puerto PIOB
19 .equ PIO_SODR, 0x030        @ Offset del Set Output Data Register
20 .equ PIO_CODR, 0x034        @ Offset del Clear Output Data Register
21 .equ LEDMSK,   0x08000000   @ El LED está en el pin 27
22 .equ CHIPID,   0x400E0940   @ Dirección del CHIP ID Register
23
24 // Comienzo de la zona de código
25 // -----------------------------
26 .text
27
28 /*
29    Subrutina 'blink'
30    Parámetros de entrada:
31      r0: Número de veces que parpadeará el LED
32      r1: Tiempo encendido/apagado (en milisegundos)
33    Parámetro de salida:
34      r0: Identificador del chip (contenido del registro CHIP_ID)
35 */
36 .thumb_func
37 blink:
38   push  {r4-r7, lr}          @ Apila de r4 a r7 y LR
39   mov   r4, r0               @ r4 <- número de parpadeos
40   mov   r5, r1               @ r5 <- tiempo encendido/apagado
41   ldr   r6, =PIOB            @ r6 <- dir. base del puerto PIOB
42   ldr   r7, =LEDMSK          @ r7 <- máscara con el bit 27 a 1
43 parpadeo:
44   str   r7, [r6, #PIO_SODR]  @ Enciende el LED escribiendo en SET
45   mov   r0, r5               @ r0 <- tiempo encendido/apagado
46   bl    delay                @ Llama a la función delay
47   str   r7, [r6, #PIO_CODR]  @ Apaga el LED escribiendo en CLEAR
48   mov   r0, r5               @ r0 <- tiempo encendido/apagado
49   bl    delay                @ Llama a la función delay
50   subs  r4, r4, #1           @ r4 <- parpadeos - 1 (¡Ojo la s!)
51   bne   parpadeo             @ Si no es cero, otro parpadeo.
52   ldr   r0, =CHIPID          @ r0 <- dirección del CHIP ID Register
53   ldr   r0, [r0]             @ r0 <- contenido del CHIP ID Register
54   pop   {r4-r7, pc}          @ Desapila r4 a r7 y vuelve
```

```
55 .end
```

Las primeras directivas del fichero «`blink_asm.s`», líneas 6 y 7, indican al programa ensamblador la variante del ensamblador de ARM que se va a utilizar. La línea 11, «`.extern delay`», sirve para indicar que se va a utilizar una subrutina definida en algún otro sitio que se llama «`delay`». Las líneas 12 y 13 sirven para hacer pública la subrutina «`blink`» que se implementa en este fichero, de forma que pueda invocarse desde el programa principal en lenguaje `C`. Las líneas 18 a 22 declaran una serie de constantes. Por último, a partir de la línea 36 comienza la subrutina «`blink`». Esta subrutina realiza las siguientes acciones para encender y apagar el led el número de veces indicado y para devolver el identificador del microcontrolador. En primer lugar, repite tantas veces como el valor indicado inicialmente en `r0` los siguientes pasos:

1. Pone a 1 el bit 27 del puerto B de E/S (PIOB).
2. Espera el número de milisegundos indicados inicialmente en `r1`.
3. Pone a 0 el bit 27 del puerto B de E/S (PIOB).
4. Espera el número de milisegundos indicados inicialmente en `r1`.

Por último, carga en `r0` el valor del registro de E/S `CHIPID`.

¿Por qué pone a 1 y a 0 el bit 27 del puerto B de E/S? El estándar Arduino otorga a cada E/S un número de identificación que es el mismo para todos los modelos de Arduino. Sin embargo, cada modelo de tarjeta puede asociar a cada pin de E/S, el puerto y bit de su GPIO que considere más conveniente. En el caso particular de la tarjeta Arduino Due, el pin 13 de E/S, al que está conectado el led «`L`», está asociado al bit 27 del puerto B de E/S del microcontrolador ATSAM3X8E. Por lo tanto, para poner a 1 o a 0 el pin 13 de la tarjeta Arduino Due, será necesario poner a 1 o a 0 el bit 27 del puerto B del microcontrolador. Es más, cuando se realicen llamadas a las funciones de Arduino, se deberá utilizar el número de pin —en este caso, el 13—, ya que estas funciones toman como argumento el número de pin. Mientras que al programar a bajo nivel se deberá modificar el bit y puerto correspondiente —en este caso, el bit 27 del puerto B—. El Cuadro A.1 muestra para cada dispositivo de la tarjeta de E/S, el número del pin al que está conectado y el correspondiente puerto y bit del microcontrolador ATSAM3X8E de la tarjeta Arduino Due.

Una vez completado el proyecto «`blink_asm`», este se puede subir a la tarjeta Arduino Due para ejecutarlo y comprobar que realmente hace lo que se espera de él. Conviene recordar aquí que en el fichero «`blink_asm.ino`» se realizaban una serie de llamadas a funciones de la biblioteca «`Serial`». Esas llamadas se han puesto con la intención de

poder supervisar qué está haciendo la tarjeta en un momento dado. Para hacerlo, una vez subido el código a la tarjeta, se debe hacer clic sobre el botón `Monitor Serie`, el que tiene forma de lupa —de los botones resaltados en la Figura 9.9, el de arriba de los dos de la derecha—. Al hacer clic sobre dicho botón, se enviará una señal de reinicio a la tarjeta Arduino, lo que provocará que la tarjeta comience a ejecutar desde el principio el programa que tenga almacenado, y en el ordenador se abrirá una nueva ventana en la que irán apareciendo los mensajes recibidos desde la tarjeta Arduino. En este ejemplo, al hacer clic sobre el botón `Monitor Serie`, debería aparecer un mensaje similar al siguiente, una y otra vez:

¿Cómo se abre el monitor serie?

```
Llamamos a blink: 5 parpadeos, 500 ms encendido/apagado
El identificador de este Arduino es: 285E0A
```

Además, justo después de mostrarse la primera línea, y antes de mostrarse la segunda, el led de la tarjeta debería parpadear 5 veces.

9.4. Ejercicios

▶ **9.1** Abre el proyecto de ejemplo «`Blink`» del entorno Arduino que se encuentra en «`Archivo` ▷ `Ejemplos` ▷ `01.Basics`» y completa los siguientes ejercicios:

9.1.1 Compila el proyecto y súbelo a la tarjeta (recuerda que basta con hacer clic sobre el botón `Subir`). Comprueba si parpadea el led incorporado en la tarjeta Arduino Due —el de color amarillo, situado aproximadamente entre los dos conectores USB e identificado con la letra «`L`»—. ¿Parpadea?

9.1.2 Sustituye en el fichero `Blink.ino` 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 esta nueva versión. ¿Qué ha pasado al ejecutarse en la tarjeta?

9.1.3 Modifica el programa `Blink.ino` para que haga parpadear el led de color rojo de la tarjeta de E/S. ¿Qué cambios has hecho?

▶ **9.2** Abre el proyecto «`blink_asm`», que se encuentra en la carpeta «`1. Primeros pasos ES`» de la colección de ejercicios para Arduino y resuelve los siguientes ejercicios:

9.2.1 Compila el proyecto, ejecútalo y comprueba si el led de la tarjeta Arduino Due parpadea. ¿Lo hace? ¿De igual forma que en el ejercicio anterior?

9.2.2 Abre el monitor serie. ¿Qué información se muestra en pantalla?

9.2.3 Recuerda que el microcontrolador ATSAM3X8E, incorporado en la tarjeta Arduino Due que se está usando, posee varios puertos PIO con varios pines de E/S en cada uno de ellos. Consulta el Cuadro A.2 para determinar la dirección base de los registros del puerto `PIOC`. ¿Cuál es?

9.2.4 Tal y como puedes comprobar en el Cuadro A.1, los pines `6`, `7` y `8` de la tarjeta Arduino están asociados a los bits `24`, `23` y `22`, respectivamente, del puerto `PIOC`. Recuerda además que, mientras que el led incorporado en la tarjeta Arduino Due se enciende escribiendo un `1` y se apaga escribiendo un `0` en el pin correspondiente, cada uno de los ledes del led RGB de la tarjeta de E/S se enciende escribiendo un `0` y se apaga escribiendo un `1` en su pin correspondiente. Sabiendo lo anterior, realiza las modificaciones necesarias en los ficheros `blink_asm.ino` y en `blink_asm.s` para hacer parpadear el led de color rojo de la tarjeta de E/S.

9.2.5 Comenta qué modificaciones has tenido que introducir en el programa.

▶ **9.3** Tal como vimos al estudiar las subrutinas en el Capítulo 6, hay dos formas de pasar parámetros del programa invocador a la subrutina: por valor y por referencia. En el proyecto del ejercicio anterior, «`blink_asm`», se ha visto cómo es posible pasar parámetros por valor desde un programa en `C` a un programa en ensamblador, y viceversa. Los parámetros de entrada, 5 y 500, se pasaron por valor a través de los registros `r0` y `r1`, y el parámetro de salida, el identificador del chip, también se devolvió por valor a través del `r0`—. Por su parte, el paso de parámetros por referencia implica indicar como parámetro la dirección de una variable en lugar de su valor (o un literal).

El proyecto «`blink_cadena`», dentro de la colección de ejercicios para Arduino, carpeta «`1. Primeros pasos ES`», es una modificación del proyecto «`blink_asm`», donde la subrutina «`blink()`», en lugar de parpadear tantas veces como se le indique, parpadeará tantas veces como caracteres tenga la cadena de caracteres que se le pase. Puesto que una cadena de texto es un tipo de dato estructurado, este parámetro se pasa por referencia —lo que se pasa es su dirección, no su contenido—.

Como se puede observar en el fichero `blink_cadena.ino`, la declaración en lenguaje `C` de la cadena se realiza de la siguiente forma:

«`char cadena[] = "mensaje";`»

Dicha expresión define la variable «cadena» como la dirección de comienzo de un vector de caracteres formado por los caracteres «m», «e», «n», «s», «a», «j», «e», y «NUL» (0). Por lo tanto, es equivalente a la siguiente declaración en ensamblador, ya conocida:

```
«cadena: .asciz "mensaje"»
```

Puesto que la variable «cadena» es realmente la dirección de comienzo de la cadena de caracteres, para pasar en C dicha variable por referencia no hay que hacer nada, basta con ponerla como parámetro, tal y como se hace en el fichero blink_cadena.ino:

```
«long_cad = blink(cadena, 500);»
```

De esta forma, cuando se llame a la subrutina «blink», el registro r0 contendrá la dirección de la variable «cadena» —y el registro r1, el valor 500—.

Teniendo en cuenta lo anterior, realiza los siguientes ejercicios:

9.3.1 Completa el programa ensamblador blink_cadena.s para que realice la función descrita. ¿Qué has cambiado?

9.3.2 Compila el proyecto, súbelo a la tarjeta Arduino Due y activa el monitor serie. ¿Qué se muestra en el monitor serie? ¿Coincide el número de parpadeos con el número de caracteres de «cadena»?

9.3.3 Modifica el contenido de la variable «cadena» por otra cadena con un tamaño distinto al de la actual. Compila el proyecto, súbelo y activa el monitor serie. ¿Qué se muestra en el monitor serie? ¿Coincide el número de parpadeos con el número de caracteres de «cadena»?

Ejercicios de nivel medio

▶ **9.4** El proyecto «lee_minutos», disponible en la colección de ejercicios para Arduino dentro de la carpeta «1. Primeros pasos ES», obtiene los minutos y segundos de la hora actual del reloj en tiempo real del ATSAM3X8E. Para hacerlo, la subrutina en ensamblador «lee_minutos» lee el registro RTC Time Register (véase la Figura A.5) y aplica las máscaras y desplazamientos necesarios para convertir la información codificada en ese registro de minutos y segundos en números enteros. Estos números se utilizarán desde el código en C para representar los minutos y segundos de la hora actual.

Este proyecto también muestra cómo es posible pasar un parámetro de salida por referencia. Para ello, comienza definiendo un vector de enteros de dos elementos con: «int minutos[2];», que sería equivalente al ensamblador: «minutos: .space 2*4». Después,

le indica a la función «`lee_minutos()`» que debe devolver los minutos y segundos modificando el contenido de dicho vector con: «`lee_minutos(minutos);`». Como se puede ver, puesto que la variable «minutos» contiene la dirección de comienzo del vector, no hay que hacer nada especial en `C` para pasar el vector por referencia, basta con indicar su nombre, como ya ocurría con la variable «`cadena`» en el proyecto anterior, «`blink_cadena`».

También se puede ver en este proyecto que la forma de acceder en `C` a los elementos del vector es la misma que en Python: «`vector[0]`» hace referencia al primer elemento del vector y «`vector[1]`», al segundo.

Teniendo en cuenta lo anterior, realiza los siguientes ejercicios:

9.4.1 Completa el programa ensamblador «`lee_minutos.s`» para que realice la función descrita. ¿Qué has cambiado?

9.4.2 Compila el proyecto, súbelo a la tarjeta Arduino Due y activa el monitor serie. ¿Qué se muestra en el monitor serie? ¿Coincide con lo esperado?

Ejercicios de nivel avanzado

▶ **9.5** El proyecto «`lee_fecha`», disponible en la colección de ejercicios para Arduino dentro de la carpeta «`1. Primeros pasos ES`», es una variante del proyecto anterior, «`lee_minutos`», que obtiene información, en lugar de sobre la hora actual, sobre la fecha actual del reloj en tiempo real del ATSAM3X8E. Para hacerlo, la subrutina en ensamblador «`lee_fecha`» lee el registro `RTC Calendar Register` (véase la Figura A.3) y aplica las máscaras y desplazamientos necesarios para convertir la información codificada en ese registro de siglo, año, mes, día y día de la semana en números enteros. Estos números se utilizarán desde el código en `C` para representar la fecha actual.

9.5.1 Completa el programa ensamblador «`lee_fecha.s`» para que realice la función descrita. ¿Qué has cambiado?

9.5.2 Compila el proyecto, súbelo a la tarjeta Arduino Due y activa el monitor serie. ¿Qué se muestra en el monitor serie? ¿Coincide con lo esperado?

9.5.3 Sobre la base de la anterior información, ¿cuál es la fecha con la que se inicia el RTC por defecto?

9.5.4 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, y tras averiguar a qué día de la semana corresponde la fecha inicial, ¿qué codificación se emplea por defecto para el día de la semana en ese campo?

Ejercicios adicionales

▶ **9.6** El proyecto «`blink_vect`» —carpeta «1. Primeros pasos ES» de la colección de ejercicios para Arduino— es una variante del proyecto visto en un ejercicio anterior, «`blink_cadena`», que añade como funcionalidad la opción de mostrar el contenido de los registros `r0` al `r7` justo antes de volver de la rutina «`blink()`».

9.6.1 Completa dicho proyecto para que funcione adecuadamente. ¿Qué cambios has realizado?

9.6.2 Sube el proyecto a la tarjeta Arduino y activa el monitor serie. ¿Coincide lo que aparece en el monitor serie con lo siguiente?

```
Llamamos a blink(cadena, 100 ms enc/apag, regs)...
La cadena tiene 7 caracteres

El contenido de los registros r0-r7 es:
r0: 0x64
r1: 0x64
r2: 0x20087FC0
r3: 0x20087FB8
r4: 0x7
r5: 0x64
r6: 0x400E1200
r7: 0x1000000
```

9.6.3 ¿Para qué se ha utilizado cada uno de los anteriores registros en la rutina «`blink()`»?

▶ **9.7** En el proyecto anterior, el valor del registro `r2` tras la ejecución del bucle de la subrutina «`blink`», se sobrescribe con la dirección del vector «`regs`». Modifica el código `blink_vect.s`, completado en el ejercicio anterior, de tal forma que el valor mostrado de todos los registros coincida con su valor al llegar a la etiqueta «`fin`». Comprueba que los valores de todos los registros, salvo el del registro `r2`, coinciden con los mostrados en el ejercicio anterior. ¿Qué valor se muestra ahora para `r2`?

Capítulo 10

Gestión de la entrada/salida y otros aspectos avanzados

Índice

Tras ver en los capítulos anteriores cuál es la función de la entrada/salida en los ordenadores, así como algunos de los dispositivos más habituales, este capítulo describe cómo se gestiona y otros aspectos avanzados de la entrada/salida. Se verán los dos mecanismos existentes para sincronizar el procesador y los dispositivos: consulta de estado e interrupciones, junto con sus ventajas e inconvenientes. Posteriormente se describirá la forma de transferir grandes cantidades de datos entre los dispositivos y la memoria principal, utilizando el acceso directo a memoria para liberar de esta tarea al procesador. Por último, se comentará brevemente la necesidad de estandarizar la entrada/salida en los

sistemas actuales, tanto en la conexión física entre el ordenador y los dispositivos, mediante buses estándar, como entre las partes del sistema operativo que los gestionan, los controladores de dispositivos.

10.1. Gestión de la entrada/salida

Tal y como vimos cuando se introdujo la productividad de los dispositivos de entrada/salida, estos son generalmente mucho más lentos que el procesador. Esta diferencia entre la velocidad del procesador y la de los dispositivos de entrada/salida no siempre supone un problema. Por ejemplo, cuando se utiliza un ordenador para ejecutar un único programa cuyo flujo de ejecución dependa de las operaciones de entrada/salida. En este caso, el procesador simplemente esperará a que se vayan produciendo cambios en los dispositivos de entrada/salida, y el funcionamiento del sistema será el deseado. Sin embargo, este no es el caso general. En un ejemplo como el utilizado en capítulos anteriores, en que el ordenador está imprimiendo textos del usuario, no parece razonable que el procesador se quede bloqueado esperando respuestas de la impresora —señales de que el trabajo en curso ha terminado o indicaciones de error—. Más bien estamos acostumbrados a realizar cualquier otra actividad con el ordenador mientras la impresora va terminando hoja tras hoja sin percibir apenas disminución en el rendimiento del sistema.

Así pues, y por lo general, se pretende que el procesador y los dispositivos se sincronicen de tal modo que aquel solo les preste atención cuando deba realizar alguna acción —recoger datos si ya se han obtenido, enviar nuevos datos si se han consumido los anteriores, solucionar algún error o avisar al usuario de alguna incidencia—. La **sincronización** entre el procesador y los dispositivos es un aspecto fundamental de la gestión de la entrada/salida. En los siguientes dos subapartados se explican los dos mecanismos que se pueden utilizar para sincronizar el procesador y los dispositivos: la consulta de estado y las interrupciones.

10.1.1. Gestión de la entrada/salida mediante consulta de estado

Como hemos visto, los registros de estado del controlador de un dispositivo sirven para, entre otras cosas, indicar que se ha producido alguna circunstancia que requiere atención. La forma más sencilla de sincronización con el dispositivo, llamada **consulta de estado**, **prueba de estado** o **encuesta** —*polling*, en inglés—, consiste en que el procesador, durante la ejecución del programa en curso, lea de cuando en cuando los registros de estado de los dispositivos de entrada/salida y, si advierte que un dispositivo requiere atención, pase a ejecutar el código

necesario para prestársela, posiblemente contenido en una subrutina de gestión del dispositivo en cuestión.

El código que aparece a continuación podría ser un ejemplo de gestión de la entrada/salida mediante consulta de estado.

10_consulta_estado.s

```
1         ldr     r7, =ST_IMPR    @ r7 <- dir. del registro de estado
2         ldr     r6, =0x00000340 @ r6 <- máscara bits de interés
3         ldr     r0, [r7]        @ r0 <- registro de estado
4         ands    r0, r6          @ r0 <- bits de interés de r0
5         beq     sigue           @ Seguimos si no hay avisos
6         bl      TRAT_IMPR       @ Si los hay, llama a subr. trat.
7 sigue:  ...                     @ Continúa
```

En este ejemplo, el procesador consulta el registro de estado de una impresora y, si ninguno de los bits 6, 8 o 9 está a `1`, ignora la impresora y continúa con el resto del programa. En el caso contrario, si alguno de los bits estaba a `1`, el procesador salta a una subrutina de tratamiento de la impresora. Lo normal es que en esa subrutina el procesador compruebe cuáles de los tres bits del registro de estado están activos, y emprenda las acciones necesarias para gestionar esa circunstancia. Una vez vuelva de la subrutina, continuará con el resto del programa.

Esta forma de gestionar la entrada/salida es muy sencilla, no añade complejidad al procesador y puede usarse en todos los sistemas. En muchos de ellos, si están dirigidos por eventos de entrada/salida —es decir, si 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 sincronización 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 programa debe incluir instrucciones para verificar cada cierto tiempo el estado del dispositivo, lo que consumirá inútilmente tiempo del procesador mientras el dispositivo no requiera atención. En un sistema con decenas de dispositivos gestionados mediante consulta de estado, la cantidad de tiempo perdida podría llegar a ser excesiva. Por otra parte, el tiempo que transcurre entre consultas provoca que la latencia, el tiempo transcurrido entre que el dispositivo requiere atención y el procesador se la preste, sea muy variable. Si un dispositivo activa un bit de estado justo antes de que el procesador lea el registro de estado, la latencia correspondiente 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 realice una nueva consulta, por lo que la latencia será mayor. Cuanto menos tiempo transcurra entre consulta y consulta, menor será la variabilidad de la latencia, pero a costa de

consumir más tiempo de forma inútil. Cuanto más tiempo transcurra entre consulta y consulta, menos tiempo se perderá inútilmente, pero a costa de aumentar la variabilidad de la latencia.

En este apartado hemos visto que la sincronización entre el procesador y los dispositivos, punto clave de la gestión de la entrada/salida, puede realizarse consultando periódicamente el estado del dispositivo para ver si requiere atención. Para poner en práctica esta estrategia, puedes realizar los siguientes ejercicios:

▶ **10.1** El proyecto «`pulsador`» —en la colección de ejercicios para Arduino, dentro de la carpeta «`2. Consulta de estado`»— configura en ensamblador el pin al que está conectado el pulsador de la tarjeta de E/S como entrada con *pull-up* —llamando a la subrutina «`PIOB_pushbutton_setup`»—, para después detectar mediante consulta de estado, también en ensamblador, cuándo se ha pulsado —llamando a la subrutina «`pulsador`»—. Abre este proyecto en el entorno Arduino y realiza los siguientes ejercicios:

10.1.1 Los puertos GPIO del ATSAM3X8E, véase el Apartado A.2, disponen de un conjunto de registros de control que permiten activar o desactivar ciertas características de sus bits. Para hacerlo, simplemente se debe escribir una palabra que tenga activos aquellos bits para los que se quiere activar o desactivar una característica determinada, en el registro de control adecuado. Por ejemplo, para evitar que las señales presentes en los bits 2 y 5 del puerto PIOB actúen como sendas señales de petición de interrupción, se debe escribir la palabra `0x0000 0024` —bits 2 y 5 a `1`, el resto a `0`— en el registro «`PIO_IDR`» —*Interrupt Disable Register*— de ese puerto. Si por el contrario se quisiera que las señales presentes en esos mismos bits actuaran como señales de petición de interrupción, se debería escribir la misma palabra que en el caso anterior, `0x0000 0024`, pero en lugar de en el registro «`PIO_IDR`», en el registro «`PIO_IER`» —*Interrupt Enable Register*—. De forma general, para modificar varias características de uno o varios bits, basta con inicializar una palabra con estos bits activos, y escribir esa misma palabra en todos aquellos registros que activen o desactiven las características deseadas.

En el ejemplo que nos ocupa se desean modificar varias características asociadas al bit 27 del puerto PIOB. Expresa en hexadecimal qué palabra escribirías en los distintos

registros de control para activar o desactivar ciertas características asociadas a este bit.

10.1.2 Completa el código del fichero «`PIOB_pushbutton_setup.s`» para que configure el bit 27 del PIOB como entrada con *pull-up*, con filtrado de espurios y rebotes, que no se utilice como señal de petición de interrupción y que forme parte del puerto PIO. Ten en cuenta tu respuesta anterior y los comentarios del fichero.

10.1.3 Sabiendo que el pulsador incorporado en la tarjeta de E/S está asociado al bit 27 del puerto PIOB, ¿qué registro de ese puerto deberíamos leer para comprobar si se ha presionado el pulsador? —La configuración y lectura/escritura de la GPIO se pueden consultar en los apartados A.2.2 y A.2.3, respectivamente.

10.1.4 Teniendo en cuenta de nuevo que el pin al que está conectado el pulsador se corresponde con el bit 27 del puerto PIOB, ¿qué máscara tendríamos que aplicar al valor leído del registro obtenido en el ejercicio anterior para ignorar el valor de todos sus bits excepto el que indica el estado del pulsador? —Exprésala en hexadecimal.

10.1.5 De acuerdo con el esquema de conexión del pulsador de la tarjeta de E/S mostrado en la Figura 9.8 y sabiendo que la entrada en la que se encuentra conectado el pulsador tiene activada la resistencia interna de *pull-up*, ¿qué valor leído del bit 27 del registro obtenido anteriormente del puerto PIOB, `0` o `1`, nos indicará que el pulsador está presionado?

10.1.6 Teniendo en cuenta tus respuestas a los ejercicios anteriores, completa el fichero «`pulsador.s`» de este proyecto.

10.1.7 Sube el proyecto a la tarjeta Arduino Due y abre el monitor serie. ¿Qué hace el programa? ¿Cómo interactúa con el usuario?

10.1.2. Gestión de la entrada/salida mediante interrupciones

A la vista de los problemas que presenta la gestión de la entrada/salida mediante consulta de estado, estaría bien que en lugar de que el procesador tuviera que consultar periódicamente el estado de los distintos dispositivos, fueran estos los que, en caso de necesitar su atención, avisaran al procesador. Para poder implementar un mecanismo de este

tipo, puesto que el procesador es el que debe gestionar todo el sistema, deberá también poder decidir qué dispositivos tienen permiso para avisarle y qué avisos puede ignorar en un momento dado.

Desarrollando esta idea, el **mecanismo de gestión de la entrada/salida mediante interrupciones** consiste en que cuando un dispositivo requiere atención, además de señalarlo en sus registros de estado, genere, si dispone de los permisos adecuados, una señal eléctrica que haga que el procesador, al terminar de ejecutar la instrucción en curso, salte automáticamente al código encargado de su gestión, que recibe el nombre de **rutina de tratamiento de la interrupción** (**RTI**) o **rutina de servicio de la interrupción**, y que, una vez completado su tratamiento, continúe con la ejecución de la siguiente instrucción como si la interrupción no hubiera tenido lugar —de forma similar a cuando se salta y vuelve de una subrutina, pero con más implicaciones que se verán más adelante— (véase la Figura 10.1). El símil que se suele utilizar para ilustrar este proceso es el de que mientras estamos leyendo tranquilamente un libro —ejecutando un programa—, por ejemplo este, llega una llamada telefónica, whatsapp, correo, lo que sea, cualquier cosa... Al sonar el teléfono —señal de interrupción—, ponemos rápidamente una marca en la página que estábamos leyendo y atendemos la llamada —tratamiento de la interrupción—. Por último, una vez termina la conversación, continuamos con la lectura a partir de donde la habíamos dejado —se continúa con la ejecución del programa interrumpido—.

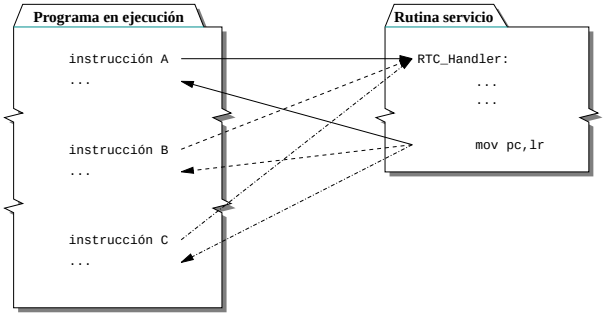


Figura 10.1: Ejemplo del proceso de atención de una interrupción. El procesador, al finalizar la ejecución de las instrucciones A, B y C, detecta que se ha producido una petición de interrupción del reloj en tiempo real (RTC), por lo que: almacena el estado actual; salta a la rutina de tratamiento correspondiente; y al volver, antes de comenzar la ejecución de la siguiente ejecución del programa, recupera el estado anterior

Volviendo al mecanismo por el que se gestiona la entrada/salida mediante interrupciones en un computador, se puede deducir fácilmente de la explicación anterior que este mecanismo permite que: i) los disposi-

tivos pueden ser atendidos con muy poca latencia —el tiempo que lleve terminar de ejecutar la instrucción en curso—, y II) los programas de aplicación no necesitan incluir instrucciones para consultar el estado de los dispositivos, ni, de hecho, ocuparse de la gestión de la entrada/salida.

Requisitos del hardware

Para poder implementar este mecanismo es necesario contar con hardware que lo sustente, tanto en el procesador como en los dispositivos de entrada/salida. Justamente por este motivo, no todos los procesadores pueden gestionar interrupciones, aunque hoy en día, esto tan solo ocurre con los microcontroladores de muy bajo coste. En los siguientes párrafos se detallan los elementos hardware que deben incorporar tanto el procesador como los dispositivos para poder llevar a cabo cada uno de los procedimientos implicados en la gestión de la entrada/salida mediante interrupciones.

En primer lugar, como hemos dicho, el aviso de que un dispositivo requiere atención, llega al procesador mediante una señal eléctrica. Esto requiere, por una parte, que el procesador —o su núcleo, en los procesadores y microcontroladores con dispositivos integrados— disponga de una o varias líneas —pines de entrada o contactos eléctricos— que puedan recibir estas señales. Estas líneas se suelen denominar **líneas de petición de interrupción** y etiquetarse en los circuitos como **IRQn** —IRQ por *Interrupt Request*—, donde la n indica el número de línea en el caso de haber varias. Por otra parte, los controladores de los dispositivos capaces de realizar estos avisos, tienen que poder generar estas señales eléctricas y contar con una línea de salida —o varias en algunos casos— para poder enviar estas señales al procesador.

También, y como ya se ha comentado, el procesador, guiado por el código con el que ha sido programado, es el que debe encargarse de gestionar todo el sistema. Esta condición dejaría de cumplirse si los dispositivos de entrada/salida pudieran interrumpirlo de forma indiscriminada. Así pues, el procesador debe poder seleccionar qué dispositivos tienen permiso para interrumpirlo en un momento dado y cuáles no. Esto se consigue de dos formas. En primer lugar, el procesador podrá modificar uno o varios bits de control propios para deshabilitar totalmente las interrupciones, hacerlo por grupos, según prioridades, etcétera. Estas opciones se detallarán más adelante. En segundo lugar, los dispositivos que puedan generar interrupciones deberán disponer de **bits de habilitación de interrupciones** que el procesador podrá activar o desactivar para que los dispositivos generen o no, respectivamente, peticiones de interrupción. Estos bits normalmente forman parte de los registros de control de los controladores de los dispositivos, que tendrán un bit o más, según los tipos de interrupciones que sean capaces de generar. Tanto el

procesador como los dispositivos, además de los elementos de memoria necesarios para almacenar la información anterior, deberán contar con lógica de control que tenga en cuenta esa información para decidir qué interrupciones atender, en el caso del procesador, y para generar o no interrupciones de un determinado tipo, en el caso de los dispositivos (véase la Figura 10.2).

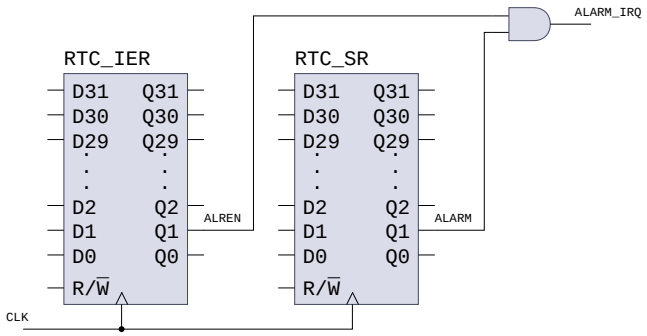


Figura 10.2: Elementos de un dispositivo que intervienen en una petición de interrupción. La petición de interrupción `ALARM_IRQ` de un reloj en tiempo real (RTC) solo se activará cuando se active el bit `ALARM` del registro de estado `RTC_SR` al saltar la alarma y esté habilitada la petición de interrupción por alarma (bit `ALREN` del registro de control `RTC_IER` a 1)

El siguiente aspecto a considerar es la respuesta del procesador ante la señalización de una interrupción habilitada. La forma en la que un procesador deberá responder a una petición de interrupción vendrá determinada por su arquitectura. La organización del procesador y sus circuitos, siempre siguiendo las directrices de su arquitectura, deberán permitir en primer lugar 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 de la rutina de tratamiento de la interrupción correspondiente. Además, y dependiendo de lo establecido por la arquitectura, el procesador deberá diseñarse de tal forma que sea capaz de realizar otras acciones antes de comenzar la ejecución de la rutina de tratamiento, como cambiar de estado a modo privilegiado o supervisor, usar otra pila u otro conjunto de registros, deshabilitar automáticamente las interrupciones, etcétera. Por último, la organización del procesador y sus circuitos deberán ser capaces de deshacer todos estos cambios al volver de la rutina de tratamiento de la interrupción, recuperando el estado en que este se encontraba al producirse la interrupción y continuando con el código de aplicación como si esta no hubiera tenido lugar. El comportamiento del procesador descrito en este párrafo se analizará con más detalle en breve.

El último mecanismo que debe proveer el hardware del procesador, siguiendo lo especificado en su arquitectura, tiene que ver con la dirección de inicio de la rutina de tratamiento de una interrupción. Esta dirección, que puede ser única o dependiente de la interrupción, recibe el nombre de **vector de interrupción**. La utilización de vectores de interrupción, en lugar de direcciones prefijadas para cada rutina de tratamiento de interrupciones, proporciona libertad a los desarrolladores de sistemas para: I) ubicar las rutinas de tratamiento en la zona de memoria que prefieran, II) emplear la cantidad de memoria que sea necesaria para cada una de ellas, y III) extender rutinas proporcionadas por la BIOS o por el sistema operativo con código propio —que es lo que hacen por ejemplo los antivirus—. En el caso más sencillo, el procesador dispondrá de una única línea de petición de interrupción y, por tanto, de un único vector de interrupción. Si este es el caso, la rutina de tratamiento de interrupciones —solo habrá una— deberá en primer lugar averiguar qué dispositivos han provocado la interrupción y esperan ser atendidos. Para ello, consultará el estado de todos y cada uno de los dispositivos habilitados para generar interrupciones. Por otra parte, en los casos más complejos, el procesador distinguirá entre varias posibles interrupciones, cada una identificada por un número de interrupción y asociada a un vector de interrupción diferente. En estos casos, el procesador deberá o bien tener varias líneas de interrupción independientes, o implementar un protocolo especial de interrupción en el que, además de una señal eléctrica, el dispositivo indique al procesador el número de interrupción que desea provocar. En estos casos, cada rutina de tratamiento se encarga de uno o de unos pocos dispositivos, por lo que la fase inicial de averiguar qué dispositivo requiere atención es siempre más rápida que en el caso de una única rutina de tratamiento para todos los dispositivos.

Recapitulando, los requisitos que debe satisfacer el hardware del procesador y los dispositivos para poder implementar un mecanismo de interrupciones se resumen en:

1. Líneas de interrupción en el procesador y en los dispositivos capaces de generarlas.

2. Bits de control para habilitar y deshabilitar las interrupciones de forma global en el procesador, y local en los dispositivos.

3. Lógica en el núcleo del procesador para implementar las acciones necesarias para atender una interrupción y continuar con el funcionamiento habitual.

4. Gestión de la dirección de inicio de la rutina de tratamiento en el procesador, normalmente mediante vectores de interrupción.

Interrupciones y excepciones

Conviene comentar que el mecanismo de interrupciones ha demostrado ser tan eficaz que su uso se ha extendido más allá de la entrada/salida bajo el nombre de **excepciones** —*exceptions* o *traps*, en inglés—. Una excepción permite señalar que se ha producido un suceso fuera de lo habitual —excepcional— durante el funcionamiento del computador. En este marco más amplio, las interrupciones son un tipo de excepciones: las generadas por los dispositivos de entrada/salida. Otras excepciones sirven para señalar errores —accesos a memoria inválidos, violaciones de privilegio, división por cero, etcétera—. En muchos casos, estos avisos de errores pueden ser tratados por el sistema y dar lugar a extensiones útiles. Por ejemplo, muchos sistemas operativos capturan la excepción de fallo de página de memoria para extender la memoria principal utilizando almacenamiento en disco. Otro ejemplo sería el caso de la excepción de coprocesador no presente, cuya captura permite utilizar un emulador del coprocesador en su lugar, como ocurría con la unidad en coma flotante emulada en los antiguos PC. Finalmente, otras excepciones son aquellas que se generan voluntariamente por software mediante la ejecución de instrucciones máquina específicas o escribiendo en registros que la arquitectura proporciona a tal efecto. Este último tipo de excepciones, aprovechando que el tratamiento de excepciones conlleva un cambio al modo de ejecución privilegiado, permiten implementar las llamadas al sistema operativo.

Como se acaba de comentar, las interrupciones se suelen clasificar como un tipo de excepciones asociado a los dispositivos de entrada/salida y a su gestión. Sin embargo, existe una diferencia fundamental entre las interrupciones y el resto de excepciones que conviene tener en cuenta. Las interrupciones, puesto que son generadas por los dispositivos de entrada/salida, son totalmente asíncronas con la ejecución de los programas y no tienen ninguna relación temporal con ellos que pueda ser conocida de antemano. Por el contrario, el resto de excepciones se generan cuando el procesador ejecuta una instrucción —la excepción de acceso incorrecto a memoria se genera cuando el procesador intenta ejecutar una instrucción de acceso a una dirección de memoria que no existe o no está correctamente alineada; la excepción de división por cero, cuando el procesador ejecuta una instrucción de división en la que el divisor es 0—. En este caso, el programador puede saber de antemano que ciertas instrucciones pueden dar lugar a una excepción y establecer mecanismos para capturarlas.

Implicaciones en los sistemas y en su diseño

Hasta ahora se han expuesto las nociones básicas de la gestión de la entrada/salida mediante interrupciones. A partir de estas nociones, a continuación se describen con más detalle las implicaciones que el uso de interrupciones tiene tanto en la implementación de un sistema informático como en su posterior utilización.

En primer lugar conviene saber que, para diseñar un sistema informático que gestione su entrada/salida por medio de interrupciones, se deberá comenzar por la fase de diseño del hardware del sistema, empezando por diseñar o seleccionar un procesador capaz de gestionar interrupciones. Un procesador de este tipo deberá contar con una o varias líneas externas de interrupción, lo que a su vez condicionará el diseño del hardware del resto del sistema. Es más, si el procesador escogido cuenta con interrupciones vectorizadas, los dispositivos, además de avisar al procesador de que solicitan una interrupción, deberán proporcionar su número de interrupción siguiendo un determinado protocolo de comunicación. Para reducir esta complejidad añadida en el diseño del resto del sistema, se suele recurrir a los controladores de interrupciones. Un **controlador de interrupciones** es un dispositivo que se conecta a las líneas de petición de interrupción del procesador y proporciona sus propias líneas de petición de interrupción a las que se conectan los restantes dispositivos. El controlador se encarga de propagar al procesador las señales que recibe en sus líneas de petición de interrupción, a la vez que le envía el número de interrupción correspondiente siguiendo el protocolo de comunicación requerido por el procesador. 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. Desde el punto de vista del procesador, el controlador de interrupciones funciona como un sencillo dispositivo de entrada/salida. El procesador puede programar estos controladores para indicar el número de interrupción que se debe asociar a cada una de sus líneas de petición de interrupción, la prioridad de las interrupciones, las máscaras de habilitación y algún otro aspecto relacionado con el comportamiento eléctrico de las interrupciones.

Otra consideración que deberá tenerse en cuenta en la fase de diseño del hardware del sistema es la forma en la que los dispositivos indicarán que requieren atención. Como ya se ha comentado, las salidas de interrupción de los distintos dispositivos se conectan a las entradas de petición de interrupción del procesador, o en su caso, a las del controlador de interrupciones, para poder avisar al procesador de que requieren atención por medio de una señal eléctrica. Existen dos modos de utilizar una señal eléctrica para avisar de que se requiere atención. En el primero de ellos, un dispositivo avisará al procesador de que requiere

atención poniendo la línea de interrupción en su nivel activo —ya sea alto o bajo— y manteniendo dicho nivel hasta que sea atendido. En este caso, el procesador puede saber que un dispositivo requiere atención si el nivel de una de sus líneas de petición de interrupciones está en el nivel activo. Este modo de señalización de interrupciones recibe el nombre de **interrupción disparada por nivel**, que puede ser bajo o alto. En el segundo modo, un dispositivo avisará al procesador generando un pulso en la línea de interrupción para después dejarla de nuevo en su estado inactivo. En este caso, el procesador detectará que ha habido una petición cuando en una de sus líneas de petición de interrupciones haya ocurrido una transición del nivel inactivo (bajo o alto) al activo (alto o bajo). Este modo de señalización de interrupciones recibe el nombre de **interrupción disparada por flanco**, que será de subida o de bajada. Cada uno de estos modos tiene una serie de implicaciones que conviene tener en cuenta cuando se conecten varios dispositivos a una misma línea de interrupción. En el modo de disparo por nivel, en el que los dispositivos mantienen la línea activa mientras no son atendidos, el procesador sabrá que aún quedan dispositivos por atender mientras la línea de petición de interrupción esté activa. Por contra, mientras posponga la atención a un determinado dispositivo, no será capaz de detectar que otros dispositivos conectados a la misma línea han pasado a requerir su atención. Por otro lado, en el modo de disparo por flanco, donde los dispositivos generan un pulso y liberan la línea, el procesador puede posponer la atención a un dispositivo y aún así detectar que otros dispositivos han pasado a requerir su atención. Por contra, el procesador deberá haberse diseñado para llevar la cuenta de aquellos dispositivos a los que aún no ha atendido.

El siguiente aspecto relacionado con el hardware del sistema tiene que ver con la utilización de vectores de interrupción para indicar la dirección de comienzo de la rutina de tratamiento que deberá ejecutar el procesador cuando atienda una determinada interrupción. Un vector de interrupción es en realidad una zona de memoria en la que se indica de alguna forma cuál es la dirección de comienzo de la rutina que debe ejecutarse para tratar la interrupción asociada a ese vector. La arquitectura del procesador determina la dirección de memoria de cada vector de interrupción,[1] el tamaño de los vectores de interrupción y la forma de codificar la dirección de comienzo de las rutinas de interrupción. Para indicar la dirección de comienzo de las rutinas de tratamiento de interrupción se puede utilizar una de las técnicas siguientes. La primera de ellas, que hace más sencillo el diseño del procesador, consiste en dejar

[1] Algunas arquitecturas permiten configurar la dirección de inicio de los vectores de interrupción. En este caso, la arquitectura determina la dirección relativa de los distintos vectores de interrupción con respecto a una dirección inicial, en lugar de su dirección absoluta.

suficiente espacio entre dos vectores de interrupción para que quepa una instrucción de salto absoluto e inicializar cada vector de interrupción con una instrucción de salto a su rutina de tratamiento. De esta forma, cuando el procesador deba atender un determinado número de interrupción, simplemente deberá actualizar su PC con la dirección del vector de interrupción correspondiente. La segunda técnica, que complica el diseño del procesador, consiste en dejar espacio entre dos vectores de interrupción para almacenar una dirección de memoria e inicializar cada vector de interrupción con la dirección de inicio de su rutina de tratamiento de interrupción. Si se utiliza esta técnica, cuando el procesador deba atender un determinado número de interrupción, deberá leer en primer lugar el contenido del vector de interrupción correspondiente, para después actualizar el PC con la dirección de memoria obtenida. La primera opción conlleva que el hardware del procesador sea capaz de modificar el PC con la dirección de un vector de interrupción. La segunda opción complica el diseño del procesador, ya que además de modificar el PC con una dirección de memoria, el hardware del procesador debe ser capaz de leer previamente la dirección de salto de la dirección de memoria del vector de interrupción correspondiente. La primera opción es utilizada, por ejemplo, en las arquitecturas ARM que no son de la familia Cortex y la segunda, en las arquitecturas ARM Cortex y las Intel de 32 y 64 bits.

La siguiente etapa a considerar, una vez diseñado —o seleccionado— el hardware del sistema de tal forma que este permita utilizar interrupciones para la gestión de la entrada/salida, es la configuración inicial 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: I) habilitar qué interrupciones se quieren tratar, y II) asignar prioridades a las distintas interrupciones. Ya se había visto previamente que el procesador puede decidir qué dispositivos pueden interrumpirle en un momento dado, lo que no se había comentado hasta ahora es que, además, puede asignar prioridades a los dispositivos que sí pueden interrumpirle. Al hacerlo, ante distintas interrupciones pendientes, puede decidir a cuál de ellas atender primero, o en el caso de que llegue una nueva interrupción mientras está atendiendo otra, decidir si debe pasar a atender la nueva —en el supuesto de que sea más prioritaria—, o simplemente ignorarla temporalmente —en el caso de que sea menos prioritaria—. Como ejemplo de la necesidad de establecer interrupciones con distinta prioridad, consideremos el caso de un teléfono móvil inteligente. Para el funcionamiento esperado del teléfono, es mucho más prioritario decodificar un paquete de voz incorporado en un mensaje de radiofrecuencia durante una conversación telefónica —interrupción solicitada por el receptor GSM—, que responder a un cambio en la orientación del aparato —interrupción solicitada por el giroscо-

pio—, lo que se consigue asignando una mayor prioridad a la petición de interrupción generada por el receptor GSM que a la generada por el giroscopio. Las prioridades se suelen asociar, al igual que ocurría con los vectores de interrupciones, al número de petición de interrupción. Cuando varios dispositivos comparten el mismo número de petición de interrupción, la priorización entre ellos vendrá dada por el orden en el que la rutina de tratamiento correspondiente verifique cuáles de ellos han solicitado atención.

La siguiente cuestión a considerar son las acciones que deberá realizar el hardware del procesador, una vez configurado el sistema, cuando reciba una petición de interrupción y tras completar la ejecución de la instrucción en curso. En primer lugar, deberá guardar el valor actual del contador de programa, para así poder continuar posteriormente con la ejecución de la instrucción siguiente a la que acaba de ejecutar. Esta dirección de retorno se almacenará por regla general en un registro especial del sistema. En segundo lugar, en aquellos procesadores que dispongan de varios modos de ejecución, el procesador cambiará al modo de ejecución privilegiado. En tercer lugar, deshabilitará las interrupciones. Finalmente, el procesador cargará en el contador de programa, de alguna de las formas vistas anteriormente, la dirección de la rutina de tratamiento de interrupción correspondiente, y comenzará su ejecución.

A continuación, se aborda la estructura de las rutinas de tratamiento de interrupciones. El código de una de estas rutinas estará formado por las siguientes partes. La primera parte se encargará de almacenar, generalmente en la pila, aquellos registros que vayan a ser modificados por la rutina. Esto es necesario puesto que el tratamiento de una interrupción no debe afectar a la ejecución del programa en curso, así pues, la rutina deberá programarse de tal forma que al completarse, el procesador retome la ejecución del programa interrumpido con el mismo estado que el que tenía al finalizar la ejecución de la última instrucción ejecutada. En el caso de que el código de la rutina vaya a permitir que interrupciones más prioritarias sean atendidas durante su ejecución, también preservará en la pila su dirección de retorno. La siguiente parte de la rutina se encargará de averiguar cuál de entre los dispositivos conectados a su número de interrupción ha solicitado ser atendido y por qué. A continuación, vendrá el código necesario para la gestión de un dispositivo concreto —que se suele programar para que consuma el menor tiempo posible—. Al finalizar esta parte, la rutina se encargará de recuperar el valor original de aquellos registros que haya modificado —los que habían sido preservados al inicio de la rutina—. Por último, concluirá con una instrucción especial —normalmente llamada de retorno de excepción—, ya que de esta forma, cuando el procesador ejecute esta instrucción, actualizará el contador de programa con la dirección de retorno y volverá

al modo de ejecución original —en el caso de que el procesador disponga de varios modos de ejecución—.

Por último, conviene saber que 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 incluso puede disponer de bancos de registros de respaldo para no modificar los del usuario durante la ejecución de la rutina de tratamiento de interrupciones. Este es el caso de la arquitectura ARM, en la que además, y debido a las acciones que realiza de forma automática un procesador que siga esta arquitectura, las rutinas de tratamiento de interrupción se programan prácticamente igual que las subrutinas.

En los párrafos anteriores se han visto las consideraciones que se deben tener en cuenta a la hora de diseñar e implementar un sistema que gestione la entrada/salida mediante interrupciones, que concisamente serían:

1. Diseñar el hardware y conexionado necesario para que los dispositivos señalen las interrupciones e indiquen el número de interrupción que van a señalar, lo que muchas veces se realiza añadiendo un circuito controlador de interrupciones.

2. Decidir si las interrupciones se señalan por nivel o por flanco.

3. Establecer los vectores de interrupción, su uso y las direcciones de comienzo de las rutinas de tratamiento asociadas.

4. Configurar el sistema durante su arranque para habilitar las interrupciones requeridas y asignarles prioridades.

5. Implementar las acciones que llevará a cabo el procesador cuando reciba una interrupción para atenderla y para volver a su estado normal al terminar su gestión.

6. Diseñar adecuadamente el código de las rutinas de tratamiento de interrupción.

Resumen

En este apartado hemos visto que si el procesador y los dispositivos incorporan el hardware necesario, pueden ser los dispositivos los que avisen al procesador de que necesitan su atención por medio de solicitudes de interrupción. Utilizando este mecanismo, aunque el procesador y los dispositivos deben contar con hardware más complejo, es posible atender a los dispositivos con una latencia mínima y los programas pueden diseñarse de forma independiente de la entrada/salida. También se

ha visto que para poder gestionar interrupciones —o excepciones, que generalizan el concepto de interrupción—, el procesador debe ser capaz de: I) habilitar qué dispositivos pueden interrumpirle; II) establecer prioridades en las interrupciones; III) detectar que se ha solicitado una interrupción, ya sea por nivel o por flanco; IV) saber dónde comienza cada una de las rutinas de tratamiento por medio de los vectores de interrupción; V) interrumpir el flujo de ejecución de instrucciones, saltando, una vez finalice la ejecución de la instrucción en curso, a la rutina de tratamiento de interrupciones correspondiente y retornando, una vez completada la rutina de tratamiento, a la siguiente instrucción del programa interrumpido; y VI) preservar su estado para poder continuar la ejecución del programa interrumpido como si la interrupción no hubiera tenido lugar.

▶ **10.2** El proyecto «`PIO_int`» —en la carpeta «`3. Interrupciones`» de la colección de ejercicios para Arduino— realiza las siguientes acciones. En primer lugar, configura el pin al que está conectado el pulsador de la tarjeta de E/S como entrada con *pull-up*. En segundo lugar, permite que la señal generada pulsando y soltando el pulsador se utilice como una señal de petición de interrupción. En tercer lugar, permite configurar el modo y la polaridad que determinarán ante qué variación o estado de esta señal se producirá una petición de interrupción. Por último, proporciona una rutina de tratamiento de interrupciones del puerto PIOB que se encarga de incrementar un contador global en el caso de que haya sido llamada debido a la señal obtenida del pulsador.

Abre este proyecto en el entorno Arduino y realiza los siguientes ejercicios:

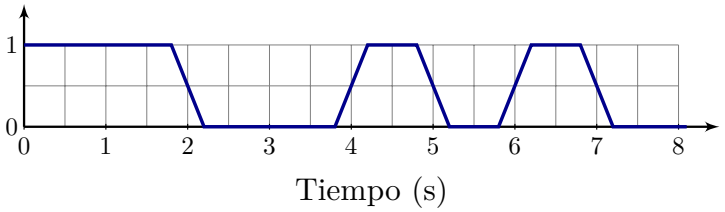
10.2.1 Sustituye el fichero «`PIOB_pushbutton_setup.s`» por el completado en el Ejercicio 10.1.

10.2.2 Completa el fichero «`PIOB_pushbutton_setup_int.s`» para que configure adecuadamente, en función de los parámetros pasados a «`PIOB_pushbutton_setup_int`», el modo y la polaridad que se utilizarán para detectar que se ha solicitado una interrupción.

10.2.3 Localiza la rutina en ensamblador que incrementa la variable global «`PIOint`», ¿qué nombre tiene dicha rutina?, ¿se llama a esa rutina desde alguna parte del proyecto?, ¿cuándo se llamará a dicha rutina?, ¿quién se encarga de llamarla?

10.2.4 Compila y sube el proyecto a la tarjeta Arduino Due. Abre el monitor serie, ¿qué hace el programa?, ¿bajo qué condiciones se incrementa el número de interrupciones detectadas?

10.2.5 Suponiendo que se ha pulsado y soltado el pulsador de la tarjeta de E/S de tal forma que se hubiera generado una señal de petición de interrupción como la mostrada a continuación, indica en qué momentos el puerto PIOB habría enviado peticiones de interrupción al procesador.



10.2. Transferencia de datos por programa y mediante acceso directo a memoria

Como ya se ha visto, la transferencia de datos entre el procesador y los dispositivos se realiza a través de los registros de datos de los controladores de los dispositivos. El procesador, siguiendo las instrucciones de un programa, leerá estos registros para obtener datos del dispositivo —si el dispositivo es de entrada— o escribirá en estos registros para enviar datos al dispositivo —si el dispositivo es de salida—. Sin embargo, los datos leídos desde un dispositivo no suelen ser procesados en el mismo momento en el que se leen, más bien suelen guardarse temporalmente en memoria, a la espera de que la ejecución de un programa los requiera. De igual forma, los datos que se quieren enviar a un dispositivo no se suelen transmitir de forma inmediata, más bien, se suelen almacenar temporalmente en memoria, a la espera de que el dispositivo vaya requiriéndolos. Estos almacenes temporales reciben el nombre de **buffers** y permiten compensar las diferencias que puedan haber entre la velocidad del procesado de los datos y la velocidad de transferencia del dispositivo. Así pues, la mayoría de las transferencias de datos se realizará en un primer momento entre un controlador de dispositivo y memoria, o viceversa —el procesador, o el controlador del dispositivo, leerán posteriormente estos datos de memoria, bien para procesarlos o para enviarlos—.

La transferencia de datos entre un dispositivo y memoria, o viceversa, puede realizarse de dos formas. La primera de ellas, denominada

transferencia de datos por programa, consiste en que el procesador transfiere los datos uno a uno ejecutando instrucciones de lectura y de escritura. Así, para transferir cada uno de los datos obtenidos de un dispositivo a memoria, el procesador ejecutaría una instrucción de lectura del dispositivo seguida de una de escritura en memoria. Análogamente, para transferir cada uno de los datos de memoria a un dispositivo, el procesador ejecutaría una instrucción de lectura de memoria seguida de una de escritura en el dispositivo.

La transferencia de datos por programa es una forma sencilla de mover datos entre un dispositivo y memoria, o viceversa, se adapta a la estructura de un computador vista hasta ahora y puede utilizarse perfectamente para transferir datos de un teclado, un ratón u otros dispositivos que tengan una productividad de pocos bytes por segundo, puesto que estas transferencias consumirán muy poco tiempo del procesador. Sin embargo, en otras aplicaciones, la transferencia de datos por programa no es una opción muy eficiente. Consideremos por ejemplo el caso de que se quiera reproducir 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 información tanto a la tarjeta gráfica como a la tarjeta de sonido. Todos los bloques de datos que se transfieren son ahora de gran tamaño, y el procesador no debe únicamente copiarlos del disco duro a la memoria, sino también decodificarlos, extrayendo el vídeo y el audio, y enviarlos por separado a los dispositivos adecuados. Así pues, el procesador deberá ser capaz de generar y enviar no menos de 24 imágenes de unos 6 megabytes por segundo a la tarjeta de vídeo y unos 25 kilobytes de audio por segundo a la tarjeta de sonido. Todo ello en tiempo real. Aunque un procesador actual de potencia media o alta es capaz de realizar estas acciones sin problemas, en el tiempo adecuado, una buena parte de su tiempo de procesamiento lo invertiría en mover datos desde el disco a la memoria y de esta a los dispositivos de salida, sin poder realizar algún otro trabajo más productivo. En un sistema multitarea, con muchas aplicaciones compitiendo por el uso del procesador y posiblemente también leyendo y escribiendo en disco, que el procesador se dedique a realizar transferencias de grandes bloques de datos supone consumir gran parte de su tiempo en una tarea repetitiva y monótona, impropia del procesador.

Así pues, la segunda forma de transferir datos entre un dispositivo y memoria, o viceversa, llamada **acceso directo a memoria**, o **DMA** —*Direct Memory Access*, en inglés—, se ideó para liberar al procesador de la tarea de transferir grandes bloques de datos, solucionando así el problema descrito en el párrafo anterior. Esta técnica consiste en que un dispositivo especializado del sistema, llamado **controlador de DMA**, se encargue de realizar los movimientos de datos. El controlador de DMA es capaz de copiar datos desde un dispositivo de entrada/salida a la me-

moria o de la memoria a los dispositivos de entrada/salida, aunque hoy en día también es frecuente que sean capaces de transferir datos entre distintas zonas de memoria o de unos dispositivos de entrada/salida a otros. Es más, un controlador de DMA suele disponer de varios canales, lo que le permite realizar varias transferencias de forma simultánea: cada canal se encarga de gestionar una transferencia de datos entre una pareja de dispositivos. En cualquier caso, para que el controlador de DMA pueda realizar transferencias de datos, es necesario que tenga acceso a los buses, gestionados por el procesador, a través de los cuales se transfieren los datos. Por ello, los controladores de DMA deben ser capaces de solicitar el uso de estos buses y de participar activamente en los procesos de arbitraje necesarios para convertirse en maestros de estos buses. Por otra parte, para que el uso del DMA sea realmente eficaz, cuando el controlador de DMA esté realizando transferencias, el procesador deberá seguir teniendo acceso a los recursos necesarios para poder continuar ejecutando instrucciones, es decir, se debe evitar que el procesador se quede en espera por no tener acceso a estos buses. Aunque no vamos a entrar en detalles, técnicas tradicionales como el **robo de ciclos** permiten compaginar sin demasiada sobrecarga los accesos al bus del procesador y del controlador de DMA. Además, hoy en día, la presencia de memorias caché y el uso de sistemas complejos de buses independientes, permiten 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. El controlador de DMA proporciona un conjunto de registros de control para cada uno de sus canales en los que el procesador puede escribir para indicarle qué datos debe transferir por ese canal. En particular, el procesador deberá indicar al controlador de DMA para cada transferencia: I) el dispositivo de origen, II) el dispositivo de destino, III) la dirección de inicio de los bloques a copiar, tanto en origen como en destino, y IV) la cantidad de datos a copiar. Una vez el controlador de DMA ha completado la transferencia de datos, almacenará su resultado —transferencia errónea o correcta— en uno de sus registros de estado, y si ha sido programado para ello, generará una petición de interrupción para avisar al procesador de que ha terminado la transferencia que le había sido encomendada.

Los controladores de DMA actuales son muy potentes y versátiles. Es normal que puedan comunicarse con ciertos dispositivos de entrada/salida para esperar a que estos tengan datos disponibles y, en cuanto los tengan, copiarlos a memoria, siendo capaces de repetir este proceso tantas veces como sea necesario hasta haber obtenido la cantidad de datos que se haya indicado. Esta característica suele utilizarse, por ejemplo, para librar al procesador de tener que leer uno a uno los datos

obtenidos por un conversor analógico-digital, ahorrándose los tiempos de espera inherentes a la conversión. También es frecuente que permitan que una sola orden del procesador encadene múltiples transferencias distintas. Esto se consigue utilizando estructuras de datos enlazadas que se almacenan en memoria. Cada una de estas estructuras contiene la dirección de origen, de destino, la cantidad de datos a copiar y la dirección de la siguiente estructura que forma parte de la cadena. De esta manera, se puede indicar al controlador de DMA con una única orden que lea datos de diversos *buffers* de un dispositivo y los copie en otros tantos pertenecientes a distintas aplicaciones, respondiendo así a una serie de llamadas al sistema realizadas desde diferentes aplicaciones.

Resumiendo, en este apartado se ha descrito, por una parte, la *transferencia de datos por programa*, que es la forma más sencilla de copiar datos entre los dispositivos de entrada/salida y la memoria, o viceversa, y en la que el procesador simplemente va ejecutando las correspondientes instrucciones de lectura y escritura. Por otra parte, se ha presentado la *transferencia de datos mediante acceso directo a memoria*, que libera de esa tarea al procesador, así como las principales características de los dispositivos necesarios para emplear esta técnica: los controladores de DMA.

▶ **10.3** El proyecto «`DMA_test`» —en la carpeta «`4. DMA, PWM y USB`» de la colección de ejercicios para Arduino— programa el controlador de DMA incorporado en el microprocesador para realizar una transferencia de un bloque de 2 048 palabras desde una zona de memoria a otra. También lo programa para que genere una interrupción cuando haya finalizado la copia. Una vez activado el controlador de DMA y mientras este realiza la transferencia, el procesador del Arduino Due ejecuta un bucle en el que va incrementando un contador. Cuando el controlador de DMA acaba, se muestra el valor del contador y el tiempo que ha tardado el controlador de DMA en realizar la copia.

10.3.1 Compila el proyecto, súbelo a la tarjeta Arduino Due y activa el monitor serie. Antes de pulsar el pulsador de la tarjeta de E/S, contesta a las siguientes preguntas. El bloque de 2 048 palabras que se quiere copiar, ¿en qué dirección de memoria se encuentra?, ¿a qué dirección de memoria se va a copiar?

10.3.2 Pulsa ahora el pulsador de la tarjeta de E/S, ¿cuántas veces se ha incrementado el contador «`cnt`»?, ¿cuánto tiempo ha tardado el controlador de DMA en realizar la copia?

10.3.3 Una vez que se ha indicado al controlador de DMA que inicie la copia, el procesador sigue ejecutando instrucciones, en concreto, el siguiente bucle: «`while(!DMADone) {cnt++;}`». Es decir, ejecutará «`cnt++;`» («`cnt = cnt + 1;`») una y otra vez mientras la variable «`DMADone`» valga 0. Puesto que esta variable no se modifica dentro del bucle, aparentemente estamos ante un bucle infinito. Sin embargo, en otra parte del código, la llamada a una función pondrá la variable «`DMADone`» a 1 y el bucle terminará. Localiza dicha función en el código fuente. ¿Qué nombre tiene esa función? ¿Qué es lo que provoca que dicha función se ejecute?

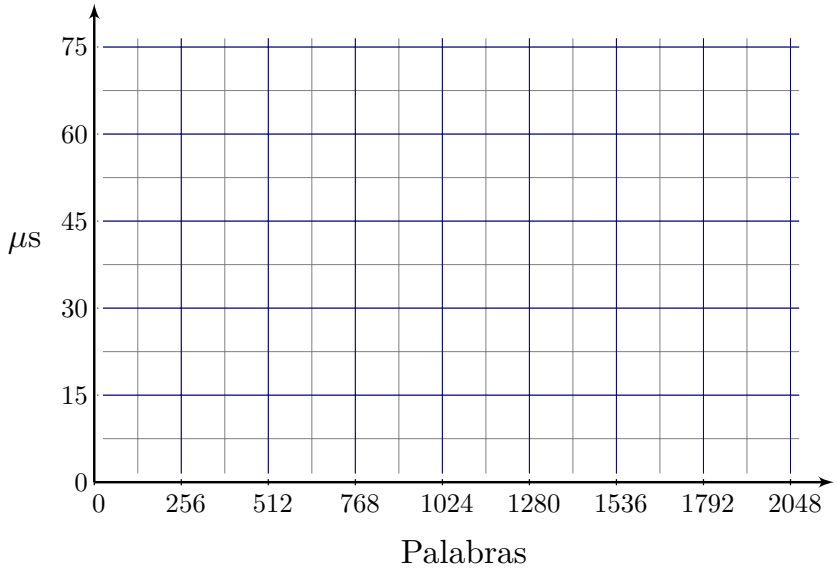
10.3.4 Una forma de calcular las características temporales de un sistema de comunicación —latencia y productividad— es midiendo el tiempo que conlleva realizar transferencias de distintos tamaños y utilizar algún método, p.e., el de mínimos cuadrados, para obtener los parámetros de la recta, $y = a + b \cdot x$, que mejor se ajuste a los tiempos obtenidos. Una vez obtenidos los parámetros a y b, la latencia será igual a a —el tiempo inicial constante e independiente del tamaño de la transmisión—, y la productividad máxima, en la que no se tiene en cuenta la latencia, será $\frac{1}{b}$ —la inversa de la pendiente de la recta—. En este ejercicio no se van a calcular dichos valores, simplemente se van a obtener tiempos para varios tamaños, que sería el primer paso para poder hacerlo. Si quieres calcular, a partir de estos tiempos, la latencia y la productividad máxima entre DMA y memoria del microcontrolador ATSAM3X8E, puedes realizar más adelante el ejercicio adicional 10.18.

Volviendo a este ejercicio, busca en el código la declaración de la constante «`SIZE`», «`#define SIZE 2048`», que se utiliza para especificar el tamaño, en número de palabras, del bloque de memoria que debe transferirse —2 048 es el máximo de palabras que puede transferir por bloque el controlador de DMA integrado en el ATSAM3X8E—. Modificando dicha declaración y ejecutando de nuevo el proyecto tantas veces como sea necesario, rellena la tabla del Cuadro 10.1 y representa los valores que has obtenido en la gráfica del mismo cuadro.

10.3.5 Los puntos que has representado, ¿podrían estar todos aproximadamente en una misma recta? Obtén algún valor adicional, ¿podría estar también en dicha recta?

Cuadro 10.1: Tiempo de una transferencia DMA en función del tamaño de los datos transferidos

SIZE	Tiempo (μs)
512	
1 024	
1 536	
2 048	



10.3. Estandarización y extensión de la entrada/salida: buses y controladores

Como se ha visto, la entrada/salida es un componente indispensable de los ordenadores, puesto que les permite comunicarse con el mundo exterior. Pero además, la tendencia actual es que los sistemas informáticos estén cada vez más orientados a la entrada/salida. Términos como *dispositivos conectados* o *el internet de las cosas* hacen referencia al uso de pequeños ordenadores empotrados con capacidad de comunicación para que constantemente midan parámetros a través de un conjunto de sensores y realicen actuaciones sobre el sistema al que están unidos. La comunicación, la lectura de sensores y el uso de actuadores se consigue en estos equipos gracias a la entrada/salida. Por su parte, el variado uso que se da a los ordenadores personales —tratamiento de textos, punto de venta, reproducción multimedia, juego, dibujo...— hace que estén conectados de forma habitual a dispositivos de entrada/salida muy variados: impresoras, escáneres de códigos de barras, subsistemas de audio,

mandos de juego, tabletas digitalizadoras... Así pues, tanto los sistemas empotrados como los ordenadores personales se comunican actualmente con una gran cantidad de dispositivos de entrada/salida. Y la tendencia es que esto aumente en el futuro.

En estas circunstancias, no es viable que cada nuevo dispositivo requiera de una conexión propia al bus del sistema —el que conecta el procesador con la memoria principal, en una visión simplificada de los ordenadores, y que además se encuentre en el interior del propio ordenador—. Tampoco es una opción válida el llevar este bus al exterior del ordenador, ya que esto supone problemas tanto mecánicos como electrónicos que, si bien es cierto que se pueden resolver, impactarían negativamente en la productividad del bus y, sobre todo, incrementarían sustancialmente el coste de los dispositivos y los conectores. Así pues, la opción adoptada desde hace años por la industria ha sido la de desarrollar buses específicos para la entrada/salida, diseñados para conectar dispositivos de entrada/salida al ordenador a través de ellos. Estos buses de entrada/salida se han diseñado siempre buscando un compromiso entre su coste económico y las prestaciones que permitían alcanzar. En el mundo de los ordenadores personales, el bus que actualmente se ha hecho casi en exclusiva con el mercado es el **Universal Serial Bus** o **USB**, en sus diferentes versiones. Otros buses utilizados en los ordenadores personales son los puertos serie RS-232 y RS-485, el puerto paralelo Centronics, SCSI, FireWire y Thunderbolt. En el mundo de los microcontroladores y los sistemas empotrados, se utilizan indistintamente el bus SPI y el bus I2C.

Cuando se conecta un dispositivo a un computador utilizando un bus de entrada/salida, aquel no existe como tal para los componentes del computador conectados al bus del sistema. Una impresora USB, o un disco duro USB, no pueden enviar una solicitud de interrupción al procesador —¿cómo podrían activar una de las líneas de petición de interrupción del procesador si están conectados a través de un cable USB estándar?—. Tampoco es posible que el controlador de DMA transfiera datos entre un dispositivo USB y los otros. En realidad, para el procesador y los otros componentes conectados al bus del sistema, el único dispositivo de entrada/salida que existe es el controlador raíz USB, o USB Host. Se trata, este sí, de un dispositivo conectado al bus del sistema —al PCI Express en los PC actuales—, que puede generar peticiones de interrupción y al que el controlador DMA puede acceder para realizar transferencias de datos. El procesador, o el controlador de DMA, solo se comunican con el USB Host, y es este el que se encarga de enviar y recibir bloques de datos para comunicarse con el dispositivo de entrada/salida siguiendo el protocolo USB.

El sistema de entrada/salida es mucho más complejo en un ordenador real de lo hemos visto en este libro de introducción a la arquitectura de

computadores. Cada dispositivo de entrada/salida no solo es gestionado a bajo nivel, sino que además requiere de una serie de reglas de comunicación específicas a otros niveles que estandarizan, por ejemplo, la forma de comunicarse con los dispositivos de almacenamiento como discos duros, Blu-Ray, DVD, etcétera —p.e., el estándar SATA *(Serial Advanced Technology Attachment)*—, o con los dispositivos de interfaz con humanos como teclados, ratones, mandos de juego, etcétera —p.e., el estándar HID *(Human Interface Device)*—. En un sistema real, toda la comunicación con los dispositivos es gestionada por lo que se conoce como **controladores software de dispositivos** o *drivers*, que son módulos del sistema operativo que siguen las mencionadas reglas para garantizar que la comunicación con los dispositivos sea la adecuada. De hecho, los sistemas operativos actuales tienen en cuenta tanto la estandarización de la entrada/salida como la diversidad de dispositivos de entrada/salida. Por ello, los controladores software de dispositivos se estructuran en niveles, permitiendo de esta forma gestionar adecuadamente todos los casos, desde las reglas generales de comunicación —compartidas por todos los dispositivos de una misma clase— hasta las comunicaciones de bajo nivel —más específicas de cada dispositivo—. Así, para gestionar la entrada/salida de un ratón USB y de un ratón Bluetooth, se utilizará el mismo controlador software de alto nivel, que entiende el conjunto estándar de reglas HID. Este controlador software de alto nivel, se comunicará, según el caso, con otro de más bajo nivel que permita comunicarse con el ratón USB conectado físicamente al bus USB a través del dispositivo USB Host, o con otro que permita comunicarse de forma inalámbrica con el ratón Bluetooth a través del dispositivo Bluetooth. En la práctica, para cada dispositivo pueden llegar a utilizarse tres o más controladores software jerarquizados que terminan permitiendo, de forma eficaz, la comunicación entre el ordenador y el dispositivo.

Hemos visto en este apartado cómo la entrada/salida está cada vez más presente en los sistemas actuales y que esta tendencia sigue creciendo. Que por ello es necesario utilizar buses estándar para conectar los dispositivos de entrada/salida, como el bus USB en los PC o los buses SPI e I2C en los sistemas empotrados. Esta complejidad física añadida complica la gestión de la entrada/salida, que los sistemas operativos resuelven eficientemente mediante jerarquías de controladores software, que abordan tanto la estandarización como la diversidad de dispositivos de entrada/salida.

▶ **10.4** Abre el proyecto «`USB_keyboard_mouse`» —en la colección de ejercicios para Arduino, carpeta «`4. DMA, PWM y USB`»—, compílalo, súbelo a la tarjeta Arduino Due y realiza los siguientes ejercicios:

10.4.1 Antes que nada, abre un editor de textos, ponlo a pantalla completa y sitúa el cursor del ratón en el centro de la pantalla. Una vez hecho esto, sigue los pasos indicados a continuación, contestando a lo que se pregunta.

a) Conecta la tarjeta Arduino Due al computador utilizando el puerto USB de comunicaciones —en lugar de el de programación—. A partir de este momento la tarjeta Arduino Due será un periférico del ordenador. ¿Observas algún cambio al mirar la pantalla del ordenador?, ¿cuál?, ¿qué color muestra el led RGB?

b) Pulsa una vez el pulsador de la tarjeta de E/S, ¿qué ha ocurrido?, ¿qué ha cambiado en el ordenador y en el led RGB?

c) Vuelve a pulsar el pulsador, ¿qué ha ocurrido ahora?, ¿qué ha sucedido en el ordenador?, ¿qué color muestra el led RGB?

10.4.2 Consulta la ayuda de Arduino para ver cómo generar un clic del ratón. ¿A qué función tendrías que llamar para que la tarjeta enviara un clic de ratón al ordenador?

10.4.3 Modifica el programa «`USB_keyboard_mouse`» para que no mueva el ratón —comenta las líneas pertinentes— y para que al pulsar el botón del pulsador de la E/S, en lugar de pasar a comportarse como un teclado, continúe actuando como un ratón y envíe un clic de ratón al computador. De esta forma, cada vez que se pulse el pulsador de la tarjeta de E/S se debería enviar un clic al ordenador.

Comprueba que efectivamente, el pulsador de la tarjeta de E/S, gracias al código que se está ejecutando en la tarjeta Arduino Due, se está comportando como el botón izquierdo de un ratón.

10.4.4 Sabiendo que no hay una función que genere un doble clic, ¿cómo podrías generar uno?

10.4.5 Por último, si el sistema operativo que estás utilizando realiza acciones distintas al hacer doble clic que al hacer dos clics, modifica de nuevo el código para que con una pulsación del pulsador de la tarjeta de E/S se envíe un doble clic. Comprueba que funcione correctamente.

10.4. Microcontroladores y conversión digital/analógica

El uso extendido de los microprocesadores para la automatización y control de todo tipo de máquinas, electrodomésticos, dispositivos multimedia, herramientas, etcétera, ha dado lugar a un tipo específico de microprocesador, denominado **microcontrolador**, que se caracteriza principalmente por incorporar, además de un procesador, una serie de dispositivos periféricos, con el propósito de simplificar su uso como controlador maestro de un **sistema empotrado** —computador de uso específico integrado en un dispositivo como los anteriormente nombrados—. La variedad de dispositivos integrados en un microcontrolador es enorme, tanto más cuanto más específico es el microcontrolador, aunque habitualmente suelen poder agruparse en dispositivos de gestión de las comunicaciones, de gestión del almacenamiento y de interacción con el mundo exterior.

Dado que los procesadores trabajan exclusivamente con información digital —bits, bytes, palabras— y nuestro mundo es analógico —presenta magnitudes continuas como la longitud, peso, potencial eléctrico, intensidad luminosa, etcétera—, existe la necesidad de convertir un tipo de información en otra, en un par de procesos que se denominan **conversión Analógica/Digital** (A/D) y **conversión Digital/Analógica** (D/A) respectivamente. La mayoría de microcontroladores incorporan dispositivos de entrada/salida que permiten realizar dichas conversiones.

Una de las formas más simples que existe para la conversión D/A, la que consiste en producir una señal analógica a partir de una secuencia de valores digitales, es la **modulación por ancho de pulsos** o **PWM** —por las siglas en inglés de *Pulse Width Modulation*—. La modulación por ancho de pulsos consiste en generar una señal cuadrada de periodo fijo y ciclo de trabajo variable. Es decir, mientras que el tiempo entre dos flancos de subida o de bajada consecutivos de la señal cuadrada —su periodo— permanece fijo, el porcentaje del periodo que la señal permanece a nivel alto —el ciclo de trabajo— se puede variar. De esta forma, al modificar el ciclo de trabajo en función de un valor digital, se puede generar una cantidad de energía proporcional a ese valor, que al pasar por un **filtro pasa-bajo**, o **LPF** (de *Low Pass Filter*) —un circuito electrónico que deja pasar las bajas frecuencias y elimina las altas—, se convertirá en una tensión fija y proporcional a la cantidad de energía generada y, por tanto, al valor digital que determinó el ciclo de trabajo. Esta técnica se ilustra en la Figura 10.3, donde se puede observar, de arriba a abajo, como conforme aumenta el ciclo de trabajo

de la señal generada por un generador de funciones, aumenta la tensión medida en un voltímetro.

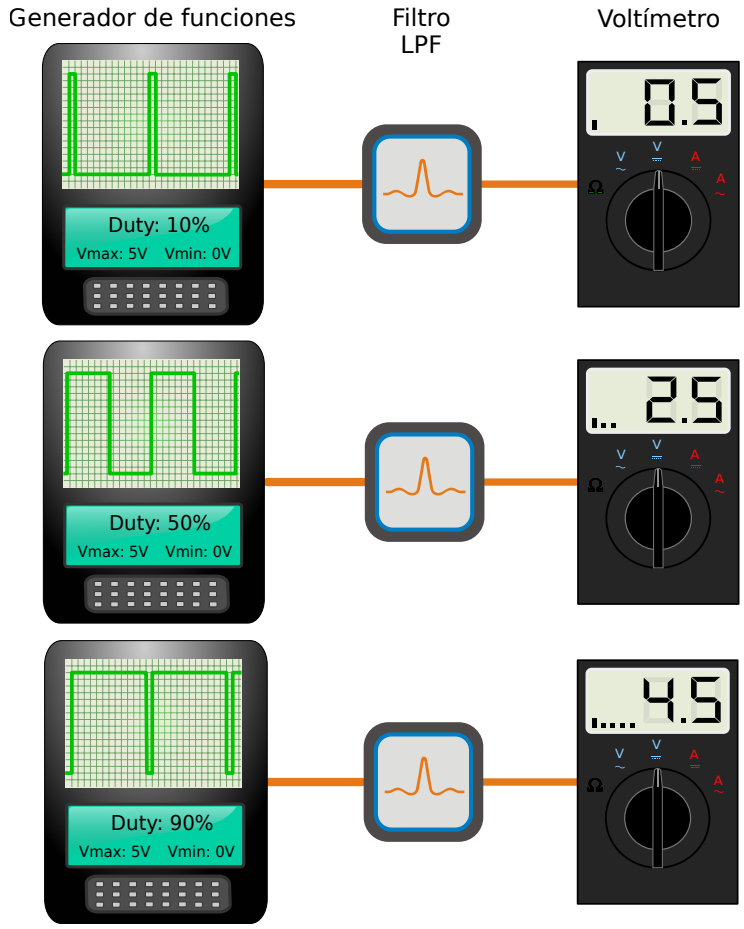


Figura 10.3: Método PWM para la conversión Digital/Analógica

▶ **10.5** Teniendo en cuenta lo comentado en este apartado, ¿cómo se podría variar la intensidad luminosa del led utilizando PWM? Dibuja una señal PWM que conseguiría que la intensidad del led fuera alta y otra señal PWM que conseguiría que la intensidad del led fuera baja.

En caso necesario, puedes obtener más información sobre PWM desde la ayuda del entorno Arduino. Selecciona la entrada de menú «`Ayuda`» ▷ «`Referencia`» y haz clic sobre «`analogWrite()`». Al final de la página de ayuda de «`analogWrite()`», haz clic sobre el enlace «`Tutorial:PWM`».

▶ **10.6** El proyecto «`PWM_LED_rojo`» —en la carpeta «`4. DMA, PWM y USB`» de la colección de ejercicios para Arduino— permite variar la luminosidad del led rojo de la placa de la E/S utilizando la técnica PWM. Con cada pulsación del pulsador de la tarjeta de E/S, la luminosidad del led disminuirá hasta llegar a apagarse. Una vez apagado el led, la siguiente pulsación volverá a encenderlo. También permite conseguir este mismo efecto manteniendo el pulsador apretado. En este caso, la luminosidad variará cada medio segundo.

Compila el proyecto, súbelo a la tarjeta Arduino Due y abre el monitor serie. Observa que con cada pulsación del pulsador —o cada medio segundo si se mantiene pulsado— se muestra en el monitor serie el valor del nuevo ciclo de trabajo y la intensidad del led rojo cambia. ¿Qué relación hay entre la variación del ciclo de trabajo y la de la intensidad del led? Dicho de otra forma, cuando el ciclo de trabajo aumenta, ¿la intensidad del led aumenta o al revés? ¿A qué es debido?

10.5. Ejercicios

▶ **10.7** Modifica el proyecto «`pulsador`» que habías completado en el Ejercicio 10.1, para que la subrutina «`pulsador`» en lugar de retornar inmediatamente tras pulsarse el pulsador, espere hasta que este se haya soltado. Esto es, la subrutina deberá comprobar que se ha pulsado el pulsador —lo que ya hace— y que después de haberse pulsado, se ha soltado.

▶ **10.8** Modifica el código del proyecto «`PIO_int`» completado en el Ejercicio 10.2, para ir probando todas las posibles combinaciones de los parámetros «`mode`» y «`polarity`» de la llamada a la función «`PIOB_pushbutton_setup_int()`». Conforme vayas probando las distintas combinaciones, completa el siguiente cuadro, indicando para cada caso, cuál es el comportamiento del programa y bajo qué circunstancias se detecta la interrupción.

Mode	Polarity	Efecto
FLANCO	BAJO	
	ALTO	
NIVEL	BAJO	
	ALTO	
CAMBIO	BAJO	
	ALTO	

▶ **10.9** Continuando con el proyecto «PWM_LED_rojo» —carpeta «4. DMA, PWM y USB»—, ve desplazando la tarjeta con suavidad describiendo un arco de unos 60 centímetros a una velocidad moderada de tal forma que puedas observar si el led está encendido o apagado. Manteniendo la vista fija en un punto, observa el patrón de encendidos y apagados del led para diferentes valores del ciclo de trabajo. Explica cómo se relacionan estos patrones con el principio de funcionamiento del PWM.

10.5.1. Ejercicios de nivel medio

▶ **10.10** Completa el proyecto «RTC_actualiza_fecha_hora», en la carpeta «2. Consulta de estado» de la colección de ejercicios para Arduino, para que configure el reloj en tiempo real (RTC) con la fecha y hora actuales en el modo de 12 horas. El programa, además de configurar el RTC, deberá comprobar que la configuración se ha completado correctamente. Puedes consultar el procedimiento para actualizar la fecha y hora del RTC, incluyendo la forma de comprobar si se ha hecho correctamente, en el Apartado A.4.4.

▶ **10.11** Cuando el proyecto «RTC_alarma_int» —que está en la carpeta «3. Interrupciones» de la colección de ejercicios para Arduino— se complete, este programará el reloj en tiempo real, definirá una alarma y detectará, mediante interrupciones, que la alarma ha saltado. Por el momento, en este ejercicio tan solo tienes que hacer lo siguiente:

10.11.1 Sobrescribe el fichero «actualiza_fecha_hora.s» con el que ya editaste en el Ejercicio 10.10.

10.11.2 Completa el fichero «configura_alarma_int.s» para que defina una alarma en la fecha y hora indicada en sus comentarios y para que active la generación de interrupciones del RTC para cuando salte la alarma.

▶ **10.12** Realiza los siguientes ejercicios relacionados con el proyecto «`PWM_LED_rojo`» —carpeta «`4. DMA, PWM y USB`»—:

10.12.1 Es posible que al realizar los ejercicios 10.6 o 10.9 hayas observado que además de disminuir la intensidad del led también se hacía apreciable un ligero parpadeo. Este parpadeo está relacionado con el periodo de la señal PWM. Cuanto más grande sea ese periodo, más fácil será que nuestra vista se dé cuenta de que lo que varía no es la intensidad del led, es el tiempo que este está encendido o apagado.

El valor de este periodo se puede configurar en el controlador de PWM seleccionando qué fuente de reloj utilizar y qué divisor de frecuencia aplicar, así como variando el contenido del registro «`PWM_CPRD`», que multiplica el periodo de la señal anterior. Esto último es debido a que el registro «`PWM_CPRD`» sirve para indicarle al controlador de PWM, el número de periodos de la señal de reloj de entrada que formarán un periodo de la señal PWM.

Vamos a ver ahora cómo podemos visualizar mejor los encendidos o apagados, o dejar de percibir el parpadeo, modificando el contenido de dicho registro. Para ello, comienza cambiando en el fichero «`PWM_setup.s`», el valor de la constante «`CPRD`» —que es la utilizada para inicializar el registro «`PWM_CPRD`»— por el valor `0x0000 0FFC`. Al hacer esto, ¿habrá aumentado o disminuido el periodo de la señal PWM?

Sube el proyecto a la tarjeta Arduino y abre el monitor serie. Compara el funcionamiento actual conforme vas variando el ciclo de trabajo con el observado en los ejercicios 10.6 y 10.9. ¿Se aprecia el parpadeo?

A continuación, modifica de nuevo el valor de la constante «`CPRD`», pero ahora por el valor `0x0000 0FC`. Abre el monitor serie y compara el funcionamiento actual con el observado en los ejercicios 10.6 y 10.9. ¿Se aprecia el parpadeo?

10.12.2 Al igual que el ejercicio anterior se ha visto que es posible configurar la señal PWM para variar la intensidad de un led sin que aprecie ningún parpadeo, en este se verá que es posible modificar la intensidad de forma más gradual de como se ha hecho hasta ahora.

Para realizar este ejercicio es preferible que no se aprecie el parpadeo del led, por lo que antes de hacerlo, habrá que

modificar el valor de la constante «`CPRD`» tal y como se ha comentado en el ejercicio anterior.

Para indicarle al controlador de PWM cuál debe ser el ciclo de trabajo de la señal que genere, se debe escribir un número, comprendido entre 0 y el valor de «`PWM_CPRD`», en su registro «`PWM_CDTY`».

En este proyecto, la rutina que se encarga de incrementar el ciclo de trabajo es «`PWM_inc_cdty`». Esta rutina modifica el valor del ciclo de trabajo incrementando el contenido del registro «`PWM_CDTY`» en 1/4 del contenido del registro «`PWM_CPRD`». Esta división por cuatro se realiza con la instrucción «`lsr r3, r2, #2`», que desplaza 2 bits a la derecha el contenido de `r2`; en el que se ha había cargado previamente el contenido del registro «`PWM_CPRD`». Así pues, para conseguir una variación menor, p.e., de 1/16 del valor de «`PWM_CPRD`», bastaría con aumentar el número de bits del desplazamiento indicado en esa instrucción, p.e., a 4 para conseguir 1/16.

Reemplaza en el fichero «`PWM_inc_cdty.s`», la instrucción «`lsr r3, r2, #2`» por «`lsr r3, r2, #4`». Ejecuta el proyecto en la tarjeta Arduino y recuerda que en lugar de pulsar de forma repetida, puedes simplemente mantener pulsado el pulsador y el ciclo de trabajo se incrementará cada medio segundo. ¿Cada cuánto se va incrementando ahora el ciclo de trabajo mostrado en el monitor serie? ¿Se aprecian más o menos los cambios de intensidad? ¿Por qué?

Puedes probar también con «`lsr r3, r2, #5`».

10.5.2. Ejercicios de nivel avanzado

▶ **10.13** El proyecto «`RTC_alarma`» —en la carpeta «`RTC`» de la carpeta «`2. Consulta de estado`» de la colección de ejercicios para Arduino—, una vez completado, hará lo siguiente:

En primer lugar, en la etapa de inicialización, función «`setup()`»:

- Actualizará la fecha y hora del RTC y la mostrará.
- Configurará y mostrará una alarma para 5 segundos después de la fecha fijada en el paso anterior.

A continuación, durante el bucle de trabajo, función «`loop()`»:

- Mientras no detecte que ha saltado la alarma, hará parpadear al led azul.

- En cuanto detecte que ha saltado la alarma, encenderá el led rojo, la reactivará y mostrará el instante en el que se ha detectado su activación.

Realiza los siguientes ejercicios relacionados con este proyecto:

10.13.1 Sustituye el fichero «`actualiza_fecha_hora.s`» por el completado en el Ejercicio 10.10.

10.13.2 Completa el fichero «`configura_alarma.s`» para que configure la alarma del RTC a la fecha y hora indicada en este fichero.

10.13.3 Completa el fichero «`consulta_estado_alarma.s`» para que detecte por consulta de estado si ha saltado la alarma. Mientras no detecte que ha saltado, deberá hacer parpadear al led azul. En cuanto detecte que ha saltado la alarma, deberá encender el led rojo, reactivarla y volver.

10.13.4 Compila y sube el proyecto a la tarjeta Arduino Due. ¿Para cuándo se había configurado la alarma? ¿Cuándo se ha detectado?

10.13.5 En este proyecto, el tiempo que transcurre entre una lectura y la siguiente del registro «`RTC_SR`» vendrá dado principalmente por el tiempo que se mantiene encendido y apagado el led azul —es decir, por el tiempo de ejecución de las dos llamadas a la función «`delay()`»—. Esto es debido a que el tiempo que tarda el procesador del Arduino Due en ejecutar el resto de instrucciones que hay entre una lectura del registro «`RTC_SR`» y la siguiente, es despreciable en comparación al retardo que se le pasa a las llamadas a la función «`delay`». Teniendo en cuenta que cada llamada a «`delay()`» consume 300 ms, ¿cuál sería aproximadamente la latencia máxima entre que salta la alarma y se detecta que ha saltado?

10.13.6 Si cada llamada a «`delay()`» consumiera 2 200 ms, y sabiendo que la alarma está programada para saltar a los 5 segundos de su activación, calcula aproximadamente la latencia entre el salto de la alarma y su detección.

10.13.7 Modifica el fichero «`consulta_estado_alarma.s`» para que las llamadas a «`delay()`» consuman 2 200 ms cada una, compila y sube el proyecto a la tarjeta, ¿cuánto tiempo ha transcurrido entre que ha saltado la alarma y se ha detectado que ha saltado?, ¿coincide con la latencia que acababas de calcular?

▶ **10.14** Continuando con el proyecto «`RTC_alarma_int`», y teniendo en cuenta los cambios ya efectuados en el Ejercicio 10.11, realiza los siguientes ejercicios:

10.14.1 Edita el fichero «`RTC_Handler.s`» para terminar de completar la rutina de tratamiento de las interrupciones generadas por el RTC. Esta rutina comprobará que la causa de la interrupción del RTC ha sido la alarma, escribirá un 1 en la variable global «`alarma`» y, antes de volver, restablecerá los bits que indican a qué causa se ha debido la interrupción.

10.14.2 Completa el fichero «`consulta_estado_alarma_int.s`» para que mientras la variable «`alarma`» no valga 1, encienda y apague el led azul.

10.14.3 Compila y sube el programa a la tarjeta Arduino Due. Describe qué mensajes se muestran en el monitor serie y desde qué parte del código se genera cada uno de ellos.

10.14.4 En el fichero «`consulta_estado_alarma_int.s`», cambia el valor de la constante «`RETARDO`» por 2 200. Describe qué efecto tiene este cambio cuando se ejecuta el proyecto. Relaciona este resultado con el obtenido cuando en el ejercicio 10.13 se modificó esta constante. ¿Qué conclusiones puedes sacar sobre los métodos de consulta de estado e interrupciones?

▶ **10.15** Abre el proyecto «`PWM_LED_RGB`» —en la carpeta «`4. DMA, PWM y USB`» de la colección de ejercicios para Arduino—. Compílalo, súbelo a la tarjeta Arduino Due y completa lo siguiente:

10.15.1 Describe el funcionamiento del sistema: ¿de qué color es el led RGB cuando se inicia la placa?, ¿qué ocurre cuándo se pulsa el pulsador?

10.15.2 Examina el código fuente del fichero «`PWM_led_RGB.ino`», ¿cómo se generan los colores que irá mostrando el led?

10.15.3 Prueba diferentes valores para las constantes «`FACTRED`», «`FACTGRN`» y «`FACTBLU`». ¿Qué efecto tienen estos parámetros en el comportamiento del programa?

10.5.3. Ejercicios adicionales

▶ **10.16** Modifica el código del ejercicio 10.13 para que la alarma se active cada vez que los segundos del reloj lleguen a 30. Además, si habías cambiado el retardo de las funciones «`delay()`» a 2 200 ms,

vuelve a ponerlo a 300 ms. Compila y sube el proyecto a la tarjeta. ¿Salta una alarma cada minuto?

▶ **10.17** Completa el proyecto «`cambia`» —en la carpeta «`Pulsador`» de la carpeta «`2. Consulta de estado`» de la colección de ejercicios para Arduino—, para que con cada pulsación del pulsador encienda cíclicamente el led RGB con cada uno de sus tres colores básicos.

▶ **10.18** En el Ejercicio 10.3 se explicó cómo calcular la latencia y la productividad de un sistema de transmisión a partir de un conjunto de tiempos obtenidos para distintos tamaños. En dicho ejercicio se obtuvo el tiempo que le costaba al controlador de DMA transferir un bloque de 512, 1 024, 1 536 y 2 048 palabras de una zona de memoria a otra. Por lo tanto, estos tiempos podrían utilizarse para calcular la latencia y la productividad de la comunicación entre DMA y memoria del microcontrolador ATSAM3X8E. Teniendo en cuenta lo explicado en el Ejercicio 10.3 y que la copia de una palabra implica transferir 2 palabras (leer una palabra de una posición de memoria y escribirla en otra posición), calcula la latencia (en μs) y la productividad máxima (en bytes/μs) de la comunicación entre DMA y memoria del microcontrolador ATSAM3X8E.

▶ **10.19** Modifica el proyecto «`USB_keyboard_mouse`» —en la carpeta «`4. DMA, PWM y USB`» de la colección de ejercicios para Arduino—, para que:

10.19.1 en lugar de una elipse, el puntero del ratón describa un rectángulo;

10.19.2 permita el uso de letras acentuadas cuando se comporte como un teclado; o

10.19.3 cuando se pulse el pulsador de la tarjeta de E/S, envíe un atajo de teclado —p.e., la combinación «CTRL+ALT+l», que, al menos en KDE, bloquea la pantalla—.

Apéndice A

Información técnica y guía de uso de la tarjeta de E/S y del microcontrolador ATSAM3X8E

Índice

Este anexo proporciona información técnica de la tarjeta de E/S, utilizada en algunos de los ejercicios de entrada/salida, y de los siguientes dispositivos del microcontrolador ATSAM3X8E: los controladores de entrada/salida de propósito general, los dispositivos de temporización, el controlador NVIC —para la gestión de interrupciones— y el contro-

lador de DMA —para la transferencia de datos mediante acceso directo a memoria—.

A.1. La tarjeta de E/S

Algunos de los ejercicios propuestos en la parte del libro dedicada a la entrada/salida requieren el uso de una tarjeta de E/S que proporciona dos dispositivos básicos de entrada/salida: un pulsador y un led RGB —formado por tres ledes: uno rojo, otro verde y otro azul, en un mismo encapsulado—. El pulsador se utilizará como un dispositivo de entrada y los tres ledes, como dispositivos de salida.

Cuando se inserte la tarjeta de E/S en los conectores de expansión de la tarjeta Arduino Due, el pulsador se conectará entre el pin 13 y masa a través de una resistencia de protección —véase la Figura 9.8—. Dada esta configuración, para que el pin 13 pueda tomar ambos valores lógicos, será necesario que este pin se conecte a Vcc a través de una resistencia de *pull-up*. Una vez hecho esto, en el pin 13 habrá un `0` lógico cuando el pulsador esté pulsado y un `1` lógico cuando no. Puesto que el pin 13 está asociado al bit 27 del puerto PIOB del ATSAM3X8E —véase el Cuadro A.1—, para poder leer el estado del pulsador se deberá configurar el bit 27 del puerto PIOB como entrada y activar su resistencia de *pull-up*. Una vez configurado, cuando se lea el registro de datos del puerto PIOB, su bit 27 estará a 0 cuando el pulsador esté oprimido y a 1 cuando no.

Al insertar la tarjeta de E/S, los tres ledes, rojo, verde y azul, se conectarán entre los pines 8, 7 y 6, respectivamente, y 3,3 V —véase la Figura 9.8—. Cada led estará conectado a través de su correspondiente resistencia, para limitar la corriente cuando esté encendido. Puesto que los tres ledes están conectados a la señal de 3,3 V, se dice que están configurados en ánodo común. Esta disposición implica que para encender un led será necesario poner un `0` lógico en el pin al que está conectado, mientras que para apagarlo, habrá que poner un `1` lógico. Como estos pines están asociados a los bits 22, 23 y 24, respectivamente, del puerto PIOC del ATSAM3X8E —véase el Cuadro A.1—, para poder encender o apagar alguno de los ledes, será necesario configurar el bit correspondiente del puerto PIOC como salida. Una vez configurado uno de estos bits, se podrá encender el led correspondiente escribiendo un 0 en ese bit del registro de datos del PIOC y apagarlo escribiendo un 1.

Cuadro A.1: Dispositivos de la tarjeta de E/S, pines a los que están conectados en la tarjeta Arduino y controladores y bits del microcontrolador ATSAM3X8E a los que están asociados

Dispositivo	Pin	Controlador	Bit
Led rojo	8	PIOC	22
Led verde	7	PIOC	23
Led azul	6	PIOC	24
Pulsador	13	PIOB	27

A.2. Controladores de entrada/salida de propósito general en el ATSAM3X8E

El microcontrolador ATSAM3X8E dispone de cuatro controladores de entrada/salida de propósito general muy versátiles y potentes. Estos controladores, llamados PIO —abreviatura del inglés *Parallel Input/Output Controller*—, se describen con detalle en la hoja de características de la familia de microcontroladores SAM3X-SAM3A de Atmel [4] —páginas 631 a 675—. En este apartado se resumen sus aspectos más importantes, centrándose en las especificaciones correspondientes al microcontrolador ATSAM3X8E, el utilizado en la tarjeta Arduino Due.

El microcontrolador ATSAM3X8E dispone de cuatro PIO, identificados como PIOA, PIOB, PIOC y PIOD, que gestionan hasta un máximo de 32 pines de E/S cada uno —véase el Cuadro A.2—. De hecho, y para recalcar de nuevo la importancia de la entrada/salida sobre todo en sistemas empotrados, 103 pines del microcontrolador —de sus 144, en el caso del encapsulado LQFP—, pueden utilizarse como pines de entrada/salida.

Cada uno de los pines de E/S, cuya estructura interna se muestra en la Figura A.1, puede configurarse bien como asociado a uno de los dos dispositivos internos[1] conectados a ese pin, bien como entrada/salida de propósito general. En el primer caso, será posible seleccionar a cuál de los dos dispositivos integrados, etiquetados como A y B, se podrá acceder a través del pin —parte izquierda superior y centro derecha de la Figura A.1—. En el segundo caso, el pin puede configurarse como entrada, parte inferior de la Figura A.1, o salida, parte superior de la Figura A.1. Cuando se configura como entrada, puede configurarse además para filtrar espúreos y rebotes o para no hacerlo; y para generar o no peticiones de interrupción cuando: i) la señal cambie de un nivel a otro,

[1] La lista de dispositivos integrados en el ATSAM3X8E, así como a qué pines están conectados, se puede consultar en el Apartado «9. Peripherals» de [4]. La lista de dispositivos también se muestra en el Cuadro A.11, aunque desde el punto de vista de la gestión de interrupciones.

II) mientras esté en un determinado nivel —alto o bajo—, o III) transite entre el nivel considerado inactivo y el activo —flanco de subida o de bajada—. También es posible configurar el comportamiento eléctrico del pin: si se utiliza o no una resistencia de *pull-up* y si se configura o no en colector abierto. En cuanto a su comportamiento como salida, el controlador dispone de un registro para poner a 0 los pines y otro distinto para ponerlos a 1. Un tercer registro, de uso más restringido, permite modificar simultáneamente varios pines, sean 0 o 1 los valores a escribir.

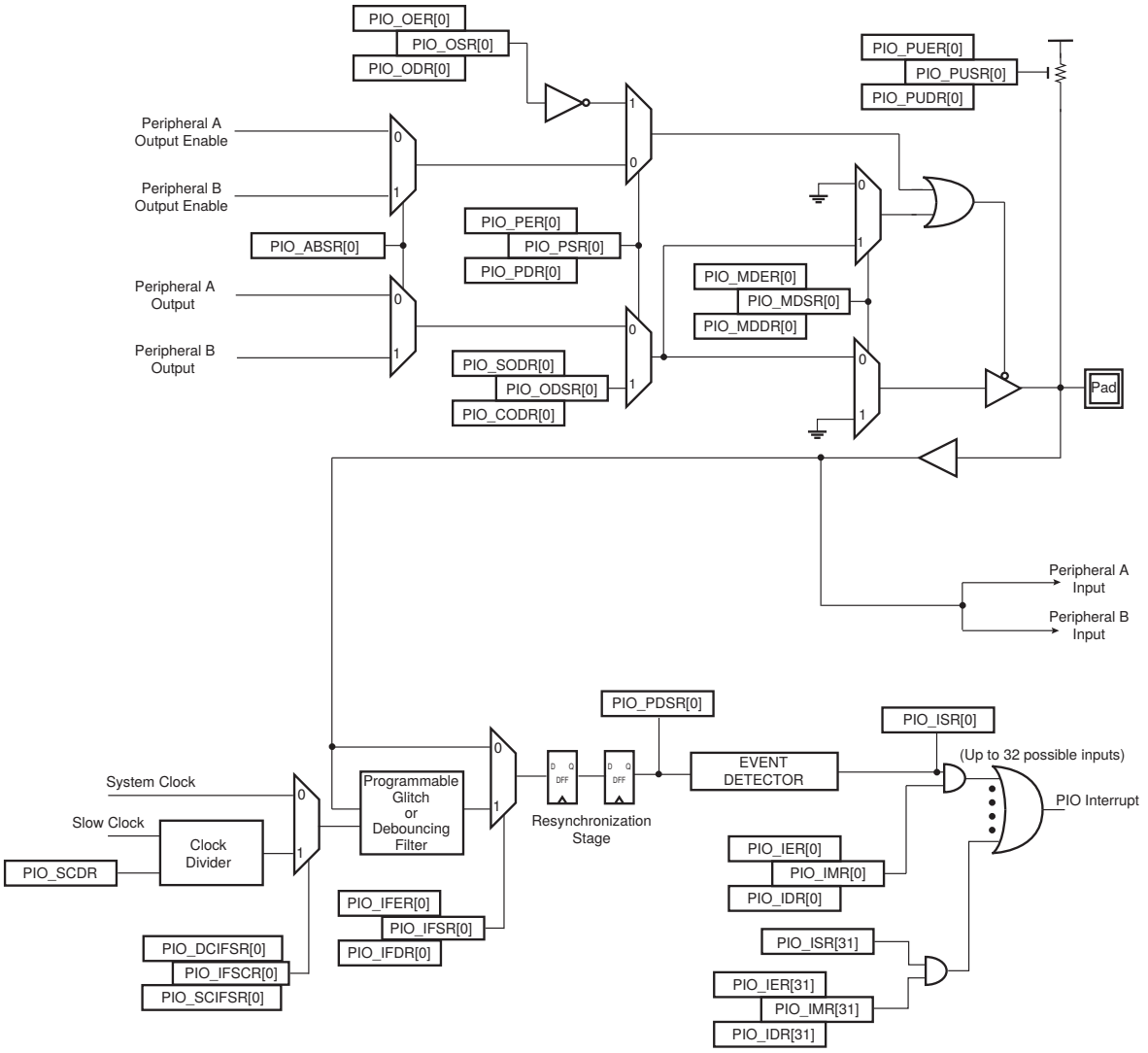


Figura A.1: Estructura interna de un pin de entrada/salida del microcontrolador ATSAM3X8E (fuente: [4])

Cuadro A.2: Número de pines y dirección base de los controladores PIO del ATSAM3X8E

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

Debido a la gran versatilidad y potencia de estos controladores, que puede intuirse a partir del anterior resumen, cada PIO proporciona 43 registros de 32 bits. Su espacio de direcciones, formado por estos registros y por varios conjuntos de direcciones reservadas, abarca un total de 324 direcciones —172 de las cuales corresponden a los 43 registros—. Por tanto, y puesto que en ARM la entrada/salida está mapeada en memoria, cada PIO ocupará 324 direcciones del mapa de memoria.

El Cuadro A.2 muestra la dirección base asignada a cada PIO y el Cuadro A.3, el nombre y alias de los registros proporcionados por cada PIO, así como su posición relativa con respecto a la dirección base. Con esta información es posible obtener la dirección de memoria en la que está mapeado cada uno de los registros de cada uno de los PIO. Por ejemplo, el registro `PIO_PSR` del PIOC está mapeado en la dirección de memoria `0x400E 1208`. Aunque sería posible utilizar esta dirección para acceder al registro, lo habitual, puesto que las instrucciones de carga y almacenamiento soportan el modo de direccionamiento indirecto con desplazamiento, es cargar en un registro la dirección base del PIO y especificar la dirección relativa del registro con respecto a dicha dirección como desplazamiento. Así, y suponiendo que las constantes «`PIOC`» y «`PIO_PSR`» se han definido previamente como `0x400E 1200` y `0x0008`, respectivamente, la forma habitual de cargar el contenido del registro `PIO_PSR` del controlador PIOC en el registro `r1`, será:

```
1  ldr r0, =PIOC           @ r0 <- Dirección base del PIOC
2  ldr r1, [r0, #PIO_PSR]  @ r1 <- Registro PIO_PSR del PIOC
```

Cuadro A.3: Registros de los controladores PIO y su dirección relativa

Registro del PIO	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`

Continúa en la siguiente página...

Registros de los controladores PIO y su dirección relativa (continuación)

Registro del PIO	Alias	Desplazamiento
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_IFSR	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
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
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

Continúa en la siguiente página...

Registros de los controladores PIO y su dirección relativa (continuación)

Registro del PIO	Alias	Desplazamiento
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

A.2.1. Configuración de un pin como GPIO o como E/S de un dispositivo integrado

Como se ha comentado, cada uno de los pines de E/S del ATSAM3X8E puede configurarse en cualquier momento bien como entrada/salida de propósito general o bien como entrada/salida de uno de los dos dispositivos internos conectados a ese pin. Para hacerlo, se deberá escribir una palabra en la que los bits asociados a los pines que se quieren configurar estarán a 1, en uno de los siguientes registros del controlador PIO correspondiente:

PIO Enable Register **(PIO_PER)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines de entrada/salida de propósito general.

PIO Disable Register **(PIO_PDR)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines asociados a uno de los dos periféricos conectados a estos.

Así por ejemplo, para configurar los 3 pines asociados a los bits 22, 23 y 24 del PIOC como entrada/salida de propósito general, se deberá escribir la palabra 0x01C00000 —en la que los bits 22, 23 y 24 están a 1— en el registro PIO_PER del PIOC. Por tanto, cuando el procesador ejecute el siguiente fragmento de código, en el que la constante «PIOC» es la dirección de inicio del mapa de direcciones del controlador PIOC y la constante «PIO_PER», el desplazamiento del registro PIO_PER con respecto a la anterior dirección, el controlador del PIOC configurará los pines asociados a los bits 22, 23 y 24 como pines de entrada/salida de propósito general:

```
ldr r0, =PIOC
ldr r1, =0x01C00000
str r1, [r0, #PIO_PER]
```

Por otro lado, para saber qué pines de entrada/salida están configurados como entrada/salida de propósito general o como entrada/salida de uno de los dos dispositivos asignados a estos, se deberá leer el siguiente registro:

***PIO Status Register* (`PIO_PSR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados como entrada/salida de propósito general estarán a 1 y los asociados a uno de los dispositivos conectados a estos estarán a 0.

Por último, para consultar o seleccionar cuál de los dos dispositivos, A o B, conectados a un determinado pin de entrada/salida, hará uso de él cuando este sea configurado como asociado a un dispositivo interno, se deberá consultar o actualizar el siguiente registro:

***PIO Peripheral AB Select Register* (`PIO_ABSR`)** Un bit a 0 en este registro indica que el dispositivo A utilizará el pin correspondiente, mientras que un bit a 1 indica que será el dispositivo B el que lo utilice.

Así por ejemplo, y sabiendo que los pines asociados a los bits 22, 23 y 24 del PIOC están conectados a las salidas 4, 5 y 6, respectivamente, del generador de PWM, como dispositivo B, el procesador debería ejecutar el siguiente código para que los pines de entrada/salida correspondientes se utilicen como salidas del generador de PWM:

```
ldr r0, =PIOC
ldr r1, =0x01C00000
ldr r2, [r0, #PIO_ABSR]  @ r2 <- disp. seleccionados (A o B)
orr r2, r1               @ Pone a 1 los bits 22, 23 y 24 de r2
str r2, [r0, #PIO_ABSR]  @ Actualiza el registro PIO_ABSR con r2
str r1, [r0, #PIO_PDR]   @ Configura pines como E/S de un disp.
```

A.2.2. Configuración de pines de E/S de propósito general

Cuando se quiere utilizar un pin como entrada/salida de propósito general, la propia especificación de su función en el sistema indica el primer parámetro a configurar: su comportamiento, es decir, si se utilizará como entrada o como salida. Para ello se deberá escribir una palabra con

el bit asociado a ese pin a 1 en uno de dos los siguientes registros —dependiendo de en cuál de los dos se escriba, se configurará como entrada o como salida—:

***Output Enable Register* (`PIO_OER`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines de salida.

***Output Disable Register* (`PIO_ODR`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines de entrada.

Así por ejemplo, para configurar los 3 pines asociados a los bits 22, 23 y 24 del PIOC como salidas, se deberá escribir la palabra `0x01C00000` —en la que los bits 22, 23 y 24 están a 1— en el registro `PIO_OER` del PIOC. Por tanto, cuando el procesador ejecute el siguiente fragmento de código, en el que la constante «`PIOC`» es la dirección de inicio del mapa de direcciones del controlador PIOC y la constante «`PIO_OER`», el desplazamiento del registro `PIO_OER` con respecto a la anterior dirección, el controlador del PIOC configurará los pines asociados a los bits 22, 23 y 24 como pines de salida —siempre y cuando se configuren como pines de entrada/salida de propósito general—:

```
1  ldr r0, =PIOC
2  ldr r1, =0x01C00000
3  str r1, [r0, #PIO_OER]
```

Por otro lado, para consultar qué bits están configurados como entrada y qué bits como salida, se deberá leer el siguiente registro:

***Output Status Register* (`PIO_OSR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados como salida estarán a 1 y los configurados como entrada, a 0.

Si el pin se va a utilizar como salida, normalmente no será necesario configurar ningún otro registro. Solo en el caso —que posiblemente quede fuera del ámbito de este curso— de que se quiera usar un nivel lógico de salida distinto al normal del microcontrolador, o que se quiera conectar con otras salidas para formar una función AND cableada, será necesario configurar su comportamiento eléctrico en colector abierto. Para ello intervienen los siguientes registros:

***Multi-driver Enable Register* (`PIO_MDER`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 en el modo de colector abierto.

***Multi-driver Disable Register* (PIO_MDDR)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que no utilicen el modo de colector abierto.

***Multi-driver Status Register* (PIO_MDSR)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados en modo de colector abierto estarán a 1 y los que no, a 0.

Por otra parte, cuando el pin va a utilizarse como entrada, hay que considerar más factores, lo que implica que hay muchas más opciones de configuración. En primer lugar, hay que tener en cuenta que los circuitos de lectura del pin —véase de nuevo la Figura A.1— requieren de una señal de reloj para actualizar el valor almacenado de la señal de entrada, que es el que podrá leerse en el registro PIO_PDSR. Para activar la señal de reloj,[2] se deberá escribir una palabra con el bit 11, 12, 13 o 14 a 1, dependiendo de si se quiere activar la señal de reloj del PIOA, PIOB, PIOC o PIOD, respectivamente, en el registro *Peripheral Clock Enable 0* (PMC_PCER0) del controlador de gestión de energía —*Power Management Controller* (PMC)—. Así pues, para activar el reloj del PIOB, como paso previo a leer sus pines, debería ejecutarse el siguiente código:

```
1  ldr r0, =PMC
2  ldr r1, =0x00001000      @ r1 <- bit 12 a 1, el resto a 0
3  str r2, [r0, #PMC_PCER0] @ Activa el reloj del PIOB (id: 12)
```

En cuanto a sus características eléctricas como entrada, se puede querer activar una resistencia de *pull-up*, como en el caso ya comentado de utilizar el pin para leer un pulsador. Para esto intervienen los siguientes registros:

pull-up *Enable Register* (PIO_PUER) Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que utilicen la resistencia de *pull-up* integrada en el pin.

pull-up *Disable Register* (PIO_PUDR) Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que no utilicen la resistencia de *pull-up* integrada en el pin.

pull-up *Status Register* (PIO_PUSR) Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados para utilizar la resistencia de *pull-up*

[2] Las señales de reloj de los periféricos integrados en el ATSAM3X8E están desactivadas por defecto. Se hace así para ahorrar energía.

estarán a 0 y los que no, a 1. *(Es importante destacar que el significado de la salida de este registro es el contrario al adoptado en los restantes registros de estado.)*

La siguiente característica que puede configurarse es si se quiere filtrar la señal de entrada o no. Dado que las entradas se muestrean en ciertos intervalos, como se ha comentado antes, la lógica de cada pin incorpora una etapa de filtrado que puede utilizarse para la eliminación de espúreos —*glitch input filter*— o para la eliminación de rebotes —*debouncing input filter*—. El filtro de espúreos elimina los cambios de muy corta duración que puedan producirse en la señal de entrada. Se utiliza para evitar errores provocados por el ruido electromagnético. Por otra parte, el filtro de rebotes elimina los pulsos cuya duración sea inferior a un tiempo prefijado. Es especialmente útil cuando la señal de entrada procede de un pulsador, ya que los contactos mecánicos del pulsador podrían provocar pulsos no deseados —rebotes— al pulsarlo o al soltarlo. Para activar, desactivar o consultar la etapa de filtrado para cada pin, se pueden utilizar los registros `PIO_IFER`, `PIO_IFDR` y `PIO_IFSR`, respectivamente. Para seleccionar cuál de los dos filtros se quiere aplicar en cada pin, se puede escribir en el registro `PIO_SCIFSR`, para seleccionar el filtrado de espúreos, o en el `PIO_DIFSR`, para seleccionar el de rebotes. Si se quiere consultar cuál de los dos filtros está seleccionado, se puede leer el registro `PIO_IFDGSR`. Por último, en el caso del filtrado de rebotes, es posible multiplicar el periodo del reloj utilizado para detectar si un pulso dura lo suficiente como para no considerarlo un rebote, escribiendo en el registro `PIO_SCDR`.[3]

Por último, cuando el pin se utiliza como entrada puede ser adecuado que sea capaz de generar interrupciones. El Apartado A.2.4 trata en detalle cómo configurar el controlador en este caso.

A.2.3. Lectura y escritura de pines de E/S de propósito general

Para obtener el valor lógico de la señal presente en un pin que se haya configurado como entrada, se deberá leer el registro:

***Pin Data Status Register* (`PIO_PDSR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados como entrada estarán a 0 o a 1 en función del valor de la señal presente en ellos, si no se ha aplicado ningún filtro, o de la misma señal una vez filtrada, en el caso de haber aplicado el filtro de espúreos o el de rebotes.

[3] El número que se escriba en `PIO_SCDR`, DIV, deberá estar entre 0 y 16 383. El nuevo periodo se calculará como $2(DIV + 1)T_{sc}$, donde T_{sc} es el periodo del reloj utilizado por el filtro de rebotes.

Así por ejemplo, si se quiere leer el valor presente en el pin asociado al bit 27 del PIOB, el procesador debería ejecutar el siguiente código, donde la constante «`PIOB`» es la dirección de inicio de los registros del PIOB y la constate «`PIO_PDSR`» es al dirección relativa del registro `PIO_PDSR`:

```
1  ldr r0, =PIOB
2  ldr r1, =0x08000000      @ r1 <- bit 27 a 1, el resto a 0
3  ldr r2, [r0, #PIO_PDSR]  @ r2 <- PIO_PDSR del PIOB
4  and r2, r1               @ Pone a 0 todos los bits salvo el 27
```

En el código anterior, cuando el procesador ejecute la instrucción «`ldr r2, [r0, #PIO_PDSR]`», el registro `r2` tendrá el valor presente en la entrada de todos los pines del PIOB. En el caso de solo querer el de algunos de sus pines, será necesario aplicar una máscara que preserve el valor de los bits asociados a esos pines y ponga a 0 el resto. Puesto que en el ejemplo anterior solo se quería el valor presente en la entrada del pin asociado al bit 27, se ha cargado una máscara con el bit 27 a 1 en el registro `r1`, «`ldr r1, =0x08000000`», y se ha aplicado esta máscara sobre el valor cargado en `r2` de `PIO_PDSR`, «`and r2, r1`». De esta forma se ha preservado el valor presente en el pin asociado al bit 27 y se ha puesto a 0 el resto.

En cuanto a la escritura en pines de salida, para modificar el nivel alto o bajo de un pin de salida, se deberá escribir una palabra con el bit asociado a ese pin a 1 en uno de dos de los siguientes registros —dependiendo de en cuál de los dos se escriba, el pin se pondrá a nivel alto o a nivel bajo—:

Set Output Data Register **(`PIO_SODR`)** Cuando se escriba una palabra en este registro, el controlador pondrá a nivel alto los pines asociados a aquellos bits que estén a 1.

Clear Output Data Register **(`PIO_CODR`)** Cuando se escriba una palabra en este registro, el controlador pondrá a nivel bajo los pines asociados a aquellos bits que estén a 1.

Escribir en estos registros es una forma conveniente de poner simultáneamente varios pines de salida bien a nivel alto, o bien a nivel bajo. Sin embargo, si fuera necesario poner simultáneamente varios pines a distintos niveles, no sería posible hacerlo —primero se tendrían que poner unos pines a nivel alto y después los restantes a nivel bajo, o viceversa—. Esta limitación es importante, pues en muchas ocasiones la salida consiste justamente en el valor conjunto de varios pines. Por ejemplo, si la información que se quiere transmitir es una secuencia de bytes, una forma de hacerlo es transmitir esta secuencia byte a byte, utilizando 8 pines de salida para codificar cada byte. En este caso, los 8

pines se deberían actualizar de forma síncrona. Si no se hiciera así, cada vez que se fuera a transmitir un nuevo byte, el valor presente en los pines sería erróneo mientras no se terminaran de actualizar todos los pines, y el receptor se vería obligado a esperar durante un tiempo prudencial a que el valor en sus entradas se estabilizara. El resultado sería un sistema de transmisión más propenso a errores y más lento que en el caso de que todos los pines se hubieran actualizado de forma síncrona. Puesto que esta situación, la de tener que actualizar varias salidas de forma síncrona, es bastante habitual, los controladores PIO del ATSAM3X8E permiten un segundo modo de actualizar sus pines de salida, que recibe el nombre de *escritura directa*. Para poder utilizar el modo de escritura directa se deberá configurar en primer lugar qué pines de salida van a participar de dicho modo. Esto se consigue por medio de los siguientes registros:

***Output Write Enable Register* (`PIO_OWER`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines de escritura directa.

***Output Write Disable Register* (`PIO_OWDR`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines a los que no afectará la escritura directa.

Se puede saber qué pines están configurados como escritura directa en un momento dado leyendo el siguiente registro:

***Output Write Status Register* (`PIO_OWSR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados para escritura directa estarán a 1 y los que no, a 0.

Una vez se ha configurado qué pines participan en el modo de escritura directa, para actualizar sus niveles se deberá escribir en el siguiente registro:

Output Data Status Register (PIO_ODSR) Cuando se escriba[4] en este registro, el controlador actualizará el nivel de los pines habilitados para escritura directa de acuerdo a los valores de los bits asociados a estos.

A.2.4. Uso de los PIO para generar peticiones de interrupción

Los controladores PIO pueden generar peticiones de interrupción en función de cómo sea la señal de entrada presente en alguno, o en varios,

[4] El registro `PIO_ODSR` es en realidad un registro de lectura/escritura. Cuando se lea, se obtendrá el valor actual de las salidas.

de sus pines de entrada/salida. Para que esta característica sea útil, será necesario satisfacer previamente los dos siguientes requisitos. En primer lugar, será necesario haber activado el reloj del PIO, en la forma en la que se ha comentado en el Apartado A.2.2. Esto es debido a que la generación de peticiones de interrupción depende de la lógica de entrada del pin, que como ya se ha visto, requiere que el reloj del PIO esté activado. En segundo lugar, para que la petición de interrupción se propague, será necesario programar también el controlador de interrupciones del sistema *(NVIC)*, tal y como se describe más adelante en el Apartado A.6.

Satisfechos los requisitos anteriores, el PIO permite configurar: I) qué pines de entrada tiene que monitorizar para generar una petición de interrupción cuando sea el caso, y II) qué debe ocurrir en la señal de entrada de los pines monitorizados para que el PIO genere una petición de interrupción.

Para poder configurar o consultar qué pines de entrada se van a monitorizar para generar una petición de interrupción, el PIO proporciona los siguientes registros:

***Interrupt Enable Register* (`PIO_IER`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines que deberán monitorizarse para generar una petición de interrupción.

***Interrupt Disable Register* (`PIO_IDR`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 como pines que no se tendrán en cuenta para generar peticiones de interrupción.

***Interrupt Mask Register* (`PIO_IMR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados para ser monitorizados estarán a 1 y los que no, a 0.

En cuanto a qué es lo que debe ocurrir en la señal de entrada de uno de los pines monitorizados para que el PIO genere una petición de interrupción, se contemplan las siguientes opciones: I) que la señal cambie de un nivel a otro —interrupción generada por cambio—, II) que esté en un determinado nivel —interrupción generada por nivel alto o bajo—, o III) que transite entre el nivel considerado inactivo y el activo —interrupción generada por flanco de subida o de bajada—.

La forma de configurar cuál de las cinco situaciones descritas anteriormente es la que va a provocar que se genere una petición de interrupción es la siguiente: en primer lugar, se distinguirá si se va a seleccionar una de las causas de interrupción que el microcontrolador considera adicionales —por nivel o por flanco—, o no —causa de interrupción por

cambio—. Si se selecciona la causa de interrupción por cambio, no hay que hacer nada más. En caso contrario, habrá que seleccionar si se utiliza la causa de interrupción por nivel o la de por flanco y si se genera la petición de interrupción cuando la señal de entrada esté en el nivel alto —describa un flanco de subida— o esté en el nivel bajo —describa un flanco de bajada—. Para configurar y consultar todas estas opciones, el controlador proporciona los siguientes registros:

Additional Interrupt Modes Enable Register (**`PIO_AIMER`**) Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que se genere una petición de interrupción cuando ocurra una de las causas adicionales de interrupción: por nivel o por flanco.

Additional Interrupt Modes Disable Register (**`PIO_AIMDR`**) Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que se genere una petición de interrupción cuando ocurra la causa básica de interrupción: un cambio en la señal de entrada de alguno de estos pines.

Additional Interrupt Modes Mask Register (**`PIO_AIMMR`**) Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados para que se genere una petición de interrupción por una de las causas adicionales estarán a 1, y los asociados a los que estén configurados para generarla cuando se produzca un cambio en la señal de entrada estarán a 0.

Edge Select Register (**`PIO_ESR`**) Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que en el caso de haber seleccionado como causa de interrupción una de las adicionales, se genere una petición de interrupción en cuanto se produzca un flanco en la señal de entrada.

Level Select Register (**`PIO_LSR`**) Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que en el caso de haber seleccionado como causa de interrupción una de las adicionales, se generen peticiones de interrupción mientras la señal de entrada se mantenga en un determinado nivel.

Edge/Level Status Register (**`PIO_ELSR`**) Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados para utilizar el flanco como causa de generación de peticiones de interrupción estarán a 1, y los asociados a los configurados para utilizar el nivel como causa, a 0.

Falling Edge/Low Level Select Register **(`PIO_FELLSR`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que en el caso de haber seleccionado como causa de interrupción una de las adicionales, se generen peticiones de interrupciones cuando haya un flanco de bajada o se esté en el nivel bajo.

Rising Edge/High Level Select Register **(`PIO_REHLSR`)** Cuando se escriba en este registro, el controlador configurará los pines asociados a los bits que estén a 1 para que en el caso de haber seleccionado como causa de interrupción una de las adicionales, se generen peticiones de interrupciones cuando haya un flanco de subida o se esté en el nivel alto.

Fall/Rise Low/High Status Register **(`PIO_FRLHSR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida configurados para utilizar el flanco de bajada o el nivel bajo estarán a 0 y los configurados para utilizar el flanco de subida o el nivel alto estarán a 1.

Si algunos de los pines del PIO se han configurado para que sus señales de entrada puedan causar una petición de interrupción, en cuanto se den las condiciones especificadas, el controlador del PIO generará una petición de interrupción. Cuando esta petición de interrupción vaya a ser tratada, será necesario saber cuál de los pines la han provocado. También es posible que la señal de entrada de varios pines hayan provocado la petición de interrupción de forma simultánea o, con mayor probabilidad, que entre que se realizó la petición de interrupción y esta es tratada, la entrada de algún otro pin haya causado otra petición de interrupción. Así pues, para poder tratar la petición de interrupción de forma adecuada, el controlador del PIO debe proporcionar un registro que indique qué pines han generado una petición de interrupción que esté a la espera de ser atendida. Este registro es:

Interrupt Status Register **(`PIO_ISR`)** Cuando se lea este registro, se obtendrá una palabra en la que los bits asociados a los pines de entrada/salida que han generado una petición de interrupción desde la última vez que fue leído y que, por tanto, están pendientes de ser atendidos, estarán a 1. Para facilitar el tratamiento de las interrupciones, este registro se pone a 0 automáticamente tras ser leído.

Cuadro A.4: Registros del temporizador *SysTick* y sus direcciones

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

A.3. El temporizador del sistema del ATSAM3X8E

La arquitectura ARM, además de especificar cómo debe ser un procesador basado en dicha arquitectura, también describe un conjunto de periféricos que, junto con el procesador, conformarán el núcleo de un sistema ARM. Entre los periféricos que forman parte de un sistema ARM Cortex-M3, se encuentra el temporizador del sistema, que recibe el nombre de *SysTick* [2]. Esta arquitectura especifica que el temporizador del sistema debe ser un contador descendente de 24 bits, que decremente su valor en 1 con cada pulso de la señal de reloj —que deberá poder seleccionarse entre la del procesador y una externa— y que cuando llegue a 0, se recargue automáticamente a un valor que deberá poder configurarse. Esta arquitectura también especifica que el temporizador deberá ser capaz, cuando la cuenta llegue a 0, de generar una petición de interrupción, que recibe el mismo nombre que el temporizador, *SysTick*, y se ha asignado al número de excepción 15. Por último, también forma parte de su especificación qué registros debe proporcionar su controlador y las direcciones de memoria en la que estos deben mapearse —véase el Cuadro A.4—.

El microcontrolador ATSAM3X8E implementa el temporizador *SysTick* siguiendo las especificaciones de la arquitectura ARM Cortex-M3. Los registros del controlador del *SysTick* del ATSAM3X8E tienen la siguiente funcionalidad:

***Control and Status Register* (`CTRL`)** De los 32 bits de este registro, solo 4 tienen una función definida: 3 son de control y 1 de estado. Los bits de control son los siguientes:

- El bit 2 —`CLKSOURCE`—, que sirve para indicar que se debe utilizar el reloj del procesador, si se pone a 1, o un reloj externo, si se pone a 0. El microcontrolador ATSAM3X8E conecta a la entrada del reloj externo del *SysTick* un reloj con una frecuencia de un octavo de la del procesador.

- El bit 1 —`TICKINT`—, que sirve para habilitar la generación de una petición de interrupción cada vez que su cuenta llegue a 0, si se pone a 1, o para deshabilitarla, si se pone a 0.
- El bit 0 —`ENABLE`— sirve para habilitar el temporizador, si se pone a 1, o para deshabilitarlo, si se pone a 0.

Por otra parte, el bit de estado es el bit 16 —`COUNTFLAG`—, que indica si el contador ha llegado alguna vez a 0 desde la última vez que se leyó este registro.

Reload Value Register **(`LOAD`)** Se utiliza para indicar, en sus 24 bits de menor peso, el valor al que deberá volver a iniciarse el contador después de llegar a 0. Cambiando el valor de este registro es posible variar el tiempo que le llevará al temporizador llegar a cero y con ello, el tiempo entre interrupciones del temporizador, en el caso de que estén habilitadas.

Current Value Register **(`VAL`)** Contiene el valor actual del contador descendente.

Calibration Value Register **(`CALIB`)** Sirve para indicar varios atributos relacionados con la calibración de la frecuencia de actualización.

Cuando el microcontrolador ATSAM3X8E se utilice en una tarjeta Arduino, el entorno de desarrollo de Arduino incorporará a los proyectos Arduino, de forma transparente para el programador, un conjunto de rutinas que configurarán y utilizarán el temporizador *SysTick*. Así, el entorno añadirá una rutina que, en el caso de la tarjeta Arduino Due, indicará al temporizador que: I) utilice el reloj del procesador, de 84 Mhz; II) fije el valor de recarga a 83 999; y III) genere una petición de interrupción cada vez que su cuenta llegue a 0. De esta forma, el temporizador cambiará de valor aproximadamente cada 12 ns y generará una petición de interrupción cada milisegundo. El entorno Arduino también incorporará una rutina de tratamiento de la interrupción *SysTick*, que cada vez que sea llamada, simplemente incrementará la variable global «`_dwTickCount`». Esta variable será utilizada por varias funciones del entorno Arduino, entre ellas: «`delay()`», que pausa el programa el número de milisegundos que se le haya indicado; y «`millis()`», que devuelve el número de milisegundos desde que el programa de la tarjeta comenzó a ejecutarse. Otras funciones del entorno Arduino que requieren mayor precisión, p.e. «`delayMicroseconds()`» y «`micros()`», utilizan directamente el valor del registro *Current Value Register* del temporizador.

Es interesante observar que todas las funciones comentadas en el párrafo anterior dependen de la existencia de un elemento hardware, un

temporizador, para llevar a cabo su labor. También es interesante ver que algunas funciones, p.e., «`delay()`» y «`millis()`», aún necesitando de la existencia de un temporizador hardware, se han desarrollado de forma que no son dependientes del hardware concreto que se haya usado. Gracias a esto, estas funciones podrán utilizarse tal cual en otros modelos de tarjetas Arduino, en los que tan solo habría que implementar una forma de actualizar la variable global «`_dwTickCount`» en función de su hardware.

A.4. El reloj en tiempo real del ATSAM3X8E

Algunos microcontroladores, como ocurre en el caso del microcontrolador ATSAM3X8E, integran un reloj en tiempo real (RTC) que les permite mantener actualizada la información de fecha y hora. Sin embargo, al contrario de lo que ocurre con los módulos de RTC independientes, es habitual que los RTC integrados no dispongan de alimentación propia, por lo que cuando se desconecte la alimentación del sistema, no podrán seguir actualizando la información de fecha y hora y perderán la información que tuvieran almacenada.

El diagrama de bloques del RTC integrado en el microcontrolador ATSAM3X8E se muestra en la Figura A.2. Como se puede apreciar, dispone de dos contadores, `Date` y `Time`, que mantienen la información de fecha y hora, respectivamente. El procedimiento utilizado para incrementar estos contadores es el siguiente. En primer lugar, el RTC recibe en su entrada de reloj la señal SLCK —*Slow Clock*—, cuya frecuencia es de 32 768 Hz. Esta señal pasa a continuación por un divisor de 32 768 que genera una señal de 1 Hz —un ciclo por segundo— y que se utilizará para incrementar el contador de horas, minutos y segundos —`Time`—. Por último, el contador `Time` generará una señal de reloj de un ciclo por día, que se utilizará para incrementar el contador de fecha —`Date`—.

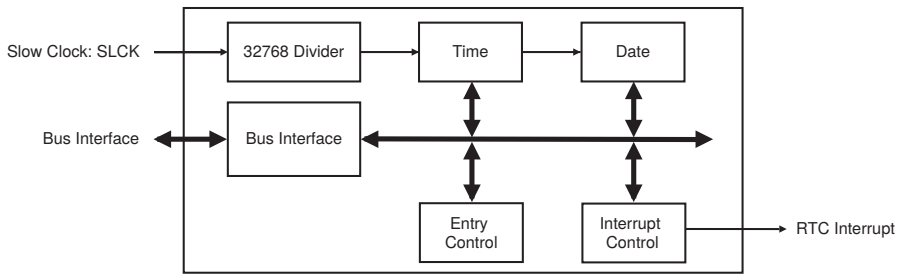


Figura A.2: Diagrama de bloques del RTC del microcontrolador ATSAM3X8E (fuente: [4])

Cuadro A.5: Dirección base del controlador del RTC del ATSAM3X8E

Alias	Dirección base
RTC	`0x400E1A60`

Por otro lado, el RTC dispone de un módulo de control y uno de gestión de interrupciones. El módulo de control se encarga de gestionar las funciones de consulta, modificación y configuración, que permite que se puedan leer y modificar la fecha y hora actuales y configurar una alarma y eventos periódicos de fecha y hora. Por su parte, el módulo de gestión de interrupciones se ocupa de generar peticiones de interrupción ante determinadas circunstancias, por ejemplo, cuando salte una alarma que haya sido previamente programada. El acceso a estos módulos y a los registros `Date` y `Time` se realiza a través de la conexión del RTC al bus interno del ATSAM3X8E, como se puede ver en la Figura A.2.

Como es habitual, el procesador se comunicará con el RTC escribiendo y leyendo sus registros, que están mapeados en memoria a partir de la dirección `0x400E1A60` —véase el Cuadro A.5—. Por su parte, en el Cuadro A.6 se muestran los nombres y alias de estos registros, así como su posición relativa con respecto a la anterior dirección. A partir de la información proporcionada en este cuadro es posible obtener la dirección de memoria en la que está mapeado cada uno de los registros del RTC. Por ejemplo, el registro `RTC_TIMR` está mapeado en la dirección de memoria `0x400E1A68`. Aunque sería posible utilizar esta dirección para acceder al registro, lo habitual, como ya se ha comentado previamente y puesto que las instrucciones de carga y almacenamiento soportan el modo de direccionamiento indirecto con desplazamiento, es cargar en un registro la dirección base del RTC y especificar la dirección relativa del registro con respecto a dicha dirección como desplazamiento. Así, y suponiendo que las constantes «`RTC`» y «`RTC_TIMR`» se han definido previamente como `0x400E1A60` y `0x08`, respectivamente, la forma habitual de cargar el contenido del registro `RTC_TIMR` del controlador del RTC en el registro `r1`, será:

```
1  ldr r0, =RTC            @ r0 <- Dirección base del RTC
2  ldr r1, [r0, #RTC_TIMR] @ r1 <- Registro RTC_TIMR del PIOC
```

Cuadro A.6: Registros del controlador del RTC y su dirección relativa

Registro del RTC	Alias	Desplazamiento
Control Register	`RTC_CR`	`0x00`
Mode Register	`RTC_MR`	`0x04`

Continúa en la siguiente página...

Sistema de representación BCD

Al contrario de lo que ocurre en la representación en binario de un número decimal, en la que un número se convierte del sistema de numeración decimal al binario, el sistema de representación BCD —*Binary Coded Decimal*—, codifica por separado cada dígito decimal en binario, obteniendo al final una secuencia de bloques de 4 bits, donde cada bloque corresponde a un dígito del número decimal original. Así, por ejemplo, el número decimal 1249 se codificaría en BCD como $\underbrace{0001}_{1}\ \underbrace{0010}_{2}\ \underbrace{0100}_{4}\ \underbrace{1001}_{9}$.

Registros del controlador del RTC y su dirección relativa (continuación)

Registro del RTC	Alias	Desplazamiento
Time Register	`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`
Write Protect Mode Register	`RTC_WPMR`	`0xE4`

En los siguientes apartados se explica lo siguiente: cómo se codifican la fecha y la hora en los registros `RTC_CALR` y `RTC_TIMR`, respectivamente; cómo se deben leer y actualizar la fecha y la hora del RTC; cómo se pueden programar la alarma y los eventos periódicos; cómo configurar y gestionar las interrupciones del RTC; y, por último, como proteger ciertos registros contra escrituras accidentales.

A.4.1. Codificación de la fecha

El reloj en tiempo real almacena la fecha en el registro *RTC Calendar Register* (`RTC_CALR`), que sigue el formato mostrado en la Figura A.3. Los campos[5] de este registro se describen a continuación.

[5] Como todos los campos del registro `RTC_CALR` se codifican en BCD, conviene consultar al menos el breve resumen proporcionado sobre este sistema de representación.

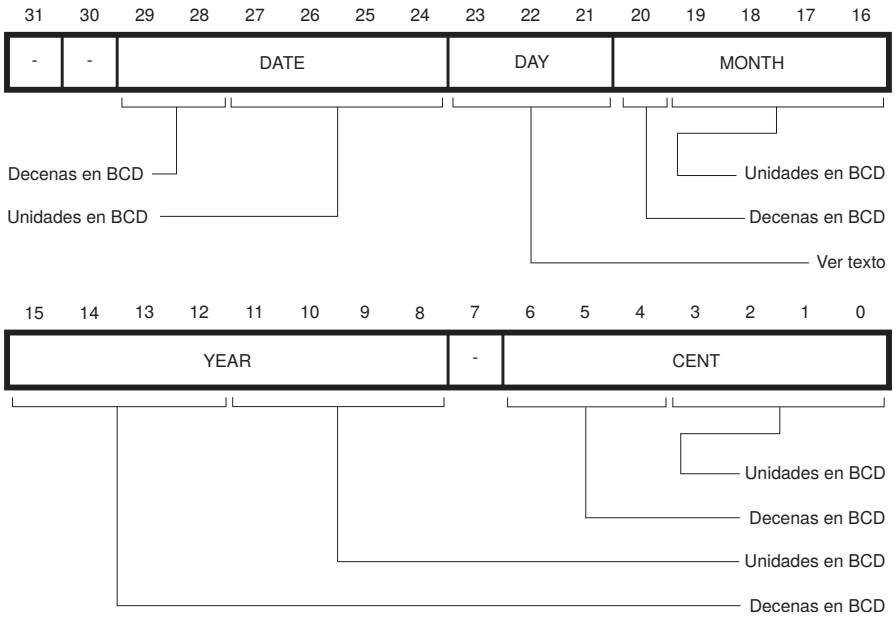


Figura A.3: Formato del registro RTC Calendar Register

CENT Codifica el siglo en BCD con 7 bits. Solo puede tomar los valores 19 y 20, para indicar los siglos XX y XXI, respectivamente. Los bits del 6 al 4 almacenan las decenas del siglo (1 o 2) y los bits del 3 al 0, las unidades (9 o 0).

YEAR Codifica las decenas y unidades del año, 00–99, en BCD con 8 bits. Los bits del 15 al 12 almacenan las decenas del año y los bits del 11 al 8, las unidades.

MONTH Codifica el mes, 01–12, en BCD con 5 bits. El bit 20 almacena las decenas del mes (0 o 1) y los bits del 19 al 16, las unidades.

DAY Codifica el día de la semana, 1–7, en BCD con 3 bits. El RTC no fuerza a que el número 1 sea un día en concreto —que normalmente suele equivaler al lunes o al domingo, dependiendo de la localización—. Por tanto, la relación entre los números del 1 al 7 y los días de la semana queda a criterio del usuario.

DATE Codifica el día del mes, 01–31, en BCD con 7 bits. Los bits 29 y 28 almacena las decenas (0 a 3) y los bits 27 al 24, las unidades.

De la información anterior se puede deducir que el RTC del microcontrolador ATSAM3X8E puede trabajar con fechas comprendidas entre el 1 de enero de 1900 y el 31 de diciembre de 2099, disponiendo de un calendario de 200 años.

A.4.2. Codificación de la hora

Antes de ver cómo se codifica la hora, conviene saber que el RTC integrado en el ATSAM3X8E es capaz de gestionar la hora tanto en el formato de 24 horas, como en el de 12 horas más indicador de AM/PM. El usuario deberá seleccionar cuál de los dos formatos prefiere utilizar, indicándolo en el bit `HRMOD` del *RTC Mode Register* (`RTC_MR`) —mostrado en la Figura A.4—. Si este bit se pone a 0, se utilizará el formato de 24 horas; si se pone a 1, el de 12 horas más AM/PM.

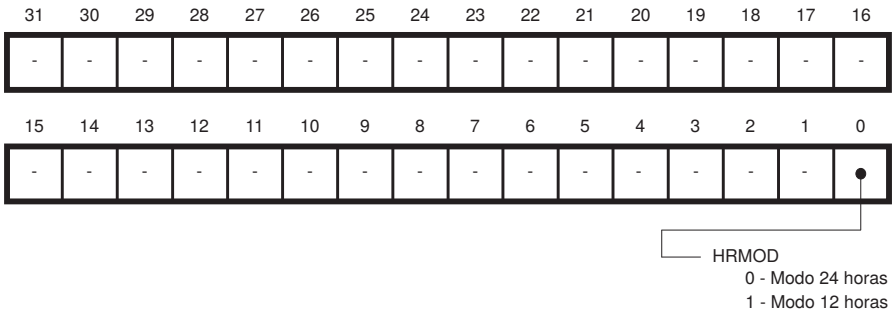


Figura A.4: Formato del registro RTC Mode Register

En cuanto a la hora en sí, el reloj en tiempo real la almacena en el registro *RTC Time Register* (`RTC_TIMR`), siguiendo el formato mostrado en la Figura A.5. Los campos[6] de este formato se describen a continuación.

`AMPM` En el caso de utilizar el formato de 12 horas, codifica si la hora es AM (0) o PM (1).

`HOUR` Codifica la hora en BCD con 6 bits: 01–12, en el modo de 12 horas, y 00–23, en el modo de 24 horas. Los bits 21 y 20 almacenan las decenas de la hora y los bits 19 al 16, las unidades.

`MIN` Codifica los minutos, 00–59, en BCD con 7 bits. Los bits 14 al 12 almacenan las decenas de minutos y los bits 11 a 8, las unidades.

`SEC` Codifica los segundos, 00–59, en BCD con 7 bits. Los bits del 6 al 4 almacenan las decenas de segundos y los bits del 3 al 0, las unidades.

A.4.3. Lectura de la fecha y hora

Debido a que el RTC es independiente del resto del sistema y asíncrono con respecto a este, para asegurar que la lectura obtenida de sus

[6] Como los campos numéricos del registro `RTC_TIMR` se codifican en BCD, conviene consultar al menos el breve resumen proporcionado al respecto.

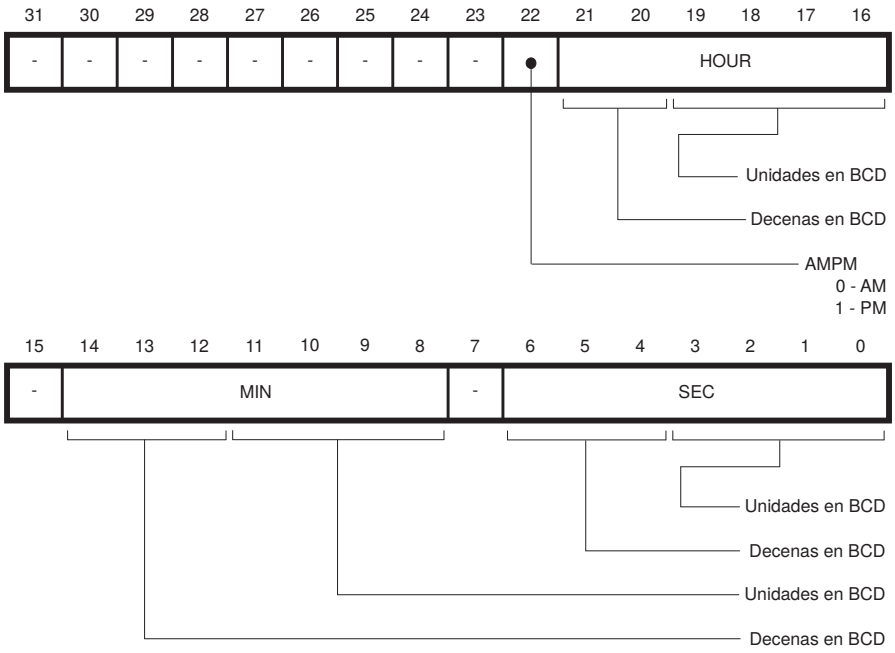


Figura A.5: Formato del registro RTC Time Register

registros es la correcta, es necesario leerlos 2 veces y verificar que ambas lecturas tienen la misma información. De lo contrario, podría ocurrir que al leer los registros de fecha u hora, se hubiera incrementado algún campo de alguno de los registros y que este cambio aún no se hubiera propagado al resto de campos —p.e., el procesador podría leer la hora 10:35:00 del registro `RTC_TIMR` justo cuando el RTC acababa de incrementar el campo de los segundos, pero aún no el de los minutos, por lo que en realidad serían las 10:36:00—. Por tanto, y tal como íbamos diciendo, es necesario leer 2 veces los registros del RTC y verificar que ambas lecturas tienen la misma información. Si la tienen, la lectura es correcta. De lo contrario, hay que realizar una tercera lectura que ya tendrá los valores correctos. Así pues, para leer la fecha o la hora hay que leer el registro `RTC_CALR` o el `RTC_TIMR` un mínimo de 2 veces y un máximo de 3 [4]. Un posible código para leer la fecha y la hora, en el que las constantes utilizadas han sido previamente definidas, sería:

```
1 read_date_time:
2   ldr r0, =RTC              @ r0 <- Dirección base del RTC
3   ldr r1, [r0, #RTC_CALR]   @ r1 <- Registro RTC_CALR
4   ldr r2, [r0, #RTC_TIMR]   @ r2 <- Registro RTC_TIMR
5   ldr r3, [r0, #RTC_CALR]   @ r3 <- Registro RTC_CALR (2a lectura)
6   ldr r4, [r0, #RTC_TIMR]   @ r4 <- Registro RTC_TIMR (2a lectura)
7   cmp r1, r3                @ Compara las lecturas de la fecha, y si
8   bne read_again            @  NO coinciden, salta a 3a lectura
```

```
  cmp r2, r4                @ Compara las lecturas de la hora, y si
  beq date_time_ok          @  coinciden, salta a date_time_ok
read_again:
  ldr r1, [r0, #RTC_CALR]   @ r1 <- Registro RTC_CALR (3a lectura)
  ldr r2, [r0, #RTC_TIMR]   @ r2 <- Registro RTC_TIMR (3a lectura)
date_time_ok:
  ...                       @ Fecha y hora correctas (en r1 y r2)
```

Una vez que se ha leído correctamente la información de los registros `RTC_CALR` y `RTC_TIMR`, simplemente hay que tener en cuenta lo comentado en los Apartados A.4.1 y A.4.2 para decodificar la fecha y la hora.

A.4.4. Actualización de la fecha y hora

Así como la lectura de los registros de fecha y hora debe realizarse de la forma indicada en el apartado anterior, y también como consecuencia de que el RTC es asíncrono con respecto al resto del sistema, para poder actualizar la fecha y la hora del RTC es necesario seguir estos pasos:

1. Inhibir la actualización de los registros de fecha y hora del RTC.
2. Esperar la confirmación de que se ha inhibido su actualización.
3. Escribir los nuevos valores de fecha y hora.
4. Activar la actualización de los registros del RTC.

A continuación se detalla cómo realizar cada uno de los pasos del proceso anterior.

Paso 1: Inhibir la actualización del RTC. Esto se consigue poniendo a 1 los bits `UPDCAL`, para la fecha, y `UPDTIM`, para la hora, del registro *RTC Control Register* (`RTC_CR`), representado en la Figura A.6.

Paso 2: Esperar la confirmación de que se ha inhibido la actualización de los registros del RTC. Cuando el RTC ejecute la orden dada en el paso anterior, detendrá la actualización de los registros de fecha y hora; pondrá a 1 el bit `ACKUPD` del *RTC Status Register* (`RTC_SR`), mostrado en la Figura A.7; y en el caso de que la generación de peticiones de interrupción esté activada, generará una petición de interrupción. Por lo tanto, este paso, consistente en esperar la confirmación de que el RTC ha detenido la actualización de los registros indicados en el paso anterior, se podrá llevar a cabo de dos formas. Bien mediante consulta de estado, comprobando periódicamente el estado del bit `ACKUPD`; bien mediante gestión de interrupciones, atendiendo la petición de interrupción del RTC y comprobando entonces si el bit `ACKUPD` está activo —ya que la petición de interrupción podría deberse a otra causa—.

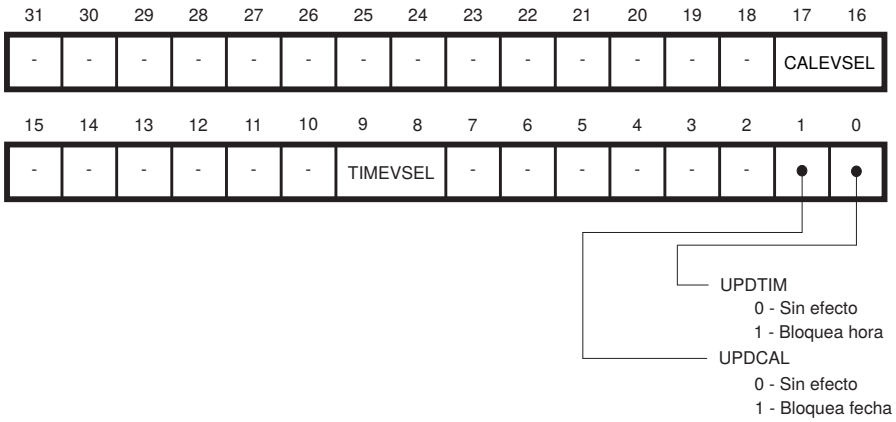


Figura A.6: Formato del registro RTC Control Register

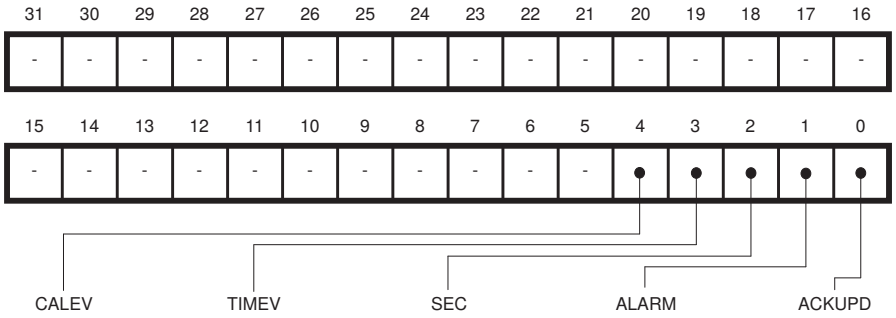


Figura A.7: Formato del registro RTC Status Register

En cualquiera de los dos casos, una vez se haya detectado que el RTC ha puesto a 1 el bit `ACKUPD`, se deberá poner este bit a 0 —ya que si no si hiciera, permanecería a 1 y la siguiente vez que fuera a utilizarse ya no cumpliría con su cometido—. Para poner el bit `ACKUPD` a 0, basta con escribir un 1 en el bit `ACKCLR`, del *RTC Status Clear Command Register* (`RTC_SCCR`), cuyo formato se muestra en la Figura A.8.

Paso 3: Escribir los nuevos valores de fecha y hora. Una vez el RTC ha detenido la actualización de sus registros `RTC_CALR` y `RTC_TIMR`, ya es posible escribir en ellos la nueva fecha y hora, para lo que habrá que tener en cuenta lo comentado en los Apartados A.4.1 y A.4.2 sobre su codificación.

Cuando se modifican los registros de fecha y hora, el RTC comprueba que los valores escritos sean correctos. De no ser así, activará el indicador correspondiente, *Non-valid Calendar* (`NVCAL`) o *Non-valid Time* (`NVTIM`), del registro *RTC Valid Entry Register* (`RTC_VER`), cuyo formato se mues-

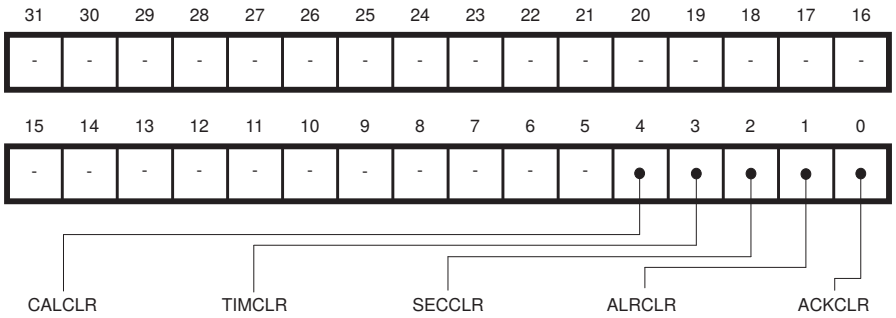


Figura A.8: Formato del registro RTC Status Clear Command Register

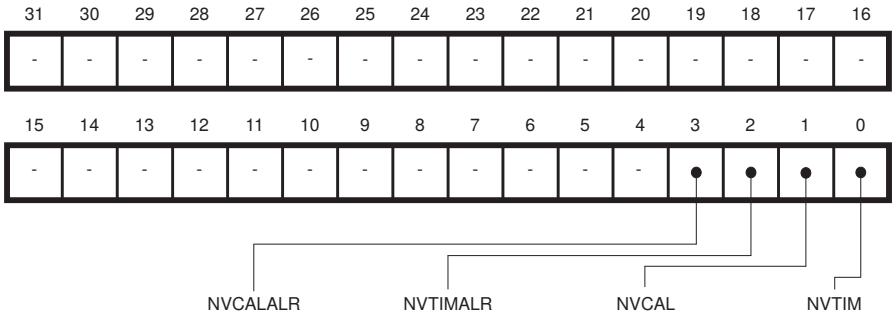


Figura A.9: Formato del registro RTC Valid Entry Register

tra en la Figura A.9. Mientras no se introduzcan valores correctos, el RTC se quedará bloqueado.

Paso 4: Activar la actualización de los registros del RTC. El último paso, una vez escritas correctamente una nueva fecha y hora, consiste en activar de nuevo la actualización de los registros `RTC_CALR` y `RTC_TIMR`. Esto se consigue poniendo a 0 los bits `UPDCAL`, para la fecha, y `UPDTIM`, para la hora, del registro *RTC Control Register* (`RTC_CR`), mostrado en la Figura A.6.

A.4.5. Configuración y detección de la alarma

El RTC posee la capacidad de establecer una alarma para un instante de tiempo basándose en los campos mes, día del mes, hora, minuto y segundo. Estos campos se pueden configurar escribiendo en los registros: *RTC Calendar Alarm Register* (`RTC_CALALR`) y *RTC Time Alarm Register* (`RTC_TIMALR`), cuyos formatos se muestran en las Figuras A.10 y A.11, respectivamente.

Como se puede ver en las Figuras A.10 y A.11, cada uno de los campos temporales tiene un bit de activación asociado. De esta forma,

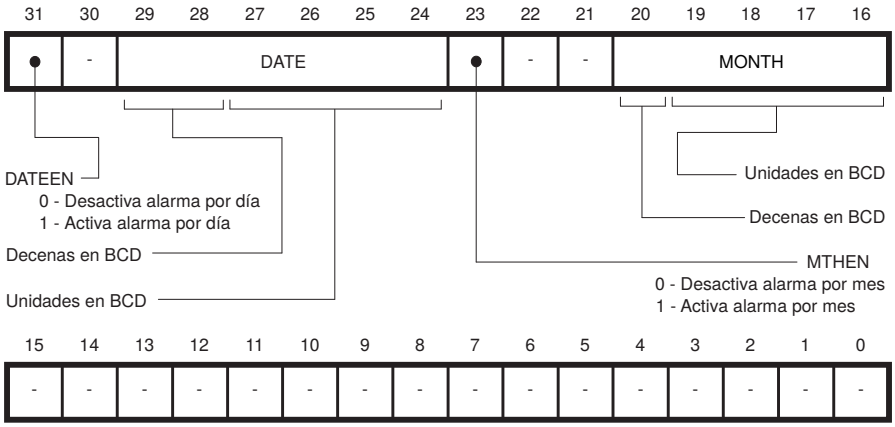


Figura A.10: Formato del registro RTC Calendar Alarm Register

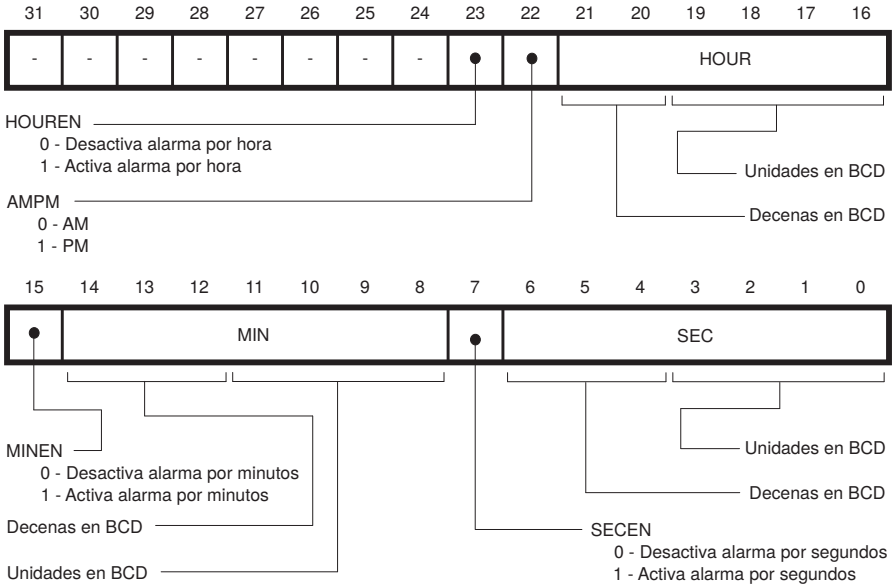


Figura A.11: Formato del registro RTC Time Alarm Register

es posible indicar si se quiere que el valor presente en cada uno de los campos forme parte o no de la definición de la alarma. Esto permite definir una alarma en la que no intervengan todos los campos. Así por ejemplo, se podría programar la alarma para que salte cada día 18 de cualquier mes, sin más que escribir un 1 en `DATEEN` —bit 31 del registro `RTC_CALALR`—, un «18» en BCD en `DATE` —bits 29 a 24 del mismo registro—, y el resto de bits de este registro y del `RTC_TIMR` a 0.

Tal y como ocurría en el caso de la escritura de la fecha y la hora, cuando se escribe en los registros `RTC_CALALR` o `RTC_TIMALR`, el RTC comprueba que los valores introducidos sean correctos. Si lo son, cuando la fecha y hora alcancen el instante indicado, el RTC activará el bit `ALARM` del registro *RTC Status Register* (`RTC_SR`), mostrado en la Figura A.7, y, en caso de estar activada la generación de peticiones de interrupción, producirá una petición de interrupción. En caso contrario, si los valores introducidos no fueran correctos, el RTC simplemente activará los indicadores correspondientes, `NVCALALR` *(Non Valid CALALR)* o `NVTIMALR` *(Non Valid TIMALR)*, del registro *RTC Valid Entry Register* (`RTC_VER`), mostrado en la Figura A.9.

Además, en el caso de que salte una alarma y el bit `ALARM` del registro `RTC_SR` esté a 1, el RTC activará el bit `SEC` de dicho registro. De esta forma, el RTC es capaz de indicar no solo que ha saltado la alarma, si no que ha saltado al menos dos veces desde que se puso a 0 el bit `ALARM`.

Por último, el programa que ha definido la alarma deberá comprobar si esta ha saltado, ya sea mediante consulta de estado —comprobando periódicamente el bit `ALARM` del registro `RTC_SR`— o mediante gestión de interrupciones —definiendo una rutina de tratamiento de la interrupción RTC que cuando se ejecute compruebe el bit `ALARM` del registro `RTC_SR` para ver si la petición de interrupción ha sido provocada por haber saltado la alarma—. En cualquier caso, e independientemente de la estrategia utilizada, una vez el programa haya detectado que ha saltado la alarma, deberá poner a 0 el bit `ALARM`, y en su caso el bit `SEC`, del registro `RTC_SR`. Para poner a 0 estos bits, se deberá escribir un 1 en los bits `ALRCLR` y `SECCLR`, respectivamente, del registro *RTC Status Clear Command Register* (`RTC_SCCR`), mostrado en la Figura A.8.

A.4.6. Configuración y detección de los eventos periódicos de fecha y hora

El RTC, además de permitir definir una alarma basada en que se alcance un instante determinado, como se ha visto en el apartado anterior, también puede detectar que se han producido diferentes tipos de eventos periódicos, pudiendo programarse para que genere una alarma ante eventos periódicos de fecha —véase el Cuadro A.7— y de hora —véase el Cuadro A.8—. Para programar un evento periódico de fecha se debe

Cuadro A.7: Tipos de eventos periódicos de fecha

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

Cuadro A.8: Tipos de eventos periódicos de hora

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

escribir el valor correspondiente a dicho evento, que se muestra en el Cuadro A.7, en el campo `CALEVSEL` del *RTC Control Register* (`RTC_CR`), mostrado en la Figura A.6. Para programar un evento periódico de hora se debe escribir el valor correspondiente a dicho evento, que se muestra en el Cuadro A.8, en el campo `TIMEVSEL` de dicho registro.

Al igual que ocurre con la alarma basada en un instante de tiempo, la notificación de que se ha producido un evento periódico se produce a través del registro *RTC Status Register* (`RTC_SR`), mostrado en la Figura A.7, donde, en caso de que se haya detectado un evento periódico de fecha, se activará el bit `CALEV` y en caso de que se haya producido un evento periódico de hora, se activará el bit `TIMEV`.

De nuevo, el programa que haya configurado alguno de los eventos periódicos del RTC deberá comprobar, bien por consulta de estado, o bien mediante gestión de interrupciones, si se ha producido dicho evento para llevar a cabo la tarea que quisiera realizar ante tal evento. Además, una vez que haya detectado que el evento se ha producido, deberá poner a 0 el correspondiente bit, `CALEV` o `TIMEV`, del registro `RTC_SR`, escribiendo para ello un 1 en el bit `CALCLR` o `TIMCLR`, respectivamente, del registro `RTC_SCCR` —Figura A.8—.

A.4.7. Interrupciones en el RTC

El RTC posee la capacidad de generar una petición de interrupción cuando se produce alguna de las siguientes circunstancias:

- Se ha inhibido la actualización de la fecha o la hora.
- Se ha alcanzado el instante programado como alarma.

- Se ha alcanzador por segunda vez el instante programado como alarma sin que esta haya sido atendida en la anterior ocasión.
- Se ha producido un evento periódico de calendario.
- Se ha producido un evento periódico de tiempo.

Para activar o desactivar la generación de una petición de interrupción cada vez que se produzca alguna de las anteriores circunstancias, se debe escribir en el registro *RTC Interrupt Enable Register* (`RTC_IER`), representado en la Figura A.12, y cuyos campos se describen a continuación.

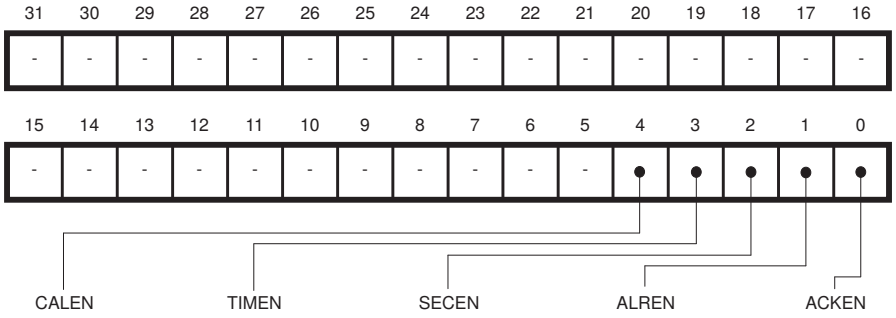


Figura A.12: Formato del registro RTC Interrupt Enable Register

ACKEN Si se pone a 1, el RTC generará una petición de interrupción cada vez que se active el bit `ACKUPD` del registro `RTC_SR`, es decir, cada vez que se detenga la actualización de los registros de fecha y hora, tal y como se describe en el Paso 2 del Apartado A.4.4.

ALREN Si se pone a 1, el RTC generará una petición de interrupción cada vez que se active el bit `ALARM` del registro `RTC_SR`, es decir, cada vez que se alcance el instante programado como alarma.

SECEN Si se pone a 1, el RTC generará una petición de interrupción cada vez que se active el bit `SEC` del registro `RTC_SR`, es decir, cada vez que se alcance por segunda vez el instante programado como alarma sin que se haya puesto a 0 el bit `ALARM` del registro `RTC_SR`.

CALEN Si se pone a 1, el RTC generará una petición de interrupción cada vez que se active el bit `CALEV` del registro `RTC_SR`, es decir, cada vez que se produzca el evento periódico indicado previamente en el campo `CALEVSEL` del registro `RTC_CR`.

TIMEN Si se pone a 1, el RTC generará una petición de interrupción cada vez que se active el bit `TIMEV` del registro `RTC_SR`, es decir, cada vez

que se produzca el evento periódico indicado previamente en el campo `TIMEVSEL` del registro `RTC_CR`.

Puesto que el RTC puede generar una petición de interrupción ante varias situaciones distintas, en la rutina de tratamiento de la interrupción del RTC se deberá consultar el registro `RTC_SR` para saber cuál de estas situaciones ha provocado la petición de interrupción. Es más, puesto que podrían haberse dado varias de estas circunstancias simultáneamente, una vez tratada una de las causas, conviene evaluar si alguna de las restantes también se ha dado. Por último, una vez completado el tratamiento de todas las causas que hayan provocado la interrupción, y antes de devolver el control al programa interrumpido, se deben poner a 0 los bits que hayan provocado la petición de interrupción —escribiendo en el registro `RTC_SCCR`, Figura A.8—.

A.4.8. Protección contra escritura

Para evitar que un programa modifique por error los registros `RTC_MR`, `RTC_CALALR` o `RTC_TIMALR`, el reloj en tiempo real del ATSAM3X8E proporciona un registro de protección contra escritura llamado *RTC Write Protect Mode Register* (`RTC_WPMR`), cuyo formato se muestra en la Figura A.13. Para que el RTC active o desactive la protección contra escritura cuando se escriba en este registro, el campo `WPKEY` debe tener el código ASCII correspondiente a «RTC» —`0x525443`—. El valor del campo `WPEN` determinará si debe activarse la protección, cuando se ponga a 1, o desactivarse, cuando se ponga a 0. La protección contra escritura está desactivada por defecto.

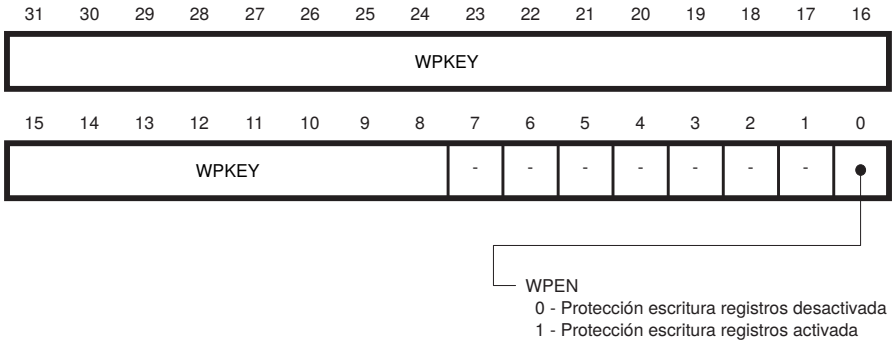


Figura A.13: Formato del registro RTC Write Protect Mode Register

A.5. El temporizador en tiempo real del ATSAM3X8E

El Temporizador en Tiempo Real (RTT, por *Real Time Timer*) del ATSAM3X8E es un temporizador simple y versátil, lo que permite utilizarlo de forma sencilla —y con plena libertad, puesto que no es utilizado por el sistema Arduino—. Se trata básicamente de un registro contador ascendente de 32 bits, con una frecuencia base de actualización de 32 768 Hz —la misma que la del RTC—. El periodo de actualización puede variarse, gracias a un divisor de frecuencia de 16 bits, entre un poco menos de una décima de milisegundo hasta casi dos segundos, por lo que se puede deducir que este RTT está pensado para realizar temporizaciones relacionadas con tiempos de usuario más que de sistema. Además, es capaz de generar interrupciones periódicas cada vez que se incrementa el valor de su contador o cuando se alcanza un valor fijado como alarma.

El controlador del RTT proporciona los siguientes cuatro registros, cuyas direcciones se muestran en el Cuadro A.9.

Mode Register **(`RTT_MR`)** Es el registro de control del dispositivo. Los 16 bits más bajos almacenan el valor del divisor de frecuencia. En estos bits se puede poner cualquier valor superior a 2. Si se pone el valor `0x8000`, la frecuencia de actualización del RTT será de un segundo. El bit 18 —`RTTRST`— de este registro sirve para reiniciar el RTT. Cuando este bit se pone a 1, el RTT pone a 0 su contador y actualiza el valor del divisor de frecuencia al valor indicado en los 16 bits de menor peso de este registro. El bit 17 —`RTTINCIEN`— sirve para habilitar o deshabilitar la generación de peticiones de interrupción con cada incremento. Por último, el bit 16 —`ALMIEN`— sirve para habilitar o deshabilitar la generación de peticiones de interrupción cada vez que se alcance el valor de la alarma.

Alarm Register **(`RTT_AR`)** Almacena el valor de la alarma, de 32 bits. Cuando el contador alcance este valor, y en el caso de estar habilitada la generación de peticiones de interrupción por alarma, el RTT generará una petición de interrupción.

Value Register **(`RTT_VR`)** Guarda el valor del contador, que se incrementa con cada ciclo de la señal de reloj —según la frecuencia base dividida por el divisor de frecuencia—, de forma cíclica.

Status Register **(`RTT_SR`)** Es el registro de estado del RTT. Su bit 1 —denominado `RTTINC`— indica que se ha incrementado el valor del contador, y su bit 0 —`ALMS`—, que se ha alcanzado el valor de la

Cuadro A.9: Registros del temporizador en tiempo real del ATSAM3X8E y sus direcciones de E/S

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

alarma. Ambas circunstancias se señalan con un 1. Cuando se lea este registro, estos bits se pondrán a 0 automáticamente.

A.6. Gestión de excepciones e interrupciones en el ATSAM3X8E

La arquitectura ARM especifica un modelo de excepciones que lógicamente incluye el tratamiento de las 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*— basado en 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, que se encuentra en una zona determinada de la memoria. Cuando se produce una excepción, el procesador carga el valor almacenado en el vector de excepción correspondiente y lo utiliza como dirección de inicio de la rutina de tratamiento. Al mismo tiempo, cada causa de excepción tiene asignada una prioridad que determina cuál de ellas será atendida primero en caso de detectarse varias a la vez, y que permite que una rutina de tratamiento sea interrumpida —expulsada— si se detecta una excepción con mayor prioridad, volviendo a aquella 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 aún no ha sido tratada, y como **activa** —*active*— cuando su rutina de tratamiento ya ha comenzado a ejecutarse. Puesto que existe la posibilidad de expulsión, en un momento dado puede haber más de una excepción activa en el sistema. También puede darse el caso de que una causa de excepción sea a la vez pendiente y activa, lo que se significa que mientras se

trataba una excepción debida a esa causa, se ha vuelto a detectar otra petición de interrupción originada por la misma causa que la que se está tratando.

Cuando el procesador, tras finalizar la ejecución de la instrucción en curso, detecte una solicitud de excepción, y siempre que no se está atendiendo a otra de mayor prioridad, guardará automáticamente en la pila, lo siguiente: I) los registros `r0` a `r3` y `r12`, II) la dirección de retorno, III) el registro de estado, y IV) el registro `LR`. A continuación, guardará en el registro `LR` un valor especial, llamado `EXC_RETURN`, saltará a la dirección guardada en el vector de interrupción y cambiará el estado de la excepción de pendiente a activa.

Gracias a que el hardware de un procesador ARM realiza estas acciones de forma automática, el código de una rutina de tratamiento en ensamblador de ARM tiene la misma estructura y se rige por el mismo convenio que el de una subrutina cualquiera. Por ejemplo, el procesador apila automáticamente los registros `r0` a `r3` porque el convenio seguido en ARM es que una subrutina puede modificar libremente estos registros. De esta forma, puesto que el hardware se encargará de apilar estos registros, y tras la vuelta, de desapilarlos, la rutina de tratamiento de interrupciones puede modificarlos libremente. Otro ejemplo es que para retornar de la rutina de tratamiento bastará con sobrescribir el registro `PC` con el contenido del registro `LR` —ya sea mediante «`mov pc, lr`» o, si se hubiera tenido que apilar `LR`, con «`pop {pc}`»—. A este respecto hay que tener en cuenta que cuando se vuelve de una rutina de tratamiento de excepciones, el procesador tiene que recuperar automáticamente el estado original, mientras que cuando vuelve de una subrutina, no tiene que hacer nada, por lo tanto, el procesador necesita poder distinguir si vuelve de una rutina de tratamiento o de una subrutina. Entonces, si la forma de retornar de una rutina de tratamiento de excepciones y de una subrutina es la misma, ¿cómo sabe el procesador si está volviendo de una rutina de tratamiento de excepciones o de una subrutina? La respuesta está en el contenido del registro `LR`. Si el contenido de este registro es `EXC_RETURN`, entonces sabe que está volviendo de una rutina de tratamiento de excepciones; si su contenido no es `EXC_RETURN`, está volviendo de una subrutina.

Algunas de las excepciones contempladas en el ATSAM3X8E, junto con su número de excepción, su número de petición de interrupción, su prioridad y el desplazamiento de su vector de interrupción se pueden ver en el Cuadro A.10. Es interesante observar que la arquitectura ha asignado a las excepciones que no son interrupciones un número de petición de interrupción negativo, quedando los números positivos para las interrupciones de dispositivos.

Cuadro A.10: Algunas de las excepciones del ATSAM3X8E, sus prioridades y el desplazamiento de sus vectores de interrupción

Número de excepción	IRQn	Tipo de excepción	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`
...	...	...	...	...

A.6.1. El controlador de interrupciones vectorizadas y anidadas (NVIC)

Como hemos visto en el apartado anterior, la arquitectura ARM especifica el modelo de tratamiento de las excepciones. Del mismo modo, y como parte del núcleo de un sistema ARM *Cortex-M3*, también describe un controlador de interrupciones llamado *Nested Vectored Interrupt Controller*, *NVIC*, al que cuando se implemente un sistema basado en ARM, se deberán conectar las líneas de petición de interrupciones de los dispositivos que formen parte de dicho sistema.

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 peticiones de interrupción según las 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 estos registros y su función.

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

Interrupt Set Enable Registers **(`ISER1`, `ISER0`)** Escribiendo un 1 en cualquier bit de uno de estos registros se habilita la interrupción asociada.

Interrupt Clear Enable Registers **(`ICER1`, `ICER0`)** Escribiendo un 1 en cualquier bit de uno de estos registros se deshabilita la interrupción asociada.

Cuando se lee cualquiera de los registros anteriores, se obtiene la máscara de interrupciones habilitadas, indicadas con 1 en los bits correspondientes.

Por otro lado, los siguientes conjuntos de registros permiten gestionar las interrupciones marcadas como pendientes:

Interrupt Set Pending Registers **(`ISPR1`, `ISPR0`)** Escribiendo un 1 en cualquier bit de uno de estos registros se marca la interrupción asociada como pendiente. Esto permite forzar el tratamiento de una interrupción aunque no se haya señalado físicamente.

Interrupt Clear Pending Registers **(`ICPR1`, `ICPR0`)** 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.

Cuando se lee cualquiera de los registros anteriores, se obtiene la lista de interrupciones pendientes, indicadas con 1 en los bits correspondientes. Conviene tener en cuenta que una interrupción puede estar en estado pendiente aún no estando habilitada, por lo que tendría su bit a 1 en la máscara de pendientes y su bit a 0 en la máscara de habilitadas, vista previamente.

En cuanto a las interrupciones activas, aunque esta lista se gestiona por hardware, se puede consultar en los registros de solo lectura:

Interrupt Active Bit Registers **(`ICPR1`, `ICPR0`)** Un 1 en un bit de uno de estos registros indica que la interrupción correspondiente está siendo tratada en el momento de la lectura.

Las prioridades asociadas a las interrupciones se pueden configurar en los siguientes registros:

Interrupt Priority Registers **(`IPR7`–`IPR0`)** Donde cada interrupción tiene asociado un campo de 8 bits. En la implementación actual, la prioridad se almacena solo en los 4 bits de mayor peso de cada campo, pudiendo estar entre 0 —máxima prioridad de las definidas por el usuario— y 15 —mínima prioridad—.

Por último, el siguiente registro permite generar interrupciones por software:

Software Trigger Interrupt Register **(`STIR`)** Escribiendo un valor en sus 9 bits menos significativos se generará una interrupción con dicho número.

Como se ha podido ver, 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 de registros, excepto para el de prioridades, que comprende 8 registros. Dado un número de interrupción, es relativamente sencillo calcular el registro y los bits que almacenan la información que se desea leer o modificar. Sin embargo, la arquitectura ARM también define una interfaz software estándar para microcontroladores Cortex *(CMSIS)*, que proporciona una serie de funciones y estructuras de datos en lenguaje C para, entre otras cosas, facilitar la gestión de las interrupciones. Las funciones del CMSIS más usadas para la gestión de interrupciones son:

- «`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.

A.6.2. Interrupciones del ATSAM3X8E en el entorno Arduino

Teniendo en cuenta las funciones del CMSIS para la gestión del NVIC descritas en el apartado anterior, lo único que se necesita para poder gestionar una determinada interrupción es conocer cuál es el número de

interrupción que se le ha asignado. Además, para que el procesador salte a la rutina de tratamiento de interrupción correspondiente, también será necesario configurar su vector de interrupción de forma que apunte a la dirección en la que se encuentre la rutina que se haya desarrollado para su tratamiento. Por tanto, para poder tratar una determinada excepción tan solo nos falta conocer el número de interrupción asociado a esa excepción y poder modificar su vector de interrupción para que apunte a nuestra rutina de tratamiento.

La interfaz software estándar para microcontroladores Cortex *(CMSIS)* de ARM también facilita estas tareas. Esta interfaz define un símbolo para cada excepción al que el fabricante asignará el número de interrupción que haya utilizado para dicha excepción. Además, también define para cada excepción el nombre que deberá tener su rutina de tratamiento, para que cuando se cree una rutina con uno de esos nombres, el vector de excepción correspondiente apunte automáticamente a esa rutina. De esta forma, si se quiere configurar y tratar una excepción determinada, bastará con utilizar el símbolo asociado a dicha excepción para configurar sus opciones —sin necesidad de saber su número de interrupción— y definir una rutina con el nombre de RTI asociada a dicha excepción para tratarla —sin tener que configurar explícitamente el vector de interrupción para que apunte a dicha rutina—. El Cuadro A.11 muestra para cada dispositivo integrado en el ATSAM3X8E, el símbolo asociado a ese dispositivo, su número de interrupción y el nombre que deberá tener su rutina de tratamiento de interrupciones. Teniendo en cuenta la información de dicho cuadro, si se quisieran gestionar por ejemplo las peticiones de interrupción generadas por el PIOB, se debería utilizar el símbolo «`PIOB_IRQn`» como parámetro de las funciones CMSIS vistas en el apartado anterior y se debería crear una función llamada «`PIOB_Handler()`» en la que se incluiría el código para tratar las interrupciones provocadas por el PIOB.

Cuadro A.11: Dispositivos del ATSAM3X8E y sus rutinas de tratamiento asociadas

Símbolo	n	Dispositivo	RTI
`SUPC_IRQn`	0	Supply Controller	«`SUPC_Handler()`»
`RSTC_IRQn`	1	Reset Controller	«`RSTC_Handler()`»
`RTC_IRQn`	2	Real Time Clock	«`RTC_Handler()`»
`RTT_IRQn`	3	Real Time Timer	«`RTT_Handler()`»
`WDT_IRQn`	4	Watchdog Timer	«`WDT_Handler()`»
`PMC_IRQn`	5	Power Management Controller	«`PMC_Handler()`»

Continúa en la siguiente página...

Dispositivos del ATSAM3X8E y sus rutinas de tratamiento asociadas (continuación)

Símbolo	n	Dispositivo	RTI
EFC0_IRQn	6	Enhanced Flash Controller 0	«EFC0_Handler()»
EFC1_IRQn	7	Enhanced Flash Controller 1	«EFC1_Handler()»
UART_IRQn	8	Universal Asynchronous Receiver Transmitter	«UART_Handler()»
SMC_IRQn	9	Static Memory Controller	«SMC_Handler()»
PIOA_IRQn	11	Parallel I/O Controller A	«PIOA_Handler()»
PIOB_IRQn	12	Parallel I/O Controller B	«PIOB_Handler()»
PIOC_IRQn	13	Parallel I/O Controller C	«PIOC_Handler()»
PIOD_IRQn	14	Parallel I/O Controller D	«PIOD_Handler()»
USART0_IRQn	17	USART 0	«USART0_Handler()»
USART1_IRQn	18	USART 1	«USART1_Handler()»
USART2_IRQn	19	USART 2	«USART2_Handler()»
USART3_IRQn	20	USART 3	«USART3_Handler()»
HSMCI_IRQn	21	Multimedia Card Interface	«HSMCI_Handler()»
TWI0_IRQn	22	Two-Wire Interface 0	«TWI0_Handler()»
TWI1_IRQn	23	Two-Wire Interface 1	«TWI1_Handler()»
SPI0_IRQn	24	Serial Peripheral Interface	«SPI0_Handler()»
SSC_IRQn	26	Synchronous Serial Controller	«SSC_Handler()»
TC0_IRQn	27	Timer Counter 0	«TC0_Handler()»
TC1_IRQn	28	Timer Counter 1	«TC1_Handler()»
TC2_IRQn	29	Timer Counter 2	«TC2_Handler()»
TC3_IRQn	30	Timer Counter 3	«TC3_Handler()»
TC4_IRQn	31	Timer Counter 4	«TC4_Handler()»
TC5_IRQn	32	Timer Counter 5	«TC5_Handler()»
TC6_IRQn	33	Timer Counter 6	«TC6_Handler()»
TC7_IRQn	34	Timer Counter 7	«TC7_Handler()»
TC8_IRQn	35	Timer Counter 8	«TC8_Handler()»
PWM_IRQn	36	Pulse Width Modulation Controller	«PWM_Handler()»
ADC_IRQn	37	ADC Controller	«ADC_Handler()»
DACC_IRQn	38	DAC Controller	«DACC_Handler()»
DMAC_IRQn	39	DMA Controller	«DMAC_Handler()»
UOTGHS_IRQn	40	USB OTG High Speed	«UOTGHS_Handler()»

Continúa en la siguiente página...

Dispositivos del ATSAM3X8E y sus rutinas de tratamiento asociadas (continuación)

Símbolo	n	Dispositivo	RTI
`TRNG_IRQn`	41	True Random Number Generator	«`TRNG_Handler()`»
`EMAC_IRQn`	42	Ethernet MAC	«`EMAC_Handler()`»
`CAN0_IRQn`	43	CAN Controller 0	«`CAN0_Handler()`»
`CAN1_IRQn`	44	CAN Controller 1	«`CAN1_Handler()`»

El siguiente código muestra un ejemplo de cómo se podrían configurar las interrupciones del PIOB dentro de la función «`setup`» de un proyecto Arduino.

```
void setup() {
        /* [...] */
        NVIC_DisableIRQ(PIOB_IRQn);
        NVIC_SetPriority(PIOB_IRQn, 0);
        /* Acciones que deban hacerse antes de activar las interr. */
        NVIC_ClearPendingIRQ(PIOB_IRQn);
        NVIC_EnableIRQ(PIOB_IRQn);
        /* [...] */
}
```

El código anterior realiza las siguientes acciones:

1. Deshabilita el tratamiento de las interrupciones generadas por el puerto PIOB. —Por si estuvieran habilitadas.
2. Establece la prioridad 0 —la máxima de las configurables— para las peticiones de interrupción generadas por el PIOB.
3. En lugar del comentario del ejemplo, aquí vendrían aquellas acciones que deberían hacerse antes de activar el tratamiento de las interrupciones del PIOB. Por ejemplo, configurar en qué circunstancias el PIOB deberá generar peticiones de interrupción; especificar el valor inicial de una variable global que vaya a ser utilizada por la rutina de tratamiento; etcétera.
4. Borra el indicador de que hay peticiones de interrupción generadas por el PIOB pendientes de atender. —Por si las hubiera.
5. Finalmente, la última línea habilita el tratamiento de las interrupciones generadas por el PIOB.

Para acabar, los siguientes fragmentos de código muestran cómo se debería declarar la rutina «`PIOB_Handler`», a la que se llamará cuando se produzca una petición de interrupción del PIOB, en lenguaje C y en lenguaje ensamblador, respectivamente:

```
// RTI en C
void PIOB_Handler() {
        /* Código para el tratamiento de las interrupciones del PIOB */
}
```

```
@ RTI en ensamblador
PIOB_Handler:
        push {lr}
        /* Código para el tratamiento de las interr. del PIOB */
        pop {pc}
```

A.7. El controlador de DMA del ATSAM3X8E

El microcontrolador ATSAM3X8E incorpora un controlador de acceso directo a memoria (DMA, por las siglas en inglés de *Direct Memory Access*) llamado *AHB DMA Controller (DMAC)*. Se trata de un controlador de DMA con seis canales, cuatro de ellos con 8 bytes de almacenamiento intermedio y los otros dos —los canales 3 y 5— con 32 bytes. Cada canal puede configurarse para transferir datos entre un dispositivo fuente y uno destino, permitiéndose cualquier configuración de dispositivos fuente y destino: de memoria a memoria, de memoria a dispositivo, de dispositivo a memoria y de dispositivo a dispositivo.

Este controlador de DMA, además de gestionar adecuadamente los accesos a los distintos buses y dispositivos, se puede programar para que genere peticiones de interrupción para avisar de que ha finalizado alguna de las transferencias programadas o de que se ha producido algún error al intentar realizar la transferencia.

En cuanto a sus registros, además de un conjunto de registros de control globales, proporciona seis registros de control por cada canal para programar las transferencias que deberá gestionar cada uno de estos. Escribiendo en estos registros, además de indicarle al canal correspondiente cuáles son los dispositivos fuente y destino y las direcciones de los datos de la transferencia, se pueden programar transferencias que involucren a múltiples bloques de datos, tanto contiguos como distantes.

Para una información más detallada sobre su funcionamiento, que queda fuera del objetivo de esta breve introducción, se puede consultar el Apartado 22. *AHB DMA Controller (DMAC)* de [4].

Apéndice B

Breve guía de programación en ensamblador

Índice

Este apéndice proporciona una breve guía en la que se muestra cómo se implementa en ensamblador determinados aspectos de programación.

B.1. Variables

Los programas utilizan variables para almacenar los datos con los que trabajan. Reciben el nombre de variables debido al hecho de que los programas pueden modificar los datos que almacenan. Desde el punto de vista del programador, cada variable se identifica con un nombre, que él mismo elige. Cuando el programador usa una variable en un programa, normalmente está interesado en el valor que tomará dicha variable conforme se vaya ejecutando el programa. De lo que no suele ser consciente, es que en realidad, una variable es simplemente una dirección de

memoria a la que se le ha asociado un nombre, elegido por él, que la identifica. Así pues, el valor de una variable es por tanto el contenido de dicha dirección de memoria.

Cuando se utiliza una variable en una asignación, lo que se hace con dicha variable depende de si aparece a la derecha o a la izquierda de dicha asignación. Cuando una variable aparece a la derecha de una asignación, lo que se hace es utilizar su valor; cuando aparece a la izquierda, lo que se hace es cambiar su valor con el resultado de la expresión que haya a la derecha de la asignación. Por tanto, en el primer caso, se leerá de memoria, en el segundo, se escribirá. Si se considera por ejemplo la siguiente asignación:

```
var1 = var2 + 5
```

Lo que realmente se realiza a nivel de la máquina es lo siguiente:

AB_add.s

```
        .data
var1:   .space  4
var2:   .word   ????

        .text
        ldr r0, =var2   @ En r0 se pone la dir. de la variable var2
        ldr r0, [r0]    @  y se lee su contenido (su valor)
        add r0, r0, #5  @ Se realiza la suma
        ldr r1, =var1   @ Se pone la dir. de la variable var1 en r1
        str r0, [r1]    @  y se escribe su nuevo valor
        wfi
```

B.1.1. Tipos de variables

Según la naturaleza de la información que se quiera almacenar, las variables pueden ser de diferentes tipos. Si se quiere guardar y trabajar con números enteros, lo más eficiente será usar la representación natural del procesador y trabajar con palabras, que en ARM son de 32 bits. Si se quieren usar caracteres de texto, entonces se deberá usar el formato de datos fijado por algún estándar de codificación de caracteres, por ejemplo, el ASCII, donde cada carácter ocupa 8 bits.

El siguiente ejemplo, que pasa a mayúsculas una letra dada, utiliza datos del tipo carácter. En C se puede pasar una letra a mayúsculas utilizando la siguiente expresión:

```
char1 = toupper(char2);
```

En Python, lo mismo se haría de la siguiente forma:

```
char1 = char2.upper()
```

Los códigos anteriores llaman a sendas funciones, que además de convertir un carácter a mayúsculas, incluyendo los caracteres acentuados, no modifican aquellos caracteres que no son letras. El siguiente código en ensamblador de ARM muestra una posible implementación de lo anterior, incluyendo una subrutina muy básica de conversión a mayúsculas que solo funcionará correctamente si el carácter que se quiere pasar a mayúsculas está entre la 'a' y la 'z', o entre la 'A' y la 'Z', lo que no incluye a los caracteres acentuados —como se puede comprobar consultando un tabla con el código ASCII—. La subrutina de conversión tiene en cuenta que para pasar a mayúsculas un carácter entre 'a' y 'z', basta con poner a 0 el bit 5 de dicho carácter.

AB_upper.s

```
1        .data
2 char1:  .space 1
3 char2:  .byte  ????     @ Un carácter entre 'a' y 'z'
4
5        .text
6 main:   ldr  r0, =char2 @ En r0 se pone la dir. de la variable char2
7        ldrb r0, [r0]   @  y se lee su contenido (un byte)
8        bl   upper      @ Llama a upper(char2)
9        ldr  r1, =char1 @ Se pone la dir. de la variable char1 en r1
10       strb r0, [r1]   @  y se escribe el carácter en mayúsculas
11       wfi
12
13 upper:  ldr  r1, =0xDF  @ Máscara para poner a 0 el bit 5
14       and  r0, r1     @ AND del carácter con la máscara
15       mov  pc, lr
```

B.1.2. Conjuntos de datos del mismo tipo: vectores y cadenas

A menudo se debe trabajar con conjuntos de elementos del mismo tipo, sean caracteres o enteros. Los modos de direccionamiento de ARM ofrecen una forma sencilla de hacer esto. A continuación se muestra a modo de ejemplo, cómo iniciar un vector de 3 elementos con los primeros 3 números naturales, y uno de 3 caracteres con las primeras letras mayúsculas:

AB_vec_cad.s

```
1        .data
2 vec:    .space 3*4        @ Espacio para 3 enteros
3 cad:    .space 3          @  y para 3 caracteres
4
5        .text
6        ldr  r0, =vec     @ r0 tiene la dirección de inicio del vector
7        ldr  r1, =1       @ Se pone un 1 en r1
```

```
8         str  r1, [r0]     @ Se escribe el 1 en la primera posición
9         add  r1, r1, #1
10        str  r1, [r0, #4] @ Un 2 en la segunda, desplazamiento 4
11        add  r1, r1, #1
12        str  r1, [r0, #8] @ Y un 3 en la tercera, desplazamiento 8
13
14        ldr  r0, =cad     @ Ahora con cad, su dirección en r0
15        ldr  r1, ='A'     @ Se pone el carácter 'A' en r1
16        strb r1, [r0]     @ Se escribe el byte 'A' en la 1era posición
17        add  r1, r1, #1
18        strb r1, [r0, #1] @ El 'B' en la segunda, desplazamiento 1
19        add  r1, r1, #1
20        strb r1, [r0, #2] @ Y el 'C' en la tercera, desplazamiento 2
```

B.1.3. Datos estructurados

Un dato estructurado es un dato formado por un conjunto de datos, generalmente de distinto tipo, que permite acceder de forma individual a los datos que lo forman. Un ejemplo de declaración de un dato estructurado en C sería el siguiente:

```
1 struct Pieza {         // Estructura de ejemplo
2   char nombre[10], ch;
3   int val1, val2;
4 }
```

Dada la declaración anterior del dato estructurado «`Pieza`», se podría definir una variable de dicho tipo, p.e. «`p`», y acceder a los componentes de la variable «`p`» utilizando «`p.nombre`», «`p.ch`», «`p.val1`» y «`p.val2`».

En Python no existen datos estructurados como tales, sin embargo, es posible agrupar varios componentes en un contenedor creando una instancia de una clase vacía o, mejor aún, por medio de la función llamada «`namedtuple()`» del módulo «`collections`»:

```
1 from collections import namedtuple
2 Pieza = namedtuple('Pieza', ['nombre', 'ch', 'val1', 'val2'])
```

En este caso, se podría crear una instancia de la clase «`Pieza`», p.e., «`p`», y acceder a sus componentes por medio de «`p.nombre`», «`p.ch`», «`p.val1`» y «`p.val2`».

Un dato estructurado se implementa a bajo nivel como una zona de memoria contigua en la que se ubican los distintos datos que lo forman. El siguiente código muestra cómo se podría reservar espacio para varias variables del dato estructurado «`Pieza`» y cómo acceder a sus componentes. Dicho ejemplo comienza reservando espacio para un vector con tres datos estructurados del tipo «`Pieza`». A continuación, el código del

programa modifica el valor de los distintos componentes de cada uno de los elementos de dicho vector. Para simplificar el código, este escribe la misma información en todos los elementos del vector y no modifica el campo «`nombre`». Así pues, en el ejemplo se puede ver:

1. Cómo se inicializa una serie de constantes utilizando la directiva «**`.equ`**» con los desplazamientos necesarios para poder acceder a cada uno de los componentes del dato estructurado a partir de su dirección de comienzo.
2. Cómo se inicializa otra constante, «`pieza`», con el tamaño en bytes que ocupa el dato estructurado, que servirá para indicar cuánto espacio se debe reservar para cada dato de dicho tipo.
3. Cómo se reserva espacio para un vector de 3 datos estructurados.
4. Cómo se escribe en las componentes «`ch`», «`val1`» y «`val2`» de los tres elementos del vector anterior.

AB_struct.s

```
1
2         .data
3         @ Constantes (se definen aquí para organizar la zona de datos,
4         @ pero no consumen memoria)
5         .equ    nombre, 0
6         .equ    ch, 10
7         .equ    val1, 12        @ Alineado a multiplo de 4
8         .equ    val2, 16
9         .equ    pieza, 20       @ El tamaño de la estructura
10        @ Datos
11 vecEj:  .space  3 * pieza       @ Vector de 3 estructuras
12
13        .text
14        ldr  r0, =vecEj         @ r0 apunta al inicio del vector
15        ldr  r1, ='A'           @ Caracter a poner en ch
16        ldr  r2, =1000          @ Número a poner en val1
17        ldr  r3, =777           @ Número a poner en val2
18        strb r1, [r0, #ch]      @ Se usan las constantes (.equ) para
19        str  r2, [r0, #val1]    @  escribir los datos sin tener que
20        str  r3, [r0, #val2]    @  memorizar los desplazamientos
21        add  r0, #pieza         @ Se pasa al siguiente elemento
22        strb r1, [r0, #ch]      @ Y se repite lo mismo
23        str  r2, [r0, #val1]
24        str  r3, [r0, #val2]
25        add  r0, #pieza         @ Tercer elemento...
26        strb r1, [r0, #ch]
27        str  r2, [r0, #val1]
28        str  r3, [r0, #val2]
29        wfi
```

B.1.4. Poniendo orden en el acceso a las variables

En los primeros ejemplos se ha seguido un método sencillo pero poco práctico para acceder a las variables, dado que cada lectura requiere del acceso previo a la dirección. Así por ejemplo, para leer la variable «`char2`» del programa «`0B_upper.s`», primero se cargaba la dirección de dicha variable en un registro y luego se utilizaba dicho registro para indicar la dirección de memoria desde la que se debía leer su contenido; más adelante, para escribir en la variable «`char1`», primero se cargaba la dirección de dicha variable en un registro y luego se utilizaba dicho registro para indicar la dirección de memoria en la que se debía escribir el resultado. Cuando se tiene un programa —o una subrutina, como se verá en su momento— que debe acceder frecuentemente a sus variables en memoria, resulta más práctico usar un registro que contenga la dirección base de todas ellas, y usar desplazamientos para los accesos. A continuación se muestra un ejemplo en el que se utiliza dicho método para acceder a: un entero, «`val`», un carácter, «`chr`», un vector de enteros, «`vect`», y una cadena, «`cad`». En el ejemplo se utiliza la etiqueta «`orig`» para marcar el origen del bloque de variables. El nombre de cada variable se usa en realidad para identificar un desplazamiento constante desde el origen hasta dicha variable, mientras que para las etiquetas de cada variable se utiliza su nombre precedido por «_». El ejemplo utiliza el registro `r7` como registro base.

AB_desp.s

```
1         .data
2         @ Constantes con los desplazamientos con respecto a orig
3         .equ ent, _ent - orig
4         .equ chr, _chr - orig
5         .equ vect, _vect - orig
6         .equ cad, _cad - orig
7         @ Número de elementos del vector y de la cadena
8         .equ vectTam, 10
9         .equ cadTam, 8
10        @ Datos
11  orig:
12  _ent:   .space 4
13  _chr:   .space 1
14        .balign 4
15  _vect:  .space vectTam * 4
16  _cad:   .space cadTam
17
18        .text
19        ldr  r7, =orig          @ r7 apunta al comienzo de los datos
20        ldr  r1,=1000           @ Se escribe el número 1000
21        str  r1, [r7, #ent]     @  en el entero
22        ldr  r1, ='A'           @ y la letra 'A'
23        strb r1, [r7, #chr]     @  en el caracter
```

```
24        ldr  r0, =vect          @ Para usar un vector, se carga su
25        add  r0, r0, r7         @  origen en r0
26        ldr  r1, =1             @  y se escribe 1, 2 y 3 como antes
27        str  r1, [r0]           @ Se escribe el 1 en la primera posición
28        add  r1, r1, #1
29        str  r1, [r0, #4]       @ El 2 en la segunda, desplazamiento 4
30        add  r1, r1, #1
31        str  r1, [r0, #8]       @ Y el 3 en la tercera, desplazamiento 8
32        ldr  r0, =cad           @ Ahora se carga la dir. de la cadena
33        add  r0, r0, r7         @  en el registro r0
34        ldr  r1, ='A'           @  y se escribe 'A', 'B' y 'C'
35        strb r1, [r0]           @ Se escribe el byte 'A'
36        add  r1, r1, #1
37        strb r1, [r0, #1]       @ 'B' en la segunda, desplazamiento 1
38        add  r1, r1, #1
39        strb r1, [r0, #2]       @ Y 'C' en la tercera, desplazamiento 2
40        wfi
```

B.2. Estructuras de programación

Una vez que se ha visto cómo se pueden usar datos de diversos tipos y cómo organizar los programas para poder acceder a ellos de forma más sencilla —desde el punto de vista del programador—, se verá ahora cómo implementar las estructuras de control del flujo más comunes en los lenguajes de alto nivel.

B.2.1. Estructuras condicionales

La primera de las estructuras condicionales nos permite, simplemente, ejecutar o no cierta parte del código en función de una expresión que se evalúa a verdadero, en cuyo caso se ejecuta el código, o a falso, caso en el que no se ejecuta. Se trata de la estructura condicional si, o *if* en inglés. La estructura *if* se escribe en C como:

if

```
if (condición) {
  //
  // Código a ejecutar si se cumple la condición
  //
}
```

Y en Python:

```
if condición:
  #
  # Código a ejecutar si se cumple la condición
  #
```

La implementación en ensamblador de ARM es evidente. Consiste en evaluar la condición y, si el resultado es falso, saltar más allá del código condicionado. En el caso de que sea una condición simple y se utilice una sola instrucción de salto condicional, la condición del salto será la negación de la que se evalúa en el lenguaje de alto nivel. Así, si se evalúa una igualdad, se saltará en caso de desigualdad; si se evalúa una condición de menor que, se saltará en caso de mayor o igual, etcétera. A continuación se muestran un par de ejemplos tanto en Python como en ensamblador en los que se utiliza la estructura condicional *if*. El primero de los ejemplos muestra cómo ejecutar un fragmento de código solo si el valor de «a» es igual a «b».

```
if a == b:
  #
  # Código a ejecutar si a == b
  #
```

AB_if_equal.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        bne  finsi      @  y se salta si se da la condición contraria
        @
        @ Código a ejecutar si a == b
        @
finsi:  @ Resto del programa
        wfi
```

El segundo de los ejemplos, mostrado a continuación, muestra cómo ejecutar un fragmento de código solo si el valor de «a» es menor que «b».

```
if a < b:
  #
  # Código a ejecutar si a < b
  #
```

AB_if_lessthan.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        bge  finsi      @  y se salta si se da la condición contraria
        @
        @ Código a ejecutar si a < b
        @
finsi:  @ Resto del programa
        wfi
```

Por otro lado, cuando lo que se desea es ejecutar alternativamente un trozo de código u otro en función de que una condición se evalúe a

verdadero o falso, se puede utilizar la estructura *if-else*. Un fragmento de código se ejecuta en caso de que la condición sea verdadera y otro, en caso de que la condición sea falsa, de manera que nunca se ejecutan ambos fragmentos. Dicha estructura se escribe en C como:

if-else

```
if (condición) {
  //
  // Código a ejecutar si se cumple la condición
  //
} else {
  //
  // Código a ejecutar si no se cumple la condición
  //
}
```

Y en Python:

```
if condición:
  #
  # Código a ejecutar si se cumple la condición
  #
else:
  #
  # Código a ejecutar si no se cumple la condición
  #
```

La implementación en ensamblador utiliza en este caso una evaluación y un salto condicional al segundo bloque, saltándose de esta forma el primero de los dos bloques, que debe terminar necesariamente con un salto incondicional para no ejecutar el bloque alternativo. Si se sigue una estructura similar al *if* visto antes, se pondría primero el código correspondiente a la evaluación verdadera y a continuación el otro. Se podría, sin embargo, invertir el orden de los bloques de código de forma que se ejecute primero el correspondiente a la evaluación falsa y luego el de la verdadera. En esta segunda implementación, el salto se evaluaría con la misma lógica que la comparación en el lenguaje de alto nivel. Así, el siguiente programa en Python:

```
if a == b:
  #
  # Código a ejecutar si a == b
  #
else:
  #
  # Código a ejecutar si a != b
  #
```

Se escribiría en ensamblador de ARM como:

AB_if_equal_else.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        bne  else       @  y se salta si se da la condición contraria
        @
        @ Código a ejecutar si a == b
        @
        b finsi         @ Se salta el segundo bloque
else:   @
        @ Código a ejecutar si a != b
        @
finsi:  @ Resto del programa
        wfi
```

O, invirtiendo la posición de los bloques:

AB_if_equal_else2.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        beq  if         @  y se salta si se da la condición
        @
        @ Código a ejecutar si a != b
        @
        b finsi         @ Se salta el segundo bloque
if:     @
        @ Código a ejecutar si a == b
        @
finsi:  @ Resto del programa
        wfi
```

Por otro lado, en el caso de que el programa en Python fuera:

```
if a < b:
  #
  # Código a ejecutar si a < b
  #
else:
  #
  # Código a ejecutar si a >= b
  #
```

El equivalente en ensamblador sería:

AB_if_lessthan_else.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
```

```
        bge  else       @  y se salta si se da la condición contraria
        @
        @ Código a ejecutar si a < b
        @
        b finsi         @ Se salta el segundo bloque
else:   @
        @ Código a ejecutar si a >= b
        @
finsi:  @ Resto del programa
        wfi
```

O, invirtiendo la posición de los bloques:

AB_if_lessthan_else2.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        blt  if         @  y se salta si se da la condición
        @
        @ Código a ejecutar si a >= b
        @
        b finsi         @ Se salta el segundo bloque
if:     @
        @ Código a ejecutar si a < b
        @
finsi:  @ Resto del programa
        wfi
```

B.2.2. Estructuras condicionales concatenadas

Con mucha frecuencia las circunstancias a evaluar no son tan sencillas como la dicotomía de una condición o su negación, sino que en caso de no cumplirse una condición, se debe evaluar una segunda, luego tal vez una tercera, etcétera. De esta forma, es frecuente encadenar estructuras *if-else* de manera que en caso de no cumplirse una condición se evalúa una segunda dentro del código alternativo correspondiente al *else*. La frecuencia de esta circunstancia hace que en algunos lenguajes de programación existan las construcciones *if-elseif-elseif-...*, pudiendo concatenarse tantas condiciones como se quiera, y terminando o no en una sentencia *else* que identifica un código que debe ejecutarse en el caso de que ninguna condición se evalúe como verdadera.

if-elseif

La implementación de estas estructuras es tan sencilla como seguir un *if-else* como el ya visto y, en el código correspondiente al *else* implementar un nuevo *if-else* o un *if* en el caso de que no haya *else* final. En este caso, se recomienda utilizar la codificación del *if-else* poniendo en primer lugar el código correspondiente al *if*, como en la primera opción del apartado anterior. En realidad, es posible hacerlo de cualquier forma,

incluso mezclarlas, pero en el caso de no proceder de forma ordenada será mucho más fácil cometer errores.

Se muestran a continuación un par de ejemplos, con y sin *else* final. El primero de los casos, con *else* final, sería en C:

```c
if (a < b) {
  // Código a ejecutar si la primera condición es verdadera
} else if (a > c)  {
  // Código a ejecutar si la segunda condición es verdadera
} else if (a < 2500)
  // Código a ejecutar si la tercera condición es verdadera
} else {
  // Código en caso de que todas las anteriores sean falsas
}
```

En Python:

```python
if a < b:
  # Código a ejecutar si la primera condición es verdadera
elif a > c:
  # Código a ejecutar si la segunda condición es verdadera
elif a < 2500:
  # Código a ejecutar si la tercera condición es verdadera
else:
  # Código en caso de que todas las anteriores sean falsas
}
```

Y en ensamblador de ARM:

AB_if_elseif.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        bge  else1      @  y se salta si se da condición contraria
        @ Código a ejecutar si la primera condición es verdadera
        b    finsi      @ Se saltan el resto de bloques
else1:  cmp  r0, r2     @ r0 es a y r2 es c, se evalúa la condición
        ble  else2      @  y se salta si se da la condición contraria
        @ Código a ejecutar si la segunda condición es verdadera
        b    finsi      @ Se saltan el resto de bloques
else2:  ldr  r3, =2500
        cmp  r0, r3     @ r0 es a y r3 vale 2500, se evalúa la condición,
        bge  else3      @  y se salta si se da la condición contraria
        @ Código a ejecutar si la tercera condición es verdadera
        b    finsi      @ Se salta el último bloque
else3:  @ Código en caso de que todas las anteriores sean falsas
finsi:  @ Resto del programa
        wfi
```

El segundo de los ejemplos no tiene *else* final. El código en C sería:

```c
if (a < b) {
  // Código a ejecutar si la primera condición es verdadera
} else if (a > c)  {
  // Código a ejecutar si la segunda condición es verdadera
} else if (a < 2500)
  // Código a ejecutar si la tercera condición es verdadera
}
```

En Python:

```python
if a < b:
  # Código a ejecutar si la primera condición es verdadera
elif a > c:
  # Código a ejecutar si la segunda condición es verdadera
elif a < 2500:
  # Código a ejecutar si la tercera condición es verdadera
}
```

Y en ensamblador de ARM:

AB_if_elseif2.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
        bge  else1      @  y se salta si se da la condición contraria
        @ Código a ejecutar si la primera condición es verdadera
        b    finsi      @ Se salta el resto de bloques
else1:  cmp  r0, r2     @ r0 es a y r2 es c, se evalúa la condición
        ble  else2      @  y se salta si se da la condición contraria
        @ Código a ejecutar si la segunda condición es verdadera
        b    finsi      @ Se salta el resto de bloques
else2:  ldr  r3, =2500
        cmp  r0, r3     @ r0 es a y r3, 2500, se evalúa la condición,
        bge  finsi      @  y se salta si se da la condición contraria
        @ Código a ejecutar si la tercera condición es verdadera
finsi:  @ Resto del programa
        wfi
```

B.3. Estructuras iterativas

Las estructuras de programación iterativas o repetitivas son las que ejecutan repetidas veces el mismo bloque de código mientras se cumpla una condición de permanencia —o hasta que se dé una condición de salida—. En estas estructuras se tiene siempre un salto hacia una dirección anterior para permitir que se vuelva a ejecutar el código previo.

Además, se evalúa una condición que determina, de una u otra forma, si el código se sigue repitiendo o no.

B.3.1. Estructura 'for'

Cuando existe un índice asociado a la estructura repetitiva, de tal manera que las iteraciones continúan en función de alguna condición asociada a él, se tiene una estructura *for*. Normalmente, el índice se va incrementando y la iteración termina cuando alcanza un cierto valor, aunque no tiene por qué ser necesariamente así. En el caso más sencillo, el número de iteraciones se conoce *a priori*. Vemos un ejemplo de cómo implementar esta estructura.

for

El código en C:

```c
for (i = 0; i < 100; i++) {
  // Código a repetir 100 veces
}
```

Python no posee una estructura *for* como la mostrada anteriormente, y que es común a muchos lenguajes de programación. En su lugar, la estructura *for* de Python recorre uno por uno los elementos de una lista. Así, para conseguir en Python una estructura *for* similar a la anterior, se debe crear primero una lista, «`range(0,n)`», para luego recorrerla:

```python
for i in range(0, 100):
  # Código a repetir 100 veces
```

En ensamblador de ARM, la estructura *for* se puede implementar de la siguiente forma:

AB_for.s

```
        .text
        ldr  r0, =0     @ r0 es el índice i
        ldr  r1, =100   @ r1 mantiene la condición de salida
for:    cmp  r0, r1     @ Lo primero que se hace es evaluar la condición
        bge  finfor     @  de salida y salir del bucle si no se cumple
          @ Código a repetir 100 veces
          @ ...
        add r0, r0, #1  @ Se actualiza el contador
        b   for         @ Se vuelve al comienzo del bucle
finfor: @ Resto del programa
        wfi
```

Un bucle *for* se puede utilizar por ejemplo para recorrer los elementos de un vector. Un programa que sume todos los elementos de un vector quedaría como sigue en C:

```c
int V[5] = {1, 2, 3, 4, 5};
```

```
int VTam = 5;
int sum = 0;
for (i = 0; i < VTam; i++) {
  sum = sum + V[i];
}
```

En Python, de la siguiente forma:

```
V = [1, 2, 3, 4, 5]
sum = 0
for i in range(0, len(V)):
  sum = sum + V[i]
```

Y en ensamblador de ARM:

AB_for_vector.s

```
        .data
V:      .word 1, 2, 3, 4, 5
VTam:   .word 5
sum:    .space 4

        .text
        ldr  r0, =0     @ r0 es el índice i
        ldr  r1, =VTam
        ldr  r1, [r1]   @ r1 mantiene la condición de salida
        ldr  r2, =V     @ r2 tiene la dirección de comienzo de V
        ldr  r3, =0     @ r3 tiene el acumulador
for:    cmp  r0, r1     @ Lo primero que se hace es evaluar la condición
        bge  finfor     @  de salida y salir del bucle si no se cumple
          lsl r4, r0, #2   @ r4 <- i*4
          ldr r4, [r2, r4] @ r4 <- V[i]
          add r3, r3, r4   @ r3 <- sum + V[i]
        add r0, r0, #1  @ Se actualiza el contador
        b   for         @ Se vuelve al comienzo del bucle
finfor: ldr r4, =sum
        str r3, [r4]    @ Se guarda el acumulado en sum
        wfi
```

B.3.2. Estructura 'while'

En el caso de la estructura *for*, vista en el apartado anterior, se mantiene un índice que se actualiza en cada iteración de la estructura sin necesidad de que en los lenguajes de alto nivel aparezca dicha actualización en el bloque de código correspondiente. Sin embargo, la permanencia o salida se realiza evaluando una condición que, al margen de incluir el índice, podría ser cualquiera. Por eso, la estructura iterativa por excelencia, con toda la flexibilidad posible en cuanto a su implementación, es la estructura *while*. En esta estructura simplemente se evalúa

while

una condición de permanencia, sea cual sea, y se sale de las repeticiones en caso de que sea falsa. El resto del código no tiene ninguna restricción. A continuación se muestra dicha estructura en C, en Python y en ensamblador de ARM.

La estructura *while* en C es:

```c
while (a < b) {
  // Código a repetir
}
```

En Python:

```python
while a < b:
  # Código a repetir
```

Y en ensamblador de ARM:

AB_while.s

```
        .text
while:  cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición
        bge  finwhile   @  y si no se da, se sale
          @ Código a repetir
        b while         @ Se vuelve al comienzo del bucle
finwhile: @ Resto del programa
        wfi
```

Sabiendo que una de las formas de indicar la longitud de una cadena es la de añadir un byte a 0 al final de la cadena, una posible utilidad de un bucle *while* sería la de recorrer todos los elementos de una cadena para, por ejemplo, calcular su longitud. Puesto que el lenguaje C utiliza esta técnica para indicar el final de una cadena, un programa en C que haría lo anterior sería:

```c
char cad[] = "Esto es una cadena";
int long = 0;
while (cad[long] != 0) {
  long = long + 1;
}
```

En Python, una cadena es en realidad un objeto y la forma correcta de obtener su longitud es pasando dicha cadena como parámetro a la función «`len()`». Por ejemplo, «`len(cad)`» devolvería la longitud del objeto «`cad`». Sin embargo, es posible simular el código C anterior en Python sin más que añadir un byte a 0 al final de la cadena y utilizando un bucle *while* para obtener su longitud.

```python
cad = "Esto es una cadena\0"
long = 0;
```

```
3 while cad[long] != 0:
4   long = long + 1
```

Por su parte, la implementación en ensamblador de ARM sería:

AB_long_cad.s

```
1         .data
2 cad:    .asciz "Esto es una cadena"
3         .align 2
4 long:   .space 4
5
6         .text
7         ldr r0, =cad     @ r0 tiene la dir. de inicio de cad
8         ldr r1, =0       @ r1 tiene la longitud
9 while:  ldrb r2, [r0,r1] @ r2 <- cad[long]
10        cmp r2, #0
11        beq finwh         @ Sale si cad[long] == 0
12          add r1, r1, #1   @ long = long + 1
13        b while          @ Vuelve al comienzo del bucle
14 finwh: ldr r2, =long
15        str r1, [r2]     @ Almacena la longitud
16        wfi
```

Volviendo a la estructura *while*, si se observa con detalle la implementación anterior, o la más esquemática mostrada en «`0B_while.s`», se puede ver que el salto de salida vuelve a una nueva evaluación de la condición seguida de otro salto. Esta forma de proceder es ineficaz porque se usan dos saltos cuando con uno sería suficiente —y los saltos suelen afectar negativamente a la velocidad de ejecución de instrucciones—. Por eso, la implementación real de un bucle *while* se realiza en dos etapas:

- Primero se evalúa la condición que se tiene que cumplir para no ejecutar el bucle.
- En el caso de entrar en el bucle, se evalúa al final del bucle si este se debe volver a ejecutar. Conviene tener en cuenta que en este segundo salto condicional, la condición de salto es la de permanencia en el bucle y, por tanto, la misma que se puede observar en un lenguaje de alto nivel, no la contraria, como ocurre habitualmente.

El ejemplo «`0B_while.s`» implementado de esta segunda forma quedaría como se muestra a continuación. Como se puede ver, la etiqueta *while* ahora está situada al comienzo del código que se quiere repetir, no en la primera evaluación. Además, al final del bucle se decide si se vuelve al comienzo del bucle, en lugar de volver de forma incondicional al comienzo del bucle.

AB_while2.s

```
        .text
        cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición
        bge  finwhile   @  y si no se da, se sale
while:    @ Código a repetir
        cmp  r0, r1     @ Se vuelve a evaluar la condición y si se da,
        blt  while      @  se vuelve al comienzo del bucle
finwhile: @ Resto del programa
        wfi
```

Apéndice C

Firmware incluido en ARMSim

Índice

Se reproduce a continuación la descripción de las subrutinas proporcionadas por el firmware incorporado en el simulador ARMSim, así como su código fuente. La descripción de las subrutinas también puede consultarse desde la ayuda del simulador QtARMSim.

Las funciones de este firmware siguen el convenio de ARM para el paso de parámetros y devolución de resultados. Los valores pasados utilizan los registros del `r0` al `r3`, y la pila en caso de requerir más de 4 parámetros. Los valores se devuelven en `r0` y en `r1` si es necesario. Las funciones modifican solamente los registros de `r0` a `r3`, preservando el valor de los demás.

C.1. Funciones aritméticas

`divide(unsigned int dividendo, unsigned int divisor)`

Calcula la división de 32 bits sin signo dividendo/divisor, con el dividendo en `r0` y el divisor en `r1`. El cociente de la división se devuelve en `r0` y el resto en `r1`.

Si el divisor, `r1`, es 0 se señala el error devolviendo el valor `0xFFFF FFFF` tanto en el cociente, `r0`, como en el resto, `r1`, lo que es un resultado imposible para una división de 32 bits correcta.

`sdivide(int dividendo, int divisor)`

Calcula la división de 32 bits con signo dividendo/divisor, con el dividendo en `r0` y el divisor en `r1`. El cociente de la división se devuelve en `r0` y el resto en `r1`. El resto siempre tiene el signo del dividendo —excepto cuando se trate de una división exacta, en cuyo caso valdrá 0—.

Si el divisor, `r1`, es 0 se señala el error devolviendo el valor `0x7FFF FFFF` tanto en el cociente, `r0`, como en el resto, `r1`, lo que es un resultado imposible para una división de 32 bits correcta.

Si el dividendo, `r0`, es `0x8000 0000` y el divisor, `r1`, -1, el cociente devuelto es `0x8000 0000` dado que el rango de enteros positivos se desborda. Todos los demás casos devuelven el cociente y el resto esperados.

`sqrt(unsigned int val)`

Devuelve en `r0` la parte entera de la raíz cuadrada positiva del valor pasado en `r0`.

C.2. Funciones del LCD

`cls()`

Borra lo mostrado en el visualizador LCD y lo deja en blanco llenando todas sus posiciones con el carácter espacio. No recibe ni devuelve ningún valor.

`fill(char c)`

Llena todas las posiciones del visualizador LCD con el carácter *c* pasado en el byte menos significativo de `r0`. No devuelve ningún valor.

`printString(int col, int fil, char *str)`

Muestra en el visualizador LCD la cadena ASCIIZ cuya dirección se pasa en `r2`, comenzando en las coordenadas (col, fil), estando col en `r0` y fil en `r1`

Devuelve en `r0` el número de caracteres mostrados, es decir, la longitud de la cadena.

La función no verifica si las coordenadas (col, fil) están dentro o fuera del visualizador, ni si la cadena se muestra en una o más líneas, ni si se muestra completa o queda en parte fuera del visualizador.

`printInt(int col, int fil, int val)`

Muestra en el visualizador LCD la representación decimal del valor con signo que se pasa en `r2`, comenzando en las coordenadas (col, fil), estando col en `r0` y fil en `r1`.

Solo se muestran los caracteres significativos, precedidos de un signo —si se trata de un número negativo—. Devuelve en `r0` el número de caracteres mostrados.

La función no verifica si las coordenadas (col, fil) están dentro o fuera del visualizador, ni si el número se muestra en una o más líneas, ni si se muestra completo o queda en parte fuera del visualizador.

`printUInt(int col, int fil, unsigned int val)`

Muestra en el visualizador LCD la representación decimal del valor sin signo que se pasa en `r2`, comenzando en las coordenadas (col, fil), estando col en `r0` y fil en `r1`.

Solo se muestran los caracteres significativos. Devuelve en `r0` el número de caracteres mostrados.

La función no verifica si las coordenadas (col, fil) están dentro o fuera del visualizador, ni si el número se muestra en una o más líneas, ni si se muestra completo o queda en parte fuera del visualizador.

`printWord(int col, int fil, unsigned int val)`

Muestra en el visualizador LCD la representación hexadecimal de la palabra que se pasa en `r2`, comenzando en las coordenadas (col, fil), estando col en `r0` y fil en `r1`.

Se muestran los 8 caracteres hexadecimales. No se devuelve ningún valor.

La función no verifica si las coordenadas (col, fil) están dentro o fuera del visualizador, ni si el número se muestra en una o más líneas, ni si se muestra completo o queda en parte fuera del visualizador.

`printHalf(int col, int fil, unsigned int val)`

Muestra en el visualizador LCD la representación hexadecimal de la media palabra que se pasa en los dos bytes bajos de `r2`, comenzando en las coordenadas (col, fil), estando col en `r0` y fil en `r1`.

Se muestran los 4 caracteres hexadecimales. No se devuelve ningún valor.

La función no verifica si las coordenadas (col, fil) están dentro o fuera del visualizador, ni si el número se muestra en una o más líneas, ni si se muestra completo o queda en parte fuera del visualizador.

`printByte(int col, int fil, unsigned int val)`

Muestra en el visualizador LCD la representación hexadecimal del byte que se pasa en el byte bajo de `r2`, comenzando en las coordenadas (col, fil), estando col en `r0` y fil en `r1`.

Se muestran los 2 caracteres hexadecimales. No se devuelve ningún valor.

La función no verifica si las coordenadas (col, fil) están dentro o fuera del visualizador, ni si el número se muestra en una o más líneas, ni si se muestra completo o queda en parte fuera del visualizador.

C.3. Código fuente del firmware de ARMSim

firmware.s

```
1  Número  .equ    display, 0x20080000            @ Origen de la memoria del display
2          .equ    fil, 6                         @ Líneas del display
3          .equ    col, 40                        @ Caracteres por línea
4
5
6          .text
7
8  /*****************************************************/
9  /* sqrt(unsigned int val)                            */
10 /*****************************************************/
11 /* Devuelve la parte entera de la raíz cuadrada de un   */
12 /* entero sin signo. Sigue el algoritmo binario que la  */
13 /* calcula bit a bit. (Wikipedia, versión en inglés).   */
14 /* Entrada:                                          */
15 /* r0: valor                                         */
16 /* Salida:                                           */
17 /* r0: raíz entera                                   */
18
19         .balign 4
20
21 sqrt:
22         ldr r2, = 0x40000000    @ Base para los desplazamientos de bit
23         mov r1, #0              @ Resultado, se inicia a 0
24         cmp r0, r2              @ Mientras el radicando sea menor que
25         bcs sqalg               @ el bit desplazado,
26 buc_s:
27         lsr r2, r2, #2          @ desplazamos a la derecha de 2 en 2 bits
28         cmp r0, r2
29         bcc buc_s
30 sqalg:
31         tst r2, r2              @ Si el bit desplazado llega a 0 se acaba
32         beq fin
33 buc_s2:
34         add r3, r1, r2          @ Se compara el radicando con el resultado
35         cmp r0, r3              @ parcial más el bit
36         bcc else                @ Si es mayor se le resta tal suma
37         sub r0, r0, r3          @ y se actualiza el resultado como
38         lsr r1, r1, #1          @ como res = (res >>l 1) + bit
39         add r1, r1, r2
40         b   finsi
41 else:
42         lsr r1, r1, #1          @ Si no, se desplaza el resultado
43 finsi:
44         lsr r2, r2, #2          @ Desplazamos el bit hasta que llegue a 0
45         bne buc_s2
46 fin:
47         mov r0, r1              @ Se pone el resultado en r0 y se termina
48         mov pc, lr
49
50 /*****************************************************/
51 /* sdivide(int dividendo, int divisor)               */
52 /*****************************************************/
53 /* Realiza una division con signo.                   */
54 /* Devuelve el resto en r1 y el cociente en r0.      */
55 /* Si el divisor es 0 devuelve 0x7FFFFFFF en ambos.  */
56 /* Utiliza la división entera por lo que el resto tiene */
57 /* signo del divisor.                                */
58 /* Entrada:                                          */
59 /* r0: dividendo                                     */
60 /* r1: divisor                                       */
61 /* Salida:                                           */
62 /* r0: cociente                                      */
63 /* r1: resto                                         */
```

```
64
65          .balign 4
66
67  sdivide:
68          push    {r4-r5, lr}
69          cmp     r1, #0          @ Si el divisor es 0 damos error
70          bne     s_hazdiv
71  s_error:
72          sub     r1, r1, #1      @ para ello ponemos resto y cociente
73          lsr     r1, r1, #1      @ al mayor valor positivo 0x7FFFFFFF
74          mov     r0, r1          @ y salimos
75          bne     s_final
76  s_hazdiv:
77          mov     r4, #0          @ Se ponen a 0 las marcas de negativo
78          mov     r5, #0
79          tst     r1, r1          @ Si el divisor es negativo
80          bpl     s_dos           @ se marca en r5 y se le cambia el signo
81          mov     r5, #1
82          neg     r1, r1
83  s_dos:                          @ Si el dividendo es negativo
84          tst     r0, r0          @ se marca en r4 y se le cambia el signo
85          bpl     s_call
86          mov     r4, #1
87          neg     r0, r0
88  s_call:
89          bl      divide          @ Se dividen los valores positivos
90          cmp     r5, r4          @ Si los signos de dividendo y divisor son distintos
91          beq     s_resto
92          neg     r0, r0          @ se cambia el signo al cociente
93  s_resto:
94          cmp     r4, #1          @ Si el cociente es negativo
95          bne     s_final
96          neg     r1, r1          @ se cambia el signo al resto
97  s_final:
98          pop     {r4-r5, pc}     @ Y se vuelve
99
100
101 /*****************************************************/
102 /* divide(unsigned int dividendo, unsigned int divisor) */
103 /*****************************************************/
104 /* Realiza una division sin signo.                   */
105 /* Devuelve el resto en r1 y el cociente en r0.      */
106 /* Si el divisor es 0 devuelve 0xFFFFFFFF en ambos.  */
107 /* Entrada:                                          */
108 /* r0: dividendo                                     */
109 /* r1: divisor                                       */
110 /* Salida:                                           */
111 /* r0: cociente                                      */
112 /* r1: resto                                         */
113
114         .balign 4
115
116 divide:
117         mov     r2, #0          @ En r2 tenemos el contador de desplazamientos
118         orr     r1, r1          @ Vemos si el divisor es 0
119         bne     a_hazdiv        @ y en caso afirmativo
120         sub     r0, r2, #1      @ ponemos 0xFFFFFFFF (-1)
121         mov     r1, r0          @ en cociente y resto
122         b       a_final
123 a_hazdiv:                       @ Empezamos a dividir
124         mov     r3, #0          @ r3 es el cociente parcial
125         and     r1, r1          @ Si el msb del cociente es 1
126         bmi     a_itera         @ evitamos los desplazamientos
127 a_compara:
128         cmp     r0, r1          @ Comparamos el dividendo con el divisor
129         bcc     a_itera         @ desplazado, y seguimos desplazando hasta
130         add     r2, r2, #1      @ que sea menor. Obtenemos en r2 el número
131         lsl     r1, r1, #1      @ de bits del cociente
```

```
132         bpl     a_compara       @ Evitamos el desbordamiento
133 a_itera:
134         lsl     r3, r3, #1      @ Desplazamos el cociente
135         cmp     r0, r1          @ Vemos si cabe a 1
136         bcc     a_finresta      @ Si no cabe vamos a finresta
137         add     r3, r3, #1      @ si cabe sumamos 1 en el cociente
138         sub r0, r0, r1          @ y restamos
139 a_finresta:
140         lsr     r1, r1, #1      @ Desplazamos el dividendo
141         sub     r2, #1          @ Seguimos si quedan iteraciones
142         bpl     a_itera
143 a_findiv:
144         mov     r1, r0          @ El resto esta en r0
145         mov     r0, r3          @ y el cociente en r3
146 a_final:
147         mov     pc, lr
148
149 /*****************************************************/
150 /* printString(int col, int fil, char *cad)          */
151 /*****************************************************/
152 /* Imprime la cadena dada en el display a partir de la  */
153 /* coordenada dada. No realiza verificaciones.       */
154 /*                                                   */
155 /* Entrada:                                          */
156 /* r0: Columna (0...col - 1)                         */
157 /* r1: Fila (0...fil - 1)                            */
158 /* r2: Cadena ASCIIZ a imprimir                      */
159 /*                                                   */
160 /* Salida:                                           */
161 /* r0: Número de caracteres impresos (strlen)        */
162
163         .balign 4
164
165 printString:
166         ldr     r3, =col        @ Se calcula en r0 la dirección de inicio según las
167         mul     r1, r1, r3      @ coordenadas, multiplicando el número de fila
168         add     r1, r1, r0      @ por las colúmnas y sumando el de columna
169         ldr     r0, =display    @ Al resultado se suma la dirección base del display
170         add     r0, r0, r1
171         mov     r3, #0          @ Se pone a 0 el contador de caracteres
172 buc_1:
173         ldrb    r1, [r2]        @ Se lee un carácter y si es 0
174         cmp     r1, #0          @ se termina
175         beq     end_1
176         strb    r1, [r0]        @ Si no, se deja en memoria y se incrementan
177         add     r3, r3, #1      @ el contador de caracteres
178         add     r0, r0, #1      @ la dirección en el display
179         add     r2, r2, #1      @ y la dirección en la cadena en memoria
180         b       buc_1           @ y se vuelve al bucle
181 end_1:
182         mov     r0, r3          @ El número de caracteres se pone en r0
183         mov     pc, lr          @ y se termina
184
185
186 /*****************************************************/
187 /* printUInt(int col, int fil, unsigned int val)     */
188 /*****************************************************/
189 /* Imprime en el display a partir de las coordenadas    */
190 /* dadas el entero sin signo en decimal. No realiza     */
191 /* verificaciones.                                   */
192 /*                                                   */
193 /* Entrada:                                          */
194 /* r0: Columna (0...col - 1)                         */
195 /* r1: Fila (0...fil - 1)                            */
196 /* r2: Entero a imprimir                             */
197 /*                                                   */
198 /* Salida:                                           */
199 /* r0: Número de caracteres impresos                 */
```

```
200
201         .balign 4
202
203 printUInt:
204         push    {r5-r7, lr}     @ Se va a crear en la pila una cadena con el número
205         mov     r5, sp          @ en decimal, por eso ponemos r5 apuntando a
206         sub     r5, r5, #4      @ SP + 4 y reservamos espacio para 12 caracteres
207         sub     sp, #12
208         mov     r6, r0          @ Se guardan las coordenadas en r6 y r7
209         mov     r7, r1
210         mov     r0, r2          @ y se copia el número en r0
211         mov     r1, #0          @ Ponemos el final de cadena en r5. Al ir dividiendo entre
212         strb    r1, [r5]        @ 10 la cadena se va a crear de la cifra menos significativa
213 buc_2:                          @ hacia la más, es decir de arriba a abajo en memoria
214         sub     r5, r5, #1      @ Se decrementa r5 para el siguiente carácter
215         mov     r1, #10         @ y se divide el número entre 10
216         bl      divide
217         add     r1, #48         @ Se pasa el resto a carácter
218         strb    r1, [r5]        @ y se escribe en la cadena
219         cmp     r0, #0          @ Si el cociente es 0 se ha terminado
220         bne     buc_2           @ en otro caso se continúa
221         mov     r0, r6          @ Recuperamos las coordenadas
222         mov     r1, r7
223         mov     r2, r5          @ y ponemos en r2 la dirección de la cadena
224         bl      printString     @ Se imprime y pone en r0 su longitud
225         add sp, #12             @ Se ajusta la pila
226         pop     {r5-r7, pc}     @ y se termina
227
228
229 /*****************************************************/
230 /* printInt(int col, int fil, int val)               */
231 /*****************************************************/
232 /* Imprime en el display a partir de las coordenadas    */
233 /* dadas el entero con signo en decimal. No realiza     */
234 /* verificaciones.                                      */
235 /*                                                      */
236 /* Entrada:                                             */
237 /* r0: Columna (0...col - 1)                            */
238 /* r1: Fila (0...fil - 1)                               */
239 /* r2: Entero a imprimir                                */
240 /*                                                      */
241 /* Salida:                                              */
242 /* r0: Número de caracteres impresos                    */
243
244         .balign 4
245
246 printInt:
247         push {r4-r7, lr}        @ Se va a crear en la pila una cadena con el número
248         mov     r5, sp          @ en decimal, por eso ponemos r5 apuntando a
249         sub     r5, r5, #4      @ SP + 4 y reservamos espacio para 12 caracteres
250         sub     sp, #12
251         mov     r6, r0          @ Se guardan las coordenadas en r6 y r7
252         mov     r7, r1
253         mov     r0, r2          @ y se copia el número en r0
254         mov     r4, #0          @ r4 guardará 1 si es negativo
255         and     r2, r2          @ Lo verificamos mirando su msb
256         bpl     mas_3
257         neg     r0, r2          @ si es negativo le cambiamos el signo
258         add     r4, r4, #1      @ y lo marcamos en r4
259 mas_3:
260         mov     r1, #0          @ Ponemos el final de cadena en r5. Al ir dividiendo entre
261         strb    r1, [r5]        @ 10 la cadena se va a crear de la cifra menos significativa
262 buc_3:                          @ hacia la más, es decir de arriba a abajo en memoria
263         sub     r5, r5, #1      @ Se decrementa r5 para el siguiente carácter
264         mov     r1, #10         @ y se divide el número entre 10
265         bl      divide
266         add     r1, #48         @ Se pasa el resto a carácter
267         strb    r1, [r5]        @ y se escribe en la cadena
```

```
268         cmp     r0, #0          @ Si el cociente es 0 se ha terminado
269         bne     buc_3           @ en otro caso se continúa
270         cmp     r4, #0          @ Si el número era negativo
271         beq     positivo
272         mov r4, #'-             @ Se añade el signo menos al principio
273         sub     r5, r5, #1
274         strb    r4, [r5]
275 positivo:
276         mov     r0, r6          @ Recuperamos las coordenadas
277         mov     r1, r7
278         mov     r2, r5          @ y ponemos en r2 la dirección de la cadena
279         bl      printString     @ Se imprime y pone en r0 su longitud
280         add sp, #12             @ Se ajusta la pila
281         pop     {r4-r7, pc}     @ y se termina
282
283
284 /*****************************************************/
285 /* printWord(int col, int fil, unsigned int val)        */
286 /*****************************************************/
287 /* Imprime en el display a partir de las coordenadas    */
288 /* dadas el entero en hexadecimal. Siempre imprime 8    */
289 /* caracteres hexadecimales. No realiza verificaciones  */
290 /*                                                      */
291 /* Entrada:                                             */
292 /* r0: Columna (0...col - 1)                            */
293 /* r1: Fila (0...fil - 1)                               */
294 /* r2: Entero a imprimir                                */
295 /*                                                      */
296 /* Salida:                                              */
297 /* ---                                                  */
298         .balign 4
299
300 printWord:
301         mov     r3, #8          @ r3 guarda los caracteres a imprimir
302         b       printHex        @ se imprimen en printHex
303
304
305 /*****************************************************/
306 /* printHalf(int col, int fil, unsigned int val)        */
307 /*****************************************************/
308 /* Imprime en el display a partir de las coordenadas    */
309 /* dadas el half en hexadecimal. Siempre imprime 4      */
310 /* caracteres hexadecimales. No realiza verificaciones. */
311 /*                                                      */
312 /* Entrada:                                             */
313 /* r0: Columna (0...col - 1)                            */
314 /* r1: Fila (0...fil - 1)                               */
315 /* r2: Entero a imprimir                                */
316 /*                                                      */
317 /* Salida:                                              */
318 /* ---                                                  */
319
320         .balign 4
321
322 printHalf:
323         mov     r3, #4          @ r3 guarda los caracteres a imprimir
324         b       printHex        @ se imprimen en printHex
325
326 /*****************************************************/
327 /* printByte(int col, int fil, unsigned int val)        */
328 /*****************************************************/
329 /* Imprime en el display a partir de las coordenadas    */
330 /* dadas el byte en hexadecimal.Siempre imprime 2       */
331 /* caracteres hexadecimales. No realiza verificaciones. */
332 /*                                                      */
333 /* Entrada:                                             */
334 /* r0: Columna (0...col - 1)                            */
335 /* r1: Fila (0...fil - 1)                               */
```

```
/* r2: Entero a imprimir                                */
/*                                                      */
/* Salida:                                              */
/* ---                                                  */

        .balign 4

printByte:
        mov     r3, #2          @ r3 guarda los caracteres a imprimir

printHex:                       @ Comienzo del código de impresion
        push    {r4-r5, lr}
        ldr     r4, =col        @ Se calcula la dirección del carácter
        mul     r1, r1, r4      @ menos significativo: se multiplica la fila por
        add     r1, r1, r0      @ el número de columnas, se suma la columna,
        ldr     r0, =display    @ la dirección del display
        add     r0, r0, r1
        add     r0, r0, r3      @ y el número de caracteres
        mov     r5, #0x0f       @ En r5 se guarda la máscara para dejar 4 bits (una cifra hex)
buc_4:
        mov     r1, r2          @ Se copia en r1 el número actual
        and     r1, r1, r5      @ se deja el nibble (4 bits) bajo
        add     r1, #'0         @ se pasa a carácter
        cmp     r1, #'9 + 1     @ Si es menor o igual que '9 ya está
        bcc     nosuma
        add     r1, #'A - '9 - 1 @ si no, lo pasamos a la letra correspondiente
nosuma:
        sub     r0, r0, #1      @ Decrementamos la posición del display
        lsr     r2, r2, #4      @ Desplazamos 4 bits el número (dividimos entre 16)
        strb    r1, [r0]        @ dejamos el carácter en el display
        sub     r3, r3, #1      @ y decrementamos la cuenta de caracteres a escribir
        bne     buc_4           @ Si no es 0, seguimos
end_4:
        pop     {r4-r5, pc}     @ Termina la función

/******************************************************/
/* fill(char val)                                     */
/******************************************************/
/* Rellena el display con el valor val.               */
/*                                                    */
/* Entrada:                                           */
/* r0: Valor a rellenar (8 lsb)                       */
/*                                                    */
/* Salida:                                            */
/* ---                                                */

        .balign 4

fill:
        mov     r1, #0xFF       @ Dejamos en r2 el carácter de r0
        and     r0, r1          @ dejando el resto del registro a 0
        mov     r2, r0          @ con el and anterior
        lsl     r0, r0, #8      @ Dejamos espacio para otro carácter igual
        orr     r2, r0          @ que añadimos con or
        lsl     r0, r0, #8      @ Repetimos tres veces para dejar el
        orr     r2, r0          @ carácter original en los 4 bytes
        lsl     r0, r0, #8      @ de r2
        orr     r2, r0
        b       limpia          @ saltamos a escribir los caracteres


/******************************************************/
/* cls()                                              */
/******************************************************/
/* Limpia el display                                  */
/*                                                    */
/* Entrada:                                           */
/* ---                                                */
```

```
/*                                                  */
/* Salida:                                          */
/* ---                                              */

        .balign 4

cls:
        ldr     r2, =0x20202020 @ Un entero con 4 caracteres espacio (blancos)
limpia:
        ldr     r0, =display    @ r0 apunta a la dirección del display
        ldr     r1, =(fil*col)/4 @ en r1 el número de caracteres entre 4
buc_5:
        str     r2, [r0]        @ Se escribe entero a entero
        add     r0, r0, #4      @ incrementando en 4 la dirección
        sub     r1, r1, #1      @ decrementando en 1 el número de enteros que faltan
        bne     buc_5           @ Terminamos al llegar a 0
        mov     pc, lr          @ y volvemos

        .end
```

Apéndice D

Sistemas de numeración

Hay 10 tipos de personas:
las que comprenden el binario y las que no.

Índice

El Peñagolosa es el pico más emblemático de la provincia de Castellón; es también el más alto, elevándose 1 813 metros sobre el nivel del mar. La altura del Peñagolosa es una característica intrínseca de la montaña. La expresión de su altura en la frase anterior depende del convenio empleado en las unidades de medida, en este caso, el sistema métrico decimal —podríamos haber escrito, con igual corrección, 5 948 pies— y de la forma de representar la cantidad de metros: y en numeración romana, que no es usual para medidas, pero sí para capítulos y fechas, el Peñagolosa mide MDCCCXIII metros.

En binario, la altura del Peñagolosa es 11100010101 metros.

D.1. Introducción

La cantidad es una propiedad inherente de hechos y objetos en el universo. La altura de una montaña, el número de reses de un rebaño, las puestas de sol entre dos determinados sucesos, son cuantificables objetivamente con independencia de cómo se exprese luego tal cantidad. El hombre, al tener la necesidad de comunicarse y de comunicar las cantidades, ha ido creando los sistemas de numeración como formas humanas de expresarlas.

Así por ejemplo la cultura romana utilizó una forma de numeración aditiva, en que a cada símbolo se le asocia un valor y la mezcla ordenada de símbolos sirve para formar la cantidad total. Para calcular la altura del Peñagolosa nos basta con saber que el símbolo M tiene el valor 1000, D vale 500, C equivale a 100 y X e I a 10 y 1, respectivamente. Por supuesto, todos los valores anteriores se han expresado en decimal, que es la forma más común hoy en día de representación de cantidades. Las otras reglas del sistema romano indican que como mucho se pueden agrupar 3 símbolos iguales y que los símbolos situados a la derecha de uno de nivel inmediatamente superior suman, mientras que a la izquierda restan. De esta manera la cantidad MDCCCXIII suma 1000 de una M, 500 de una D, 300 de tres C, 10 de una X y 3 de tres I, con lo que llegamos a los 1 813 metros del Peñagolosa.

Los sistemas aditivos son limitados, pues tienden a aumentar casi linealmente el número de símbolos a medida que aumentan las cantidades, por lo que solo se usan, y por razones históricas, en algunos casos en cierto modo nostálgicos. La cultura árabe trajo a Europa algo que a su vez había heredado del Lejano Oriente: el cero y los sistemas de numeración posicionales.

Todos entendemos de forma natural, porque es parte de nuestra cultura, el valor de 1 813 sin necesidad de realizar ningún cálculo. Sin embargo, ¿cómo se obtiene y de qué depende matemáticamente ese valor? Estrictamente hablando, el valor de la cantidad escrita se obtiene de la

siguiente expresión:

$$1\,813 = 1 \times 10^3 + 8 \times 10^2 + 1 \times 10^1 + 3 \times 10^0$$

Expresar los números de esta forma es muy útil para trabajar con sus valores con independencia de la representación. De esta forma, se pueden demostrar los criterios para saber si un número en base diez es divisible entre 5, entre 9, entre 3... Es un ejercicio de matemática recreativa que se deja propuesto como pasatiempo.

Pero volvamos a los fundamentos de los sistemas de numeración posicionales. Si estudiamos la expresión que calcula el valor de 1 813 nos daremos cuenta de que el 10 es un número especial pues aparece en todos los términos, elevado a un exponente que depende, efectivamente, de la posición. Y dicha posición es la que ocupa, de derecha a izquierda, la cifra —o símbolo— en la expresión de la cantidad numérica y que determina el peso de cada potencia de 10 en el total.

Si ignoramos nuestro lastre cultural y dejamos por un momento la base diez, podemos definir un sistema de numeración posicional genérico. Para ello nos hace falta una base B y un conjunto de B símbolos —tantos como el valor de la base— para representar los valores desde 0 hasta $B - 1$. De esta manera, si representamos, en un ejemplo, los símbolos mediante letras mayúsculas, podemos expresar una igualdad genérica similar a la anterior:

$$HCXDT = H \times B^4 + C \times B^3 + X \times B^2 + D \times B^1 + T \times B^0$$

En el caso del sistema decimal, la base es, obviamente, 10 y los diez símbolos, las cifras por todos conocidas, del 0 al 9. Es interesante destacar que, dado que cierta potencia de la base puede no tener que sumarse para obtener la cantidad expresada, por ejemplo 1 000 en 60 344, es necesario que exista un símbolo con valor nulo, el 0, en el conjunto. Lo que conceptualmente es complejo —¿para qué contar cero manzanas?— e hizo que los sistemas posicionales aparecieran mucho más tarde que los aditivos en la historia de la humanidad.

D.2. El sistema binario

El ENIAC, considerado el primer computador electrónico de la historia, almacenaba los datos en base diez pese a utilizar tecnología digital. Afortunadamente pronto se constató que la forma más sencilla de diseñar circuitos de computador para realizar operaciones era utilizando la base natural de la electrónica digital binaria: la base dos. Efectivamente, en electrónica digital se trabaja con dos valores o estados lógicos que de forma natural se pueden asociar a dos símbolos, el 1 y el 0. De esta

forma el sistema binario se ha hecho con un nicho de representación tan importante en el universo de la informática como el decimal en el de la especie humana.

Los fundamentos del sistema binario son lógicamente los mismos que se han explicado antes para el caso general. El hecho de trabajar con solo dos símbolos aporta propiedades interesantes, algunas de las cuales las veremos más adelante, al tiempo que hace que el sistema sea incómodo de representar e interpretar —fuera de los circuitos de un ordenador—. Al haber solo dos símbolos, estos se repiten mucho, haciendo las cantidades en binario difíciles de usar. Por otra parte, siendo la menor base posible —es fácil darse cuenta de que la base uno no permite un sistema posicional como el explicado— los números representados tienen gran cantidad de cifras y su representación ocupa mucho espacio. El valor 1 813, como hemos visto antes, necesita 11 cifras para ser representado en binario.

Pese a estos problemas, el sistema binario sigue siendo el utilizado por los circuitos digitales y es tan sencillo en la teoría como cualquier otro. El número 205 en decimal se expresaría en binario según 11001101 y su valor se puede verificar calculando la expresión conocida y operando en decimal:

$$11001101 = 1\times 2^7 + 1\times 2^6 + 0\times 2^5 + 0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$$

Lo bueno de trabajar en base dos es que podemos eliminar rápidamente los 0 y los 1 de la expresión, quedándonos únicamente con la suma de las potencias de 2 que contribuyen al resultado, es decir:

$$11001101 = 2^7 + 2^6 + 2^3 + 2^2 + 2^0 = 128 + 64 + 8 + 4 + 1 = 205$$

Expresión en la que esperamos se nos disculpe el hecho de comenzar en binario y terminar en decimal, sin añadir nada a la notación que indique la base en la que escribimos un número. Hecho en el que radica la sorpresa y el humor de la cita con que se abre este documento. Más adelante incidiremos en que, en informática, se debe expresar adecuadamente tanto los números como las bases en que se representan.

D.3. El hexadecimal como expresión más cómoda del binario

Los procesadores de la penúltima generación —ahora estamos de lleno en los 64 bits— son arquitecturas de 32 bits. Si tuviéramos que expresar en binario el mapa de memoria de uno de estos computadores, agotaríamos el tóner de nuestra impresora en listas de 32 unos y ceros que, además, sería muy difícil incluir luego en un programa sin cometer

errores. Afortunadamente, desde hace tiempo, se pensó en utilizar el sistema hexadecimal para llevar una notación relacionada con el binario al ámbito de lo razonable —en términos de comprensión humana y tinta—.

En teoría, el sistema hexadecimal no es más que un sistema de numeración posicional con base 16. Dado que los números solo nos aportan 10 símbolos, para la representación hexadecimal se completó el conjunto de 16 con las letras entre la A y la F —indistintamente mayúsculas o minúsculas— que representan los valores entre 10 y 15 según su orden alfabético. El número 205 que hemos tratado antes se escribe en hexadecimal como CD, y el 1 813 como 715 —de nuevo vemos que es un lío poner números sin indicar su base de representación—. Lo que hace que el sistema hexadecimal sea adecuado como representación más cómoda en informática es que 16 es una potencia de 2, en particular 2^4. Veamos matemáticamente cómo nuestro número 205 —en decimal, 11001101 en binario— se convierte de forma sencilla en CD —en hexadecimal—.

Según teníamos más arriba,

$$11001101 = 1\times2^7+1\times2^6+0\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$$

Si agrupamos de cuatro en cuatro las potencias de 2, obtenemos:

$$\begin{aligned} 11001101 \;=\; & (1\times2^3+1\times2^2+0\times2^1+0\times2^0)\times(2^4)^1 + \\ & (1\times2^3+1\times2^2+0\times2^1+1\times2^0)\times(2^4)^0 \end{aligned}$$

Como 2^4 es 16, hemos expresado nuestro número en función de potencias de 16. Además, la suma dentro de cada paréntesis es un valor entre 0 y 15, es decir, se corresponde con un símbolo en base 16. De esta manera, podemos continuar desarrollando —y mezclando libremente las bases, como hasta ahora— para tener:

$$11001101 = 12\times16^1+13\times16^0 = C\times16^1+D\times16^0 = CD$$

Como se acaba de ver el hexadecimal y el binario se relacionan de forma inmediata dado que cada cuatro cifras —unos y ceros— en binario se agrupan en una —entre 0 y F— en hexadecimal. Por supuesto, para hacer las agrupaciones de 4 unos y ceros, hay que comenzar desde la derecha del número binario y completar con 0 a la izquierda —los ceros a la izquierda y algunas veces los unos, como veremos luego, no modifican el valor—. De esta manera, una cifra hexadecimal se convierte siempre en el mismo conjunto de 1 y 0 en binario, ocupe el lugar que ocupe en la expresión de la cantidad. Esto es debido, como se ha dicho, a que 16 es 2^4. Es fácil darse cuenta de que las cifras de un valor expresado en decimal no se transforman en binario tan fácilmente.

El Cuadro D.1 muestra una tabla con las cifras en hexadecimal, su valor en decimal y su representación en binario.

Cuadro D.1: Valores en decimal y binario de los números hexadecimales del 0 al F

Hex.	Dec.	Binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

D.4. Cambiando entre distintas bases

Ya sabemos algo de los sistemas de numeración posicionales, y hemos visto tres sistemas de numeración: binario, hexadecimal y decimal, cuyo uso es común en informática. Antes de ver cómo cambiar de sistema o, resumiendo, de base un número expresado en una de ellas, vamos por fin a establecer un criterio que nos permita saber la base en que se expresa una cantidad escrita. Hay varias formas de hacerlo, en este texto utilizaremos el convenio del ensamblador GNU. Veámoslo:

- Una cantidad escrita sin más, de la forma habitual y sin ceros no significativos a la izquierda, será siempre un valor en decimal. De tal forma, 11 es el número once, 1 813, la altura en metros del Peñagolosa y CD, un soporte óptico muy popular para almacenar música y otra información digital, dado que en decimal, la C y la D no se usan.

- Los números en hexadecimal se preceden de 0x, de modo que 0x11 es diecisiete, 0xCD el 205 decimal del ejemplo anterior, y 0x715 la altura de nuestra montaña preferida —al menos en este texto—.

- Los números en binario comienzan por 0b, de forma que 0b11 es tres, el valor de altura que usamos en este texto, 0b11100010101 y 205 se expresa como 0b11001101.

- El sistema octal, aunque no es tan usado hoy en día como el hexadecimal, también se emplea en algunos casos —códigos de caracteres y listas de permisos de acceso a ficheros, por ejemplo—. En él se emplea la base 8 y los números entre el 0 y el 7. Su relación con el binario es tan sencilla como la del hexadecimal. Un número representado en octal comienza simplemente por 0, de ahí que se deba evitar el cero a la izquierda al expresar números decimales. Se deja como ejercicio expresar en octal las cantidades ya famosas en este texto.

Ahora que ya podemos expresar completamente una cantidad y su base, vamos a estudiar brevemente cómo cambiar la expresión de una cantidad, de una base a otra. En realidad, los cambios entre binario y hexadecimal o entre binario y octal, no requieren mayor explicación. Basta con agrupar los unos y ceros de cuatro en cuatro —de binario a hexadecimal— o de tres en tres —de binario a octal— comenzando por la derecha y asociar a cada agrupación su valor —entre 0 y F o entre 0 y 7—. Simétricamente, para pasar de hexadecimal o de octal a binario, nos basta con cambiar cada símbolo por su representación en binario —cuatro o tres cifras, según la base de partida— y el conjunto de unos y ceros resultante será la expresión en binario del número. Para cambiar entre hexadecimal y octal, y viceversa, usaremos el binario como representación intermedia —aunque realmente dudamos de que sea necesario realizar tal cambio de base nunca—. Veamos algunos ejemplos:

$$\begin{aligned} \text{0xB73} &= \text{0b1011 0111 0011} \\ &= \text{0b101 101 110 011} = \text{05563} \end{aligned}$$

$$\begin{aligned} \text{0x9318} &= \text{0b1001 0011 0001 1000} \\ &= \text{0b001 001 001 100 011 000} = \text{0111430} \end{aligned}$$

Para una mejor comprensión, en los ejemplos anteriores los valores binarios se han agrupado, respectivamente, en bloques de 4 y 3 cifras, añadiendo ceros a la izquierda según fuera necesario.

Después de ver estos ejemplos sencillos, observaremos que el caso más difícil se da en los cambios entre base diez y cualquiera de las otras bases. Caso que por otra parte suele ser el más frecuente. No aporta en realidad más que mayor complejidad de cálculo, siendo conceptualmente tan sencillo como los otros. Efectivamente, para convertir de cualquier base a base diez, basta con aplicar la expresión de una cantidad según

el sistema posicional, con todos los valores en decimal, y realizar las operaciones. De hecho, más arriba ya habíamos utilizado las expresiones:

$$0b11001101 = 1\times2^7+1\times2^6+0\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$$

$$0b11001101 = 2^7 + 2^6 + 2^3 + 2^2 + 2^0 = 128 + 64 + 8 + 4 + 1 = 205$$

Con la única diferencia de que ahora expresamos correctamente el número binario. Un ejemplo en hexadecimal sería:

$$0xFE0 = 15 \times 16^2 + 14 \times 16^1 + 0 \times 16^0 = 4\,064$$

De estas expresiones se deduce que la forma de pasar de un número en decimal a cualquier otra base —realizando siempre cálculos en base diez, que es lo que sabemos hacer— debe hacerse de la forma inversa, mediante divisiones. De esta manera, para pasar un número de decimal a hexadecimal realizaremos divisiones sucesivas entre 16 hasta que el cociente sea menor que 16. Este último cociente —que marcará la mayor potencia de 16 del número— nos dará la cifra más a la izquierda del resultado —expresada con el símbolo correspondiente— y luego tomaremos todos los restos hasta el primero, que nos indicará la cifra menos significativa. Veámoslo: 4 046 entre 16 da resto 0 y cociente 254; 254 entre 16 da resto 14 y cociente 15. Así las cifras a tomar son, por orden, 15, 14 y 0, es decir 0xFE0.

Si en lugar de dividir entre 16 lo hacemos entre 2 u 8, obtendremos la representación en binario u octal, respectivamente.

D.5. El signo de los números binarios

Como hemos dicho al principio, el sistema binario es útil e interesante por ser el sistema de representación natural en los circuitos digitales. ¿Qué ocurre, sin embargo, cuando queremos representar tanto números positivos como negativos? Lo que nosotros expresamos sencillamente poniendo un signo − (menos) delante de nuestra representación numérica en decimal debe encontrar una forma sencilla y eficaz de tratarse en binario. Podría pensarse que dedicar un valor binario —un bit— para indicar el signo es una buena solución, de hecho así lo hace el sistema de representación signo-magnitud, pero consideremos en este caso lo difícil que resulta realizar una suma o una resta: además de efectuar la operación en sí, hemos de comparar los valores en caso de tener distinto signo para ver cuál asignar al resultado. Afortunadamente el complemento a 2 (que denotaremos normalmente como Ca2) nos ofrece una forma sencilla y natural de representar números positivos y negativos, como veremos a continuación.

En general, para cualquier base, se define el complemento a la base de un número como la resta de dicho número a una potencia de la base mayor que dicho número. Para definir de forma completa el complemento a la base hemos de fijar un número de cifras, siendo entonces el complemento a la base de un número como la resta de la base elevada al total de cifras menos ese número. Si el número de cifras es n y usamos base 10, el complemento a la base de B es $10^n - B$.

Veamos cómo funciona el complemento a la base para hacer restas —primera aproximación a los números negativos—. Para restar $A - B$, partimos de la identidad siguiente:

$$A + (10^n - B) = A - B + 10^n$$

Como vemos, hemos sustituido la resta de B por una suma con el complemento a la base de B, y, al hacerlo, nos sobra un término, 10^n, que luego analizaremos.

A continuación se muestran un par de ejemplos usando base diez y 3 cifras. Aunque antes conviene indicar que el complemento a la base se calcula de forma sencilla en base diez cambiando cada cifra por su diferencia con 9 —los ceros a la izquierda se convierten en nueves a la izquierda— y sumando uno al resultado. Efectivamente: $10^n - A = 999\ldots9 - A + 1$. Veamos ahora los ejemplos:

$$487-85 = \begin{cases} 402, & \text{restando} \\ 487 + (914 + 1) = 1402, & \text{sumando complemento a la base} \end{cases}$$

$$134-252 = \begin{cases} -118, & \text{restando} \\ 134 + (747 + 1) = 882, & \text{sumando complemento a la base} \end{cases}$$

En el primero de los ejemplos, si se limita el resultado obtenido cuando se ha sumado el complemento a la base a 3 cifras, entonces se puede ver como con ambos métodos obtienen 402 como resultado. En el segundo ejemplo, si se obtiene el complemento a la base de -118, $881 + 1 = 882$, se puede ver como este coincide con el resultado que se ha obtenido al sumar el complemento a la base.

Asombrosamente —si lo vemos así de sopetón, matemáticamente es una verdad evidente— si hacemos restas sumando el complemento a la base, obtenemos: el resultado correcto con un 10^n de más —ese uno que sobra como cifra más significativa en el primer ejemplo—, si el resultado es positivo; y ¡el complemento a la base del resultado si este es negativo! Estas propiedades, que ocurren siempre al usar el complemento a la base, nos permiten definir el sistema de representación en complemento a la base como aquel que no modifica los números positivos y utiliza el complemento a la base del valor absoluto del número para representar los negativos.

Lo único que nos queda por ver es dónde se pone el límite entre números positivos y negativos. Como n nos limita el número de cifras y un uno que se sale de ese rango nos indica que el resultado de una resta —o una suma entre números de distinto signo— es positivo, hemos de garantizar que la suma de dos números positivos cualesquiera no supere esas n cifras, con lo que el mayor número positivo será $499\ldots9$, con n cifras, y, por tanto, el mayor negativo será $-500\ldots0$, con n cifras.

Además, otra característica interesante es que el complemento a la base del menor número, el negativo de mayor valor absoluto, es el tal valor absoluto. Si seguimos con tres cifras tenemos:

$$-500 = 499 + 1 = 500$$

Así pues un sistema de numeración en complemento a diez, complemento a la base en base 10, cumple las siguientes propiedades:

- Se define para un número de cifras fijo.
- Los números positivos se representan de forma natural y van desde 0 hasta $\frac{10^n}{2} - 1$.
- Los números negativos se representan mediante el complemento a diez de su valor absoluto, y van desde -1 hasta $-\frac{10^n}{2}$.

Estas propiedades en realidad son comunes a todos los sistemas de representación en complemento a la base.

Otra propiedad interesante, que se deduce de las anteriores, es que un sistema en complemento a diez con n cifras, se puede extender a uno con más cifras de forma sencilla: añadiendo ceros a la izquierda de los números positivos y añadiendo nueves a la izquierda de los negativos, hasta completar el límite de cifras del nuevo sistema. El valor de la nueva cifra así expresada se mantiene. Veámoslo recuperando el ejemplo con tres cifras que hemos visto antes:

$$487 - 85 = \begin{cases} 402, & \text{restando} \\ 487 + 915 = 1402, & \text{sumando complemento a 10} \end{cases}$$

Si ahora pasamos a cinco cifras:

$$37487 - 85 = \begin{cases} 37402, & \text{restando} \\ 37487 + 99915 = 137402, & \text{sumando complemento a 10} \end{cases}$$

Que como vemos, da el resultado esperado extendiendo 915 a 5 cifras con dos nueves más.

Si reflejamos estas propiedades en el sistema de numeración binario, tenemos el complemento a dos o Ca2. Según las reglas operativas que

hemos visto en el caso anterior, para realizar la operación Ca2 de un número basta con cambiar sus ceros por unos y viceversa y sumar uno al resultado. Veamos cómo expresar en binario, con 8 bits, el número decimal -78:

$$\begin{aligned} 78 &= 0\text{b}01001110; \\ -78 &= 0\text{b}10110001 + 1 = 0\text{b}10110010 \end{aligned}$$

Por si dudamos, nos basta con sumarlos:

$$0\text{b}01001110 + 0\text{b}10110010 = 0\text{b}100000000$$

Y ese uno que sobra en la cifra novena, ya sabemos que indica que el resultado es positivo —o cero—. Resumiendo, el sistema de representación en complemento a dos:

- Se define para un número de cifras fijo.
- Los números positivos se representan de forma natural y van desde 0 hasta $2^{n-1} - 1$ (aquí la división entre 2 se representa restando uno al exponente).
- Los números negativos se representan mediante el complemento a dos de su valor absoluto, y van desde -1 hasta -2^{n-1}

Si en el complemento a diez teníamos que una cifra más significativa —aquella más a la izquierda— igual o superior a 5 servía para identificar un número negativo, en complemento a dos esta cifra es el uno. Así, todo número cuya cifra más significativa sea un 1 es negativo, mientras que si esta cifra es 0 es positivo.

Si representamos el número en hexadecimal, podemos mirar el Cuadro D.1 para ver que todo número que comience por un símbolo superior o igual a 8 es negativo, mientras que entre 0 y 7, será positivo. Eso sí, utilizando todas las cifras del rango de representación. Ahora las cifras a la izquierda sí son importantes, aunque sean ceros.

En el caso del Ca2, para extender el rango de representación a un número mayor de cifras —bits— bastará con añadir ceros a la izquierda hasta el nuevo rango si el número es positivo —si su cifra más significativa en binario es 0— o unos si es negativo —si su cifra más significativa es 1—.

Los sistemas de representación en complemento a dos se utilizan en rangos de 8, 16 y 32 bits —o cifras binarias—. Los dos mayores sobre todo son muy incómodos de representar en binario, por lo que se utiliza habitualmente la notación en hexadecimal. En este caso hemos de recordar que un número cuya cifra más significativa es menor que 8 se extiende con 0, añadiendo esta cifra tal cual al número resultante en

hexadecimal, mientras que si es 8 o mayor se extiende con 1, añadiendo el carácter F —recordemos que 0xF equivale a 0b1111—. Veamos para terminar algunos ejemplos de extensión de 8 a 16 y 32 bits:

$$\begin{aligned} -78 = 0\text{b}10110010 &= 0\text{xB2 (8 bits)} \\ &= 0\text{xFFB2 (16 bits)} \\ &= 0\text{xFFFFFFB2 (32 bits)} \end{aligned}$$

$$\begin{aligned} 105 = 0\text{b}01101001 &= 0\text{x69 (8 bits)} \\ &= 0\text{x0069 (16 bits)} \\ &= 0\text{x00000069 (32 bits)} \end{aligned}$$

Apéndice E

Guía rápida del ensamblador Thumb de ARM

Las siguientes dos páginas proporcionan una guía rápida del ensamblador Thumb de ARM.

Conjunto de instrucciones ARM Thumb (1/2)

Operación		Ensamblador	Acción	Actualiza	Notas
Mover	Inmediato 8 bits	`mov Rd, #Inm8`	Rd ← Inm8	N Z	Rango Inm8: 0–255.
	Lo a Lo	`mov Rd, Rm`	Rd ← Rm	N Z	
	Hi a Lo, Lo a Hi, Hi a Hi	`mov Rd, Rm`	Rd ← Rm		
Sumar	Inmediato 3 bits	`add Rd, Rn, #Inm3`	Rd ← Rn + Inm3	N Z C V	Rango Inm3: 0–7.
	Inmediato 8 bits	`add Rd, Rd, #Inm8`	Rd ← Rd + Inm8	N Z C V	Rango Inm8: 0–255.
	Lo a Lo	`add Rd, Rn, Rm`	Rd ← Rn + Rm	N Z C V	
	Hi a Lo, Lo a Hi, Hi a Hi	`add Rd, Rd, Rm`	Rd ← Rd + Rm		
	Valor a SP	`add SP, SP, #Inm`	SP ← SP + Inm		Rango Inm: 0–508 (alineado a palabra).
	Crear dirección desde SP	`add Rd, SP, #Inm`	Rd ← SP + Inm		Rango Inm: 0–1020 (alineado a palabra).
Restar	Inmediato 3 bits	`sub Rd, Rn, #Inm3`	Rd ← Rn − Inm3	N Z C V	Rango Inm3: 0–7.
	Inmediato 8 bits	`sub Rd, Rd, #Inm8`	Rd ← Rd − Inm8	N Z C V	Rango Inm8: 0–255.
	Lo a Lo	`sub Rd, Rn, Rm`	Rd ← Rn − Rm	N Z C V	
	Valor de SP	`sub SP, SP, #Inm`	SP ← SP − Inm		Rango Inm: 0–508 (alineado a palabra).
	Negar	`neg Rd, Rn`	Rd ← −Rn	N Z C V	
Multiplicar	Multiplica	`mul Rd, Rm, Rd`	Rd ← Rm ∗ Rd	N Z	
Comparar	Compara	`cmp Rn, Rm`	Act. flags según Rn − Rm	N Z C V	Cualquier registro a cualquier registro.
	Compara con negado	`cmn Rn, Rm`	Act. flags según Rn + Rm	N Z C V	
	Inmediato	`cmp Rn, #Inm8`	Act. flags según Rn − Inm8	N Z C V	Rango Inm8: 0–255.
Lógicas	AND	`and Rd, Rd, Rm`	Rd ← Rd AND Rm	N Z	
	AND NOT (borrar bits)	`bic Rd, Rd, Rm`	Rd ← Rd AND NOT Rm	N Z	
	OR	`orr Rd, Rd, Rm`	Rd ← Rd OR Rm	N Z	
	XOR (or exclusiva)	`eor Rd, Rd, Rm`	Rd ← Rd XOR Rm	N Z	
	NOT	`mvn Rd, Rm`	Rd ← NOT Rm	N Z	
	Comprueba bits	`tst Rn, Rm`	Act. flags según Rn AND Rm	N Z	
Desplazar	Lógico a la izquierda	`lsl Rd, Rm, #Shift`	Rd ← Rm << Shift	N Z C	Rango Shift: 0–31
		`lsl Rd, Rd, Rs`	Rd ← Rd << $[Rs]_{7:0}$	N Z C	
	Lógico a la derecha	`lsr Rd, Rm, #Shift`	Rd ← Rm $>>_l$ Shift	N Z C	Rango Shift: 0–31
		`lsr Rd, Rd, Rs`	Rd ← Rd $>>_l$ $[Rs]_{7:0}$	N Z C	
	Aritmético a la derecha	`asr Rd, Rm, #Shift`	Rd ← Rm $>>_a$ Shift	N Z C	Rango Shift: 0–31
		`asr Rd, Rd, Rs`	Rd ← Rd $>>_a$ $[Rs]_{7:0}$	N Z C	
	Rotación a la derecha	`ror Rd, Rd, Rs`	Rd ← Rd ROR $[Rs]_{7:0}$	N Z C	

Conjunto de instrucciones ARM Thumb (2/2)

Operación		Ensamblador	Acción	Notas
Cargar	Con desp. inm., palabra	`ldr Rd, [Rn, #Inm]`	Rd ← [Rn + Inm]	Rango Inm: 0–124, múltiplos de 4.
	media palabra	`ldrh Rd, [Rn, #Inm]`	Rd ← ZeroExtend($[Rn + Inm]_{15:0}$)	Bits 31:16 a 0. Rango Inm: 0–62, pares.
	byte	`ldrb Rd, [Rn, #Inm]`	Rd ← ZeroExtend($[Rn + Inm]_{7:0}$)	Bits 31:8 a 0. Rango Inm: 0–31.
	Con desp. en registro, palabra	`ldr Rd, [Rn, Rm]`	Rd ← [Rn + Rm]	
	media palabra	`ldrh Rd, [Rn, Rm]`	Rd ← ZeroExtend($[Rn + Rm]_{15:0}$)	Bits 31:16 a 0.
	media palabra con signo	`ldrsh Rd, [Rn, Rm]`	Rd ← SignExtend($[Rn + Rm]_{15:0}$)	Bits 31:16 igual al bit 15.
	byte	`ldrb Rd, [Rn, Rm]`	Rd ← ZeroExtend($[Rn + Rm]_{7:0}$)	Bits 31:8 a 0.
	byte con signo	`ldrsb Rd, [Rn, Rm]`	Rd ← SignExtend($[Rn + Rm]_{7:0}$)	Bits 31:8 igual al bit 7.
	Relativo al PC	`ldr Rd, [PC, #Inm]`	Rd ← [PC + Inm]	Rango Inm: 0–1020, múltiplos de 4.
	Relativo al SP	`ldr Rd, [SP, #Inm]`	Rd ← [SP + Inm]	Rango Inm: 0–1020, múltiplos de 4.
Almacenar	Con desp. inm., palabra	`str Rd, [Rn, #Inm]`	[Rn + Inm] ← Rd	Rango Inm: 0–124, múltiplos de 4.
	media palabra	`strh Rd, [Rn, #Inm]`	$[Rn + Inm]_{15:0}$ ← $Rd_{15:0}$	$Rd_{31:16}$ se ignora. Rango Inm: 0–62, pares.
	byte	`strb Rd, [Rn, #Inm]`	$[Rn + Inm]_{7:0}$ ← $Rd_{7:0}$	$Rd_{31:8}$ se ignora. Rango Inm: 0–31.
	Con desp. en registro, palabra	`str Rd, [Rn, Rm]`	[Rn +Rm] ← Rd	
	media palabra	`strh Rd, [Rn, Rm]`	$[Rn + Rm]_{15:0}$ ← $Rd_{15:0}$	$Rd_{31:16}$ se ignora.
	byte	`strb Rd, [Rn, Rm]`	$[Rn + Rm]_{7:0}$ ← $Rd_{7:0}$	$Rd_{31:8}$ se ignora.
	Relativo al SP	`str Rd, [SP, #Inm]`	[SP + Inm] ← Rd	Rango Inm: 0–1020, múltiplos de 4.
Apilar	Apilar	`push <loreglist>`	Apila registros en la pila	
	Apilar y enlazar	`push <loreglist+LR>`	Apila LR y registros en la pila	
Desapilar	Desapilar	`pop <loreglist>`	Desapila registros de la pila	
	Desapilar y retorno	`pop <loreglist+PC>`	Desapila registros y salta a la dirección cargada en el PC	
Saltar	Salto condicional	`b{cond} <label>`	Si {cond}, PC ← label (rango salto: −252 a +258 bytes de la instrucción actual).	
	Salto incondicional	`b <label>`	PC ← label (rango salto: ±2 KiB de la instrucción actual).	
	Salto largo y enlaza	`bl <label>`	LR ← dirección de la siguiente instrucción, PC ← label (Instrucción de 32 bits. Rango salto: ±4 MiB de la instrucción actual).	
Extender	Con signo, media a palabra	`sxth Rd, Rm`	$Rd_{31:0}$ ← SignExtend($Rm_{15:0}$)	
	Con signo, byte a palabra	`sxtb Rd, Rm`	$Rd_{31:0}$ ← SignExtend($Rm_{7:0}$)	
	Sin signo, media a palabra	`uxth Rd, Rm`	$Rd_{31:0}$ ← ZeroExtend($Rm_{15:0}$)	
	Sin signo, byte a palabra	`uxtb Rd, Rm`	$Rd_{31:0}$ ← ZeroExtend($Rm_{7:0}$)	

{cond}	EQ	Igual	NE	Distinto	MI	Negativo	PL	Positivo	VS	Desbordamiento
	HI	Mayor sin signo	CS	Mayor o igual sin signo	CC	Menor sin signo	LS	Menor o igual sin signo	VC	No desbordamiento
	GT	Mayor que	GE	Mayor o igual	LT	Menor que	LE	Menor o igual que		

Índice de figuras

Índice de cuadros

Índice alfabético

Bibliografía

[1] ARM Limited: *ARM7TDMI data sheet*, 1995. http://pdf1.alldatasheet.es/datasheet-pdf/view/88658/ETC/ARM7TDMI.html.

[2] ARM Limited: *Cortex-M3 Devices. Generic User Guide*, 2010. http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf.

[3] Atmel Corporation: *ATmega128: 8-bit Atmel microcontroller with 128KBytes in-system programmable flash*, 2011. http://www.atmel.com/Images/doc2467.pdf.

[4] Atmel Corporation: *Atmel SAM3X-SAM3A Datasheet*, 2015. http://www.atmel.com/ru/ru/Images/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf.

[5] Barrachina Mir, Sergio, Maribel Castillo Catalán, José Manuel Claver Iborra y Juan Carlos Fernández Fernández: *Prácticas de introducción a la arquitectura de computadores con el simulador SPIM*. Pearson Educación, 2013.

[6] Barrachina Mir, Sergio, Maribel Castillo Catalán, Germán Fabregat Llueca, Juan Carlos Fernández Fernández, Germán León Navarro, José Vicente Martí Avilés, Rafael Mayo Gual y Raúl Montoliu Colás: *Prácticas de introducción a la arquitectura de computadores con Qt ARMSim y Arduino*, 2014. http://lorca.act.uji.es/libro/practARM/.

[7] Barrachina Mir, Sergio, Germán Fabregat Llueca, Juan Carlos Fernández Fernández y Germán León Navarro: *Utilizando ARMSim y QtARMSim para la docencia de arquitectura de computadores*. ReVisión, 8(3), 2015.

[8] Barrachina Mir, Sergio, Germán Fabregat Llueca y José Vicente Martí Avilés: *Utilizando Arduino DUE en la docencia de la entrada/salida*. En *Actas de las XXI Jornadas de la Enseñanza Universitaria de la Informática (JENUI)*, páginas 58–65. Universitat Oberta La Salle, 2015.

[9] Barrachina Mir, Sergio, Germán León Navarro y José Vicente Martí Avilés: *Conceptos elementales de computadores*, 2014. http://lorca.act.uji.es/libro/conceptos_elementales_de_computadores.pdf.

[10] Clements, Alan: *Selecting a processor for teaching computer architecture*. Microprocessors and Microsystems, 23(5):281–290, 1999, ISSN 0141-9331.

[11] Clements, Alan: *The undergraduate curriculum in computer architecture*. IEEE Micro, 20(3):13–22, 2000, ISSN 0272-1732.

[12] Clements, Alan: *Arms for the poor: Selecting a processor for teaching computer architecture*. En *Proceedings of the 2010 IEEE Frontiers in Education Conference (FIE)*, páginas T3E–1. IEEE, 2010.

[13] Clements, Alan: *Computer Organization and Architecture: Themes and Variations. International Edition.* Cengage Learning, 2014.

[14] Harris, Sarah y David Harris: *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2015.

[15] IEEE/ACM Joint Task Force on Computing Curricula. Computer Engineering.: *Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*. Informe técnico, IEEE Computer Society Press and ACM Press, Diciembre 2004.

[16] O'Sullivan, Dan y Tom Igoe: *Physical computing: sensing and controlling the physical world with computers*. Course Technology Press, 2004.

[17] Patterson, David A y John L Hennessy: *Estructura y diseño de computadores: la interfaz software/hardware*. Reverté, 2011.

[18] Shiva, Sajjan G: *Computer Organization, Design, and Architecture*. CRC Press, 2013.

[19] Texas Instruments Incorporated: *Cortex-M3 Instruction Set. Technical User's Manual*, 2010. http://users.ece.utexas.edu/~valvano/EE345M/CortexM3InstructionSet.pdf.