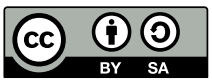
Introducción a la arquitectura de computadores con QtARMSim y Arduino

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, ¡bienvenida!

Índice general

Prólogo

Históricamente, el contenido de los cursos de Arquitectura de Computadores ha seguido el frenético ritmo marcado primero por los avances tecnológicos y arquitectónicos en el diseño de grandes computadores, y a partir de los 80, por la evolución en el diseño de los microprocesadores. Así pues, la docencia en Arquitectura de Computadores ha pasado por seis grandes eras: *mainframes*, *minicomputadores*, primeros microprocesadores, microprocesadores, RISC y post RISC [Cle00].

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 fuimos optando 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 dos motivos. En primer lugar, la arquitectura ARM presenta muchas características que la distinguen de otras arquitecturas contemporáneas, a la vez que al estar basada en RISC, es relativamente sencilla [Cle99]. En segundo lugar, el hecho de que ARM sea una arquitectura actual y ampliamente difundida, especialmente en dispositivos móviles, *smartphones* y *tablets*, es un factor especialmente motivador [Cle10]. 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 mil millones de procesadores ARM; más del 75 % de la población mundial utiliza productos con procesadores ARM [HH15].

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 [PH11], 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 [BMCCCIFF13], 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* [OI04]. 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 comporta como teóricamente debería. Es más, puesto que mucha de la problemática de la entrada/salida está directamente relacionada con la

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

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.

En base a 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 [BMFLFFLN15], para aquellas prácticas no relacionadas con la entrada/salida. Por otro lado, y tras valorar varias alternativas, optamos por la tarjeta Arduino Due [BMFLMA15] para las prácticas de entrada/salida.

La versión anterior de este libro, *Prácticas de introducción a la arquitectura de computadores con QtARMSim y Arduino* [BMCCFL$^+$14], 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 puede utilizarse 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 se pueda abarcar 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* [IEE04]:

- 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 sitio web se puede consultar y obtener material adicional relacionado con este libro. Entre otros, el entorno de desarrollo de Arduino modificado para la ejecución de programas en ensamblador, las guías para su instalación 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.

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 40 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, etc. 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 1949, 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.

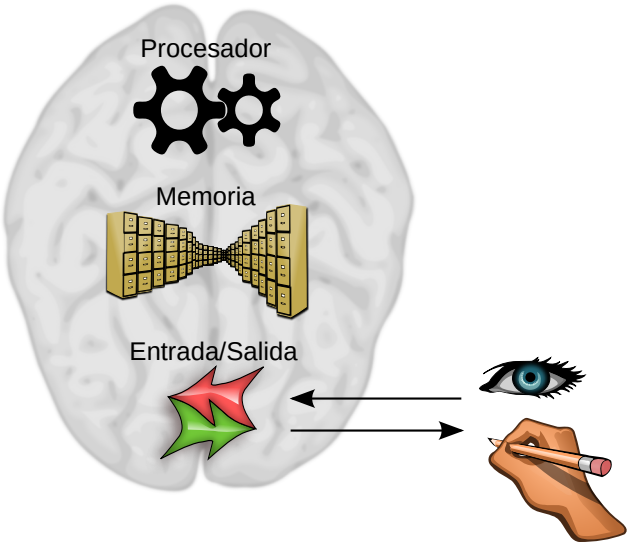


Figura 1.1: Componentes de un computador

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 todas 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 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**. Más adelante 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 el funcionamiento de un ordenador diciendo que el procesador ejecuta un programa, o secuencia de instrucciones, almacenado en memoria para realizar, guiado por aquél, 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 y 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 funcione un ordenador, 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] (de *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 calculo masivo, como por ejemplo, el supercomputador MareNostrum[2]. 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 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.

[1] El conjunto de instrucciones también recibe los nombres de **juego de instrucciones** y **repertorio de instrucciones**.

[2] http://www.bsc.es/marenostrum-support-services

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, un **microcontrolador** es 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 cualquier tarea de un electrodoméstico. Los fabricantes de microcontroladores suelen ofrecer una amplia gama de modelos con distintas 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 microcontrolador SAM3X8E, utilizado en las tarjetas Arduino DUE y que incorpora: un procesador ARM Cortex M3 de 32 bits, memoria 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 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 continuación 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 li-

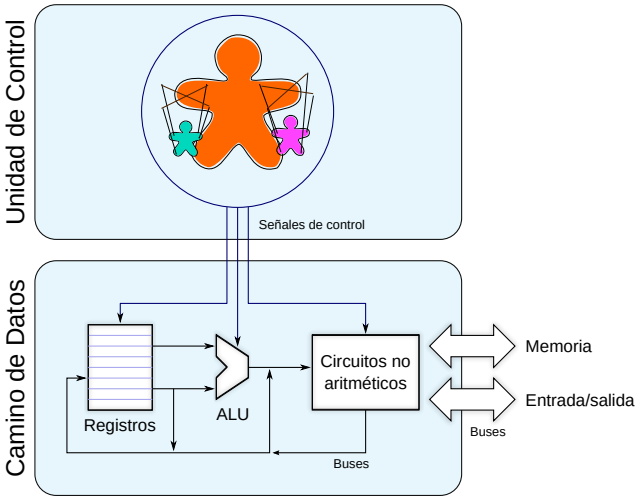


Figura 1.2: Componentes de un procesador

mitación 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 asignados 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 generalde 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 éstas, 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 verán 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[3] 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 nombrar en la literatura como **búsqueda de la 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

[3] 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.

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 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 llamado operando destino.

Una vez completadas las fases anteriores, se ha completado la ejecución de una instrucción y el procesador vuelve a empezar por la primera de ellas. Como el PC se ha incrementado para contener la dirección de la siguiente instrucción a ejecutar, el procesador repetirá los mismos pasos, pero esta vez para la siguiente instrucción, por tanto, continuará con la ejecución de una nueva instrucción. Así hasta el infinito o 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 con ordenaciones diferentes, sobre todo en lo que respecta a la ejecución simultáneamente 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 otros textos más avanzados se puede encontrar todo lo referente a la ejecución simultánea, la ejecución fuera de orden, etcétera.

1.2.3. Tipos de instrucciones

Las instrucciones que puede ejecutar cualquier procesador se pueden clasificar en un conjunto reducido de tipos. El número y la forma de

las instrucciones dentro de cada tipo marca las diferencias entre las distintas arquitecturas de procesador. Se describen a continuación 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 éste por tanto el operando destino de dichas instrucciones, se conseguiría alterar el flujo de ejecución de un programa para que éste no fuera el estrictamente secuencial. Estas son las 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 iterativas —*for*, *while*, etc.— o condicionales —*if*, *else*, etc.—.

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 código del sistema operativo y 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 los dos primeras fases. Analizando 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[4], pudiendo haber instrucciones de la 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 llama 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

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

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[5], si bien aquéllas 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 identificado 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 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.

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

Estudiando 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 pro-

[5]En realidad, el lenguaje ensamblador suele proporcionar más instrucciones que las estrictamente soportadas por la arquitectura. Estas instrucciones adicionales, llamadas **pseudo-instrucciones**, 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

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

pia instrucción[6], II) en registros del procesador y III) en memoria[7]. 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 dividen 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

[6]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

[7]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.

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 esta 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 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.

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-

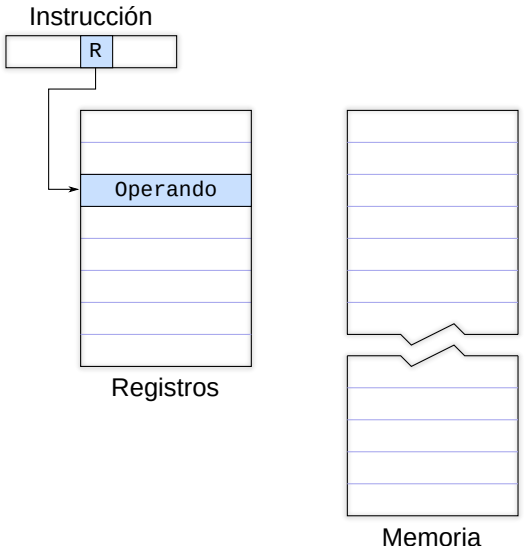


Figura 1.3: Modo de direccionamiento directo a registro. En este modo, la dirección de efectiva del operando es uno de los registros de la arquitectura. En la instrucción se codifica el número de registro en el que se encuentra el operando

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—.

El modo **indirecto con registro** (véase la Figura 1.5) permite refe-

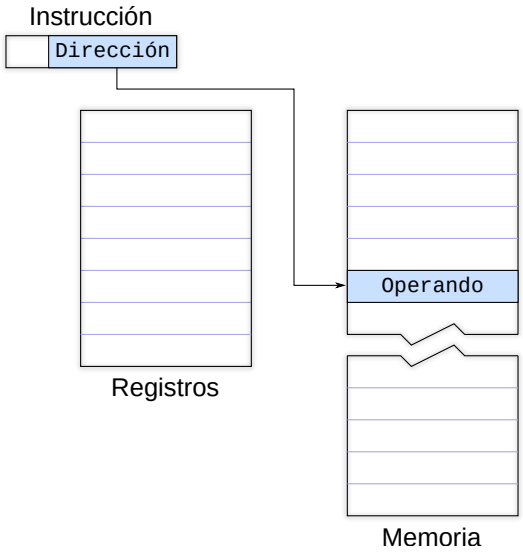


Figura 1.4: Modo de direccionamiento directo a memoria. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica la dirección de memoria en la que se encuentra el operando

rirse 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.

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 aquél 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 operando 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.

Otra variación del modo indirecto con registro es el modo **indirec-**

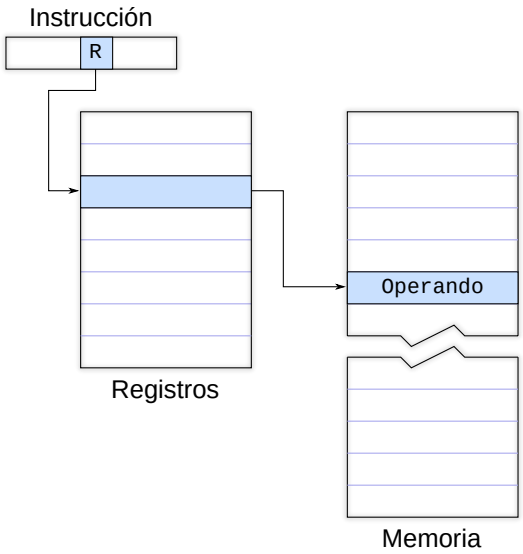


Figura 1.5: Modo de direccionamiento indirecto con registro. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica un número de registro, el contenido del cual indica la dirección de memoria en la que se encuentra el operando

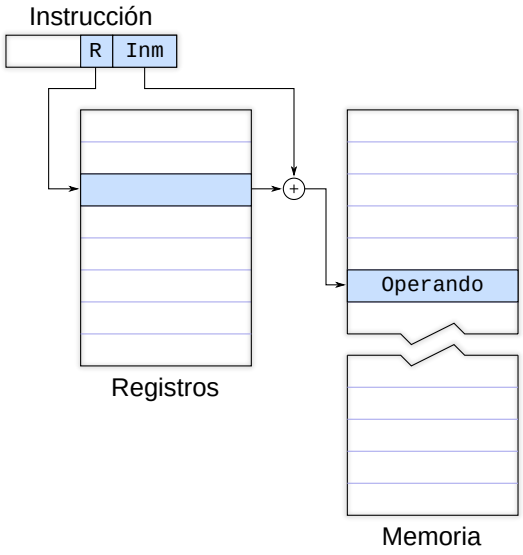


Figura 1.6: Modo de direccionamiento indirecto con desplazamiento. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica el número de un registro y un dato inmediato, la suma del contenido de dicho registro y el dato inmediato proporciona la dirección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

to 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[8] 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 y 1.2 utiliza 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. Se tienen también 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 utiliza en el contexto de los ordenadores, hace referencia a su modelo de funcionamiento. Más precisamente,

[8] 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.

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 éste.

Siguiendo la definición, centrada en el programador, que se ha dado, la arquitectura de un procesador se llama también **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; 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 sobreescribir 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 le 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, se utilizan los llamados lenguajes de alto nivel. Las razones de este cambio son históricas, ya que 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. A medida que fueron apareciendo más sistemas, comenzaron a desarrollarse los lenguajes de programación de alto nivel. Estos lenguajes permitían desarrollar programas que eran independientes del hardware en el que se fueran a ejecutar, lo que permitía que un mismo programa pudiera acabar ejecutándose en multitud de plataformas. De hecho, la tendencia en el desarrollo de lenguajes de programación de alto nivel continuó de tal manera que los procesadores, en particular sus conjuntos de instrucciones, se diseñan desde hace tiempo para adaptarse a las estructuras utilizadas 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, 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 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, en 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 aquél. 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 esa 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 por ello, se consideran lógicamente agrupadas en un único bus.

Profundizando y extendiendo 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 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[9], 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 el procesador con el resto del sistema, es decir, con la memoria y con los dispositivos de entrada/salida.

Hemos comentado que el procesador genera todas las señales eléctricas que sincronizan el funcionamiento del ordenador. De esta forma, 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 del procesador, que se conoce como FSB —*Front Side Bus*—. Esto es así porque 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[10].

Guiado por la ejecución de instrucciones, el procesador solo debe indicar al exterior, además de la dirección a la que dirige el acceso, si se trata de una lectura o una escritura y, posiblemente, el tamaño en bytes de los datos a los que quiere acceder. En algunos sistemas también será necesario indicar si el acceso es a una dirección de memoria o del subsistema de entrada/salida. Como vemos, el bus del procesador necesita tres tipos de información para realizar un acceso: la dirección, generada y puesta en el bus por el procesador; 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 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 diferencian tres tipos de líneas: direcciones, datos y control. Las primeras 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

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

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

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.

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 los conceptos.

Retomando los conceptos básicos, la memoria principal es el dispositivo que almacena las instrucciones y los datos de los programas en ejecución, con los que trabaja 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 hace que las direcciones que aparecen en el bus sean siempre múltiplos del número máximo de bytes que se pueden leer en paralelo. Así, si se quiere leer el número máximo de bytes en un solo acceso, debe hacerse en una dirección que sea múltiplo de este número. Si no se hace así, sería necesario efectuar una transacción para la primera parte del dato y otra para el resto. Este concepto se llama **alineamiento de datos** y se define enunciando que los datos alineados en memoria deben comenzar en una dirección múltiplo de su tamaño en bytes. Muchas arquitecturas solo permiten accesos a datos alineados en memoria; otras no generan un error en caso de datos desalineados, pero los accesos en este caso son necesariamente más lentos.

Cuando se va a almacenar 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

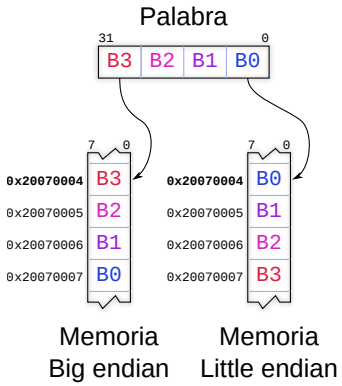


Figura 1.7: Los bytes de una palabra almacenada en la dirección de memoria `0x2007 0004` se dispondrán de una forma u otra dependiendo de si se sigue la organización *big endian* o la *little endian*

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 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 tiene que ver con la velocidad en la ejecución de las instrucciones, cuando se hace de forma solapada —es decir, que una instrucción comienza sus fases de ejecución antes de que termine la anterior—. Con una memoria

de instrucciones separada de la de datos, y por supuesto con buses de acceso también separados, una instrucción puede estar accediendo a la memoria de datos mientras la siguiente está siendo adquirida 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 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 están totalmente separadas. Es más, la tecnología de la memoria de datos es RAM, mientras que la de instrucciones es Flash. De esta manera, los programas se llevan a la memoria de instrucciones mediante un proceso externo al procesador, que normalmente no debe modificar el código. Aún así, en el caso en que el procesador tuviera que modificar el código, este es capaz de modificar bloques de la memoria de programa, tratando la memoria 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 que están siendo accedidos por el procesador con mayor frecuencia. Cuando el procesador debe acceder a un dato o instrucción que no se encuentra en la caché, copia de la memoria principal no solo lo que necesita, sino toda la información cercana, que previsiblemente 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 del procesador. 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, estos son llevados a 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 implementado la arquitectura Harvard. Los demás, más lentos, suelen ser unificados y compartidos por 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 estos ordenadores se puede instalar un sistema operativo con 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 introduce 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. De hecho, uno de los motivos por el que la arquitectura ARM ha acaparado el mercado de los dispositivos empotrados ha sido justamente por proporcionar el juego de instrucciones Thumb. Si se utiliza dicho juego de instrucciones, es posible reducir prácticamente a la mitad el tamaño de los programas, lo que permite diseñar dispositivos con menores requisitos de memoria, con lo que se disminuye su coste de fabricación, a la vez que se mejora su rendimiento al reducir los accesos a memoria.

Para generar código máquina a partir del ensamblador de ARM, ya

sea con el juego de instrucciones de 32 bits o con el Thumb, se puede optar por dos convenios para escribir el código en ensamblador: 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, etc.— es totalmente diferente en ambos.

Así, para programar en ensamblador para ARM es necesario tener en cuenta 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.

Por su parte, QtARMSim es una interfaz gráfica para 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`».

El resto del capítulo está 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. 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 tener claras 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

[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.

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).

Notación RTL

La notación RTL –del inglés *Register Transfer Language*— permite describir las operaciones llevadas a cabo por las instrucciones máquina de forma genérica. Lo que permite evitar la sintaxis propia de una arquitectura determinada. A continuación se describen los principales aspectos de la notación RTL utilizada en este libro.
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 contenido del registro `r4` es 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 sobreescribir con la suma del contenido del registro `r4` más el número 1, se utilizará la siguiente expresión: «[`0x2000 004`] ← `r4` + 1».

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`.

Teniendo 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 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 utilizar.

No obstante lo anterior, podemos considerar que los lenguajes ensambladores de las diferentes arquitecturas son más bien dialectos, no 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 el desarrollo de la computación consistió en el desarrollo de programas capaces de leer un lenguaje más cercano a nosotros y traducirlo a una secuencia de instrucciones máquina que el procesador fuera capaz de interpretar y ejecutar. Uno de estos, 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 dichos 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 microprocesador.

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 prácticamente obligatorio 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, este 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 «#» tan solo puede estar precedido de espacios. Así que para evitar problemas, es mejor utilizar siempre «@» para escribir comentarios de una línea. En el caso de querer escribir un comentario que ocupe varias líneas, es posible utilizar los delimitadores «/*» y «*/» para marcar dónde empieza y acaba el comentario, respectivamente.

Pseudo-instrucciones. Extienden el conjunto de instrucciones disponibles para el programador. Las pseudo-instrucciones 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 ensamblador para facilitar la programación en lenguaje ensamblador. Por tanto, es el programa ensamblador el que se encarga de traducir automáticamente cada pseudo-instrucción por aquella instrucción máquina o secuencia de instrucciones máquina que realicen la operación asociada.

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.

Etiquetas. Se utilizan para poder referenciar a la dirección de memoria del elemento definido en la línea en la que se encuentran. Para

declarar una etiqueta, ésta debe aparecer al comienzo de una línea, 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 en la que se haga referencia a dicha etiqueta, sustituirá la etiqueta por un valor numérico, que puede ser directamente la dirección de memoria de dicha etiqueta o un desplazamiento relativo a la dirección de memoria de la instrucción actual.

Directivas. Sirven para informar al ensamblador sobre cómo 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 suele tener la siguiente forma[3]:

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

El siguiente ejemplo de programa en ensamblador está formado por tres líneas. Cada línea de dicho programa 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 referenciada por la instrucción que hay en la tercera línea. Cuando se ensamble dicho programa, el ensamblador traducirá la instrucción «**`bne`** `Bucle`» por la instrucción máquina «**`bne`** `pc, #-8`». Es decir, sustituirá, sin entrar por el momento en más detalles, la etiqueta «`Bucle`» por un número, el «`-8`».

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

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
```

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

```
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, e 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**». Dicha 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 `0x0000 1000`.

«**.text**»

La penúltima de las líneas del código anterior contiene la instrucción «**wfi**». Dicha instrucción se usa para indicar al simulador QtARMSim que debe concluir la ejecución del programa en curso. Su uso es específico del simulador QtARMSim. Cuando se programe para otro entorno,

«**wfi**»

habrá que averiguar cuál es la forma adecuada de indicar el final de la ejecución en dicho entorno.

La última de las líneas del código anterior presenta la directiva «`.end`», que sirve para señalar el final del módulo que se quiere ensamblar. Por regla general, no es necesario utilizarla. En la práctica, tan solo tiene sentido hacerlo en el caso de que se quiera escribir algo a continuación de dicha línea de tal manera que ese texto sea ignorado por el ensamblador.

«`.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 una interfaz gráfica para el 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, etc.

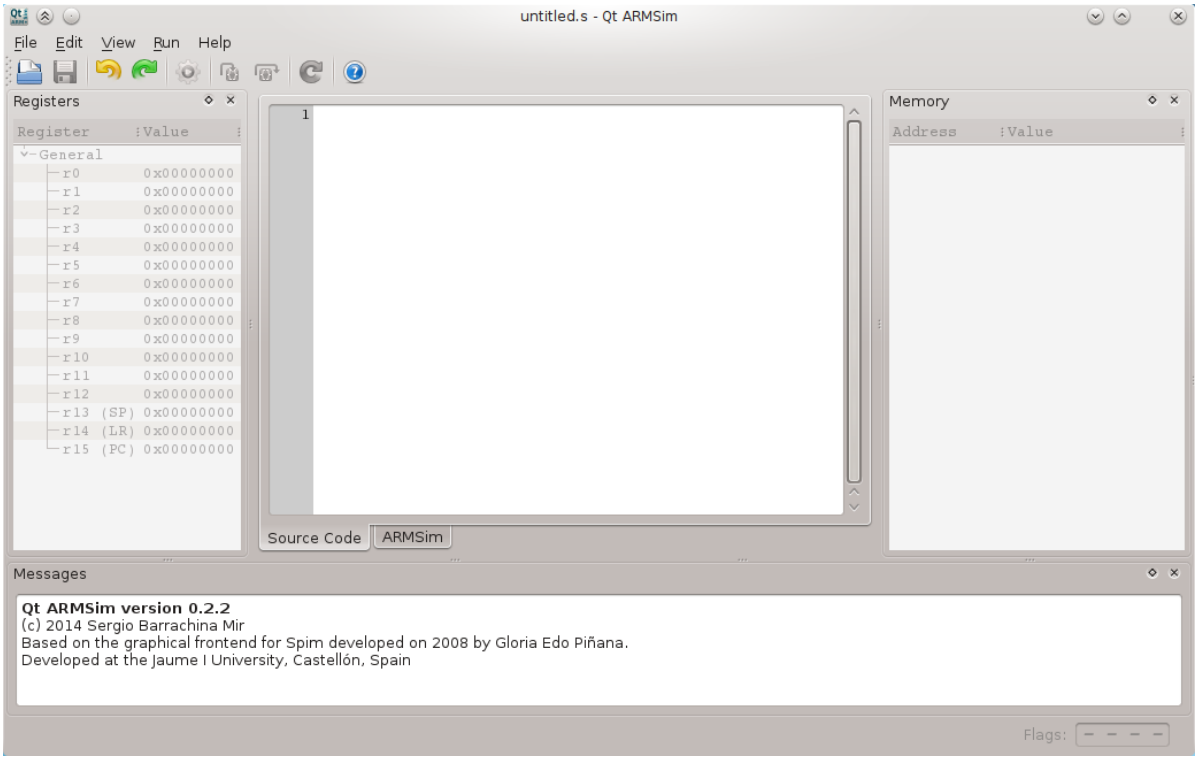


Figura 2.1: Ventana principal de QtARMSim

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 la entrada «`Preferences...`» dentro del menú «`Edit`». En dicho cuadro de diálogo 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

¿Mostrar el cuadro de diálogo de preferencias?

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.

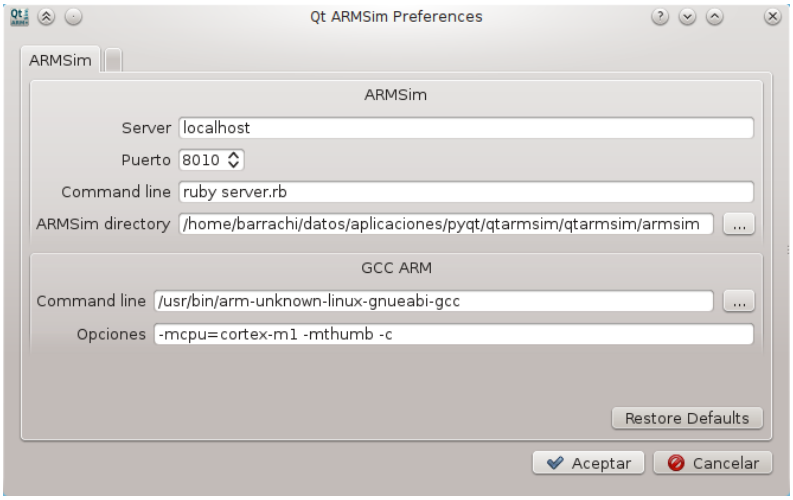


Figura 2.2: Cuadro de diálogo de preferencias de QtARMSim

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 a disco.

2.2.3. Modo de simulación

Una vez se ha escrito y guardado en disco un programa en ensamblador, el siguiente paso es ensamblar dicho código y simular su ejecución. Para ensamblar el código y pasar al modo de simulación, basta con pulsar sobre la pestaña «`ARMSim`» que se encuentra debajo de la sección

¿Cambiar al modo de simulación?

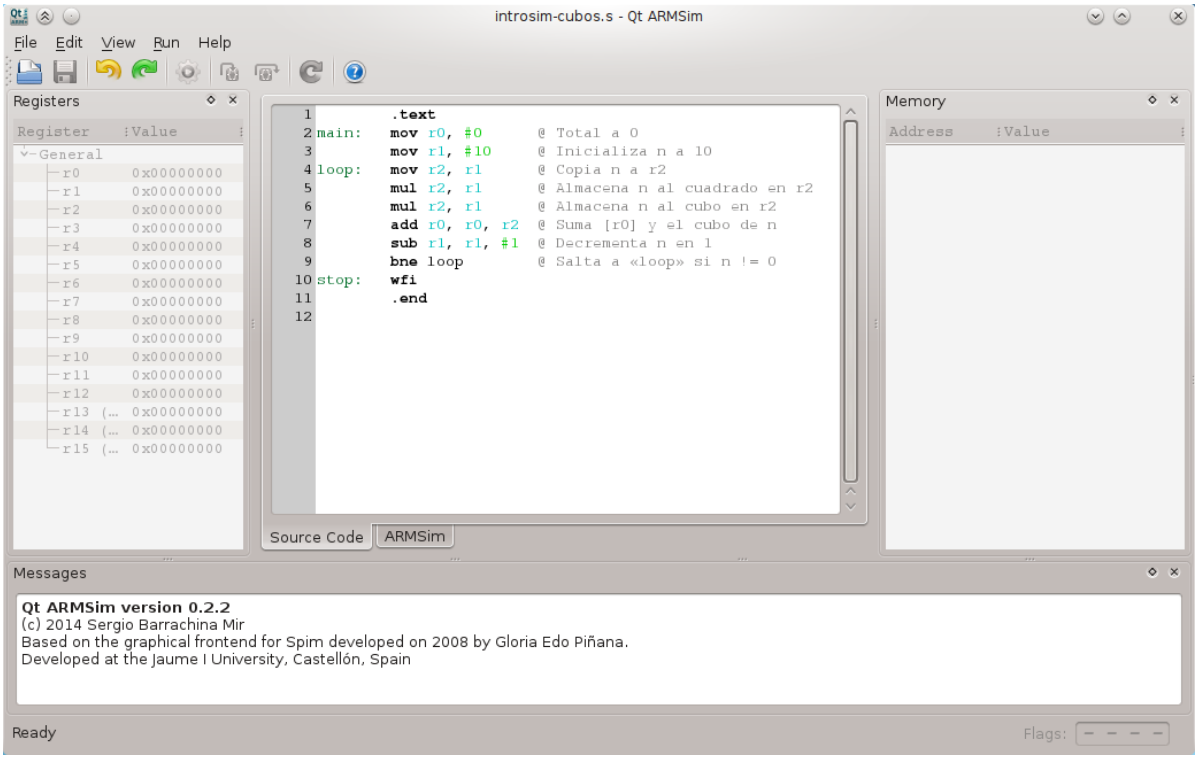


Figura 2.3: QtARMSim mostrando el programa «`02_cubos.s`»

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) llamar al ensamblador de GNU para ensamblar el código fuente; II) actualizar el contenido de la memoria ROM con las instrucciones máquina generadas por el ensamblador; III) inicializar, si es el caso, el contenido de la memoria RAM con los datos indicados en el código fuente; y, por último, IV) inicializar los registros del computador simulado. Si se produjera algún error al intentar pasar al modo de simulación, se mostrará un cuadro de diálogo informando del error, se volverá automáticamente al modo de edición y en el panel de mensajes se mostrarán las causas del error. Es de esperar que la mayor parte de las veces, el error sea debido 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

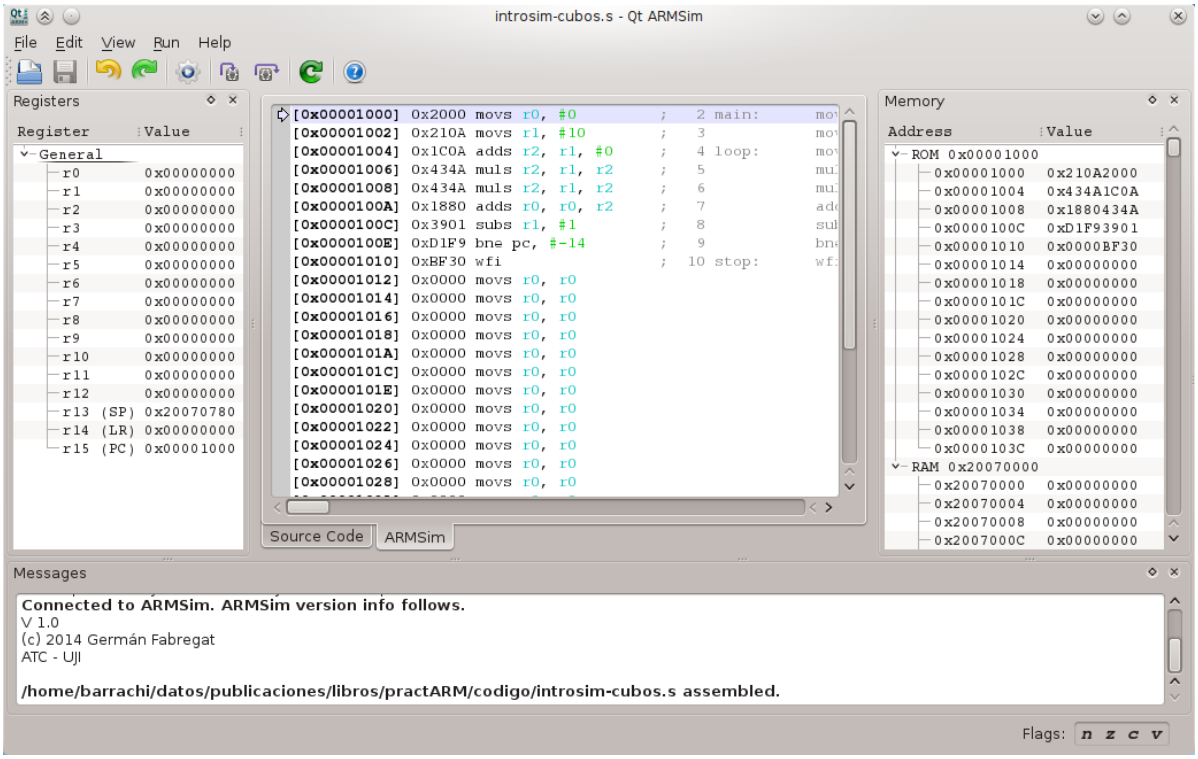


Figura 2.4: QtARMSim en el modo de simulación

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, el contenido de la memoria del computador simulado se muestra en el panel de memoria. En la Figura 2.4 se puede ver que el computador simulado dispone en este ejemplo de dos bloques de memoria: un bloque de memoria ROM que comienza en la dirección `0x0000 1000` y un bloque de memoria RAM que comienza en la dirección `0x2007 0000`. También se puede ver cómo las celdas de la memoria ROM contienen algunos valores distintos de cero (que corresponden a las instrucciones máquina del programa ensamblado) y las celdas de la memoria RAM están todas a cero.

Por otro lado, el contenido de los registros del «`r0`» al «`r15`» se muestra en el panel de registros. El registro «`r15`» merece una mención especial, ya que se trata del **contador de programa** (PC, por las siglas en inglés de *Program Counter*). Como se puede ver en la Figura 2.4, el `PC` contiene en este caso a la dirección de memoria `0x0000 1000`. Como se ha comentado en el párrafo anterior, la dirección `0x0000 1000` es justamente la dirección de memoria en la que comienza el bloque de memoria ROM

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

del computador simulado, donde se encuentra almacenado el programa en código máquina. Así pues, el `PC` está apuntando en este caso a la primera instrucción presente en la memoria ROM, por lo que la primera instrucción en ejecutarse será justamente la primera del programa.

Puesto que los paneles del simulador son empotrables, es posible cerrarlos de manera individual, reubicarlos en una posición distinta, o desacoplarlos y mostrarlos como ventanas flotantes. 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 la disposición por defecto del simulador seleccionando la entrada «`Restore Default Layout`» del menú «`View`» (o pulsando la tecla «`F3`»). Naturalmente, es posible 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 mostrar.

¿Cómo restaurar la disposición por defecto?

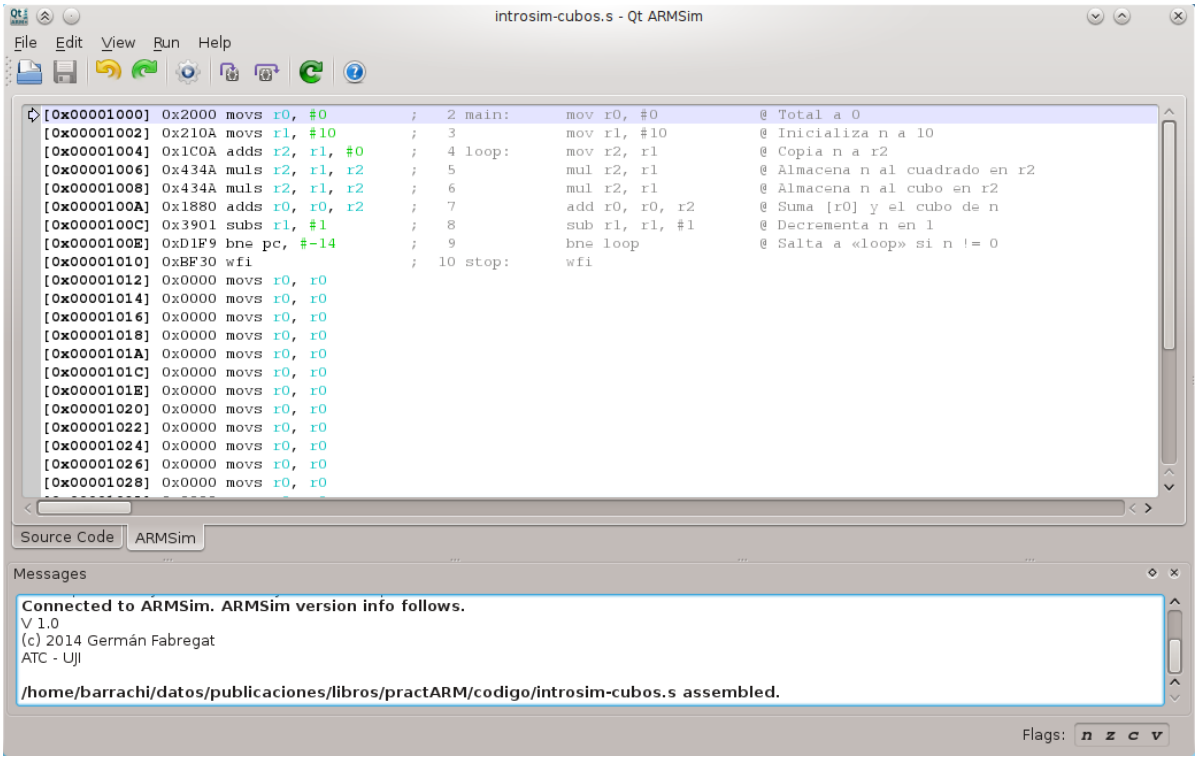


Figura 2.5: QtARMSim sin paneles de registros y memoria

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

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

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

2º La instrucción máquina expresada en hexadecimal.

3º La instrucción máquina expresada en ensamblador.

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

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

- La instrucción máquina está almacenada en la dirección de memoria `0x0000 1000`.
- La instrucción máquina expresada en hexadecimal es `0x2000`.
- La instrucción máquina expresada en ensamblador es «`movs r0, #0`».
- La instrucción 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)?

2.2.3 Localiza el número anterior en el panel 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 dicha instrucción máquina (en hexadecimal)?

2.2.6 Localiza el número anterior en el panel 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 son modificados durante la ejecución del código máquina —no se ha resaltado ninguna de las posiciones de memoria debido a que dicho programa no escribe en memoria—. 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:

```
$ python3 codigo/02_cubos.py
El resultado es: 3025
```

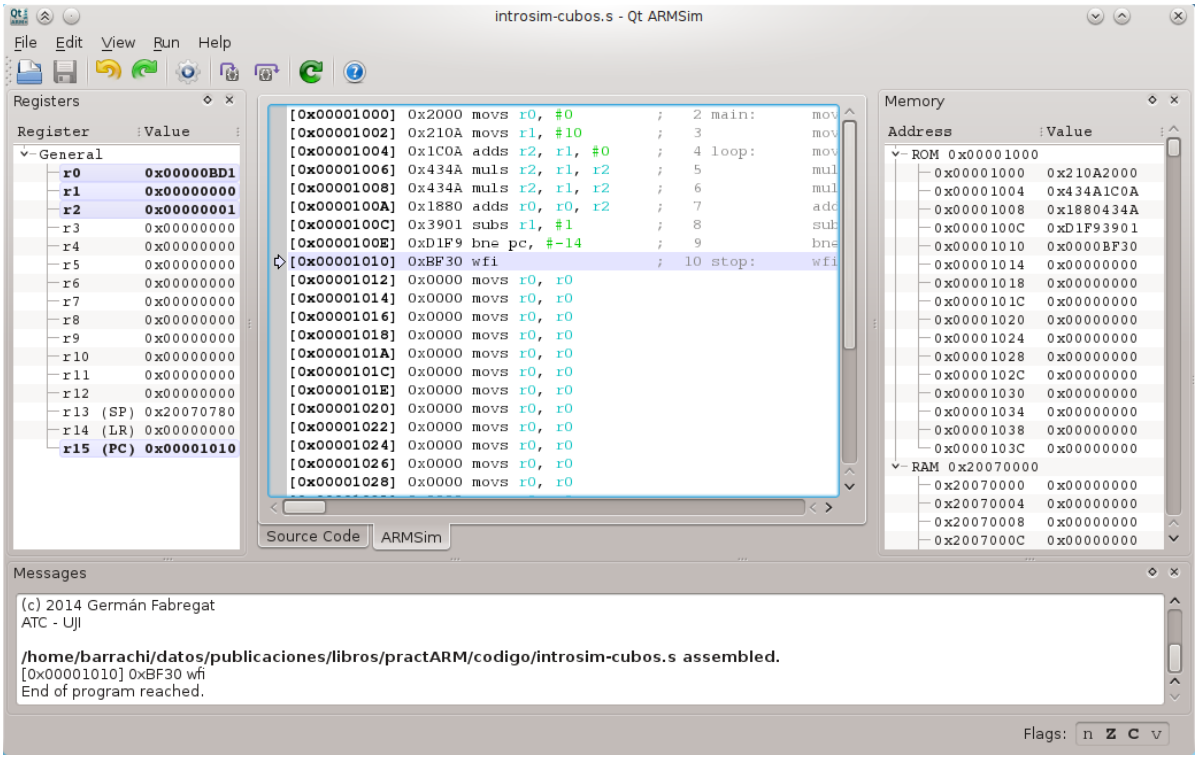
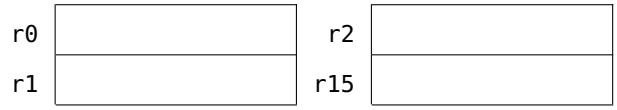


Figura 2.6: QtARMSim después de ejecutar el código máquina

```
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 toman los siguientes registros?



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 dicha instrucción, y al hacerlo, se detendrá, y por tanto, el PC seguirá apuntando a dicha instrucción. Así que para poder ejecutar de nuevo el código, o para iniciar una ejecución paso a paso, como se verá en el siguiente apartado, es necesario recargar previamente la simulación. Para recargar la simulación se debe seleccionar la entrada de menú «`Run > Refresh`», o pulsar la tecla «`F4`».

Ejecución paso a paso

Aunque la ejecución completa de un programa pueda servir para comprobar si el programa hace lo que se espera de él, no permite ver con detalle cómo se ejecuta el programa. Tan solo se puede observar el estado inicial del computador simulado y el estado al que se llega cuando se termina la ejecución del programa. Para poder ver qué es lo que ocurre al ejecutar cada instrucción, el simulador proporciona la opción de ejecutar 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 ésto 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—.

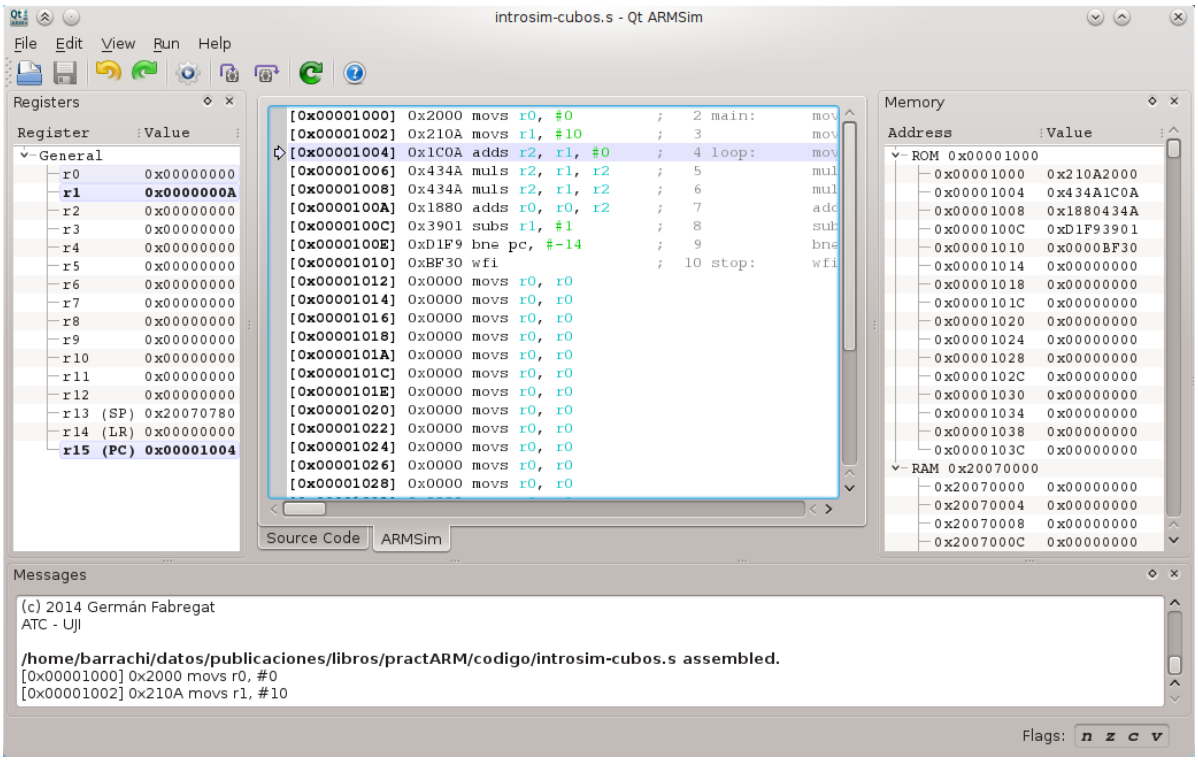


Figura 2.7: QtARMSim después de ejecutar dos instrucciones

Para modificar el contenido del registro `r1` se debe hacer doble clic sobre la celda en la que está su contenido actual (véase Figura 2.8), teclear el nuevo número y pulsar la tecla «`Retorno`». El nuevo valor numérico[4] puede introducirse en decimal, en hexadecimal (si se precede de «0x», p.e., «0x3»), o en binario (si se precede de «0b», p.e., «0b11»). Una vez modificado el contenido del registro `r1` para que contenga el valor 3, se puede ejecutar el resto del código de golpe (menú «`Run > Run`»), no hace falta ir paso a paso. Cuando finalice la ejecución, el registro `r0`

[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.

deberá tener el valor `0x0000 0024`, que en decimal es el número 36, que efectivamente es $3^3 + 2^3 + 1^3$.

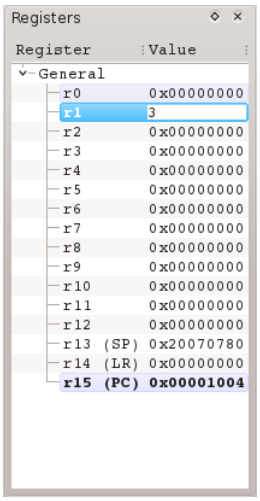


Figura 2.8: Edición del registro `r1`

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 rutinas (también llamadas procedimientos, funciones o subrutinas). Si el código en ensamblador incluye llamadas a rutinas, al utilizar el modo de ejecución paso a paso visto hasta ahora sobre una instrucción de llamada a una rutina, la siguiente instrucción que se ejecutará será la primera instrucción de dicha rutina. Sin embargo, en ocasiones no interesa tener que ejecutar paso a paso todo el contenido de una determinada rutina, puede ser preferible ejecutar la rutina entera como si de una única instrucción se tratara, y que una vez ejecutada la rutina, el PC pase a apuntar directamente a la instrucción siguiente a la de la llamada a la rutina. De esta forma, sería fácil para el programador ver y comparar el estado del computador simulado antes de llamar a la rutina y justo después de volver de ella. Para poder hacer lo anterior, se proporciona una 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 rutina. Ante cualquier otra instrucción, las dos ejecuciones paso a paso harán lo mismo.

Una *rutina* es un fragmento de código que puede ser llamado desde varias partes del programa y que cuando acaba, devuelve el control a la instrucción siguiente a la que le llamó.

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.

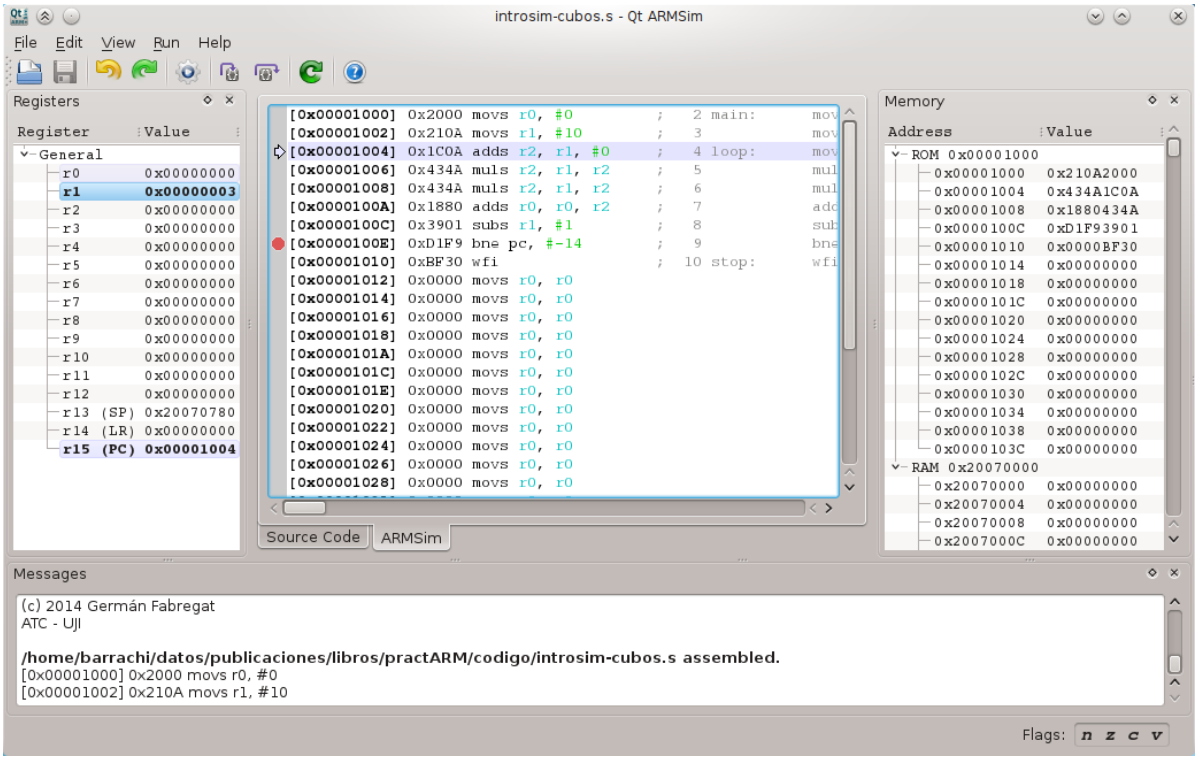


Figura 2.9: Punto de ruptura en la dirección `0x0000100E`

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 `0x0000100E`. Por su parte, la Figura 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).

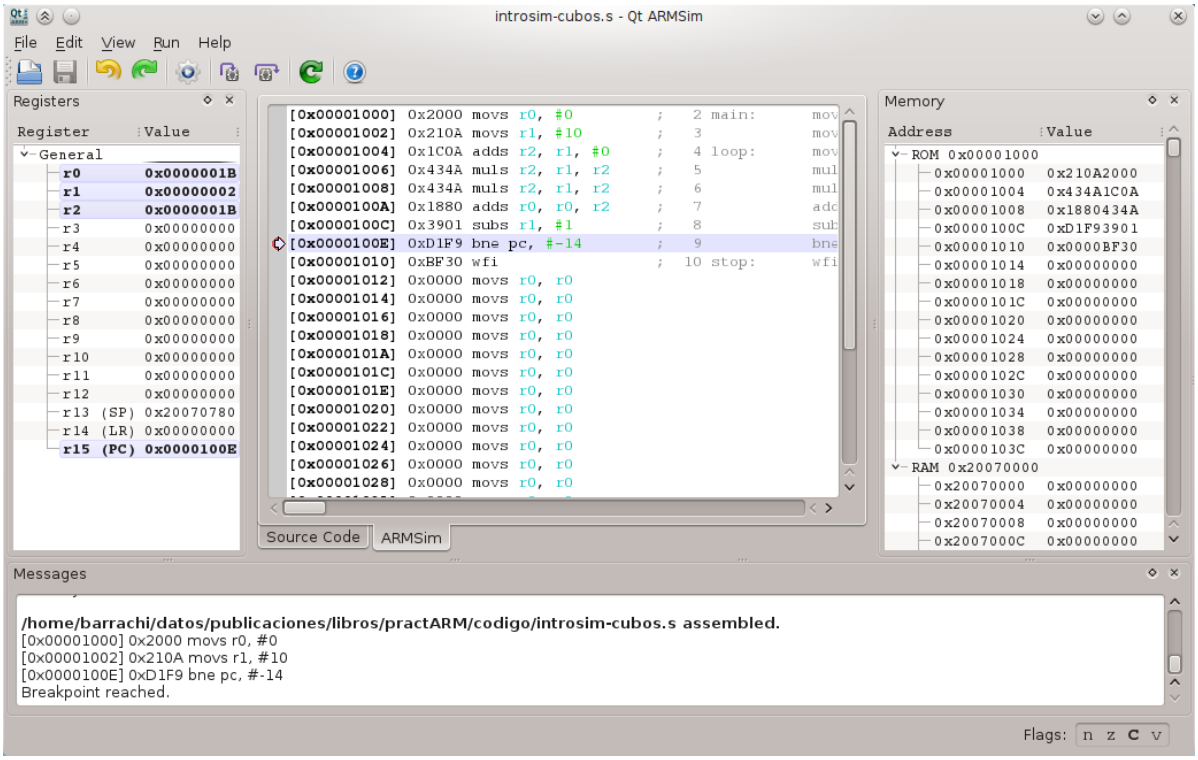


Figura 2.10: Programa detenido al llegar a un punto de ruptura

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, se indica que un determinado dato es un literal precediéndolo de un carácter «`#`». En los apartados anteriores se han visto algunos ejemplos de instrucciones que incluían datos literales. Por ejemplo, el «`#2`» de la instrucción «`mov r0, #2`» indica que se quiere guardar un 2 en el registro `r0`, es decir, en notación RTL: $r0 \leftarrow 2$. Ese 2 de la instrucción «`mov r0, #2`» es un valor literal. Si en lugar de querer guardar un 2 se hubiera querido almacenar otro número, p.e., un 42, se habría tenido que utilizar una instrucción distinta, «`mov r0, #42`», puesto que el dato literal forma parte de la propia instrucción.

Puesto que la representación de un literal que se utiliza más frecuentemente es la de un número en decimal, la forma de indicar un número en decimal es la más sencilla de todas. Basta con anteponer, como ya se ha comentado, el carácter «`#`» al número en decimal —la única precaución que hay que tener es que el número no puede comenzar por 0—. Por ejemplo, como ya se ha visto, la instrucción «`mov r0, #2`» guarda un 2, especificado de forma literal y en decimal, en el registro `r0`.

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 empezar al igual que antes por el carácter «`#`»; a continuación, uno de los siguientes prefijos: «`0x`» para hexadecimal, «`0`» para octal y «`0b`» para binario[5]; y, por último, el número en la base seleccionada.

▶ **2.4** El siguiente código muestra 4 instrucciones que inicializan los registros `r0` al `r3` con 4 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] 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.

```
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`?

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.

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 entrecomilla entre comillas simples[6] y se precede de «`#`», p.e., «`mov r0, #'L'`». Si en lugar de querer especificar un carácter, se quiere especificar una cadena de caracteres[7], entonces se debe utilizar el prefijo «`#`» y entrecomillar la cadena entre comillas dobles. Por ejemplo, «`#"Hola mundo!"`».

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

[7] Puesto que la instrucción en ensamblador «`mov rd, #Offset8`» tan solo puede escribir el byte indicado por `Offset8` en el registro `rd`, no es posible indicar en dicha instrucción una cadena de caracteres, ya que la cadena de caracteres ocupará más de un byte, y el registro destino tan solo puede almacenar 4 bytes. Así pues, se dejan para más adelante los ejemplos de cómo utilizar los literales de cadenas en un programa en ensamblador.

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, etc. 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`»[8]. 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         @ ...
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_diasreq.s

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

[8] 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           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 referenciar a diversos tipo de datos. Será el compilador (o el intérprete, según sea el caso) quien se encargará de decidir en qué posiciones de memoria se almacenarán y cuánto ocuparán los tipos de datos utilizados por dichas variables. El programador simplemente declara e inicializa dichas 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 qué y cuántas posiciones de memoria se deben utilizar para almacenar las variables de un programa, así como indicar sus valores iniciales.

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

«`.data`»

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, etc.

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, la «`.word 15`», inicializará[9] 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?

[9] 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.

2.5.3 Recuerda que las etiquetas sirven para referenciar 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?

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*).

2.4.2. 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 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 ¿Qué dirección de memoria referencia la etiqueta «`byte`»?

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

2.4.5. Reserva de espacio

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

«`.space N`»

▶ **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. Ejercicios

Ejercicios de nivel medio

▶ **2.11** 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

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

2.11.1 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?

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

▶ **2.12** 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
```

2.12.1 ¿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?

2.12.2 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?

2.12.3 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?

▶ **2.13** 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

▶ **2.14** Sea el siguiente programa

02_byte-palabra.s

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

2.14.1 ¿Qué valores se han almacenado en memoria?

2.14.2 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?

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

▶ **2.15** 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[10] de cinco palabras (*words*), asociado a la etiqueta «`vector`», que tenga los siguientes valores: `0x10`, `30`, `0x34`, `0x20` y `60`. Cambia al modo simulador y comprueba que el vector se ha almacenado de forma correcta en memoria.

▶ **2.16** El siguiente programa utiliza las 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
```

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

Ejercicios adicionales

▶ **2.17** 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.

▶ **2.18** 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`.

▶ **2.19** 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
```

▶ **2.20** 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.

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

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

▶ **2.23** 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.)

▶ **2.24** 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 dichas instrucciones operan con el contenido de registros, antes de describir la funcionalidad de dichas instrucciones, se introducirá brevemente el banco de registros de ARM. Además, una vez visto el banco de registros y la funcionalidad de dichas instrucciones, se verán con detalle los modos de direccionamiento utilizados para codificar sus operandos y el formato de estas instrucciones.

3.1. Banco de registros de ARM

El banco de registros de ARM está formado por 16 registros visibles por el programador (véase la Figura 3.1) 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.

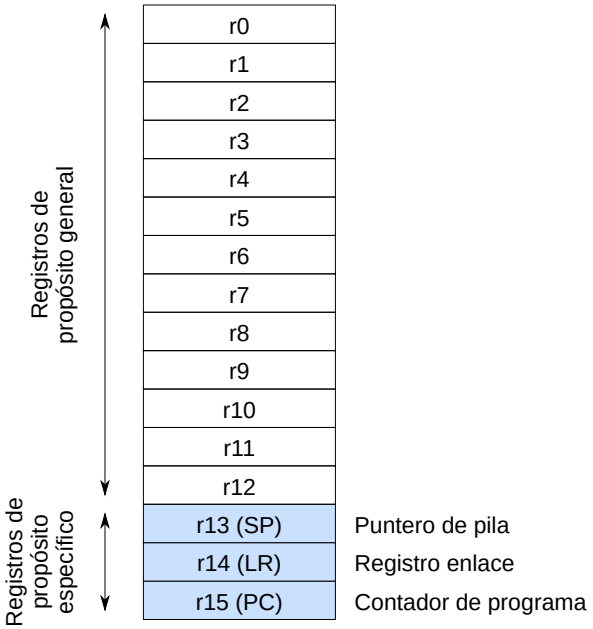


Figura 3.1: Registros visibles de ARM

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. Seleccionar uno de entre 16 registros

requiere 4 bits 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—. Este ahorro en los bits necesarios para codificar los operandos de una instrucción permite a la arquitectura ARM disponer de más bits para codificar instrucciones distintas, lo que le ha permitido proporcionar un juego de instrucciones más rico que el de otras arquitecturas RISC [Cle14].

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 8 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 8 primeros— y registros altos —los 8 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*).

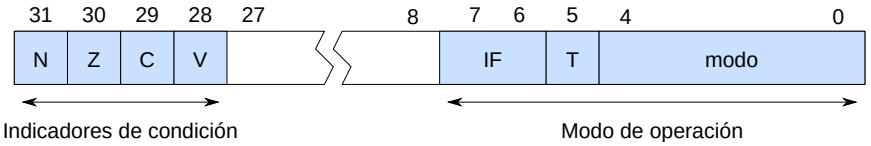


Figura 3.2: Registro de estado —*current processor status register*—

En cuanto al registro de estado (véase la Figura 3.2), los cuatro bits de mayor peso almacenan los indicadores de condición (*condition flags*), que se describen a continuación, y los 8 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[1] o desactiva cada uno de los indicadores:

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

[1] 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 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.

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).

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

Los indicadores de condición del registro de estado se muestran en QtARMSim en la esquina inferior derecha de la ventana del simulador. La Figura 3.3 muestra un ejemplo donde los indicadores `Z` y `C` están activos y los otros dos no.

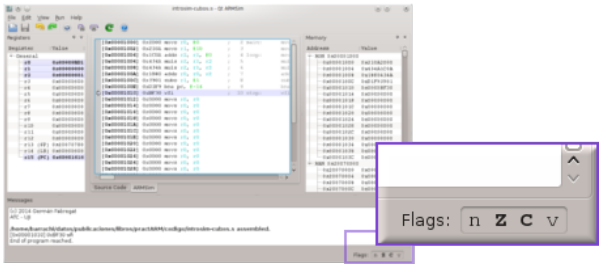


Figura 3.3: Visualización de los indicadores de condición

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` $\leftarrow Inm8$.

3.2. 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,

«**`add`**»

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

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

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

▶ **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 el 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
```

«**cmp**»

Otra operación aritmética que se utiliza con frecuencia es la de comparación, «`cpm 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` *se utiliza para indicar si el resultado cabe en una palabra (*`C` *activado) o, por el contrario, si no cabe en una palabra (*`C` *desactivado). Así, que* `C` *esté desactivado equivale al* «me llevo una» *de cuando se resta a mano.*

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.

▶ **3.5** Copia el siguiente programa en el simulador, ejecútalo y completa la tabla que le sigue.

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.

3.3. 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»).
- «**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 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:

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	$\bar{b}$

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

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 dedicados a la entrada/salida, el uso de máscaras de bits es muy frecuente en la gestión de la entrada/salida.

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.

▶ **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
```

3.4. 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 las tres siguientes: «**asr**», «**lsr**» y «**lsl**»

- «**asr** `rd, rs`», del inglés *arithmetic shift right*, que desplaza el contenido de `rd` hacia la derecha el número de posiciones indicadas por el contenido de `rs` conservando el signo (es decir, $rd \leftarrow rd\ ASR\ rs$).
- «**lsr** `rd, rs`», del inglés *logic shift right*, que desplaza el contenido de `rd` hacia la derecha el número de posiciones indicadas por el contenido de `rs` rellenando con ceros por la izquierda. (es decir, $rd \leftarrow rd >> rs$).
- «**lsl** `rd, rs`», del inglés *logic shift left*, que desplaza el contenido de `rd` hacia la izquierda el número de posiciones indicadas por el contenido de `rs` rellenando con ceros por la derecha. (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 >> 1
5         lsr r0, r1      @ r0 <- r0 >> 1
6         lsr r0, r1      @ r0 <- r0 >> 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...0100000`$_2$
3:	«`mov r1, #1`»	32	`0x00000020`	`0...0100000`$_2$
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.

3.5. 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 de los operandos, principalmente por los siguientes motivos [BMLNMA14]:

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

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

- Para poder reubicar el código. Si la dirección de los operandos en memoria solo se pudiera expresar por medio de una dirección de memoria fija, cada vez que se ejecutara un determinado programa, éste buscaría los operandos en las mismas direcciones de memoria, por lo que tanto el programa como sus datos habrían de cargarse siempre en las mismas direcciones de memoria. ¿Qué pasaría entonces con el resto de programas que el computador puede

ejecutar? ¿También tendrían direcciones de memoria reservadas? ¿Cuántos programas distintos podría ejecutar un computador sin que éstos se solaparan? ¿De cuánta memoria dispone el computador? Así pues, es conveniente poder indicar la dirección de los operandos de tal forma que un programa pueda ejecutarse independientemente de la zona de memoria en la que haya sido cargado para su ejecución.

Por 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 correspon-

diente 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 [ARM95].

3.5.1. Direccionamiento directo a registro

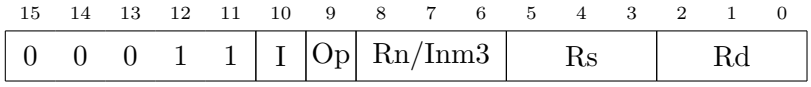
El direccionamiento directo a registro es el más simple de los modos de direccionamiento ya que el operando se encuentra en un registro y en la instrucción simplemente se debe codificar en cuál. Este modo de direccionamiento se utiliza en la mayor parte de instrucciones, tanto de transferencia, como 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 8 registros bajos y 8 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.

3.5.2. 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 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.

3.5.3. 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 3 registros o con 2 registros



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 3.4: 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

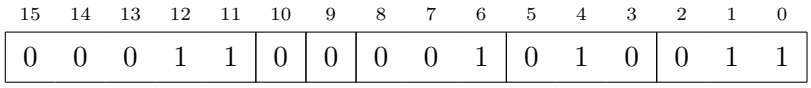


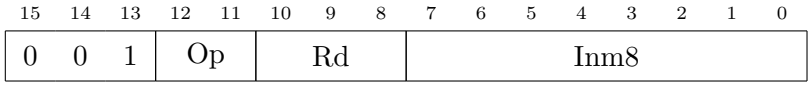
Figura 3.5: Codificación de la instrucción «`add r3, r2, r1`»

y un dato inmediato de 3 bits, ocupa 16 bits y está formado, de izquierda a derecha, por los siguientes campos (véase Figura 3.4):

- 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 3.5.

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



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

Rd Registro fuente/destino.

Inm8 Dato inmediato.

Figura 3.6: Formato de las instrucciones «`mov rd, #Inm8`», «`cmp rd, #Inm8`», «`add rd, #Inm8`» y «`sub rd, #Inm8`»

3.5.4. 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 «`sub rd, #Inm8`», se codifican utilizando el formato de instrucción mostrado en la Figura 3.6. 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`».

▶ **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`»

3.6. Ejercicios

Ejercicios de nivel medio

▶ **3.12** ¿Qué ocurre si se intenta ensamblar la siguiente 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	0x0000 000A	0xFFFF FFF6	0xFFFF FFF5
-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 (pseudo-)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 >> 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 (*logic shift right*), también desplaza a la derecha un determinado número de bits el valor indicado. Sin embargo, no tiene en cuenta el signo y rellena siempre con ceros. Modifica el programa original para que utilice la instrucción «**`lsr`**» en lugar de la «**`asr`**» ¿Qué valor se obtiene ahora en `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 el valor `0x00FF F0F0`.

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 fuera a utilizar un programa y cómo inicializar dicho espacio. El Capítulo 3 abordó las instrucciones de transformación de datos —aritméticas, lógicas y de desplazamiento—. Tal como se vio en dicho capítulo, las instrucciones de transformación requerían que los datos se hubieran cargado previamente en registros o formaran parte de la propia instrucción. De hecho, se utilizó frecuentemente en dicho capítulo una instrucción de transferencia: «`mov rd, #Inm8`», que carga un dato inmediato de 8 bits en el registro indicado.

Este capítulo se centra 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, 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 la ejecución canalizada (*pipelined execution*) de las instrucciones de transformación —gracias a que siempre operan 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

«`mov rd, #Inm8`»

La instrucción «`mov rd, #Inm8`», utilizada profusamente en el capítulo anterior, permite cargar una dato inmediato que quepa en un byte en el registro `rd` (es decir, $[\texttt{rd}] \leftarrow \#Inm8$).

▶ **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 pseudo-instrucción que sí puede hacerlo: «**ldr** rd, =Inm32». Dicha pseudo-instrucción 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 pseudo-instrucción? 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 pseudo-instrucción 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 pseudo-instrucción por una instrucción de carga relativa al `PC`.

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

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

4.2.2 La pseudo-instrucción «**`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 `0x0000 1004`. Por otro lado, como has podido ver en la pregunta anterior, la instrucción anterior tiene un dato inmediato con valor 4. ¿Cuánto es `0x0000 1004` + 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 pseudo-instrucción «**`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 pseudo-instrucción 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, empezado por cómo cargar palabras. Para cargar una palabra de memoria a 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]]). Enp 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», debe ser un número múltiplo de 4 entre 0 y 124, puesto que los datos deben estar alineados. Conviene observar que la variante anterior, «**ldr** rd, [rb]», es en realidad una pseudo-instrucción que será sustituida por el ensamblador por una instrucción de este tipo con un desplazamiento de 0, es decir, por «**ldr** rd, [rb, #0]». 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 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 utilizar una de las siguientes instrucciones: «`str rd, [...]`»

- «`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 pseudo-instrucción que será sustituida por el ensamblador por una instrucción de este tipo con un desplazamiento de 0, es decir, por «`str rd, [rb, #0]`». 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 el código paso a paso hasta la instrucción «`str r2, [r0]`» inclusive y comprueba si es correcto lo que has contestado en el ejercicio anterior.

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

```
        .data
byte1:  .space 1
byte2:  .space 1
byte3:  .space 1

        .text
main:   ldr r0, =byte1   @ r0 <- 0x20070000
        mov r1, #2       @ r1 <- 2
        mov r2, #10      @ r2 <- 10
        strb r2, [r0]
        strb r2, [r0,#1]
        strb r2, [r0,r1]
stop:   wfi
```

Por su parte, ara 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

```
        .data
hword1: .space 2
hword2: .space 2
hword3: .space 2

        .text
main:   ldr r0, =hword1  @ r0 <- 0x20070000
        mov r1, #4       @ r1 <- 4
        mov r2, #10      @ r2 <- 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]`», realiza la operación $[\texttt{rd}] \leftarrow [[\texttt{rb}] + \texttt{Offset5}]$[2], 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,

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

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

`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.

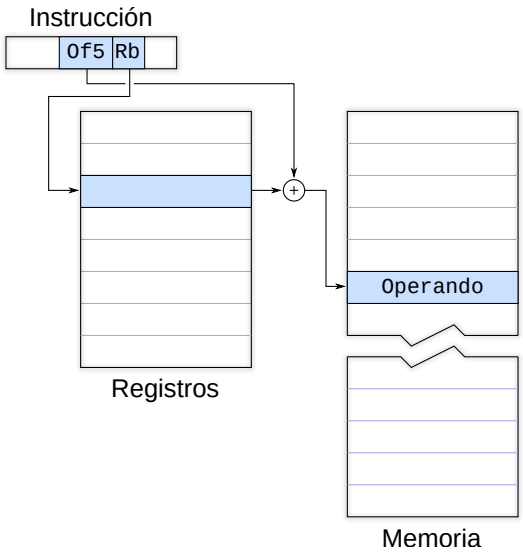


Figura 4.1: Modo de direccionamiento indirecto con desplazamiento. En este modo, la dirección efectiva del operando es una posición de memoria. En la instrucción se codifica el número del registro `rb` y un dato inmediato, `Offset5`, la suma del contenido de dicho registro y el dato inmediato proporciona 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 pseudo-instrucciones «`ldr rd, =Inm32`» y «`ldr rd, =Label`».

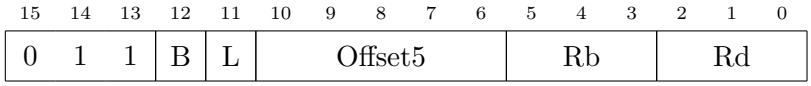
4.3.3. Direccionamiento indirecto con registro de desplazamiento

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 3 bits 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 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]`»

- 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 ($L = 0$).
- 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

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.

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

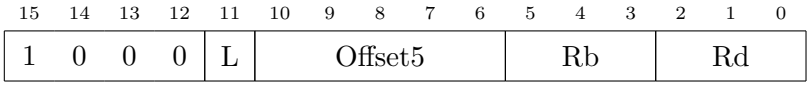
¿Cómo se codificaría un desplazamiento de, por ejemplo, 20 bytes en una instrucción de carga de bytes?, ¿y en una de carga de palabras? En la de carga de bytes, p.e., «`ldrb r1, [r0, #20]`»), el número 20 se codificaría tal cual en el campo `Offset5`. Por tanto, en `Offset5` se pondría directamente el 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 Figura 4.3) ligeramen-



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

Offset5 Dato inmediato.

Rb Registro base.

Rd Registro fuente/destino.

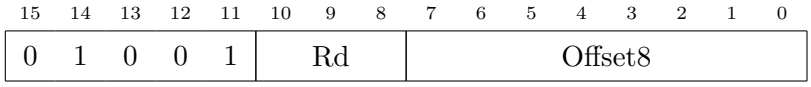
Figura 4.3: Formato de las instrucciones de carga/almacenamiento de medias palabras con direccionamiento indirecto con desplazamiento: «`ldrh rd, [rb, #Offset5]`» y «`strh rd, [rb, #Offset5]`»

te 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 cuanto al cuarto bit de mayor peso, el bit 12, en el caso del formato de instrucción para bytes y palabras, éste podía tomar como valor un 0 o un 1, mientras que en el formato de instrucción para medias palabras siempre vale 0. 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—.

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*.

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 `PC` + 2 (o `PC` + 4, dependiendo del tamaño de la instrucción) alineado a 4 más el dato inmediato `Offset8`. ¿Por qué `PC` + 2, o `PC` + 4, en lugar de `PC`? Cuando el procesador va a calcular la dirección del operando fuente de esta instrucción, que viene dado por la suma del contenido del registro `PC` más el desplazamiento `Offset8`, en la fase de ejecución de la instrucción



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]`»

(véase el Apartado 1.2.2), el `PC` se habrá actualizado en la fase previa de incremento del contador de programa a `PC`+2 (o a `PC`+4, dependiendo del tamaño de la instrucción). Por eso la suma se realiza con respecto al `PC` actualizado en lugar de con respecto al `PC` inicial. Ahora, ¿por qué `PC` + 2, o `PC` + 4, alineado a 4? Puesto que las instrucciones Thumb ocupan una o media palabra, el contenido actualizado del `PC` podría no ser múltiplo de 4, por lo que al hacer la suma con el desplazamiento `Offset8`, no se podría garantizar que el resultado fuera múltiplo de 4 y, por tanto, que fuera una dirección de memoria válida para una palabra. Así pues, el contenido actualizado del `PC` se alinea a 4 —lo que se realiza poniendo el bit 1 del `PC` a 0— para garantizar que dicho sumando sea múltiplo de 4 [ARM95].

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. Además, puesto que la instrucción «`ldr rd, [PC, #Offset8]`» carga una palabra cuya dirección de memoria tiene que estar alineada a 4, el campo Offset8, de 8 bits, 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 serán siempre 0—, por lo que el rango del desplazamiento es en realidad de $[0, 1020]$ bytes —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]`».

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:

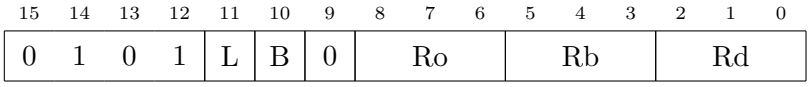
- «`ldr rd, [rb, ro]`»,
- «`ldrb rd, [rb, ro]`»,
- «`str rd, [rb, ro]`» y
- «`strb rd, [rb, ro]`».

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.

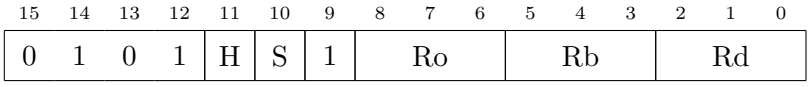
B Byte/Word: 1, transferir byte; 0, palabra.

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 la instrucción «**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 la instrucción «**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 para una palabra a continuación; y transfiera los cuatro bytes iniciales a la palabra reservada.

Capítulo 5

Instrucciones de control de flujo

Índice

Los programas mostrados en los anteriores capítulos constaban de una serie de instrucciones que se ejecutaban una tras otra, siguiendo el orden en el que se habían escrito, hasta que el programa finalizaba. Este tipo de ejecución, en el que las instrucciones se ejecutan una tras otra, siguiendo el orden en el que están en memoria, recibe el nombre de **ejecución secuencial**. 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. Pero incluso en ese caso, el programa sería incapaz de variar el número de veces que dichas instrucciones se ejecutarían.

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.

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 condiciona-

les y repetitivas. Estas estructuras de control permiten modificar el flujo secuencial de instrucciones. En particular, las estructuras de control condicionales permiten la ejecución de ciertas partes del código en función de una serie de condiciones, mientras que las estructuras de control repetitivas permiten la repetición de cierta parte del código hasta que se cumpla una determinada condición de parada.

Este capítulo 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.

El registro de estado (recordatorio)

Tal y como se vio en el capítulo anterior (§ 3.1), 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.

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 incodicionales, y a continuación, los condicionales.

5.1.1. Saltos incondicionales

Los saltos incondicionales son aquéllos 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 «`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 tienen la 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`»

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** 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

```
        .text
main:   mov r0, #5
        mov r1, #10
        mov r2, #5
        mov r3, #0
        cmp r0, r2
        beq salto
        add r3, r0, r1
salto:  add r3, r3, r1
stop:   wfi
```

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

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.

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

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

05_if.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 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.

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

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

apartado anterior, que además de permitir indicar qué se quiere hacer cuando se cumpla una determinada condición, permite especificar qué acciones se deben ejecutar en el caso de que no se verifique la condición indicada. 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`.

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

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

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

```

```
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
```

```
20          str r0, [r4]          @ [X] <- r0
21          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 hacen 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*. Ejercicio para el lector: ¿cómo se resolvería el mismo problema utilizando la estructura *while*?

```
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] ← [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 ¿Para qué se utilizan 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 ¿Qué contiene el registro `r0`? ¿Para qué sirve la instrucción «`ldr r4, [r0]`»?

5.7.4 ¿Para qué sirve la instrucción «`add r0, r0, #4`»?

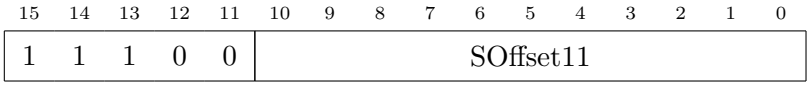
5.7.5 ¿Qué valor contiene la dirección de memoria «`suma`» cuando finaliza la ejecución del programa?

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 incondicionales, «`b etiqueta`», como condicionales, «`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? Puesto que las direcciones de memoria en ARM ocupan 32 bits, si se quisieran codificar los 32 bits del salto en la instrucción, sería necesario recurrir a instrucciones que ocuparan más de 32 bits (al menos para las instrucciones de salto, ya que no todas las instrucciones tienen por qué ser del mismo tamaño). Pero además, en el caso de utilizar esta aproximación, la de codificar la dirección completa del salto como un valor absoluto, se forzaría a que el código se tuviera que ejecutar siempre en las mismas direcciones de memoria. Esta limitación se podría evitar durante la fase de carga del programa. Bastaría con que el



SOffset11 Dato inmediato con signo.

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

programa cargador, cuando cargue un código para su ejecución, sustituya las direcciones de salto absolutas por nuevas direcciones que tengan en cuenta la dirección de memoria a partir de la cual se esté cargando el código [Shi13]. En cualquier caso, esto 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 codifica como un desplazamiento de 12 bits con respecto al contenido del registro `PC`. Sin embargo, no hemos dicho mucho del contenido del registro `PC`. Teóricamente, el contenido del registro `PC` se actualiza al valor `PC`+2 al comienzo de la ejecución de la instrucción «`b etiqueta`», puesto que la instrucción «`b etiqueta`» ocupa 2 bytes. Por tanto, el valor del desplazamiento debería ser tal que al sumarse a `PC`+2 diera la dirección de salto. En la práctica, cuando el procesador va a ejecutar el salto, el contenido del `PC` se ha actualizado al valor del `PC`+4. Esto es debido a que se utiliza una técnica conocida como precarga de instrucciones. Por tanto, en realidad el desplazamiento debe ser tal que al sumarse al `PC`+4 proporcione la dirección de memoria a la que se quiere saltar. Para ilustrar lo anterior, fíjate que el siguiente código está formado por varias instrucciones que saltan al mismo sitio:

05_mod_dir_b.s

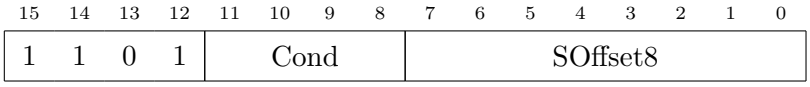
```
1         .text
2 main:   b  salto
3         b  salto
4         b  salto
5         b  salto
6 salto:  mov r0, r0
7         mov r1, r1
8         b   salto
9         b   salto
10
11 stop:  wfi
```

▶ **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 regis-

[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`.



Cond Condición.

SOffset8 Dato inmediato con signo.

Figura 5.3: Formato de las instrucciones de salto condicional

tro `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 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 $[-512, 510]$ con respecto al `PC+4`.

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 $[2, 4, 6, 3, 10, 12, 2]$ 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 una parte del programa a la que se puede llamar para resolver una tarea específica.

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 progra-

ma 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, etc. 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`».

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 completada 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 instruc-

ción «`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 esta 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 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 ésta 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, `LR` ← `PC` + 4[1].

- 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, sobreescribirá 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.

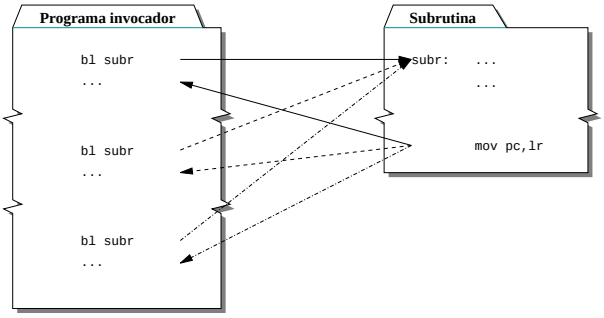


Figura 6.1: Llamada y retorno de una subrutina

El siguiente código muestra un programa de ejemplo en ensamblador de ARM que utiliza una subrutina llamada «`suma`». Esta subrutina suma

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

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`»—.

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.

En base a 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.

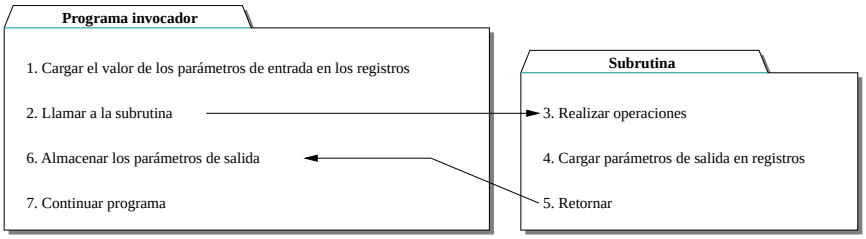


Figura 6.2: Paso de parámetros por valor

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 a la subrutina.

6.2.2 ¿Los anteriores registros se han utilizado para pasar parámetros de entrada o de salida?

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

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

6.2.5 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.

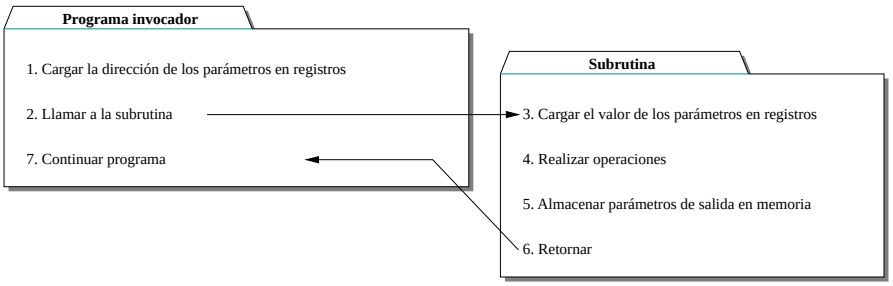


Figura 6.3: Paso de parámetros por referencia

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 quiere desarrollar tiene por objeto calcular cuántos elementos de un vector dado tienen un determinado valor n, p.e., `12`. Para poder practicar el desarrollo de subrutinas y comprobar que es posible llamar a la misma subrutina con parámetros distintos, la solución que se propone incluye el desarrollo de una subrutina que devuelva el número de elementos de un vector que son menores a un número dado. De esta forma, será necesario llamar a dicha subrutina con los valores $n + 1$, primero, y n, después, `13` y `12` en el

ejemplo, y restar los valores devueltos por dichas llamadas para obtener el resultado buscado —el número de elementos que son iguales a n—.

A continuación se muestra una posible implementación en Python3 de dicho programa. Como se puede ver, el programa consta de una subrutina, «`nummenorque`», 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 menores que «`dato_s`»; y devuelve dicho número como resultado. Por su parte, el programa principal inicializa un vector, su tamaño y la variable «`num`»; a continuación, llama dos veces a la anterior subrutina, con «`num+1`» y «`num`», respectivamente, almacenando el resultado de ambas llamadas; y finalmente, resta ambos resultados para obtener el número de elementos del vector que son iguales a «`num`».

```
1  def nummenorque(vector_s, dim_s, dato_s):
2        n = 0
3        for i in range(dim_s):
4             if vector_s[i] < dato_s:
5                n = n + 1;
6        return n
7
8  vector = [5, 3, 5, 5, 8, 12, 12, 15, 12]
9  dim = 9
10 num = 12
11 res1 = nummenorque(vector, dim, num + 1)
12 res2 = nummenorque(vector, dim, num)
13 res = res1 - res2
```

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 menores a un valor dado. Para ello, hay que determinar qué parámetros debe recibir dicha subrutina, así como qué registros se van a utilizar para ello, y cuáles se pasarán por referencia y cuáles por valor. Una vez desarrollada la subrutina, y teniendo en cuenta los parámetros que requiere, se deberá desarrollar la parte del programa que llama dos veces a dicha subrutina y calcula el número de elementos del vector dado que son iguales a n restando 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 pasan 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`», ejecuta de nuevo el programa y comprueba que el resultado obtenido 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 iguales a dicho número. Cambia el nombre de la subrutina a «`numigualque`».

b) Modifica el programa principal del Ejercicio 6.6 para que almacene en memoria el número de elementos de un vector —p.e el que aparecía en el ejercicio original, $[5, 3, 5, 5, 8, 12, 12, 15, 12]$— que son iguales a 3 y el número de elementos que son 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, te hagas las preguntas del 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. Naturalmente, 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). Se debe tener en cuenta que la matriz debería poder tener cualquier dimensión, así que se deberá utilizar un bucle para recorrer sus filas.

Capítulo 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 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 una subrutina, será conveniente utilizar un registro del `r4` en adelante, ya que la subrutina tiene la obligación de preservar su contenido. Desde el punto de vista del desarrollo de la subrutina, el que solo se puedan modificar los registros del `r0` al `r3` implica que, salvo si la subrutina es muy sencilla y es 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 necesite 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 coma 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 proponen una serie de ejercicios.

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, éste deberá colocarse encima de los que ya hay (lo que implica que no es posible insertar un libro entre los que ya están en el montón). Por otro lado, cuando se quiera quitar un libro, solo se podrá quitar el libro que esté más arriba en el montón (por tanto, no se puede quitar un libro en particular si previamente no se han quitado todos los que estén encima de él). Teniendo en cuenta dichas restricciones, el montón de libros actúa como una pila, ya que solo se pueden colocar nuevos libros sobre los que ya están en el montón y el último libro colocado en la pila de libros será el primero en ser sacado de ella.

Un computador 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`.

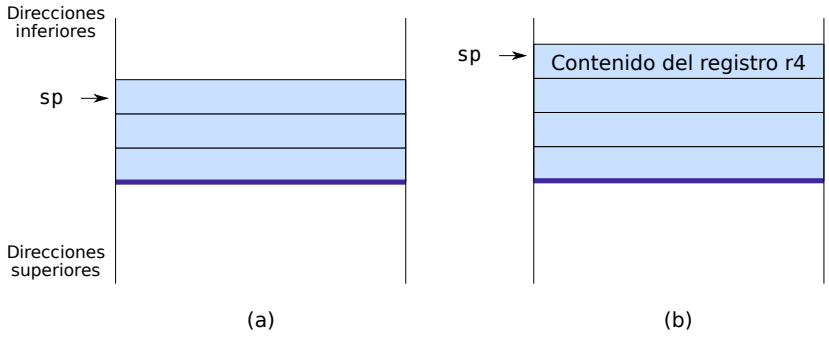


Figura 7.1: La pila antes y después de apilar el registro `r4`

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]`**», e 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.

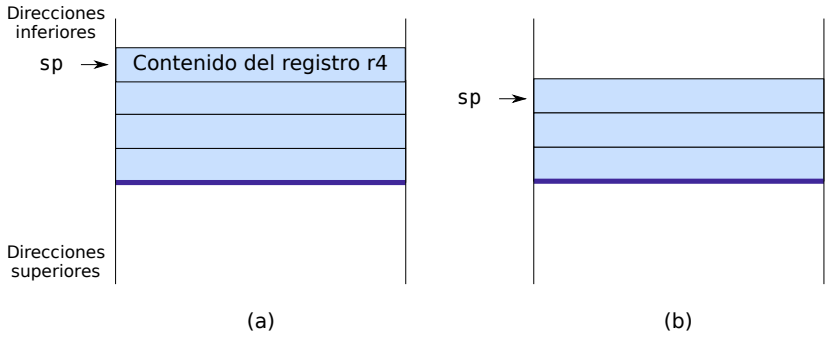


Figura 7.2: La pila antes y después de desapilar el registro `r4`

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

7.1.1 Suponiendo que el puntero de pila contiene `0x7fffeffc` y que se desea apilar una palabra (4 bytes), ¿qué valor deberá pasar a tener el puntero de pila antes de almacenar la nueva palabra en la pila? ¿Qué instrucción se utilizará para hacerlo en el ensamblador ARM?

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 `0x7fffeff8`, ¿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»

«push»
«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 guión (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**».

07_apilar_r4r5_v2.s

```
1         .text
2 main:
3         mov r4, #10
4         mov r5, #13
5         push {r5, r4}
6         add r4, r4, #3
7         sub r5, r5, #3
8         pop {r5, r4}
```

```
9
10 stop:   wfi
11         .end
```

▶ **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 tienen los registros `r4` y `r5` una vez realizadas las operaciones de suma y resta?

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 {r5, r4}`». Si se hubiese empleado la instrucción «`push {r4, r5}`», ¿en qué posiciones de memoria, pertenecientes a la pila, se hubieran almacenado los contenidos de los registros `r4` y `r5`? Según esto, ¿cuál es el criterio que sigue ARM para copiar los valores de los registros en la pila mediante la instrucción «`push`»?

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 cumple los siguientes cometidos:

- Almacenar la dirección de retorno original, en el caso 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.

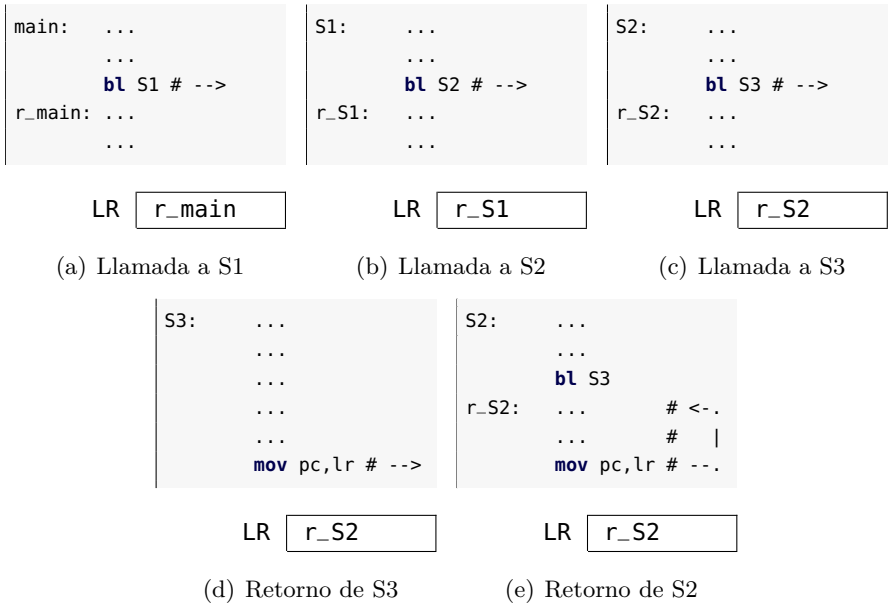
Los siguientes subapartados describen los distintos aspectos de la gestión de las subrutinas en los que se hace uso del bloque de activación de una subrutina. El Subapartado 7.2.1 trata el problema del anidamiento de subrutinas —subrutinas que llaman a otras o a sí mismas—. El Subapartado 7.2.2 cómo utilizar el bloque de activación para almacenar las variables locales de la subrutina. El Subapartado 7.2.3, 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 deberá devolver el control del programa a la instrucción siguiente a la que la llamó una vez finalizada su ejecución. 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 sobreescribir el registro `PC` con la dirección de vuelta, p.e., utilizando la instrucción «`mov pc, lr`». Si durante la ejecución de la subrutina, ésta llama a otra,

o a sí misma, al ejecutarse la instrucción «**bl**», se almacenará en el registro LR la nueva dirección de retorno, sobreescribiendo 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

```
        .data
datos:  .word 5, 8, 3, 4
res:    .space 8
        .text

main:   ldr r0, =datos  @ Parámetros para sumas
        ldr r1, =res
salto1: bl sumas        @ Llama a la subrutina sumas
stop:   wfi             @ Finaliza la ejecucion

sumas:  mov r7, #2
        mov r5, r0
        mov r6, r1
for:    cmp r7, #0
        beq salto4
          ldr r0, [r5]  @ Parámetros para suma
          ldr r1, [r5,#4]
salto2:   bl suma       @ Llama a la subrutina suma
          str r0, [r6]
          add r5, r5, #8
          add r6, r6, #4
          sub r7, r7, #1
        b for
salto4: mov pc, lr

suma:   add r0, r0, r1
salto3: mov pc, lr
        .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.

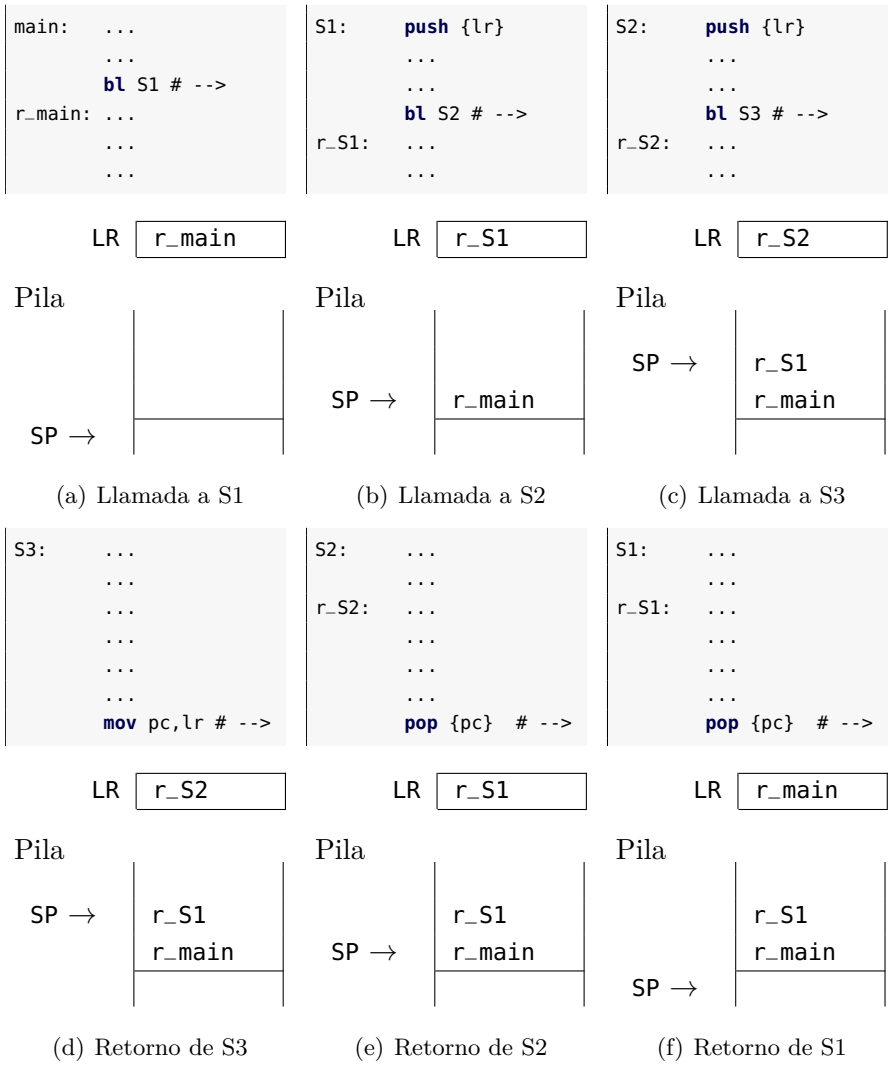
Como ha quedado patente en la Figura 7.3 y en el Ejercicio 7.5, cuando se realizan llamadas anidadas, es necesario almacenar de alguna forma las distintas direcciones de retorno. Dicho almacenamiento debe satisfacer dos requisitos. En primer lugar, debe ser capaz de permitir recuperar las direcciones 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 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.



(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.

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 sobreescribirá durante la ejecución de la subrutina. Así que si la información que contenían dichos registros era 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 del 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 utilizando la instrucción «`ldr r4, [sp, #8]`».

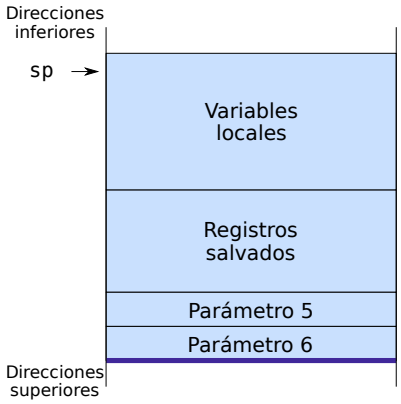


Figura 7.5: Esquema del bloque de activación

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.

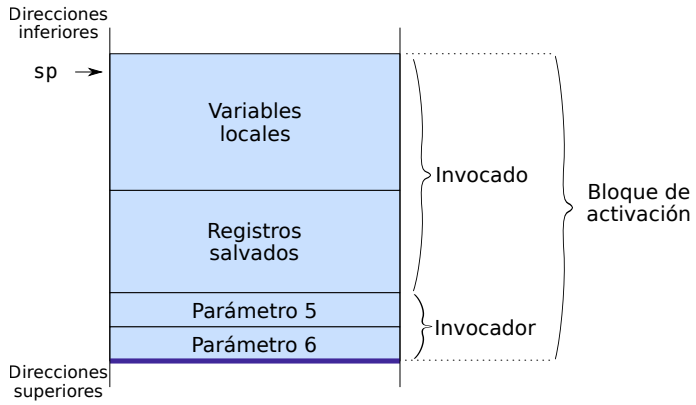


Figura 7.6: Estado de la pila después de una llamada a subrutina

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, $a'_i = \sum_{j=i}^{j=dim-1} a_j, \forall i \in [0, dim]$.

```
def sumatorios(A, dim):
      B = [0]*dim
      for i in range(dim):
           B[i] = sumatorio(A[i:], dim-i)

      for i in range(dim):
           A[i] = B[i]
      return

def sumatorio(A, dim):
      suma = 0;
      for i in range(dim):
           suma = suma + A[i]
      return suma

A = [6, 5, 4, 3, 2, 1]
dim = 6
sumatorios(A, dim)
```

A continuación se muestra el equivalente en ensamblador ARM del anterior programa en Python3.

07_varlocal.s

```
            .data
A:          .word 7, 6, 5, 4, 3, 2
dim:        .word 6

            .text
@---------------------------------------
@ Programa invocador
@---------------------------------------
main:       ldr r0, =A
            ldr r1, =dim
            ldr r1, [r1]
            bl sumatorios

fin:        wfi

@---------------------------------------
@ Subrutina sumatorios
@---------------------------------------
```

```
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, `subrutina-varlocal-v0.s`, contesta 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 pasará como parámetros la matriz, su dimensión y devolverá el máximo buscado.
- «`subr2`» calculará la suma de los elementos de un vector. Se le pasará como parámetros un vector y su dimensión y la subrutina devolverá la suma de sus elementos.
- «`subr3`» calculará el máximo de los elementos de un vector. Se le pasará como parámetros un vector y su dimensión y devolverá el máximo de dicho vector.
- El programa principal se encargará de realizar la inicialización de la dimensión de la matriz y de sus elementos y llamará a la subrutina «`subr1`», quién devolverá el máximo buscado. El programa principal deberá almacenar este dato en la 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 lado, la subrutina «`subr3`» devolverá un 1 si el contenido de los dos vectores que se le pasa como parámetros son iguales.

Suponer que las cadenas están compuestas por el conjunto de letras que componen el abecedario en minúsculas.

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

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

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 tengamos en el mismo equipo unos altavoces para reproducir audio, una impresora para generar copias de nuestros trabajos, un disco duro externo y, por supuesto, una conexión a internet —aunque pueda no verse al no utilizar cables—. Todos estos elementos enumerados, aunque sean de naturaleza y función completamente distinta, se consideran dispositivos periféricos del ordenador y son una parte —en algunos casos la más alejada del ordenador, como los altavoces— de su entrada/salida.

8.1.1. Características cualitativas de los dispositivos de entrada/salida

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

comunica el ordenador con el mundo exterior—. Esta clasificación de interlocutores no pretende ser un dogma ni está exenta de consideraciones filosóficas. Según ella es evidente que una interfaz de un computador con las terminaciones nerviosas de un ratón en un experimento de bioingeniería no tiene interlocutor humano, pero ¿qué diríamos entonces si las terminaciones nerviosas fueran de una persona?

En resumen, las características 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 la información entre él y el ordenador.

8.1.2. Características temporales de los dispositivos de entrada/salida

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

Además, la tasa de transferencia, considerada sin más, no describe correctamente el comportamiento temporal de los dispositivos, pudiendo ser medida de distintas formas, todas ellas correctas aunque no igualmente significativas. Veamos un par de ejemplos que permiten caracterizar mejor la tasa de transferencia. Comparando la reproducción de una

película en alta definición almacenada en un disco duro con la pulsación de un teclado, es evidente que la primera actividad requiere mayor tasa de transferencia. Sin embargo, nuestra experiencia al disfrutar de la película no se verá mermada si, desde que ejecutamos el programa de reproducción hasta que aparecen las primeras imágenes transcurren diez o quince segundos. Sería imposible, por otra parte, trabajar con un ordenador si cada vez que pulsamos una tecla transcurrieran varios segundos —no ya diez, simplemente uno o dos— hasta que dicha pulsación se hace evidente en la respuesta del sistema. Estos ejemplos revelan los dos factores que caracterizan el comportamiento temporal de los dispositivos de entrada/salida:

- La **latencia**, que se entiende como el tiempo transcurrido desde que se inicia una operación de entrada/salida hasta que el primer dato comunicado llega a su destino. En una operación de entrada sería el tiempo transcurrido desde que se inicia la petición de datos hasta que se recibe el primero de ellos. Un teclado, con una baja tasa de transferencia, requiere sin embargo una latencia de decenas de milisegundos para funcionar adecuadamente.
- La **productividad**, que se refiere a la cantidad de datos transferidos por unidad de tiempo, y que coincide con la primera definición que hemos dado de tasa de transferencia.

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

Ejemplo: Un procesador realiza una petición de un bloque de datos de 448 bytes a un dispositivo. 20 ms después 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 tasa de transferencia máxima y la productividad total?

Solución: La latencia, medida como el tiempo transcurrido desde que se inicia la operación hasta que se comienzan a recibir los datos, es de 20 ms. La tasa de transferencia máxima podemos calcularla sabiendo que cada bloque de 64 bytes se recibe en 2 ms, por lo que es de 32000 B/s, o 256000 b/s. La productividad se calcula dividiendo el total de datos transferidos entre la duración del acceso. Para calcular el tiempo hemos de tener en cuenta:

- que el primero de los bloques tarda 20 ms en llegar;
- que se reciben 7 bloques (448/64 = 7), en 2 ms cada uno de ellos;
- y que entre bloque y bloque transcurren 5 ms (seis veces, por tanto).

Así pues el tiempo total es de:

$$20 + 7 * 2 + 6 * 5 = 64 \text{ ms}$$

Como se reciben 448 bytes, la productividad es de:

$$448 \text{ B}/64 \text{ ms} = 7000 \text{ B/s, o } 56000 \text{ b/s}$$

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

capaces de gestionar eficazmente esta complejidad. En este libro, en los siguientes capítulos, 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.

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 es complicada e incluye 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 vamos a describir ambas estructuras.

8.2.1. Estructura física de los dispositivos de entrada/salida

Como parte constituyente del mismo, todo dispositivo acaba relacionándose con el ordenador, por lo que dispone de circuitos electrónicos digitales de la misma tecnología. En el otro extremo, el dispositivo es capaz de generar luz, mover una rueda, medir la salinidad del agua o registrar los desplazamientos producidos en una palanca. Buena parte de esta estructura, pero no toda, es electrónica. Es sin embargo posible encontrar una estructura general a la que, como siempre con excepciones, se adaptan de una u otra forma todos los dispositivos de entrada/salida. Esta configuración incluye tres tipos de tecnología, que enumeradas desde el ordenador hacia el exterior serían las siguientes (véase la Figura 8.1):

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

entrada/salida en el ordenador, que iremos describiendo a lo largo de este documento.

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

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

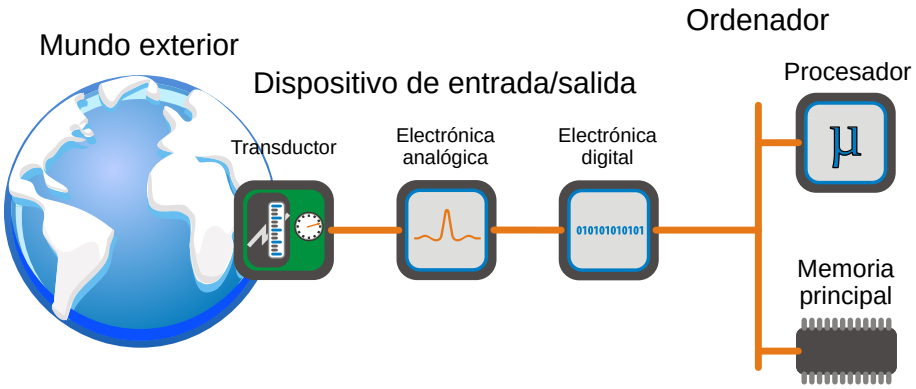


Figura 8.1: Estructura de un dispositivo de entrada/salida

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

En el caso de un teclado comenzaríamos con las partes mecánicas de las teclas —incluyendo resortes y membranas, según el tipo— y los contactos eléctricos que completan los transductores. La electrónica analógica estaría formada por resistencias para adaptar los niveles eléctricos y diodos para evitar corrientes inversas. La parte digital, en los teclados

más corrientes, la forma un microcontrolador que gestiona el teclado y encapsula la información de las teclas pulsadas en un formato estandarizado que se envía a través de un bus de entrada/salida estándar —USB hoy en día; antes PS/2 o el bus de teclado de los primeros PC—.

8.2.2. Arquitectura de los controladores de dispositivos

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

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

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

- **Registros de control**, que se utilizan para que el procesador configure parámetros en el dispositivo o le indique las operaciones de entrada/salida que debe realizar. Son registros en los que puede escribir el procesador, pero no el dispositivo.

- **Registros de estado**, que permiten al dispositivo mantener información acerca de su estado y del estado de las operaciones de entrada/salida que va realizando. Son registros que escribe el dispositivo y puede leer el procesador.
- **Registros de datos**, que sirven para realizar el intercambio de datos entre el procesador y el dispositivo en las operaciones de entrada/salida. En el caso de salida, el procesador escribirá los datos que el periférico se encargará de llevar al mundo exterior. En el caso de entrada, el periférico escribirá los datos en estos registros, que de este modo serán accesibles para el procesador mediante lecturas.

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

Aunque la clasificación de los registros y sus características, tal y como se han presentado, son correctas desde un punto de vista teórico, es frecuente que en los controladores reales, para simplificar los circuitos y la gestión, se mezcle información de control y estado en un mismo registro lógico —es decir, un único registro desde el punto de vista del procesador— e incluso que un bit tenga doble uso, de control y estado, según el momento. Un ejemplo común en los conversores analógico-digitales es disponer de un bit de control que escribe el procesador para iniciar la conversión —poniéndolo a `1`, por ejemplo— y que el dispositivo cambia de valor —a `0` en este ejemplo— cuando ha terminado —típica información de estado— y el resultado está disponible en un registro de datos.

8.2.3. Acceso a los registros de los controladores

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

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

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

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

En este apartado hemos visto la estructura de los dispositivos de entrada/salida, que normalmente incluye un bloque de tecnología específica para interactuar con el mundo exterior, otro electrónico analógico que se relaciona con el anterior mediante transductores, y un bloque de electrónica digital, de la misma naturaleza que el resto de circuitos del ordenador. Este último bloque se llama *controlador del dispositivo* y facilita que el procesador se comunique con aquél mediante registros de control para enviar órdenes y configuraciones, registros de estado para comprobar el resultado de las operaciones y los posibles errores, y registros de datos para intercambiar información. Estos registros pueden ser accesibles en el mapa de memoria del procesador, mediante instrucciones de acceso a memoria, o en un mapa específico de entrada/salida que

solo puede darse si el procesador incorpora instrucciones especiales. En los siguientes capítulos se verá cómo se usan todos estos registros para relacionar el procesador con los 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, baja o media; para la tasa de transferencia basta con una estimación razonable. Busca información sobre aquéllos que no conozcas.

8.1.1 Una pantalla táctil de 1024 x 768 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 que se usará en las prácticas. Consúltala 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 (*Real-Time Timer 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 16 bits, de los que solo utiliza 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 1024 x 768 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.

Capítulo 9

Dispositivos de Entrada/Salida

Índice

En el capítulo anterior se presentó la problemática y características de la entrada/salida en los ordenadores, así como la estructura tanto física como lógica de los dispositivos. El siguiente paso para estudiar los sistemas de entrada/salida es conocer los detalles de algunos de estos dispositivos y disponer de un entorno para poder realizar programas que interactúen con ellos. En este capítulo se van a describir las generalidades de los dos grupos de dispositivos más comúnmente usados: la entrada/salida de propósito general y los dispositivos de gestión del tiempo. Todos los sistemas incluyen y utilizan estos tipos de dispositivos, que son fundamentales en los sistemas empotrados y en los microcontroladores. Para poder trabajar con aplicaciones reales, este capítulo también describe el entorno de desarrollo que se va a utilizar para realizar programas de entrada/salida: la tarjeta *Arduino Due* y su entorno de desarrollo, junto con una sencilla tarjeta de expansión que añade al sistema un LED RGB y un pulsador.

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

La forma más sencilla de entrada/salida que podemos encontrar en un procesador son sus propios pines de conexión eléctrica con el exterior. Si la organización del procesador permite relacionar direcciones del mapa de memoria o de entrada salida con algunos pines, la escritura de un `1` o `0` lógicos por parte de un programa —arquitectura— en esas direcciones se reflejará en cierta tensión eléctrica en el pin, normalmente 0V para el nivel bajo y 5 o 3,3V para el alto, que puede ser utilizada para activar o desactivar algún dispositivo externo. Por ejemplo, esto nos permitiría encender o apagar un LED mediante instrucciones de nuestro programa. De modo análogo, en el caso de las entradas, si el valor eléctrico presente en el pin se ve traducido por el diseño eléctrico del circuito en un `1` o `0` lógico que se puede leer en una dirección del sistema, podremos detectar cambios en el exterior de nuestro procesador. De esta manera, por ejemplo, nuestro programa podrá consultar si un pulsador está libre u oprimido, y tomar decisiones en función de su estado.

Veámoslo en un sencillo ejemplo:

09_entrada_salida.s

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

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

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

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

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

En el ejemplo que hemos visto antes se trabajaba exclusivamente con un bit, que se corresponde con un pin del circuito integrado, tanto en entrada como en salida, utilizando instrucciones de acceso a una palabra de memoria, 32 bits en la arquitectura ARM. En la mayor parte de los sistemas, los diversos pines de entrada/salida se agrupan en *palabras*, de tal forma que cada acceso como los del ejemplo afectaría a todos los pines asociados a la palabra a cuya dirección se accede. De esta manera, se habla de *puertos* refiriéndose a cada una de las direcciones asociadas a conjuntos de pines en el exterior del circuito, y cada pin individual es un bit del puerto. Así por ejemplo, si hablamos de *PB12* —en el microcontrolador ATSAM3X8E— nos estamos refiriendo al bit 12 del puerto de salida llamado PB —que físicamente se corresponde con el pin 86 del encapsulado LQFP del microcontrolador, algo que es necesario saber para diseñar el hardware del sistema—. En este caso, para actuar —modificar o comprobar su valor— sobre bits individuales o sobre conjuntos de bits es necesario utilizar máscaras y operaciones lógicas para no afectar a otros pines del mismo puerto. Suponiendo que el LED se encuentra en el bit 12 y el pulsador en el bit 20 del citado puerto PB, una versión más verosímil del ejemplo propuesto sería:

09_acceso_es.s

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

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

Obviando esta complejidad, el ejemplo que se acaba de presentar es válido para mostrar la gestión por programa de la entrada y salida tipo

GPIO. Sin embargo, es necesario que nos surja alguna duda al considerar —no lo olvidemos— que los pines se relacionan realmente con el exterior mediante magnitudes eléctricas. Efectivamente, el comportamiento eléctrico de un pin que funciona como entrada es totalmente distinto al de otro que se utiliza como salida, como veremos más adelante. Para resolver esta paradoja volvemos a hacer hincapié en que el ejemplo que se ha comentado es de gestión de la entrada/salida durante el funcionamiento del sistema, pero no se ha querido comentar, hasta ahora, que previamente hace falta una configuración de los puertos de la GPIO en que se indique qué pines van a actuar como entrada y cuáles como salida. Así pues, asociado a la dirección en la que se leen o escriben los datos y que hemos llamado *PB* en el ejemplo, habrá al menos otra que corresponda a un registro de control de la GPIO en la que se indique qué pines se comportan como entradas y cuáles como salidas, lo que se conoce como *dirección de los pines*. Consideremos de nuevo el hecho diferencial de utilizar un pin —y su correspondiente bit en un puerto— como entrada o como salida. En el primer caso son los circuitos exteriores al procesador los que determinan la tensión presente en el pin, y la variación de ésta no depende del programa, ni en valor ni en tiempo. Sin embargo, cuando el pin se usa como salida, es el procesador ejecutando instrucciones de un programa el que modifica la tensión presente en el pin al escribir en su bit asociado. Se espera además —pensemos en el LED encendido— que el valor se mantenga en el pin hasta que el programa decida cambiarlo escribiendo en él otro valor. Si analizamos ambos casos desde el punto de vista de necesidad de almacenamiento de información, veremos que en el caso de la entrada nos limitamos a leer un valor eléctrico en cierto instante, valor que además viene establecido desde fuera y no por el programa ni el procesador, mientras que en la salida es necesario asociar a cada pin un espacio de almacenamiento para que el `1` o `0` escrito por el programa se mantenga hasta que decidamos cambiarlo, de nuevo de acuerdo con el programa. Esto muestra por qué la GPIO a veces utiliza dos puertos, con direcciones distintas, para leer o escribir en los pines del sistema. El registro —o *latch*— que se asocia a las salidas suele tener una dirección y las entradas —que no requieren registro pues leen el valor lógico fijado externamente en el pin— otra. Así, en el caso más común, un puerto GPIO ocupa al menos tres direcciones en el mapa: una para el registro de control que configura la dirección de los pines, otra para el registro de datos de salida y otra para leer directamente los pines a través del registro de datos de entrada. En la Figura 9.1 (obtenida del manual [Atm11]) se muestra la estructura interna de un pin de E/S de un microcontrolador de la familia Atmel AVR.

En este caso, que como hemos dicho es muy común, ¿qué ocurre si escribimos en un pin configurado como entrada, o si leemos un pin configurado como salida? Esto depende en gran medida del diseño electrónico

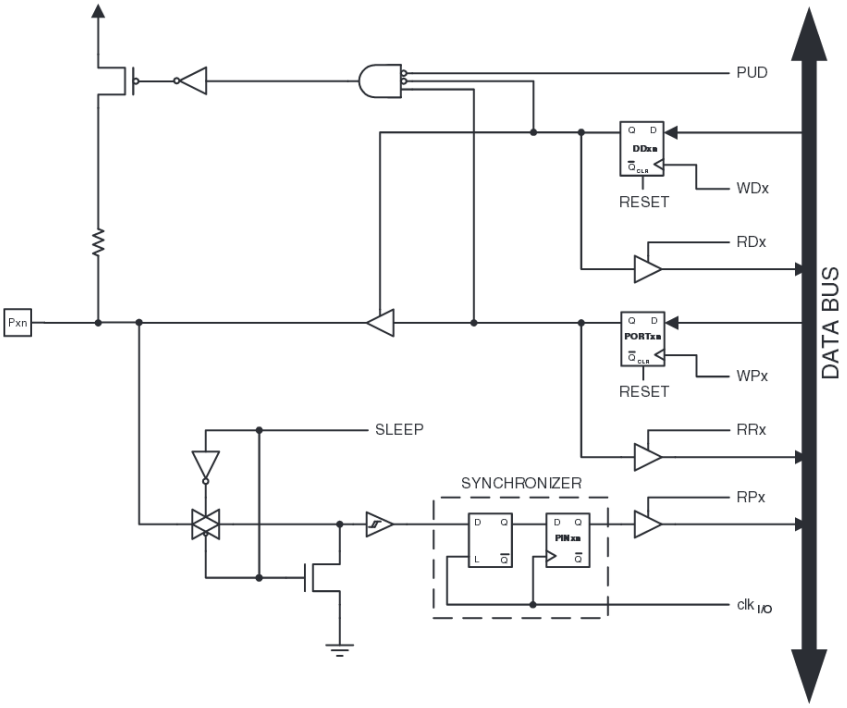


Figura 9.1: Estructura interna de un pin de E/S de un microcontrolador de la familia Atmel AVR

de los circuitos de E/S del microcontrolador, pero en muchos casos el comportamiento es el siguiente: si leemos un pin configurado como salida podemos leer bien el valor almacenado en el registro, bien el valor presente en el pin. Ambos deberían coincidir a nivel lógico, salvo que algún error en el diseño del circuito o alguna avería produjeran lo contrario. Por ejemplo, en un pin conectado a masa es imposible que se mantenga un `1` lógico. Por otra parte, si escribimos en un pin configurado como entrada es común que, en realidad, se escriba en el registro de salida, sin modificar el valor en el pin. Este comportamiento es útil, dado que permite fijar un valor lógico conocido en un pin, antes de configurarlo como salida. Dado que las entradas son eléctricamente más seguras, los pines suelen estar configurados como tales tras el reinicio del sistema. Así, el procedimiento normal para inicializar un pin de salida es escribir su valor mientras está configurado como entrada, y luego configurarlo como salida. Esto permite además fijar valores lógicos en el exterior mediante resistencias, que si son de valor elevado permitirán posteriormente el funcionamiento normal del pin como salida.

Para comprender adecuadamente esta última afirmación, vamos a

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

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

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

20mA, lo requerido para encender con brillo suficiente un LED—. Así pues la especificación de los valores de tensión de las salidas se garantiza siempre y cuando la corriente que circule por el pin no supere el valor máximo correspondiente.

La naturaleza y el comportamiento de las entradas son radicalmente distintos, aunque se defina para ellas un conjunto similar de parámetros. Mediante un puerto de entrada queremos leer un valor lógico que se relacione con la tensión presente en el pin, fijada por algún sistema eléctrico exterior. Así pues, la misión de la circuitería del pin configurado como entrada es detectar niveles de tensión del exterior, con la menor influencia en ellos que sea posible. De este modo, una entrada aparece para un circuito externo como una resistencia muy elevada, lo que se llama una *alta impedancia*. Dado que es un circuito activo y no una resistencia, el valor que se especifica es la intensidad máxima que circula entre el exterior y el circuito integrado, que suele ser despreciable en la mayor parte de los casos —del orden de pocos microamperios o menor—. Los valores especificados son `IIHMAX` e `IILMAX` o simplemente `IIMAX` si son iguales. En este caso la 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 ser entendidas como `0` o `1` al leer el pin de entrada. Para ello se especifican `VIHMIN` y `VILMAX` como la mínima tensión de entrada que se lee como un `1` lógico y la máxima que se lee como un `0`, respectivamente. En ambos casos, por diseño del microcontrolador, se sabe que la corriente de entrada está limitada, independientemente del circuito externo.

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

Hagamos un pequeño estudio de los circuitos eléctricos relacionados con los dispositivos de nuestro ejemplo, el LED y el pulsador. Comencemos como viene siendo habitual por el circuito de salida. En la Figura 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

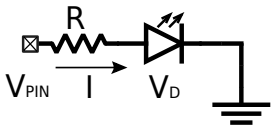


Figura 9.2: Conexión de un LED a un pin de E/S de un microcontrolador

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

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

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

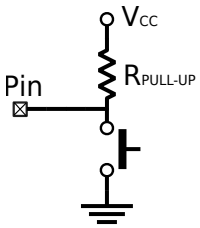


Figura 9.3: Conexión de un pulsador a un pin de E/S de un microcontrolador

Veamos ahora cómo conectar un pulsador a una entrada del circuito. En la Figura 9.3 se muestra el circuito esquemático de un pulsador conectado a un pin de E/S de un microcontrolador. Un pulsador no es más que una placa de metal que se apoya o se separa de dos conectores, permitiendo o no el contacto eléctrico entre ellos. Es, pues, un dispositivo electromecánico que no genera de por sí ninguna magnitud eléctrica. Para ello, hay que conectarlo en un circuito y, en nuestro caso, en uno que genere las tensiones adecuadas. Para seguir estrictamente los ejemplos, podemos pensar que el pulsador hace contacto entre los 5V de la

alimentación y el pin. De este modo, al pulsarlo, conectaremos el pin a la alimentación y leeremos en él un `1`, tal y como se ha considerado en el código de ejemplo. Sin embargo, si no está pulsado, el pin no está conectado a nada por lo que el valor presente en él sería, en general, indefinido. Por ello el montaje correcto requiere que el pin se conecte a otro nivel de tensión, masa —0V— en este caso, a través de una resistencia para limitar la corriente al pulsar. Como la corriente de entrada en el pin es despreciable, el valor de la resistencia no es crítico, siendo lo habitual usar decenas o cientos de KΩ. Según este circuito y siguiendo el ejemplo, al pulsar leeríamos un nivel alto y en caso de no pulsar, un `0` lógico.

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

9.1.2. Interrupciones asociadas a la GPIO

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

En primer lugar se pueden utilizar los pines como líneas de interrupción, bien para señalar un cambio relativo al circuito conectado al pin, como oprimir un pulsador, bien para conectar una señal de un circuito externo y que así el circuito sea capaz de generar interrupciones. En este último caso el pin de la GPIO haría el papel de una línea de interrupción externa de un procesador. En ambos casos suele poder configurarse si la interrupción se señala por nivel o por flanco y su polaridad. En segundo lugar, y asociado a las características de bajo consumo de los

microcontroladores, se tiene la interrupción por cambio de valor. Esta interrupción puede estar asociada a un pin o un conjunto de ellos, y se activa cada vez que alguno de los pines de entrada del grupo cambia su valor, desde la última vez que se leyó. Esta interrupción, además, suele usarse para sacar al procesador de un modo de bajo consumo y activarlo otra vez para reaccionar frente al cambio indicado.

El uso de interrupciones asociadas a la GPIO requiere añadir nuevos registros de control y estado, para configurar las interrupciones y sus características —control— y para almacenar los indicadores —*flags*— que informen sobre las circunstancias de la interrupción.

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 5V y 3,3V para el nivel alto—. También es frecuente encontrar entradas con disparador de Schmitt para generar flancos más rápidos en las señales eléctricas en el interior del circuito, por ejemplo en entradas que generen interrupciones, lo que produce que los valores `VIHMIN` y `VILMAX` en estos pines estén más próximos, reduciendo el rango de tensiones indeterminadas —a nivel lógico— entre ellos. Tenemos también salidas que pueden configurarse como colector abierto —*open drain*, en nomenclatura CMOS— lo que permite utilizarlas en sistemas AND cableados, muy utilizados en buses.

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

La última característica asociada a la GPIO que vamos a tratar surge de la necesidad de versatilidad de los microcontroladores. Los dispositi-

vos 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 entrada/salida. Anotar correctamente la fecha y hora de modificación de un archivo, arrancar automáticamente tareas con cierta periodicidad, determinar si una tecla se ha pulsado 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 entrada/salida de los ordenadores.

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

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

él, normalmente mediante proceso de tiempo real, el conjunto de dispositivos y mecanismos relacionados con el tiempo es mucho más variado e importante.

En todos los ordenadores se encuentra, al menos, un dispositivo tipo contador que se incrementa de forma periódica y permite medir intervalos de tiempo de corta duración —milisegundos o menos—. A partir de esta base de tiempos se puede organizar toda la gestión temporal del sistema, sin más que incluir los programas necesarios. Sin embargo se suele disponer de otro dispositivo que gestiona el tiempo en formato humano —formato de fecha y hora— que permite liberar de esta tarea al software del sistema y además, utilizando alimentación adicional, mantener esta información aún con el sistema apagado. Por último, para medir eventos externos muy cortos, para generar señales eléctricas con temporización precisa y elevadas frecuencias, se suelen añadir otros dispositivos que permiten generar pulsos periódicos o aislados o medir por hardware cambios eléctricos en los pines de entrada/salida.

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

El temporizador del sistema

El temporizador —*timer*— del sistema es el dispositivo más común y sencillo. Constituye la base de medida y gestión de tiempos del sistema. Se trata de un registro contador que se incrementa de forma periódica a partir de cierta señal de reloj generada por el hardware del sistema. Para que su resolución y tiempo máximo puedan configurarse según las necesidades, es habitual encontrar un divisor de frecuencia o *prescaler* que permite disminuir con un margen bastante amplio la frecuencia de incremento del contador. De esta manera, si la frecuencia final de incremento es f, se tiene que el tiempo mínimo que se puede medir es el periodo, $T = 1/f$, y el tiempo que transcurre hasta que se desborde el contador $2^n \cdot T$, siendo n el número de bits del registro temporizador. Para una frecuencia de 10KHz y un contador de 32 bits, el tiempo mínimo sería 100μs y transcurrirían unos 429 496s —casi cinco días— hasta que se desbordara el temporizador. El temporizador se utiliza, en su forma más simple, para medir tiempos entre dos eventos —aunque uno de ellos pueda ser el inicio del programa—. Para ello, se guarda el valor del contador al producirse el primer evento y se resta del valor que tiene

al producirse el segundo. Esta diferencia multiplicada por el periodo nos da el tiempo transcurrido entre ambos eventos.

El ejemplo comentado daría un valor incorrecto si entre las dos lecturas se ha producido más de un desbordamientos del reloj. Por ello, el temporizador activa una señal de estado que generalmente puede causar una interrupción cada vez que se desborda, volviendo a `0`. El sistema puede tratar esta información, sobre todo la interrupción generada, de distintas formas. Por una parte se puede extender el contador con variables en memoria para tener mayor rango en la cuenta de tiempos. Es habitual también utilizarla como interrupción periódica para gestión del sistema —por ejemplo, medir los tiempos de ejecución de los procesos en sistemas multitarea—. En este último caso es habitual poder recargar el contador con un valor distinto de `0` para tener un control más fino de la periodicidad de la interrupción, por ello suele ser posible escribir sobre el registro que hace de contador.

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

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 debe elegir entre tener una medida de tiempos de larga duración —hasta de varios

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

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

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

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

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

El reloj en tiempo real

En un computador, el reloj en tiempo real o *RTC* (*Real-time Clock*) es un circuito específico encargado de mantener la fecha y hora actuales incluso cuando el computador está desconectado de la alimentación

eléctrica. Es por este motivo que suele estar asociado a una batería o a un condensador que le proporciona la energía necesaria para seguir funcionando cuando se interrumpe la alimentación.

Habitualmente este periférico emplea como frecuencia base una señal de 32 768 Hz, es decir, una señal cuadrada que completa 32 768 veces un ciclo OFF-ON cada segundo. Esta frecuencia es la empleada habitualmente por los relojes de cuarzo, dado que coincide con 2^{15} ciclos por segundo, con lo cual el bit de peso 15 del contador de ciclos cambia de valor exactamente una vez por segundo y puede usarse como señal de activación del segundero en el caso de un reloj analógico o del contador de segundos en uno digital.

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

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

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, en Italia, desarrollaron una plataforma de hardware libre basada en un microcontrolador y un entorno de desarrollo diseñados para facilitar la realización de proyectos de electrónica. Banzi y su grupo se reunían habitualmente en el Bar del Rey Arduino, en la localidad de Ivrea, de ahí el nombre del sistema.

Arduino está compuesto por una plataforma de hardware libre y un entorno de desarrollo. A grandes rasgos, esto significa que el diseño está a disposición de quien lo quiera emplear y modificar, dentro de unos límites de beneficio económico y siempre publicando las modificaciones introducidas.

Existen diferentes versiones de la arquitectura Arduino que emplean diversos microcontroladores respetando las dimensiones físicas de los

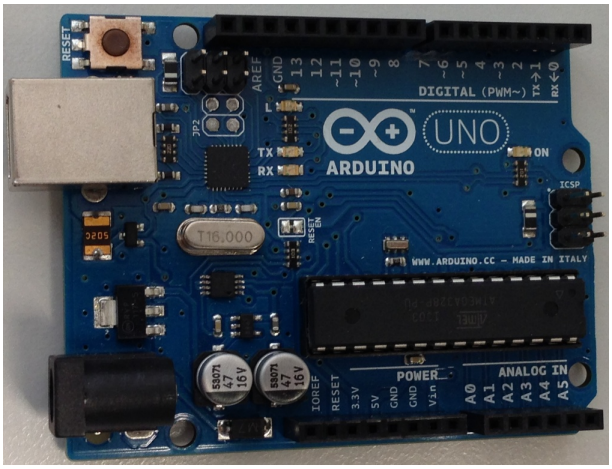


Figura 9.4: Tarjeta Arduino Uno

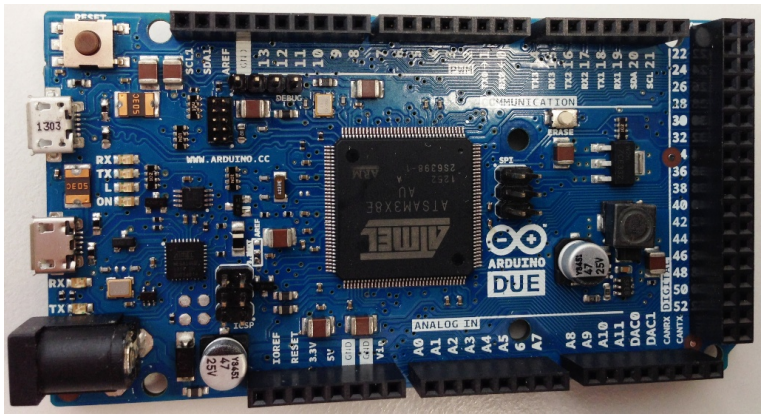


Figura 9.5: Tarjeta Arduino Due

conectores de ampliación y la ubicación de las señales en los mismos. El entorno de desarrollo introduce una capa de abstracción mediante la cual un conjunto de comandos y funciones puede ser empleado en cualquier plataforma Arduino.

En nuestro caso, usaremos la versión *Arduino Due* —véase la Figura 9.5— que, respecto de la tarjeta Arduino Uno original —mostrada en la Figura 9.4—, entre otras diferencias, presenta un microprocesador ATSAM3X8E, más potente que el ATmega328 de aquella y con mayor número de entradas/salidas.

En la primera parte de las prácticas se ha utilizado el conjunto de instrucciones *Thumb* correspondiente a la versión Cortex-M0 de la ar-

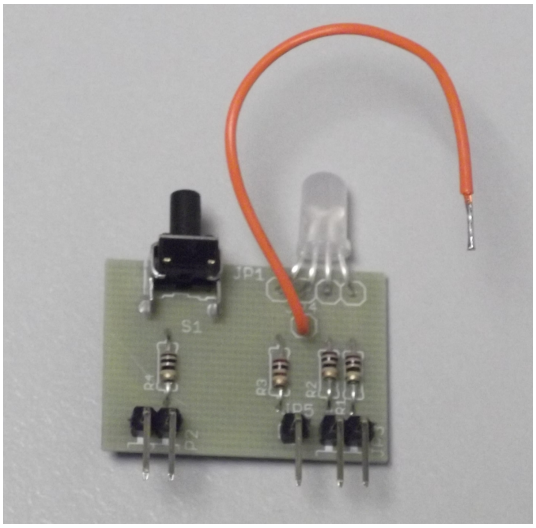


Figura 9.6: Tarjeta de E/S de prácticas de laboratorio

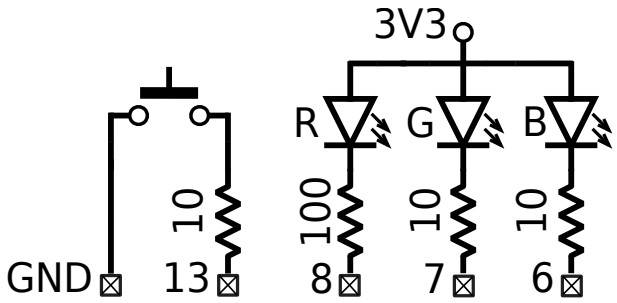


Figura 9.7: Esquema de la tarjeta de E/S de prácticas de laboratorio

quitectura ARM. El microcontrolador ATMSAM3X8E de la tarjeta Arduino Due implementa la versión Cortex-M3 que utiliza un conjunto de instrucciones mayor, el *Thumb II*. Aunque todas las instrucciones *Thumb* están incluidas en *Thumb II* existe una diferencia crítica en el lenguaje ensamblador, que se señala aquí porque puede dar lugar a errores.

Las instrucciones aritméticas y lógicas del conjunto *Thumb II* pueden decidir modificar o no los indicadores de estado —*flags*—. 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
```

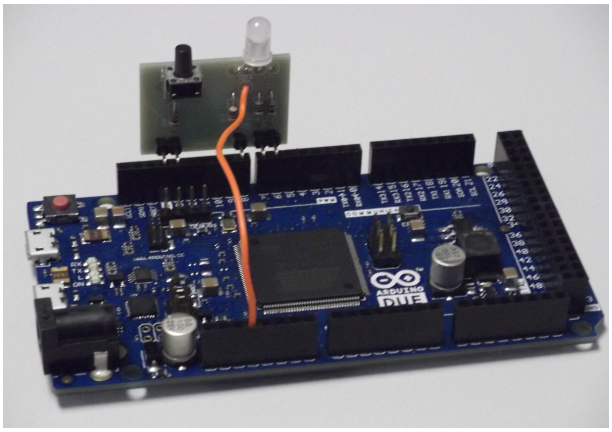


Figura 9.8: Tarjeta de E/S insertada en la Arduino Due

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 el caso del juego de instrucciones *Thumb*, todas las instrucciones afectan a los indicadores de estado.

Puesto que el juego de instrucciones *Thumb II* es más potente que el visto hasta ahora, es recomendable consultar el manual *CortexM3 Instruction Set* [Tex10] para conocer todas las posibilidades de este conjunto de instrucciones.

Para el desarrollo de las prácticas, se ha confeccionado una tarjeta específica —mostrada en la Figura 9.6— que contiene un pulsador y un LED RGB conectados como muestra la Figura 9.7 que se emplearán como dispositivos de E/S. La Figura 9.8 muestra la tarjeta instalada sobre la Arduino Due, donde puede apreciarse que, además de insertar la tarjeta correctamente, hay que conectar un cable de alimentación al pin de la Arduino Due rotulado con el texto `3.3V`. Como se puede ver en la Figura 9.7, el LED RGB es del tipo ánodo común, por lo que será necesario escribir un `0` en cada salida conectada a un LED para encenderlo y un `1` para mantenerlo apagado. Igualmente se puede ver cómo se conecta el pulsador a un pin y a masa. Se infiere, pues, que será necesario activar el `pull-up` de la salida 13 para leer y que en el estado no pulsado se obtendrá un `1` en el pin. Al pulsarlo, lógicamente, cambiará a `0`.

El entorno de programación de Arduino —véase la Figura 9.9— se presenta como un editor de código en el cual podemos escribir nuestro programa, guardarlo, compilarlo y subirlo al dispositivo Arduino que tengamos conectado. La versión del entorno de Arduino que se va a usar en las prácticas ha sido modificada para permitir el uso de archivos

fuente en ensamblador. Las instrucciones para instalar esta versión se pueden consultar en el sitio web del libro.

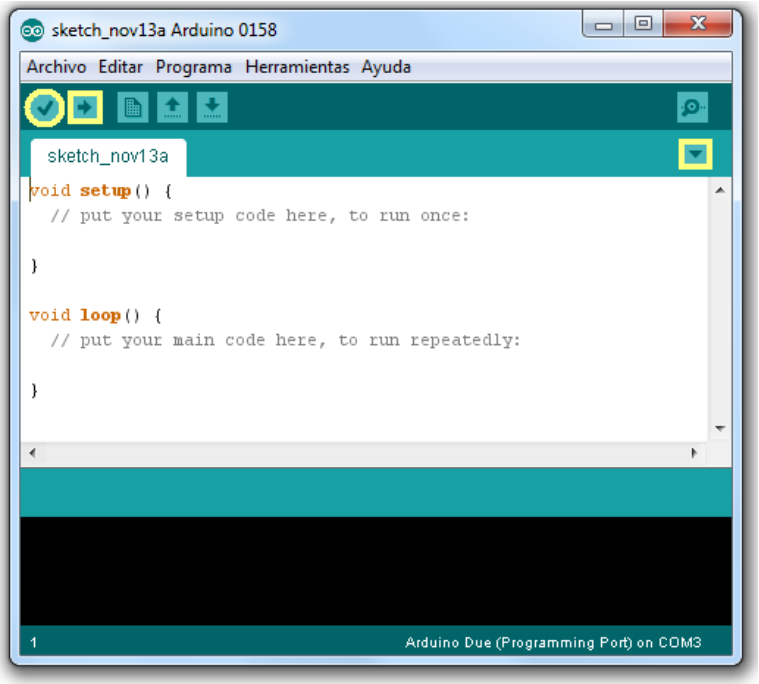


Figura 9.9: Entorno de programación Arduino

La estructura de un programa típico de Arduino consta de dos funciones:

- «`void setup()`»: Contiene el conjunto de acciones que se realizarán al inicio de nuestro programa. Aquí habitualmente configuraremos las entradas/salidas que vayamos emplear y daremos valores iniciales a las variables.
- «`void loop()`»: Contiene el código que se ejecutará indefinidamente.

Es necesario especificar al entorno el modelo de sistema Arduino que tenemos conectado para que se usen las bibliotecas adecuadas y se asignen correctamente las señales de E/S, se acceda correctamente a los registros del procesador, etcétera. Para ello emplearemos la entrada «`Placa`» dentro del menú «`Herramientas`». En nuestro caso seleccionaremos la opción «`Arduino Due (Programming Port)`» —véase la Figura 9.10 para Windows y la Figura 9.11 para GNU/Linux—. Los dos

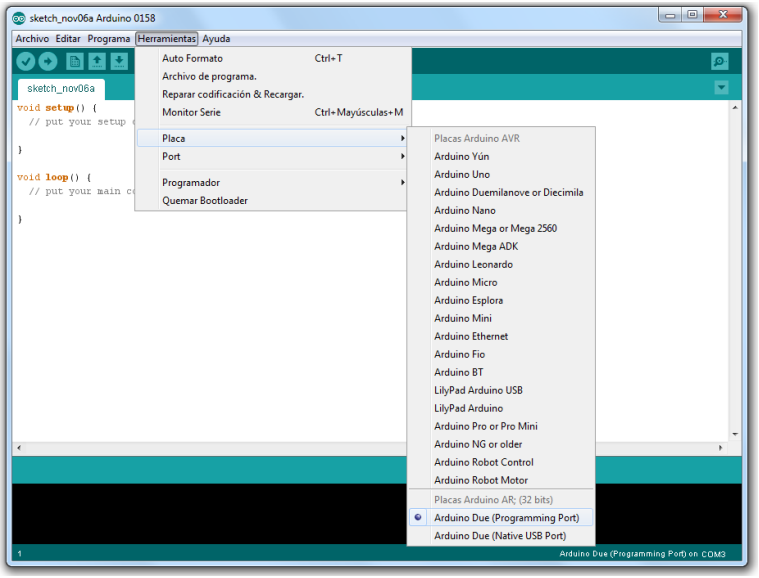


Figura 9.10: Selección del sistema Arduino a emplear en Windows

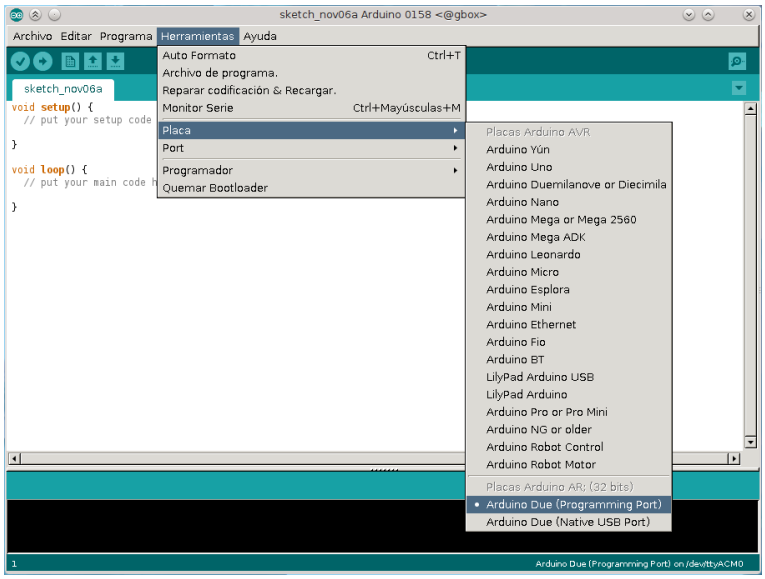


Figura 9.11: Selección del sistema Arduino a emplear en GNU/Linux

puertos USB de la tarjeta Arduino Due están identificados en la cara de soldaduras de la misma.

De la misma forma, hay que indicar el puerto serie de comunicaciones

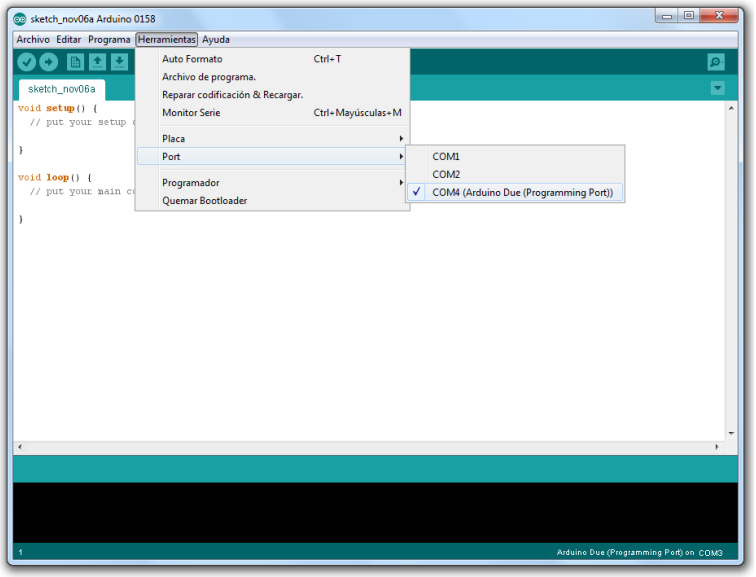


Figura 9.12: Selección del puerto de comunicaciones en Windows

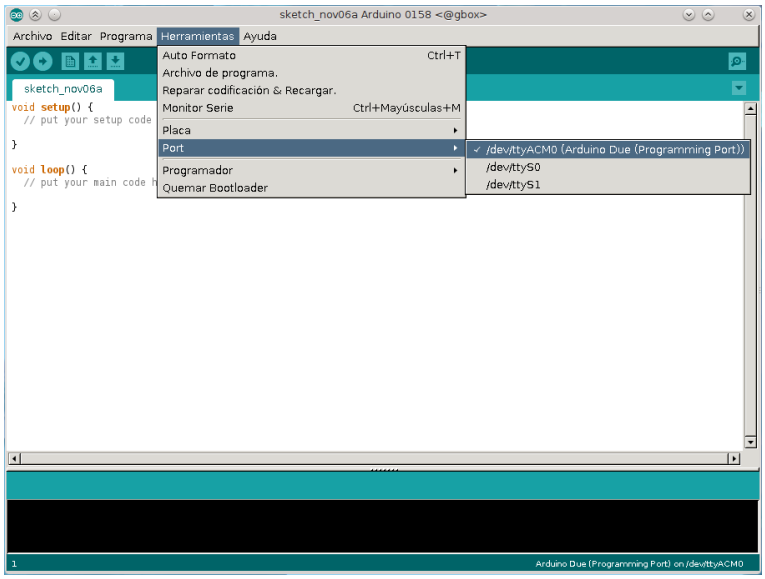


Figura 9.13: Selección del puerto de comunicaciones en GNU/Linux

en que está conectado el sistema Arduino, lo cual especificaremos mediante la entrada «`Port`» dentro del menú «`Herramientas`», como muestran la Figura 9.12 para Windows y la Figura 9.13 para GNU/Linux.

Como opciones se nos ofrecerán los puertos serie —`COMx` o `/dev/ttyx` en Windows y GNU/Linux, respectivamente— de que disponga el sistema. Tanto en Windows como en GNU/Linux aparece como opción un puerto —que es donde se encontrará conectada nuestra tarjeta Arduino Due— junto a cuyo nombre aparecerá el texto *(Arduino Due (Programming Port))* y es, por tanto, el que debemos seleccionar.

9.4. Creación de proyectos

El entorno Arduino posee una estructura denominada *proyecto* —*sketch* en la bibliografía Arduino— que contiene los archivos correspondientes a cada programa. Cuando se inicia el entorno, se nos muestra un proyecto vacío con un nombre del tipo «`sketch_mmmdda`» —donde «`mmm`» es la abreviatura del nombre del mes actual y «`dd`» es el día del mes— que podemos usar como base para desarrollar un nuevo programa. De la misma forma, mediante la entrada «`Abrir...`» dentro del menú «`Archivo`» podemos abrir un proyecto existente.

Los archivos que componen un proyecto se guardan en una carpeta cuyo nombre coincide con el del proyecto. Generalmente un proyecto consta de un solo archivo de extensión «`.ino`» —con el mismo nombre que la carpeta— que contiene el código en lenguaje «`C / C++`» del programa principal.

Un programa puede, sin embargo, constar de más de un archivo. En tal caso, para añadir archivos al proyecto emplearemos el botón del entorno marcado con un recuadro en la esquina superior derecha en la Figura 9.9, que desplegará un menú del cual elegiremos la opción `Nueva Pestaña`. Al hacerlo, en la parte inferior del entorno aparecerá una barra en la que se nos solicitará el nombre de la nueva pestaña —y por tanto del archivo en que se guardará su contenido— donde deberemos especificar tanto el nombre como la extensión de dicho archivo y crear tanto la pestaña como el archivo pinchando en el botón `Ok`.

9.4.1. Ejemplo

Como ejemplo básico vamos a mostrar un programa que hace parpadear el LED incorporado en la tarjeta Arduino. De momento no es necesario que la tarjeta esté conectada al computador, con lo que esta fase se puede llevar a cabo sin necesidad de poseer una tarjeta Arduino. El código es el mostrado en la Figura 9.14.

Una vez introducido el código como texto, podemos comprobar que es correcto —lo cual implica *compilarlo*— mediante el botón de la parte izquierda de la barra de botones —marcado con un círculo en la Figura 9.9 y que hace aparecer el texto `Verificar` cuando situamos el

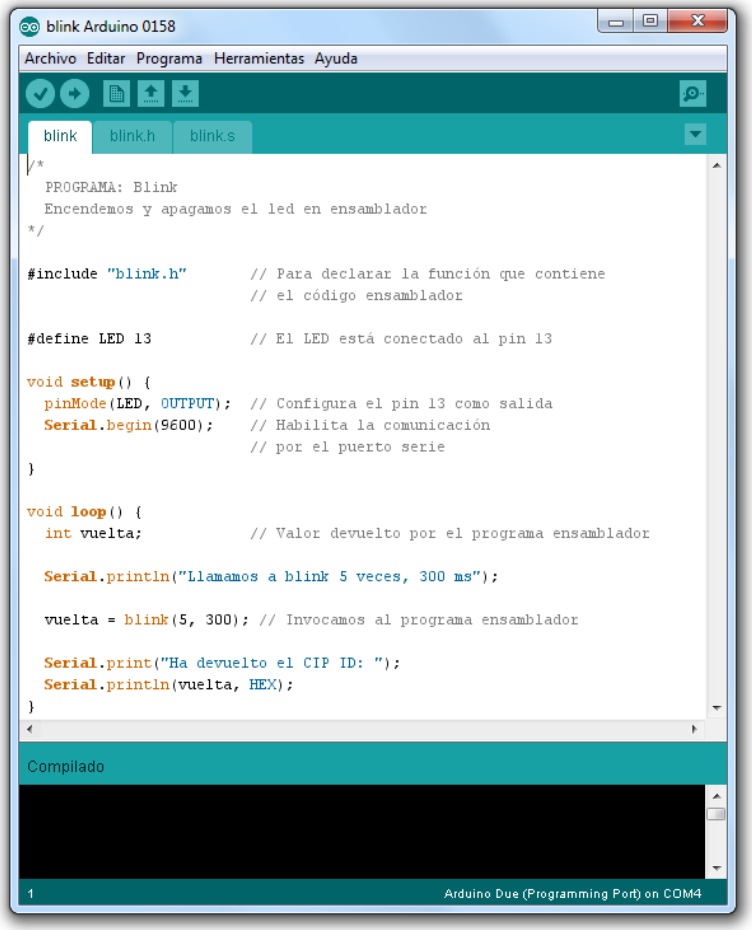


Figura 9.14: Entorno Arduino con el programa «blink» cargado

cursor sobre él—. Tras el proceso de compilación se mostrarán los posibles errores detectados indicando el número de línea en que se encuentra cada uno de ellos o, si no hay errores, la cantidad de memoria ocupada por el código generado y el máximo de que dispone la tarjeta Arduino seleccionada actualmente, como puede apreciarse en la Figura 9.15.

Mediante el segundo botón de la barra de botones —marcado con un cuadrado en la Figura 9.9 y que hace aparecer el texto `Subir` (en Windows) o `Cargar` (en GNU/Linux) cuando situamos el cursor sobre él— se desencadena el mismo proceso pero, si en la compilación no se han producido errores, el código generado es enviado a la tarjeta Arduino —que ahora sí debe estar conectada al computador— y ejecutado de inmediato. De hecho, la programación se verifica comunicando el ordenador con la tarjeta y forzando a que se ejecute un programa especial llama-

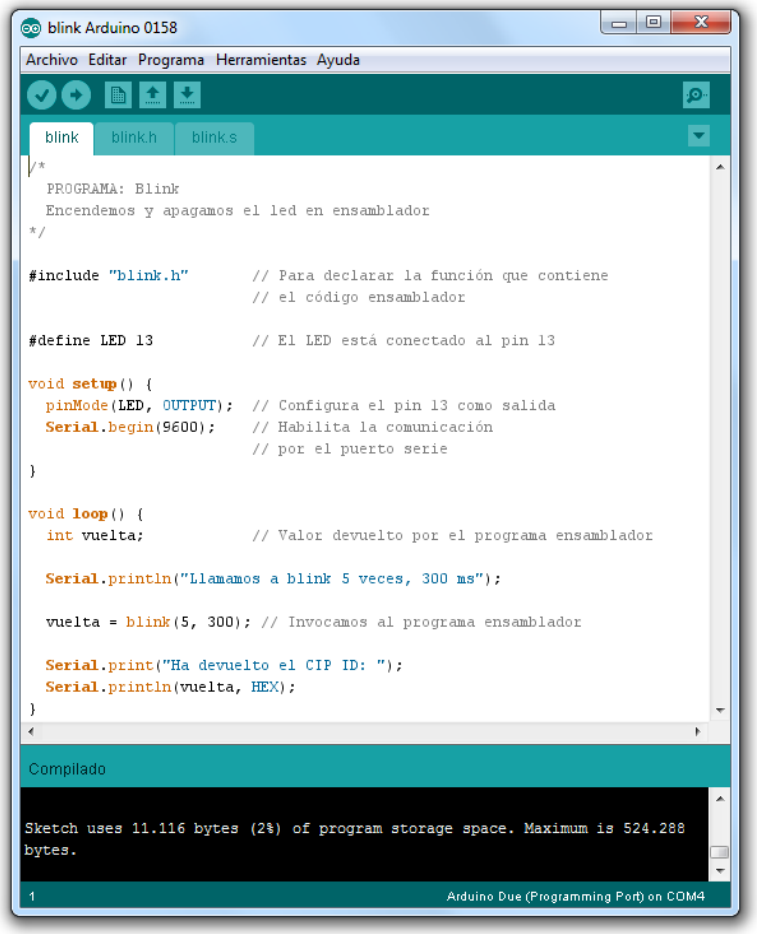


Figura 9.15: Resultado de la compilación del programa «blink»

do *bootloader*. Este programa lee del puerto USB-serie las instrucciones a cargar en la ROM. Una vez terminada la comunicación, el ordenador fuerza un RESET y el microcontrolador comienza a ejecutar el programa descargado.

Siguiendo el procedimiento descrito se puede programar el microcontrolador empleando el lenguaje de alto nivel `C / C++` y las funciones específicas de Arduino. Nuestro objetivo, sin embargo, es programar el microcontrolador empleando directamente su lenguaje ensamblador y para conseguirlo vamos a introducir unas ligeras modificaciones en el código.

En primer lugar, escribiremos nuestro programa ensamblador en un archivo con la extensión `.s` para identificarlo como código ensamblador. En el caso del programa `blink` que hace parpadear el LED de Arduino,

el código ensamblador correspondiente será:

blink.s

```
# blink.s - Parpadeo en ensamblador
# Acceso al controlador del PIO B

.syntax unified
.cpu cortex-m3
.text
.align  2
.thumb
.thumb_func
.extern delay                   @ No es necesario
.global blink                   @ Función externa
.type blink, %function

.equ PIOB,      0x400E1000      @ Dir. base del puerto B
.equ SODR,      0x030           @ OFFSET Set Output Data Reg
.equ CODR,      0x034           @ OFFSET Clear Output Data Reg
.equ CHIPID,    0x400E0940      @ Registro CHIP ID
.equ LEDMSK,    0x08000000      @ El LED está en el pin 27

/* int blink(int times, int delay)
     r0 = times. Número de veces que parpadea
     r1 = delay. Retardo del parpadeo
     Devuelve el CHIP_ID, porque sí
     Los parámetros se pasan en r0-r3
     El valor devuelto en r0 ó r0-r1 si ocupa 8 bytes
     Cualquier función puede modificar r0-r3
     El resto se han de preservar */

blink:
   push         {r4-r7, lr}     @ Vamos a usar de r4 a r7
                                @ porque llamamos a delay
   mov          r4, r0          @ r4 contiene el número de veces
   mov          r5, r1          @ r5 contiene el retardo a pasar a delay
   ldr          r6, =PIOB       @ Dirección base del Controlador PIO B
   ldr          r7, =LEDMSK     @ Máscara con el bit 27 a 1 (pin del LED)
principio:
   str          r7, [r6, #SODR] @ Encendemos el LED escribiendo en SET
   mov          r0, r5          @ Preparamos el parámetro de delay en r0
   bl           delay           @ Invocamos a la función delay
   str          r7, [r6, #CODR] @ Apagamos el LED escribiendo en CLEAR
   mov          r0, r5          @ Volvemos a llamar a delay como antes
   bl           delay
   subs         r4, r4, #1      @ Decrementamos el número de veces
   bne          principio       @ y si no es cero seguimos. ¡Ojo a la s!
   ldr          r6, =CHIPID     @ Leemos CHIPID_CIDR
   ldr          r0, [r6]        @ y devolvemos el valor en r0
```

```
47    pop          {r4-r7, pc}     @ ret con pop al pc.
48 .end
```

Para poder ejecutar este código desde el entorno de programación de Arduino es necesario indicar durante el proceso de compilación que se utilizan funciones en otro módulo, y que siguen el convenio de llamada de funciones de lenguaje `C`, ligeramente distinto del de `C++`. Para ello emplearemos un fichero de cabecera con extensión `.h` que llamaremos `blink.h` y cuyo contenido será:

blink.h

```
1 // Declaración de las funciones externas
2
3 extern "C" {
4   int blink(int times, int del);
5   }
```

Este código define una función llamada `blink` que acepta dos argumentos. El primer argumento indica el número de veces que se desea que el LED parpadee y el segundo argumento el periodo de tiempo en milisegundos que el LED permanecerá encendido y apagado en cada ciclo de parpadeo, es decir, el ciclo completo tendrá una duración del doble de milisegundos que el valor de este argumento.

Finalmente, el programa principal se encarga de definir los valores iniciales de las variables y de invocar el código en ensamblador que efectivamente hará parpadear el LED.

blink.ino

```
1  /*
2    PROGRAMA: Blink
3    Encendemos y apagamos el led en ensamblador
4  */
5
6  #include "blink.h"       // Para declarar la función que
7                           // contiene el código ensamblador
8
9  #define LED 13           // El LED está conectado al pin 13
10
11 void setup() {
12   pinMode(LED, OUTPUT);  // Configura el pin 13 como salida
13   Serial.begin(9600);    // Habilita la comunicación por el puerto serie
14   int vuelta;            // Valor devuelto por el programa ensamblador
15 }
16
17 void loop() {
18
19   Serial.println("Llamamos a blink 5 veces, 300 ms");
```

```
20
21    vuelta = blink(5, 300); // Invocamos el programa ensamblador
22
23    Serial.print("Ha devuelto el CIP ID: ");
24    Serial.println(vuelta, HEX);
25  }
```

En este programa podemos, además, señalar que se ha hecho uso de la comunicación serie incorporada en la plataforma Arduino para obtener mensajes durante la ejecución del programa. Para ello hay que activar esta funcionalidad dentro de «`setup()`» mediante la llamada a la función «`Serial.begin(9600)`», donde el argumento indica la velocidad de comunicación en baudios. Posteriormente, ya dentro de la función «`loop`», se pueden enviar mensajes a través del puerto serie —asociado al USB— empleando las funciones «`Serial.print`» —muestra el texto que se le pasa como argumento y permite seguir escribiendo en la misma línea— y «`Serial.println`» —muestra el texto y pasa a la línea siguiente—. El argumento de estas funciones puede ser una cadena de caracteres entre comillas —que se mostrará textualmente— o una variable, en cuyo caso se mostrará su valor. En la página *www.arduino.cc* —o desde el propio entorno— se puede acceder a la referencia para obtener información sobre las funciones de Arduino. Para visualizar los mensajes recibidos hay que iniciar el *Monitor Serie*, lo cual se consigue pinchando sobre el botón del extremo derecho de la barra de botones —que contiene el icono de una lupa y que hace aparecer el texto `Monitor Serie` (en Windows) o `Monitor Serial` (en GNU/Linux) cuando situamos el cursor sobre él—. Hay que tener en cuenta que al iniciar el *Monitor Serie* se envía a la tarjeta Arduino una señal de *RESET*.

Identificación de las entradas/salidas

El estándar Arduino otorga a cada entrada/salida un número de identificación que es independiente del modelo de tarjeta Arduino empleada. Así pues, la salida número 13 está conectada a un diodo LED incorporado en la tarjeta Arduino y es la salida que usa el programa mostrado. En nuestro caso, el diodo LED RGB de la tarjeta de prácticas está conectado a los pines de E/S números 6 —azul—, 7 —verde— y 8 —rojo—, mientras que el pin 13 está conectado al pulsador, como se muestra en el Cuadro A.5 del Apéndice A.

9.5. Ejercicios

▶ **9.1** Conecta a la tarjeta Arduino la tarjeta de prácticas de forma que los tres pines bajo el LED se correspondan con los pines `6`, `7` y `8`

y los otros dos pines estén conectados al pin `13` y al pin `GND` que hay junto a él. Recuerda conectar el cable al pin `3.3V` de la tarjeta Arduino.

9.1.1 Inicia el entorno Arduino y abre el proyecto `blink` mediante la opción `Archivo - Ejemplos - 01.Basics - Blink` del menú. Compílalo y súbelo a la tarjeta. Comprueba que el LED incorporado en la tarjeta Arduino Due parpadea —de color amarillo, situado aproximadamente entre los dos conectores USB de la misma e identificado con la letra `L`—.

9.1.2 Sustituye en `blink.c` las tres apariciones del número 13 (como argumento de la función «`pinMode`» y de las dos llamadas a la función «`digitalWrite`») por el número 6. Compila y sube a la tarjeta el nuevo programa. ¿Cómo ha cambiado el comportamiento del programa?

9.1.3 Modifica el programa `blink.c` para que haga parpadear el LED de color rojo.

▶ **9.2** Abre el proyecto `blink_asm`, que se encuentra en la carpeta «1. Introducción a la ES» de la colección de ejercicios para Arduino.

9.2.1 Compílalo y ejecútalo. Comprueba que el LED de la tarjeta Arduino Due parpadea. Recordemos que el microcontrolador ATSAM3X8E, incorporado en la tarjeta Arduino Due que estamos usando, posee varios `PIOs` (*Parallel Input Output*) con varios pines de E/S cada uno de ellos, de forma que los pines `6`, `7` y `8` de la tarjeta Arduino están físicamente conectados a los pines `24`, `23` y `22` del `PIO C` del ATSAM3X8E respectivamente, como se muestra en el Cuadro A.5 del Apéndice A.

9.2.2 Consulta el Cuadro A.1 del Apéndice A para determinar la dirección del registro base del `PIO C`.

9.2.3 Realiza las modificaciones necesarias en `blink_asm.c` y en `blink.s` para que haga parpadear el LED de color rojo. Ten en cuenta que, mientras que el LED incorporado en la tarjeta Arduino Due se enciende escribiendo un `1` y se apaga escribiendo un `0` en el puerto correspondiente, cada componente del LED RGB de la tarjeta de prácticas se enciende escribiendo un `0` y se apaga escribiendo un `1` en su puerto de E/S.

9.2.4 Comenta qué modificaciones has tenido que introducir en el programa.

▶ **9.3** Tal como vimos al estudiar las funciones, hay dos formas de pasar parámetros: por valor y por referencia. En el ejemplo propuesto se muestra la técnica de paso de parámetros por valor a un programa en ensamblador —a través de los registros `r0` y `r1`—. Alguna vez, sin embargo, será necesario poder acceder desde el programa en ensamblador a una variable del programa principal. Debemos establecer un mecanismo para poder transferir información entre el programa en `C` y el código ensamblador. Para ello declaramos un vector en `C` y declaramos su nombre como `.extern` en el programa ensamblador para acceder a él usando su propio nombre como etiqueta.

Abre el proyecto «`blink_cadena`», que se encuentra en la carpeta «1. Introducción a la ES» de la colección de ejercicios para Arduino, y observa que presenta las siguientes diferencias con respecto al proyecto «`blink_asm`»:

- Se ha modificado la declaración de la función `blink` en el fichero «`blink.h`» para que no acepte parámetros pero siga devolviendo un resultado. Para ello se han eliminado las declaraciones de los dos parámetros: «`int time`» e «`int del`». También se han eliminado los parámetros entre paréntesis en la invocación a la función `blink` en el fichero «`blink_asm.ino`».
- Se ha declarado una cadena de caracteres al principio del programa en `C`, con el contenido «`mensaje`» donde es importante que el último elemento de la cadena sea el carácter `0`.
- Se ha declarado el nombre de la cadena en el programa ensamblador como «`.extern`» para que dicho nombre pueda utilizarse como una etiqueta dentro del programa ensamblador.
- Al eliminar los parámetros de la función `blink`, en el programa en ensamblador hemos asignado al retardo una cantidad fija —#300— y el número de veces que parpadea el LED será el número de caracteres de que consta la cadena. Para ello se ha confeccionado un bucle que recorre la cadena y realiza un parpadeo por cada carácter de la misma hasta encontrar el `0` del final.

9.3.1 Completa el programa ensamblador para que realice la función descrita.

9.3.2 Compílalo y súbelo a la tarjeta Arduino Due.

9.3.3 Modifica la longitud de la cadena para comprobar que realmente el programa hace lo que se espera.

9.5.1. Ejercicios de nivel medio

▶ **9.4** Modifica el programa ensamblador para que devuelva el número de caracteres de la cadena. Para ello simplemente tienes que copiar dicho valor en el registro `r0` antes de que el programa en ensamblador regrese.

▶ **9.5** Modifica el programa en `C` para que muestre en pantalla el número de caracteres de la cadena. Recuerda que para ello debes emplear las funciones que muestran información en pantalla, es decir, «`Serial.print`» y «`Serial.println`».

9.5.2. Ejercicios de nivel avanzado

▶ **9.6** Consulta el Cuadro A.8 del Apéndice A y, a partir de su contenido, completa el proyecto «`leefecha`», disponible en la carpeta «1. Introducción a la ES» de la colección de ejercicios para Arduino, para que acceda a los registros `RTC_CALR` y `RTC_TIMR` y lea la configuración de fecha y hora actuales del ATSAM3X8E.

9.6.1 Comparte esa información con el programa principal mediante los vectores `fecha` y `hora` y muestra en pantalla la fecha y hora actuales usando adecuadamente las funciones «`Serial.print`» y «`Serial.println`».

9.6.2 ¿Cuál es la fecha con que se configura el RTC por defecto?

9.6.3 En el campo `DAY` del registro `RTC_CALR` se almacena el día de la semana dejando la codificación al criterio del usuario. Atendiendo al contenido de este campo cuando se inicializa el sistema, ¿qué codificación se emplea por defecto para el día de la semana?

9.5.3. Ejercicios adicionales

▶ **9.7** La técnica de compartición de variables se puede emplear también para devolver información desde el programa en ensamblador al programa invocador. En lenguaje `C` se puede reservar espacio para un vector de enteros llamado «`vector`», de forma equivalente a como haríamos con «`.space m`» en ensamblador, de la siguiente forma:

«`int vector[n];`»

Hay que tener en cuenta, sin embargo, que el parámetro «`m`» indica número de bytes a reservar, mientras que cada elemento del vector

ocupa 4 bytes —1 word— en memoria. Así pues, para realizar la reserva en memoria de un vector de «`n`» elementos, en ensamblador debemos usar $m = 4 * n$.

Así mismo, desde el programa en `C` podemos acceder al elemento i-ésimo del vector mediante «`vector[i]`», pudiendo «`i`» tomar valores entre «`0`» y «`n-1`».

9.7.1 Abre el proyecto «`blink_vect`» —carpeta «1. Introducción a la ES» de la colección de ejercicios para Arduino—.

9.7.2 Compléta dicho proyecto considerando lo expuesto anteriormente y empleándolo de forma adecuada, para que devuelva en el vector llamado «`retorno`» el contenido de los registros `r0` al `r7`.

9.7.3 Introduce las modificaciones necesarias para que el programa en `C` muestre en pantalla el contenido de los registros `r0` al `r7` empleando «`Serial.print`» y «`Serial.println`». Ten en cuenta que estas funciones pueden mostrar valores numéricos en hexadecimal si se les añade el modificador «`HEX`», como por ejemplo en «`Serial.println(n, HEX);`».

CAPÍTULO 10

Gestión de la Entrada/Salida y otros aspectos avanzados

Índice

Tras ver en capítulos anteriores la función de la entrada/salida en los ordenadores, así como algunos de los dispositivos más comúnmente usados, este capítulo se va a dedicar a describir tanto la forma de gestionar la entrada/salida por parte del procesador, como otros aspectos avanzados. En los primeros apartados se tratará la forma que deben tener los programas para poder interactuar con los dispositivos de entrada/salida. Se verán los mecanismos de *consulta de estado* y de *gestión mediante interrupciones* como soluciones de sincronización, cada una 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 *DMA* (*Direct Memory Access*, 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 a nivel de conexión física entre el ordenador y los dispositivos, mediante buses estándar, como a nivel

de las partes del sistema operativo que los gestionan, los controladores de dispositivos.

10.1. Gestión de la entrada/salida

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

Así pues, el aspecto fundamental de la gestión de la entrada/salida, que intenta en lo posible evitar que el procesador preste atención al dispositivo mientras no sea necesario, es la *sincronización*. Se pretende que el procesador y los dispositivos se sincronicen de tal modo que aquél solo les preste atención cuando hay alguna actividad que realizar —recoger datos si ya se han obtenido, enviar nuevos datos si se han consumido los anteriores, solucionar algún error o avisar al usuario de ello—.

10.1.1. Gestión de la entrada/salida mediante consulta de estado

Sabemos que los registros de estado del controlador del dispositivo sirven para este fin, indicar que se ha producido alguna circunstancia que posiblemente requiere de atención. Por lo tanto, la forma más sencilla de sincronización con el dispositivo, llamada *consulta de estado*, *prueba de estado* o *encuesta* —en inglés, *polling*— consiste en que el procesador, durante la ejecución del programa en curso, lea de cuando en cuando los registros de estado necesarios y, si advierte que el dispositivo requiere atención, pase a ejecutar el código necesario para prestarla, posiblemente contenido en una subrutina de gestión del dispositivo.

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

10_consulta_estado.s

```
1       ldr     r7, =ST_IMPR    @ Dirección del registro de estado
```

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

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

Esta forma de gestionar la entrada/salida es muy sencilla, no 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 sincronizarse con la entrada/salida.

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

En este apartado hemos visto que la sincronización entre el procesador y los dispositivos es el punto clave de la gestión de la entrada/salida. El procesador puede consultar de forma sencilla los bits de estado de un dispositivo para ver si necesita atención, gestionando la entrada/salida mediante consulta de estado o encuesta.

▶ **10.1** Sabiendo que el pulsador incorporado en la tarjeta de prácticas de laboratorio está conectado a un pin del PIOB, ¿qué registro deberíamos leer para comprobar si se ha presionado el pulsador?

▶ **10.2** El pin al que está conectado el pulsador es el correspondiente al bit 27 del PIOB. ¿Qué máscara tenemos que aplicar al valor leído del registro para desechar todos sus bits excepto el que indica el estado del pulsador?

▶ **10.3** De acuerdo con el esquema de conexión del pulsador de la tarjeta de prácticas mostrado en la Figura 9.7 y si la entrada en que se encuentra conectado el pulsador tiene activada la resistencia interna de pull-up, ¿qué valor leído del bit 27 del registro de E/S del PIOB (`0` o `1`) nos indicará que el pulsador está presionado?

▶ **10.4** Abre el proyecto «`pulsa`» —en la carpeta «Pulsador» de la carpeta «2. Consulta de estado» de la colección de ejercicios para Arduino— y complétalo para que, mediante consulta de estado, espere la pulsación del pulsador de la tarjeta de prácticas. Cuando regrese, el programa debe devolver el `CHIPID` (código de identificación del chip) del procesador.

10.1.2. Gestión de la entrada/salida mediante interrupciones

A la vista de los problemas descritos, sería bueno que fuera el propio dispositivo el que avisara al procesador en caso de necesitar su atención, sin que éste tuviera que hacer nada de forma activa. Dado que el procesador es el encargado de gestionar todo el sistema, en último término sería quien podría decidir qué dispositivos tienen permiso para avisarle y si hacer o no caso a sus avisos una vez recibidos. Estas ideas se recogen en el mecanismo de gestión de la entrada/salida mediante interrupciones.

Según esta idea, el mecanismo de gestión de entrada/salida mediante interrupciones permite que cuando un dispositivo, con permisos para ello, activa un aviso en sus registros de estado, provoque una señal eléctrica que fuerza al procesador, al terminar de ejecutar la instrucción en curso, a saltar automáticamente al código que permite gestionar el dispositivo. Cuando se completa este tratamiento, el procesador continúa ejecutando la instrucción siguiente a la que estaba ejecutando cuando llegó la interrupción, como si hubiera retornado de una subrutina —pero con más implicaciones que estudiaremos a continuación—. El símil

más usado es el de la llamada telefónica, que llega mientras estamos leyendo tranquilamente un libro. Al sonar el teléfono —señal de interrupción— dejamos una marca en la página que estamos leyendo, y vamos a atenderla. Al terminar, continuamos con la lectura donde la habíamos dejado. De esta manera, los dispositivos son atendidos con muy poca latencia y los programas de aplicación no necesitan incluir instrucciones de consulta ni preocuparse de la gestión de la entrada/salida.

10.1.3. Mecanismo de interrupciones en el procesador y en el sistema

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

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

- El procesador, guiado por el código con el que ha sido programado, es el gestor de todo el sistema. Así pues, debe poder seleccionar qué dispositivos tienen permiso para interrumpirlo y cuáles no. Esto se consigue mediante los **bits de habilitación de interrupciones**. Normalmente residen en registros de control de los controladores de dispositivos, que tendrán uno o más, según los tipos de interrupciones que puedan generar. Además, el procesador dispone de uno o varios bits de control propios para deshabilitar totalmente las interrupciones, o hacerlo por grupos, según prioridades, etcétera. Más adelante explicaremos este aspecto con más detalle.

- La arquitectura de un procesador especifica cómo debe responder a la señalización de una interrupción habilitada. La organización

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

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

El mecanismo de interrupciones ha demostrado ser tan eficaz que se ha generalizado más allá de la entrada/salida en lo que se llama **excepciones** —*exceptions* o *traps*, en inglés—. Una excepción sirve para señalar cualquier circunstancia fuera de lo habitual —por lo tanto, excepcional— durante el funcionamiento de un procesador en su relación con el resto de componentes del ordenador. En este marco más amplio, las **interrupciones** son excepciones generadas por los dispositivos de entrada/salida. Otras excepciones se utilizan para señalar errores —accesos a memoria inválidos, violaciones de privilegio, división por cero,

etcétera— que en muchos casos pueden ser tratados por el sistema y dar lugar a extensiones útiles. Por ejemplo, el uso del disco duro como extensión de la memoria principal se implementa mediante la excepción de fallo de página; las excepciones de coprocesador no presente permiten incorporar emuladores al sistema, como la unidad en coma flotante emulada en los antiguos PC. Entre las excepciones se incluyen también las generadas voluntariamente por software mediante instrucciones específicas o registros a tal efecto de la arquitectura, que al provocar un cambio al modo de ejecución privilegiado, permiten implementar las llamadas al sistema como se verá al estudiar sistemas operativos.

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

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

El uso de interrupciones para la gestión de la entrada/salida se debe tener en cuenta desde el diseño del hardware del sistema. El procesador seleccionado deberá ser capaz de gestionarlas y contar, por tanto, con una o varias líneas externas de interrupción. Es posible además que utilice interrupciones vectorizadas y el número de interrupción —asociado al vector— se entregue al señalarla mediante cierto protocolo de comunicación específico. En este caso, habrá que añadir al sistema algún **controlador de interrupciones**. Estos dispositivos suelen ampliar el número de líneas de interrupción del sistema para conectar diversos dispositivos y se encargan de propagar al procesador las señales en tales líneas, enviando además el número de interrupción correspondiente. Desde el punto de vista del procesador, el controlador de interrupciones funciona como un sencillo dispositivo de entrada/salida. Ejemplos de estos dispositivos son el *NVIC* utilizado en la arquitectura ARM y que se verá más adelante o el *8259A* asociado a los procesadores Intel x86. En estos controladores se puede programar el número de interrupción asociado a cada línea física, la prioridad, las máscaras de habilitación y algún otro aspecto relacionado con el comportamiento eléctrico de las interrupciones.

Es interesante comentar que una interrupción puede señalarse eléc-

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

Una vez diseñado el hardware hay que programar el código de las diversas RTI y configurar los vectores de interrupción para que se produzcan las llamadas adecuadamente. Las direcciones de los vectores suelen estar fijas en el mapa de memoria del procesador, es decir en su arquitectura, de forma que a cada número de interrupción le corresponde una dirección de vector determinada. Algunos sistemas permiten configurar el origen de la tabla de vectores de interrupción, entonces la afirmación expuesta antes hace referencia, no al valor absoluto del vector, sino a su desplazamiento con respecto al origen. Sea como sea, los vectores de interrupción suelen reservar una zona de memoria de tamaño fijo —y reducido— en la que no hay espacio para el código de las RTI. Para poder ubicar entonces el código de tratamiento con libertad en la zona de memoria que se decida al diseñar el software del sistema, y con espacio suficiente, se utilizan dos técnicas. La más sencilla consiste en dejar entre cada dos vectores de interrupción el espacio suficiente para una instrucción de salto absoluto. Así pues, al llegar la interrupción se cargará el PC con la dirección del vector y se ejecutará la instrucción de salto que nos llevará al código de la RTI en la dirección deseada. La segunda opción requiere más complejidad para el hardware del procesador. En este caso, en la dirección del vector se guarda la dirección de inicio de la RTI, usando una especie de direccionamiento indirecto. Cuando se produce la interrupción, lo que copiamos en el PC no es la

dirección del vector, sino la dirección contenida en el vector, lo que otorga al sistema flexibilidad para ubicar las RTI libremente en memoria. Esta última opción es la utilizada en las arquitecturas ARM e Intel de 32 y 64 bits.

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

Una vez el sistema está en funcionamiento, ya configurado, las interrupciones pueden llegar de forma asíncrona con la ejecución de instrucciones. En este caso, el procesador, al terminar la instrucción en curso, de alguna de las formas vistas, carga en el contador de programa la dirección de la RTI. Previamente debe haber guardado el valor que

contenía el contador de programa para poder retornar a la instrucción siguiente a la que fue interrumpida. Al mismo tiempo, se produce un cambio a modo de funcionamiento privilegiado —por supuesto, solo en aquellos procesadores que disponen de varios modos de ejecución— y se deshabilitan las interrupciones. Bajo estas circunstancias comienza la ejecución de la RTI.

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

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

En este apartado hemos visto que si el procesador incorpora el hardware necesario, puede ser el dispositivo el que le avise de que necesita su atención, mediante interrupciones. En este mecanismo, el procesador debe incorporar más complejidad en sus circuitos, pero los programas pueden diseñarse de forma independiente de la entrada/salida. Para la gestión de excepciones, que generaliza e incluye la de interrupciones, el procesador debe ser capaz de interrumpir la ejecución de la instrucción en curso y de volver a la siguiente una vez terminado el tratamiento; de saber dónde comienza la rutina de servicio RTI mediante vectores; de establecer prioridades entre aquéllas y de preservar el estado para poder continuar la ejecución donde se quedó al llegar la interrupción.

▶ **10.5** En el proyecto «`PIOint`» —en la carpeta «3. Interrupciones» de la colección de ejercicios para Arduino—, completa la función «`PIOsetupInt`» para que configure adecuadamente el modo de interrupción seleccionado por los parámetros que se le suministran.

Una vez completada, comprueba que funciona correctamente compilando y subiendo el programa a la tarjeta Arduino Due.

. . . . . .

10.2. Transferencia de datos y DMA

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

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

sistema, llamado **controlador de DMA**, se encarga de realizar dichos movimientos de datos.

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

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

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

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

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

▶ **10.6** Abre el proyecto «`testDMA`» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino—, compílalo y súbelo a la tarjeta Arduino Due. Observa en el funcionamiento del programa que el contador de iteraciones se incrementa al mismo tiempo que el DMA realiza las transferencias de datos.

▶ **10.7** Busca en el código del programa el grupo de líneas encerrado en el comentario «`TAMAÑO DE LOS BLOQUES A TRANSFERIR`» y prueba diferentes valores de los parámetros «`SIZE0`», «`SIZE1`», «`SIZE2`» y «`SIZE3`». Ten en cuenta que deben tener valores comprendidos entre 100 y 2 048. Explica cómo cambia el tiempo de ejecución en función de dichos valores.

▶ **10.8** ¿Para qué valores de los parámetros se obtiene el tiempo máximo de realización de las transferencias?¿De qué tiempo se trata?¿Qué valor ha alcanzado el contador de iteraciones para dicho tiempo máximo?

10.3. Estandarización y extensión de la entrada/salida: buses y controladores

La entrada/salida es un componente indispensable en los ordenadores porque permite que se comuniquen con el mundo exterior. La tendencia actual de los sistemas informáticos es que cada vez estén más orientados a la entrada/salida: el concepto de dispositivos conectados o internet de las cosas implica disponer de pequeños ordenadores empotrados con capacidad de comunicación —entrada/salida— que estén constantemente midiendo datos a través de un conjunto de sensores y realizando actuaciones sobre el sistema al que están unidos. De nuevo,

entrada/salida. Pero esta tendencia se descubre también en los ordenadores personales: su disposición para el ocio, la reproducción multimedia y gráfica hace que estén conectados de forma habitual a mandos de juego, subsistemas de audio, impresoras, tabletas digitalizadoras, etcétera. De esta manera, tanto los sistemas empotrados como los ordenadores personales en la actualidad deben comunicarse con una gran cantidad de dispositivos de entrada/salida. Y la tendencia es que todo 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— que además se encuentre en el interior del propio ordenador. Por otra parte, llevar al exterior este bus supone problemas tanto mecánicos como electrónicos que, si bien es cierto que se pueden resolver, siempre presentan un elevado coste en la productividad del bus y, sobre todo, en el coste de los dispositivos y conectores. Por todo esto, la solución ha sido, desde hace años, desarrollar buses específicos para la entrada salida que permitan conectar dispositivos al ordenador a través de estos buses. El compromiso de diseño para estos buses de entrada/salida es que sean económicos y presenten a la vez unas buenas prestaciones. En el mundo de los ordenadores personales el bus que ha ganado casi en exclusividad el mercado es el **Universal Serial Bus** o **USB**, en sus diferentes versiones. En el mundo de los microcontroladores y los sistemas empotrados se utilizan indistintamente el bus SPI y el bus I2C.

Cuando se utiliza un bus de entrada/salida en un sistema, los dispositivos conectados al bus dejan de exisitir como tales para la arquitectura: una impresora USB, un disco duro USB, no pueden generar interrupciones al procesador —¿cómo activan una línea de interrupción, si están conectados a cinco metros y a través de un cable USB estándar?— ni transferir datos mediante DMA. A nivel del procesador, al nivel que hemos tratado en estos temas, el único dispositivo de entrada/salida para el procesador es el controlador raíz USB o *HOST*. Se trata en este caso de un dispositivo que sí se conecta al bus del sistema —el PCI express en los PC actuales— y que sí genera interrupciones y permite transferencias mediante DMA. Pero nuestro procesador, en un primer nivel, solo se comunica con este dispositivo HOST que se encarga de enviar y recibir bloques de datos para implementar la comunicación con cada dispositivo y el propio protocolo que sustenta los niveles lógicos del bus USB.

El sistema de entrada/salida es mucho más complejo en un ordenador real de lo que podemos ver en este libro introductorio. 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 o de interfaz humana —mediante el estándar HID, en inglés *Human Interface Device*— como teclados, ratones, mandos de juego, etcétera. En un sistema real toda la comunicación con los dispositivos forma parte de lo que se conoce como **controladores de dispositivos**, que son módulos del sistema operativo que utilizan las mencionadas reglas para permitir que la comunicación con los dispositivos sea adecuada. Los sistemas operativos actuales tienen en cuenta tanto la diversidad como la estandarización de la entrada/salida, de forma que los controladores de dispositivos se estructuran en niveles, y permiten gestionar adecuadamente todos los casos. Un ratón USB y un ratón Bluetooth utilizarán el mismo manejador de alto nivel que entienda el conjunto estándar de reglas HID. Este coltrolador se comunicará, según el caso, con otro de más bajo nivel que permita la comunicación física con el dispositivo a través del bus USB mediante el dispositivo HOST o Bluetooth utilizando el dispositivo de comunicaciones inalámbricas. En la realidad, para cada dispositivo pueden utilizarse tres o más controladores jerarquizados que terminan permitiendo, de forma eficaz, la comunicación entre el ordenador y el dispositivo.

En este apartado hemos visto cómo la entrada/salida está cada vez más presente en los sistemas actuales y esta tendencia sigue creciendo. Por ello es necesario utilizar buses estándar, como el USB en los PC o los SPI e I2C en otros sistemas empotrados, para comunicarse con los dispositivos. Esta complejidad física lleva a una complejidad en la gestión que los sistemas operativos resuelven mediante jerarquías de manejadores que gestionan tanto la diversidad como la estandarización.

▶ **10.9** Abre el proyecto «`kbmouse`» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino—, compílalo y súbelo a la tarjeta Arduino Due. Abre un editor de textos. Desconecta el cable USB del puerto de programación de la tarjeta Arduino Due y conéctalo al puerto de comunicaciones. Describe el funcionamiento del programa.

▶ **10.10** Consulta la ayuda de Arduino y modifica el programa para que simule un doble click en el instante en que se presiona el pulsador de la tarjeta Arduino Due. Presiona el pulsador cuando el puntero se encuentre sobre un icono del escritorio. ¿Qué ocurre?

10.4. Otros dispositivos

El uso extendido de los microprocesadores para la automatización y control de todo tipo de máquinas, electrodomésticos, dispositivos multimedia, herramientas, etc., ha dado lugar a un tipo específico de microprocesador, denominado *microcontrolador*, que se caracteriza principalmente por incorporar 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 microcontolador, aunque habitualmente suelen poder agruparse en dispositivos de gestión de las comunicaciones, dispositivos de gestión del almacenamiento y dispositivos de interacción con el mundo exterior. Dado que los microprocesadores trabajan exclusivamente con información digital (bits, bytes, words) y nuestro mundo es analógico (presenta magnitudes continuas como la longitud, peso, potencial eléctrico, intensidad luminosa, etc.), 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, que la mayoría de microcontroladores incorporan como dispositivos integrados.

Uno de los métodos más simples que existe para la conversión D/A (la que consiste en producir una señal analógica a partir de un valor numérico digital) es el denominado **Modulación de Anchura de Pulso** o **PWM** (*Pulse Width Modulation*, en inglés). Consiste en generar una señal cuadrada de periodo —tiempo entre dos flancos de subida o de bajada consecutivos— fijo y ciclo de trabajo —tiempo que la señal permanece a nivel alto — variable. De esta forma se transmite una cantidad de energía variable que un **filtro pasa-bajo** o **LPF** (*Low Pass Filter*, en inglés) —dispositivo electrónico que deja pasar bajas frecuencias y elimina las altas frecuencias— se encarga de convertir en un voltaje estable proporcional a dicha cantidad de energía, como muestra la Figura 10.1.

▶ **10.11** Consulta en la ayuda de Arduino el funcionamiento de la función «`analogWrite()`» y, al final de la misma, pincha en el enlace «`Tutorial:PWM`». ¿Cómo se consigue variar la intensidad luminosa del LED mediante la técnica PWM?

▶ **10.12** Abre el proyecto «`pwm`» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino—, compílalo y súbelo a la tarjeta Arduino Due.

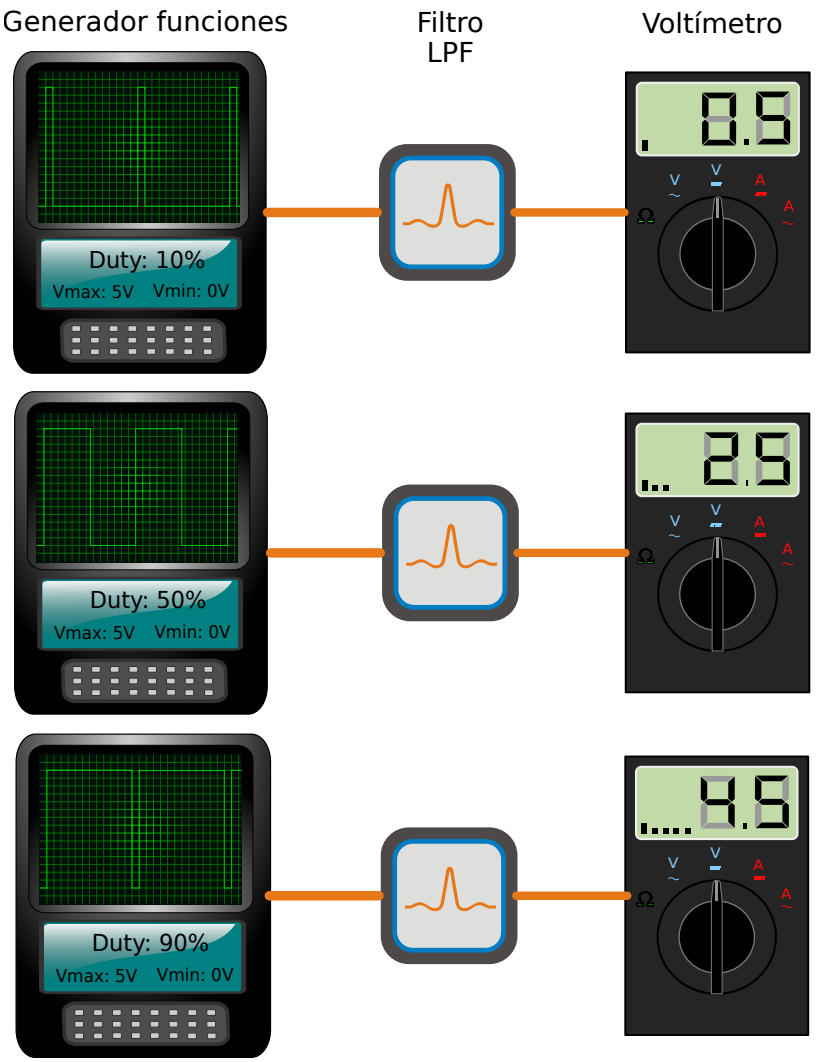


Figura 10.1: Método PWM para conversión Digital/Analógico

▶ **10.13** Observa que con cada pulsación cambia la intensidad del LED rojo mientras se indica el valor del ciclo de trabajo (*Duty Cycle*) que provoca la intensidad observada. Explica qué relación hay entre el valor del registro `PWM Channel Duty Cycle Register` y la intensidad del LED.

. . . . . .

10.5. Ejercicios

▶ **10.14** Modifica el proyecto «`pulsa`» —en la carpeta «Pulsador» de la carpeta «2. Consulta de estado» de la colección de ejercicios para Arduino— que has completado en el Ejercicio 10.4 para que espere hasta que se haya soltado el pulsador tras haberlo presionado antes de regresar al programa principal.

▶ **10.15** Modifica el código del proyecto «`PIOint`» —en la carpeta «3. Interrupciones» de la colección de ejercicios para Arduino— que has completado en el Ejercicio 10.5 para ir probando todas las posibles combinaciones de los parámetros «`mode`» y «`polarity`». Completa la siguiente tabla, comentando en cada caso cuál es el comportamiento del programa.

Mode	Polarity	Efecto
FLANCO	BAJO	
	ALTO	
NIVEL	BAJO	
	ALTO	
CAMBIO	BAJO	
	ALTO	

▶ **10.16** Abre el proyecto «`pwm`» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino—, compílalo y súbelo a la tarjeta Arduino Due. Desplaza la tarjeta con suavidad describiendo un arco de unos 60 centímetros a una velocidad moderada y observa el patrón que se visualiza para los diferentes valores del contenido del registro `PWM Channel Duty Cycle Register`. Explica cómo estos patrones obedecen al principio de funcionamiento del PWM visto en el tutorial de Arduino.

10.5.1. Ejercicios de nivel medio

▶ **10.17** Para poder modificar los contenidos de los registros de fecha —`RTC_CALR`— y hora —`RTC_TIMR`— es preciso detener su actualización escribiendo un `1` en los bits `UPDCAL` y `UPDTIM` del registro de control `RTC_CR` y esperando la confirmación de que se ha detenido la actualización en el bit `ACKUPD` del registro de estado `RTC_SR`. Puedes consultar los detalles de este procedimiento en el apartado A.4.4 *Actualización de la fecha y hora actuales*. Completa el proyecto

«cambiahora» para configurar el RTC en el modo de 12 horas con la fecha y hora actuales. Haz que el programa compruebe los valores de los bits NVCAL y NVTIM para confirmar que la configuración ha sido correcta.

▶ **10.18** Completa en el proyecto «RTCint» —en la carpeta «3. Interrupciones» de la colección de ejercicios para Arduino—, los valores de las etiquetas «HOY» y «AHORA» para configurar el RTC con la fecha y hora indicadas en los comentarios de las líneas de código que asignan valor a dichas etiquetas.

▶ **10.19** Completa asimismo los valores de las etiquetas «ALR_FECHA» y «ALR_HORA» para configurar la alarma del RTC de forma que se active a la fecha y hora indicadas en los comentarios de las líneas de código que asignan valor a dichas etiquetas.

▶ **10.20** Abre el proyecto «pwm» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino—, compílalo y súbelo a la tarjeta Arduino Due. Cambia el valor actual de la etiqueta «CLK» (0x0000 0680) por 0x0000 0180. Repite el experimento del Ejercicio 10.16 y comenta las diferencias apreciadas. ¿Qué crees que ha cambiado?

10.5.2. Ejercicios de nivel avanzado

▶ **10.21** Partiendo del proyecto «alblink» —en la carpeta «RTC» de la carpeta «2. Consulta de estado» de la colección de ejercicios para Arduino—, configura una alarma que se active dentro de 8 segundos. Completa el código proporcionado para que consulte el estado del bit de alarma y espere a que ésta se produzca para regresar al programa en C, el cual mostrará el instante en que se detectó la activación de la alarma. Ten en cuenta que tienes que copiar la función cambiahora confeccionada en el apartado anterior en la zona indicada del código fuente proporcionado.

▶ **10.22** Modifica el periodo del parpadeo del LED para que sea de 6 segundos (3 000 ms encendido y 3 000 ms apagado). ¿Coincide el instante de detección de la alarma con el configurado?¿A qué crees que es debido?

▶ **10.23** Completa la función «RTCsetAlarm» de forma que configure adecuadamente la fecha y hora de activación de la alarma contenidas en las etiquetas «ALR_FECHA» y «ALR_HORA» y que active la generación de interrupciones de alarma por parte del RTC.

▶ **10.24** Completa en el proyecto anterior la función «`RTC_Handler`» para que atienda la interrupción de alarma del RTC realizando correctamente las acciones que se describen en los comentarios del código proporcionado. Una vez completada la función, compila y sube el programa a la tarjeta Arduino Due. Explica qué acciones realiza.

▶ **10.25** Prueba diferentes valores de la etiqueta «`RETARDO`», comprueba su efecto en el comportamiento del programa y compáralo con el obtenido con el programa «`alblink`» del ejercicio 10.21. Comenta las diferencias observadas y relaciónalas con los métodos de consulta de estado e interrupciones.

▶ **10.26** Abre el proyecto «`EjemploPWM`» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino—, compílalo y súbelo a la tarjeta Arduino Due. ¿Cómo se consiguen los diferentes colores en el LED RGB?

▶ **10.27** Prueba diferentes valores de «`FACTRED`», «`FACTGRN`» y «`FACTBLU`». ¿Qué efecto tienen estos parámetros en el comportamiento del programa?

10.5.3. Ejercicios adicionales

▶ **10.28** 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.

▶ **10.29** Modifica el proyecto «`kbmouse`» —en la carpeta «4. DMA, PWM y USB» de la colección de ejercicios para Arduino— para que, en lugar de una elipse, el puntero del ratón describa un rectángulo.

APÉNDICE A

Información técnica ATSAM3X8E

Índice

En el presente anexo se recoge información técnica detallada sobre algunos de los dispositivos del microcontrolador ATSAM3X8E de la tarjeta Arduino Due, sobre la propia Arduino Due y sobre la tarjeta de expansión. Se describe con detalle el funcionamiento de la GPIO y los dispositivos de temporización; la gestión de las interrupciones y el controlador NVIC; y, por último, se describe brevemente el funcionamiento del controlador de DMA en el ATSAM3X8E.

A.1. GPIO en el Atmel ATSAM3X8E

El microcontrolador ATSAM3X8E dispone de bloques de GPIO muy versátiles y potentes. Dichos bloques, llamados *Parallel Input/Output*

Controller (*PIO* en inglés) se describen en detalle a partir de la página 641 del manual. Vamos a resumir en este apartado los aspectos más importantes, centrándonos en la versión ATSAM3X8E del microcontrolador, presente en la tarjeta *Arduino Due*. En la Figura A.1 (obtenida del manual [Atm12]) se muestra la estructura interna de un pin de E/S del microcontrolador ATSAM3X8E.

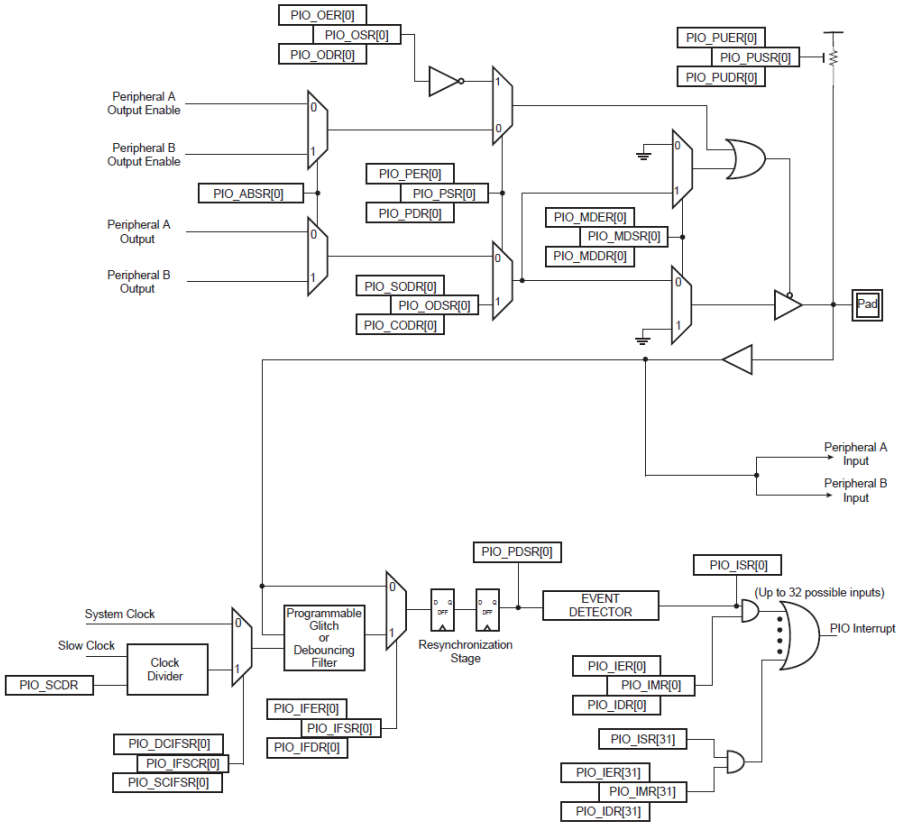


Figura A.1: Estructura interna de un pin de E/S del microcontrolador ATSAM3X8E

Con un encapsulado LQFP de 144 pines, el microcontrolador dispone de 4 controladores PIO capaces de gestionar hasta 32 pines cada uno de ellos, para un total de 103 líneas de GPIO. Cada una de estas líneas de entrada/salida es capaz de generar interrupciones por cambio de valor o como línea dedicada de interrupción; de configurarse para filtrar o eliminar rebotes de la entrada; de actuar con o sin *pull-up* o en colector abierto. Además, los pines de salida pueden ponerse a uno o a cero individualmente, mediante registros dedicados de *set* o *clear*, o conjuntamente a cualquier valor escribiendo en un tercer registro. To-

das estas características hacen que cada bloque PIO tenga una gran complejidad, ocupando un espacio de 324 registros de 32 bits —1 296 direcciones— en el mapa de memoria. Veamos a continuación los registros más importantes y su uso.

A.1.1. Configuración como GPIO o E/S específica de otro periférico

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

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

A.1.2. Configuración y uso genéricos como GPIO

Una vez los pines se han asignado a la GPIO, es necesario realizar su configuración específica. Esto incluye indicar si su dirección es entrada

o salida, y activar o no las resistencias de *pull-up* o la configuración en colector abierto. Veamos los registros del PIO que se utilizan:

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

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

Una vez configurada la GPIO, nuestro programa debe únicamente trabajar con los pines, escribiendo y leyendo valores según la tarea a realizar. De nuevo el bloque GPIO ofrece una gran versatilidad, a costa de cierta complejidad, como veremos a continuación. Comencemos con la lectura de los valores de entrada, algo sencillo dado que basta con leer el registro *Pin Data Status Register* (`PIO_PDSR`) para obtener los valores lógicos presentes en ellos. Es conveniente indicar que para poder leer los pines de entrada —igual que para muchas otras funciones del PIO— el reloj que lo sincroniza debe estar activado. La gestión de las salidas presenta algo más de complejidad dado que existen dos modos de actuar sobre cada una de ellas. Por una parte, tenemos la forma común en este microcontrolador, disponiendo de un registro para escribir unos y otro para escribir ceros. Este modo, que en muchas ocasiones simplifica la escritura en los pines, presenta el problema de que no se pueden escribir de forma simultánea unos y ceros. Por ello existe un segundo modo, en que se escribe en una sola escritura el valor, con los unos y ceros deseado, sobre el registro de salida. Para que este modo no afecte a todos los pines gestionados por el PIO —hasta 32— existe un registro adicional para seleccionar aquéllos a los que va a afectar esta escritura. Para poder implementar todos estos modos, el conjunto de registros relacionados con la escritura de valores en las salidas, es el siguiente:

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

A.1.3. Gestión de interrupciones asociadas a la GPIO

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

- *Interrupt Enable Register* (`PIO_IER`): escribiendo un `1` en cualquier bit de este registro se habilita la interrupción correspondiente.

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

A.1.4. Registros adicionales y funciones avanzadas del PIO

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

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

A.1.5. Controladores PIO en el ATSAM3X8E

Conocida la información que aparece en el texto anterior, para hacer programas que interactúen con la GPIO del microcontrolador solo falta conocer las direcciones del mapa de memoria en que se sitúan los registros de los controladores PIO del ATSAM3X8E y la dirección —más bien desplazamiento u *offset*— de cada registro dentro del bloque. El Cuadro A.1 muestra las direcciones base de los controladores PIO de que dispone el sistema.

PIO	Pines de E/S disponibles	Dirección base
PIOA	30	`0x400E0E00`
PIOB	32	`0x400E1000`
PIOC	31	`0x400E1200`
PIOD	10	`0x400E1400`

Cuadro A.1: Direcciones base de los controladores PIO del ATSAM3X8E

Así mismo, en los Cuadross A.2, A.3 y A.4 se muestran los desplazamientos (*offsets*) de los registros de E/S de cada uno de los controladores PIO, de forma que para obtener la dirección de memoria efectiva de uno

de los registros hay que sumar a la dirección base del controlador PIO al que pertenece, el desplazamiento indicado para el propio registro:

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

Cuadro A.2: Registros de E/S de cada controlador PIO y sus desplazamientos. Parte I

A.2. La tarjeta de entrada/salida

Para poder practicar con la GPIO se ha diseñado una pequeña tarjeta que se inserta en los conectores de expansión de la Arduino Due. La tarjeta dispone de un LED RGB —rojo *Red* verde *Green* azul *Blue*— conectado a tres pines que se usarán como salidas, y un pulsador conectado a un pin de entrada. Los tres diodos del LED están configurados en ánodo común, por lo que se encienden al escribir un `0` en la salida correspondiente. Cada canal del LED lleva su correspondiente resistencia para limitar la corriente; el terminal común se debe conectar, a través del cable soldado a la tarjeta, a la salida de `3.3V` de la Arduino Due. El pulsador se conecta a un pin de entrada a través de una resistencia de

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

Cuadro A.3: Registros de E/S de cada controlador PIO y sus desplazamientos. Parte II

protección, y a masa. Activando la resistencia de *pull-up* asociada al pin, se leerá un `1` lógico si el interruptor no está pulsado, y un `0` al pulsarlo.

El Cuadro A.5 y las Figuras 9.6 y 9.7 completan la información técnica acerca de la tarjeta.

A.3. El temporizador del Atmel ATSAM3X8E y del sistema Arduino

La arquitectura ARM especifica un temporizador llamado *System Timer* como la base principal de tiempos del sistema, con una frecuencia de incremento similar a la del procesador lo que le permite medir

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

Cuadro A.4: Registros de E/S de cada controlador PIO y sus desplazamientos. Parte III

intervalos de tiempo muy pequeños —del orden de microsegundos—. Para este temporizador, que no es otra cosa que un dispositivo de entrada/salida, aunque especial en el sistema, reserva sin embargo una excepción del sistema, llamada *SysTick*, con un número de interrupción fijo en el sistema, a diferencia del resto de dispositivos, cuyos números de interrupción no están fijados por la arquitectura.

El microcontrolador ATSAM3X8E implementa el *System Timer* especificado en la arquitectura. Se trata de un dispositivo de entrada/sa-

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

Cuadro A.5: Pines y bits de los dispositivos de la tarjeta de E/S en la tarjeta Arduino Due

lida que se comporta como un temporizador convencional, que se decrementa con cada pulso de su reloj. Dispone de cuatro registros, que se describen a continuación:

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

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

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

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

En el Cuadro A.6 aparecen las direcciones de los registros citados.

El entorno Arduino añade a cada programa el código necesario para la configuración del sistema y las rutinas de soporte necesarias. Entre ellas se tiene la configuración del *System Timer* y la rutina de tratamiento de la excepción *SysTick*. Este código configura el reloj del sistema, de 84MHz, como frecuencia de actualización del temporizador, y escribe el valor `0x01481F`, 83 999 en decimal, en el registro de recarga. De esta manera se tiene un cambio en el contador cada 12 nanosegundos más o

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

Cuadro A.6: Registros del temporizador del ATSAM3X8E y sus direcciones de E/S

menos y una interrupción cada milisegundo, lo que sirve de base para las funciones «`delay()`» y «`millis()`» del entorno. Ambas utilizan el contador de milisegundos del sistema «`_dwTickCount`», que es una variable en memoria que se incrementa en la rutina de tratamiento de *SysTick*. Las funciones de mayor precisión «`delayMicroseconds()`» y «`micros()`» se implementan leyendo directamente el valor del registro `VAL`.

A.4. El reloj en tiempo real del Atmel ATSAM3X8E

Algunos microcontroladores incorporan un RTC entre sus periféricos integrados, lo cual les permite disponer de fecha y hora actualizadas. En este caso, sin embargo, no suele existir alimentación específica para el módulo RTC, con lo cual, al desaparecer la alimentación externa, se pierde la información de fecha y hora actuales.

El microcontrolador ATSAM3X8E posee un RTC cuya estructura se muestra en la Figura A.2. Como puede apreciarse, recibe una señal de reloj `SCLK` (*Slow Clock*) generada internamente por el microcontrolador que presenta la ya mencionada frecuencia de 32 768 Hz. Esta señal se hace pasar por un divisor por 32 768 para obtener una señal de reloj de exactamente 1 Hz, que se encargará de activar las actualizaciones de los contenidos de los registros que mantienen la hora y la fecha actuales, en ese orden.

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

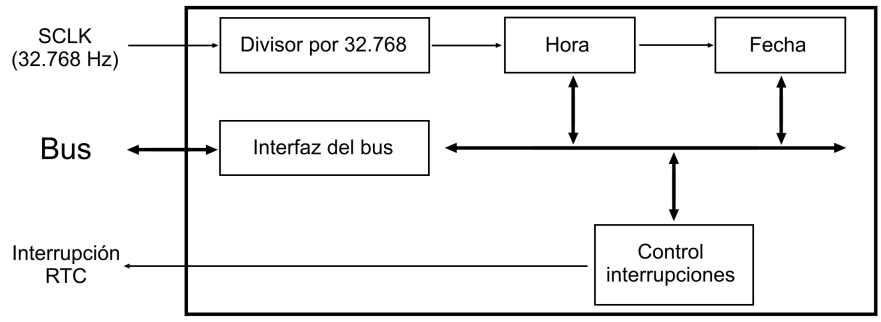


Figura A.2: Estructura interna del RTC del ATSAM3X8E

A.4.1. Hora actual

La hora actual se almacena en un registro de 32 bits denominado `RTC_TIMR` (*RTC Time Register*), cuyo formato se muestra en la Figura A.3, donde la hora puede estar expresada en formato de 12 horas más indicador AM/PM —bit 22— o en formato de 24 horas. Todos los valores numéricos están codificados en *BCD* (*Binary Coded Decimal*, decimal codificado en binario), cuya equivalencia se muestra en el Cuadro A.7.

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

Cuadro A.7: Equivalencia entre decimal y BCD

Los segundos —`SEC`— se almacenan en los bits 0 al 6, conteniendo los bits del 0 al 3 el valor de las unidades. Dado que las decenas adoptan como máximo el valor 5, para este dígito solamente son necesarios tres bits —del 4 al 6—, por lo cual el bit 7 no se usa nunca y siempre debe valer cero.

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

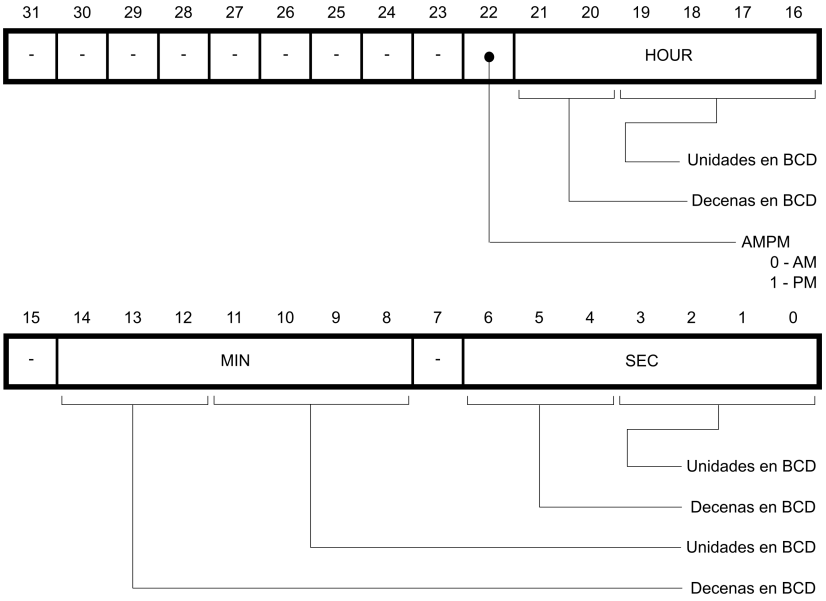


Figura A.3: Formato del registro RTC Time Register

14.

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

El resto de bits —del 24 al 31— no se usan. El valor de la hora actual se lee y se escribe como un valor de 32 bits accediendo a este registro con una sola operación de lectura o escritura.

A.4.2. Fecha actual

La fecha actual se almacena en el registro `RTC_CALR` (*RTC Calendar Register*) organizado como se muestra en la Figura A.4:

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

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

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

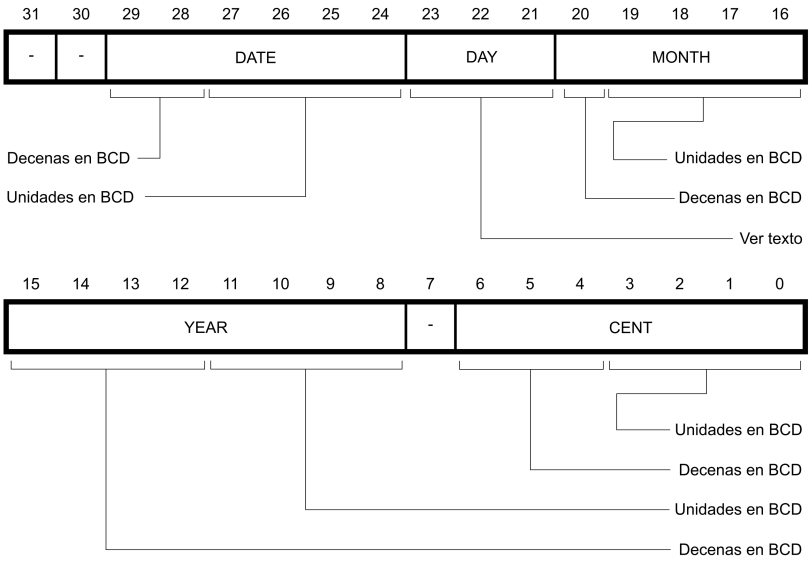


Figura A.4: Formato del registro RTC Calendar Register

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

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

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

A.4.3. Lectura de la fecha y hora actuales

Para poder acceder a los registros del RTC se debe conocer tanto la dirección base que ocupa el periférico en el mapa de memoria como el desplazamiento del registro al que se desea acceder. En este caso, el RTC del ATSAM3X8E abarca 256 direcciones —desplazamientos comprendidos entre `0x00` y `0xFF`— a partir de la dirección `0x400E1A60`. Los desplazamientos de los diferentes registros del RTC pueden consultarse en el Cuadro A.8, donde puede verse que el correspondiente al registro `RTC_TIMR` es `0x08` y el del registro `RTC_CALR` es `0x0C`. Así pues, para leer la fecha actual será necesario realizar una operación de lectura sobre la dirección `0x400E1A6C` y para obtener la hora actual será necesario leer el contenido de la dirección `0x400E1A68`.

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

Cuadro A.8: Desplazamientos de los registros del RTC

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

A.4.4. Actualización de la fecha y hora actuales

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

1. Inhibir la actualización del RTC. Esto se consigue mediante los bits UPDCAL para la fecha y UPDTIM para la hora. Ambos se encuentran, como muestra la Figura A.5, en el registro de control RTC_CR. Cada uno de estos bits detiene la actualización del contador correspondiente cuando toma al valor 1, permitiendo el funcionamiento

normal del RTC cuando vale `0`. Así pues, si deseamos establecer la fecha actual, deberemos escribir un `1` en el bit de peso 1 del registro `RTC_CR` antes de modificar el registro `RTC_CALR`. Este registro, junto con los que sirven para configurar las alarmas, dispone de una protección contra escritura que se puede habilitar en el registro `RTC_WPMR` (*RTC Write Protect Mode Register*) introduciendo la clave correcta en el registro, cuyo formato se muestra en la Figura A.6. Para que el cambio de modo de protección contra escritura de los registros protegidos —`RTC_CR`, `RTC_CALALR` y `RTC_TIMALR`— se produzca, la clave introducida en el campo `WPKEY` debe ser `0x525443` —'RTC' en ASCII—, mientras el byte de menor peso de la palabra de 32 bits debe tomar el valor `0x00` para permitir la escritura y el valor `0x01` para impedirla. Por defecto, la protección contra escritura está deshabilitada.

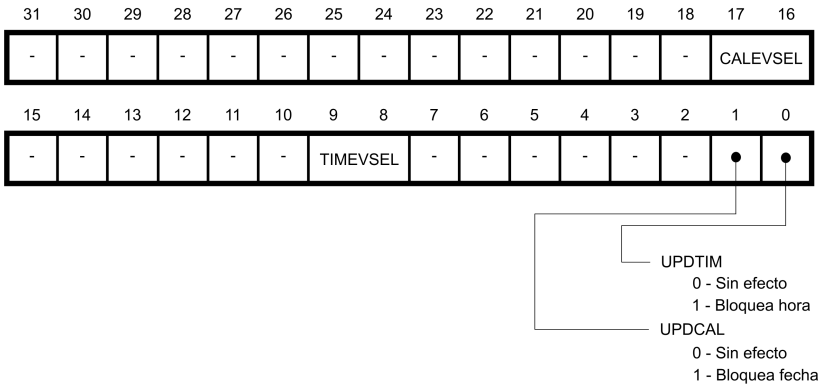


Figura A.5: Formato del registro RTC Control Register

2. Esperar la activación de `ACKUPD`, que es el bit de peso 0 del registro de estado `RTC_SR` —mostrado en la Figura A.7—. En caso de que la generación de interrupciones esté activada, no será necesario consultar el registro, dado que se producirá una interrupción.

3. Una vez se haya detectado que el bit `ACKUPD` ha tomado el valor `1`, es necesario restablecer este indicador escribiendo un `1` en el bit de peso 0, denominado `ACKCLR`, del registro `RTC_SCCR` (*RTC Status Clear Command Register*) cuyo formato se muestra en la Figura A.8.

4. Ahora se puede escribir el nuevo valor de la hora y/o fecha actuales en sus correspondientes registros —`RTC_TIMR` y `RTC_CALR` respectivamente—. El RTC comprueba que los valores que se escriben sean

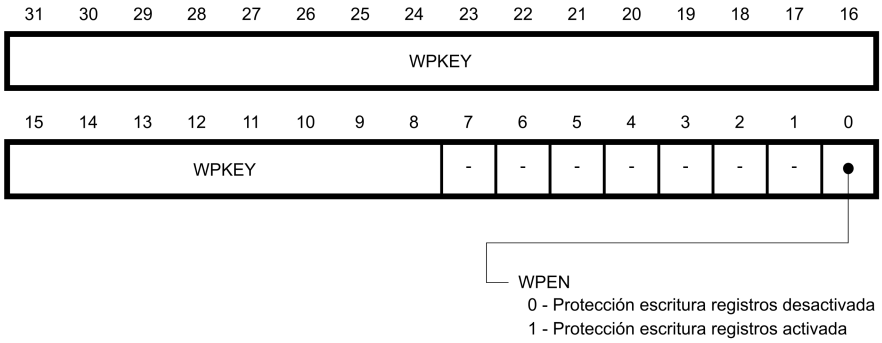


Figura A.6: Formato del registro RTC Write Protect Mode Register

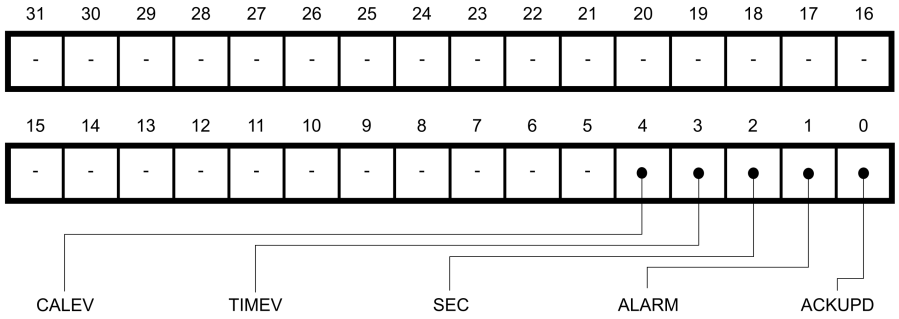


Figura A.7: Formato del registro RTC Status Register

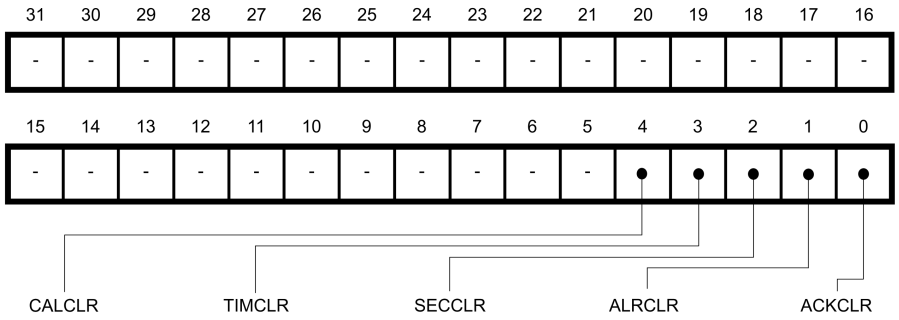


Figura A.8: Formato del registro RTC Status Clear Command Register

correctos. De no ser así, se activa el indicador correspondiente en el `RTC_VER` (*RTC Valid Entry Register*) cuyo formato se muestra

en la Figura A.9.

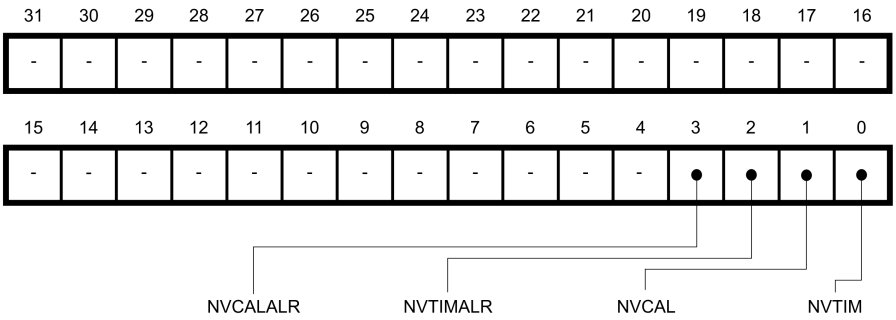


Figura A.9: Formato del registro RTC Valid Entry Register

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

5. Restablecer el valor de los bits de inhibición de la actualización —`UPDCAL` y/o `UPDTIM`— para permitir la reanudación del funcionamiento del RTC. Si solamente se modifica el valor de la fecha actual, la porción del RTC dedicada al cálculo de la hora actual sigue en funcionamiento, mientras que si solo se modifica la hora, el calendario también es detenido. La modalidad de 12/24 horas se puede seleccionar mediante `HRMOD`, bit de peso 0 del registro `RTC_MR` (*Mode Register*) cuyo formato se muestra en la Figura A.10. Escribiendo en `HRMOD` el valor `1` se configura el RTC en modo 24 horas, mientras que el valor `0` establece la configuración en el modo de 12 horas.

A.4.5. Alarmas

El RTC posee la capacidad de establecer valores de alarma para cinco campos: mes, día del mes, hora, minuto, segundo. Estos valores están repartidos en dos registros: `RTC_TIMALR` (*RTC Time Alarm Register*) cuyo formato es mostrado en la Figura A.11, y `RTC_CALALR` (*RTC Calendar Alarm Register*) cuyo formato es mostrado en la figura A.12.

Cada uno de los campos configurables posee un bit de activación asociado, de forma que su valor puede ser considerado o ignorado en

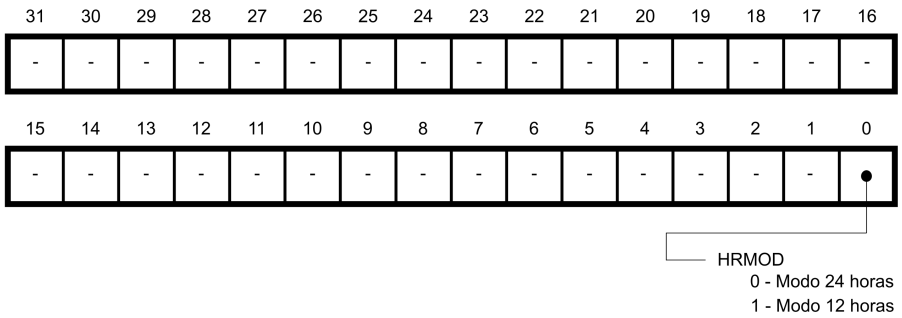


Figura A.10: Formato del registro RTC Mode Register

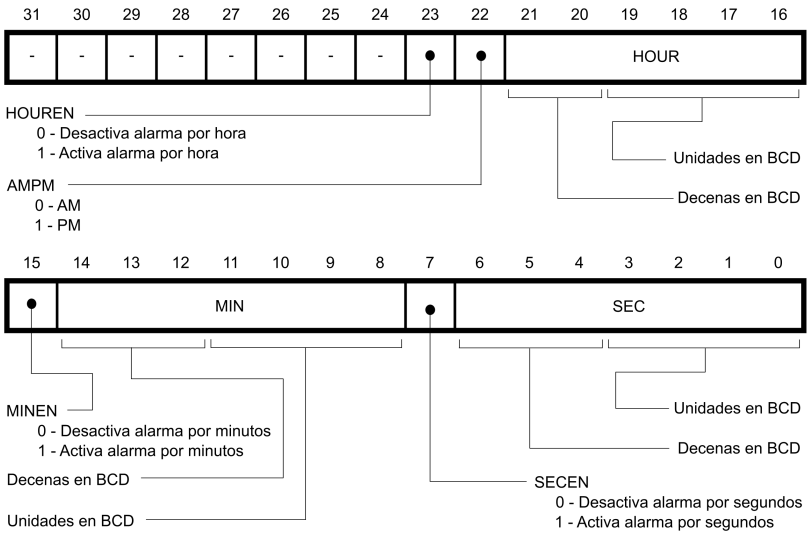


Figura A.11: Formato del registro RTC Time Alarm Register

la activación de la alarma. Así pues, si por ejemplo escribimos un `1` en `DATEEN` —bit 23 del registro `RTC_CALALR`— y el valor '18' en BCD en `DATE` —bits 16 a 20 del mismo registro— generaremos una alarma el día 18 de cada mes.

Los valores introducidos en los campos configurables se comprueban al igual que los de fecha y hora anteriormente comentados y, si se detecta un error, se activan los indicadores correspondientes del registro `RTC_VER` (*RTC Valid Entry Register*) mostrado en la Figura A.9. Si se activan todos los campos configurables y se establece un valor válido para cada uno de ellos, se configura una alarma para un instante determinado, llegado el cual se activará el bit `ALARM` (bit 1 del registro `RTC_SR` (*RTC Status*

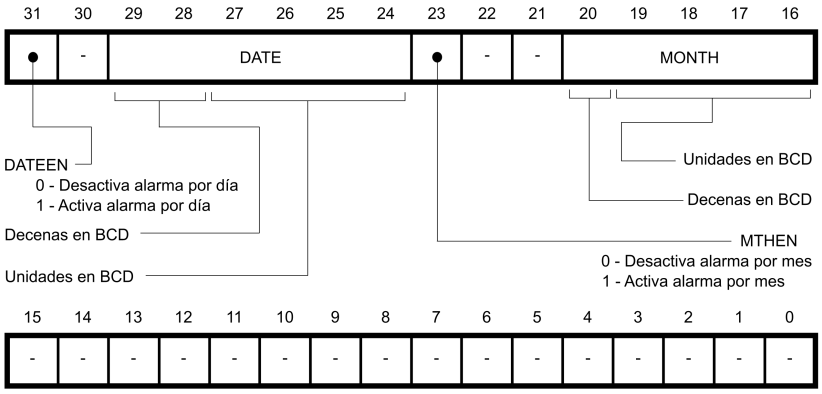


Figura A.12: Formato del registro RTC Calendar Alarm Register

Register) cuyo formato se muestra en la Figura A.7) y, en caso de estar activada la generación de interrupciones, se producirá una interrupción. Para restablecer los indicadores del registro `RTC_SR` (*RTC Status Register*) hay que escribir un `1` en cada uno de los bits correspondientes del registro `RTC_SCCR` (*RTC Status Clear Command Register*).

Si se produce una segunda alarma antes de que se haya leído el registro `RTC_SR` (*RTC Status Register*), mostrado en la Figura A.7, tras una alarma, se activará `SEC` —bit de peso 2 del registro `RTC_SR`— que indica que al menos dos alarmas se han producido desde que se restableció el valor del indicador por última vez.

A.4.6. Eventos periódicos

Además de las alarmas en instantes programados, como se ha visto en el apartado anterior, el RTC también posee la capacidad de producir alarmas periódicas con diferentes cadencias configurables a través del registro `RTC_CR` (*RTC Control Register*), cuyo formato se muestra en la Figura A.5. En sus bits 8 y 9 se encuentra el valor del campo `TIMEVSEL` que activa/desactiva la generación de eventos periódicos de hora y en los bits 16 y 17 el campo `CALEVSEL` que activa/desactiva la generación de eventos periódicos de calendario.

Un evento de hora puede ser a su vez de cuatro tipos diferentes, mostrados en el Cuadro A.9, dependiendo del valor que tome el campo `TIMEVSEL`.

De la misma forma, un evento de fecha puede ser a su vez de tres tipos diferentes, mostrados en el Cuadro A.10, dependiendo del valor que tome el campo `CALEVSEL`.

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

Cuadro A.9: Tipos de eventos periódicos de hora

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

Cuadro A.10: Tipos de eventos periódicos de fecha

Al igual que ocurre con las alarmas de tiempo concreto, la notificación de que se ha producido un evento periódico se produce a través del registro `RTC_SR` (*RTC Status Register*) mostrado en la Figura A.7, donde, en caso de que se haya producido un evento periódico de hora, se activará `TIMEV` —bit de peso 3— y en caso de que se haya detectado un evento periódico de fecha de acuerdo con lo configurado, se activará `CALEV` —bit de peso 4—.

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

A.4.7. Interrupciones en el RTC

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

- Actualización de fecha/hora.
- Evento de tiempo.
- Evento de calendario.
- Alarma.

- Segundo evento de alarma periódica.

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

La activación de la generación de interrupciones se consigue a través del registro `RTC_IER` (*RTC Interrupt Enable Register*), cuyo formato se muestra en la Figura A.13, donde puede apreciarse que se dispone de cinco bits de configuración para activar la generación de interrupciones:

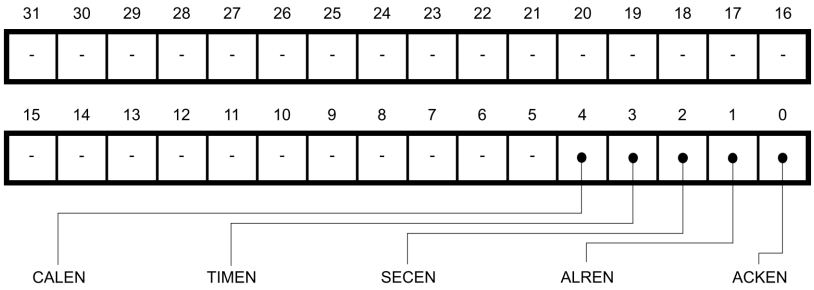


Figura A.13: Formato del registro RTC Interrupt Enable Register

- *Inhibición de la actualización del RTC*: escribiendo un `1` en `ACKEN` —bit de peso 0— activamos la generación de una interrupción cuando se active el bit `ACKUPD` del registro `RTC_SR` (*RTC Status Register*), mostrado en la Figura A.7, como consecuencia de haber inhibido la actualización del RTC mediante uno de los bits `UPDCAL` o `UPDTIM` del registro `RTC_CR` —véase la Figura A.5— o ambos.
- *Condición de alarma*: escribiendo un `1` en `ALREN` —bit de peso 1— activamos la generación de una interrupción al activarse el bit `ALARM` del registro `RTC_SR` (*RTC Status Register*). Esto ocurre cuando se cumple la condición de generación de alarma especificada en uno de los registros `RTC_TIMALR` (*RTC Time Alarm Register*) o `RTC_CALALR` (*RTC Calendar Alarm Register*).
- *Segunda alarma*: escribiendo un `1` en `SECEN` —bit de peso 2—, activamos la generación de una interrupción al activarse el bit `SEC` del registro `RTC_SR` (*RTC Status Register*), lo cual indica que se ha cumplido la condición de alarma en una segunda ocasión sin que el registro *RTC Status Register* haya sido leído.

- *Evento periódico de hora*: escribiendo un `1` en `TIMEN` —bit de peso 3— se activa la generación de una interrupción cuando se activa el bit `TIMEV` del registro `RTC_SR`, lo cual ocurrirá cuando se cumpla la condición periódica configurada en el campo `TIMEVSEL` del registro `RTC_CR` ((*RTC Control Register*) —véanse la Figura A.5 y el Cuadro A.9—.

- *Evento periódico de fecha*: escribiendo un `1` en `CALEN` —bit de peso 4— se activa la generación de una interrupción cuando se activa el bit `CALEV` del registro `RTC_SR` lo cual ocurrirá cuando se cumpla la condición periódica configurada en el campo `CALEVSEL` del registro `RTC_CR` (*RTC Control Register*) —véanse la Figura A.5 y el Cuadro A.10—.

Cada vez que se produzca una interrupción, en la correspondiente rutina de servicio se deberá acceder al registro `RTC_SR` (*RTC Status Register*) para averiguar cuál es la causa de la misma. Es posible que varias circunstancias hayan concurrido para la generación de la interrupción, con lo cual es aconsejable comprobar todos y cada uno de los bits del registro de estado. Una vez averiguadas las causas de la interrupción y tomadas las acciones pertinentes, antes de regresar de la rutina de servicio, se deben restablecer los indicadores escribiendo ceros en el registro `RTC_SCCR` (*RTC Status Clear Command Register*), cuyo formato se muestra en la Figura A.8, para dejar el sistema en disposición de que se produzcan nuevas interrupciones.

A.5. El Temporizador en Tiempo Real (RTT) del Atmel ATSAM3X8E

El *Temporizador en Tiempo Real* (o *Real Time Timer, RTT*) es un temporizador del ATSAM3X8E simple y versátil, por lo que se puede utilizar de forma sencilla —y con plena libertad dado que no es utilizado por el sistema Arduino—. Se trata básicamente de un registro contador de 32 bits, que tiene una frecuencia base de actualización de 32 768 Hz, como el RTC. Esta frecuencia se puede dividir gracias a un prescaler de 16 bits. El dispositivo dispone además de un registro de alarma para generar interrupciones cuando la cuenta del temporizador alcanza el valor almacenado en él, y es capaz además de generar interrupciones periódicas cada vez que se incrementa el valor del temporizador. Utiliza los cuatro registros que se describen a continuación:

- *Mode Register* (`RTT_MR`): es el registro de control del dispositivo. Los 16 bits más bajos almacenan el prescaler. Con un valor de `0x8000` se tiene una frecuencia de actualización de un segundo,

lo que indica que está pensado para temporizaciones relacionadas con tiempos de usuario más que de sistema. No obstante, se puede poner en estos bits cualquier valor superior a 2. El bit 18 —`RTTRST`— sirve para reiniciar el sistema —escribiendo un `1`—, poniendo el contador a `0` y actualizando el valor del prescaler. El bit 17 —`RTTINCIEN`— es la habilitación de interrupción por incremento, y el bit 16 —`ALMIEN`— la de interrupción por alarma. Ambas se habilitan con un `1`.

- *Alarm Register* (`RTT_AR`): almacena el valor de la alarma, de 32 bits. Cuando el contador alcance este valor, se producirá una interrupción en caso de estar habilitada.
- *Value Register* (`RTT_VR`): guarda el valor del contador, que se va incrementando con cada pulso —según la frecuencia base dividida por el prescaler—, de forma cíclica.
- *Status Register* (`RTT_SR`): es el registro de estado. Su bit 1 —`RTTINC`— indica que se ha producido un incremento del valor, y su bit 0 —`ALMS`— que ha ocurrido una alarma. Ambas circunstancias se señalan con un `1`, que se pone a `0` al leer el registro.

En el Cuadro A.11 aparecen las direcciones de los registros citados.

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

Cuadro A.11: Registros del temporizador en tiempo real del ATSAM3X8E y sus direcciones de E/S

Es interesante realizar un comentario acerca del uso del RTT y de la forma en que se generan las interrupciones. La interrupción por incremento está pensada para generar una interrupción periódica; según el valor del prescaler, el periodo puede ser desde inferior a una décima de milisegundo hasta casi dos segundos. Eléctricamente la interrupción se genera por flanco, por lo que al producirse basta con leer el `RTT_SR` para evitar que se genere hasta el próximo incremento, comportamiento típico de una interrupción periódica.

La interrupción de alarma, sin embargo, se produce por nivel mientras el valor del contador sea igual al de la alarma. Leer en este caso el `RTT_SR` no hace que deje de señalarse la interrupción, que se seguirá

disparando hasta que llegue otro pulso de la frecuencia de actualización. Este comportamiento no se evita cambiando el valor del `RTT_AR` pues la condición de interrupción se actualiza con el mismo reloj que incrementa el temporizador. Esto quiere decir que la interrupción por alarma está pensada para producirse una vez y deshabilitarse, hasta que el programa decida configurar una nueva alarma por algún motivo. Su uso como interrupción periódica no es, por tanto, recomendable.

A.6. Gestión de excepciones e interrupciones en el ATSAM3X8E

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

Se trata de un modelo vectorizado, con expulsión —*preemption*— en base a prioridades. Cada causa de excepción, de las que las interrupciones son un subconjunto, tiene asignado un número de orden que identifica un vector de excepción en una zona determinada de la memoria. Cuando se produce una excepción, el procesador carga el valor almacenado en ese vector y lo utiliza como dirección de inicio de la rutina de tratamiento. Al mismo tiempo, cada excepción tiene una prioridad que determina cuál de ellas será atendida en caso de detectarse varias a la vez, y permite que una rutina de tratamiento sea interrumpida —expulsada— si se detecta una excepción con mayor prioridad, volviendo a aquélla al terminar de tratar la más prioritaria. El orden de la prioridad es inverso a su valor numérico, así una prioridad de 0 es mayor que una de 7, por ejemplo.

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

Cuando se detecta una excepción, si no se está atendiendo a otra de mayor prioridad, el procesador guarda automáticamente en la pila los registros `r0` a `r3` y `r12`; la dirección de retorno, el registro de estado y el registro `lr`. Entonces realiza el salto a la dirección guardada en el vector de interrupción y pasa la excepción de pendiente a activa. En el registro `lr` se escribe entonces un valor especial, llamado `EXC_RETURN`, que indica que se está tratando una excepción, y que será utilizado para

volver de la rutina de tratamiento. La rutina de tratamiento tiene la estructura de una rutina normal, dado que los registros ya han sido salvados adecuadamente. Cuando se termina el tratamiento se retorna a la ejecución normal con una instrucción «`pop`» que incluya el registro `pc`. De esta manera, el procesador recupera de forma automática los valores de los registros que había guardado previamente, continuando con la ejecución de forma normal.

En el Cuadro A.12 aparecen las excepciones del sistema, con su número, su prioridad y el número de interrupción asociado. Por conveniencia, la arquitectura asigna a las excepciones un número de interrupción negativo, quedando el resto para las interrupciones de dispositivos.

Excepción	IRQ	Tipo	Prioridad	Vector (offset)
1	–	Reset	−3	`0x00000004`
2	−14	NMI	−2	`0x00000008`
3	−13	Hard fault	−1	`0x0000000C`
4	−12	Memory management fault	Configurable	`0x00000010`
5	−11	Bus fault	Configurable	`0x00000014`
6	−10	Usage fault	Configurable	`0x00000018`
11	−5	SVCall	Configurable	`0x0000002C`
14	−2	PendSV	Configurable	`0x00000038`
15	−1	SysTick	Configurable	`0x0000003C`
16	0	IRQ0	Configurable	`0x00000040`
17	1	IRQ1	Configurable	`0x00000044`
...	...	...	...	...

Cuadro A.12: Algunas de las excepciones del ATSAM3X8E y sus vectores de interrupción

A.6.1. El Nested Vectored Interrupt Controller (NVIC)

Como hemos visto en el apartado anterior, la arquitectura ARM especifica el modelo de tratamiento de las excepciones. Del mismo modo, incluye entre los dispositivos periféricos del núcleo —*Core Peripherals*— de la versión *Cortex-M3* de dicha arquitectura, un controlador de interrupciones llamado *Nested Vectored Interrupt Controller*, *NVIC*. Cada implementación distinta de la arquitectura conecta al NVIC las

interrupciones generadas por sus distintos dispositivos periféricos. En el caso del ATSAM3X8E, dichas conexiones son fijas, de manera que cada dispositivo tiene un número de interrupción —y por tanto, un vector— fijo y conocido de antemano.

El NVIC, además de implementar el protocolo adecuado para señalar las interrupciones al núcleo, contiene una serie de registros que permiten al software del sistema configurar y tratar las interrupciones según las necesidades de la aplicación. Para ello, dispone de una serie de registros especificados en la arquitectura, que en el caso del ATSAM3X8E permiten gestionar las interrupciones de los periféricos del microcontrolador. Veamos cuáles son esos registros y su función.

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

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

Al leer cualquiera de los registros anteriores se obtiene la máscara de interrupciones habilitadas, indicadas con `1` en los bits correspondientes.

Para gestionar las interrupciones pendientes se tienen:

- *Interrupt Set Pending Registers* (`ISPRx`): escribiendo un `1` en cualquier bit de uno de estos registros se marca la interrupción asociada como pendiente. Esto permite forzar el tratamiento de una interrupción aunque no se haya señalado físicamente.
- *Interrupt Clear Pending Registers* (`ICPRx`): escribiendo un `1` en cualquier bit de uno de estos registros se elimina la interrupción asociada del estado pendiente. Esto permite evitar que se produzca el salto por hardware a la rutina de tratamiento.

Al leer cualquiera de los registros anteriores se obtiene la lista de interrupciones pendientes, indicadas con `1` en los bits correspondientes. Una interrupción puede estar en estado pendiente aunque no esté habilitada en la máscara vista más arriba.

La lista de interrupciones activas se gestiona por hardware, pero se puede consultar en los registros de solo lectura *Interrupt Active Bit Registers* (`ICPRx`). En ellos, un `1` indica que la interrupción correspondiente está siendo tratada en el momento de la lectura. Las prioridades asociadas a las interrupciones se configuran en los registros *Interrupt Priority*

Registers (`IPRx`). Cada interrupción tiene asociado un campo de 8 bits en estos registros, aunque en la implementación actual solo los 4 de mayor peso almacenan el valor de la prioridad, entre 0 y 15.

Por último, existe un registro que permite generar interrupciones por software, llamado *Software Trigger Interrupt Register* (`STIR`). Escribiendo un valor en sus 9 bits menos significativos se genera la interrupción con dicho número.

Para dar cabida a los bits asociados a todas las posibles interrupciones —que pueden llegar a ser hasta 64 en la serie de procesadores SAM3X—, el NVIC implementado en el ATSAM3X8E dispone de 2 registros para cada uno de los conjuntos, excepto para el de prioridades que comprende 8 registros. Dado un número de IRQ es sencillo calcular cómo encontrar los registros y bits que almacenan la información que se desea leer o modificar, sin embargo ARM recomienda utilizar ciertas funciones en lenguaje C que suministra —en el modelo de código del sistema llamado *CMSIS*— y que son las que utilizaremos en nuestros programas. A continuación se describen las primitivas más usadas.

- «`void NVIC_EnableIRQ(IRQn_t IRQn)`». Habilita la interrupción cuyo número se le pasa como parámetro.
- «`void NVIC_DisableIRQ(IRQn_t IRQn)`». Deshabilita la interrupción cuyo número se le pasa como parámetro.
- «`uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)`». Devuelve un valor distinto de `0` si la interrupción cuyo número se pasa está pendiente; `0` en caso contrario.
- «`void NVIC_SetPendingIRQ (IRQn_t IRQn)`». Marca como pendiente la interrupción cuyo número se pasa como parámetro.
- «`void NVIC_ClearPendingIRQ (IRQn_t IRQn)`». Elimina la marca de pendiente de la interrupción cuyo número se pasa como parámetro.
- «`void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)`». Asigna la prioridad indicada a la interrupción cuyo número se pasa como parámetro.
- «`uint32_t NVIC_GetPriority (IRQn_t IRQn)`». Devuelve la prioridad de la interrupción cuyo número se pasa como parámetro.

A.6.2. Interrupciones del ATSAM3X8E en el entorno Arduino

Como se ha visto, buena parte de las tareas de salvaguarda y recuperación del estado en la gestión de interrupciones son realizadas por el

hardware en las arquitecturas ARM. Gracias a esto el diseño de rutinas de tratamiento queda muy simplificado.

La estructura de una RTI es idéntica a la de una rutina normal, con la única restricción de que no admite ni devuelve parámetros. El hecho de que los registros que normalmente no se guardan —`r0 ... r3`— sean preservados automáticamente hace que el código de entrada y salida de una RTI no difiera del de una rutina, y por ello, las únicas diferencias entre ambas estriban en su propio código.

Teniendo en cuenta esta estructura, y teniendo en cuenta las funciones de gestión del NVIC que se han descrito más arriba, lo único que se necesita saber para implementar una RTI es el número de interrupción. Además, para configurarla habría que modificar el vector correspondiente para que apunte a nuestra función. De nuevo, en el estándar CMSIS se dan las soluciones para estos requisitos. Todos los dispositivos tienen definido su número de IRQ —que recordemos es fijo— de forma regular. Del mismo modo, los nombres de las funciones de tratamiento están igualmente predefinidos, de manera que para crear una RTI para un cierto dispositivo simplemente hemos de programar una función con el nombre adecuado. El proceso de compilación de nuestro programa se encarga de configurar el vector de manera transparente para el programador.

Los Cuadros A.13, A.14 y A.15 muestran los símbolos que se deben usar para referirse a las distintas IRQ según el dispositivo, la descripción del dispositivo y el nombre de la rutina de tratamiento.

Una vez creada la RTI solo es necesario configurar el NVIC adecuadamente. El proceso suele consistir en deshabilitar primero la interrupción correspondiente, limpiar el estado pendiente por si se produjo alguna falsa interrupción durante el arranque del sistema y establecer la prioridad —que por defecto suele ser `0` en el entorno Arduino—. Una vez hecho esto, hemos de configurar el dispositivo de la manera que se desee y habilitar la interrupción correspondiente.

A continuación se muestran unos fragmentos de código que permitirían establecer una RTI para el dispositivo `PIOB`.

AA_RTI_PIOB.c

```
1  void setup() {
2          /* Otra configuración del sistema*/
3          NVIC_DisableIRQ(PIOB_IRQn);
4          NVIC_ClearPendingIRQ(PIOB_IRQn);
5          NVIC_SetPriority(PIOB_IRQn, 0);
6          PIOsetupInt(EDGE, 1);
7          NVIC_EnableIRQ(PIOB_IRQn);
8  }
9
10 // RTI en C
```

Símbolo	Núm	Dispositivo	RTI
SUPC_IRQn	0	Supply Controller (SUPC)	«void SUPC_Handler()»
RSTC_IRQn	1	Reset Controller (RSTC)	«void RSTC_Handler()»
RTC_IRQn	2	Real Time Clock (RTC)	«void RTC_Handler()»
RTT_IRQn	3	Real Time Timer (RTT)	«void RTT_Handler()»
WDT_IRQn	4	Watchdog Timer (WDT)	«void WDT_Handler()»
PMC_IRQn	5	Power Management Controller (PMC)	«void PMC_Handler()»
EFC0_IRQn	6	Enhanced Flash Controller 0 (EFC0)	«void EFC0_Handler()»
EFC1_IRQn	7	Enhanced Flash Controller 1 (EFC1)	«void EFC1_Handler()»
UART_IRQn	8	Universal Asynchronous Receiver Transmitter	«void UART_Handler()»
SMC_IRQn	9	Static Memory Controller (SMC)	«void SMC_Handler()»
PIOA_IRQn	11	Parallel I/O Controller A, (PIOA)	«void PIOA_Handler()»
PIOB_IRQn	12	Parallel I/O Controller B (PIOB)	«void PIOB_Handler()»
PIOC_IRQn	13	Parallel I/O Controller C (PIOC)	«void PIOC_Handler()»
PIOD_IRQn	14	Parallel I/O Controller D (PIOD)	«void PIOD_Handler()»

Cuadro A.13: IRQs del ATSAM3X8E y sus rutinas de tratamiento asociadas. Parte I

```
11 void PIOB_Handler() {
12         /* Código específico del tratamiento */
13 }
```

AA_RTI_PIOB.s

```
1 @ RTI en ensamblador
2 PIOB_Handler:
3         push {lr}
4         /* Código específico del tratamiento */
5         pop {pc}
```

Símbolo	Núm	Dispositivo	RTI
USART0_IRQn	17	USART 0 (USART0)	«void USART0_Handler()»
USART1_IRQn	18	USART 1 (USART1)	«void USART1_Handler()»
USART2_IRQn	19	USART 2 (USART2)	«void USART2_Handler()»
USART3_IRQn	20	USART 3 (USART3)	«void USART3_Handler()»
HSMCI_IRQn	21	Multimedia Card Interface (HSMCI)	«void HSMCI_Handler()»
TWI0_IRQn	22	Two-Wire Interface 0 (TWI0)	«void TWI0_Handler()»
TWI1_IRQn	23	Two-Wire Interface 1 (TWI1)	«void TWI1_Handler()»
SPI0_IRQn	24	Serial Peripheral Interface (SPI0)	«void SPI0_Handler()»
SSC_IRQn	26	Synchronous Serial Controller (SSC)	«void SSC_Handler()»
TC0_IRQn	27	Timer Counter 0 (TC0)	«void TC0_Handler()»
TC1_IRQn	28	Timer Counter 1 (TC1)	«void TC1_Handler()»
TC2_IRQn	29	Timer Counter 2 (TC2)	«void TC2_Handler()»
TC3_IRQn	30	Timer Counter 3 (TC3)	«void TC3_Handler()»
TC4_IRQn	31	Timer Counter 4 (TC4)	«void TC4_Handler()»
TC5_IRQn	32	Timer Counter 5 (TC5)	«void TC5_Handler()»
TC6_IRQn	33	Timer Counter 6 (TC6)	«void TC6_Handler()»
TC7_IRQn	34	Timer Counter 7 (TC7)	«void TC7_Handler()»
TC8_IRQn	35	Timer Counter 8 (TC8)	«void TC8_Handler()»

Cuadro A.14: IRQs del ATSAM3X8E y sus rutinas de tratamiento asociadas. Parte II

A.7. El controlador de DMA del ATSAM3X8E

El *AHB DMA Controller* (*DMAC*) es el dispositivo controlador de acceso directo a memoria en el ATSAM3X8E. Se trata de un dispositivo con seis canales, con capacidad para almacenamiento intermedio de 8 o 32 bytes —en los canales 3 y 5— y que permite transferencias entre dispositivos y memoria, en cualquier configuración. Cada movimiento de información requiere de la lectura de datos de la fuente a través de los buses correspondientes, su almacenamiento intermedio y su posterior escritura en el destino, lo que requiere siempre dos accesos de transferencia

Símbolo	Núm	Dispositivo	RTI
PWM_IRQn	36	Pulse Width Modulation Controller (PWM)	«void PWM_Handler()»
ADC_IRQn	37	ADC Controller (ADC)	«void ADC_Handler()»
DACC_IRQn	38	DAC Controller (DACC)	«void DACC_Handler()»
DMAC_IRQn	39	DMA Controller (DMAC)	«void DMAC_Handler()»
UOTGHS_IRQn	40	USB OTG High Speed (UOTGHS)	«void UOTGHS_Handler()»
TRNG_IRQn	41	True Random Number Generator (TRNG)	«void TRNG_Handler()»
EMAC_IRQn	42	Ethernet MAC (EMAC)	«void EMAC_Handler()»
CAN0_IRQn	43	CAN Controller 0 (CAN0)	«void CAN0_Handler()»
CAN1_IRQn	44	CAN Controller 1 (CAN1)	«void CAN1_Handler()»

Cuadro A.15: IRQs del ATSAM3X8E y sus rutinas de tratamiento asociadas. Parte III

de datos.

Además de un conjunto de registros de configuración globales, cada canal dispone de seis registros asociados que caracterizan la transacción. Mediante estos registros, además de indicar el dispositivo fuente y destino y las direcciones de datos correspondientes, se pueden programar transacciones múltiples de bloques de datos contiguos o dispersos, tanto en la fuente como en el destino.

El dispositivo, además de gestionar adecuadamente los accesos a los distintos buses y dispositivos, es capaz de generar interrupciones para indicar posibles errores o la finalización de las transacciones de DMA programadas.

Para una información más detallada, fuera del objetivo de esta breve introducción, se puede consultar el *apartado 24. AHB DMA Controller (DMAC)* en la página 349 del manual de Atmel.

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 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 funcionara 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 caracter 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 caracter 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 caracter '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 «`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, éste 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 inicializan 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 «`chr`», «`val1`» y «`val2`» de los tres elementos del vector anterior.

AB_struct.s

```
        .data
        @ Constantes (se definen aquí para organizar la zona de datos,
        @ pero no consumen memoria)
        .equ    nombre, 0
        .equ    ch, 10
        .equ    val1, 12        @ Alineado a multiplo de 4
        .equ    val2, 16
        .equ    pieza, 20       @ El tamaño de la estructura
        @ Datos
vecEj:  .space  3 * pieza       @ Vector de 3 estructuras

        .text
        ldr  r0, =vecEj         @ r0 apunta al inicio del vector
        ldr  r1, ='A'           @ Caracter a poner en chr
        ldr  r2, =1000          @ Número a poner en val1
        ldr  r3, =777           @ Número a poner en val2
        strb r1, [r0, #ch]      @ Se usan las constantes (.equ) para
        str  r2, [r0, #val1]    @  escribir los datos sin tener que
        str  r3, [r0, #val2]    @  memorizar los desplazamientos
        add  r0, #pieza         @ Se pasa al siguiente elemento
        strb r1, [r0, #ch]      @ Y se repite lo mismo
        str  r2, [r0, #val1]
        str  r3, [r0, #val2]
        add  r0, #pieza         @ Tercer elemento...
        strb r1, [r0, #ch]
        str  r2, [r0, #val1]
        str  r3, [r0, #val2]
        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 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*

```
1 if (condición) {
2   //
3   // Código a ejecutar si se cumple la condición
4   //
5 }
```

Y en Python:

```
1 if condición:
2   #
3   # Código a ejecutar si se cumple la condición
```

```
4   #
```

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`».

```
1 if a == b:
2   #
3   # Código a ejecutar si a == b
4   #
```

AB_if_equal.s

```
1         .text
2         cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
3         bne  finsi      @  y se salta si se da la condición contraria
4         @
5         @ Código a ejecutar si a == b
6         @
7 finsi:  @ Resto del programa
8         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`».

```
1 if a < b:
2   #
3   # Código a ejecutar si a < b
4   #
```

AB_if_lessthan.s

```
1         .text
2         cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición,
3         bge  finsi      @  y se salta si se da la condición contraria
4         @
5         @ Código a ejecutar si a < b
6         @
7 finsi:  @ Resto del programa
8         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

```c
if (condición) {
  //
  // Código a ejecutar si se cumple la condición
  //
} else {
  //
  // Código a ejecutar si no se cumple la condición
  //
}
```

Y en Python:

```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:

```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:

```
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:

```
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 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  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 indice 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:

```python
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 indice 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 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.

while

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.

```
cad = "Esto es una cadena\0"
long = 0;
while cad[long] != 0:
  long = long + 1
```

Por su parte, la implementación en ensamblador de ARM sería:

AB_long_cad.s

```
        .data
cad:    .asciz "Esto es una cadena"
        .align 2
long:   .space 4

        .text
        ldr r0, =cad     @ r0 tiene la dir. de inicio de cad
        ldr r1, =0       @ r1 tiene la longitud
while:  ldrb r2, [r0,r1] @ r2 <- cad[long]
        cmp r2, #0
        beq finwh         @ Sale si cad[long] == 0
          add r1, r1, #1   @ long = long + 1
        b while         @ Vuelve al comienzo del bucle
finwh:  ldr r2, =long
        str r1, [r2]    @ Almacena la longitud
        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 éste 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

```
1         .text
2         cmp  r0, r1     @ r0 es a y r1 es b, se evalúa la condición
3         bge  finwhile   @  y si no se da, se sale
4 while:    @ Código a repetir
5         cmp  r0, r1     @ Se vuelve a evaluar la condición y si se da,
6         blt  while      @  se vuelve al comienzo del bucle
7 finwhile: @ Resto del programa
8         wfi
```

Apéndice C

Guía rápida del ensamblador Thumb de ARM

En la siguiente hoja se proporciona 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 >> Shift	N Z C	Rango Shift: 0–31
		`lsr Rd, Rd, Rs`	Rd ← Rd >> $[Rs]_{7:0}$	N Z C	
	Aritmético a la derecha	`asr Rd, Rm, #Shift`	Rd ← Rm ASR Shift	N Z C	Rango Shift: 0–31
		`asr Rd, Rd, Rs`	Rd ← Rd ASR $[Rs]_{7:0}$	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

Bibliografía

[ARM95] ARM Limited: *ARM7TDMI data sheet*, 1995. http://pdf1.alldatasheet.es/datasheet-pdf/view/88658/ETC/ARM7TDMI.html.

[Atm11] Atmel Corporation: *ATmega128: 8-bit Atmel microcontroller with 128KBytes in-system programmable flash*, 2011. http://www.atmel.com/Images/doc2467.pdf.

[Atm12] Atmel Corporation: *AT91SAM ARM-based flash MCU datasheet*, 2012. http://www.alldatasheet.es/datasheet-pdf/pdf/476834/ATMEL/AT91SAM.html.

[BMCCCIFF13] Barrachina Mir, Sergio, Maribel Castillo Catalán, José M Claver Iborra y Juan C Fernández Fernández: *Prácticas de introducción a la arquitectura de computadores con el simulador SPIM*. Pearson Educación, 2013.

[BMCCFL+14] 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/.

[BMFLFFLN15] 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.

[BMFLMA15] 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.

[BMLNMA14] Barrachina Mir, Sergio, Germán León Navarro y José Vicente Martí Avilés: *Conceptos elementales de computadores*, 2014. http://lorca.act.uji.es/docs/conceptos_elementales_de_computadores.pdf.

[Cle99] Clements, Alan: *Selecting a processor for teaching computer architecture*. Microprocessors and Microsystems, 23(5):281–290, 1999, ISSN 0141-9331.

[Cle00] Clements, Alan: *The undergraduate curriculum in computer architecture*. IEEE Micro, 20(3):13–22, 2000, ISSN 0272-1732.

[Cle10] 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.

[Cle14] Clements, Alan: *Computer Organization and Architecture: Themes and Variations. International Edition.* Cengage Learning, 2014.

[HH15] Harris, Sarah y David Harris: *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2015.

[IEE04] 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.

[OI04] O'Sullivan, Dan y Tom Igoe: *Physical computing: sensing and controlling the physical world with computers*. Course Technology Press, 2004.

[PH11] Patterson, David A y John L Hennessy: *Estructura y diseño de computadores: la interfaz software/hardware*. Reverté, 2011.

[Shi13] Shiva, Sajjan G: *Computer Organization, Design, and Architecture*. CRC Press, 2013.

[Tex10] Texas Instruments Incorporated: *Cortex-M3 Instruction Set. Technical User's Manual*, 2010. http://users.ece.utexas.edu/~valvano/EE345M/CortexM3InstructionSet.pdf.