Fundamentos de los Sistemas Telemáticos

Tema 3: Estructura y funcionamiento de procesadores

Gregorio Fernández Fernández

Departamento de Ingeniería de Sistemas Telemáticos
Escuela Técnica Superior de Ingenieros de Telecomunicación
Universidad Politécnica de Madrid

Este documento está específicamente diseñado para servir de material de estudio a los alumnos de la asignatura «Fundamentos de los Sistemas Telemáticos» de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Madrid.

Parte del documento es un autoplagio: casi la mitad de los capítulos 1 y 2 están adaptados de textos publicados en el libro «Curso de Ordenadores. Conceptos básicos de arquitectura y sistemas operativos», 5ª ed., Fundación Rogelio Segovia para el Desarrollo de la Telecomunicaciones, Madrid, 2004.

La licencia cc-by-sa significa que usted es libre de copiar, de distribuir, de hacer obras derivadas y de hacer un uso comercial del documento, con dos condiciones: reconocer el origen citando los datos de la portada, y utilizar la misma licencia.

🄯 DIT-UPM, 2012. Algunos derechos reservados.
Este material se distribuye con la licencia Creative Commons «by–sa» disponible en: http://creativecommons.org/licenses/by-sa/3.0/deed.es

Índice general

1 Terminología y objetivos **1**
- 1.1 Ordenador, procesador, microprocesador, microcontrolador, SoC... 1
- 1.2 Niveles y modelos 2
- 1.3 Arquitectura de ordenadores 5
- 1.4 Objetivos y contenido del Tema 6

2 La máquina de von Neumann y su evolución **7**
- 2.1 Modelo estructural 7
- 2.2 Modelo procesal 11
- 2.3 Modelo funcional 11
- 2.4 Evolución 13
- 2.5 Modelos estructurales 13
- 2.6 Modelos procesales 15
- 2.7 Modelos funcionales 18
- 2.8 Comunicaciones con los periféricos e interrupciones 25
- 2.9 Conclusión 26

3 El procesador BRM **27**
- 3.1 Modelo estructural 28
- 3.2 Modelo procesal 30
- 3.3 Modelo funcional 32
- 3.4 Instrucciones de procesamiento y de movimiento 33
- 3.5 Instrucciones de transferencia de datos 38
- 3.6 Instrucciones de transferencia de control 41
- 3.7 Comunicaciones con los periféricos e interrupciones 42

4 Programacion de BRM **45**
- 4.1 Primer ejemplo 45
- 4.2 Etiquetas y bucles 47
- 4.3 El *pool* de literales 49
- 4.4 Más directivas 51
- 4.5 Menos bifurcaciones 54
- 4.6 Subprogramas 55
- 4.7 Módulos 62
- 4.8 Comunicaciones con los periféricos e interrupciones 64
- 4.9 Software del sistema y software de aplicaciones 69
- 4.10 Dos programas para Android 70

5 Procesadores software y lenguajes interpretados **75**
5.1 Ensambladores 75
5.2 El proceso de montaje 77
5.3 Compiladores e intérpretes 79
5.4 Lenguajes de marcas 83
5.5 Lenguajes de *script* 90

A El simulador ARMSim# y otras herramientas **95**
A.1 Instalación 95
A.2 Uso 96
A.3 Otras herramientas 98

Bibliografía **100**

Capítulo 1

Terminología y objetivos

Conviene aclarar el significado de algunos términos e introducir varios conceptos generales antes de concretar los objetivos de este Tema.

1.1. Ordenador, procesador, microprocesador, microcontrolador, SoC...

Aunque el ordenador sea ya un objeto común en la vida cotidiana, resulta interesante analizar cómo se define en un diccionario. El de la Real Academia Española, en el avance de la 23ª edición, remite una de las acepciones de «ordenador» a «computadora electrónica»:

> *Máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos.*

Estando estas notas dirigidas a lectores de España, utilizaremos siempre el término «**ordenador**»[1], entendiendo, además, que es de naturaleza *digital*.

En otro artículo enmendado para la 23ª edición, se define así «**procesador**»[2]:

> *Unidad funcional de una computadora que se encarga de la búsqueda, interpretación y ejecución de instrucciones.*
> *~ de textos.*
> *Programa para el tratamiento de textos.*

Dos comentarios:

- Se deduce de lo anterior que un ordenador tiene dos partes: una memoria, donde se almacenan programas y datos, y un procesador. Los programas están formados por **instrucciones**, que el procesador va extrayendo de la memoria y ejecutando. En realidad, la mayoría de los ordenadores contienen varios procesadores y varias memorias. A esto hay que añadir los dispositivos periféricos (teclado,

---

[1] En España, en los años 60 se empezó a hablar de «cerebros electrónicos» o «computadoras», pero muy pronto se impuso la traducción del francés, «ordenador», que es el término más habitual, a diferencia de los países americanos de habla castellana, en los que se usa «**computador**» o «**computadora**». También se prefiere «computador» en los círculos académicos españoles: «Arquitectura de Computadores» es el nombre de un «área de conocimiento» oficial y de muchos Departamentos universitarios y títulos de libros. Creemos que tras esa preferencia solo se esconde un afán elitista, por lo que utilizamos el nombre más común.

[2] En la edición actual, la 22ª, se define de este curioso modo: «Unidad central de proceso, formada por uno o dos chips.»

ratón, pantalla, discos...). Los procesadores pueden ser genéricos o especializados (por ejemplo, el procesador incorporado en una tarjeta de vídeo). Al procesador o conjunto de procesadores genéricos se le llama **unidad central de procesamiento** (UCP, o CPU, por las siglas en inglés), como ya habíamos visto en la introducción del Tema 1.

- El procesador de textos es un programa. Pero hay otros tipos de programas que son también procesadores: los que traducen de un lenguaje de programación a otro, los que interpretan el lenguaje HTML para presentar las páginas web en un navegador, los que cifran datos, los que codifican datos de señales acústicas en MP3....

Podemos, pues, hablar de dos tipos de procesadores:

- *Procesador hardware:* Sistema físico que ejecuta programas previamente almacenados en una memoria.
- *Procesador software:* Programa que transforma unos datos de entrada en unos datos de salida.

«**Microprocesador**» tiene una definición fácil[3]: es un procesador integrado en un solo chip. Pero hay que matizar que muchos microprocesadores actuales contienen varios procesadores genéricos (que se llaman *«cores»*) y varias memorias de acceso muy rápido (*«caches»*, ver apartado 2.6).

Hasta aquí, seguramente, está usted pensando en lo que evoca la palabra «ordenador»: un aparato (ya sea portátil, de sobremesa, o más grande, para centros de datos y servidores) que contiene la memoria y la UCP, acompañado de discos y otros sistemas de almacenamiento y de periféricos para comunicación con personas (teclado, ratón, etc.). Pero hay otros periféricos que permiten que un procesador hardware se comunique directamente con el entorno físico, habitualmente a través de sensores y conversores analógico-digitales y de conversores digitales-analógicos y «actuadores». De este modo, los programas pueden controlar, sin intervención humana, aspectos del entorno: detectar la temperatura y actuar sobre un calefactor, detectar la velocidad y actuar sobre un motor, etc. Para estas aplicaciones hay circuitos integrados que incluyen, además de una UCP y alguna cantidad de memoria, conversores y otros dispositivos. Se llaman **microcontroladores** y se encuentran *embebidos* en muchos sistemas: *routers*, discos, automóviles, semáforos, lavadoras, implantes médicos, juguetes...

SoC significa «System on Chip». Conceptualmente no hay mucha diferencia entre un microcontrolador y un SoC, ya que éste integra también en un solo chip uno o varios procesadores, memoria, conversores y otros circuitos para controlar el entorno. Pero un microcontrolador normalmente está diseñado por un fabricante de circuitos integrados que lo pone en el mercado para que pueda ser utilizado en aplicaciones tan diversas como las citadas más arriba, mientras que un SoC generalmente está diseñado por un fabricante de dispositivos para una aplicación concreta. Por ejemplo, varios fabricantes de dispositivos móviles y de tabletas utilizan como «core» el mismo procesador (ARM) y cada uno le añade los elementos adecuados para su dispositivo. Podemos decir que un microcontrolador es un SoC genérico, o que un SoC es un microcontrolador especializado.

1.2. Niveles y modelos

El estudio de los sistemas complejos se puede abordar en distintos niveles de abstracción. En el **nivel de máquina convencional**, un procesador hardware es un sistema capaz de interpretar y ejecutar

[3]De momento, la R.A.E. no ha enmendado la más que obsoleta definición de microprocesador: «Circuito constituido por millares de transistores integrados en un chip, que realiza alguna determinada función de los computadores electrónicos digitales». La barrera de los millares se pasó ya en 1989 (Intel 80486: 1.180.000 transistores); actualmente, el Intel Xeon Westmere-EX de 10 núcleos tiene más de dos millardos: 2.500.000.000.

órdenes, llamadas **instrucciones**, que se expresan en un lenguaje binario, el **lenguaje de máquina**. Así, «1010001100000100» podría ser, para un determinado procesador, una instrucción que le dice «envía un dato al puerto USB». Y, como hemos visto en el Tema 2, los datos también están representados en binario. *Hacemos abstracción* de cómo se materializan físicamente esos «0» y «1»: podría ser que «0» correspondiese a 5 voltios y «1» a -5 voltios, u otros valores cualesquiera. Estos detalles son propios de niveles de abstracción más «bajos», los niveles de **micromáquina**, de **circuito lógico** y de **circuito eléctrico**, que se estudian en la Electrónica digital y la Electrónica analógica.

El lenguaje de máquina es el «lenguaje natural» de los procesadores hardware. Pero a la mente humana le resulta muy difícil interpretarlo. Los lenguajes de programación se han inventado para expresar de manera inteligible los programas. Podríamos inventar un lenguaje en el que la instrucción anterior se expresase así: «`envía dato,#USB`». Este tipo de lenguaje se llama **lenguaje ensamblador**.

Procesadores y niveles de lenguajes

Siguiendo con el ejemplo ficticio, el procesador hardware no puede entender lo que significa «`envía dato,#USB`». Es preciso traducírselo a binario. Esto es lo que hace un **ensamblador**[4], que es un *procesador de lenguaje:* un programa que recibe como datos de entrada una secuencia de expresiones en lenguaje ensamblador y genera el programa en lenguaje de máquina para el procesador.

Al utilizar un lenguaje ensamblador estamos ya estudiando al procesador en un nivel de abstracción más «alto»: hacemos abstraccion de los «0» y «1», y en su lugar utilizamos símbolos. Es el **nivel de máquina simbólica**. Pero los lenguajes ensambladores tienen dos inconvenientes:

- Aunque sean más inteligibles que el binario, programar aplicaciones reales con ellos es una tarea muy laboriosa y propensa a errores.
- Las instrucciones están muy ligadas al procesador: si, tras miles de horas de trabajo, consigue usted hacer un programa para jugar a *StarCraft* en un ordenador personal (que tiene un procesador Intel o AMD) y quiere que funcione también en un dispositivo móvil (que tiene un procesador ARM) tendría que empezar casi de cero y emplear casi las mismas horas programando en el ensamblador del ARM.

Por eso se han inventado los **lenguajes de alto nivel**. El desarrollo de ese programa *StarCraft* en C, C#, Java u otro lenguaje sería independiente del procesador hardware que finalmente ha de ejecutar el programa, y el esfuerzo de desarrollo sería varios órdenes de magnitud más pequeño. «Alto» y «bajo» nivel de los lenguajes son, realmente, *subniveles* del nivel de máquina simbólica.

Dado un lenguaje de alto nivel, para cada procesador hardware es preciso disponer de un procesador software que *traduzca* los programas escritos en ese lenguaje al lenguaje de máquina del procesador. Estos procesadores de lenguajes de alto nivel se llaman **compiladores**, y, aunque su función es similar a la de los ensambladores (traducir de un lenguaje simbólico a binario) son, naturalmente, mucho más complejos.

Si los componentes básicos de un lenguaje de bajo nivel son las **instrucciones**, los de un lenguaje de alto nivel son las **sentencias**. Una sentencia típica del lenguaje C (y también de muchos otros) es la de asignación: «`x = x + y`» significa «calcula la suma de los valores de las variables `x` e `x` y el resultado asígnalo a la variable `x` (borrando su valor previo)». Un compilador de C para el procesador hardware X traducirá esta sentencia a una o, normalmente, varias instrucciones específicas del procesador X.

Hay otro nivel de abstracción, intermedio entre los de máquina convencional y máquina simbólica. Es el **nivel de máquina operativa**. A la máquina convencional se le añade un conjunto de programas

[4] «Ensamblador» es un término ambiguo: se refiere al lenguaje y al procesador (capítulo 4).

que facilitan el uso de los recursos (las memorias, los procesadores y los periféricos), ofreciendo **servicios** al nivel de máquina simbólica. Este es el nivel estudiado (aunque muy superficialmente) en el Tema 1.

Modelos funcionales, estructurales y procesales

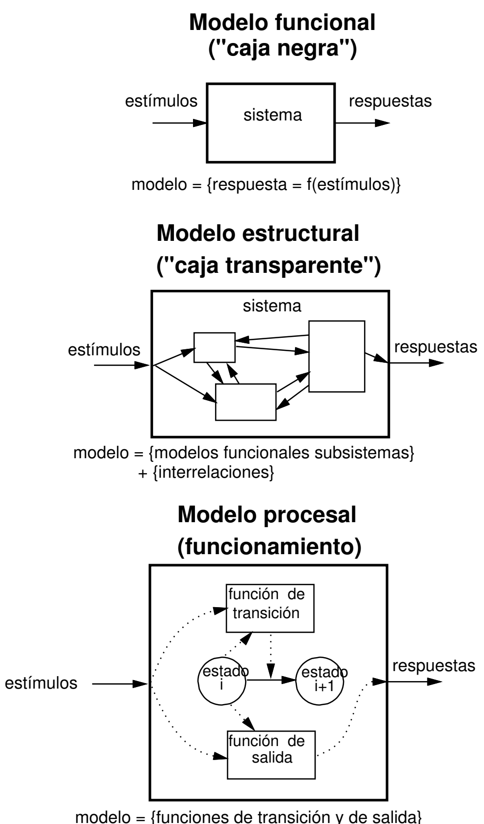


Figura 1.1 Tipos de modelos.

Un modelo es una representación de un sistema en la que se hace abstracción de los detalles irrelevantes. Al estudiar un sistema en un determinado nivel de abstracción nos podemos interesar en unos aspectos u otros, y, por tanto, utilizar distintos modelos (figura 1.1). Si lo que nos interesa es cómo se hace uso del sistema, no es necesario que entremos en su composición interna: describimos la forma que tiene de responder a los diferentes estímulos. Esta descripción es un **modelo funcional** del sistema. Pero para estudiarlo más a fondo hay que describir las partes que lo forman y cómo se relacionan, es decir, un **modelo estructural**, y también su funcionamiento, un **modelo procesal** que explique los estados en los que puede encontrarse el sistema y las transiciones de uno a otro a lo largo del tiempo. Según sean los objetivos que se pretenden, las descripciones combinan los tres tipos de modelos de forma más o menos detallada.

Un ejemplo:

Imagénese usted viajando a una lejana galaxia y desembarcando en un acogedor planeta que alberga a una civilización primitiva. Consigue comunicarse y hacer amistad con los nativos y, tratando de contarles la forma de vida terrestre, llega un momento en que les habla de automóviles. Para empezar, definirá de qué se trata con un *modelo funcional* muy básico: un artefacto que, obedeciendo ciertas manipulaciones, permite desplazarse sin esfuerzo. Ante la incredulidad de su audiencia, les explicará que tiene ruedas, un motor, un habitáculo... es decir, un *modelo estructural*. Y cuando le pregunten cómo funciona todo eso habrá de esforzarse para que entiendan que la energía de combustión se transforma en mecánica, que la fuerza generada por el motor se transmite a las ruedas...: un *modelo procesal* rudimentario.

Regresa usted a la tierra acompañado de su mejor amigo, que, fascinado ante la majestuosidad de un Opel Corsa, quiere aprender a manejarlo. Le describirá mejor el *modelo estructural* (volante, pedales...), y luego le proporcionará, mediante la experiencia, un *modelo funcional* mucho más completo: cómo hay que actuar sobre los distintos elementos para conseguir el comportamiento deseado. Pero su amigo alienígena, ya con el permiso de conducir, resulta ser un individuo inquieto y ávido de conocimientos y quiere convertirse en un mecánico. Entre otras cosas, necesitará *modelos estructurales y procesales* mucho más detallados que, seguramente, exceden ya sus competencias, y tendrá que matricularlo en un módulo de FP.

Pero volvamos a la cruda realidad de los procesadores hardware.

En el nivel de máquina convencional, el modelo estructural es lo que habitualmente se llama «**diagrama de bloques**»: un dibujo en el que se representan, a grandes rasgos y sin entrar en el detalle de los circuitos, los componentes del sistema y sus relaciones. Si el sistema es un ordenador, el modelo incluirá también la memoria y los periféricos. Si es un SoC, los componentes asociados al procesador (conversores, sensores, amplificadores, antenas...). En el Tema 1 se han presentado algunos modelos estructurales muy genéricos. Avanzando unas páginas, en las figuras 2.1, 2.7, 2.8 y 3.1 puede usted ver otros más concretos.

El modelo funcional está compuesto por la descripción de los convenios de representación binaria de los tipos de datos (enteros, reales, etc.) que puede manejar el procesador y por los **formatos** y el **repertorio** de instrucciones. El repertorio es la colección de todas las instrucciones que el procesador es capaz de entender, y los formatos son los convenios de su representación. El modelo funcional, en definitiva, incluye todo lo que hay que saber para poder programar el procesador en su lenguaje de máquina binario. Es la información que normalmente proporciona el fabricante del procesador en un documento que se llama «**ISA**» (Instruction Set Architecture).

El modelo procesal en el nivel de máquina convencional consiste en explicar las acciones que el procesador realiza en el proceso de lectura, interpretación y ejecución de instrucciones. Como ocurre con el modelo estructural, para entender el nivel de máquina convencional solamente es necesaria una comprensión somera de este proceso. Los detalles serían necesarios si pretendiésemos diseñar el procesador, en cuyo caso deberíamos descender a niveles de abstracción propios de la electrónica.

1.3. Arquitectura de ordenadores

En el campo de los ordenadores, la palabra «arquitectura» tiene tres significados distintos:

- En un sentido que es *no contable*, la arquitectura de ordenadores es una especialidad de la ingeniería en la que se identifican las necesidades a cubrir por un sistema basado en ordenador, se consideran las restricciones de tipo tecnológico y económico, y se elaboran modelos funcionales, estructurales y procesales en los niveles de máquina convencional y micromáquina, sin olvidar los niveles de máquina operativa y máquina simbólica, a los que tienen que dar soporte, y los de circuito lógico y circuito eléctrico, que determinan lo que tecnológicamente es posible y lo que no lo es. Es éste el significado que se sobreentiende en un curso o en un texto sobre «arquitectura de ordenadores»: un conjunto de *conocimientos y habilidades* sobre el diseño de ordenadores.

  La arquitectura es el «arte de proyectar y construir edificios» habitables. Los edificios son los ordenadores, y los habitantes, los programas. En la última edición, la R.A.E. añadió la acepción informática: «estructura lógica y física de los componentes de un computador». Pero esta definición tiene otro sentido, un sentido contable.

- En el sentido *contable*, la arquitectura *de un ordenador*, o *de un procesador*, es su *modelo funcional en el nivel de máquina convencional*. Es en este sentido en el que se habla, por ejemplo, de la «arquitectura x86» para referirse al *modelo funcional* común a una cierta familia de microprocesadores. En tal modelo, los detalles del hardware, transparentes al programador, son irrelevantes. Es lo que en el apartado anterior hemos llamado «**ISA**» (Instruction Set Architecture).

- Finalmente, es frecuente utilizar el término «arquitectura» (también en un sentido contable) para referirse a un *modelo estructural*, especialmente en los sistemas con muchos procesadores y periféricos y en sistemas de software complejos.

Arquitectura, implementación y realización

Hay tres términos, «arquitectura», «implementación» y «realización», que se utilizan con frecuencia y que guardan relación con los niveles de abstracción. La **arquitectura** (en el segundo de los sentidos) de un procesador define los elementos y las funciones que debe conocer el programador. La **implementación** entra en los detalles de la estructura y el funcionamiento internos (componentes y conexiones, organización de los flujos de datos e instrucciones, procesos que tienen lugar en el procesador para controlar a los componentes de la estructura e interpretar la arquitectura), y la **realización** se ocupa de los circuitos lógicos, la interconexión y cableado, y la tecnología adoptada.

De acuerdo con la jerarquía de niveles de abstracción, la arquitectura (ISA) corresponde al modelo funcional en el nivel de máquina convencional, la implementación a los modelos estructural y procesal más detallados (se llama «nivel de micromáquina»), y la realización corresponde a los modelos en los niveles de circuito lógico e inferiores.

1.4. Objetivos y contenido del Tema

Según la Guía de aprendizaje de la asignatura, los principales resultados de aprendizaje de este Tema han de ser «conocer los principios básicos de la arquitectura de ordenadores, comprender el funcionamiento de los procesadores en el nivel de máquina convencional, conocer los niveles y tipos de lenguajes de programación, conocer los procesadores de lenguajes, programar en un lenguaje de marcas, conocer los distintos tipos de software».

Empezaremos con un capítulo en el que se describen los conceptos básicos del nivel de máquina convencional, tomando como base un documento histórico. Hay que dejar bien claro que ninguna persona en su sano juicio trabaja estrictamente en el nivel de máquina convencional (es decir, programando un procesador en su lenguaje de máquina binario). Normalmente, la programación se hace en lenguajes de alto nivel y, raramente, en ensamblador. El objetivo de estudiar el nivel de máquina convencional es el de comprender los procesos que realmente tienen lugar cuando se ejecutan los programas. De la misma manera que un piloto de Fórmula 1 no va a diseñar motores, pero debe comprender su funcionamiento para obtener de ellos el máximo rendimiento.

Los conceptos generales solo se asimilan completamente cuando se explican sobre un procesador concreto. En los capítulos siguientes veremos uno, al que llamamos (veremos por qué) «BRM», y su programación. Luego, ya brevemente, estudiaremos los principios de algunos procesadores software y de algunos lenguajes de marcas y de *script*.

En un apéndice se dan las instrucciones para instalar y utilizar un simulador con el que practicar la programación de BRM. También se sugieren otras herramientas más avanzadas por si a usted le gusta tanto este Tema que está interesado en explorarlo más a fondo.

Capítulo 2

La máquina de von Neumann y su evolución

Un hito importante en la historia de los ordenadores fue la introducción del concepto de **programa almacenado**: previamente a su ejecución, las instrucciones que forman el programa deben estar guardadas, o *almacenadas*, en una memoria. También se dice que el programa tiene que estar **cargado** en la memoria. Esto condiciona fuertemente los modelos estructurales, funcionales y procesales de los procesadores en todos los niveles.

Modelos estructurales que contenían unidades de memoria, de procesamiento, de control y de entrada/salida, se habían propuesto tiempo atrás, pero la idea de programa almacenado y el modelo procesal que acompaña a esta idea, se atribuyen generalmente a John von Neumann. Hay un documento escrito en 1946 por Burks, Goldstine y von Neumann (cuando aún no se había construido ninguna máquina de programa almacenado) que mantiene hoy toda su vigencia conceptual. Haciendo abstracción de los detalles ligados a las tecnologías de implementación y de realización, esa descripción no ha sido superada, pese a que seguramente se habrán publicado decenas de miles de páginas explicando lo mismo. Por eso, nos serviremos de ese texto, extrayendo y glosando algunos de sus párrafos.

2.1. Modelo estructural

«... Puesto que el dispositivo final ha de ser una máquina computadora de propósito general, deberá contener ciertos órganos fundamentales relacionados con la aritmética, la memoria de almacenamiento, el control y la comunicación con el operador humano...».

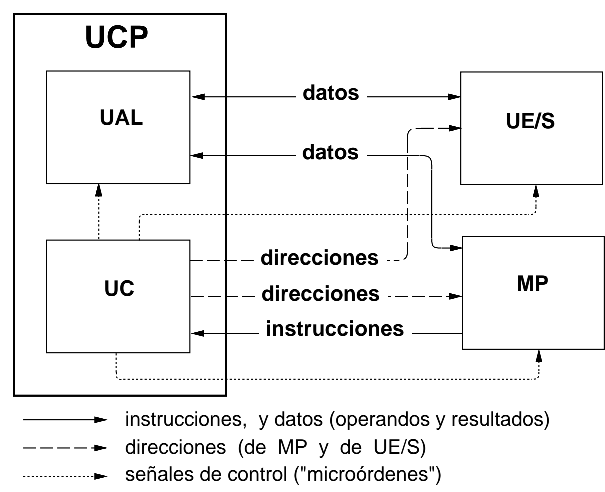


Figura 2.1 La «máquina de von Neumann».

Aquí el documento introduce el *modelo estructural básico* en el nivel de máquina convencional. Los «órganos» son los subsistemas denominados «UAL» (unidad aritmética y lógica), «MP» (memoria principal), «UC» (unidad de control) y «UE/S» (unidades de entrada y salida) en el diagrama de la figura 2.1. Al conjunto de la

UC y la UAL se le llama «UCP» (**unidad central de procesamiento**, o **procesador central**). Como hemos dicho en el capítulo anterior, la tecnología electrónica actual permite encapsular uno o varios procesadores en un chip, resultando un microprocesador.

Programa almacenado y propósito general

> *«... La máquina debe ser capaz de almacenar no solo la información digital necesaria en una determinada computación* [...] *sino también las instrucciones que gobiernen la rutina a realizar sobre los datos numéricos. En una máquina de propósito especial, estas instrucciones son un componente integrante del dispositivo y constituyen parte de su estructura de diseño. Para que la máquina sea de propósito general, debe ser posible instruirla de modo que pueda llevar a cabo cualquier computación formulada en términos numéricos. Por tanto, debe existir algún órgano capaz de almacenar esas órdenes de programa...»*

En el escrito se utilizan indistintamente, y con el mismo significado, «instrucciones», «órdenes» y «órdenes de programa». Actualmente se habla siempre de **instrucciones**. El conjunto de instrucciones diferentes que puede ejecutar el procesador es el **juego** o **repertorio de instrucciones**.

Esbozado el modelo estructural en el nivel de máquina convencional, es preciso *describir sus subsistemas mediante modelos funcionales*.

Memoria

> *«... Hemos diferenciado, conceptualmente, dos formas diferentes de memoria: almacenamiento de datos y almacenamiento de órdenes. No obstante, si las órdenes dadas a la máquina se reducen a un código numérico, y si la máquina puede distinguir de alguna manera un número de una orden, el órgano de memoria puede utilizarse para almacenar tanto números como órdenes.»*

Es decir, en el subsistema de memoria se almacenan tanto las instrucciones que forman un programa como los datos. Esto es lo que luego se ha llamado «arquitectura Princeton». En ciertos diseños se utiliza una memoria para datos y otra para instrucciones, siguiendo una «arquitectura Harvard».

> *«... Planeamos una facilidad de almacenamiento electrónico completamente automático de unos* 4.000 *números de cuarenta dígitos binarios cada uno. Esto corresponde a una precisión de* $2^{-40} \approx 0,9 \times 10^{-12}$*, es decir, unos doce decimales. Creemos que esta capacidad es superior en un factor de diez a la requerida para la mayoría de los problemas que abordamos actualmente... Proponemos además una memoria subsidiaria de mucha mayor capacidad, también automática, en algún medio como cinta o hilo magnético.»*

Y en otro lugar dice:

> *«... Como la memoria va a tener* $2^{12} = 4.096$ *palabras de cuarenta dígitos* [...] *un número binario de doce dígitos es suficiente para identificar a una posición de palabra...»*

Por tanto, las **direcciones** (números que identifican a las posiciones) eran de doce bits, y, como las palabras identificadas por esas direcciones tenían cuarenta bits, la capacidad de la MP, expresada en unidades actuales, era $40/8 \times 2^{12} = 5 \times 4 \times 2^{10}$ bytes, es decir, 20 KiB. Obviamente, los problemas «actuales» a los que se dirigía esa máquina no eran los que se resuelven con los procesadores de hoy... Se habla de una *memoria secundaria* de «cinta o hilo». La cinta y el hilo son reliquias históricas; las memorias secundarias actuales son de discos o de estado sólido (memorias *flash*).

> *«... Lo ideal sería [...] que cualquier conjunto de cuarenta dígitos binarios, o palabra, fuese accesible inmediatamente –es decir, en un tiempo considerablemente inferior al tiempo de operación de un multiplicador electrónico rápido–*[...] *De aquí que el tiempo de disponibilidad de una palabra de la memoria*

debería ser de 5 a 50 µseg. Asimismo, sería deseable que las palabras pudiesen ser sustituidas por otras nuevas a la misma velocidad aproximadamente. No parece que físicamente sea posible lograr tal capacidad. Por tanto, nos vemos obligados a reconocer la posibilidad de construir una jerarquía de memorias, en la que cada una de las memorias tenga una mayor capacidad que la precedente pero menor rapidez de acceso...»

La velocidad también se ha multiplicado por varios órdenes de magnitud: el tiempo de acceso en las memorias de semiconductores (con capacidades de millones de bytes), es alrededor de una milésima parte de esos 5 a 50 μs que «no parece que sea posible lograr».

Aquí aparece una característica esencial de la memoria principal: la de ser de **acceso aleatorio** («sería deseable que las palabras pudiesen ser sustituidas por otras nuevas a la misma velocidad»), así como el concepto de *jerarquía de memorias*.

La figura 2.2 ilustra que para extraer una palabra de la memoria (operación de **lectura**) se da su dirección y la señal («microorden») **lec**; tras un *tiempo de acceso para la lectura* se tiene en la salida el contenido de esa posición. Para introducir una palabra en una posición (operación de **escritura**), se indica también la dirección y la microorden **esc**; tras un *tiempo de acceso para la escritura* queda la palabra grabada. Como vimos en el Tema 2 (apartado 1.3), decir que la memoria tiene acceso **aleatorio** (o que es una **RAM**) significa simplemente que los tiempos de acceso (los que transcurren desde que se le da una microorden de lectura o escritura hasta que la operación ha concluido) son independientes de la dirección. Normalmente, ambos (lectura y escritura) son iguales.

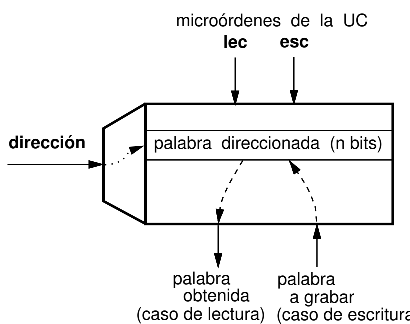


Figura 2.2. Lectura y escritura en una RAM.

Dos propiedades importantes de la MP son:

- Una posición nunca puede estar «vacía»: siempre tiene un contenido: una palabra formada por un conjunto de bits («ceros» y «unos»).
- La operación de lectura no es «destructiva»: el contenido leído permanece tal como estaba previamente en la celda; la escritura sí lo es: el contenido anterior desaparece y queda sustituido por el nuevo (sin embargo, algunas partes de la MP pueden ser de solo lectura, es decir, ROM, y su contenido no puede modificarse).

Unidad aritmética y lógica

«... Puesto que el dispositivo va a ser una máquina computadora, ha de contener un órgano que pueda realizar ciertas operaciones aritméticas elementales. Por tanto, habrá una unidad capaz de sumar, restar, multiplicar y dividir...»

«... Las operaciones que la máquina considerará como elementales serán, evidentemente, aquellas que se le hayan cableado. Por ejemplo, la operación de multiplicación podrá eliminarse del dispositivo como proceso elemental si la consideramos como una sucesión de sumas correctamente ordenada. Similares observaciones se pueden aplicar a la división. En general, la economía de la unidad aritmética queda determinada por una solución equilibrada entre el deseo de que la máquina sea rápida – una operación no elemental tardará normalmente mucho tiempo en ejecutarse, al estar formada por una serie de órdenes dadas por el control – y el deseo de hacer una máquina sencilla, o de reducir su coste...»

Como su nombre indica, la UAL incluye también las operaciones de tipo lógico: negación («NOT»), conjunción («AND»), disyunción («OR»), etc.

Al final de la cita aparece un ejemplo de la *disyuntiva hardware/software*, habitual en el diseño de los procesadores: muchas funciones pueden realizarse indistintamente de manera *cableada* (por hardware) o de manera *programada* (por software). La elección depende de que predomine el deseo de reducir el coste (soluciones software) o de conseguir una máquina más rápida (soluciones hardware).

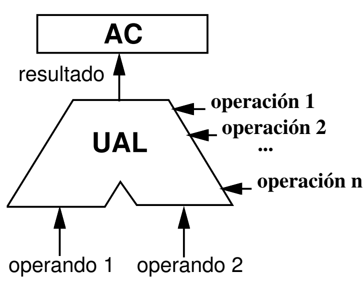


Figura 2.3. Diagrama de la UAL.

En resumen, la unidad aritmética y lógica, o UAL (o ALU, «Arithmetic and Logic Unit»), es un subsistema que puede tomar dos operandos (o solo uno, como en el caso de «NOT») y generar el resultado correspondiente a la operación que se le indique, de entre un conjunto de operaciones previstas en su diseño. En lo sucesivo seguiremos el convenio de representar la UAL por el diagrama de la figura 2.3. El resultado, en este modelo primitivo, se introducía en un **registro acumulador** (AC). Se trata de una memoria de acceso mucho más rápido que la MP, pero que solo puede albergar una palabra de 40 bits.

Unidades de entrada y salida

> *«... Por último, tiene que haber dispositivos, que constituyen el órgano de entrada y de salida, mediante los cuales el operador y la máquina puedan comunicarse entre sí* [...] *Este órgano puede considerarse como una forma secundaria de memoria automática...»*

Las unidades de entrada y salida, o *dispositivos periféricos* (que, abreviadamente, se suelen llamar **periféricos**), representadas en la figura 2.1 como un simple bloque (UE/S), en aquella época eran muy básicas: simples terminales electromecánicos. Actualmente son muchas y variadas. Periféricos son los que permiten la comunicación con las personas (teclado, pantalla, ratón, micrófono, altavoz...), pero también las memorias secundarias, los puertos de comunicaciones, y los sensores y actuadores para la interacción con el entorno.

Unidad de control

> *«... Si la memoria para órdenes es simplemente un órgano de almacenamiento, tiene que haber otro órgano que pueda ejecutar automáticamente las órdenes almacenadas en la memoria. A este órgano le llamaremos el control ...»*

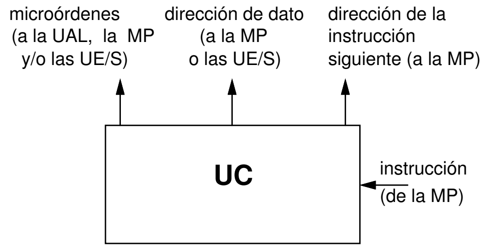


Figura 2.4 Unidad de control.

La UC examina cada instrucción («orden») y la *ejecuta* emitiendo las señales de control, llamadas **microórdenes**, adecuadas para activar a las otras unidades y, si es necesario, las direcciones de MP o de entrada/salida oportunas (figura 2.4).

> *«... Tiene que ser posible extraer números de cualquier parte de la memoria en cualquier momento. Sin embargo, en el caso de las órdenes, el tratamiento puede ser más metódico, puesto que las instrucciones se pueden poner, por lo menos parcialmente, en secuencia lineal. En consecuencia, el control se construirá de forma que normalmente proceda de la posición* n *de memoria a la posición* (n+1) *para su siguiente instrucción...»*

Tras la ejecución de una instrucción que está almacenada en la palabra de dirección d de la memoria, la siguiente a ejecutar es, normalmente, la almacenada en la dirección $d + 1$ (las excepciones son las instrucciones de bifurcación, que explicaremos enseguida).

2.2. Modelo procesal

El documento detalla informalmente el funcionamiento al describir la unidad de control. Interpretándolo en términos más actuales, para cada instrucción han de completarse, sucesivamente, tres fases:

1. La fase de **lectura de instrucción** consiste en leer de la MP la instrucción e introducirla en la UC.
2. En la fase de **descodificación** la UC analiza los bits que la componen y «entiende» lo que «pide» la instrucción.
3. En la fase de **ejecución** la UC genera las microórdenes para que las demás unidades completen la operación. Esta fase puede requerir un nuevo acceso a la MP para extraer un dato o para escribirlo.

Concretando un poco más, la figura 2.5 presenta el modelo procesal. «CP» es el **contador de programa**, un registro de doce bits dentro de la UC que contiene la dirección de la instrucción siguiente. Hay un estado inicial en el que se introduce en CP la dirección de la primera instrucción. «0 → CP» significa «introducir el valor 0 (doce ceros en binario) en el registro CP» (suponemos que, al arrancar la máquina, el programa ya está cargado a partir de la dirección 0 de la MP). Después se pasa cíclicamente por los tres estados. En la lectura de instrucción «(MP[(CP)]) → UC» significa «transferir la palabra de dirección indicada por CP a la unidad de control», y «(CP)+1 → CP», «incrementar en una unidad el contenido del contador de programa».

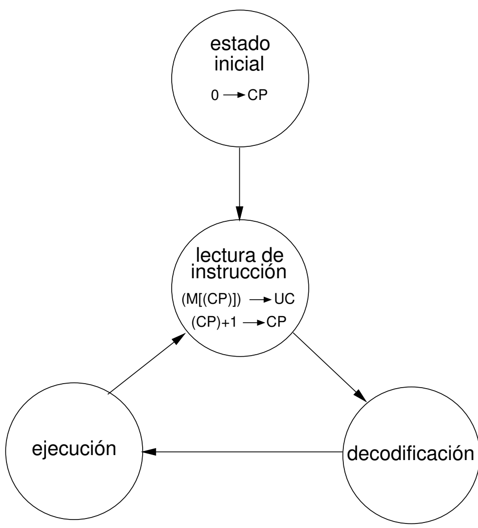


Figura 2.5 Modelo procesal.

Las **instrucciones de bifurcación** son aquellas que le dicen a la unidad de control que la siguiente instrucción a ejecutar no es la que está a continuación de la actual, sino la que se encuentra en la dirección indicada en el campo CD. En la fase de ejecución de estas instrucciones se introduce un nuevo valor en el registro CP.

Este modelo procesal está un poco simplificado con respecto al que realmente debía seguir la máquina propuesta. En efecto, como veremos ahora, cada palabra de cuarenta bits contenía no una, sino dos instrucciones, por lo que normalmente solo hacía falta una lectura para ejecutar dos instrucciones seguidas.

2.3. Modelo funcional

El *modelo funcional* es lo que se necesita conocer para usar (programar) la máquina: los convenios para la representación de números e instrucciones.

Representación de números

> *«... Al considerar los órganos de cálculo de una máquina computadora nos vemos naturalmente obligados a pensar en el sistema de numeración que debemos adoptar. Pese a la tradición establecida de construir máquinas digitales con el sistema decimal, para la nuestra nos sentimos más inclinados por el sistema binario...»*

Luego de unas consideraciones para justificar esta elección, muy ligadas a la tecnología de la época (aunque podrían aplicarse también, en esencia, a la tecnología actual), continúa con las ventajas del sistema binario:

«... La principal virtud del sistema binario frente al decimal radica en la mayor sencillez y velocidad con que pueden realizarse las operaciones elementales...»

«... Un punto adicional que merece resaltarse es éste: una parte importante de la máquina no es de naturaleza aritmética, sino lógica. Ahora bien, la lógica, al ser un sistema del sí y del no, es fundamentalmente binaria. Por tanto, una disposición binaria de los órganos aritméticos contribuye de manera importante a conseguir una máquina más homogénea, que puede integrarse mejor y ser más eficiente...»

«... El único inconveniente del sistema binario desde el punto de vista humano es el problema de la conversión. Sin embargo, como se conoce perfectamente la manera de convertir números de una base a otra, y puesto que esta conversión puede efectuarse totalmente mediante la utilización de los procesos aritméticos habituales, no hay ninguna razón para que el mismo computador no pueda llevar a cabo tal conversión.»

Formato de instrucciones

«... Como la memoria va a tener $2^{12} = 4.096$ *palabras de cuarenta dígitos* [...] *un número binario de doce dígitos es suficiente para identificar a una posición de palabra...»*

«... Dado que la mayoría de las operaciones del computador hacen referencia al menos a un número en una posición de la memoria, es razonable adoptar un código en el que doce dígitos binarios de cada orden se asignan a la especificación de una posición de memoria. En aquellas órdenes que no precisan extraer o introducir un número en la memoria, esas posiciones de dígitos no se tendrán en cuenta...»

«... Aunque aún no está definitivamente decidido cuántas operaciones se incorporarán (es decir, cuántas órdenes diferentes debe ser capaz de comprender el control), consideraremos por ahora que probablemente serán más de 2^5*, pero menos de* 2^6*. Por esta razón, es factible asignar seis dígitos binarios para el código de orden. Resulta así que cada orden debe contener dieciocho dígitos binarios, los doce primeros para identificar una posición de memoria y los seis restantes para especificar una operación. Ahora podemos explicar por qué las órdenes se almacenan en la memoria por parejas. Como en este computador se va a utilizar el mismo órgano de memoria para órdenes y para números, es conveniente que ambos tengan la misma longitud. Pero números de dieciocho dígitos binarios no serían suficientemente precisos para los problemas que esta máquina ha de resolver [...] De aquí que sea preferible hacer las palabras suficientemente largas para acomodar dos órdenes...»*

«... Nuestros números van a tener cuarenta dígitos binarios cada uno. Esto permite que cada orden tenga veinte dígitos binarios: los doce que especifican una posición de memoria y ocho más que especifican una operación (en lugar del mínimo de seis a que nos referíamos más arriba)...»

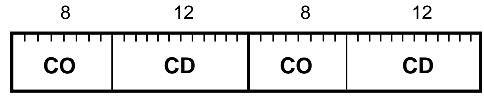


Figura 2.6 Formato de instrucciones.

O sea, el **formato de instrucciones** propuesto es el que indica la figura 2.6, con dos instrucciones en cada palabra. En muchos diseños posteriores (CISC, ver apartado 2.7) se ha seguido más bien la opción contraria: palabras relativamente cortas e instrucciones que pueden ocupar una o más palabras, y lo mismo para los números.

En el formato de la figura 2.6 hay dos **campos** para cada una de las instrucciones: uno de ocho bits, «CO», que indica de qué instrucción se trata (el **código de operación**), y otro de doce bits, CD, que contiene la **dirección** de la MP a la que **hace referencia** la instrucción.

Por ejemplo, tras leer y descodificar una instrucción que tuviese el código de operación correspondiente a «sumar» y 0b000000101111 = 0x02F = 47 en el campo CD, la unidad de control, ya en la fase de ejecución, generaría primero las microórdenes oportunas para leer el contenido de la dirección 47 de la MP, luego llevaría este contenido a una de las entradas de la UAL y el contenido actual del acumulador a la otra, generaría la microorden de suma para la UAL y el resultado lo introduciría en el acumulador.

Programas

> *«... La utilidad de un computador automático radica en la posibilidad de utilizar repetidamente una secuencia determinada de instrucciones, siendo el número de veces que se repite o bien previamente determinado o bien dependiente de los resultados de la computación. Cuando se completa cada repetición hay que seguir una u otra secuencia de órdenes, por lo que en la mayoría de los casos tenemos que especificar dos secuencias distintas de órdenes, precedidas por una instrucción que indique la secuencia a seguir. Esta elección puede depender del signo de un número [...] En consecuencia, introducimos una orden (la* orden de transferencia condicionada*) que, dependiendo del signo de un número determinado, hará que se ejecute la rutina apropiada de entre las dos...»*

Es decir, los programas constan de una secuencia de instrucciones que se almacenan en direcciones consecutivas de la MP. Normalmente, tras la ejecución de una instrucción se pasa a la siguiente. Pero en ocasiones hay que pasar no a la instrucción almacenada en la dirección siguiente, sino a otra, almacenada en otra dirección. Para poder hacer tal cosa están las **instrucciones de transferencia de control**, o **de bifurcación** (éstas son un caso particular de las primeras, como veremos enseguida).

Para completar el modelo funcional sería preciso especificar todas las instrucciones, explicando para cada una su código de operación y su significado. Esto es lo que haremos en el capítulo 3, ya con un procesador concreto y moderno.

2.4. Evolución

En el año 1946 la tecnología electrónica disponible era de válvulas de vacío. El transistor se inventó en 1947, el primer circuito integrado se desarrolló en 1959 y el primer microprocesador apareció en 1971. Y la **ley de Moore**, enunciada en 1965 («el número de transistores en un circuito integrado se duplica cada dos años»)[1], se ha venido cumpliendo con precisión sorprendente.

Ese es un brevísimo resumen de la evolución en los niveles inferiores al de máquina convencional, que le habrán explicado con más detalle en la asignatura «Introducción a la ingeniería». Pero también en este nivel se han ido produciendo innovaciones. Veamos, de manera general, las más importantes.

2.5. Modelos estructurales

La figura 2.7 es un modelo estructural de un sistema monoprocesador muy básico, pero más actual que el de la figura 2.1. Se observan varias diferencias:

[1] Inicialmente, Gordon Moore estimó el período en un año, pero luego, en 1975, lo amplió a dos. Si se tiene en cuenta que el primer microprocesador, el 4004 de Intel, tenía 2.300 transistores, la ley predice que en 2011, cuarenta años después, se integrarían $2.300\times2^{20} \approx 2.400$ millones. El Intel Xeon Westmere-EX de 10 núcleos tiene más de 2.500 millones.

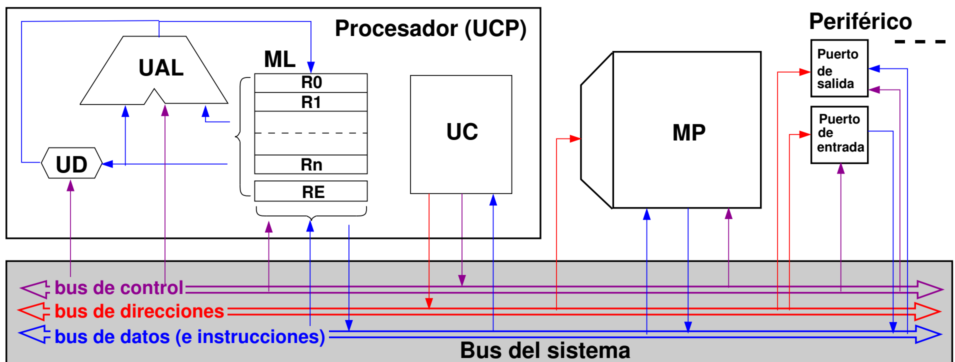


Figura 2.7 Modelo estructural de un sistema basado en procesador

- En lugar de un solo registro acumulador (figura 2.3), hay un conjunto de registros, R0, R1...Rn, que forman una **memoria local** (ML). Esta memoria es muy rápida pero de muy poca capacidad, en relación con la de la MP: como mucho tiene 64 o 128 registros (y en cada registro se puede almacenar una sola palabra).
- Hay un registro especial, el **registro de estado** (RE), que, entre otras cosas, contiene los indicadores Z, N, C y V, cuya función hemos visto en el Tema 2 (apartado 3.3), y uno o varios bits que indican el **modo** en que se encuentra el procesador: usuario, supervisor, etc.
- Hay un nuevo componente, la **unidad de desplazamiento** (UD) que permite realizar sobre cualquier registro las operaciones de desplazamientos y rotaciones que también hemos estudiado en el apartado 3.3 del Tema 2.
- Los datos y las instrucciones se transfieren de una unidad a otra a través de buses. Un **bus** es un conjunto de «líneas», conductores eléctricos (cables o pistas o de un circuito integrado) al que pueden conectarse varias unidades. El ancho del bus es su número de líneas[2]. Se trata de un recurso compartido: en cada momento, solo una de las unidades puede depositar un conjunto de bits en el bus, pero varias pueden leerlo simultáneamente.
- Los periféricos se conectan al bus a través de registros incluidos en los controladores, que se llaman **puertos**.

Solo es cuestión de convenio considerar que hay un único bus, el **bus del sistema** (la zona sombreada) formado por tres subuses, o que realmente hay tres buses:

- El **bus de direcciones** transporta direcciones de la MP, o de los puertos, generadas por la UC.
- Por el **bus de datos** viajan datos leídos de o a escribir en la MP y los puertos, y también las instrucciones que se leen de la MP y van a la UC.
- El **bus de control** engloba a las microórdenes que genera la UC y las que proceden de peticiones que hacen otras unidades a la UC. Generalmente este bus no se muestra explícitamente: sus líneas están agrupadas con las de los otros dos (o en un solo bus del sistema).

En la figura 2.7 no se han incluido otros registros que tiene el procesador y que son «transparentes» en el nivel de máquina convencional. Esto quiere decir que ninguna instrucción puede acceder directamente a ellos. El contador de programa (apartado 2.2) es un caso especial: en algunas arquitecturas es también transparente, pero en otras (como la que estudiaremos en el capítulo 3) es uno de los registros de la ML.

[2]Esto es suponiendo un *bus paralelo*, en el que se transmite un flujo de bytes o de n bytes. También hay *buses serie*, en los que se transmite un flujo de bits serializados (USB), o $2n$ flujos de bits, si el bus es bidireccional y tiene n enlaces (PCIe).

Un detalle que no refleja la figura es que, aunque toda la MP es «RAM» (es decir, de *acceso aleatorio*), una pequeña parte de ella es «ROM» (es decir, de *solo lectura*). En esta parte están grabados los programas que se ejecutan al iniciarse el sistema.

En sistemas muy sencillos, como algunos microcontroladores, éste es un modelo estructural que refleja bastante bien la realidad. Pero incluso en ordenadores personales, el tener un solo bus es muy ineficiente. En efecto, las transferencias entre la UCP y la MP son muy rápidas, pero hay una gran variedad de periféricos, más o menos lentos. Esto conduce a separar los buses y a establecer una jerarquía de buses, como puede apreciarse en la figura 2.8, que es el modelo estructural de un *pecé* del año 2010.

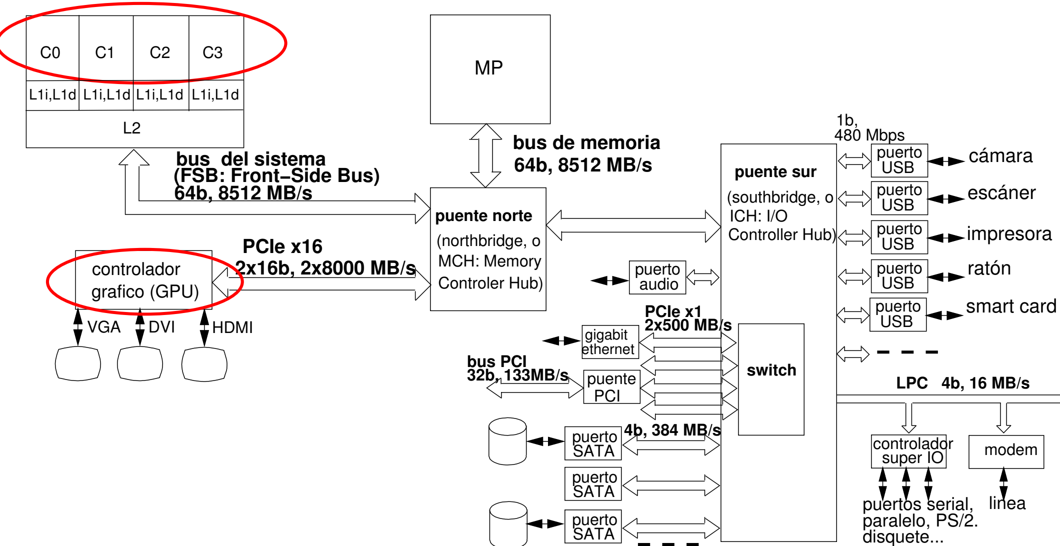


Figura 2.8 Modelo estructural de un ordenador personal.

Sin entrar en detalles, cabe destacar que:

- Hay varios tipos de buses (cada uno contiene líneas de datos, de direcciones y de control). Algunos (USB, SATA y PCIe) son buses serie. Recuerde que, como vimos en el apartado 1.2 del Tema 2, para los caudales, o tasas de transferencia, de los buses, se utilizan prefijos multiplicadores decimales: «16 MB/s» quere decir «16×10^6 bytes por segundo»),
- El microprocesador contiene cuatro «cores» (procesadores), C0-C3, es decir es un **sistema multiprocesador** integrado en un solo chip, y cinco memorias «cache» (apartado 2.6).
- Hay un procesador especializado para los gráficos, la **GPU** (Graphical Processing Unit).

2.6. Modelos procesales

La mayoría de las innovaciones procesales en la ejecución de las instrucciones se han realizado en el nivel de micromáquina y son transparentes en el nivel de máquina convencional, es decir, no afectan (o afectan a pocos detalles) a los modelos funcionales en este nivel. Pero hay dos que son especialmente importantes y vale la pena comentar: el encadenamiento y la memoria oculta.

Encadenamiento *(pipelining)*

Uno de los inventos que más ha contribuido a aumentar la velocidad de los procesadores es el **encadenamiento** (en inglés, *pipelining*). Es, conceptualmente, la misma idea de las cadenas de producción: si la fabricación de un producto se puede descomponer en etapas secuenciales e independientes y si se dispone de un operador para cada etapa, finalizada la etapa 1 del producto 1 se puede pasar a la etapa 2 de ese producto y, *simultáneamente* con ésta, llevar a cabo la etapa 1 del producto 2, y así sucesivamente.

El modelo procesal de la figura 2.5 supone que hay tres fases, o etapas, y que se llevan a cabo una detrás de otra. En la primera se hace uso de la MP, pero no así en la segunda ni (salvo excepciones) en la tercera. En la segunda se descodifica y se usa la ML para leer registros, y en la tercera se usa la UAL y la ML para escribir. Suponiendo que en la ML se pueden leer dos registros y escribir en otro al mismo tiempo, en cada fase se usan recursos distintos, por lo que pueden superponerse. El resultado, gráficamente, es el que ilustra la figura 2.9. Con respecto a un modelo procesal estrictamente secuencial, la velocidad de ejecución se multiplica (teóricamente) por tres.

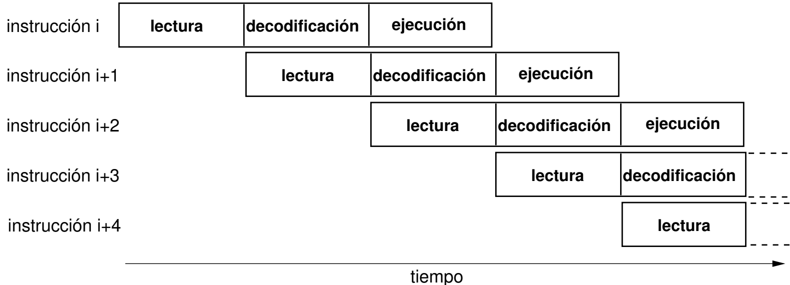


Figura 2.9 Una cadena de tres etapas.

Naturalmente, se presentan **conflictos** («*hazards*»). Por ejemplo:

- Si la instrucción opera con un dato de la MP, la fase de ejecución no puede solaparse con la de lectura (a menos que la MP tenga dos vías de acceso, una para instrucciones y otra para datos). Esta situación sería frecuente si solo hubiese un acumulador, pero la mayoría de las operaciones de procesamiento se realizan sobre la ML.
- Si se trata de una instrucción de bifurcación, esto no se sabe hasta la etapa de descodificación, cuando ya se ha leído, inútilmente, la instrucción que está en la dirección siguiente.

Estos conflictos se resuelven de diversas maneras en el nivel de micromáquina.

El número de etapas se llama **profundidad de la cadena** y varía mucho de unos diseños a otros. Algunos procesadores tienen una profundidad de varias decenas.

Memorias ocultas *(caches)*

Una expresión ya clásica en arquitectura de ordenadores es la de «cuello de botella de von Neumann». Se refiere a que, desde los primeros tiempos, los procesadores se implementan con tecnologías mucho más rápidas que las de la MP. Como toda instrucción tiene que ser leída de la MP, el tiempo se acceso se convierte en el factor que más limita el rendimiento. En este caso, de poco serviría el encadenamiento. Por ejemplo, si modificamos la figura 2.9 para el caso de que la etapa de lectura tenga una duración cien veces superior a las de descodificación y ejecución, la velocidad del procesador, medida

en instrucciones por segundo, no se multiplicaría por tres, sino por un número ligeramente superior a uno[3].

En realidad, sí que hay tecnologías de memoria muy rápidas, pero son demasiado costosas para implementar con ellas una RAM de gran capacidad. Lo que sí cabe hacer es una modificación del modelo estructural, introduciendo una **memoria oculta**, MO (figura 2.10).

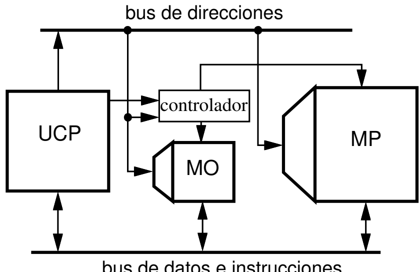


Figura 2.10. Memoria oculta *(cache)*.

Se trata de una RAM de capacidad y tiempo de acceso muy pequeños con relación a los de la MP (pero no tan pequeños como los de la ML incluida en el procesador). En la MO se mantiene una *copia* de *parte del contenido* de la MP. Siempre que el procesador hace una petición de acceso a una dirección de MP primero se mira en la MO para ver si el contenido de esa dirección está efectivamente copiado en ella; si es así, se produce un **acierto**, y el acceso es muy rápido, y si no es así hay un **fracaso**, y es preciso acceder a la MP. Un controlador se ocupa de las comprobaciones y gestiones de transferencias para que el procesador «vea» un sistema de memoria caracterizado por la capacidad total de la MP y un tiempo de acceso *medio* que, idealmente, debe acercarse al de la MO.

El buen funcionamiento del sistema de memoria construido con memoria oculta depende de una propiedad que tienen todos los procesos originados por la ejecución de los programas. Esta propiedad, conocida como **localidad de las referencias**, consiste en que las direcciones de la MP a las que se accede durante un período de tiempo en el proceso suelen estar localizadas en zonas pequeñas de la MP. Esta definición es, desde luego, imprecisa, pero no por ello menos cierta; su validez está comprobada empíricamente. Estas zonas se llaman **partes activas**, y el principio consiste en mantener estas partes activas en la memoria oculta (y rápida) durante el tiempo en que efectivamente son activas.

Y como hay una variedad de tecnologías (y a mayor velocidad mayor coste por bit), es frecuente que se combinen memorias de varias tecnologías, formando varios niveles de MO: una pequeña (del orden de varios KiB), que es la más próxima al procesador y la más rápida y se suele llamar «L1», otra mayor (del orden de varios MiB), L2, etc. Cuando el procesador intenta acceder a la MP, se direcciona en la L1, y generalmente resulta un acierto, pero si hay fracaso, el controlador accede a la L2, etc.

En las UCP que contienen varios procesadores («cores») la L2 (y, en su caso, la L3) es común para todos, pero cada procesador tiene su propia L1 y, además está formada por dos (es una arquitectura Harvard, apartado 2.1): L1d es la MO de datos y L1i es la MO de instrucciones, lo que, como hemos dicho antes, reduce los conflictos en el encadenamiento.

Los sistemas operativos ofrecen herramientas para informar del hardware instalado. En el ordenador con el que se está escribiendo este documento, la orden `lscpu` genera, entre otros, estos datos:

```
Architecture:          x86_64
CPU(s):                4
...
L1d cache:             32K
L1i cache:             32K
L2 cache:              3072K
```

Esto significa (aunque la salida del programa `lscpu` no es muy explícita) que el microprocesador contiene cuatro procesadores[4], cada uno de ellos acompañado de una MO de datos y una MO de

[3]Concretamente (es un sencillo ejercicio comprobarlo), e ignorando los posibles conflictos, por 1,02.

[4]A los procesadores, o «cores», el programa `lscpu` les llama «CPU». Nosotros llamamos UCP (CPU) al conjunto formado

instrucciones, de 32 KiB cada una, y que hay una MO de nivel 2 de 3 MiB común a los cuatro (es el ordenador de la figura 2.8).

En todo caso, como decíamos al principio de este apartado, la existencia de las memorias ocultas es transparente en el nivel de máquina convencional. Es decir, las instrucciones de máquina hacen referencia a direcciones de la MP, y el programador que las utiliza no necesita ni siquiera saber de la existencia de tales memorias ocultas. Es en el nivel de micromáquina en el que el hardware detecta los aciertos o fracasos y los resuelve. En el peor (y poco frecuente) caso ocurrirá que habrá que acceder hasta la MP, y la única consecuencia «visible»[5] será que la ejecución de esa instrucción tardará más tiempo.

2.7. Modelos funcionales

Paralelamente con la evolución de las tecnologías de implementación y la introducción de memorias locales en el procesador, las arquitecturas (en el sentido de «ISA», es decir, los modelos funcionales, apartado 1.3) se han ido enriqueciendo. Hay una gran variedad de procesadores, cada uno con su modelo funcional. Resumiremos, de manera general, los aspectos más importantes, y en el siguiente capítulo veremos los detalles para un procesador concreto.

Direcciones y contenidos de la MP

En la máquina de von Neumann las palabras tenían cuarenta bits, lo que se justificaba, como hemos leído, porque en cada una podía representarse un número entero binario con suficiente precisión. Y en cada acceso a la MP se leía o se escribía una palabra completa, que podía contener o bien un número o bien dos instrucciones. Pero pronto se vio la necesidad de almacenar caracteres, se elaboraron varios códigos para su representación y se consideró conveniente poder acceder no ya a una palabra (que podía contener varios caracteres codificados), sino a un carácter. Algunos códigos primitivos eran de seis bits y se llamó **byte** a cada conjunto de seis bits. Más tarde se impuso el código ASCII de siete bits y se redefinió el byte como un conjunto de ocho bits (el octavo, en principio, era un bit de paridad), por lo que también se suele llamar **octeto**.

En todas las arquitecturas modernas la MP es accesible por bytes, es decir, cada byte almacenado tiene su propia dirección. Pero los datos y las instrucciones ocupan varios bytes, por lo que también es accesible por **palabras**, medias palabras, etc. El número de bits de una palabra es siempre un múltiplo de ocho, y coincide con el ancho del bus de datos[6]. La expresión «arquitectura de x bits» se refiere al número de bits de la palabra, normalmente 16, 32 o 64. Cuando inicia una operación de acceso, el procesador pone una dirección en el bus de direcciones y le comunica a la MP, mediante señales de control, si quiere leer (o escribir) un byte, o una palabra, o media palabra, etc.

En una arquitectura de dieciséis bits, la palabra de dirección d está formada por los bytes de direcciones d y $d + 1$. Si es de treinta y dos bits, por los cuatro bytes de d a $d + 3$, etc. Y como hemos visto

por los cuatro.

[5]En principio, haría falta una vista de «tipo Superman», pero se puede escribir un programa en el que se repitan millones de veces instrucciones que provocan fracasos y ver el tiempo de respuesta frente a otra versión del programa en la que no haya fracasos.

[6]A veces el ancho real del bus de datos es de varias palabras, de modo que se pueda anticipar la lectura de varias instrucciones y la UC pueda identificar bifurcaciones que provocarían conflictos en el encadenamiento, pero esto sucede en el nivel de micromáquina y es transparente en el nivel de máquina convencional.

en el Tema 2 (apartado 1.4), el convenio puede ser *extremista mayor* (el byte de dirección más baja contiene los bits *más* significativos de la palabra) o el contrario, *extremista menor*.

Junto con el convenio de almacenamiento, hay otra variante: el alineamiento de las palabras. En algunas arquitecturas no hay ninguna restricción sobre las direcciones que pueden ocupar las palabras, pero en otras se requiere que las palabras no se «solapen». Así, en una arquitectura de dieciséis bits sería obligatorio que las palabras tuviesen siempre dirección par, en una de treinta y dos bits, que fuesen múltiplos de cuatro (figura 2.11(a)), etc. En este segundo caso, se dice que las direcciones de palabra deben estar **alineadas**.

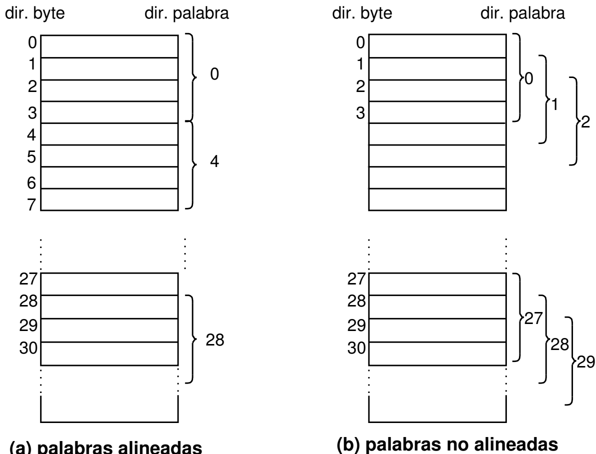


Figura 2.11 Direcciones alineadas y no alineadas en una arquitectura de 32 bits.

Si el ancho del bus de datos está ligado a la longitud de palabra, el ancho del bus de direcciones está ligado al **espacio de direccionamiento**, es decir, el número de direcciones diferentes que puede generar el procesador. Un bus de direcciones de diez líneas puede transportar 2^{10} direcciones distintas, por lo que el espacio de direccionamiento es de 1 KiB; con dieciseis líneas el espacio es de 2^{16} = 64 KiB, etc.

Observe que hablamos de «espacio de direccionamiento», no de «capacidad de la MP». La MP no necesariamente alcanza la capacidad que es capaz de direccionar el procesador. Si las direcciones son de 32 bits, la capacidad de direccionamiento es 2^{32} = 4 GiB. No todos los sistemas tienen una memoria de esta capacidad. Por este motivo, las direcciones generadas por el procesador se suelen llamar **direcciones virtuales**. Un mecanismo que combina hardware (la «unidad de gestión de memoria») y software (el «sistema de gestión de memoria») puede ocuparse de que los programas «vean» ese espacio de direccionamiento como si realmente tuviesen disponible una MP de esa capacidad, ayudándose de memoria auxiliar en disco. Este es el principio de funcionamiento de la llamada **memoria virtual**.

Formatos y repertorios de instrucciones

En las décadas que siguieron a la propuesta de von Neumann y a los primeros ordenadores la evolución de los procesadores se caracterizó por un enriquecimiento del repertorio de instrucciones, tanto en cantidad (varias centenas) como en modos de acceder a la MP. Esto obedecía a razones prácticas[7]: cuantas más posibilidades ofrezca el nivel de máquina convencional más fácil será desarrollar niveles

[7]Se ha demostrado teóricamente que un procesador con solo dos instrucciones, «sumar uno al acumulador» y «decrementar el acumulador y bifurcar si resulta cero» puede hacer lo mismo que cualquier otro (o, por decirlo también de manera teórica, dotado de una MP infinita, tendría la misma potencia computacional que una máquina de Turing).

superiores (sistemas operativos, compiladores, aplicaciones, etc.). Además, la variedad de instrucciones conducía a que su formato fuese de longitud variable: para las sencillas puede bastar con un byte, mientras que las complejas pueden necesitar varios. Todo esto implica que la unidad de control es más complicada, y se inventó una técnica llamada **microprogramación**: la unidad de control se convierte en una especie de «procesador dentro del procesador», con su propia memoria que alberga microprogramas, donde cada microprograma es un algoritmo para interpretar y ejecutar una instrucción. (De ahí el «nivel de micromáquina»).

Sin embargo, muchos estudios sobre estadísticas de uso de instrucciones durante la ejecución de los programas demostraban que gran parte de tales instrucciones apenas se utilizaban. En media, del análisis de programas reales resultaba que alrededor del 80 % de las operaciones se realizaba con solo un 20 % de las instrucciones del repertorio.

Esto condujo a enfocar de otro modo el diseño: si se reduce al mínimo el número de instrucciones (no el mínimo teórico de la nota a pie de página, pero sí el que resulta de eliminar todas las que tengan un porcentaje muy bajo de utilización) los programas necesitarán más instrucciones para hacer las operaciones y, en principio, su ejecución será más lenta. Pero la unidad de control será mucho más sencilla, y su implementación mucho más eficiente. Es decir, que quizás el resultado final sea que la máquina no resulte tan «lenta».

CISC y RISC

A un ordenador cuyo diseño sigue la tendencia «tradicional» se le llama **CISC** (Complex Instruction Set Computer), por contraposición a **RISC** (Reduced Instruction Set Computer).

A pesar de su nombre, lo que mejor caracteriza a los RISC no es el tener pocas instrucciones, sino otras propiedades

- Tienen **arquitectura «load/store»**. Esto quiere decir que los únicos accesos a la MP son para extraer instrucciones y datos y para almacenar datos. Todas las operaciones de procesamiento se realizan en registros del procesador.
- Sus instrucciones son «sencillas»: realizan únicamente las operaciones básicas, no como los CISC, que tienen instrucciones sofisticadas que facilitan la programación, pero cuyo porcentaje de uso es muy bajo.
- Su formato de instrucciones es «regular»: todas las instrucciones tienen la misma longitud y el número de formatos diferentes es reducido. En cada formato, todos los bits tienen el mismo significado para todas las instrucciones, y esto permite que la unidad de control sea más sencilla.

Actualmente las arquitecturas CISC y RISC conviven pacíficamente. Los procesadores diseñados a partir de los años 1980 generalmente son RISC, pero otros (incluidos los más usados en los ordenadores personales), al ser evoluciones de diseños anteriores y tener que conservar la compatibilidad del software desarrollado para ellos, son CISC.

Modos de direccionamiento

En la máquina de von Neumann, las instrucciones que hacen referencia a memoria (almacenar el acumulador en una dirección de la MP, sumar al acumulador el contenido de una dirección, bifurcar a una dirección...) indican la dirección de memoria *directamente* en el campo CD. Hay diversos mecanismos mediante los cuales la dirección real, o **dirección efectiva** no se obtiene directamente de CD,

sino combinándolo con otras cosas, o, incluso, prescindiendo de CD, lo que facilita la programación y hace más eficiente al procesador. Dos de ellas son:

- El **direccionamiento indirecto** consiste, en principio, en interpretar el contenido de CD como la dirección de MP en la que se encuentra la dirección efectiva. Pero esto implica que para acceder al operando, o para bifurcar, hay que hacer dos accesos a la MP, puesto que primero hay que leer la palabra que contiene la dirección efectiva. A esta palabra se le llama **puntero**: *apunta a* el operando o a la instrucción siguiente.
  Como normalmente el procesador tiene una ML, es más eficiente que el puntero sea un registro. En tal caso, la instrucción no tiene campo CD, solo tiene que indicar qué registro actúa como puntero.
- En el **direccionamiento relativo a programa** el contenido de CD es una **distancia** *(«offset»)*: un número con signo que se suma al valor del contador de programa para obtener la dirección efectiva. La utilidad es, sobre todo, para las instrucciones de bifurcación, ya que la dirección a la que hay que bifurcar suele ser cercana a la dirección de la instrucción actual, por lo que el campo CD puede tener pocos bits.

Veamos un ejemplo de uso de un modelo funcional que disponga de estos modos: sumar los cien elementos de un vector que están almacenados en cien bytes de la MP, siendo D la dirección del primero. Supongamos que todas las instrucciones ocupan una palabra de 32 bits (cuatro bytes), y que la primera se almacena en la dirección I. Utilizando el lenguaje natural, esta secuencia de instrucciones resolvería el problema, dejando el resultado en el registro R2:

Dirección	Instrucción
I	Poner D en el registro R0 (puntero)
$I+4$	Poner 100 en el registro R1 (contador de 100 a 1)
$I+8$	Poner a cero el registro R2 (suma)
$I+12$	Copiar en R3 el contenido de la palabra de la MP *apuntada* por R0 e incrementar el contenido de R0
$I+16$	Sumar al contenido de R2 el de R3
$I+20$	Decrementar el contenido de R1 en una unidad
$I+24$	Si el contenido de R1 no es cero, *bifurcar a* la instrucción que está en la dirección de ésta menos 12

Observe que:

- El programa escrito en lenguaje natural se llama **pseudocódigo**. No hay ningún procesador software que traduzca del lenguaje natural al lenguaje de máquina. Las mismas instrucciones se pueden escribir así en el lenguaje ensamblador que estudiaremos en el capítulo 4:

```
        ldr     r0,=D
        mov     r1,#100
        mov     r2,#0
bucle:  ldrb    r3,[r0],#1
        add     r2,r2,r3
        subs    r1,r1,#1
        bne     bucle
```

- En la instrucción $I + 24$ se usa direccionamiento relativo a programa. Si el procesador tiene una cadena de profundidad 3 (figura 2.9), en el momento en que esta instrucción se esté ejecutando ya se está decodificando la instrucción siguiente ($I+28$) y leyendo la siguiente a ella ($I+32$). El contador de programa siempre contiene la dirección de la instrucción que está en la fase de lectura, por lo que en ese momento su contenido es $I + 32$. Es decir, la instrucción debe decir: «bifurcar a la dirección

que resulta de restar 20 al contador de programa» ($I + 32 - 20 = I + 12$).

- Este cálculo de la distancia a sumar o a restar del contenido del contador de programa es, obviamente, engorroso. Pero solo tendríamos que hacerlo si tuviésemos que escribir los programas en lenguaje de máquina. Como ponen de manifiesto las instrucciones anteriores en lenguaje ensamblador, al programar en este lenguaje utilizamos símbolos como «`bucle`» y será el ensamblador (procesador) el que se ocupe de ese cálculo.
- Esta instrucción $I+24$ podría utilizar direccionamiento directo: «... bifurcar a la instrucción que está en la dirección $I + 12$», pero, aparte de que I puede ser mucho mayor que la distancia (-20) y no caber en el campo CD, eso hace que este programa solamente funcione si se carga en la MP a partir de la dirección I. Sin embargo, con el direccionamiento relativo, se pueden cargar a partir de otra dirección cualquiera y funcionará igual.
- Cuando el programa se ejecuta, las instrucciones $I+12$ a $I+24$ se repiten 100 veces (o el número que inicialmente se ponga en R1). Se dice que forman un **bucle**, del que se sale gracias a una **instrucción de bifurcación condicionada**. La condición en este caso es: (R1) $\neq 0$ (los paréntesis alrededor de R1 indican «contenido»). Cuando la condición no se cumpla, es decir, cuando (R1) $= 0$, la UC pasará a ejecutar la instrucción que esté en la dirección siguiente a la de bifurcación, $I + 28$.

Subprogramas

Ocurre con mucha frecuencia que en varios puntos de un programa hay que realizar una determinada secuencia de operaciones, siempre la misma, con operandos distintos cada vez. Por ejemplo, en un mismo programa podríamos necesitar el cálculo de la suma de componentes de distintos vectores, almacenados en distintas zonas de la MP y con distintas dimensiones. Una solución trivial es repetir esa secuencia cuantas veces sea necesario, pero más razonable es tenerla escrita una sola vez en una zona de la MP y «llamarla» cuando sea necesario mediante una instrucción que permita saltar a la primera instrucción de la secuencia.

Ahora bien, para que esta solución (la «razonable») funcione correctamente deben cumplirse dos condiciones:

- **Pasar los parámetros**: antes de saltar a esa secuencia (que en adelante llamaremos **subprograma**) hay que indicarle los datos sobre los que ha de operar («parámetros de entrada»), y después de su ejecución el subprograma ha de devolver resultados al que lo llamó («parámetros de salida»).
- **Preservar la dirección de retorno**: una vez ejecutado el subprograma hay que volver al punto adecuado del programa que lo llamó, y este punto es distinto cada vez.

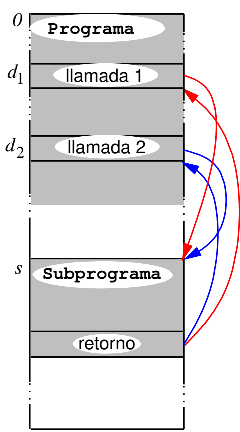


Figura 2.12. Llamadas y retornos.

El mapa de memoria de la figura 2.12 corresponde a una situación en la que hay un programa cargado a partir de la dirección 0 que llama dos veces a un subprograma cargado a partir de la dirección s. En este ejemplo, «llamada 1» y «llamada 2» son instrucciones almacenadas en las direcciones d_1 y d_2. Al final del subprograma habrá que poner alguna instrucción para volver a la instrucción siguiente a la de llamada, que es distinta cada vez.

Para el paso de parámetros, lo más sencillo es colocarlos en registros antes de la llamada, de donde los puede recoger el subprograma. Así, utilizando de nuevo pseudocódigo, las llamadas desde un programa que primero tiene que sumar los elementos de un vector que empieza en la dirección D_1 y tiene L_1 elementos y luego los de otro que empieza en D_2 y tiene L_2 elementos serían:

Dirección	Instrucción
I_1	Poner D_1 en el registro R0 (puntero)
$I_1 + 4$	Poner L_1 en el registro R1
$I_1 + 8$	**Llamar al subprograma (transferencia a s)**
$I_1 + 12$	[Instrucciones que operan con la suma del vector 1, en R2]
...	...
...	...
I_2	Poner D_2 en el registro R0 (puntero)
$I_2 + 4$	Poner L_2 en el registro R1
$I_2 + 8$	**Llamar al subprograma (transferencia a s)**
$I_2 + 12$	[Instrucciones que operan con la suma del vector 2, en R2]
...	...
...	...

Los parámetros de entrada son la dirección del vector y el número de elementos, que se pasan, respectivamente, por R0 y R1. El parámetro de salida es la suma, que el subprograma ha de devolver por R2. Las direcciones $I_1 + 8$ e $I_2 + 8$ son las que en la figura 2.12 aparecen como d_1 y d_2.

El subprograma contendrá las instrucciones $I + 8$ a $I + 24$ del ejemplo inicial, pero tras la última necesitará una instrucción para volver al punto siguiente a la llamada:

Dirección	Instrucción
s	Poner a cero el registro R2 (suma)
$s + 4$	Copiar en R3 el contenido de la palabra de la MP *apuntada* por R0 e incrementar el contenido de R0
$s + 8$	Sumar al contenido de R2 el de R3
$s + 12$	Decrementar el contenido de R1 en una unidad
$s + 16$	Si el contenido de R1 no es cero, *bifurcar a* la instrucción que está en la dirección que resulta de restar 20 al contador de programa ($s + 16 + 8 - 20 = s + 4$)
$s + 20$	**Retornar a la instrucción siguiente a la llamada**

Las instrucciones de llamada y retorno son, como las bifurcaciones, **instrucciones de transferencia de control**, ya mencionadas al final del apartado 2.3: hacen que la instrucción siguiente a ejecutar no sea la que está en la dirección ya preparada en el contador de programa (CP), sino otra distinta. Pero en una bifurcación la dirección de la instrucción siguiente se obtiene de la propia instrucción, mientras que las instrucciones de llamada y retorno tienen que trabajar de forma conjunta: la de llamada, además de bifurcar tiene que dejar en algún sitio la **dirección de retorno**, y la de retorno tiene que acceder a ese sitio para poder volver.

En el ejemplo, al ejecutarse $s + 20$ se debe introducir en CP un valor (la dirección de retorno) que tendrá que haber dejado la instrucción de llamada ($I_1 + 8$ o $I_2 + 8$; dirección de retorno: $I_1 + 12$ o $I_2 + 12$). La cuestión es: ¿dónde se guarda ese valor?

Algunos procesadores tienen un registro especial, llamado **registro de enlace**. La instrucción de llamada, antes de poner el nuevo valor en CP, introduce la dirección de la instrucción siguiente en ese registro. La instrucción de retorno se limita a copiar el contenido del registro de enlace en CP. Es una solución sencilla y eficaz al problema de preservación de la dirección de retorno. Pero tiene un inconveniente: si el subprograma llama a su vez a otro subprograma, la dirección de retorno al primero se pierde. Una solución más general es el uso de una pila.

Pilas

Una **pila** es un tipo de memoria en la que solo se pueden leer los datos en el orden inverso en que han sido escritos, siguiendo el principio de «*el último en entrar es el primero en salir*». Las dos operaciones posibles son **push** (*introducir* o *apilar:* escribir un elemento en la pila) y **pop** (*extraer* o *desempilar:* leer un elemento de la pila). Pero, a diferencia de lo que ocurre con una memoria de acceso aleatorio, no se puede introducir ni extraer en cualquier sitio. Únicamente se tiene acceso a la *cima* de la pila, de modo que un nuevo elemento se introduce siempre sobre el último introducido, y la operación de extracción se realiza siempre sobre el elemento que está en la cima, como indica la figura 2.13. Por tanto, *en una memoria de pila no hay direcciones*. La primera posición es el *fondo* de la pila. Si la pila está vacía, no hay cima; si solo contiene un elemento, la cima coincide con el fondo.

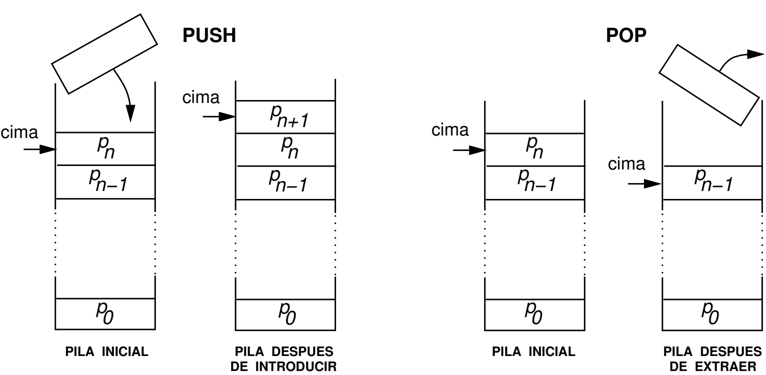


Figura 2.13 Memoria de pila.

Una memoria de pila se puede implementar en hardware, y hay procesadores con arquitectura diseñada para trabajar con operandos en la pila. En ellos, la instrucción «sumar», por ejemplo, no hace referencia a ninguna dirección de memoria ni utiliza ningún registro: lo que hace es extraer de la pila el elemento que está en la cima y el que está debajo de él, sumarlos, y almacenar el resultado en el lugar que ocupaba el segundo. Pero estos procesadores son raros[8]. Lo que sí tienen todos los procesadores hardware es un mecanismo para «simular» una pila en una parte de la MP, que consiste en lo siguiente:

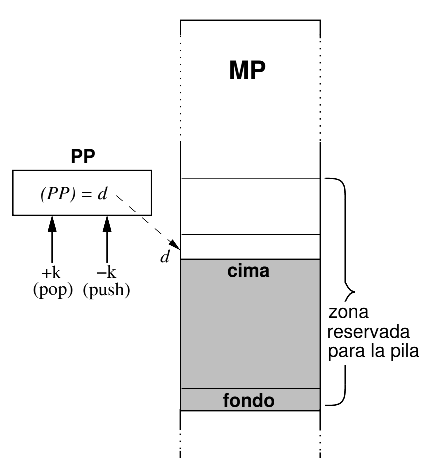


Figura 2.14 Una pila en una RAM.

El procesador tiene registro llamado **puntero de pila**, PP, que contiene en todo momento la primera dirección de la MP libre por encima de la pila (figura 2.14) (o bien la dirección de la cima). En la instrucción de llamada a subprograma, la UC guarda la dirección de retorno (el contenido del contador de programa en ese momento) en la pila, haciendo un *push*. Esta operación *push* se realiza en dos pasos: primero se escribe la dirección de retorno en la dirección apuntada por PP y luego se decrementa el contenido de PP en tantas unidades como bytes ocupe una dirección (si el convenio es que PP apunta a la cima, se invierten los pasos).

La ejecución de la instrucción de retorno se hace con un *pop* para extraer de la pila la dirección de retorno y ponerla

[8]Un ejemplo es la «máquina virtual Java», que se utiliza como paso intermedio en la ejecución de programas escritos en el lenguaje Java, que estudirá usted en la asignatura «Programación». Normalmente se implementa con software.

en el contador de programa. La operación *pop* consiste en incrementar el contenido de PP y luego leer de la dirección apuntada por PP (si el convenio es que PP apunta a la cima, se invierten los pasos).

Repare en tres detalles de esta solución:

- Las operaciones *push* y *pop* mencionadas son transparentes al programador. Es decir, éste solamente tiene que saber que existe una instrucción (en ensamblador se suele llamar `CALL`) que le permite transferir el control a una dirección donde comienza un subprograma, y que la última instrucción de este subprograma debe ser la de retorno (en ensamblador, `RET`).
- El «**anidamiento**» de subprogramas (que un subprograma llame a otro, y éste a otro... o, incluso, que un subprograma se llame a sí mismo de manera recursiva) no plantea ningún problema: las direcciones de retorno se irán apilando y recuperando en el orden adecuado. (El único problema práctico radica en que la pila no es infinita).
- La pila puede utilizarse también para transmitir parámetros entre el programa y el subprograma: antes de `CALL`, el subprograma utiliza instrucciones «`PUSH`» para introducirlos en la pila, y el subprograma puede acceder a ellos con un modo de direccionamiento que sume una distancia al puntero de pila.

2.8. Comunicaciones con los periféricos e interrupciones

Los dispositivos periféricos se comunican con el procesador a través de «puertos»: registros incluidos en el hardware del controlador del periférico en los que este controlador puede poner o recoger datos, o poner informaciones sobre su estado, o recoger órdenes procedentes del procesador.

Cada puerto tiene asignada una **dirección de entrada/salida**, y los procesadores siguen uno de estos dos convenios:

- **Espacios de direccionamiento independientes**: las direcciones de entrada/salida no tienen relación alguna con las direcciones de la memoria. La dirección *d* puede referirse a una posición de memoria o a un puerto. En estos procesadores son necesarias instrucciones específicas para la entrada/salida.
- **Espacio de direccionamiento compartido**: ciertas direcciones generadas por el procesador no corresponden a la memoria, sino a puertos. Por tanto, el acceso a estos puertos no requiere instrucciones especiales, son las mismas que permiten el acceso a la memoria. Se dice que la entrada/salida está *«memory mapped»*.

La forma de programar estas operaciones depende de las características de cada periférico, fundamentalmente, de su *tasa de transferencia*. Hay dos tipos (son los que el sistema de gestión de ficheros conoce como «ficheros especiales de caracteres» y «ficheros especiales de bloques», Tema 2, apartado 5.1):

- **Periféricos de caracteres**: son los «lentos», en los que la tasa de transferencia de datos es sustancialmente inferior a la velocidad con que el procesador puede procesarlos (teclado, ratón, sensor de temperatura, etc.). Cada vez que el periférico está preparado para enviar o recibir un dato se ejecuta una instrucción que transfiere el dato (normalmente, un byte o «carácter») entre un registro del procesador y el puerto correspondiente.

- **Periféricos de bloques**: son aquellos cuya tasa de transferencia es comparable a la velocidad del procesador (disco, pantalla gráfica, USB, controlador ethernet, etc.), y se hace necesario que el controlador del periférico se comunique directamente con la memoria, leyendo o escribiendo en cada operación no ya un byte, sino un bloque de bytes. Es la técnica de **DMA** (acceso directo a memoria, *Direct Memory Access*).

En ambos casos juega un papel importante el mecanismo de interrupciones.

Interrupciones

El mecanismo de interrupciones, presente en todos los procesadores hardware de uso general actuales, permite intercambiar datos entre el procesador y los periféricos, implementar llamadas al sistema operativo (mediante instrucciones de interrupción por software) y tratar situaciones excepcionales (instrucciones inexistentes, fallos en el acceso a la memoria, instrucciones privilegiadas, etc.).

El tratamiento de las interrupciones es uno de los asuntos más fascinantes y complejos de la arquitectura de procesadores y de la ingeniería en general. Fascinante, porque gracias a él dos subsistemas de naturaleza completamente distinta, el hardware (material) y el software (intangible) trabajan conjuntamente, y de esa simbiosis diseñada resultan los artefactos que disfrutamos actualmente. Y es complejo, sobre todo para explicar, porque los diseñadores de procesadores han ingeniado distintas soluciones.

En el capítulo siguiente concretaremos los detalles de una de esas soluciones. De momento, señalemos algunos aspectos generales:

- La atención a una interrupción consiste en ejecutar una **rutina de servicio**, un programa que, naturalmente, tiene que estar previamente cargado en la MP.
- Atender a una interrupción implica abandonar temporalmente el programa que se está ejecutando para pasar a la rutina de servicio. Y cuando finaliza la ejecución de la rutina de servicio, el procesador debe volver al programa interrumpido para continuar con su ejecución.
- Esto recuerda a los subprogramas, pero hay una diferencia fundamental: la interrupción puede no tener ninguna relación con el programa, y puede aparecer en cualquier momento. Por tanto, no basta con guardar automáticamente la dirección de retorno (dirección de la instrucción siguiente a aquella en la que se produjo la atención a la interrupción), también hay que guardar los contenidos del registro de estado y de los registros que use la rutina (para que el programa continúe como si nada hubiese pasado). Naturalmente, al volver al programa interrumpido deben recuperarse.
- Como puede haber muchas causas que provocan la interrupción, el mecanismo debe incluir la identificación para poder ejecutar la rutina de servicio que corresponda. En algunos diseños esta identificación se hace por software, en otros por hardware, o, lo que es más frecuente, mediante una combinación de ambos.

2.9. Conclusión

Hemos visto los conceptos más importantes del nivel de máquina convencional que se implementan en los procesadores hardware actuales. En este sentido, ya hemos cubierto uno de los objetivos planteados. Pero es un hecho que estos conceptos no se asimilan completamente si no se materializan y se practican sobre un procesador concreto. Eso es lo que haremos en los dos capítulos siguientes.

Capítulo 3

El procesador BRM

ARM (Advanced RISC Machines) es el nombre de una familia de procesadores, y también el de una compañía, ARM Holdings plc. A diferencia de otras, la principal actividad de esta empresa no consiste en fabricar microprocesadores, sino en «licenciar» la propiedad intelectual de sus productos para que otras empresas puedan incorporarlos en sus diseños. Por este motivo, aunque los procesadores ARM sean los más extendidos, la marca es menos conocida que otras. En 2010 se estimaba en 25 millardos el número total acumulado de procesadores fabricados con licencia de ARM, a los que se sumaban 7,9 millardos en 2011[1]. Estos procesadores se encuentran, sobre todo, en dispositivos móviles (el 90 % de los *smartphones* contienen al menos un SoC basado en ARM), pero también en controladores de discos, en televisores digitales, en dispositivos de ayuda al frenado, en cámaras digitales, en impresoras, en consolas, en *routers* etc.

Parece justificado basarse en ARM para ilustrar los principios de los procesadores hardware actuales. Ahora bien, se trata de una *familia* de procesadores que comparten muchas características, tan numerosas que sería imposible, en los créditos asignados a este Tema, describirlas por completo. Es por eso que nos hemos quedado con los elementos esenciales, esquematizando de tal modo la arquitectura que podemos hablar de un «ARM simplificado», al que hemos bautizado como «BRM» (Basic RISC Machine).

BRM es totalmente compatible con los procesadores ARM. De hecho, todos los programas que se presentan en el capítulo siguiente se han probado con un simulador de ARM.

En este capítulo, tras una descripción somera de los modelos estructural y procesal, veremos con todo detalle el modelo funcional, es decir, todas las instrucciones, sus modos de direccionamiento y sus formatos. Como la expresión binaria es muy farragosa, iremos introduciendo las expresiones en lenguaje ensamblador (apartado 1.2). Hay varios lenguajes para la arquitectura ARM, y en el capítulo siguiente seguiremos la sintaxis del *ensamblador GNU*, que es también el que se utiliza en el simulador *ARMSim#* que nos servirá para practicar. De todas formas, en este capítulo no hay programas, solo instrucciones sueltas, y los convenios (nemónicos para códigos de operación y registros) son los mismos en todos los ensambladores.

[1]Fuentes: `http://www.arm.com/annualreport10/20-20-vision/forward-momentum.html`
`http://financialreports.arm.com/downloads/pdfs/ARM_AR11_Our_Story_v1.pdf`

3.1. Modelo estructural

La figura 3.1 muestra todos los componentes que es necesario conocer para comprender y utilizar el modelo funcional en el nivel de máquina convencional. El procesador (lo que está dentro del rectángulo grande) se comunica con el exterior (la MP y los periféricos) mediante los buses de direcciones (A) y de datos e instrucciones (D) y algunas señales binarias que entran o salen de la unidad de control[2]

Ya sabe usted lo que es la UAL y la UC (no se han dibujado las micrórdenes que genera ésta para controlar a las demás unidades, lo que complicaría innecesariamente el diagrama). La «UD» es una **unidad de desplazamiento**, que permite desplazar o rotar a la derecha o a la izquierda un dato de 32 bits.

Veamos primero las funciones de los distintos registros (todos de 32 bits), aunque muchas no se comprenderán bien hasta que no expliquemos los modelos procesal y funcional.

Registros de comunicación con el exterior

En la parte superior izquierda, el *registro de direcciones*, RA, contendrá una dirección de memoria o de puerto de entrada/salida de donde se quiere leer o donde se pretende escribir. Tanto este registro como el bus A tienen 32 bits, por lo que el **espacio de direccionamiento** es de 2^{32} bytes (4 GiB). Este espacio está *compartido* entre la MP y los puertos de entrada/salida. Es decir, algunas direcciones corresponden a puertos. Pero estas direcciones no están fijadas: será el diseñador de los circuitos externos que conectan la MP y los periféricos a los buses el que, mediante circuitos descodificadores, determine qué direcciones no son de la MP sino de puertos.

Abajo a la izquierda hay dos **registros de datos**. En RDL se introducirán los datos que se lean del exterior, y en RDE los que vayan a escribirse.

Si hay un registro, RDL, en el que se introducen los datos leídos (de la MP o de los puertos periféricos), hace falta otro distinto, RI (**registro de instrucción**), para introducir las *instrucciones* que se leen de la MP. Pero no solo hay uno, sino tres: RIL, RID y RIE. La explicación se encuentra en el modelo procesal de BRM, en el que las fases de ejecución de las instrucciones se solapan en el tiempo (figura 2.9): mientras una se ejecuta (la que está en RIE) la siguiente (en RID) se está descodificando y la siguiente a ésta (en RIL) se está leyendo.

Cuando la UC ha de ejecutar una transferencia con el exterior (ya sea en el momento de la lectura de una instrucción o como respuesta a determinadas instrucciones que veremos en el apartado 3.5), pone la dirección en el registro RA, si es una salida de datos pone también el dato en RDE, y activa las señales **req**, **r/w** y **size**. Si ha sido una petición de lectura, después de un ciclo de reloj la UC introduce

[2]Para comprender el nivel de máquina convencional no es estrictamente necesario, pero sí instructivo, conocer la funcionalidad de estas señales de control:
clk es la entrada de reloj, una señal periódica que provoca las transiciones de cada fase a la siguiente en el proceso de ejecución de las instrucciones.
reset fuerza que se introduzca 0 en el contador de programa.
irq es la petición de interrupción externa.
req es la petición del procesador a la memoria o a un periférico, y va acompañada de **r/w**, que indica si la petición es de lectura (entrada, **r/w** = 1) o de escritura (salida, **r/w** = 0) y de **size** (dos líneas) en las que el procesador codifica si la petición es de un byte, de media palabra (16 bits) o de una palabra (32 bytes) (aunque en BRM, versión simplificada de ARM, no hay accesos a medias palabras).
wait la puede poner un controlador externo (el **árbitro del bus**) para que el procesador se abstenga, durante el tiempo que está activada, de acceder a los buses A y D.

Estas señales son un subconjunto de las que tienen los procesadores ARM.

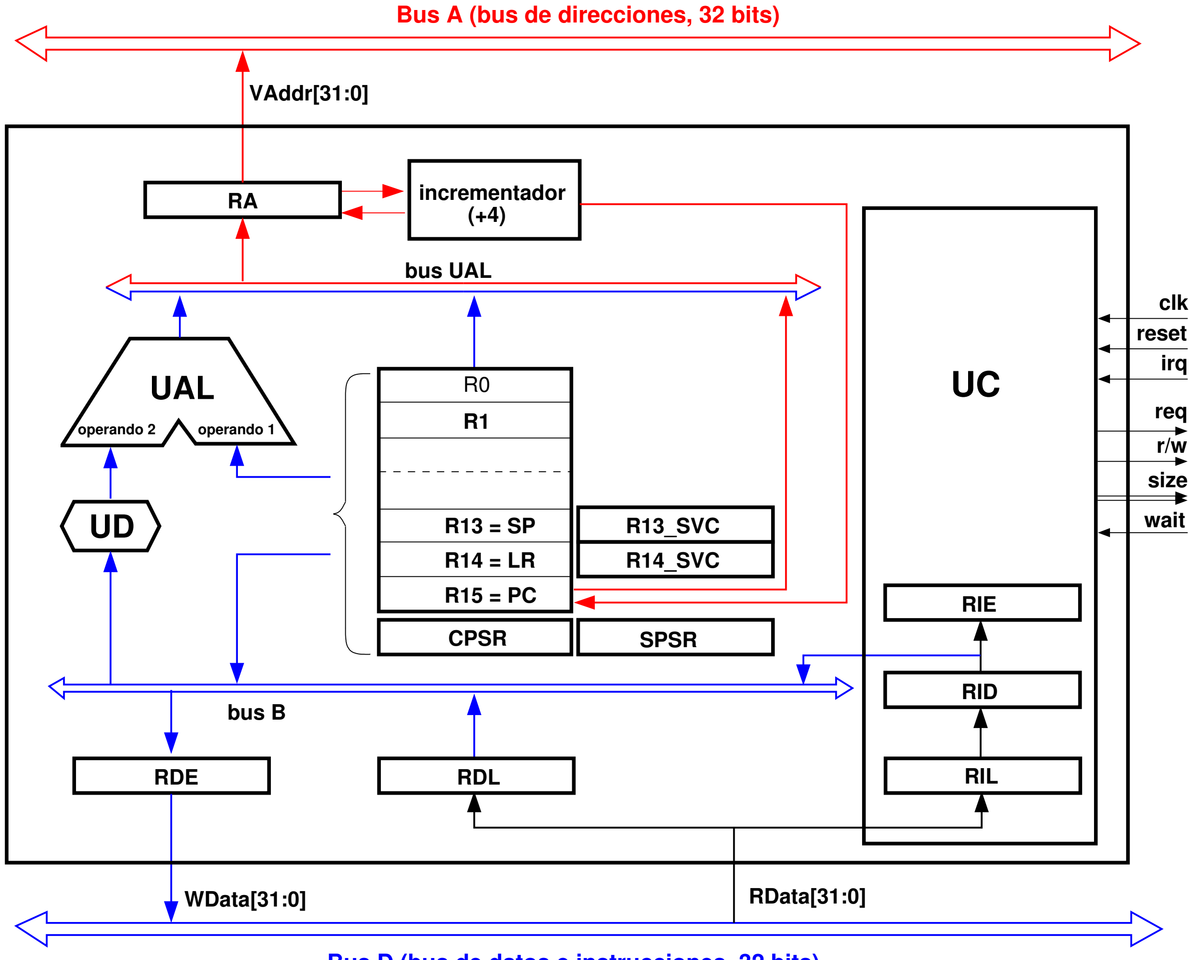


Figura 3.1 Modelo estructural de BRM.

el contenido del bus D en RDL o RIL, según proceda.

Todos estos registros son *transparentes* en el nivel de máquina convencional: ninguna instrucción los utiliza *directamente*. No así los demás.

Memoria local y registros especiales

La memoria local contiene dieciséis registros, R0 a R15. Los trece primeros son de propósito general (pueden contener operandos, resultados o direcciones), pero los tres últimos tienen funciones especiales que ya hemos avanzado en el capítulo anterior:

- R13, que también se llama SP (Stack Pointer), es el **puntero de pila**.
- R14, también conocido como LR (Link Register) es el **registro de enlace**.
- R15, o PC (Program Counter) es el **contador de programa**[3].

[3]Venimos traduciendo muchas siglas al castellano (UCP por CPU, UAL por ALU, etc.), pero para estos tres registros, y para algún otro, es preferible conservar sus nombres en inglés, porque son los que se utilizan en el lenguaje ensamblador.

Modos usuario y supervisor

Hay dos R13 y dos R14. El motivo es que el procesador en todo momento está en uno de entre dos modos de funcionamiento: modo usuario o modo supervisor. Normalmente está en modo usuario, pero cuando pasa a atender a una interrupción y cuando recibe la señal **reset** cambia a modo supervisor. Y en cada modo se reserva una zona de la MP distinta para la pila. El procesador, según el modo en que se encuentre, elije uno u otro registro. Por ejemplo, ante una instrucción que diga «sacar de la pila *(pop)* y llevar a R5», el procesador, si está en modo usuario, copiará en R5 el contenido en la dirección apuntada por R13 (e incrementará R13), mientras que si está en modo supervisor lo hará con la dirección apuntada por R13_SVC (e incrementará R13_SVC). Con el registro de enlace, R14 pasa algo parecido: la instrucción «bifurca y enlaza» (`BL`) guarda la dirección de retorno en R14 si está en modo usuario, o en R14_SVC si está en modo supervisor.

Registro de estado

El registro de estado se llama «CPSR» (Current Program Status Register), y contiene las informaciones esenciales del estado del procesador que hay que guardar al suspender el programa para atender a una interrupción. Precisamente se guardan en SPSR (Saved Program Status Register). Al volver del servicio de la interrupción se copia el contenido de SPSR en CPSR.

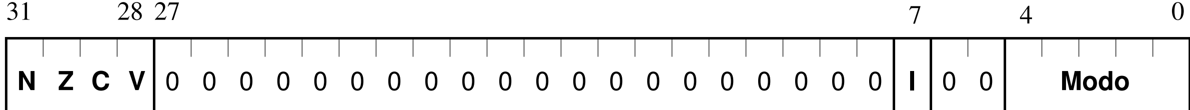


Figura 3.2 Registro de estado, CPSR.

Las informaciones que contiene CPSR son (figura 3.2):

- El **modo**[4], bits 0 a 4: 10000 indica *modo usuario*, y 10011 *modo supervisor*.
- El **permiso de interrupciones**, bit 7: Normalmente, I = 1, pero si I = 0 el procesador no atiende interrupciones.
- Los **indicadores de código de condición**, bits 28 a 31: Sus valores depende del resultado de la última operación realizada en la UAL. Recuerde (apartado 3.3 del Tema 2) que estos valores son:
  N = 1 si ha sido un número negativo
  Z = 1 si ha sido cero (*zero*)
  C = 1 si ha habido acarreo (*carry*)
  V = 1 si ha habido desbordamiento (*overflow*)

Los demás bits de CPSR tienen siempre el valor 0.

3.2. Modelo procesal

La UC sigue el modelo procesal de tres fases de la figura 2.5 con encadenamiento (figura 2.9). Pero hay que matizar un detalle con respecto a esas figuras: como las instrucciones ocupan cuatro bytes, si

[4]En BRM bastaría un bit para indicar el modo, pero conservamos cinco para no perder la compatibilidad con ARM, que tiene siete modos de funcionamiento.

una instrucción está en la dirección i, la siguiente no está en $i + 1$, sino en $i + 4$. Consecuentemente, el registro PC no se incrementa de 1 en 1, sino de 4 en 4.

En este momento conviene ver el conflicto en el encadenamiento que se genera con las instrucciones de bifurcación y sus consecuencias. Recuerde el ejemplo del apartado 2.7 referente a la suma de las componentes de un vector. Las instrucciones del bucle, en pseudocódigo, eran:

Dirección	Instrucción
$I + 12$	Copiar en R3 el contenido de la palabra de la MP *apuntada* por R0 e incrementar el contenido de R0
$I + 16$	Sumar al contenido de R2 el de R3
$I + 20$	Decrementar el contenido de R1 en una unidad
$I + 24$	Si el contenido de R1 no es cero, *bifurcar a* la instrucción que está en la dirección que resulta de restar 20 al contador de programa
$I + 28$	[Contenido de la palabra de dirección $I + 28$]
$I + 32$	[Contenido de la palabra de dirección $I + 32$]
...	...

Fíjese en lo que ocurre en la cadena a partir de un instante, t, en el que empieza la fase de lectura de la instrucción de bifurcación, suponiendo que en ese momento se cumple la condición (figura 3.3). Es en la fase de ejecución de esa instrucción, en el intervalo de $t + 2$ a $t + 3$, cuando la UC calcula la dirección efectiva, $I + 32 - 20 = I + 12$, y la introduce en el contador de programa. Pero la UC, suponiendo que la instrucción siguiente está en la dirección $I + 28$, ya ha leído y descodificado el contenido de esa dirección, y ha leído el contenido de la siguiente, que tiene que desechar (zonas sombreadas en la figura). Por tanto, en el instante $t + 3$ se «vacía» la cadena y empieza un nuevo proceso de encadenamiento, habiéndose perdido dos ciclos de reloj[5].

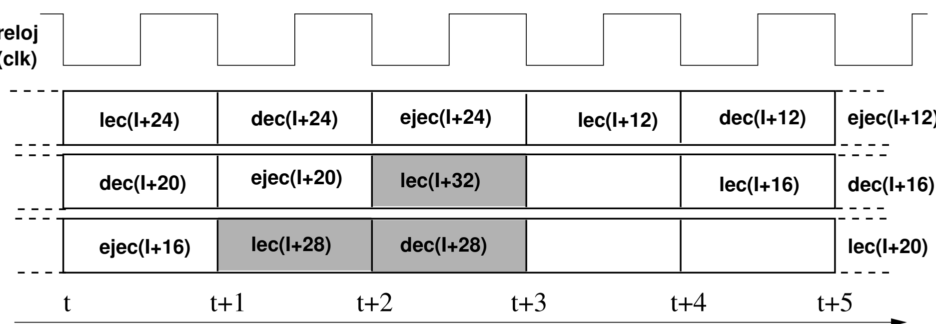


Figura 3.3 Bifurcación en la cadena.

Así pues, las instrucciones de bifurcación retrasan la ejecución, tanto más cuanto menos instrucciones tenga el bucle. En el apartado 4.5 veremos que en BRM es posible, en ciertos casos, prescindir de instrucciones de bifurcación, gracias a que todas las instrucciones son condicionadas.

[5]En algunos modelos de ARM la cadena tiene profundidad 4 o más, por lo que la penalización es de tres o más ciclos. Y mucho mayor en los procesadores que tienen cadenas de gran profundidad. Generalmente la UC incluye una pequeña memoria («*prefetch buffer*») en la que se adelanta la lectura de varias instrucciones y unos circuitos que predicen si la condición de un bifurcación va a cumplirse o no, y actúan en consecuencia. Pero esto es propio del nivel de micromáquina, que no estudiamos aquí.

3.3. Modelo funcional

Las instrucciones aritméticas de BRM operan sobre enteros representados en treinta y dos bits.

- El convenio para número negativos es el de complemento a 2.
- El convenio de almacenamiento en memoria es *extremista menor*.
- Las direcciones de la MP deben estar *alineadas* (figura 2.11(a)): un número representado en una palabra (32 bits) tiene que tener su byte menos significativo en una dirección múltiplo de cuatro.

Representación de instrucciones

Todas las instrucciones ocupan treinta y dos bits y también deben estar almacenadas en la MP en direcciones alineadas: toda instrucción tiene que estar en una dirección múltiplo de cuatro (la dirección de una palabra es la de su byte menos significativo).

Una particularidad de esta arquitectura es que *todas las instrucciones pueden ser condicionadas*, es decir, ejecutarse, o no, dependiendo del estado de los indicadores. En la mayoría de los procesadores solo las bifurcaciones pueden ser condicionadas.

Hay cuatro tipos de instrucciones, y cada tipo tiene su formato. Pero los cuatro formatos comparten dos campos (figura 3.4):

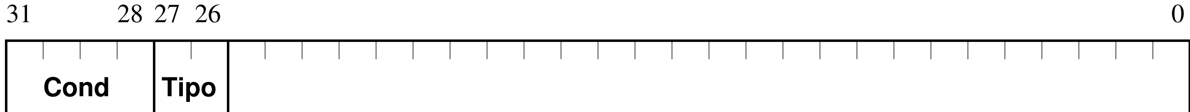


Figura 3.4 Formato de instrucciones: campos «condición» y «tipo».

- En los cuatro bits más significativos, «**Cond**» codifica la condición para que la instrucción se ejecute. La tabla 3.1 resume las dieciséis condiciones. Para cada una se da también un código nemónico, que es el que se utiliza como sufijo del código de la instrucción en el lenguaje ensamblador. Por ejemplo, el código de la instrucción de bifurcación es «`B`». Pero «`BEQ`» es «bifurcación si cero» (Z = 1). «`BAL`» *(«branch always»)* es lo mismo que «`B`»: «bifurcación incondicional». En la tabla, «significado», cuando contiene un signo de comparación, se refiere a la condición expresada en términos de lo que resulta si previamente se ha realizado en la UAL una operación de comparación de dos operandos (instrucción `CMP`, que veremos en el siguiente apartado). Puede usted comprobar que estos significados son coherentes con lo que vimos sobre los indicadores en el apartado 3.3 del Tema 2.
- Los dos bits siguientes, «**Tipo**», indican el tipo de instrucción:

  Tipo 00: Instrucciones de procesamiento y de movimiento de datos entre registros (dieciséis instrucciones).

  Tipo 01: Instrucciones de transferencia de datos con el exterior (cuatro instrucciones).

  Tipo 10: Instrucciones de bifurcación (dos instrucciones).

  Tipo 11: Instrucción de interrupción de programa.

  A diferencia de otros procesadores, éste no tiene instrucciones específicas para desplazamientos y rotaciones: estas operaciones se incluyen en las instrucciones de procesamiento y movimiento.

Binario	Hexadecimal	Nemónico	Condición	Significado
0000	0	`EQ`	Z = 1	=
0001	1	`NE`	Z = 0	≠
0010	2	`CS`	C = 1	≥ sin signo
0011	3	`CC`	C = 0	< sin signo
0100	4	`MI`	N = 1	Negativo
0101	5	`PL`	N = 0	Positivo
0110	6	`VS`	V = 1	Desbordamiento
0111	7	`VC`	V = 0	No desbordamiento
1000	8	`HI`	C = 1 y Z = 0	> sin signo
1001	9	`LS`	C = 0 o Z = 1	≤ sin signo
1010	A	`GE`	N = V	≥ con signo
1011	B	`LT`	N≠V	< con signo
1100	C	`GT`	Z = 0 y N = V	> con signo
1101	D	`LE`	Z = 1 o N≠V	≤ con signo
1110	E	`AL`		Siempre
1111	F	`NV`		Nunca

Tabla 3.1 Códigos de condición.

Según esto, BRM tendría un número muy reducido de instrucciones: veintitrés. Pero, como enseguida veremos, la mayoría tienen variantes. De hecho, desde el momento en que todas pueden ser condicionadas, ese número ya se multiplica por quince (la condición «nunca» convierte a todas las instrucciones en la misma: no operación).

Describimos a continuación todas las instrucciones, sus formatos y sus variantes. Para expresar abreviadamente las transferencias que tienen lugar usaremos una notación ya introducida antes, en la figura 2.5: si Rd es un registro, (Rd) es su contenido, y «(Rf) → Rd» significa «copiar el contenido de Rf en el registro Rd».

Debe usted entender esta descripción como una referencia. Lo mejor es que haga una lectura rápida, sin pretender asimilar todos los detalles, para pasar al capítulo siguiente y volver a éste cuando sea necesario.

3.4. Instrucciones de procesamiento y de movimiento

Estas instrucciones realizan operaciones aritméticas y lógicas sobre dos operandos (o uno, en el caso de la complementación), y de «movimiento» de un registro a otro, o de una constante a un registro. El entrecomillado es para resaltar que el nombre es poco afortunado: cuando algo «se mueve» desaparece de un lugar para aparecer en otro. Y aquí no es así: una instrucción de «movimiento» de R0 a R1 lo que hace, realmente, es *copiar* en R1 el contenido de R0, sin que desaparezca de R0.

El formato de estas instrucciones es el indicado en la figura 3.5.

Uno de los operandos, el «operando1», es el contenido de un registro (R0 a R15) codificado en los cuatro bits del campo «Rn». El otro depende del campo «Op2» y del bit I. El resultado queda (en su caso) en el registro codificado en el campo «Rd» (registro destino). Las dieciséis operaciones codificadas en el campo «Cop» son las listadas en la tabla 3.2.

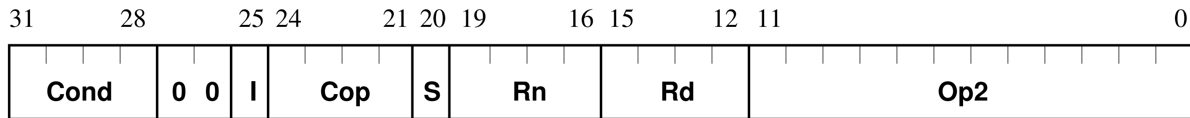


Figura 3.5 Formato de las instrucciones de procesamiento y movimiento.

Cop	Hex.	Nemónico	Acción
0000	0	`AND`	operando1 AND operando2 → Rd
0001	1	`EOR`	operando1 OR exclusivo operando2 → Rd
0010	2	`SUB`	operando1 – operando2 → Rd
0011	3	`RSB`	operando2 – operando1 → Rd
0100	4	`ADD`	operando1 + operando2 → Rd
0101	5	`ADC`	operando1 + operando2 + C → Rd
0110	6	`SBC`	operando1 – operando2 + C –1 → Rd
0111	7	`RSC`	operando2 – operando1 + C –1 → Rd
1000	8	`TST`	como AND, pero no se escribe en Rd
1001	9	`TEQ`	como EOR, pero no se escribe en Rd
1010	A	`CMP`	como SUB, pero no se escribe en Rd
1011	B	`CMN`	como ADD, pero no se escribe en Rd
1100	C	`ORR`	operando1 OR operando2 → Rd
1101	D	`MOV`	operando2 (el operando1 se ignora) → Rd
1110	E	`BIC`	operando1 AND NOT operando2 → Rd
1111	F	`MVN`	NOT operando2 (el operando1 se ignora) → Rd

Tabla 3.2 Códigos de operación de las instrucciones de procesamiento y movimiento.

Los «nemónicos» son los nombres que reconoce el lenguaje ensamblador. Por defecto, entenderá que la instrucción se ejecuta sin condiciones, o sea, el ensamblador, al traducir, pondrá «1110» en el campo «Cond» (tabla 3.1). Para indicar una condición se añade el sufijo correspondiente según la tabla 3.1. Así, `ADDPL` significa «sumar si N = 0».

El bit «S» hace que el resultado, además de escribirse en Rd (salvo en los casos de `TST`, `TEQ`, `CMP` y `CMN`) afecte a los indicadores N, Z, C y V (figura 3.2): si S = 1, los indicadores se actualizan de acuerdo con el resultado; en caso contrario, no. Para el ensamblador, por defecto, S = 0; si se quiere que el resultado modifique los indicadores se añade «S» al código nemónico: `ADDS`, `ADDPLS`, etc. Sin embargo, las instrucciones `TST`, `TEQ`, `CMP` y `CMN` siempre afectan a los indicadores (el ensamblador siempre pone a uno el bit S).

Nos falta ver cómo se obtiene el operando 2, lo que depende del bit «I»: si I = 1, se encuentra *inmediatamente* en el campo «Op2»; si I = 0 se calcula a través de otro registro.

Operando inmediato

En los programas es frecuente tener que poner una constante en un registro, u operar con el contenido de un registro y una constante. Los procesadores facilitan incluir el valor de la constante en la propia instrucción. Se dice que es un *operando inmediato*. Pero si ese operando puede tener cualquier valor representable en 32 bits, ¿cómo lo incluimos en la instrucción, que tiene 32 bits?[6]

[6]Recuerde (apartado 2.7) que BRM es un RISC. En los CISC no existe este problema, porque las instrucciones tienen un número variable de bytes.

Para empezar, la mayoría de las constantes que se usan en un programa son números pequeños. Según el formato de la figura 3.5, disponemos de doce bits, por lo que podríamos incluir números comprendidos entre 0 y $2^{12} - 1 = 4{,}095$ (luego veremos cómo los números pueden ser también negativos).

Concretemos con un primer ejemplo. A partir de ahora vamos a ir poniendo ejemplos de instrucciones y sus codificaciones en binario (expresadas en hexadecimal) obtenidas con un ensamblador. Debería usted hacer el ejercicio, laborioso pero instructivo y tranquilizador, de comprobar que las codificaciones binarias son acordes con los formatos definidos.

Para introducir el número 10 (decimal) en R11, la instrucción, codificada en hexadecimal y en ensamblador es:

```
E3A0B00A      MOV R11,#10
```

(Ante un valor numérico, 10 en este ejemplo, el ensamblador lo entiende como decimal. Se le puede indicar que es hexadecimal o binario anteponiéndole los prefijos habituales, «0x» o «0b»).

Escribiendo en binario y separando los bits por campos según la figura 3.5 resulta:

```
1110 - 00 - 1 - 1101 - 0 - 0000 - 1011 - 0000000000001010
```

Cond = 1110 (0xE): sin condición
Tipo = 00
I = 1: operando inmediato
Cop = 1101 (0xD): `MOV`
S = 0: es `MOV`, no `MOVS`
Rn = 0: Rn es indiferente con la instrucción `MOV` y operando inmediato
Rd = 1011 (0xB): R11
Op2 = 1010 (0xA): 10

(Este tipo de comprobación es el que debería usted hacer para los ejemplos que siguen).

¿Y si el número fuese mayor que 4095? El convenio de BRM es que solo ocho de los doce bits del campo Op2 son para el número (lo que, en principio, parece empeorar las cosas), y los cuatro restantes dan la magnitud de una *rotación a la derecha* de ese número multiplicada por 2 (figura 3.6). Conviene volver a mirar la figura 3.1: la entrada 2 a la UAL (en este caso, procedente del registro RID a través del bus B) pasa por una unidad de desplazamiento, y una de las posibles operaciones en esta unidad es la de rotación a la derecha (ROR, figura 3.2 del Tema 2). Los ocho bits del campo «inmed_8» se extienden a 32 bits con ceros a la izquierda y se les aplica una rotación a la derecha de tantos bits como indique el campo «rot» multiplicado por dos. Si «rot» = 0 el resultado está comprendido entre 0 y el valor codificado en los ocho bits de «inmed_8» (máximo: $2^8 = 255$). Con otros valores de «rot» se pueden obtener valores de hasta 32 bits.

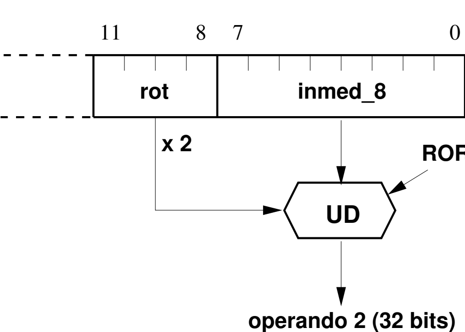


Figura 3.6 Operando inmediato.

Veamos cómo funciona. La traducción de `MOV R0,#256` es:

```
E3A00C01    MOV R0,#256
```

El ensamblador ha puesto 0x01 en «inmed_8» y 0xC = 12 en el campo «rot». Cuando el procesador ejecute esta instrucción entenderá que tiene que tomar un valor de 32 bits con todos ceros salvo 1 en

el menos significativo y rotarlo $12 \times 2 = 24$ bits *a la derecha*. Pero rotar 24 bits a la derecha da el mismo resultado que rotar $32 - 24 = 8$ bits *a la izquierda*. Resultado: el operando 2 tiene un 1 en el bit de peso 8 y los demás son ceros; $2^8 = 256$. Lo maravilloso del asunto es que el ensamblador hace el trabajo por nosotros: no tenemos que estar calculando la rotación a aplicar para obtener el operando deseado.

Probemos con un número muy grande:

```
E3A004FF    MOV R0,#0xFF000000
```

En efecto, como «rot» = 4, el número de rotaciones a la izquierda es $32-4\times 2 = 24$. Al aplicárselas a 0x000000FF resulta 0xFF000000.

¿Funcionará este artilugio para cualquier número entero? Claro que no: solamente para aquellos que se escriban en binario como una sucesión de ocho bits con un número par de ceros a la derecha y a la izquierda hasta completar 32. Instrucciones como `MOV R0,#257`, `MOV R0,#4095`... no las traduce el ensamblador (da errores) porque no puede encontrar una combinación de «inmed_8» y «rot» que genere el número. La solución en estos casos es introducir el número en una palabra de la memoria y acceder a él con una instrucción `LDR`, que se explica en el siguiente apartado. De nuevo, la buena noticia para el programador es que no tiene que preocuparse de cómo se genera el número: el ensamblador lo hace por él, como veremos en el apartado 4.3.

En los ejemplos hemos utilizado `MOV` y R0. Pero todo es aplicable al segundo operando de cualquiera de las instrucciones de este tipo y a cualquier registro. (Teniendo en cuenta que R13, R14 y R15 tienen funciones especiales. Así, `MOV R15,#100` es una transferencia de control a la dirección 100).

¿Cómo proceder para los operandos inmediatos negativos? Con las instrucciones aritméticas no hay problema, porque en lugar de escribir `ADD R1,#-5` se puede poner `SUB R1,#5`[7]. Pero ¿para meter -5 en un registro? Probemos:

```
E3E00004    MOV R0,#-5
```

Otra vez el ensamblador nos ahorra trabajo: ha traducido igual que si se hubiese escrito `MVN R0,#4` (si analiza usted el binario comprobará que el código de operación es 1111). En efecto, según la tabla 3.2, `MVN` (*move negativo*), produce el complemento a 1. Y el complemento a 1 de 4 es igual que el complemento a 2 de 5.

Operando en registro

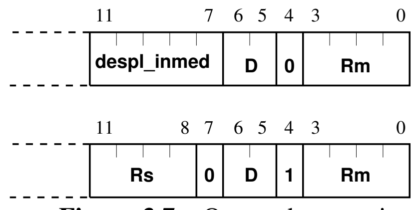


Figura 3.7 Operando en registro.

Cuando el bit 25, o bit I, es I = 0, el operando 2 se obtiene a partir de otro registro, el indicado en los cuatro bits menos significativos del campo Op2 (Rm). Si el resto de los bits de ese campo (de 4 a 11) son ceros, el operando es exactamente el contenido de Rm. Pero si no es así, ese contenido se desplaza o se rota de acuerdo con los siguientes convenios (figura 3.7):

- bit 4: si es 0, los bits 7 a 11 contienen el número de bits que ha de desplazarse Rm.
  si es 1, el número de bits está indicado en los cinco bits menos significativos de otro registro, Rs.

[7]El ensamblador GNU entiende `ADD R1,#-5` y lo traduce igual que `SUB R1,#5`. Sin embargo, ARMSim# da error (es obligatorio escribirlo como `SUB`).

- bits 5 y 6 (D): tipo de desplazamiento (recuerde el apartado 3.3 y la figura 3.2 del Tema 2):
  D = 00: desplazamiento lógico a la izquierda (LSL).
  D = 01: desplazamiento lógico a la derecha (LSR).
  D = 10: desplazamiento aritmético a la derecha (ASR).
  D = 11: rotación a la derecha (ROR).

A continuación siguen algunos ejemplos de instrucciones de este tipo, con la sintaxis del ensamblador y su traducción, acompañada cada una de un pequeño comentario (tras el símbolo «@»). Debería usted comprobar mentalmente que la instrucción hace lo que dice el comentario. Cuando, después de estudiar el siguiente capítulo, sepa construir programas podrá comprobarlo simulando la ejecución.

Ejemplos con operando inmediato:

```
E2810EFA   ADD R0, R1, #4000       @ (R1) + 4000 → R0
E2500EFF   SUBS R0, R0, #4080      @ (R0) - 4080 → R0,
                                   @  y pone indicadores según el resultado
E200001F   AND R0, R0, #0x1F       @ 0 → R0[5..31]
                                   @    (pone a cero los bits 5 a 31 de R0)
E38008FF   ORR R0, R0, #0xFF0000   @ 1 → R0[24..31]
                                   @    (pone a uno los bits 24 a 31 de R0)
E3C0001F   BIC R0, R0, #0x1F       @ 0 → R0[0..4]
                                   @    (pone a cero los bits 0 a 4 de R0)
```

La última operación sería equivalente a AND R0, R0, #0xFFFFE0, pero ese valor inmediato no se puede generar.

Ejemplos con operando en registro:

```
E1A07000   MOV R7, R0              @ (R0) → R7
E0918000   ADDS R8, R1, R0         @ (R1) + (R0) → R8,  y pone indicadores
E0010002   AND R0, R1, R2          @ (R1) AND (R2) → R0 y pone indicadores
E1A01081   MOV R1, R1, LSL #1      @ 2×(R1) → R1
E1A00140   MOV R0, R0, ASR #2      @ (R0)÷4 → R0
E1B04146   MOVS R4, R6, ASR #5     @ (R6)÷32 → R4, y pone indicadores
E0550207   SUBS R0, R5, R7, LSL #4 @ (R5) + 16×(R7) → R0,
                                   @  y pone indicadores
```

Movimientos con los registros de estado

Como hay dos registros de estado, CPSR y SPSR (apartado 3.1), que no están incluidos en la memoria local, se necesitan cuatro instrucciones para copiar su contenido a un registro o a la inversa.

Estas cuatro instruccciones son también de este grupo (T = 00) y comparten los códigos de operación con los de TST, TEQ, CMP y CMN, pero con S = 0 (TST, etc. siempre tienen S = 1, de lo contrario no harían nada). No las vamos a utilizar en los ejercicios ni en las prácticas. Las presentamos esquemáticamente solo para completar la descripción de BRM. La figura 3.8 muestra los formatos.

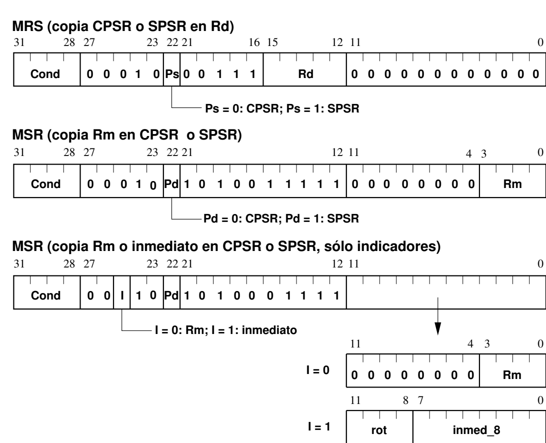


Figura 3.8 Instrucciones MRS y MSR

`MRS` simplemente hace una copia del contenido de un registro de estado en un registro de la ML, pero hay que tener en cuenta que en modo usuario SPSR no es accesible:

```
E10F0000    MRS R0, CPSR  @ (CPSR) → R0
E14F0000    MRS R0, SPSR  @ (SPSR) → R0 (solo en modo supervisor)
```

Para copiar en un registro de estado hay dos versiones. En una se modifica el registro completo (pero en modo usuario solo pueden modificarse los indicadores, o «flags»: bits N, Z, C y V, figura 3.2), y en otra solo los indicadores. Esta segunda tiene, a su vez, dos posibilidades: copiar los cuatro bits más significativos de un registro (Rm), o los de un valor inmediato de 8 bits extendido a 32 bits y rotado a la derecha.

Ejemplos:

```
E129F005    MSR CPSR, R5 @ Si modo supervisor, (R5) → CPSR
                         @ Si modo usuario, (R5[28..31])  → CPSR[28..31]
E128F005    MSR CPSR_flg, R5 @igual que la anterior en modo usuario
E328F60A    MSR CPSR_flg, #0xA00000 @ 1 → N, C; 0 → Z, V
```

3.5. Instrucciones de transferencia de datos

Las instrucciones de transferencia de datos son las que copian el contenido de un registro en una dirección de la MP o en un puerto, o a la inversa. Su formato es el de la figura 3.9

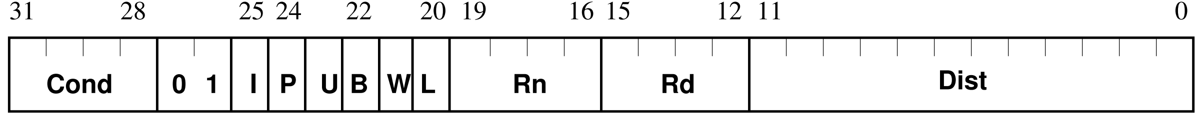


Figura 3.9 Formato de las instrucciones de transferencia de datos.

A pesar de la aparente complejidad del formato, hay solo cuatro instrucciones, que se distinguen por los bits 20 (L: *load*) y 22 (B: *byte*). En la tabla 3.5 se dan sus códigos en ensamblador y se resumen sus funciones.

L	B	Nemónico	Acción	Significado
0	0	`STR`	(Rd) → M[DE]	Copia el contenido de Rd en la palabra de dirección DE
0	1	`STRB`	(Rd[0..7]) → M[DE]	Copia los ocho bits menos significativos de Rd en el byte de dirección DE
1	0	`LDR`	(M[DE]) → Rd	Copia la palabra de dirección DE en el registro Rd
1	1	`LDRB`	(M[DE]) → Rd[0..7] 0 → Rd[8..31]	Copia el byte de dirección DE en los ocho bits menos significativos de Rd, y pone los demás bits a cero

Tabla 3.3 Instrucciones de transferencia de datos.

«DE» es la *dirección efectiva* (apartado 2.7), que se obtiene a partir del contenido de un **registro de base**, Rn (que puede actuar como **registro de índice**). «M[DE]» puede ser una posición de la memoria o un puerto de entrada/salida, y «(M[DE])» es su contenido.

El cálculo de la dirección efectiva depende de los bits I (25) y P (24). Si I = 0, el contenido del campo «Dist» es una «distancia» *(offset)*, un número entero que se suma (si el bit 23 es U = 1) o se resta (si U = 0) al contenido del registro de base Rn. Si I = 1, los bits 0 a 3 codifican un registro que contiene la distancia, sobre el que se puede aplicar un desplazamiento determinado por los bits 4 a 11, siguiendo el mismo convenio que para un operando en registro (figura 3.7). Por su parte, P determina cómo se aplica esta distancia al registro de base Rn. W *(write)* no influye en el modo de direccionamiento, indica solamente si el resultado de calcular la dirección efectiva se escribe o no en el registro de base.

Resultan así cuatro modos de direccionamiento:

Modo postindexado inmediato (I = 0, P = 0)

- distancia = contenido del campo Dist
- DE = (Rn)
- (Rn)±distancia → Rn (independientemente de W)

Este modo es útil para acceder sucesivamente (dentro de un bucle) a elementos de un vector almacenado a partir de la dirección apuntada por Rn. Por ejemplo:

```
E4935004    LDR R5, [R3], #4     @ (M[(R3)]) → R5; (R3) + 4 → R3
    @ (carga en R5 la palabra de dirección  apuntada por R3 e incrementa R3)
```

Si R3 apunta inicialmente al primer elemento del vector y cada elemento ocupa una palabra (cuatro bytes), cada vez que se ejecute la instrucción R3 quedará apuntando al elemento siguiente, suponiendo que estén almacenados en direcciones crecientes. Si estuviesen en direcciones decrecientes pondríamos:

```
E4135004    LDR R5, [R3], #-4    @ (M[(R3)]) → R5; (R3) - 4 → R3
```

Si la distancia es cero resulta un modo de direccionamiento **indirecto a registro** (apartado 2.7):

```
E4D06000  LDRB R6, [R0], #0 @ (M[(R0)]) → R6[0..7]; 0 → R6[8..31]
    @ (carga en R6 el byte de dirección apuntada por R0
    @ y pone los bits más significativos de R6 a cero; R0 no se altera)
```

Podemos escribir la misma instrucción como `LDRB R6, [R0]`, pero en este caso el ensamblador la traduce a una forma equivalente, la que tiene P = 1

Modo preindexado inmediato (I = 0, P = 1)

- distancia = contenido del campo Dist
- DE = (Rn)±distancia
- Si W = 1, (Rn)±distancia → Rn

Observe que el registro de base, Rn, solo se actualiza si el bit 21 (W) está puesto a uno (en el caso anterior siempre se actualiza, siendo indiferente el valor de W). El convenio en el lenguaje ensamblador es añadir el símbolo «!» para indicar que debe actualizarse.

Ejemplos:

```
E5D06000    LDRB R6, [R0]      @ Igual que LDRB R6, [R0, #0]

E5035004    STR R5, [R3, #-4]  @ (R5) → M[(R3)-4] (almacena el contenido
                               @   de R5 en la dirección apuntada por R4
                               @   menos cuatro; R3 no se actualiza)

E5235004    STR R5, [R3, #-4]! @ como antes, pero a R3 se le resta cuatro
```

Si Rn = R15 (es decir, PC) con W = 0 resulta un modo **relativo a programa**:

```
E5DF0014    LDRB R0, [PC, #20] @ carga en R0 el byte de dirección
                               @   de la instrucción actual más 8 más 20
```

Como se explica en el capítulo siguiente, en lenguaje ensamblador lo normal para el direccionamiento relativo es utilizar una «etiqueta» para identificar a la dirección, y el ensamblador se encargará de calcular la distancia.

Modo postindexado con registro (I = 1, P = 0)

En los modos «registro» se utiliza un registro auxiliar, Rm, cuyo contenido se puede desplazar o rotar siguiendo los mismos convenios de la figura 3.7.

- distancia = despl(Rm)
- DE = (Rn)
- (Rn)±distancia → Rn (independientemente de W)

Ejemplos:

```
E6820005  STR R0, [R2], R5            @ (R0) → M[R2]; (R2) + (R5) → R2

E6020005  STR R0, [R2], -R5           @ (R0) → M[R2]; (R2) - (R5) → R2

E6920245  LDR R0, [R2], R5, ASR #4    @ (M[(R2)]) → R0;
                                      @   (R2) + (R5)/16 → R2

06D20285  LDREQ R0, [R2], R5, LSL #5  @ Si Z = 0 no hace nada; si Z = 1,
                                      @    (M[(R2)]) → R0[0..7];
                                      @    0 → R0[8..31];
                                      @    (R2) + (R5)*32 → R2
```

Modo preindexado con registro (I = 1, P = 1)

- distancia = despl(Rm)
- DE = (Rn)±distancia
- Si W = 1, Rn±distancia → Rn

Ejemplos:

```
E7921004   LDR R1, [R2, R4] @ (M[(R2)+(R4)]) → R1
```

```
E7121004   LDR R1, [R2, -R4] @ (M[(R2)-(R4)]) → R1

E7321004   LDR R1, [R2, -R4]! @ (M[(R2)-(R4)]) → R1; (R2)-(R4) → R2

E7921104   LDR R1, [R2, R4, LSL #2] @ (M[(R2)+(R4)*4]) → R1
```

3.6. Instrucciones de transferencia de control

Puesto que el contador de programa es uno más de los registros de la ML, cualquier instrucción que lo modifique realiza una transferencia de control.

Por ejemplo:

- `MOVNE R15, #4096` (o, lo que es lo mismo, `MOVNE PC, #4096`) realiza una bifurcación a la dirección 4.096 si Z = 0. Es un **direccionamiento directo**.
- `MOVMI R15, R5` (o, lo que es lo mismo, `MOVMI PC, R5`) realiza una bifurcación a la dirección apuntada por R5 si N = 1. Es un **direccionamiento indirecto**.

Pero esta forma de bifurcar solo se aplica cuando la dirección de destino está muy alejada de la instrucción, y también para retornar de una interrupción, como veremos en el siguiente apartado. Normalmente se utilizan instrucciones de bifurcación con direccionamiento relativo.

Hay tres instrucciones específicas para las transferencias de control: las de bifurcación y la de interrupción de programa.

Instrucciones de bifurcación

Las dos instrucciones de bifurcación, `B` *(branch)* y `BL` *(branch and link)*, tienen el formato de la figura 3.10. El bit 24 (L) distingue entre `B` (L = 0) y `BL` (L = 1).

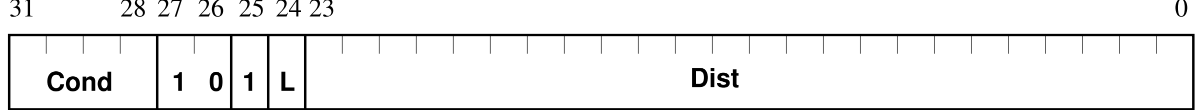


Figura 3.10 Formato de las instrucciones de bifurcación.

Ambas usan direccionamiento relativo: el campo «Dist» se interpreta como un número con signo que se suma al valor que tenga en el momento de su ejecución el contador de programa (dirección de la instrucción más ocho). Pero antes de sumarse, y como todas las instrucciones tienen que estar en direcciones alineadas (múltiplos de cuatro), se desplaza dos bits a la izquierda. De este modo, la dirección efectiva resulta: DE = dirección de la instrucción + 8 + 4×Dist. Con 26 bits se puede representar un número con signo entre -2^{25} y $2^{25} - 1$, es decir, se puede bifurcar a una dirección comprendida en un rango aproximado de ±32 MiB sobre la dirección de la instrucción, lo que es más que suficiente para la mayoría de los casos.

La instrucción `B` simplemente modifica el contenido del registro PC (R15) copiando en él la dirección efectiva. `BL`, además, modifica el contenido del registro de enlace, LR (R14) copiando en él la dirección de la instrucción más cuatro. De este modo, se bifurca a un subprograma del que se vuelve con `MOV PC, LR` (o `MOV R15, R14`).

Veremos ejemplos de uso de estas instrucciones en los apartados 4.2 y 4.6, cuando ya conozcamos cómo el ensamblador nos puede liberar de la tediosa tarea de calcular la distancia.

Instrucción de interrupción de programa

Esta instrucción, de código ensamblador «SWI», tiene un formato muy sencillo, el de la figura 3.11.

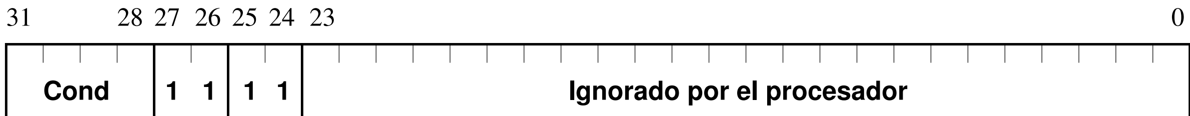


Figura 3.11 Formato de la instrucción SWI.

El efecto de la instrucción es, simplemente, provocar una interrupción del programa en el que está incluida (siempre que la condición se cumpla, como en todas las instrucciones). Como indica la figura, el contenido de los 24 bits menos significativos es indiferente para el procesador, pero la rutina de servicio puede recuperar el contenido de la dirección en la que está almacenada la instrucción y analizar esos bits para realizar la acción oportuna, de acuerdo con algún convenio establecido.

3.7. Comunicaciones con los periféricos e interrupciones

En el apartado 2.8 decíamos que unos procesadores siguen el convenio de que el espacio de direccionamiento para los puertos de periféricos es independiente del de la memoria, mientras que en otros el espacio de direccionamiento es compartido. Pues bien, BRM sigue el segundo convenio. Ciertas direcciones están reservadas para los puertos: para leer de un puerto se utiliza una instrucción LDRB, y para escribir en un puerto una STRB. ¿Qué direcciones son esas? No están fijadas en el diseño del procesador. Los circuitos externos que decodifican las direcciones que el procesador pone en el bus A se ocuparán de activar a la memoria o a un periférico. Por tanto, para programar las operaciones de entrada/salida no basta con conocer el modelo funcional del procesador, es necesario tener el cuenta el diseño de los circuitos externos para saber qué direcciones son de puertos y la función de cada puerto. Veremos un ejemplo concreto en el apartado 4.8.

También nos referíamos a las interrupciones en términos generales en el apartado 2.8: la atención a una interrupción implica salvaguardar el estado del procesador, identificar la causa y ejecutar una **rutina de servicio**.

Y al final del apartado 3.1 ya veíamos algo del caso concreto de BRM: al pasar a atender a una interrupción el procesador hace una copia del contenido del registro CPSR (Current Program Status Register) en el registro SPSR (Saved Program Status Register). En realidad, hace algo más:

- Copia CPSR en SPSR.
- En CPSR pone el modo supervisor e inhibe las interrupciones (modo = 10011, I = 0, figura 3.2).
- Copia el valor actual del contador de programa, PC, en el registro R14_SVC = LR_SVC.
- Introduce en PC la dirección de comienzo de la rutina de servicio.

Todas estas operaciones las hace automáticamente el hardware.

Causas y vectores de interrupción

¿Y cuál es esa dirección que automáticamente se introduce en PC? Como cada causa de interrupción tiene su propia rutina de servicio, depende de la causa. Cada una tiene asignada una dirección fija en la parte baja de la memoria (direcciones 0, 4, 8, etc.) que contiene el llamado **vector de interrupción**. En BRM, cada vector de interrupción es la primera instrucción de la correspondiente rutina[8]. Como esas direcciones reservadas son contiguas, cada vector es una instrucción de transferencia de control a otra dirección en la que comienzan las instrucciones que realmente proporcionan el servicio.

En BRM hay previstas cuatro causas de interrupción. Por orden de prioridad (para el caso de que coincidan en el tiempo) son: *Reset*, *IRQ*, *SWI* e *Instrucción desconocida*. En la tabla 3.4 se indican los orígenes de las causas y las direcciones de los vectores[9].

Interrupción	Causa	Dirección del vector
Reset	Activación de la señal *reset* (figura 3.1)	0x00
Instrucción desconocida	El procesador no puede descodificar la instrucción	0x04
de programa	Instrucción `SWI`	0x08
IRQ	Activación de la señal *irq*	0x18

Tabla 3.4 Tabla de vectores de interrupción.

En el caso de *reset* el procesador abandona la ejecución del programa actual sin guardar CPSR ni la dirección de retorno, pero poniendo el modo supervisor e inhibiendo interrupciones. La rutina de servicio consiste en realizar las operaciones de inicialización, normalmente incluidas en un programa grabado en ROM (apartado 2.5).

Las otras causas solo se atienden si I = 1. Cuando se trata de instrucción desconocida o de `SWI` el procesador, en cuanto la descodifica, realiza inmediatamente las operaciones descritas. En ese momento, el registro PC está apuntando a la instrucción siguiente (dirección de la actual más cuatro). Sin embargo, cuando atiende a una petición de interrupción externa *(IRQ)* el procesador termina de ejecutar la instrucción en curso, por lo que el valor de PC que se salva en LR_SVC es el de la instrucción de la siguiente más ocho.

Retorno de interrupción

Al final de toda rutina de servicio, para volver al programa interrumpido son necesarias dos operaciones (salvo si se trata de *reset*):

- Copiar el contenido del registro SPSR en CPSP (con esto se vuelven a permitir interrupciones y al modo usuario, además de recuperar los valores de los indicadores para el programa interrumpido).
- Introducir en el registro PC el valor adecuado para que se ejecute la instrucción siguiente a aquella en la que se produjo la interrupción.

De acuerdo con lo dicho en los párrafos anteriores, lo segundo es fácil: si la causa de interrupción fue de programa (`SWI`) o instrucción desconocida, basta copiar el contenido de R14_SVC = LR_SVC

[8] En otros procesadores, por ejemplo, los de Intel y los de Motorola, los vectores de interrupción no son instrucciones, sino punteros a los comienzos de las rutinas.

[9] El «hueco» de 12 bytes que hay entre las direcciones 0x0C y 0x18 se debe a que en el ARM hay otras causas no consideradas en BRM.

en PC con una instrucción `MOV PC,LR` (recuerde que estamos en modo supervisor, y el registro LR afectado por la instrucción es LR_SVC). Y si fue *IRQ* hay que restar cuatro unidades: `SUB PC,LR,#4`.

Pero antes hay que hacer lo primero, y esto plantea un conflicto. En efecto, podríamos pensar en aplicar las instrucciones que vimos al final del apartado 3.4 para, utilizando un registro intermedio, hacer la copia de SPSR en CPSR. Sin embargo, en el momento en que esta copia sea efectiva habremos vuelto al modo usuario, y habremos perdido la dirección de retorno, puesto que LR ya no será LR_SVC.

Pues bien, los diseñadores de la arquitectura ARM pensaron, naturalmente, en este problema e ingeniaron una solución elegante y eficiente. ¿Recuerda la función del bit «S» en las instrucciones de procesamiento y movimiento (figura 3.5)? Normalmente, si está puesto a uno, los indicadores se modifican de acuerdo con el resultado. Pero en el caso de que el registro destino sea PC es diferente: automáticamente copia SPSR en CPSR. Por tanto, la última instrucción de una rutina de servicio debe ser:

- `MOVS PC,LR` (o `MOVS R15,R14`) si la interrupción era de programa o de instrucción desconocida.
- `SUBS PC,LR,#4` (o `SUBS R15,R14,#4`) si la interrupción era *IRQ*.

De este modo, con una sola instrucción se realizan las dos operaciones.

Mapa de memoria (ejemplo)

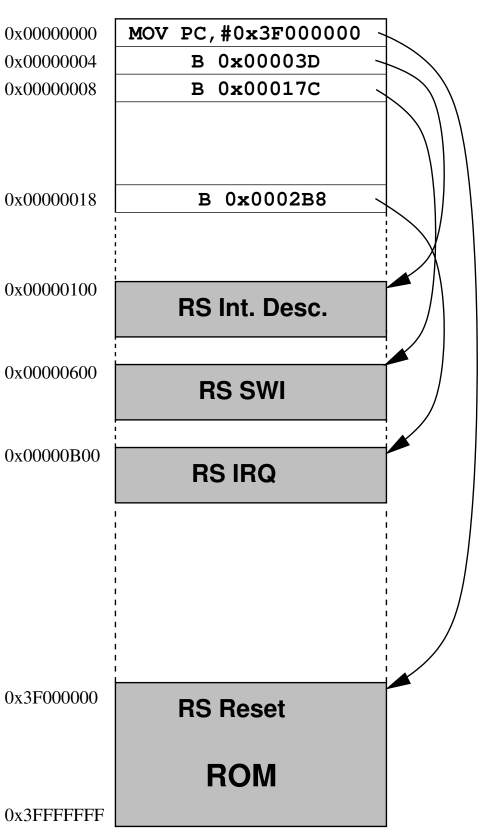


Figura 3.12. Mapa de memoria de vectores y rutinas de servicio.

Las direcciones de los vectores están fijadas por el hardware: el procesador está diseñado para introducir automáticamente su contenido en el registro PC una vez identificada la causa. Pero los contenidos no se rellenan automáticamente, se introducen mediante instrucciones `STR`. Normalmente estas instrucciones están incluidas en el programa de inicialización y los contenidos no se alteran mientras el procesador está funcionando. Podríamos tener, por ejemplo, el mapa de memoria de la figura 3.12, una vez terminada la inicialización. Se trata de una memoria de 1 GiB en la que las direcciones más altas, a partir de 0x3F000000, son ROM. Precisamente en esa dirección comienza la rutina de servicio de reset, por lo que en la dirección 0 se ha puesto como vector de interrupción una transferencia de control. Como la distancia es mayor que 32 MiB, se utiliza una `MOV` con operando inmediato (gracias a que el operando puede obtenerse mediante rotación).

Las rutinas de servicio de las otras tres causas se suponen cargadas en direcciones alcanzables con la distancia de instrucciones de bifurcación, por lo que los vectores de interrupción son instrucciones de este tipo.

En el apartado 4.8 comentaremos lo que pueden hacer las rutinas de servicio.

Capítulo 4

Programacion de BRM

Ya hemos dicho en el apartado 1.2 que los lenguajes ensambladores, a diferencia de los lenguajes de alto nivel, dependen de la arquitectura (ISA) del procesador. Pero no solo es que cada procesador tenga su lenguaje, es que para un mismo procesador suele haber distintos convenios sintácticos para codificar las instrucciones de máquina y otras informaciones en ensamblador.

Los programas de este capítulo están escritos en el ensamblador GNU, que es también el que se utiliza en el simulador ARMSim#. En el apartado A.3 del apéndice se mencionan otros lenguajes ensambladores para la arquitectura ARM. El ensamblador GNU es insensible a la caja *(case insensitive)*. Es decir, se pueden utilizar indistintamente letras mayúsculas o minúsculas: «`MOV`» es lo mismo que «`mov`», o que «`MoV`»... «`R1`» es lo mismo que «`r1`», etc. En este capítulo usaremos minúsculas.

Los listados se han obtenido con un **ensamblador cruzado** (apartado A.3): un programa ensamblador que se ejecuta bajo un sistema operativo (Windows, Linux, MacOS...) en una arquitectura (x86 en este caso) y genera código binario para otra (ARM en este caso). Los mismos programas se pueden introducir en ARMSim#, que se encargará de ensamblarlos y de simular la ejecución. Debería usted no solo probar los ejemplos que se presentan, sino experimentar con variantes.

La palabra «ensamblador» tiene dos significados: un tipo de lenguaje *(assembly language)* y un tipo de procesador *(assembler)*. Pero el contexto permite desambiguarla: si decimos «escriba un programa en ensamblador para...» estamos aplicando el primer significado, mientras que con «el ensamblador traducirá el programa...» aplicamos el segundo.

4.1. Primer ejemplo

Construimos un programa en lenguaje ensamblador escribiendo secuencialmente instrucciones con los convenios sintácticos que hemos ido avanzando en el capítulo anterior. Por ejemplo, para sumar dos números enteros, 8 y 10, y dejar el resultado en R2, la secuencia puede ser:

```
mov   r0,#8
mov   r1,#10
add   r2,r0,r1
```

Un conjunto de instrucciones en lenguaje ensamblador se llama **programa fuente**. Se traduce a binario con un ensamblador, que da como resultado un **programa objeto**, o **código objeto** en un fichero binario (Tema 2, apartado 5.5).

Pero a nuestro programa fuente le faltan al menos dos cosas:

- Suponga que el código objeto se carga a partir de la dirección 0x1000 de la MP. Las tres instrucciones ocuparán las direcciones 0x1000 a 0x100B inclusive. El procesador las ejecutará en secuencia (no hay ninguna bifurcación), y, tras la tercera, irá a ejecutar lo que haya a partir de la dirección 0x100C. Hay que decirle que no lo haga, ya que el programa ha terminado. Para esto está la instrucción `SWI` (apartado 3.6). Normalmente, tendremos un sistema operativo o, al menos, un programa de control, que se ocupa de las interrupciones de programa. En el caso del simulador ARMSim#, `swi 0x11` hace que ese programa de control interprete, al ver «0x11», que el programa ha terminado (tabla A.1). Por tanto, añadiremos esta instrucción.
- El programa fuente puede incluir informaciones u órdenes para el programa ensamblador que no afectan al código resultante (no se traducen por ninguna instrucción de máquina). Se llaman **directivas**. Una de ellas es «`.end`» (en el ensamblador GNU todas las directivas empiezan con un punto). Así como la *instrucción* `SWI` provoca, *en tiempo de ejecución*, una interrupción para el procesador (hardware), la *directiva* `.end` le dice al ensamblador (procesador software), *en tiempo de traducción*, que no siga leyendo porque se ha acabado el programa fuente. Siempre debe ser la última línea del programa fuente.

Iremos viendo más directivas. De momento vamos a introducir otra: «`.equ`» define un símbolo y le da un valor; el ensamblador, siempre que vea ese símbolo, lo sustituye por su valor.

Por último, para terminar de acicalar nuestro ejemplo, podemos añadir **comentarios** en el programa fuente, que el ensamblador simplemente ignorará. En nuestro ensamblador se pueden introducir de dos maneras:

- Empezando con la pareja de símbolos «`/*`», todo lo que sigue es un comentario hasta que aparece la pareja «`*/`».
- Con el símbolo «`@`», todo lo que aparece *hasta el final de la línea* es un comentario[1].

Para facilitar la legibilidad se pueden añadir espacios en blanco al comienzo de las líneas. Entre el código nemónico (por ejemplo, «`mov`») y los operandos (por ejemplo, «`r0,r1`») debe haber *al menos* un espacio en blanco. Los operandos se separan con una coma seguida o no de espacios en blanco.

De acuerdo con todo esto, nuestro programa fuente queda así:

```
/************************************************
*                                               *
*              Primer ejemplo                   *
*                                               *
************************************************/
.equ   cte1,10
.equ   cte2,8
   mov   r0,#cte1   @ Carga valores en R0
   mov   r1,#cte2   @  y R1
   add   r2,r0,r1   @  y hace (R0) + (R1) → R2
   swi   0x11
.end
```

Si le entregamos ese programa fuente (guardado en un fichero de texto cuyo nombre debe acabar en «`.s`») al ensamblador y le pedimos que, aparte del resultado principal (que es un programa objeto

[1] En el ensamblador GNU se puede usar, en lugar de «`@`», «`#`» (siempre que esté al principio de la línea), y en ARMSim# se puede usar «`;`».

en un fichero binario), nos dé un listado (en el apartado A.3 se explica cómo hacerlo), obtenemos lo que muestra el programa 4.1.

```
                         /*********************************************
                         *                                             *
                         *              Primer ejemplo                 *
                         *                                             *
                         *********************************************/
                         .equ   cte1,10
                         .equ   cte2,8
00000000 E3A0000A           mov   r0,#cte1   @ Carga valores en R0
00000004 E3A01008           mov   r1,#cte2   @  y R1
00000008 E0802001           add   r2,r0,r1   @  y hace (R0) + (R1) → R2
0000000C EF000011           swi   0x11
                         .end
```

Programa 4.1 Comentarios y directivas equ y end.

Vemos que nos devuelve lo mismo que le habíamos dado (el programa fuente), acompañado de dos columnas que aparecen a la izquierda, con valores en hexadecimal. La primera columna son direcciones de memoria, y la segunda, contenidos de palabras que resultan de haber traducido a binario las instrucciones.

Si carga usted el mismo fichero en el simulador ARMSim# verá en la ventana central el mismo resultado. Luego, puede simular la ejecución paso a paso poniendo puntos de parada (apartado A.2) para ver cómo van cambiando los contenidos de los registros. La única diferencia que observará es que la primera dirección no es 0x0, sino 0x1000. El motivo es que el simulador carga los programas a partir de esa dirección[2].

Este listado y los siguientes se han obtenido con el ensamblador GNU (apartado A.3), y si prueba usted a hacerlo notará que la segunda columna es diferente: los contenidos de memoria salen escritos «al revés». Por ejemplo, en la primera instrucción no aparece «E3A0000A», sino «0A00A0E3». En efecto, recuerde que el convenio es extremista menor, por lo que el byte menos significativo de la instrucción, que es 0A, estará en la dirección 0, el siguiente, 00, en la 1, y así sucesivamente. Sin embargo, la otra forma es más fácil de interpretar (para nosotros, no para el procesador), por lo que le hemos pasado al ensamblador una opción para que escriba como extremista mayor. En el diseño de ARMSim# han hecho igual: presentan las instrucciones como si el convenio fuese extremista mayor, pero internamente se almacenan como extremista menor. Es fácil comprobarlo haciendo una vista de la memoria por bytes (apartado A.2).

4.2. Etiquetas y bucles

En el apartado 2.7, al hablar de modos de direccionamiento, pusimos, en pseudocódigo, un ejemplo de bucle con una instrucción de bifurcación condicionada, e, incluso, adelantamos algo de su codificación en ensamblador. Como ese ejemplo requiere tener unos datos guardados en memoria, y aún no hemos visto la forma de introducir datos con el ensamblador, lo dejaremos para más adelante, y codificaremos otro que maneja todos los datos en registros.

[2]Aunque esto se puede cambiar desde el menú: File > Preferences > Main Memory.

Los «números de Fibonacci» tienen una larga tradición en matemáticas, en biología y en la literatura. Son los que pertenecen a la sucesión de Fibonacci:

$f_0 = 0; f_1 = 1; f_n = f_{n-1} + f_{n-2}\ (n \geq 2)$

Es decir, cada término de la sucesión, a partir del tercero, se calcula sumando los dos anteriores. Para generarlos con un programa en BRM podemos dedicar tres registros, R0, R1 y R2 para que contengan, respectivamente, f_n, f_{n-1} y f_{n-2}. Inicialmente ponemos los valores 1 y 0 en R1 (f_{n-1}) y R2 (f_{n-2}), y entramos en un bucle en el que a cada paso calculamos el contenido de R0 como la suma de R1 y R2 y antes de volver al bucle sustituimos el contenido de R2 por el de R1 y el contenido de R1 por el de R0. De este modo, los contenidos de R0 a cada paso por el bucle serán los sucesivos números de Fibonacci.

Para averiguar si un determinado número es de Fibonacci basta generar la sucesión y a cada paso comparar el contenido de R0 (f_n) con el número: si el resultado es cero terminamos con la respuesta «sí»; si ese contenido es mayor que el número es que nos hemos pasado, y terminamos con la respuesta «no».

El programa 4.2 resuelve este problema. Como aún no hemos hablado de comunicaciones con periféricos, el número a comprobar (en este caso, 233) lo incluimos en el mismo programa. Y seguimos el convenio de que la respuesta se obtiene en R4: si es negativa, el contenido final de R4 será 0, y si el número sí es de Fibonacci, el contenido de R4 será 1.

```
                   /**************************************************
                   *                                                *
                   *             ¿Número de Fibonacci?              *
                   *                                                *
                   **************************************************/
                          .equ  Num, 233  @ Número a comprobar
00000000 E3A02000         mov   r2,#0     @  (R2) = f(n-2)
00000004 E3A01001         mov   r1,#1     @  (R1) = f(n-1)
00000008 E3A030E9         mov   r3,#Num
0000000C E3A04000         mov   r4,#0     @ saldrá con 1 si Num es Fib
00000010 E0810002  bucle: add   r0,r1,r2  @  fn = f(n-1)+f(n-2)
00000014 E1500003         cmp   r0,r3
00000018 0A000003         beq   si
0000001C CA000003         bgt   no
00000020 E1A02001         mov   r2,r1     @ f(n-2) = f(n-1)
00000024 E1A01000         mov   r1,r0     @ f(n-1) = f(n)
00000028 EAFFFFF8         b     bucle
0000002C E3A04001  si:    mov   r4,#1
00000030 EF000011  no:    swi   0x11
                          .end
```

Programa 4.2 Ejemplo de bucle.

Fíjese en cuatro detalles importantes:

- Aparte del símbolo «`Num`», que se iguala con el valor 233, en el programa se definen tres símbolos, que se llaman **etiquetas**. Para no confundirla con otro tipo de símbolo, la etiqueta, en su definición, debe terminar con «`:`», y toma el valor de la dirección de la instrucción a la que acompaña: «`bucle`» tiene el valor 0x10, «`si`» 0x2C, y «`no`» 0x30. De este modo, al escribir en ensamblador nos podemos despreocupar de valores numéricos de direcciones de memoria.

- Debería usted comprobar cómo ha traducido el ensamblador las instrucciones de bifurcación:
  - En la que está en la dirección 0x18 (bifurca a «`si`» si son iguales) ha puesto «3» en el campo «Dist» (figura 3.10). Recuerde que la dirección efectiva se calcula (en tiempo de ejecución) como la suma de la dirección de la instrucción más 8 más 4×Dist. En este caso, como 0x18 = 24, resulta $24 + 8 + 4 \times 3 = 34$ = 0x2C, que es el valor del símbolo «`si`».
  - En la que está en 0x1C (bifurca a «`no`» si mayor) ha puesto la misma distancia, 3. En efecto, la «distancia» a «`no`» es igual que la anterior.
  - En la que está en 0x28 (40 decimal) (bifurca a «`bucle`») ha puesto Dist = FFFFF8, que, al interpretarse como número negativo en complemento a dos, representa «-8». DE = $40 + 8 + 4 \times (-8) = 16$ = 0x10, dirección de «`bucle`».

- Las direcciones efectivas se calculan en tiempo de ejecución, es decir, con el programa ya cargado en memoria. Venimos suponiendo que la primera dirección del programa es la 0 (el ensamblador no puede saber en dónde se cargará finalmente). Pero si se carga, por ejemplo, a partir de la dirección 0x1000 (como hace ARMSim#) todo funciona igual: `b bucle` (en binario, por supuesto) no estará en 0x28, sino en 0x1028, y en la ejecución bifurcará a 0x1020, donde estará `add r0,r1,r2`.

- Una desilusión provisional: la utilidad de este programa es muy limitada, debido a la forma de introducir el número a comprobar en un registro: `mov r3, #Num`. En efecto, ya vimos en el apartado 3.4 que el operando inmediato tiene que ser o bien menor que 256 o bien poderse obtener mediante rotación de un máximo de ocho bits.

Vamos a ver cómo se puede operar con una constante cualquiera de 32 bits. Pero antes debería usted comprobar el funcionamiento del programa escribiendo el código fuente en un fichero de texto y cargándolo en el simulador. Verá que inicialmente (R4) = 0 y que la ejecución termina con (R4) = 1, porque 233 es un número de Fibonacci. Pruebe con otros números (editando cada vez el fichero de texto y recargándolo): cuando es mayor que 255 y no se puede generar mediante rotación ARMSim# muestra el error.

4.3. El *pool* de literales

Para aquellas constantes cuyos valores no pueden obtenerse con el esquema de la rotación de ocho bits no hay más remedio que tener la constante en una palabra de la memoria y acceder a ella con una instrucción `LDR` (apartado 3.5).

Aún no hemos visto cómo poner constantes en el código fuente, pero no importa, porque el servicial ensamblador hace el trabajo por nosotros, y además nos libera de la aburrida tarea de calcular la distancia necesaria. Para ello, existe una **pseudoinstrucción** que se llama «`LDR`». Tiene el mismo nombre que la instrucción, pero se usa de otro modo: para introducir en un registro una constante cualquiera, por ejemplo, para poner 2.011 en R0, escribimos «`ldr r0, =2011`» y el ensamblador se ocupará de ver si la puede generar mediante rotación. Si puede, la traduce por una instrucción `mov`. Y si no, la traduce por una instrucción `LDR` que carga en R0, con direccionamiento relativo, la constante 2.011 que él mismo introduce al final del código objeto.

Compruebe cómo funciona escribiendo un programa que contenga estas instrucciones (u otras similares):

```
ldr r0, =2011       ldr r1, =-2011      ldr r2, =0xfff
ldr r3, =0xffff     ldr r4, =4096       ldr r5, =0xffffff
ldr r6, =2011       ldr r7, =-2011      ldr r7, =0xfff
ldr r8, =0xffff
```

y cargándolo en el simulador. Obtendrá este resultado:

```
00001000 E59F0024        ldr r0, =2011
00001004 E59F1024        ldr r1, =-2011
00001008 E59F2024        ldr r2, =0xfff
0000100C E59F3024        ldr r3, =0xffff
00001010 E3A04A01        ldr r4, =4096
00001014 E3E054FF        ldr r5, =0xffffff
00001018 E59F600C        ldr r6, =2011
0000101C E59F700C        ldr r7, =-2011
00001020 E59F700C        ldr r7, =0xfff
00001024 E59F800C        ldr r8, =0xffff
00001028 EF000011        swi   0x11
```

Veamos lo que ha hecho al ensamblar:

- La primera instrucción la ha traducido como `0xE59F0024`. Mirando los formatos, esto corresponde a `ldr r0, [pc, #0x24]` (figura 3.9). Cuando el procesador la ejecute PC tendrá el valor 0x1000 + 8, y la dirección efectiva será 0x1008 + 0x24 = 0x102C. Es decir, carga en R0 el contenido de la dirección 0x102C. ¿Y qué hay en esa dirección? La ventana central del simulador solo muestra el código, pero los contenidos de la memoria se pueden ver como se indica en el apartado A.2, y puede usted comprobar que el contenido de esa dirección es 0x7DB = 2011. *¡El ensamblador ha creado la constante en la palabra de dirección inmediatamente siguiente al código, y ha sintetizado la instrucción `LDR` con direccionamiento relativo y la distancia adecuada para acceder a la constante!*
- Para las tres instrucciones siguientes ha hecho lo mismo, creando las constantes 0xFFFF825 (-2011 en complemento a 2), 0xFFF y 0xFFFF en las direcciones siguientes (0x1030, 0x1034 y 0x1038), y generando las instrucciones `LDR` oportunas.
- Para 4.096, sin embargo, ha hecho otra cosa, porque la constante la puede generar con «1» en el campo «inmed_8» y «10» en el campo «rot» (figura 3.6). Ha traducido igual que si hubiésemos escrito `mov r4,#4096`.
- La siguiente es interesante: `E3E054FF`, si la descodificamos de acuerdo con su formato, que es el de la figura 3.5, resulta que la ha traducido como una instrucción `MVN` con «0xFF» en el campo «inmed_8» y «4» en el campo «rot»: un operando inmediato que resulta de rotar 2×4 veces a la derecha (o $32 - 8 = 24$ veces a la izquierda) 0xFF, lo que da 0xFF000000. Es decir, la ha traducido igual que si hubiésemos escrito `mvn r5,#0xFF000000`. La instrucción `MVN` hace la operación NOT (tabla 3.2), y lo que se carga en R5 es NOT(0xFF000000) = 0x00FFFFFF, como queríamos. Tampoco ha tenido aquí necesidad el ensamblador de generar una constante.
- Las cuatro últimas `LDR` son iguales a las cuatro primeras y si mira las distancias comprobará que el ensamblador no genera más constantes, sino accesos a las que ya tiene.

En resumen, utilizando la *pseudoinstrucción* `LDR` podemos poner cualquier constante (entre -2^{-31} y $2^{31} - 1$) y el ensamblador se ocupa, si puede, de generar una `MOV` (o una `MVN`), y si no, de ver si ya tiene la constante creada, de crearla si es necesario y de generar la *instrucción* `LDR` adecuada. Al final, tras el código objeto, inserta el «pool de literales», que contiene todas las constantes generadas.

Moraleja: en este ensamblador, en general, para cargar una constante en un registro, lo mejor es utilizar la *pseudoinstrucción* `LDR`.

Si modifica usted el programa 4.2 cambiando la instrucción `mov r3,#Num` por `ldr r3,=Num` verá que el ensamblador ya no genera errores para ningún número.

4.4. Más directivas

Informaciones para el montador

Si ha seguido usted los consejos no debe haber tenido problemas para ejecutar en el simulador los ejemplos anteriores. Pero para poder ejecutarlos en un procesador real los programas fuente tienen que procesarse con un ensamblador, que, como ya sabemos, genera un código objeto binario. Es binario, pero *no es ejecutable*. Esto es así porque los sencillos ejemplos anteriores son «autosuficientes», pero en general un programa de aplicación está compuesto por módulos que se necesitan unos a otros pero se ensamblan independientemente. Cada módulo puede *importar* símbolos de otros módulos y *exportar* otros símbolos definidos en él.

La generación final de un código ejecutable la realiza otro procesador software llamado **montador** *(linker)*, que combina las informaciones procedentes del ensamblaje de los módulos y produce un fichero binario que ya puede cargarse en la memoria. Una de las informaciones que necesita es una etiqueta que debe estar en uno de los módulos (aunque solo haya uno) que identifica a la primera instrucción a ejecutar. Esta etiqueta se llama «`_start`», y el módulo en que aparece tiene que exportarla, lo que se hace con la directiva «`.global _start`»

En el apartado 4.7 volveremos sobre los módulos, y en el 5.2 sobre el montador. La explicación anterior era necesaria para justificar que los ejemplos siguientes comienzan con esa directiva y tienen la etiqueta `_start` en la primera instrucción a ejecutar.

Hasta ahora nuestros ejemplos contienen solamente instrucciones. Con frecuencia necesitamos incluir también datos. Enseguida veremos cómo se hace, pero otra información que necesita el montador es dónde comienzan las instrucciones y dónde los datos. Para ello, antes de las instrucciones debemos escribir la directiva `.text`, que identifica a la *sección de código* y antes de los datos, la directiva `.data`, que identifica a la *sección de datos*.

Introducción de datos en la memoria

Las directivas que sirven para introducir valores en la sección de datos son: `.word`, `.byte`, `.ascii`, `.asciz` y `.align`.

`.word` y `.byte` permiten definir un valor en una palabra o en un byte, o varios valores en varias palabras o varios bytes (separándolos con comas). Como ejemplo de uso, el programa 4.3 es autoexplicativo, pero conviene fijarse en tres detalles:

- El listado se ha obtenido con el ensamblador GNU, que pone tanto la sección de código (`.text`) como la de datos a partir de la dirección 0. Será luego el montador el que las coloque una tras otra. Sin embargo, el simulador ARMSim# hace también el montaje y carga el resultado en la memoria a partir de la dirección 0x1000, de modo que el contenido de «palabra1» queda en la dirección 0x1024. Ahora bien, si, como es de esperar, está usted siguiendo con interés y atención esta explicación, debe estar pensando: «aquí tiene que haber un error». Porque como la última instrucción ocupa las

```
                  /***********************************************
                  *     Este programa intercambia los contenidos    *
                  *     de dos palabras de la memoria               *
                  ***********************************************/
                     .text
                     .global     _start
00000000 E59F0014  _start: ldr   r0,=palabra1
00000004 E59F1014          ldr   r1,=palabra2
00000008 E5902000          ldr   r2,[r0]
0000000C E5913000          ldr   r3,[r1]
00000010 E5812000          str   r2,[r1]
00000014 E5803000          str   r3,[r0]
00000018 EF000011          swi   0x011
                     .data
00000000 AAAAAAAA  palabra1: .word 0xAAAAAAAA
00000004 BBBBBBBB  palabra2: .word 0xBBBBBBBB
                     .end
```

Programa 4.3 Intercambio de palabras.

direcciones 0x1018 a 0x101B, ese contenido debería estar a partir de la dirección 0x101C. Veamos por qué no es así, y no hay tal error.

- La pseudoinstrucción `ldr r0,=palabra1` no carga en R0 el contenido de la palabra, sino *el valor de la etiqueta* «`palabra1`», que es su dirección. En los listados no se aprecia, pero mire usted en el simulador los contenidos de la memoria a partir de la instrucción `SWI`:

Dirección	**Contenido**
00001018	EF000011
0000101C	00001024
00001020	00001028
00001024	AAAAAAAA
00001028	BBBBBBBB

  Lo que ha hecho el ensamblador en este caso al traducir las pseudoinstrucciones `LDR` es crear un «pool de literales» con dos constantes, que después del montaje han quedado en las direcciones 0x101C y 0x1020, con las direcciones de las palabras que contienen los datos, 0x1024 y 0x1028.

- En tiempo de ejecución, R0 se carga con la dirección del primer dato, y R1 con la del segundo. Por eso, para el intercambio (instrucciones `LDR` y `STR`) se usan R0 y R1 como punteros.

Otro ejemplo: el programa 4.4 es la implementación de la suma de las componentes de un vector (apartado 2.7). En el simulador puede verse que con los datos puestos el resultado de la suma, 210, queda en R2.

Dos comentarios:

- Al cargar en la memoria un número de bytes que no es múltiplo de cuatro, la dirección siguiente al último no está alineada. En este ejemplo no importa, pero si, por ejemplo, después de los diez primeros bytes ponemos una `.word` la palabra no estaría alineada y la ejecución fallaría al intentar acceder a ella. Para evitarlo, antes de `.word` se pone otra directiva: `.align`. El ensamblador rellena con ceros los bytes hasta llegar a la primera dirección múltiplo de cuatro.

```
                    /*************************************************
                    *  Suma de las componentes de un vector en la MP  *
                    *  (suma bytes, pero solo si son positivos)       *
                    *                                                 *
                    *************************************************/
                          .text
                          .global  _start
                          .equ  N, 20     @ El vector tiene 20 elementos
                                          @ (bytes)
                      _start:
00000000 E59F0018          ldr  r0,=vector  @ R0 = puntero al vector
00000004 E3A01014          mov  r1,#N       @ R1 = contador
00000008 E3A02000          mov  r2,#0       @ R2 = suma
                      bucle:
0000000C E4D03001          ldrb r3,[r0],#1  @ Carga un elemento en R3
                                            @ y actualiza R0
                                            @ (cada elemento ocupa 1 byte)
00000010 E0822003          add   r2,r2,r3
00000014 E2511001          subs  r1,r1,#1
00000018 1AFFFFFB          bne   bucle
0000001C EF000011          swi   0x11
                          .data
                      vector:
00000000 01020304         .byte 1,2,3,4,5,6,7,8,9,10
         05060708
         090A
0000000A 0B0C0D0E         .byte 11,12,13,14,15,16,17,18,19,20
         0F101112
         1314
                          .end
```

Programa 4.4 Suma de las componentes de un vector.

- Otra desilusión provisional: pruebe usted a ejecutar el programa poniendo algún número negativo. El resultado es incorrecto. En el apartado 4.6 veremos una solución general con un subprograma, pero de momento le invitamos a pensar en arreglarlo con solo dos instrucciones adicionales que hacen desplazamientos.

Las directivas `.ascii` y `.asciz`, seguidas de una cadena de caracteres entre comillas, introducen la sucesión de códigos ASCII de la cadena. La diferencia es que `.asciz` añade al final un byte con el contenido 0x00 («NUL»), que es el convenio habitual para señalar el fin de una cadena.

El programa 4.5 define una cadena de caracteres y reserva, con la directiva `.skip`, 80 bytes en la memoria. En su ejecución, hace una copia de la cadena original en los bytes reservados.

En la ventana principal del simulador no se muestran los contenidos de la sección de datos. Es interesante que los mire con la vista de memoria por bytes a partir de la dirección 0x1018 (siga las instrucciones del apartado A.2). Verá que después de la traducción de la instrucción `SWI` el ensamblador ha puesto un «pool de literales» de dos palabras que contienen los punteros, a continuación 80 bytes con contenido inicial 0x00 (si se ejecuta el programa paso a paso se puede ver cómo se van rellenando esos bytes), y finalmente los códigos ASCII de la cadena original.

```
                   /**************************************************
                   *                                                 *
                   *   Copia una cadena de caracteres (máximo: 80)   *
                   *                                                 *
                   **************************************************/
                           .text
                           .equ     NUL, 0     @  código ASCII de NUL
                           .global _start
                           _start:
00000000 E59F0014             ldr   r0,=original @  R0 y R1: punteros a
00000004 E59F1014             ldr   r1,=copia    @   las cadenas
                       bucle:
00000008 E4D02001             ldrb  r2,[r0],#1
0000000C E4C12001             strb  r2,[r1],#1
00000010 E3520000             cmp   r2,#NUL
00000014 1AFFFFFB             bne   bucle
00000018 EF000011             swi   0x11
                            .data
                       copia:
00000000 00000000             .skip   80  @ reserva 80 bytes
         00000000
         00000000
         00000000
         00000000
                       original:
00000050 45737461             .asciz  "Esta es una cadena cualquiera"
         20657320
         756E6120
         63616465
         6E612063
                                  .end
```

Programa 4.5 Copia una cadena de caracteres

4.5. Menos bifurcaciones

Una instrucción de bifurcación, cuando la condición se cumple, retrasa la ejecución en dos ciclos de reloj (apartado 3.2). Cualquier otra instrucción, si la condición *no* se cumple avanza en la cadena aunque no se ejecute, pero no la para, y retrasa un ciclo de reloj. Es interesante, por tanto, prescindir de bifurcaciones cuando sea posible, especialmente si están en un bucle que se ejecuta millones de veces.

Un ejemplo sencillo: poner en R1 el valor absoluto del contenido de R0. Con una bifurcación y la instrucción RSB (tabla 3.2) puede hacerse así:

```
        mov  r1,r0
        cmp  r1,#0
        bge  sigue     @ bifurca si (R1)>=0
        rsb  r1,r1,#0  @ Si (R1)< 0, 0-(R1)→ R1
sigue: ...
```

Pero condicionando la RSB podemos prescindir de la bifurcación:

```
        mov    r1,r0
        cmp    r1,#0
        rsblt  r1,r1,#0  @ Si (R1) < 0, 0-(R1) → R1
sigue: ...
```

Un ejemplo un poco más elaborado es el del algoritmo de Euclides para calcular el máximo común divisor de dos enteros. En pseudocódigo, llamando x e y a los números, una de sus versiones es:

```
mientras que x sea distinto de y {
   si x > y, x = x - y
   si no, y = y - x
}
```

Traduciendo a ensamblador para un procesador «normal», usaríamos instrucciones de bifurcación. Suponiendo x en R0 e y en R1:

```
MCD:   cmp  r0,r1      @ ¿x>y?
       beq  fin
       blt  XmasY      @ si x<y...
       sub  r0,r0,r1   @ si x>y, x-y→ x
       b    MCD
XmasY: sub  r1,r1,r0   @ si x<y, y-x→ y
       b    MCD
fin:...
```

En BRM también funciona ese programa, pero este otro es más eficiente:

```
MCD:   cmp    r0,r1      @ ¿x>y?
       subgt  r0,r0,r1   @ si x>y, x-y→ x
       sublt  r1,r1,r0   @ si x<y, y-x→ y
       bne    MCD
```

Observe que la condición para la bifurcación BNE depende de CMP, ya que las dos instrucciones intermedias no afectan a los indicadores.

4.6. Subprogramas

La instrucción BL (apartado 3.6), que, además de bifurcar, guarda la dirección de retorno en el registro de enlace (LR = R14) permite llamar a un subprograma (apartado 2.7), del que se vuelve con «MOV PC,LR». Así, el programa MCD puede convertirse en un subprograma sin más que añadirle esa instrucción al final, y utilizarlo como muestra el programa 4.6, que incluye dos llamadas para probarlo.

Con subprogramas se pueden implementar operaciones comunes que no están previstas en el repertorio de instrucciones. ¿Ha pensado usted en cómo resolver el problema de la suma de bytes cuando alguno de ellos es negativo (programa 4.4)? El problema se produce porque, por ejemplo, la representación de −10 en un byte con complemento a 2 es 0xF6. Y al cargarlo con la instrucción LDRB en un registro, el contenido de ese registro resulta ser 0x000000F6, que al interpretarlo como entero con signo, es +246. Entre las instrucciones de ARM de las que hemos prescindido en BRM hay una, LDRSB (por «signed byte»), que extiende hacia la izquierda el bit de signo del byte. Con ella se puede cargar en el registro la representación correcta en 32 bits: 0xFFFFFFF6.

Decíamos que el problema se puede arreglar con dos sencillas instrucciones. ¿De verdad que ha pensado en ello y no ha encontrado la respuesta? Pues es muy sencillo: desplazando 0x000000F6

```
                  /*************************************************
                  *                                               *
                  * Prueba del subprograma del algoritmo de Euclides *
                  *                                               *
                  *************************************************/
                     .text
                     .global     _start
                  _start:
00000000 E3A00009     ldr r0,=9
00000004 E3A010FF     ldr r1,=255
00000008 EB000005     bl  MCD     @ llamada a MCD
0000000C E1A08000     mov r8,r0   @ MCD(9,255) = 3 → R8
00000010 E3A00031     ldr r0,=49
00000014 E59F101C     ldr r1,=854
00000018 EB000001     bl  MCD     @ llamada a MCD
0000001C E1A09000     mov r9,r0   @ MCD(49,854) = 7 → R9
00000020 EF000011     swi 0x11
                  /* Subprograma MCD */
                   MCD:
00000024 E1500001       cmp    r0,r1
00000028 C0400001       subgt  r0,r0,r1
0000002C B0411000       sublt  r1,r1,r0
00000030 1AFFFFFB       bne    MCD
00000034 E1A0F00E       mov    pc,lr
00000038 00000356     .end
```

Programa 4.6 Subprograma MCD y llamadas.

venticuatro bits a la izquierda resulta 0xF6000000: el bit de signo de la representación en un byte queda colocado en el bit de signo de la palabra. Si luego se hace un desplazamiento *aritmético* de venticuatro bits a la derecha se obtiene 0xFFFFFFF6.

Podemos implementar una «pseudoinstrucción» `LDRSB` con este subprograma que hace esas dos operaciones tras cargar en R3 el byte apuntado por R0 y actualizar R0:

```
LDRSB:  ldrb  r3,[r0],#1
        mov   r3,r3,lsl #24
        mov   r3,r3,asr #24
        mov   pc,lr
```

Basta modificar el programa 4.4 sustituyendo la instrucción «`ldrb r3,[r0],#1`» por «`bl LDRSB`» e incluyendo estas instrucciones al final (antes de «`.data`») para ver que si ponemos cualquier número negativo como componente del vector el programa hace la suma correctamente.

Paso de valor y paso de referencia

En los dos ejemplos, `MCD` y `LDRSB`, el programa invocador le pasa **parámetros de entrada** al subprograma (programa invocado) y éste devuelve **parámetros de salida** al primero.

En `MCD`, los parámeros de entrada son los números de los que se quiere calcular su máximo común divisor, cuyos valores se pasan por los registros R0 y R1. Y el parámetro de salida es el resultado, cuyo valor se devuelve en R0 (y también en R1).

El parámetro de entrada a `LDRSB` es el byte original, y el de salida, el byte con el signo extendido. El valor de este último se devuelve en R3, pero observe que el parámetro de entrada no es *el valor* del byte, sino *su dirección*. En efecto, R0 no contiene el valor del byte, sino la dirección en la que se encuentra. Se dice que es un **paso de referencia**.

El paso de referencia es a veces imprescindible. Por ejemplo, cuando el parámetro a pasar es una cadena de caracteres: en un registro solo podemos pasar cuatro bytes; una cadena de longitud 52 ya nos ocuparía los trece registros disponibles.

Un sencillo ejercicio que puede usted hacer y probar en el simulador es escribir un subprograma para copiar cadenas de una zona a otra de la memoria (programa 4.5) y un programa que lo llame varias veces pasándole cada vez como parámetros de entrada la dirección de la cadena y la dirección de una zona reservada para la copia.

Paso de parámetros por la pila

Naturalmente, el programa invocador y el invocado tienen que seguir algún convenio sobre los registros que utilizan. Una alternativa más flexible y que no depende de los registros es que el invocador los ponga en la pila con instrucciones `PUSH` y el invocado los recoja utilizando el puntero de pila (apartado 2.7). Es menos eficiente que el paso por registros porque requiere accesos a memoria, pero, por una parte, no hay limitación por el número de registros, y, por otra, es más fácil construir subprogramas que se adapten a distintas arquitecturas. Por eso, los compiladores suelen hacerlo así para implementar las funciones de los lenguajes de alto nivel.

En BRM no hay instrucciones `PUSH` ni `POP`, pero estas operaciones se realizan fácilmente con `STR` y direccionamiento inmediato postindexado sobre el puntero de pila (registro R13 = SP) y con `LDR` y direccionamiento inmediato preindexado respectivamente (apartado 3.5). Si seguimos el convenio de que SP apunte a la primera dirección libre sobre la cima de la pila y que ésta crezca en direcciones decrecientes de la memoria (o sea, que cuando se introduce un elemento se decrementa SP), introducir R0 y extraer R3, por ejemplo, se hacen así:

Operación	Instrucción
PUSH R0	`str r0,[sp],#-4`
POP R3	`ldr r3,[sp,#4]!`

El puntero de pila, SP, siempre se debe decrementar (en `PUSH`) o incrementar (en `POP`) cuatro unidades, porque los elementos de la pila son palabras de 32 bits.

Y acceder a un elemento es igual de fácil, utilizando preindexado pero sin actualización del registro de base (SP). Así, para copiar en R0 el elemento que está en la cabeza (el último introducido) y en R1 el que está debajo de él:

Operación	Instrucción
(MP[(sp)+4]) $\rightarrow$ R0	`ldr r0,[sp,#4]`
(MP[(sp)+8]) $\rightarrow$ R1	`ldr r1,[sp,#8]`

La pila, además, la debe usar el subprograma para guardar los registros que utiliza y luego recuperarlos, a fin de que al codificar el programa invocador no sea necesario preocuparse de si alguno de los registros va a resultar alterado (salvo que el convenio para los parámetros de salida sea dejarlos en registros).

El programa 4.7 es otra versión del subprograma MCD y de las llamadas en la que los parámetros y el resultado se pasan por la pila. Observe que ahora el subprograma puede utilizar cualquier pareja de registros, pero los salva y luego los recupera por si el programa invocador los está usando para otra cosa (aquí no es el caso, pero así el subprograma es válido para cualquier invocador, que solo tiene que tener en cuenta que debe introducir en la pila los parámetros y recuperar el resultado de la misma).

```
                  /**************************************************
                  * Versión del subprograma del algoritmo de Euclides *
                  * con paso de parámetros por la pila                *
                  *                                                   *
                  ***************************************************/
                     .text
                     .global     _start
                  /* Programa que llama al subprograma MCD */
                  _start:
00000000 E3A00009      ldr r0,=9
00000004 E3A010FF      ldr r1,=255
00000008 E40D0004      str r0,[sp],#-4 @ pone parámetros en la pila
0000000C E40D1004      str r1,[sp],#-4 @  (PUSH)
00000010 EB000009      bl  MCD         @ llamada a MCD
00000014 E5BD8004      ldr  r8,[sp,#4]!@ MCD(9,255) = 3 → R8 (POP R8)
00000018 E28DD004      add sp,sp,#4    @ para dejar la pila como estaba
0000001C E3A00031      ldr r0,=49
00000020 E59F1044      ldr r1,=854
00000024 E40D0004      str r0,[sp],#-4 @ pone parámetros en la pila
00000028 E40D1004      str r1,[sp],#-4 @  (PUSH)
0000002C EB000002      bl  MCD         @ llamada a MCD
00000030 E5BD9004      ldr  r9,[sp,#4]!@ MCD(49,854) = 7 → R9 (POP R9)
00000034 E28DD004      add sp,sp,#4    @ para dejar la pila como estaba
00000038 EF000011      swi 0x11
                  /* Subprograma MCD */
0000003C E40D0004  MCD:  str  r3,[sp],#-4 @ PUSH R3 y R4 (salva
00000040 E40D1004        str  r4,[sp],#-4 @  los registros que utiliza)
00000044 E59D000C        ldr  r3,[sp,#12] @ coje el segundo parámetro
00000048 E59D1010        ldr  r4,[sp,#16] @ coje el primer parámetro
                  bucle:
0000004C E1500001        cmp    r3,r4
00000050 C0400001        subgt  r3,r3,r4
00000054 B0411000        sublt  r4,r4,r3
00000058 1AFFFFFB        bne    bucle
0000005C E58D100C        str    r4,[sp,#12] @ sustituye el segundo
                             @ parámetro de entrada por el resultado
00000060 E5BD1004        ldr  r4,[sp,#4]!   @ POP R4 y R3
00000064 E5BD0004        ldr  r3,[sp,#4]!   @ (recupera los registros)
00000068 E1A0F00E        mov    pc,lr
0000006C 00000356     .end
```

Programa 4.7 Subprograma MCD y llamadas con paso por pila

Anidamiento

Ya decíamos en el apartado 2.7 que si solo se usa la instrucción `BL` para salvar la dirección de retorno en el registro LR, el subprograma no puede llamar a otro subprograma, porque se sobreescribiría el contenido de LR y no podría volver al programa que le ha llamado a él. Comentábamos allí que algunos procesadores tienen instrucciones `CALL` (para llamar a un subprograma) y `RET` (para retornar de él). La primera hace un *push* del contador de programa, y la segunda, un *pop*. De este modo, en las llamadas sucesivas de un subprograma a otro los distintos valores del contador de programa (direcciones de retorno) se van apilando con las `CALL` y «desempilando» con las `RET`.

BRM no tiene estas instrucciones, pero, como antes, se puede conseguir la misma funcionalidad con `STR` y `LDR`. Si un subprograma llama a otro, al empezar tendrá una instrucción «`str lr,[sp],#-4`» para salvar en la pila el contenido del registro LR, y, al terminar, «`ldr lr,[sp],#4!`» para recuperarlo inmediatamente antes de «`mov pc,lr`».

Este ejemplo va a ser un poco más largo que los anteriores, pero es conveniente, para que le queden bien claras las ideas sobre la pila, que lo analice usted y que siga detenidamente su ejecución en el simulador. Se trata de un subprograma que calcula el máximo común divisor de tres enteros basándose en que $\text{MCD}(x, y, z) = \text{MCD}(x, \text{MCD}(y, z))$.

Como no cabe en una sola página, se ha partido en dos (programa 4.8 y programa 4.9) el listado que genera el ensamblador GNU para el código fuente. Este código fuente contiene un programa principal que llama al subprograma MCD3 que a su vez llama dos veces a MCD.

```
                      .text
                      .global  _start
                   _start:
00000000 E3A0DA06      ldr sp,=0x6000   @ inicializa el puntero de pila
00000004 E3A0000A      ldr r0,=10
00000008 E3A010FF      ldr r1,=255
0000000C E3A02E19      ldr r2,=400
00000010 E40D0004      str r0,[sp],#-4  @ mete los tres parámetros
00000014 E40D1004      str r1,[sp],#-4  @ en la pila (PUSH)
00000018 E40D2004      str r2,[sp],#-4
0000001C EB00000B      bl  MCD3         @ llama a MCD3
00000020 E5BD8004      ldr  r8,[sp,#4]! @ POP R8: recoje el resultado
                                        @  MCD(10,255,400) = 5 → R8
00000024 E28DD008      add sp,sp,#8     @ para dejar la pila como estaba
00000028 E3A00009      ldr r0,=9
0000002C E3A01051      ldr r1,=81
00000030 E3A0209C      ldr r2,=156
00000034 E40D0004      str r0,[sp],#-4  @ mete otros tres parámetros
00000038 E40D1004      str r1,[sp],#-4  @ en la pila (PUSH)
0000003C E40D2004      str r2,[sp],#-4
00000040 EB000002      bl  MCD3         @ llama a MCD3
00000044 E5BD9004      ldr  r9,[sp,#4]! @ POP R9: recoje el resultado
                                        @  MCD(9,81,156) = 3 → R9
00000048 E28DD008      add sp,sp,#8     @ para dejar la pila como estaba
0000004C EF000011      swi 0x11
```

Programa 4.8 Programa principal para probar MCD3

```
                  /* Subprograma MCD3 */
00000050 E40DE004  MCD3: str  lr,[sp],#-4 @ PUSH LR
00000054 E40D1004        str  r1,[sp],#-4 @ PUSH R1, R2 y R3
00000058 E40D2004        str  r2,[sp],#-4 @ (salva los registros
0000005C E40D3004        str  r3,[sp],#-4 @   que utiliza)
00000060 E59D3014        ldr  r3,[sp,#20] @ coje el tercer parámetro
00000064 E59D2018        ldr  r2,[sp,#24] @ coje el segundo parámetro
00000068 E59D101C        ldr  r1,[sp,#28] @ coje el primer parámetro
0000006C E40D2004        str  r2,[sp],#-4 @ pone R2 y R3 en la pila
00000070 E40D3004        str  r3,[sp],#-4
00000074 EB00000C        bl   MCD
00000078 E5BD2004        ldr  r2,[sp,#4]!  @ POP R2: MCD(R2,R3) → R2
0000007C E28DD004        add  sp,sp,#4     @ pone la pila como estaba
00000080 E40D1004        str  r1,[sp],#-4  @ pone R1 y R2 en la pila
00000084 E40D2004        str  r2,[sp],#-4
00000088 EB000007        bl   MCD
0000008C E5BD1004        ldr  r1,[sp,#4]!   @ POP R1: MCD(R1,R2) → R1
00000090 E28DD004        add sp,sp,#4       @ pone la pila como estaba
00000094 E58D1014        str    r1,[sp,#20] @ sustituye el tercer
                                @ parámetro de entrada por el resultado
00000098 E5BD3004        ldr  r3,[sp,#4]!   @ POP R3, R2 y R1
0000009C E5BD2004        ldr  r2,[sp,#4]!   @ (recupera los registros
000000A0 E5BD1004        ldr  r1,[sp,#4]!   @   salvados)
000000A4 E5BDE004        ldr  lr,[sp,#4]!   @ POP LR
000000A8 E1A0F00E        mov    pc,lr       @ vuelve con el resultado en
                                @ MP[(SP)+4] y los parámetros segundo
                                @ y primero en MP[(SP)+8] y MP[(SP)+12]
                  /* Subprograma MCD */
                  /* Como no llama a nadie, no es necesario salvar LR */
000000AC E40D3004  MCD:  str  r3,[sp],#-4 @ PUSH R3 y R4
000000B0 E40D4004        str  r4,[sp],#-4 @ (salva los registros)
000000B4 E59D300C        ldr  r3,[sp,#12] @ coje el segundo parámetro
000000B8 E59D4010        ldr  r4,[sp,#16] @ coje el primer parámetro
                   bucle:
000000BC E1530004        cmp    r3,r4
000000C0 C0433004        subgt  r3,r3,r4
000000C4 B0444003        sublt  r4,r4,r3
000000C8 1AFFFFFB        bne    bucle
000000CC E58D400C        str    r4,[sp,#12] @ sustituye el segundo
                                @ parámetro de entrada por el resultado
000000D0 E5BD4004        ldr  r4,[sp,#4]!   @ POP R4 y R3
000000D4 E5BD3004        ldr  r3,[sp,#4]!   @ (recupera los registros)
000000D8 E1A0F00E        mov    pc,lr       @ vuelve con el resultado
                                            @ en MP[(SP)+4] y el primer
                                            @ parámetro en MP[(SP)+8]
                        .end
```

Programa 4.9 Subprogramas MCD3 y MCD

Es muy instructivo seguir paso a paso la ejecución, viendo cómo evoluciona la pila (en la ventana derecha del simulador, activándola, si es necesario, con «View > Stack»). Por ejemplo, tras ejecutar la primera vez la instrucción que en el listado aparece en la dirección 0x00B0 (y que en el simulador estará cargada en 0x10B0), es decir, tras la primera llamada del programa a MCD3 y primera llamada de éste a MCD, verá estos contenidos:

Dirección	**Contenido**	**Explicación**
00005FD8	00000190	R4 salvado por MCD
00005FDC	00000190	R3 salvado por MCD
00005FE0	00000190	R2 con el contenido del tercer parámetro, previamente cargado con la instrucción 0x0060 (en el simulador, 1060)
00005FE4	000000FF	R2 con el contenido del segundo parámetro, previamente cargado con la instrucción 0x0064 (en el simulador, 1064)
00005FE8	00000000	R3 salvado por MCD3
00005FEC	00000190	R2 salvado por MCD3
00005FF0	000000FF	R1 salvado por MCD3
00005FF4	00001020	Dirección de retorno al programa (en el código generado por el ensamblador, 00000020) introducida por MCD3 al hacer PUSH LR
00005FF8	00000190	Tercer parámetro (400) introducido por el programa
00005FFC	000000FF	Segundo parámetro (255) introducido por el programa
00006000	0000000A	Primer parámetro (10) introducido por el programa

Conforme avanza la ejecución el puntero de pila, SP, va «subiendo» o «bajando» a medida que se introducen o se extraen elementos de la pila. Cuando se introduce algo *(push)* se sobrescribe lo que pudiese haber en la dirección apuntada por SP, pero si un elemento se extrae *(pop)* no se borra. De manera que a la finalización del programa SP queda como estaba inicialmente, apuntando a 0x6000, y la zona que había sido utilizada para la pila conserva los últimos contenidos.

Ahora bien, a estas alturas, y si tiene usted el deseable espíritu crítico, se estará preguntando si merece la pena complicar el código de esa manera solamente para pasar los parámetros por la pila. Además, para que al final resulte un ejecutable menos eficiente. Y tiene usted toda la razón. Se trataba solamente de un ejercicio académico, para comprender bien el mecanismo de la pila. En la práctica, si los parámetros no son muchos, se pasan por registros, acordando un convenio. De hecho, hay uno llamado APCS (ARM Procedure Call Standard):

- Si hay menos de cinco parámetros, el primero se pasa por R0, el segundo por R1, el tercero por R2 y el cuarto por R3.
- Si hay más, a partir del quinto se pasan por la pila.
- El parámetro de salida se pasa por R0. Generalmente, los subprogramas implementan funciones, que, o no devuelven nada (valor de retorno «*void*»), o devuelven un solo valor.
- El subprograma puede alterar los contenidos de R0 a R3 y de R12 (que se usa como «borrador»), pero debe reponer los de todos los otros registros.

4.7. Módulos

Si ya en el último ejemplo, con un problema bastante sencillo, resulta un código fuente que empieza a resultar incómodo por su longitud, imagínese una aplicación real, con miles y miles de líneas de código.

Cualquier programa que no sea trivial se descompone en módulos, no solo por evitar un código fuente excesivamente largo, también porque los módulos pueden ser reutilizables por otros programas. Cada módulo se ensambla (o, en general, se traduce) independientemente de los otros, y como resultado se obtiene un código binario y unas tablas de símbolos para enlazar posteriormente, con un montador, todos los módulos.

Para que el ensamblador entienda que ciertos símbolos están definidos en otros módulos se utiliza una directiva: «`extern`». Y los símbolos del módulo que pueden ser necesarios para otros se exportan con la directiva «`global`». En el simulador ARMSim# se pueden cargar varios módulos: seleccionando «File > Open Multiple» se abre un diálogo para irlos añadiendo. Si no hay errores (como símbolos declarados «`extern`» que no aparecen como «`global`» en otro módulo), el simulador se encarga de hacer el montaje y cargar el ejecutable en la memoria simulada a partir de la dirección 0x1000.

Se puede comprobar adaptando el último ejemplo. Solo hay que escribir, en diferentes ficheros, los tres módulos «`pruebaMCD3`» (programa 4.8), «`MCD3`» y «`MCD`», añadiendo las oportunas `extern` y `global`. Le invitamos a hacerlo como ejercicio.

Veamos mejor otro ejemplo que exije un cierto esfuerzo de análisis por su parte. Se trata de implementar un algoritmo recursivo para el cálculo del factorial de un número natural, N. Hay varias maneras de calcular el factorial. Una de ellas se basa en la siguiente definición de la función «factorial»:

> *Si N es igual a 1, el factorial de N es 1;*
> *si no, el factorial de N es N multiplicado por el factorial de* $N-1$

Si no alcanza usted a apreciar la belleza de esta definición y a sentir un cierto prurito por investigar cómo puede materializarse en un programa que calcule el factorial de un número, debería preguntarse si su vocación es realmente la ingeniería.

Se trata de una definición **recursiva**, lo que quiere decir que recurre a sí misma. La palabra no está en el diccionario de la R.A.E. La más próxima es «recurrente»: «4. adj. Mat. Dicho de un proceso: Que se repite.». Pero no es lo mismo, porque el proceso no se repite exactamente igual una y otra vez, lo que daría lugar a una definición *circular*. Gráficamente, la recursión no es un círculo, sino una espiral que, recorrida en sentido inverso, termina por *cerrarse* en un punto (en este caso, cuando $N = 1$).

El programa 4.10 contiene una llamada para calcular el factorial de 4 (un número pequeño para ver la evolución de la pila durante la ejecución). La función está implementada en el módulo que muestra el programa 4.11. Como necesita multiplicar y BRM no tiene instrucción para hacerlo[3], hemos añadido otro módulo (programa 4.12) con un algoritmo trivial para multiplicar[4].

La función se llama a sí misma con la instrucción que en el código generado por el ensamblador está en la dirección 0x0018, y en cada llamada va introduciendo y decrementando el parámetro de entrada y guardando la dirección de retorno (siempre la misma: 0x101C) y el parámetro decrementado. Si el parámetro de entrada era N, se hacen $N-1$ llamadas, y no se retorna hasta la última.

[3] ARM sí la tiene. Puede probar a sustituir la llamada a `mult` por la instrucción `mul r2,r1,r2` y funcionará igual en el simulador.

[4] En la práctica se utiliza un algoritmo un poco más complejo pero mucho más eficiente, que se basa en sumas y desplazamientos sobre la representación binaria: el algoritmo de Booth.

```
                /* Programa con una llamada para calcular el factorial */
                             .text
                             .global  _start
                             .extern  fact
                             .equ   N,4
00000000 E3A0DA05      _start: ldr   sp,=0x5000
00000004 E3A01004              ldr  r1,=N
00000008 E40D1004              str  r1,[sp],#-4 @ PUSH del número
0000000C EBFFFFFE              bl   fact        @ llamada a fact
                                                @  devuelve N! en R0
00000010 E28DD004              add  sp,sp,#4    @ para dejar la pila
00000014 EF000011              swi  0x11        @  como estaba
                                .end
```

Programa 4.10 Llamada a la función para calcular el factorial de 4

```
                /* Subprograma factorial recursivo */
                                .text
                                .global fact
                                .extern mult
00000000 E40DE004         fact:   str lr,[sp],#-4 @ PUSH LR
00000004 E59D1008                 ldr r1,[sp,#8]
00000008 E3510001                 cmp r1,#1
0000000C DA000007                 ble cierre
00000010 E2411001                 sub r1,r1,#1
00000014 E40D1004                 str r1,[sp],#-4 @ PUSH R1
00000018 EBFFFFFE                 bl fact
0000001C E28DD004                 add sp,sp,#4
00000020 E2811001                 add r1,r1,#1
00000024 EBFFFFFE                 bl mult         @ R0*R1 → R0
00000028 E5BDE004                 ldr lr,[sp,#4]! @ POP LR
0000002C E1A0F00E                 mov pc,lr
00000030 E3A00001         cierre: mov r0,#1
00000034 E5BDE004                 ldr lr,[sp,#4]! @ POP LR
00000038 E1A0F00E                 mov pc,lr
                             .end
```

Programa 4.11 Implementación de la función factorial recursiva

```
                 /* Algoritmo sencillo de multiplicación
                          para enteros positivos
                    Recibe x e y en R0 y R1 y devuelve prod en R0 */
                                 .text
                                .global mult
00000000 E3A0C000   mult:    mov     r12, #0       @ (R12) = prod
00000004 E3500000            cmp     r0, #0        @ Comprueba si x = 0
00000008 0A000004            beq     retorno
0000000C E3510000            cmp     r1, #0        @ Comprueba si y = 0
00000010 0A000002            beq     retorno

00000014 E08CC001   bucle:   add     r12, r12, r1  @ prod = prod + y
00000018 E2500001            subs    r0, r0, #1    @ x = x - 1
0000001C 1AFFFFFC            bne     bucle

00000020 E1A0000C   retorno: mov   r0, r12         @ resultado a R0
00000024 E1A0F00E            mov   pc, lr
                                .end
```

Programa 4.12 Subprograma para multiplicar

La figura 4.1 muestra los estados sucesivos de la pila para $N = 4$, suponiendo que el módulo con la función (programa 4.11) se ha cargado a partir de la dirección 0x1000 y que el montador ha puesto el módulo que contiene el programa con la llamada (programa 4.10) a continuación de éste, es decir, a partir de la dirección siguiente a 0x1038, 0x103C. La instrucción `bl fact` de esa llamada, a la que correspondía la dirección 0x000C a la salida del ensamblador quedará, pues, cargada en la dirección 0x103C + 0x000C = 0x1048. La dirección siguiente, 0x104C, es la de retorno, que, tras la llamada del programa, ha quedado guardada en la pila por encima del parámetro previamente introducido.

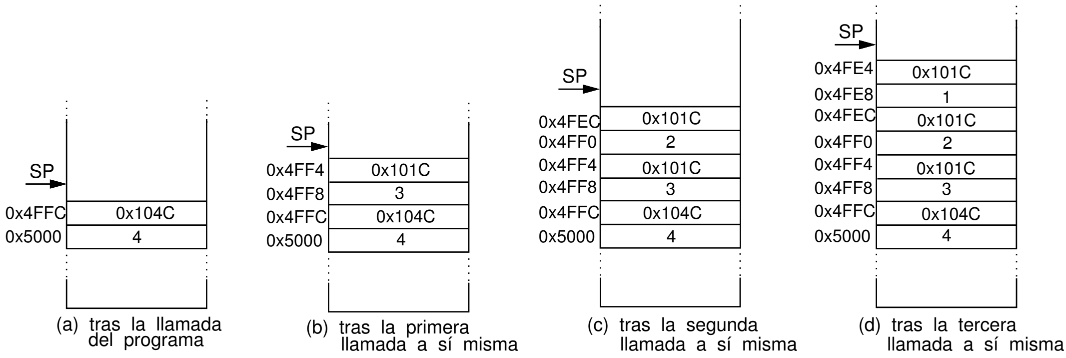


Figura 4.1 Estados de la pila en el cálculo del factorial.

Tras la tercera llamada, al comprobar que el último parámetro metido es igual a 1, tiene lugar el primer retorno pasando por `cierre` (es la única vez que se retorna por ahí), donde se inicializa R0 con 1 y se vuelve a la dirección 0x101C. A partir de ahí, en los sucesivos retornos, se van recuperando los valores de N (sumándole 1) y actualizando R0 para que al final quede con el valor de $N!$ Es en esta segunda fase (la de los retornos) en la que realmente se aplica la definición recursiva.

Conseguir que el simulador ARMSim# monte los módulos en el orden supuesto (primero la función factorial, luego el programa con la llamada y finalmente el de la multiplicación), y así poder comprobar los datos de la explicación anterior, es muy fácil: basta añadirlos en ese orden en la ventana de diálogo que se abre con «Load Multiple». Por cierto, se puede comprobar también que el registro PC se inicializa con el valor 0x103C, que es donde queda la primera instrucción a ejecutar (la que en el código fuente tiene la etiqueta «`_start`»).

4.8. Comunicaciones con los periféricos e interrupciones

Decíamos en el apartado 3.7 que para programar las comunicaciones entre el procesador y los puertos de los periféricos es necesario asignar direcciones a éstos mediante circuitos externos al procesador. Veamos un ejemplo.

Espera activa

Imaginemos dos periféricos de caracteres conectados a un procesador BRM a través de los buses A y D: un teclado simplificado y un *display* que presenta caracteres uno tras otro. Cada uno de ellos tiene un controlador con dos puertos (registros de un byte): un puerto de estado y un puerto de datos, que tienen asignadas estas direcciones y estas funciones:

Dirección	Puerto	Función
0x2000	estado del teclado	El bit menos significativo indica si está preparado para enviar un carácter (se ha pulsado una tecla)
0x2001	datos del teclado	Código de la última tecla pulsada
0x2002	estado del *display*	El bit menos significativo indica si está preparado para recibir un carácter (ha terminado de presentar el anterior)
0x2003	datos del *display*	Código del carácter a presentar

El «protocolo de la conversación» entre estos periféricos y el procesador es así:

Cuando se pulsa una tecla, el controlador pone a 1 el bit menos significativo del puerto 0x2000 (bit «preparado»). Si el programa necesita leer un carácter del puerto de datos tendrá que esperar a que este bit tenga el valor 1. Cuando se ejecuta la instrucción `LDRB` sobre el puerto 0x2001 el controlador lo pone a 0, y no lo vuelve a poner a 1 hasta que no se pulse de nuevo una tecla.

Cuando el procesador ejecuta una instrucción `STRB` sobre el puerto 0x2003 para presentar un carácter en el *display*, su controlador pone a 0 el bit preparado del puerto 0x2002, y no lo pone a 1 hasta que los circuitos no han terminado de escribirlo. Si el programa necesita enviar otro carácter tendrá que esperar a que se ponga a 1.

Si se ponen previamente las direcciones de los puertos en los registros R1 a R4, éste podría ser un subprograma para leer un carácter del teclado y devolverlo en R0, haciendo «eco» en el *display*:

```
esperatecl:
     ldrb r12,[r1]
     ands r12,r12,#1 @ si preparado = 0 pone Z = 1
     beq  esperatecl
     ldrb r0,[r2]    @ lee el carácter
esperadisp:
     ldrb r12,[r3]
     ands r12,r12,#1 @ si preparado = 0 pone Z = 1
     beq  esperadisp
     strb r0,[r4]    @ escribe el carácter
     mov  pc,lr
```

Y un programa para leer indefinidamente lo que se escribe y presentarlo sería:

```
      ldr  r1,=0x2000
      ldr  r2,=0x2001
      ldr  r3,=0x2002
      ldr  r4,=0x2003
sgte: bl esperatecl
      b  sgte
```

Supongamos que BRM tiene un reloj de 1 GHz. De las instrucciones que hay dentro del bucle «`esperatecl`», `LDRB` y `ANDS` se ejecutan en un ciclo de reloj, pero `BEQ`, cuando la condición se cumple (es decir, mientras no se haya pulsado una tecla y, por tanto, se vuelva al bucle) demora tres ciclos (apartado 3.2). En total, cada paso por el bucle se ejecuta en un tiempo $t_B = 5/10^9$ segundos. Por otra parte, imaginemos al campeón o campeona mundial de mecanografía tecleando con una velocidad de 750 pulsaciones por minuto[5]. El intervalo de tiempo desde que pulsa una tecla hasta que pulsa la siguiente es $t_M = 60/750$ segundos. La relación entre uno y otro es el número de veces que se ha

[5] `http://www.youtube.com/watch?v=M9lpqG9ZvGY`

ejecutado el bucle durante t_M: $t_M/t_B = 16 \times 10^6$. Por tanto, entre tecla y tecla el procesador ejecuta $3 \times 16 \times 10^6$ instrucciones que no hacen ningún trabajo «útil». Podría aprovecharse ese tiempo para ejecutar un programa que, por ejemplo, amenizase la tarea del mecanógrafo reproduciendo un MP3 con la séptima de Mahler o una canción de Melendi, a su gusto.

Interrupciones de periféricos de caracteres

El mecanismo de interrrupciones permite esa «simultaneidad» de la ejecución de un programa cualquiera y de la transferencia de caracteres. Como decíamos en el apartado 2.8, el mecanismo es una combinación de hardware y software:

En el hardware, el procesador, al atender a una interrupción, realiza las operaciones descritas en el apartado 3.7. Por su parte, y siguiendo con el ejemplo del *display* y el teclado, en los puertos de estado de los controladores, además del bit «preparado», PR, hay otro bit, «IT» que indica si el controlador tiene permiso para interrumpir o no. Cuando IT = 1 y PR se pone a uno (porque se ha pulsado una tecla, o porque el *display* está preparado para otro carácter) el controlador genera una interrupción.

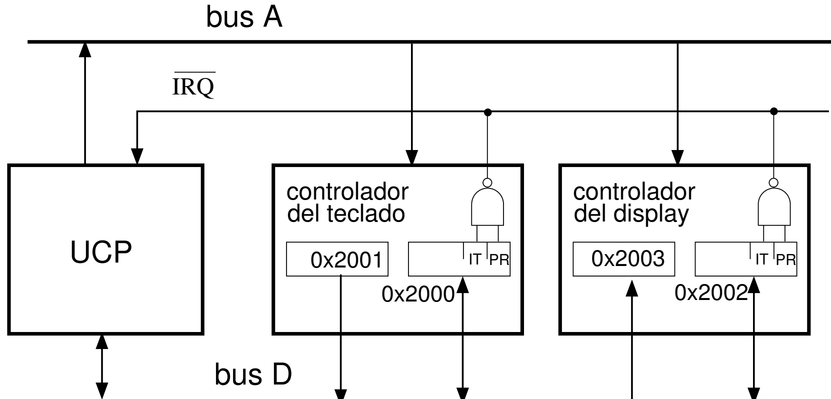


Figura 4.2 Conexión de controladores.

El procesador y los controladores están conectados como indica la figura 4.2. El símbolo sobre los bits PR e IT de cada controlador representa un circuito que realiza la operación lógica NAND. Es decir, cuando ambos bits son «1» (y solo en ese caso) la salida de ese circuito es «0», lo que activa a la línea $\overline{\text{IRQ}}$, que es la entrada de interrupciones externas al procesador.[6] Inicialmente, el hardware, antes de ejecutarse el programa de arranque en ROM (figura 3.12), inhibe las interrupciones de los controladores (IT = 0) y el permiso general de interrupciones (I = 0 en el CPSR, figura 3.2)

Por lo que respecta al software, veamos esquemáticamente lo que debe hacer, sin entrar ya en todos los detalles de la codificación en ensamblador (que no sería difícil, si se comprende bien lo que hay que hacer, pero alargaría excesivamente la explicación).

En primer lugar, hay unas operaciones iniciales que se realizan en modo supervisor. Para el ejemplo anterior (ir haciendo eco de los caracteres que se van tecleando) no es necesario que el *display*, que se supone más rápido que el mecanógrafo, interrumpa: basta atender a las interrupciones del teclado. Por tanto, la operación inicial sería poner «1» en el bit IT del puerto de estado del teclado y en el bit I del CPSR, después de haber cargado la rutina de servicio del teclado (la única) a partir de la dirección

[6] El motivo de que sea NAND y no AND y de que la línea se active con «0» y no con «1» lo comprenderá usted al estudiar electrónica digital. La línea $\overline{\text{IRQ}}$ es una «línea de colector abierto».

0xB00 (suponiendo el mapa de memoria de la figura 3.12). Esta rutina de servicio consistiría simplemente en leer (`LDRB`) del puerto de datos del teclado y escribir (`STRB`) en el del *display*, sin ninguna espera. Y terminaría con la instrucción `SUBS PC,LR,#4`, como explicamos en el apartado 3.7. Recuerde que al ejecutarse `LDRB` sobre el puerto de datos del teclado se pone a cero el bit PR, y no se vuelve a poner a uno (y, por tanto, generar una nueva interrupción) hasta que no se vuelve a pulsar una tecla. De este modo, para cada pulsación de una tecla solo se realizan las operaciones que automáticamente hace el procesador para atender a la interrupción y se ejecutan tres instrucciones, quedando todo el tiempo restante libre para ejecutar otros programas.

Rutinas de servicio de periféricos de caracteres

Ahora bien, la situación descrita es para una aplicación muy particular. Las rutinas de servicio tienen que ser generales, utilizables para aplicaciones diversas. Por ejemplo, un programa de diálogo, en el que el usario responde a cuestiones que plantea el programa. En estas aplicaciones hay que permitir también las interrupciones del *display*. Para que el software de interrupciones sea lo más genérico posible debemos prever tres componentes:

1. Para cada periférico, una **zona de memoria** *(buffer)* con una capacidad predefinida. Por ejemplo, 80 bytes. La idea es que conforme se teclean caracteres se van introduciendo en la zona del teclado hasta que el carácter es «ret» (ASCII 0x0D) que señala un «fin de línea», o hasta que se llena con los 80 caracteres (suponiendo que son ASCII o ISO). Igualmente, para escribir en el *display* se rellenará previamente la zona con 80 caracteres (o menos, siendo el último «ret»).

2. Para cada periférico, una **rutina de servicio**:

   - La del teclado debe contener una variable que se inicializa con cero y sirve de índice para que los caracteres se vayan introduciendo en bytes sucesivos de la zona. Cada vez que se ejecuta esta rutina se incrementa la variable, y cuando llega a 80, o cuando el carácter es «ret», la rutina inhibe las interrupciones del teclado (poniendo IT = 0). Desde el programa que la usa, y que procesa los caracteres de la zona, se volverá a inicializar poniendo a cero la variable y haciendo IT = 1 cuando se pretenda leer otra línea.
   - Análogamente, la rutina de servicio del *display* tendrá una variable que sirva de índice. El programa que la usa, cada vez que tenga que escribir una línea, rellenará la zona de datos con los caracteres, inicializará la variable y permitirá interrupciones. Cuando el periférico termina de escribir un carácter el controlador pone a uno el bit PR, provocando una interrupción, y cuando se han escrito 80, o cuando el último ha sido «ret» la rutina inhibe las interrupciones del periférico.

   Las zonas de memoria de cada periférico pueden estar integradas, cada una en una sección de datos, en las mismas rutinas de servicio.

3. La **identificación de la causa de interrupción**. En efecto, las peticiones de los dos periféricos llegan ambas al procesador por la línea $\overline{\text{IRQ}}$, y la primera instrucción que se ejecuta al atender a una es la de dirección 0xB00, de acuerdo con el mapa de memoria que estamos suponiendo (figura 3.12). Para saber si la causa ha sido el teclado o el *display*, a partir de esa dirección pondríamos unas instrucciones que leyeran uno de los puertos de estado, por ejemplo, el del teclado y comprobasen si los bits IT y PR tienen ambos el valor «1». Si es así, se bifurca a las operaciones de la rutina de servicio del teclado, y si no, a la del *display*.

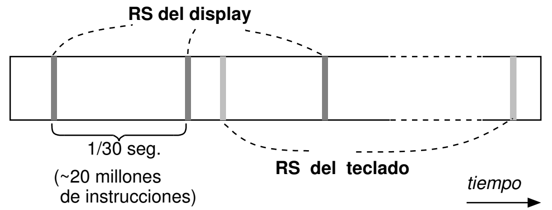


Figura 4.3 Tiempos de ejecución de las rutinas.

En una aplicación en la que ambos periféricos están trabajando resultaría un reparto de tiempos como el indicado en la figura 4.3. Se supone que el *display* tiene una tasa de transferencia de 30 caracteres por segundo, y que la del teclado es menor. Las zonas sombreadas representan los tiempos de ejecución de las rutinas de servicio, y las blancas todo el tiempo que puede estar ejecutándose el programa (o los programas) de aplicación. Así, la mayor parte del tiempo el procesador está ejecutando este programa, interrumpiéndose a veces para escribir un carácter en el *display* o para leer un carácter del teclado. Como todo esto es muy rápido (para la percepción humana), un observador que no sepa lo que ocurre podría pensar que el procesador está haciendo tres cosas al mismo tiempo.

Este efecto se llama **concurrencia aparente**. La **concurrencia real** es la que se da en sistemas multiprocesadores (apartado 2.5).

Consulta de interrupciones

En los ejemplos anteriores solo hay dos periféricos, pero en general puede haber muchos. Identificar cuál ha sido el causante de la interrupción implicaría una secuencia de instrucciones de comprobación de los bits PR e IT. Ésta sería una **consulta por software**. Normalmente no se hace así (salvo si es un sistema especializado con pocas causas de interrupción).

Como decíamos al final del apartado 2.8, los mecanismos de interrupciones suelen combinar técnicas de hardware y de software. De hecho, en el mecanismo que hemos visto el hardware, tanto el del procesador como el de los controladores de periféricos, se ocupa de ciertas cosas, y el resto se hace mediante software. Añadiendo más componentes de hardware se puede hacer más eficiente el procesamiento de las interrupciones.

Un esquema muy utilizado consiste en disponer de un **controlador de interrupciones** que permite prescindir de las instrucciones de indentificación de la causa, haciendo una **consulta por hardware**. El controlador tiene varias entradas, una para cada una de las líneas de interrupción de los periféricos, y una salida que se conecta a la línea $\overline{\text{IRQ}}$. Cuando un periférico solicita interrupción el controlador pone en un puerto un número identificativo y activa la línea $\overline{\text{IRQ}}$. El programa que empieza en la dirección correspondiente a $\overline{\text{IRQ}}$ (en nuestro mapa de memoria, la dirección 0xB00) solo tiene que leer ese puerto para inmediatamente bifurcar a la rutina de servicio que corresponda.

Periféricos de bloques

Imagínese, mirando la figura 4.3, que el *display* es mucho más rápido, digamos mil veces: 30.000 caracteres por segundo. El tiempo entre interrupciones se reduce de 33 ms a 33 μs. Suponiendo que el procesador ejecuta 600 MIPS (millones de instrucciones por segundo), el número de instrucciones que pueden ejecutarse en ese intervalo se reduce de 20 millones a veintemil. Está claro que cuanto mayor sea la tasa de transferencia del periférico menor será la proporción que hay entre las zonas blancas y las sombreadas de la figura. Si se trata de un periférico rápido, como un disco, con una tasa de 300 MB/s, el tiempo entre interrupciones se reduciría a 3,3 ns, y... ¡el número de instrucciones a 2! No hay ni tiempo para ejecutar una rutina de servicio.

Por esta razón, para periféricos rápidos, o periféricos de bloques, se sigue la técnica de acceso directo a memoria, o **DMA** (apartado 2.8). Para realizar una transferencia, el programa inicializa al controlador de DMA con los datos necesarios para que el controlador, y no la CPU, se ocupe de esa transferencia. Si se trata de un disco, estos datos son:

- Tamaño del bloque (número de bytes a transferir).
- Sentido de la transferencia (lectura del disco o escritura en el mismo).
- Dirección de la zona de memoria *(buffer)* que contiene los bytes o en la que se van a recibir.
- Localización física en el disco (número de sector, pista...).

Esta inicialización se realiza mediante instrucciones `STRB` sobre direcciones que corresponden a puertos del controlador. Una vez hecha, el controlador accede periódicamente a la memoria a través de los buses A y D, «robando» ciclos al procesador (para eso está la señal de control **wait**, figura 3.1) y se encarga de transferir los bytes *directamente* con la memoria, sin pasar por la CPU.

Cuando se han transferido todos los bytes del bloque, y solo en ese momento, el controlador de DMA genera una interrupción. La rutina de servicio comprueba si se ha producido algún error leyendo en un registro de estado del controlador.

4.9. Software del sistema y software de aplicaciones

En el apartado anterior hablábamos de «aplicaciones diversas» que utilizan los recursos a través del sistema de interrupciones y de los controladores de DMA. Estas aplicaciones son programas que se diseñan para resolver problemas generales (edición de textos, reproducción y conversión de formatos de audio y vídeo, navegadores…) o específicos (juegos, nóminas, cálculo de trayectorias…). Si el programador de una de esas aplicaciones tuviese que trabajar como a primera vista se deduce del apartado anterior, es decir, inicializando y manejando el sistema de interrupciones, programando los controladores, etc., el desarrollo sería una tarea inmensa. Como hemos visto en el Tema 1, para eso, entre otras cosas, están los *servicios* que ofrece el sistema operativo a través de *abstracciones*: el sistema de ficheros, el de entrada/salida, etc.

El software de aplicaciones, pues, hace uso de esos servicios, cuya implementación es «software del sistema». Esto ya lo sabíamos del Tema 1. Lo que quizás pueda usted entender mejor en este momento, en el que hemos descendido al nivel de máquina convencional, es el mecanismo básico en en el que se basa esa implementación. Y este mecanismo se apoya en una sola instrucción, la de *interrupción de programa*, `SWI` (en otros procesadores se llama `SVC`, o `TRAP`, o `INT`, pero esencialmente tiene la misma función). Con esta instrucción los programas de aplicación hacen **llamadas al sistema**.

La instrucción `SWI` tiene que ir acompañada de datos que identifiquen la operación a realizar. En el simulador ARMSim# se indica mediante un número incluido en la misma instrucción (que tiene el formato de la figura 3.11), como hemos visto en los ejemplos. En las implementaciones de los sistemas operativos es más común que ese número y otros datos se pongan en registros, siguiendo un convenio preestablecido, antes de la instrucción `SWI`. La rutina de servicio (cuya primera instrucción estará en la dirección 0x08, según la tabla 3.4) identificará la operación leyendo estos registros y llamará al componente necesario del sistema para realizarla.

4.10. Dos programas para Android

Los ejemplos anteriores de este Capítulo están preparados para ejecutarse en el simulador ARMSim#, y quizás se esté usted preguntando cómo podrían ejecutarse en un procesador real. Vamos a ver, con un par de ejemplos sencillos, que sí pueden adaptarse para que hagan uso de las llamadas en un dispositivo basado en ARM y un sistema operativo Linux o Android.

Hola mundo

En un primer curso de programación es tradicional empezar con el «Hola mundo»: un programa que escribe por la pantalla (la «salida estándar») esa cadena. El programa 4.13 lo hace en el ensamblador de BRM.

```
                   .text
                   .global _start
                   _start:
00000000 E3A00001     mov r0, #1      @ R0=1: salida estándar
00000004 E59F1014     ldr r1, =cadena @ R1: dirección de la cadena
00000008 E3A0200E     mov r2, #14     @ R2: longitud de la cadena
0000000C E3A07004     mov r7, #4      @ R7=4: número de "write"
00000010 EF000000     swi 0x00000000
00000014 E3A00000     mov r0, #0      @ R0=0: código de retorno normal
00000018 E3A07001     mov r7, #1      @ R7=1: número de "exit"
0000001C EF000000     swi 0x00000000
                   .data
                    cadena:
00000000 C2A1486F     .asciz "¡Hola mundo!\n"
         6C61206D
         756E646F
         210A00
                   .end
```

Programa 4.13 Programa «Hola mundo».

En un lenguaje de alto nivel, la llamada para escribir en un fichero es una función de nombre «`write`» que tiene como parámetros un número entero (el «descriptor del fichero», que en el caso de la salida estándar es 1), un puntero a la zona de memoria que contiene los datos a escribir y el número de bytes que ocupan éstos. En ensamblador, estos parámetros se pasan por registros: R0, R1 y R2, respectivamente. El número que identifica a `write` es 4, y se pasa por R7. Esto explica las cuatro primeras instrucciones que, *junto con `SWI`, forman la llamada al sistema.* Cuando el programa se ejecute, al llegar a `SWI` se produce una interrupción. La rutina de servicio de `SWI` (que ya forma parte del sistema operativo), al ver el contenido de R7 llama al sistema de gestión de ficheros, que a su vez interpreta los contenidos de R0 a R2 y llama al gestor del periférico, que se ocupa de iniciar la transferencia mediante interrupciones de la pantalla y el sistema operativo vuelve al programa interrumpido.

Después el programa hace otra llamada de retorno «normal» (sin error, parámetro que se pasa por R0), y el número de llamada (que en alto nivel se llama «`exit`»), 1, se introduce en R7 antes de ejecutarse la instrucción `SWI`. Ahora, la rutina de servicio llama al sistema de gestión de memoria, que libera el espacio ocupado por el programa y hace activo a otro proceso.

Saludo

Si el descriptor de la salida estándar es 1, el de la entrada estándar (el teclado) es 0. El programa 4.14 pregunta al usuario por su nombre, lo lee, y le saluda con ese nombre.

Adaptación de otros programas

Como muestran esos dos ejemplos, es fácil hacer uso de las llamadas «`read`» y «`write`» y que el sistema operativo se encargue de los detalles de bajo nivel para leer del teclado y escribir en la pantalla. Los programas de Fibonacci, del máximo común divisor o del factorial (que en los ejemplos prescindían de la entrada de teclado por limitaciones de ARMSim#, vea la tabla A.1) se pueden adaptar para que realicen los cálculos con datos del teclado.

Ahora bien, esas llamadas sirven para leer o escribir *cadenas de caracteres*. Si, por ejemplo, adaptamos el programa 4.2 para que diga «Dame un número y te digo si es de Fibonacci» y luego lea, y si escribimos «233» lo que se almacena en memoria es la cadena de caracteres «233». Por tanto, hace falta un subprograma que convierta esas cadenas a la representación en coma fija en 32 bits. Y si se trata de devolver un resultado numérico por la pantalla es necesario otro subprograma para la operación inversa. Le dejamos como ejercicio su programación, que conceptualmente es sencilla (recuerde el ejercicio planteado en clase, transparencia 36 del Tema 2), pero laboriosa (especialmente si, como debe ser, se incluye la detección de errores: caracteres no numéricos, etc.)

Ensamblando, montando, grabando y ejecutando

«¿Puedo comprobar que esto funciona en mi teléfono o tableta con Android?» Vamos a ver que no es difícil, pero antes hagamos unas observaciones:

- Como ejercicio es interesante comprobarlo, pero si está usted pensando en aprender a desarrollar aplicaciones es recomendable (por no decir imprescindible) utilizar entornos de desarrollo para lenguajes de alto nivel, como el que estudiará en la asignatura «Análisis de diseño de software».
- El procedimiento se ha probado en dos dispositivos Samsung: Galaxy S con Android 2.2 y Galaxy Tab 10.1 con Android 3.2, y debería funcionar con cualquier otro dispositivo basado en ARM y cualquier otra versión de Android.
- La grabación del programa ejecutable en el dispositivo requiere adquirir previamente privilegios de administrador (en la jerga, «rootearlo») y ejecutar algunos comandos como tal. Como esto puede conllevar algún riesgo (en el peor de los casos, y también en la jerga, «brickearlo»: convertirlo en un ladrillo), ni el autor de este documento ni la organización en la que trabaja se hacen responsables de eventuales daños en su dispositivo.
- Necesitará:
  - un ordenador personal con un ensamblador cruzado, un montador y un emulador como `qemu` (aunque éste último no es imprescindible), y leer las indicaciones del Apéndice (apartado A.3) sobre su uso;
  - el dispositivo con Android modificado para tener permisos de administrador y con una aplicación de terminal (por ejemplo, «Terminal emulator»);
  - una conexión por cable USB (o por alguna aplicación wifi) para transferir el programa ejecutable del ordenador al dispositivo.

```
                  .text
                  .global _start
                  _start:
00000000 E3A00001    mov r0, #1      @ R0 = 1: salida estándar
00000004 E59F1064    ldr r1, =cad1
00000008 E3A02014    mov r2, #20     @ En UTF-8 son 20 bytes
0000000C E3A07004    mov r7, #4      @ R7 = 4: número de "write"
00000010 EF000000    swi 0x00000000
00000014 E3A00000    mov r0, #0      @ R0 = 0: entrada estándar
00000018 E59F1054    ldr r1, =nombre
0000001C E3A02019    mov r2, #25
00000020 E3A07003    mov r7, #3      @ R7 = 3: número de "read"
00000024 EF000000    swi 0x00000000
00000028 E3A00001    mov r0, #1      @ R0 = 1: salida estándar
0000002C E59F1044    ldr r1, =cad2
00000030 E3A02006    mov r2, #6
00000034 E3A07004    mov r7, #4      @ R7 = 4: número de "write"
00000038 EF000000    swi 0x00000000
0000003C E3A00001    mov r0, #1
00000040 E59F102C    ldr r1, =nombre
00000044 E3A02019    mov r2,#25
00000048 E3A07004    mov r7, #4
0000004C EF000000    swi 0x00000000
00000050 E3A00001    mov r0, #1
00000054 E59F1020    ldr r1, =cad3
00000058 E3A02002    mov r2,#2
0000005C E3A07004    mov r7, #4
00000060 EF000000    swi 0x00000000
00000064 E3A00000    mov r0, #0      @ R0 = 0: código de retorno normal
00000068 E3A07001    mov r7, #1      @ R7 = 1: número de "exit"
0000006C EF000000    swi 0x00000000
                  .data
00000000 C2BF43C3    cad1: .asciz "¿Cómo te llamas?\n>"
         B36D6F20
         7465206C
         6C616D61
         733F0A3E
00000015 00000000    nombre: .skip 25
         00000000
         00000000
         00000000
         00000000
0000002E 486F6C61    cad2: .asciz "Hola, "
         2C2000
00000035 0A00        cad3: .asciz "\n"
              .end
```

Programa 4.14 Programa «Saludo».

El procedimiento, resumido, y ejemplificando con el programa «saludo», cuyo programa fuente estaría en un fichero de nombre `saludo.s`, sería:

1. *Ensamblar* el programa fuente: `arm-linux-gnueabi-as -o saludo.o saludo.s`
2. *Montar* el programa (ya hemos comentado en el apartado 4.4 que este paso es necesario aunque sólo haya un módulo): `arm-linux-gnueabi-ld -o saludo saludo.o`
3. *Simular* la ejecución para comprobar que funciona correctamente: `qemu-arm saludo`
4. *Grabar* el programa ejecutable en un directorio del dispositivo con permisos de ejecución, por ejemplo, en `/data/local/bin`. La forma más cómoda de hacerlo es con `adb` (apartado A.3), pero para este sencillo ejercicio se puede conseguir de otro modo sin necesidad de instalar más herramientas en el ordenador:

   4a. Conectar el dispositivo al ordenador y transferir el fichero `saludo`.

   4b. Para lo anterior no hacen falta privilegios, pero el fichero queda grabado en `/mnt/sdcard` (o en algún subdirectorio), y ahí no puede ejecutarse. Es preciso moverlo a, o copiarlo, en un directorio en el que se pueda ejecutar. Por ejemplo, `/data/local/bin`, y esto sí requiere privilegios.

   4c. Abrir el terminal (en el dispositivo) y hacerse «root» (`su`).

   4d. Comprobar que existe `/data/local/bin`, y si no es así, crearlo (`mkdir`).

   4e. Ahora procedería hacer: `#mv /mnt/sdcard/saludo /data/local/bin`, pero hay un problema: Android utiliza distintos sistemas de ficheros para la memoria en la que está montado `sdcard` y la memoria del sistema, y el programa que implementa `mv` da un error. Una manera de conseguirlo es: `#cat /mnt/sdard/saludo >/data/local/bin/saludo`.

   4f. Asegurarse de que `saludo` tiene permisos de ejecución (755), y si es necesario ponérselos con `chmod`.

5. Salir de administrador (`#exit`). Para que el programa sea accesible desde cualquier punto, incluir la ruta: `export PATH=/data/local/bin:$PATH` (la aplicación «Terminal emulator» lo hace por defecto al iniciarse).
6. Ya puede ejecutarse el programa (como usuario normal) escribiendo en el terminal `saludo`. (Si no aparecen los caracteres no-ascii «¿» y «ó», configurar el terminal para que acepte UTF-8).

La figura 4.4 muestra esas operaciones realizadas en el terminal (se ha ocultado el teclado virtual para que se vean todas las líneas), y en la figura 4.5 puede verse el resultado de la ejecución.

```
$ su
# ls -l /mnt/sdcard/saludo
-rwxrwxr-x system   sdcard_rw     1123 2012-06-13 21:30 salud
o
# ls /data/local/bin
/data/local/bin: No such file or directory
# mkdir /data/local/bin
# mv /mnt/sdcard/saludo /data/local/bin
failed on '/mnt/sdcard/saludo' - Cross-device link
# cat /mnt/sdcard/saludo > /data/local/bin/saludo
# ls -l /data/local/bin
-rw-rw-rw- root     root          1123 2012-06-13 20:00 saludo
# chmod 755 /data/local/bin/saludo
# exit
$
```

Figura 4.4 Órdenes para copiar el programa ejecutable en un directorio con permisos de ejecución.

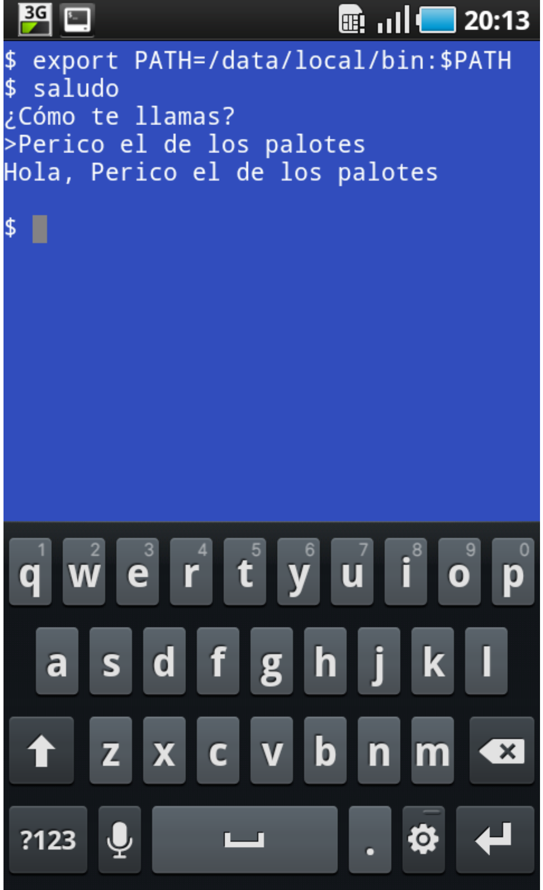


Figura 4.5 Ejecución del programa `saluda`.

Capítulo 5

Procesadores software y lenguajes interpretados

En el apartado 1.2 introducíamos las ideas básicas de los lenguajes simbólicos de bajo y alto nivel. Los programas fuente escritos en estos lenguajes necesitan ser traducidos al lenguaje de máquina del procesador antes de poderse ejecutar. El traductor de un lenguaje de bajo nivel, o lenguaje ensamblador, se llama ensamblador, y el de uno de alto nivel, compilador.

Algunos lenguajes de alto nivel están diseñados para que los programas se ejecuten mediante un proceso de interpretación, no de traducción. Funcionalmente, la diferencia entre un traductor y un intérprete es la misma que hay entre las profesiones que tienen los mismos nombres. El traductor recibe como entrada todo el programa fuente, trabaja con él, y tras el proceso genera un código objeto binario que se guarda en un fichero que puede luego cargarse en la memoria y ejecutarse. El intérprete analiza también el programa fuente, pero no genera ningún fichero, sino las órdenes oportunas (instrucciones de máquina o llamadas al sistema operativo) para que se vayan ejecutando las *sentencias* (apartado 1.2) del programa fuente. Abusando un poco de la lengua, se habla de «lenguajes compilados» y de «lenguajes interpretados» (realmente, lo que se compila o interpreta no es el lenguaje, sino los programas escritos en ese lenguaje).

En este capítulo estudiaremos los principios de los procesos de traducción, montaje e interpretación de una manera general (independiente del lenguaje). En la asignatura «Programación» estudiará usted lenguajes compilados, por lo que no entraremos aquí en ellos (salvo por algún ejemplo para ilustrar el proceso de compilación). Sí veremos las ideas básicas de algunos lenguajes interpretados que tienen una importancia especial en las aplicaciones telemáticas: lenguajes de marcas y lenguajes de *script*, e introduciremos mediante ejemplos tres de ellos: HTML, XML y JavaScript.

5.1. Ensambladores

Para generar el código objeto, un ensamblador sigue, esencialmente, los mismos pasos que tendríamos que hacer para traducir manualmente el código fuente. Tomemos como ejemplo el programa 4.2:

Después de saltar las primeras líneas de comentarios y de guardar memoria de que cuando aparezca el símbolo «`Num`» tenemos que reemplazarlo por 233 = 0xE9, encontramos la primera instrucción, `mov r2,#0`. El código de operación nemónico nos dice que es una instrucción del tipo «movimiento»,

cuyo formato es el de la figura 3.5. Como no tiene condición (es `MOV`, no `MOVEQ`, ni `MOVNE`, etc.), los bits de condición, según la tabla 3.1, son 1110 = 0xE. El operando es inmediato, por lo que el bit 25 es `I = 1`, y el código de operación de `MOV` (tabla 3.2) es 1101. El bit 20 es S = 0 (sería S = 1 si el código nemónico fuese `MOVS`). Ya tenemos los doce bits más significativos: 1110-00-1-1101-0 = 0xE3A. Los cuatro bits siguientes, Rn, son indiferentes para `MOV`, y ponemos cuatro ceros. Los siguientes indican el registro destino, Rd = R2 = 0010 y finalmente, como el operando inmediato es 0 (menor que 256), se pone directamente en los doce bits del campo Op2. Llegamos así a la misma traducción que muestra el listado del programa 4.2: `0xE3A02000`.

De esta manera mecánica y rutinaria seguiríamos hasta la instrucción `beq si` que, a diferencia de las anteriores, no podemos traducir aisladamente. En efecto, su formato es el de la figura 3.10, y, como la condición es «`eq`» y L = 0, podemos decir que los ocho bits más significativos son 0x0A, pero luego tenemos que calcular una distancia que depende del valor que tenga el símbolo «`si`». Para ello, debemos avanzar en la lectura del código fuente hasta averiguar que su valor (su dirección) es 0x2C = 44. Como la dirección de la instrucción que estamos traduciendo es 0x18 = 24, habremos de poner en el campo «Dist» un número tal que DE = 44 = 24 + 8 + 4×(Dist), es decir, (Dist) = 3, resultando así la traducción completa de la instrucción: 0x0A000003.

Símbolo	Valor
`Num`	0xE9
`bucle`	0x10
`si`	0x2C
`no`	0x30

Tabla 5.1. Tabla de símbolos para el programa 4.2.

En lugar de este proceso, que implica «avanzar» a veces en la lectura para averiguar el valor correspondiente a un símbolo y luego «retroceder» para completar la traducción de una instrucción, podemos hacer la traducción en *dos pasos:* en el primero nos limitamos a leer el programa fuente e ir anotando los valores que corresponden a los símbolos; terminada esta primera lectura habremos completado una **tabla de símbolos internos**. En el ejemplo que estamos considerando resulta la tabla 5.1

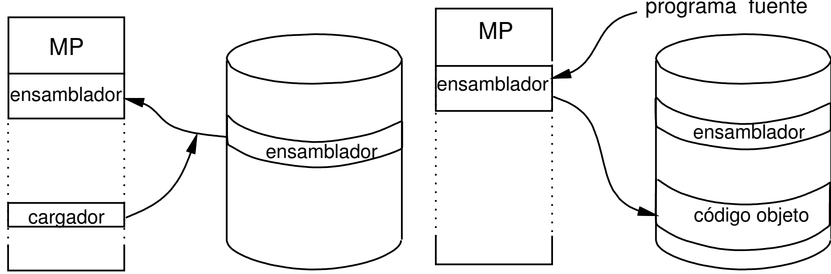
En el *segundo paso* recomenzamos la lectura y vamos traduciendo, consultando cuando es necesario la tabla.

Pues bien, los programas ensambladores también pueden ser de un paso o (más frecuentemente) de dos pasos, y su ejecución sigue, esencialmente, este modelo procesal. Hemos obviado varias operaciones que puede usted imaginar fácilmente: las comprobaciones de errores sintácticos en el primer paso, como códigos de operación que no existen, instrucciones mal formadas, etc.

Por otra parte, es importante tener una idea clara de la secuencia de actividades que deben seguirse para ejecutar un programa escrito en lenguaje ensamblador. La figura 5.1 ilustra estas actividades, aunque le falta algo. El símbolo en forma de cilindro representa un disco. En este disco tendremos disponible, entre otros muchos ficheros, el programa ensamblador en binario («ejecutable»).

1. Se carga el programa ensamblador mediante la ejecución de otro programa, el cargador, que tiene que estar previamente en la MP[1].
2. Al ejecutarse el ensamblador, las instrucciones que forman nuestro programa fuente *son los datos* para el ensamblador. Éste, a partir de esos datos, produce como resultado un programa escrito en lenguaje de máquina (un «código objeto»), que, normalmente, se guarda en un fichero del disco.
3. Se carga el código objeto en la MP. El ensamblador ya no es necesario, por lo que, como indica la figura, se puede hacer uso de la zona que éste ocupaba.
4. Se ejecuta el código objeto.

[1]Si el cargador está en la MP, tiene que haber sido cargado. Recuerde del Tema 1 que este círculo vicioso se rompe gracias al cargador inicial *(bootstrap loader)*.



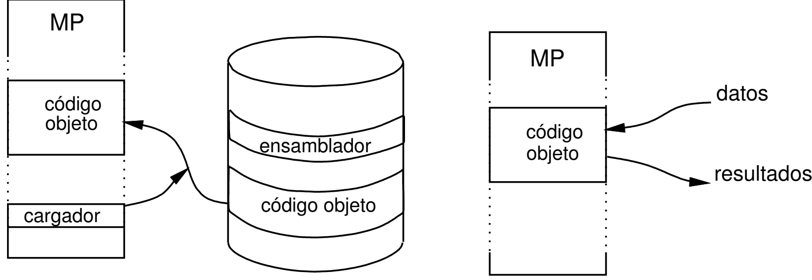


Figura 5.1 Procesos para el ensamblaje y la ejecución (incompleto).

¿Qué es lo que falta? Ya lo hemos avanzado en el apartado 4.4: el programa fuente normalmente contiene símbolos que se importan de otros módulos, por lo que el código objeto que genera el ensamblador no es aún ejecutable. Falta un proceso intermedio entre los pasos 2 y 3.

5.2. El proceso de montaje

El ensamblador genera, para cada módulo, un código objeto *incompleto:* además del código, genera una **tabla de símbolos de externos** que contiene, para cada símbolo a importar, la dirección en la que debe ponerse su valor, y una **tabla de símbolos de acceso** que contiene, para cada símbolo exportado, su valor definido en este módulo. Además, el ensamblador no sabe en qué direcciones de la memoria va a cargarse finalmente el módulo, por lo que, como hemos visto en los ejemplos, asigna direcciones a partir de 0. Pero hay contenidos de palabras que pueden tener que ajustarse en función de la dirección de carga, por lo que genera también un **diccionario de reubicación**, que no es más que una lista de las direcciones cuyo contenido debe ajustarse.

Tomemos ahora como ejemplo el de los tres módulos del apartado 4.7. Probablemente no habrá usted reparado en la curiosa forma que ha tenido el ensamblador de traducir la instrucción `bl fact` del programa 4.10 (felicitaciones si lo ha hecho): `0xEBFFFFFE`. Los ocho bits más significativos, 0xEB = 0b11101011, corresponden efectivamente, según el formato de la figura 3.10, a la instrucción `BL`, pero para la distancia ha puesto 0xFFFFFE, que es la representación de −2. La dirección efectiva resultaría así: DE = 0xC +8 + (−4 × 2) = 0xC, es decir, la misma dirección de la instrucción. Lo que ocurre es que el ensamblador no puede saber el valor del símbolo `fact`, que está declarado como un símbolo a importar («`.extern`»), y genera una tabla de símbolos externos que en este caso solamente tiene uno: el

nombre del símbolo y la dirección de la instrucción cuya traducción no ha podido completar. Lo mismo ocurre en el programa 4.11 con la instrucción `bl mult`. En este otro programa, `fact` está declarado como símbolo a exportar («`.global`»), y el ensamblador lo pone en la tabla de símbolos de acceso con su valor (0x0). Será el montador el que finalmente ajuste las distancias en las instrucciones de cada módulo teniendo en cuenta las tablas de símbolos de acceso de los demás módulos y el orden en que se han montado todos. Si, por ejemplo, ha puesto primero el programa 4.11 (a partir de la dirección 0) y a continuación el 4.10 (a partir de 0x38 + 4 = 60), la instrucción `bl fact` quedará en la dirección 0x0C + 60 = 72, y su distancia deberá ser Dist = $-72 \div 4 + 8 = -26$ =-0x1A, o, expresada en 24 bits con signo, 0xFFFFEC. La codificación de la instrucción después del montaje será: 0xEBFFFFEC, como puede usted comprobar si carga en ARMSim# los tres módulos en el orden indicado.

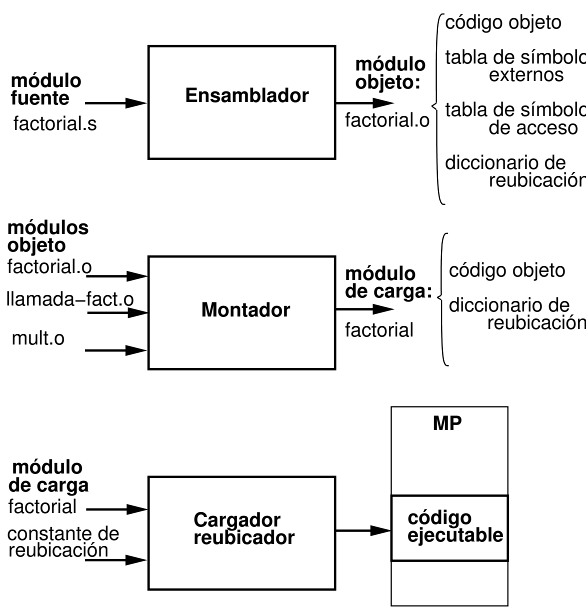


Figura 5.2. Procesos para el ensamblaje, el montaje y la ejecución.

Por si analizando los detalles ha perdido usted la idea general, la figura 5.2 la resume. Los tres módulos con los programas fuente, que suponemos guardados en ficheros de nombres `factorial.s`, `llamada-fact.s` y `mult.s`, se ensamblan independientemente, dando como resultado módulos objeto que se guardan en los ficheros a los que hemos llamado `factorial.o`, `llamada-fact.o` y `mult.o`. El montador, con estos tres módulos, resuelve las referencias incompletas y genera un fichero ejecutable («módulo de carga») con el nombre que le digamos, por ejemplo, `factorial`.

Pero no hemos mencionado en este ejemplo a los «diccionarios de reubicación». El montador genera un código que luego el cargador pondrá en la memoria a partir de alguna dirección libre (la que le diga el sistema de gestión de la memoria, Tema 1). Esta dirección de carga es desconocida para el montador. De hecho, si el programa se carga varias veces, será distinta cada vez. Lo que ocurre en este ejemplo es que no hay nada que «reubicar»: el código generado por el montador funciona independientemente de la dirección a partir de la cual se carga, y los diccionarios de reubicación están vacíos.

Consideremos el programa 4.3. El ensamblador crea un «pool de literales» inmediatamente después del código, en las direcciones 0x0000001C y 0x00000020, cuyos contenidos son las direcciones de los símbolos `palabra1` y `palabra2`. Estos símbolos están definidos en la sección «`.data`», que es independiente de la sección «`.text`», y sus direcciones (relativas al comienzo de la sección) son 0 y 4, como se puede ver en el listado del programa 4.3. Naturalmente, esos valores se tendrán que modificar para que sean las direcciones donde finalmente queden los valores `0xAAAAAAAA` y `0xBBBBBBBB`. Pero el ensamblador no puede saberlo, por lo que pone sus direcciones relativas, 0x0 y 0x4 y acompaña el código objeto con la lista de dos direcciones, 0x0000001C y 0x00000020, cuyo contenido habrá que modificar. Esta lista es lo que se llama «diccionario de reubicación». Si el montador coloca la sección de datos inmediatamente después de la de código, reubica los contenidos de las direcciones del diccionario, sumándoles la dirección de comienzo de la sección de datos, 0x24. El código que genera

el montador empieza en la dirección 0 y va acompañado también de su diccionario de reubicación (en este caso no hay más que un módulo, y el diccionario es el mismo de la salida del ensamblador; en el caso general, será el resultado de todos los diccionarios). Si luego se carga a partir de la dirección 0x1000, a todos los contenidos del diccionario hay que sumarles durante el proceso de la carga el valor 0x1000 («constante de reubicación»). De ahí resultan los valores que veíamos en los comentarios del programa 4.3.

5.3. Compiladores e intérpretes

Un lenguaje de alto nivel permite expresar los algoritmos de manera independiente de la arquitectura del procesador. El pseudocódigo del algoritmo de Euclides para el máximo común divisor (apartado 4.5) conduce inmediatamente, sin más que obedecer unas sencillas reglas sintácticas, a una función en el lenguaje C:

```
int MCD(int x, int y) {
/* Esta es una función que implementa el algoritmo de Euclides
   Recibe dos parámetros de entrada de tipo entero
    y devuelve un parámetro de salida de tipo entero */
  while (x != y) {
    if (x > y) x = x - y;
    else y = y - x;
  }
  return x;
}
```

Compiladores

A partir de este código fuente, basado en construcciones que son generales e independientes de los registros y las instrucciones de ningún procesador (variables, como «x» e «y», sentencias como «while» e «if»), un compilador genera un código objeto para un procesador concreto. El proceso es, por supuesto, bastante más complejo que el de un ensamblador. A grandes rasgos, consta de cuatro fases realizadas por los componentes que muestra la figura 5.3.

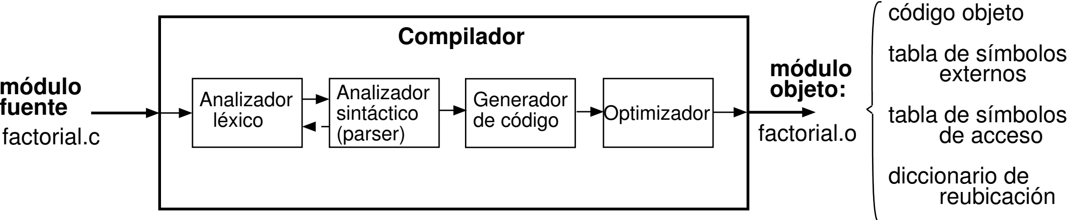


Figura 5.3 Componentes de un compilador.

Análisis léxico

El **análisis léxico** consiste en la exploración de los caracteres del código fuente para ir identificando secuencias que tienen un significado y se llaman «**tokens**»: «while» es un *token*, y «x» es otro. El

analizador léxico los clasifica y se los va entregando al siguiente componente: «`while`» es una palabra clave del lenguaje, mientras que «`x`» es un símbolo definido en este programa. En este proceso, el analizador va descartando los comentarios (como los que aparecen entre «`/*`» y «`*/`»), de modo que en la siguiente fase ya se trabaja a un nivel superior al de los caracteres individuales.

Análisis sintáctico *(parsing)*

El **análisis sintáctico** se suele llamar (sobre todo, por brevedad) **parsing**, y el programa que lo realiza, **parser**. Su tarea es analizar sentencias completas, comprobar su corrección y generar estructuras de datos que permitirán descomponer cada sentencia en instrucciones de máquina. Como indica la figura, el *parser* trabaja interactivamente con el analizador léxico: cuando recibe un *token*, por ejemplo, «`while`», la sintaxis del lenguaje dice que tiene que ir seguido de una expresión entre paréntesis, por lo que le pide al analizador léxico que le entregue el siguiente *token*: si es un paréntesis todo va bien, y le pide el siguiente, etc. Cuando llega al final de la sentencia, señalado por el *token* «`;`», el *parser* entrega al generador de código un **árbol sintáctico** que contiene los detalles a partir de los cuales pueden obtenerse las instrucciones de máquina necesarias para implementar la sentencia.

Para que el *parser* funcione correctamente es necesario que esté basado en una definición de la sintaxis del lenguaje mediante estrictas reglas gramaticales. Un metalenguaje para expresar tales reglas es la **notación BNF** (por «Backus–Naur Form»). Por ejemplo, tres de las reglas BNF que definen la sintaxis del lenguaje C son:

```
<sentencia> ::= <sentencia-for>|<sentencia-while>|<sentencia-if>|...
<sentencia-while> ::= 'while' '(' <condición> ')' <sentencia>
<sentencia-if> ::= 'if' '('<condición>')'<sentencia>['else'<sentencia>]
```

La primera dice que una sentencia puede ser una sentencia «for», o una «while», o... La segunda, que una sentencia «while» está formada por la palabra clave «`while`» seguida del símbolo «`(`», seguido de una condición, etc.

BNF es un metalenguaje porque es un lenguaje para definir la gramática de los lenguajes de programación. Las reglas se pueden expresar gráficamente con **diagramas sintácticos**, como los de la figura 5.4, que corresponden a las tres reglas anteriores.

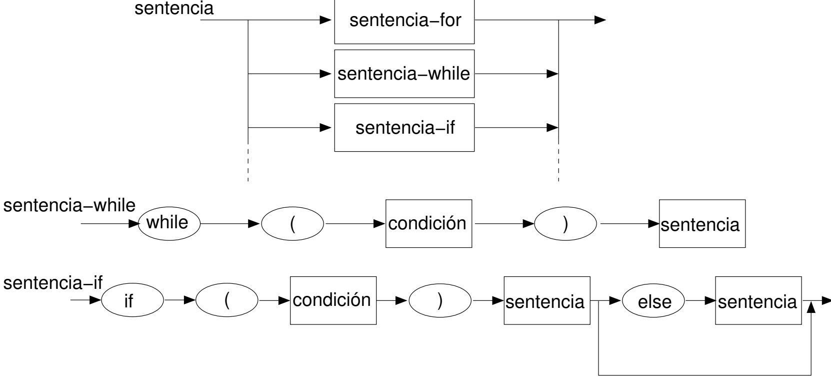


Figura 5.4 Diagramas sintácticos.

Si el programa fuente está construido respetando estas reglas, el *parser* termina construyendo el árbol sintáctico que expresa los detalles de las operaciones a realizar. La figura 5.5 presenta el correspondiente a la función MCD. Esta representación gráfica es para «consumo humano». Realmente, se implementa como una estructura de datos (las estructuras de datos se estudian en la asignatura «Programación»).

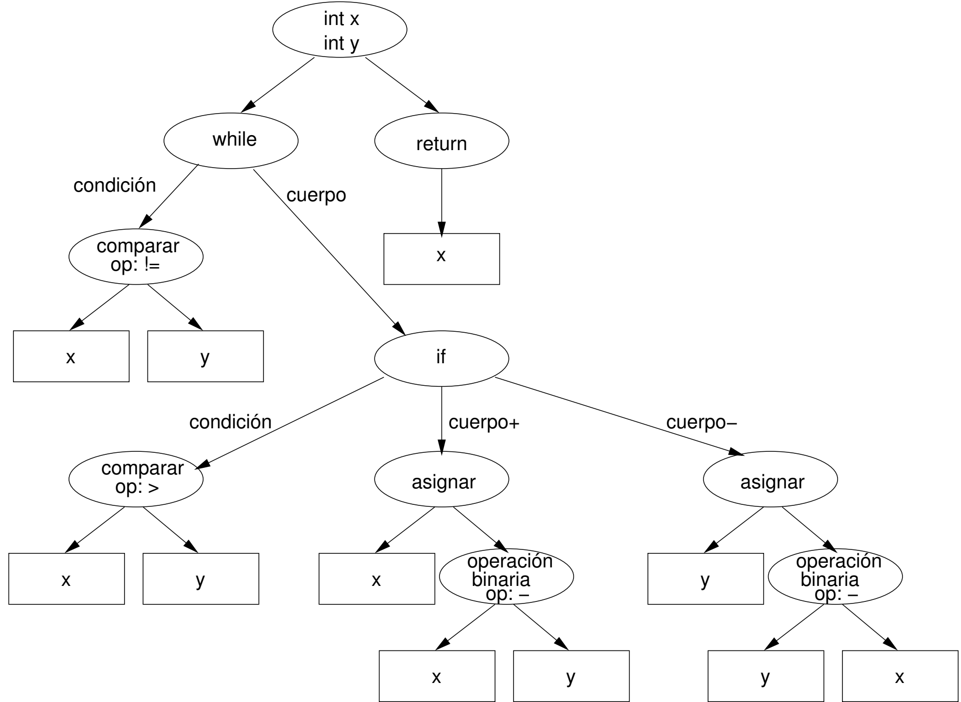


Figura 5.5 Árbol sintáctico de la función MCD.

Generación de código

La **generación de código** es ya una operación dependiente de la arquitectura, puesto que se trata de generar instrucciones de máquina específicas.

Recuerde (apartado 4.6) que en los compiladores se suele seguir el convenio de pasar los parámetros de las funciones por la pila. En el ejemplo de la función MCD, el generador, al ver que hay dos parámetros de entrada que son de tipo entero, les asigna dos registros, por ejemplo, r0 y r1, y genera las instrucciones de máquina adecuadas para extraer los valores de la pila e introducirlos en los registros. A continuación ve «while» con su condición, que es «comparar utilizando el operando !=»; si está generando código para ARM produce cmp r0,r1 y beq <dirección> (el valor de <dirección> lo pondrá en un segundo paso), y así sucesivamente, va generando instrucciones conforme explora el árbol sintáctico.

Optimización

En la última fase el compilador trata de conseguir un código más eficiente. Por ejemplo, si compilamos la función MCD (factorial.c) para la arquitectura ARM *sin* optimización, se genera un

código que contiene instrucciones `b`, `beq` y `blt`, como la primera de las versiones en ensamblador que habíamos visto en el apartado 4.5. Pidiéndole al compilador que optimice y que entrege el resultado en lenguaje ensamblador[2] (sin generar el código binario) se obtiene este resultado:

```
.L7:    cmp     r0, r1
        rsbgt   r0, r1, r0
        rsble   r1, r0, r1
        cmp     r1, r0
        bne     .L7
```

(«`.L7`» es una etiqueta que se ha «inventado» el compilador).

No utiliza las mismas instrucciones, pero puede usted comprobar que es equivalente al que habíamos escrito en el apartado 4.5. Ha optimizado bastante bien, pero no del todo (le sobra una instrucción).

Intérpretes

La diferencia básica entre un compilador y un intérprete es que éste no genera ningún tipo de código: simplemente, va ejecutando el programa fuente a medida que lo va leyendo y analizando. Según cómo sea el lenguaje del programa fuente, hay distintos tipos de intérpretes:

- Un procesador hardware es un intérprete del lenguaje de máquina.
- La implementación del lenguaje Java (que se estudia en la asignatura «Programación») combina una fase de compilación y otra de interpretación. En la primera, el programa fuente se traduce al lenguaje de máquina de un procesador llamado «máquina virtual Java» (JVM). Es «virtual» porque normalmente no se implementa en hardware. El programa traducido a ese lenguaje (llamado «bytecodes») se ejecuta mediante un programa intérprete.
- Hay lenguajes de alto nivel, por ejemplo, JavaScript o PHP, diseñados para ser interpretados mediante un programa intérprete. Son los «lenguajes interpretados».

En el último caso, el intérprete realiza también las tareas de análisis léxico y sintáctico, pero a medida que va analizando las sentencias las va ejecutando. El proceso está dirigido por el bloque «ejecutor» (figura 5.6): así como el *parser* le va pidiendo *tokens* al analizador léxico, el ejecutor le va pidiendo sentencias al *parser*.

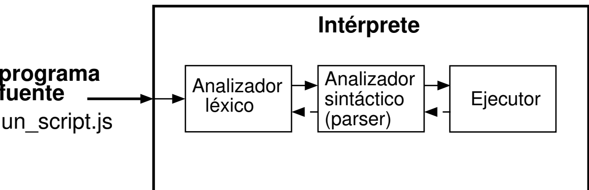


Figura 5.6 Componentes de un intérprete.

En JavaScript la sintaxis de las sentencias `while` e `if` es la misma de C. Para la función MCD, el *parser* genera el mismo árbol sintáctico de la figura 5.5. Cuando el ejecutor encuentra `while` y su condición simplemente comprueba si la condición se cumple; si no se cumple, de acuerdo con el árbol sintáctico, termina devolviendo el valor de la variable `x`; si se cumple le pide al *parser* el análisis del cuerpo de `while`, y así sucesivamente.

[2]Se puede comprobar con las herramientas que se explican en el apartado A.3, concretamente, con el compilador GNU (`gcc`) y las opciones «`-O`» (para optimizar) y «`-S`» (para generar ensamblador).

5.4. Lenguajes de marcas

Tanto C como JavaScript como Java son lenguajes **imperativos** o **procedimentales**: sus sentencias son «órdenes» que o bien se interpretan o bien se traducen a otras órdenes para un procesador; el programador tiene que especificar, paso a paso, la secuencia de acciones que debe ejecutar el intérprete (o que tiene que traducir el compilador). Hay otra clase de lenguajes que son **declarativos**: el programador no especifica *cómo* se resuelve el problema, sino *el* problema. El intérprete se encarga de generar las órdenes adecuadas para resolverlo.

En el Tema 5 estudiaremos SQL, un lenguaje de consulta de bases de datos, que es en parte declarativo. El más conocido de los lenguajes declarativos es «Prolog», pero no es de él que nos vamos a ocupar ahora, sino de una clase de lenguajes que pueden considerarse declarativos porque con ellos no se componen «programas» en el sentido de «secuencias de órdenes para un procesador», sino textos con anotaciones sobre cómo deben presentarse, o cómo deben interpretarse algunos fragmentos. Esto es muy abstracto. Lo mejor es verlo con lo esencial de dos lenguajes concretos: HTML y XML.

HTML

HTML (Hypertext Markup Language) recibe el nombre de su característica principal: un «hipertexto» es un documento de texto acompañado de anotaciones, o marcas, que indican propiedades para la presentación (tamaño, color, etc.), o remiten a otras partes del documento o a otros documentos (hiperenlaces, o simplemente, «enlaces», para abreviar). En la red formada por los enlaces no solo hay documentos de texto: también se enlazan ficheros con imágenes, sonidos y vídeos, de modo que el concepto de «hipertexto» se generaliza a «hipermedia».

Desde 1995 se han sucedido varias versiones de HTML. Nos vamos a referir a la última, HTML5, que aún está en desarrollo, pero las últimas versiones de los navegadores aceptan la mayoría de sus nuevos elementos.

La aplicación principal de HTML es la composición de páginas web. Los datos para la presentación de una página web están en uno o varios ficheros del servidor y se transmiten mediante el protocolo HTTP (Hypertext Transfer Protocol) o HTTPS (HTTP Secure). Cuando el cliente (el navegador) los recibe, el intérprete que tiene incorporado los analiza y genera las órdenes al sistema gráfico para que haga la presentación. El «esqueleto» de un página web es así:

```
<HTML>
<HEAD>
  <!-- Esto es un comentario. Dentro de "HEAD" se incluyen el título
       del documento y marcas "META" que describen el contenido
       (codificación de caracteres, palabras clave, autor, etc.)
  -->
</HEAD>
<BODY>
  <!-- Cuerpo del documento: marcas y texto
       que el navegador presenta en pantalla -->
</BODY>
</HTML>
```

Unas cadenas de caracteres corresponden a marcado y otras a contenido. El marcado y el contenido se diferencian por unas reglas sintácticas muy sencillas: todas las cadenas de marcado empiezan con el carácter «<» y terminan con «>» (o, como veremos luego, pueden empezar con «&» y terminar con «;»).

Los espacios en blanco y los cambios de línea entre cadenas de marcado solo sirven para hacer más legible el fichero fuente: para el navegador sería exactamente igual esto otro:

```
<HTML><HEAD>...</HEAD><BODY>...</BODY></HTML>
```

Las cadenas que empiezan con «<» y terminan con «>» son **etiquetas**. Las hay de apertura, como <BODY>, de cierre, como </BODY> y sin contenido, como <BR>, que provoca un salto de línea, o como <!-- ... -->, que solo sirve para poner comentarios en el documento fuente, comentarios que el intérprete ignora.

Se llaman **elementos** los componentes del documento que o bien empiezan con una etiqueta de apertura y terminan con la de cierre o bien consisten solo en una etiqueta sin contenido (como <BR>).

En el «esqueleto» anterior, el elemento <HTML> es el **elemento raíz**, y tiene como contenido todo el documento. Este contenido incluye dos elementos («hijos» de <HTML>): la cabecera (<HEAD>) y el cuerpo (<BODY>). Estas etiquetas deben ir por parejas, señalando el principio y el fin (</HTML>, </HEAD>, </BODY>) de cada sección, y deben estar convenientemente anidadas: no se puede poner <BODY> antes de «cerrar» <HEAD>, por ejemplo, pero dentro tanto de <HEAD> como de <BODY> van otros elementos («hijos») con sus etiquetas de apertura y cierre (o con solo etiqueta, si no tienen contenido).

Hemos utilizado mayúsculas para mejor identificar las marcas, pero el lenguaje es insensible a la caja, y con minúsculas el documento resulta más legible, como puede apreciarse en este ejemplo, que incluye algunos de los elementos definidos en HTML5:

```
<!DOCTYPE html>
<html lang="es-ES">
<head>
 <meta charset="UTF-8">
 <title>Introducción a HTML5</title>
</head>
<body>
<h1 style="text-align:center;">Algunos elementos de HTML5</h1>
<p>El elemento <kbd>&lt;p&gt;</kbd> contiene párrafos de texto
y otros elementos.En cualquier punto se puede forzar
un salto de línea<br> con el elemento vacío <kbd>&lt;br&gt;</kbd>
</p>
<h2>Titular de segundo nivel</h2>
<h3>Titular de tercer nivel</h3>
  <p>Y así hasta h6</p>
<h2>Listas</h2>
 <h3>Una lista:</h3>
  <ul>
   <li> uno,</li>
   <li> dos y</li>
   <li> tres</li>
  </ul>
 <h3>Una lista numerada:</h3>
  <ol>
   <li> uno,</li>
   <li> dos y</li>
   <li> tres</li>
  </ol>
<h2>Tipografía y colores</h2>
 <ul>
```

```
  <li style="color:red"><i>Cursiva y rojo</i>
  <li style="color:blue"><b>Negrita y azul</b>
 </ul>
<p><b  style="font-size:160%">Dos enlaces:</b>
 <a href="http://validator.w3.org/check/referer">
  Pincha aquí para comprobar la sintaxis de este documento</a>.
O bien
 <a href="http://html5.validator.nu/?doc=http://www.lab.dit.upm.es
                      /ftel/demos-tema3/intro-html5.html">aquí</a>
</p>
<p>HTML5 está evolucionando. Los servicios de validación y/o este
documento pueden no estar actualizados.</p>
 <table role="presentation">
 <tr>
    <th style="font-size:150%">Una imagen con enlace:</th>
    <th style="font-size:150%">Y un vídeo:</th></tr>
  <tr>
  <td><a href="http://www.lab.dit.upm.es/ftel/">
      <img src="./ftel-dit-upm.png" width="140" alt="Logo de FTEL"></a>
      <br><em style="font-size:80%">
           Pinchando en la imagen<br>vas a la página de FTEL</em>
  </td>
  <td>
      <video width="200" controls autoplay>
      <source src="big_buck_bunny.mp4" type="video/mp4">
      <source src="big_buck_bunny.webm" type="video/webm">
      <source src="big_buck_bunny.ogv" type="video/ogg">
       Este navegador no conoce el elemento vídeo, nuevo en HTML5.
      </video><br>
      <a href="http://www.bigbuckbunny.org">
      <em style="font-size:80%">
        (c) Copyright 2008, Blender Foundation</em></a>
  </td>
 </tr>
</table>
</body>
</html>
```

Al recibir esta secuencia de caracteres, el navegador los interpreta y los presenta como se reproduce en la figura 5.7, o como puede usted ver en la dirección
`http://www.lab.dit.upm.es/ftel/demos-tema3/intro-html5.html`.

Si observa con atención la correspondencia entre las declaraciones del código fuente y el resultado entenderá la mayor parte de los convenios sintácticos para los elementos que se han incluido (que son solo unos pocos: en el borrador actual de la especificación de HTML5 hay más de cien). Algunos detalles a destacar son:

- La primera línea, `<!DOCTYPE html>`, no es necesaria para el navegador, pero sí para que el validador al que se enlaza al final acepte el documento.
- Las etiquetas pueden incluir valores de ciertos atributos del contenido. Por ejemplo, en la segunda línea: se especifica un atributo, «`lang`», para todo el documento, cuyo valor es «`es-ES`» (español de España). Esta información tampoco es necesaria a efectos de presentación, pero el alcance del

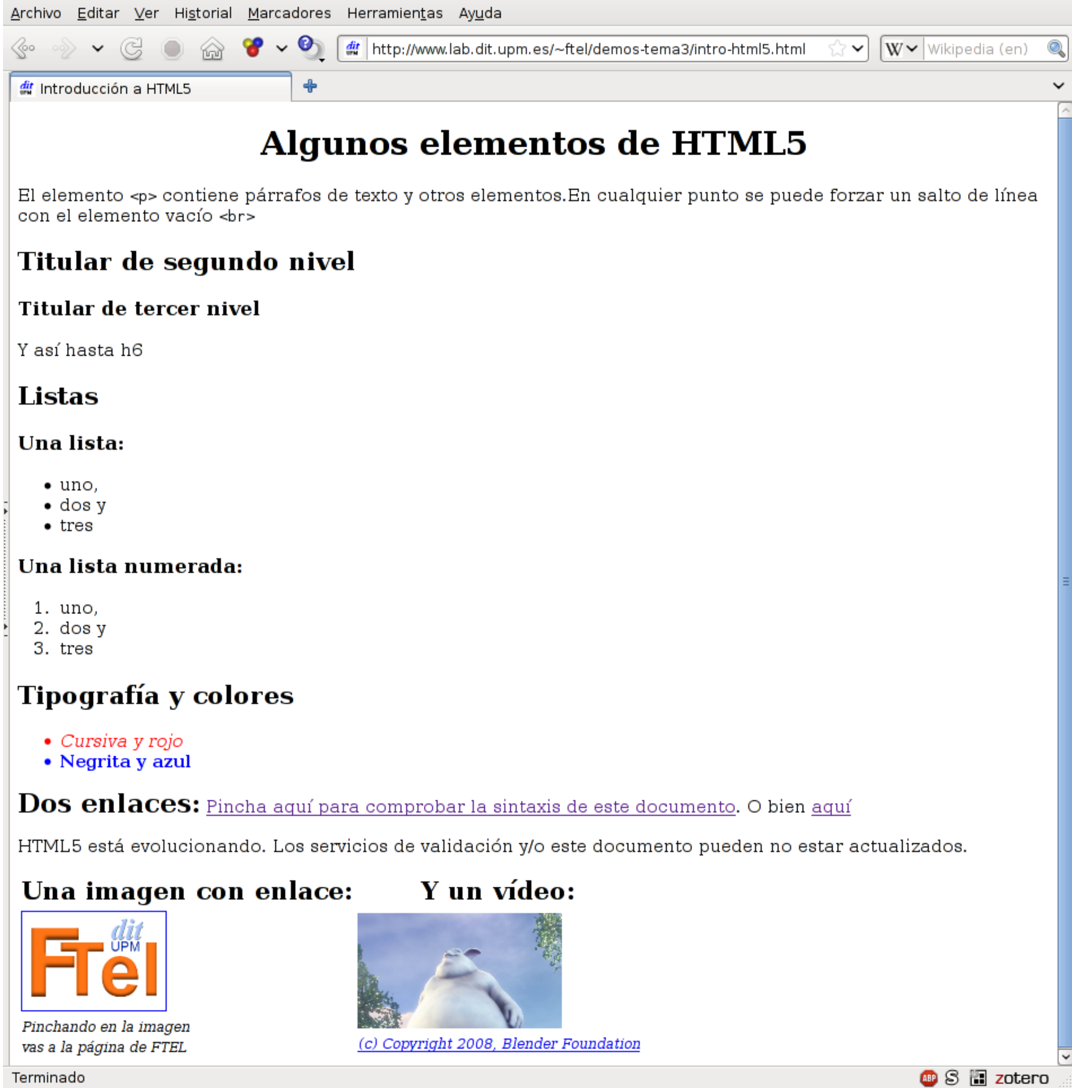


Figura 5.7 La página de muestra en el navegador

lenguaje va más allá de la mera presentación de páginas web estáticas. En otra aplicación se podría codificar un mensaje para un agente que, con este dato, podría saber cómo interpretar el contenido y cómo responder.

- En la cabecera, aparte del título, hay un elemento vacío «<meta>» que informa al intérprete de la codificación de caracteres. Este dato sí es importante para el navegador: si éste está configurado para interpretar por defecto los caracteres como ISO Latin 1 o cualquier otro código, el dato es necesario para que los caracteres no ASCII se presenten adecuadamente.
- En algunas marcas puede usted ver que hay atributos de estilo («style="..."») que especifican detalles sobre la forma de la presentación. Siguen la sintaxis de otro lenguaje, CSS (Cascading Style Sheets). Aquí se han incluido en el mismo documento HTML5, pero en general es mejor incluirlos en un documento aparte.
- Observe el elemento <kbd><p></kbd>: la etiqueta «<kbd>» indica que el texto contenido

debe presentarse con tipografía mecanográfica *(keyboard)*, y este contenido debe verse como «<p>». Ahora bien, los símbolos «<» y «>» tienen un significado especial en el lenguaje y no se pueden escribir directamente. Para que el navegador entienda que tiene que presentar «<» y no interpretarlo como la apertura de un nueva marca se utiliza una **referencia a entidad de carácter** (o, simplemente, «entidad»): «<» se refiere al carácter «<» («lt» es por «less than», menor que). Otras entidades para caracteres reservados son «>» (>: greater than), «&» (&: ampersand) y «"» (": quotation).

- En la parte final se presentan una imagen y un vídeo acompañados cada uno de una leyenda, y para que salgan en dos columnas se ha recurrido al elemento <table> (este elemento es para presentar datos en forma tabular; para presentaciones de otros elementos, como aquí, se aconseja hacerlo con CSS, pero así también funciona y es menos complicado). Hijos de <table>son <tr> (*table row*: fila), <th> (*table header*, cabecera) y <td> (*table data*).
- El elemento «<video>» es nuevo en HTML5. Si el navegador no está preparado, presenta el texto alternativo que se incluye en el contenido. Dentro de él hay dos elementos «<source>» que hacen referencia al mismo vídeo en tres formatos diferentes, para que pueda reproducirse en distintos navegadores (Tema 2, final del apartado 5.4).

XML

A diferencia de HTML, cuyo objetivo es la *presentación* de datos en un navegador, XML (eXtensible Markup Language) está diseñado para la *representación* de datos a efectos de almacenamiento y comunicación entre agentes. Pero XML es más que un lenguaje: es un **metalenguaje**. Quiere esto decir que sirve para definir lenguajes concretos diferentes. Su sintaxis es muy parecida a la de HTML. Por ejemplo, éste es un documento XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE asignaturas2010 SYSTEM "asignaturas.dtd">
<asignaturas2010>
<!-- Asignaturas del plan de estudios 2010 -->
  <asignatura codigo="95000005" acronimo="FTEL">
    <nombre>Fundamentos de los Sistemas Telemáticos</nombre>
    <creditos>4.5</creditos>
    <semestre S="1" />
    <tipo T="obligatoria" />
    <guia>
       <url href="http//www.lab.dit.upm.es/ftel" />
       <url href="http://www.etsit.upm.es/estudios/
       graduado-en-ingenieria-de-tecnologias-y-servicios-de-telecomunicacion/
       plan-de-estudios/listado-de-asignaturas.html" />
    </guia>
  </asignatura>
  <asignatura codigo="95000019">
    <nombre>Métodos Matemáticos</nombre>
    <creditos>4.5</creditos>
    <semestre S="2" />
    <tipo T="basica" />
  </asignatura>
<!--otras asignaturas-->
</asignaturas2010>
```

Independientemente de lo que significa su contenido, éste es un documento XML **bien formado**. Eso significa que está construido respetando las reglas sintácticas de XML. Algunas de ellas, expresadas informalmente, son:

- Debe tener al principio un *prólogo* que empieza con la *declaración xml*: versión y codificación de caracteres (si se omite esta última, por defecto es UTF-8, de manera que en el documento anterior este dato es redundante), y sigue con el *tipo de documento*, que es una referencia a otro documento en el que se define el lenguaje concreto para interpretar el contenido.
- Debe tener un **elemento raíz** (en este caso, «`asignaturas2010`»), que es el «padre» de otros elementos. (En HTML, el elemento raíz es «`html`» o «`HTML`»).
- Todos los elementos con contenido tienen que tener una etiqueta de cierre, y deben estar correctamente anidados (en HTML, los navegadores admiten excepciones a esta regla).
- Los elementos vacíos deben terminar con «`/>`» (en HTML pueden terminar así o con «`>`», salvo en la versión «XHTML», donde debe ser «`/>`»).
- Los elementos pueden tener atributos, y los valores de éstos debe ir entrecomillados, con comillas dobles o simples (como en HTML).
- Otra diferencia con HTML es que los nombres de los elementos y los atributos son sensibles a la caja: `<curso>` y `<Curso>` son nombres distintos.

Ahora bien, una cosa es que el documento esté «bien formado» y otra que el agente que lo lea lo entienda. Usted lo ha entendido, pero para que un programa lo interprete y pueda hacer algo con él (por ejemplo, construir una página web, o generar un pdf, o integrarlo con otros documentos) es preciso definir el lenguaje concreto que estamos utilizando, en el que hemos inventado elementos y nombres como «asignatura», «tipo», etc. Esta es la diferencia básica entre XML y HTML: en éste, los nombres de elementos y atributos («html», «video», «width», etc.) tienen un significado predefinido para el intérprete incluido en el navegador. En XML inventamos nombres adecuados para cada aplicación, definiendo un lenguaje concreto. Por eso decimos que XML es un «metalenguaje».

El lenguaje concreto se define mediante reglas sintácticas específicas para ese lenguaje. Esto se puede hacer de dos formas: mediante una **DTD** (Document Type Definition) o mediante un «esquema» **XSD** (XML Schema Definition). Veamos las ideas básicas de la primera. Una DTD para el lenguaje anterior sería:

```
1 <!ELEMENT asignaturas2010 (asignatura+)>
2 <!ELEMENT asignatura (nombre, creditos, semestre, tipo, guia?)>
3 <!ELEMENT nombre (#PCDATA)>
4 <!ELEMENT creditos (#PCDATA)>
5 <!ELEMENT semestre EMPTY>
6 <!ELEMENT tipo EMPTY>
7 <!ELEMENT guia (url+)>
8 <!ELEMENT url EMPTY>
9 <!ATTLIST asignatura codigo CDATA #REQUIRED>
10 <!ATTLIST asignatura acronimo CDATA #IMPLIED>
11 <!ATTLIST semestre S (1 | 2) #REQUIRED>
12 <!ATTLIST tipo T (basica | obligatoria | optativa) #REQUIRED>
13 <!ATTLIST url href CDATA #REQUIRED>
```

La numeración de las líneas es solo para identificarlas, no forma parte de la DTD.

Primero se especifica cómo debe ser el contenido de cada elemento:

- La línea 1 dice que el elemento raíz, `asignaturas2010`, debe tener como hijos uno o más (eso significa «+») elementos `asignatura`.
- La línea 2, que cada elemento `asignatura` debe contener un elemento `nombre`, uno `creditos`, uno `semestre`, uno `tipo` y uno o ninguno (eso significa «?») `guia`.
- Las líneas 3 y 4, que el contenido de esos elementos es «PCDATA» (Parsed Character Data): texto que no contiene marcas.
- Según las líneas 5, 6 y 8, los elementos `semestre`, `tipo` y `url` son vacíos.
- La línea 7 dice que `guia` (si existe, de acuerdo con la línea 2) debe tener uno o más elementos hijos `url`.

Después están las especificaciones de valores de atributos:

- La línea 9 dice que el elemento `asignatura` tiene que llevar obligatoriamente («`#REQUIRED`») un atributo de nombre `codigo` con valor `CDATA` (Character Data) (texto que no se analiza).
- Sin embargo, el atributo `acronimo` (línea 10) es opcional (`#IMPLIED`).
- La línea 11 dice que el elemento `semestre` tiene necesariamente el atributo `S` cuyo valor puede ser `"1"` o `"2"`
- Similar es la línea 12: el atributo `T` de `semestre` debe tener uno de los tres valores indicados.
- Finalmente, según la línea 13, el atributo `href` de `url`, con valor `CDATA`, es obligatorio.

Si analiza usted el documento XML verá que respeta estas reglas. Y eso es lo que hace un analizador *(parser)* de XML: **validar** un documento XML conforme a un DTD (o a un esquema) y entregar un árbol sintáctico a la aplicación.

Estas definiciones de lenguaje se incluyen en un documento aparte (según indica la segunda línea del documento XML estarían en el fichero `asignaturas.dtd` del mismo directorio del documento, pero podrían estar en otra URL), o bien en el prólogo del mismo documento XML de la siguiente forma:

```
<?xml version="1.0" ?>
<!DOCTYPE asignaturas2010 [
<!-- definiciones de elementos y atributos -->
]>
```

Resumiendo, un documento XML está **bien formado** si su sintaxis respeta las reglas generales del metalenguaje XML. Y es un documento **válido** si, además, respeta las reglas del lenguaje concreto definido por una DTD, o por un esquema XSD (no entramos ya en explicar XSD, que es una forma más moderna de definir el lenguaje).

Después de validar un documento, el intérprete ha de ejecutarlo (figura 5.6). Esto ya depende de la aplicación. En el ejemplo anterior, el mismo documento XML podría servir para generar una página web, para generar un pdf, para generar un mensaje de correo, etc. Para esto hay procesadores que, basándose en reglas que se definen en XSLT (Extensible Stylesheet Language Transformations), obtienen los resultados deseados.

La flexibilidad y la universalidad de XML lo han convertido en la lengua franca de Internet. Cada vez más lenguajes para aplicaciones diversas se basan en XML. De uno de ellos vimos un ejemplo en el apartado 5.4 del Tema 2: SVG, para la representación de imágenes vectoriales. La DTD de SVG (bastante más compleja que la anterior) está publicada en `http://www.w3.org/TR/SVG/svgdtd.html`.

5.5. Lenguajes de *script*

Un lenguaje de *script*[3], o de *scripting*, es un lenguaje interpretado para programas que controlan el funcionamiento de otros programas. De este tipo, son, por ejemplo, los lenguajes de los intérpretes de órdenes que vimos en el Tema 1, que controlan las funciones del sistema operativo. Otro tipo es el de los diseñados para controlar las aplicaciones web, ya sea ejecutándose en el cliente (el navegador), como JavaScript, o en el servidor, como PHP o ASP.

JavaScript

No hay que confundir JavaScript (que, para ser más precisos, deberíamos llamar «ECMAScript», nombre del estándar) con Java, un lenguaje compilado que estudirá usted en la asignatura «Programación».

El uso principal de JavaScript es la inclusión de funciones en las páginas web para conseguir efectos dinámicos e interactivos añadidos a los efectos estáticos de HTML. Por eso, se habla de «DHTML» (Dynamic HTML). Efectos como mostrar una alerta cuando el cursor pasa por una zona, validar valores de un formulario antes de enviarlo al servidor, o cambiar una imagen por otra cuando el cursor pasa por ella.

Se pueden insertar fragmentos de JavaScript en el mismo documento HTML. Por ejemplo:

```
<html>
<head>
 <meta charset="UTF-8">
 <title>Introducción a JavaScript (1)</title>
</head>
<body>
<h1>Fechas y datos del documento</h1>
<script type="text/javascript">
<!--
 document.write("Hora local: ");
 document.write(Date());
 document.write("<br>URL: "+document.location+"<br>");
 document.write("Modificado el "+document.lastModified);
// -->
</script>
</body>
</html>
```

El resultado en el navegador puede usted comprobarlo accediendo a la dirección `http://www.lab.dit.upm.es/ftel/demos-tema3/intro-JS1.html`: escribe la fecha y hora actual, la URL del documento y la fecha y hora de su última modificación.

Una breve explicación del código:

- «`script`» es el nombre del elemento cuyo contenido son las sentencias del lenguaje. El valor del atributo indica el lenguaje que el navegador debe interpretar.
- Las marcas de comentario «`<!--...-->`» son para los navegadores que no aceptan JavaScript. Al final, «`//`» es la indicación de comentario de JavaScript, para que el intérprete omita «`-->`».

[3] Se puede traducir *script* por «guión», pero lo habitual es conservar el término inglés.

- «`write`», «`location`» y «`lastModified`» son procedimientos incorporados en el lenguaje que se aplican al objeto «`document`» (el documento).
- «`Date()`» es una función de JavaScript que devuelve los datos que pueden verse en el resultado. Se pueden obtener los datos en español, pero escribiendo una función adecuada; `Date()` está ya incorporada «de serie».

Ahora bien, lo anterior no tiene nada de «interactivo»: las sentencias se ejecutan en el momento de cargarse la página en el navegador, y el resultado es estático. Sería interactivo si, por ejemplo, se mostrasen los resultados al pulsar en alguna parte de la página. Esto puede hacerse poniendo el *script* como una función en la cabecera, no en el cuerpo, y llamándola cuando aparezca el *evento* de pulsar:

```
<html>
<head>
 <meta charset="UTF-8">
 <title>Introducción a JavaScript (2)</title>
 <script type="text/javascript">
   function Datos() {
   document.getElementById("fecha").innerHTML=
           "Fecha y hora actuales: "+Date();
   document.getElementById("url").innerHTML="URL: "+document.location;
   document.getElementById("mod").innerHTML=
           "Última modificación: "+document.lastModified;
   }
 </script>
</head>
<body>
<h1>Fecha y datos del documento</h1>
<p>Pincha en el botón para saber la hora actual y datos de esta página</p>
<button type="button" onClick="Datos()">Escribe datos</button>
<p id="fecha">Aquí verás la fecha y hora actual</p>
<p id="url">Aquí, la URL de este documento</p>
<p id="mod">Aquí, cuándo fue modificado</p>
</body>
</html>
```

Los datos aparecen al pulsar en el botón, como puede usted comprobar en la dirección
`http://www.lab.dit.upm.es/ftel/demos-tema3/intro-JS2.html`

Esto requiere alguna explicación más. Empezando por el código HTML (el que está dentro de `<body>`):

- El elemento `<button>` crea un botón. Aparte del atributo `type`, cuyo valor indica que es un simple botón (hay otros tipos), hay un atributo `onClick`, cuyo valor es el nombre de una función JavaScript que se ejecuta cuando se produce el evento de pulsar en el botón.
- Los tres elementos `<p>` que siguen tienen un atributo `id` cuyo valor sirve para identificarlos desde la función JavaScript.

En cuanto a la función `Datos()` de la cabecera, que solamente se ejecuta cuando se pulsa el botón:

- Contrariamente al ejemplo anterior, no utiliza `document.write`, porque esto sustituiría toda la página por una nueva, y lo que queremos es escribir en ciertos lugares sin sustituir la página.
- La función `getElementById("nombre")`, aplicada al objeto `document`, devuelve el elemento identificado por `"nombre"`, y en este elemento el procedimiento `innerHTML` sustituye su contenido HTML por el que le digamos. Así es como cambiamos los contenidos de texto de los tres `<p>`.

Además de `onClick` hay otros atributos cuyo valor indica la función que se «dispara» al producirse el evento: `onMouseOver`, `onMouseOut`, `onKeyPress`, etc. Unido a la facilidad de escribir funciones en general (no como las utilizadas en el ejemplo, que son llamadas a funciones ya predefinidas) permite diseñar páginas con efectos muy variados.

Como último ejemplo, ya que conocemos la función del máximo común divisor en C (apartado 5.3), y en JavaScript es prácticamente idéntica, veamos cómo se puede integrar en una página HTML:

```
<html>
<head>
 <meta charset="UTF-8">
 <title>Introducción a JavaScript (2)</title>
 <script type="text/javascript">
  function MCD(x,y)          // Calcula el máximo común divisor
    while (x != y) {         // de x e y mediante el
     if (x > y) x = x - y;   // algoritmo de Euclides
     else y = y - x;
    }
    return x;                // Devuelve el resultado en x
  }
  function numero(elemento) {  //Traduce los caracteres a número entero
    return (parseInt(elemento.value));
  }
  function Calcular(x,y) {  //Llama a MCD y escribe el resultado
    document.getElementById("resultado").innerHTML=
       "El máximo común divisor de "+x+" y "+y+" es "+MCD(x,y);
  }
 </script>
</head>
<body>
 <form>
   Escribe un número:
   <input type="text" value=""  size="10" onChange="x = numero(this)">
   <br>Escribe otro:
   <input type="text" value="" size="10" onChange="y = numero(this)">
   <br><input type="button" value="Calcular MCD" onClick="Calcular(x,y)">
 </form>
 <p id="resultado"></p>
</body>
</html>
```

En el *script* de la cabecera hay ahora tres funciones. Normalmente, en lugar de insertar el código JavaScript (que puede contener muchas funciones) en la página HTML se incluye en un fichero aparte con la extensión `.js`, y en la cabecera se enlaza con
`<script type="text/javascript" src="fichero.js"></script>`.

Pero veamos antes algunos elementos HTML del cuerpo que no habían aparecido en los ejemplos anteriores:

- `<form>` se utiliza normalmente para rellenar y enviar datos al servidor; aquí solo para recoger dos datos y llamar a un función JavaScript. En sí mismo, no es un elemento «visible»: sirve para contener otros elementos.

- `<input>` es para recoger datos del usuario. Puede adoptar distintas formas dependiendo del valor del atributo `type`. En los dos primeros casos del ejemplo, `"text"` presenta un cuadro para que se introduzca un texto con un máximo de 10 caracteres.
- El valor del atributo `onChange` se ejecuta cuando se cambia el contenido del texto: se asigna a la variable `x` el resultado de la función `numero(this)`. «`this`» es el propio elemento `<input>`.
- En la siguiente línea, el tipo es `button`. Al pulsarlo se ejecuta `Calcular(x,y)`.

Las funciones de la cabecera son:

- `MCD(x,y)` es la misma que ya hemos comentado tanto en ensamblador (apartado 4.5) como en C (apartado 5.3) para calcular el máximo común divisor mediante el algoritmo de Euclides.
- `numero(elemento)` recibe el elemento (los dos `<input>` que la llaman con `this`) y devuelve el número que se ha escrito. Para ello, hace uso de la función incorporada en el lenguaje `parseInt`. En efecto, con `elemento.value` lo que se obtiene es la cadena de caracteres. `parseInt` convierte esa cadena en un número entero. Para hacer las cosas bien se debería comprobar previamente que todos los caracteres introducidos son numéricos (hay un comentario sobre esto en el código fuente de `http://www.lab.dit.upm.es/ftel/demos-tema3/intro-JS3.html`). Si se introducen caracteres no numéricos el resultado es imprevisible. Puede usted comprobarlo, bajo su responsabilidad: es posible que el navegador, o todo el sistema, quede bloqueado por falta de memoria o uso excesivo de la UCP[4].
- `Calcular(x,y)` simplemente le pasa a MCD los valores numéricos obtenidos y luego sustituye la cadena vacía que habíamos puesto (`<p id="resultado"></p>`) por el resultado.

AJAX

Hay muchas más posibilidades para conseguir efectos dinámicos con JavaScript. Esta introducción debe ser suficiente para que pueda usted profundizar en su estudio mediante la consulta de las fuentes que se citan en la bibliografía.

Para terminar, y sin entrar ya en detalles de implementación, seguramente le interesará saber cómo funcionan, por ejemplo, las «sugerencias» de Google. Habrá usted observado que a medida que se escriben caracteres en el cuadro de búsqueda van apareciendo, debajo, posibles continuaciones. La implementación de interfaces de usuario «dialogantes» como ésta se hace con «AJAX» (Asynchronous JavaScript and XML). Se trata de una combinación de técnicas de JavaScript y de transmisión de datos mediante los protocolos de la web con la que se consigue una interacción dinámica entre las acciones del usuario y las respuestas del servidor.

La figura 5.8 resume el principio. Ante un evento en el cliente, es decir, el navegador (como pulsar una tecla) se crea, mediante JavaScript, un objeto que se llama HttpRequest. En la asignatura «Programación» entenderá usted qué es eso de un «objeto». Digamos, simplificadamente, que es un conjunto estructurado de datos junto con unas funciones asociadas (abrir, enviar, etc.). El objeto se envía al servidor, que, tras analizarlo, elabora la respuesta adecuada y la remite al cliente. A diferencia de una interacción web «clásica», al recibir la respuesta el cliente no abre una página nueva: ejecuta funciones de JavaScript para modificar lo necesario en la página actual. Los datos se transmiten codificados en XML o en una alternativa más ligera como JSON (JavaScript Object Notation). Y en el caso del buscador de Google esta secuencia se produce cada vez que se pulsa una tecla.

[4] De aquí se extrae una lección interesante: no se fíe de cualquier página que utilice JavaScript, póngale a su navegador una extensión para bloquearlo y admita únicamente las páginas conocidas.

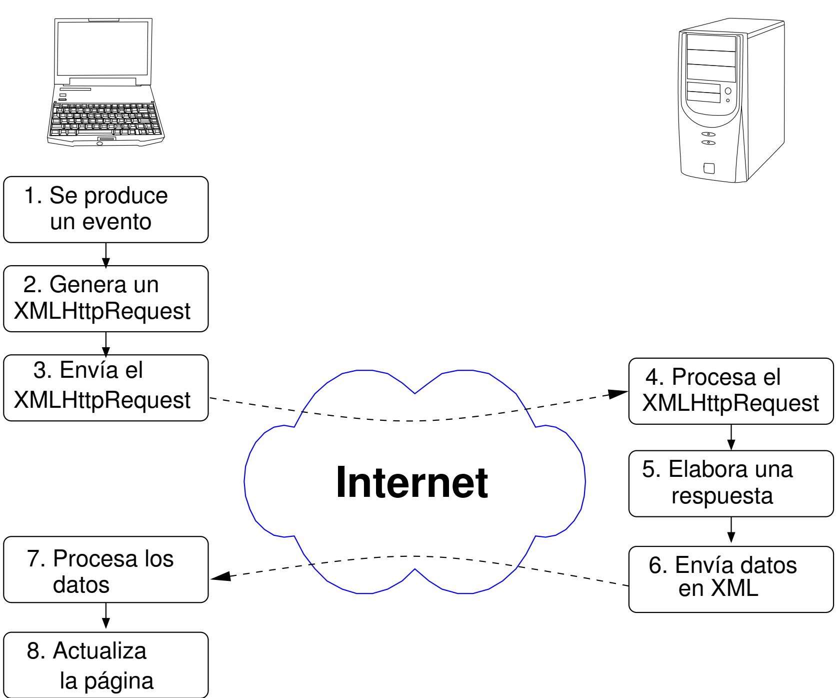


Figura 5.8 Procesos en AJAX

Apéndice A

El simulador ARMSim# y otras herramientas

ARMSim# es una aplicación de escritorio para un entorno Windows desarrollada en el Departamento de Computer Science de la University of Victoria (British Columbia, Canadá). Incluye un ensamblador, un montador (linker) y un simulador del procesador ARM7TDMI. Cuando se carga en él un programa fuente (o varios), ARMSim# automáticamente lo ensambla y lo monta. Después, mediante menús, se puede simular la ejecución. Es de distribución libre para uso académico y se puede descargar de `http://armsim.cs.uvic.ca/`

Lo que sigue es un resumen de los documentos que se encuentran en `http://armsim.cs.uvic.ca/Documentation.html`.

A.1. Instalación

Instalación en Windows

Se puede instalar en cualquier versión a partir de Windows 98. Requiere tener previamente instalado el «framework» .NET, versión 3.0 o posterior, que se puede obtener libremente del sitio web de Microsoft.

En la página de descargas de ARMSim# se encuentra un enlace para descargar el programa instalador (y también hay un enlace para descargar .NET de Microsoft). La versión actual (mayo 2010) es `ARMSim1.91Installer.msi`. Al ejecutarlo se crea un subdirectorio llamado «University of Victoria» dentro de «Archivos de Programa».

Instalación en UN*X

«Mono» es un proyecto de código abierto que implementa .NET. Permite ejecutar el programa ARMSim# en un sistema operativo de tipo UN*X: Linux, *BSD, Mac OS X, etc. Muchas distribuciones (por ejemplo, Debian y Ubuntu) contienen ya todo el entorno de Mono. Si no se dispone de él, se puede descargar de aquí: `http://www.go-mono.com/mono-downloads/download.html`

Estas son las instrucciones para instalar el simulador:

1. En la misma página de descargas de ARMSim# hay un enlace a la versión para Mono. Es un fichero comprimido con el nombre ARMSim-191-MAC.zip.

2. En el directorio $HOME (en Linux y *BSD, /home/<usuario>/), crear un directorio de nombre dotnet y mover a él el fichero descargado previamente.

3. Descomprimir: unzip ARMSim-191-MAC.zip. Deberán obtenerse los ficheros:

```
ARMPluginInterfaces.dll
ARMSim.exe
ARMSim.exe.config
ARMSim.Plugins.EmbestBoardPlugin.dll
ARMSim.Plugins.UIControls.dll
ARMSimWindowManager.dll
DockingWindows.dll
DotNetMagic2005.dll
StaticWindows.dll
```

4. Utilizando un editor de textos, modificar el fichero ARMSim.exe.config para que la línea

```
<!-- <add key="DockingWindowStyle" value="StaticWindows"></add> -->
```

   pase a ser:

```
<add key="DockingWindowStyle" value="StaticWindows"></add>
```

5. Con el editor, crear un fichero de nombre ARMSim.sh (por ejemplo) con este contenido:

```
#!/bin/sh
mono $HOME/dotnet/ARMSim.exe
```

6. Hacerlo ejecutable (chmod +x ARMSim.sh) y colocarlo en un directorio que esté en el PATH (por ejemplo, en $HOME/bin). Opcionalmente, copiarlo en el escritorio.

El programa ARMSim# se ejecutará desde un terminal con el comando ARMSim.sh (o pinchando en el icono del escritorio).

A.2. Uso

La interfaz gráfica es bastante intuitiva y amigable, y sus posibilidades se van descubriendo con el uso y la experimentación. Para cargar un programa, File > Load. Puede ser un programa fuente (con la extensión «.s») o un programa objeto («.o») procedente, por ejemplo, del ensamblador GNU (apartado A.3). Se muestran en varias ventanas las vistas que se hayan habilitado en «View». Todas, salvo la central (la del código) se pueden separar y colocar en otro lugar («docking windows»).

(Nota para usuarios de UN*X: con las versiones actuales de Mono no funcionan las «docking windows»; hay que inhabilitar esta característica haciendo las ventanas estáticas, como se ha indicado en el paso 4 de las instrucciones de instalación).

Vistas

Las vistas muestran la salida del simulador y el contenido de los registros y la memoria. Se pueden seleccionar desde el menú «View».

- Code View (en el centro): Instrucciones del programa en hexadecimal y en ensamblador. Siempre visible, no puede cerrarse. Por defecto, el programa se carga a partir de la dirección 0x00001000, pero puede cambiarse en `File > Preferences > Main Memory`.
- Registers View (a la izquierda): Contenido de los dieciséis registros y del CPSR.
- Output View - Console (abajo): Mensajes de error.
- Output View - Stdin/Stdout/Stderr (abajo): texto enviado a la salida estándar.
- Memory View (abajo): Contenido de la memoria principal. Por defecto, está deshabilitada; para verla hay que seleccionarla en el menú «View». Se presenta solo un fragmento de unos cientos de bytes, a partir de una dirección que se indica en el cuadro de texto de la izquierda. En la tabla resultante, la primera columna son direcciones (en hexadecimal) y en cada línea aparecen los contenidos (también en hexadecimal) de las direcciones sucesivas. Se pueden ver los contenidos de bytes, de medias palabras y de palabras (seleccionando, a la derecha de la ventana, «8 bits», «16 bits» o «32 bits», respectivamente). En la presentación por bytes se muestra también su interpretación como códigos ASCII.
- Stack View (a la derecha): Contenido de la pila del sistema. La palabra que está en la cima aparece resaltada. El puntero de pila se inicializa con el valor 0x00005400. El simulador reserva 32 KiB para la pila, pero también puede cambiarse en `File > Preferences > Main Memory`.
- Watch View (abajo): valores de las variables que ha añadido el usuario a una lista para vigilar durante la ejecución
- Cache Views (abajo): Contenido de la cache L1.
- Board Controls View (abajo): interfaces de usuario de plugins (si no se cargó ninguno al empezar, no está visible).

Botones de la barra de herramientas

Debajo del menú principal hay una barra con seis botones:

- Step Into: hace que el simulador ejecute la instrucción resaltada y resalte la siguiente a ejecutar. Si es una llamada a subprograma (`bl` o `bx`), la siguiente será la primera del subprograma.
- Step Over: hace que el simulador ejecute la instrucción resaltada en un subprograma y resalte la siguiente a ejecutar. Si es una llamada a subprograma (`bl` o `bx`), se ejecuta hasta el retorno del subprograma.
- Stop: detiene la ejecución.
- Continue (o Run): ejecuta el programa hasta encontrar un «breakpoint», una instrucción `swi 0x11` (fin de ejecución) o un error de ejecución.
- Restart: ejecuta el programa desde el principio.
- Reload: carga un fichero con una nueva versión del programa y lo ejecuta.

Puntos de parada («breakpoints»)

Para poner un *breakpoint* en una instrucción, mover el cursor hasta ella y hacer doble click. La ejecución se detendrá justo antes de ejecutarla. Para seguir hasta el siguiente *breakpoint*, botón «run». Para quitar el *breakpoint*, de nuevo doble click.

Códigos para la interrupción de programa

El simulador interpreta la instruccción `SWI` leyendo el código (número) que la acompaña. En la tabla A.1 están las acciones que corresponden a los cuatro más básicos. El simulador interpreta otros códigos para operaciones como abrir y cerrar ficheros, leer o escribir en ellos, etc., que se pueden consultar en la documentación. «Stdout» significa la «salida estándar»; en el simulador se presenta en la ventana inferior de resultados.

Código	Descripción	Entrada	Salida
`swi 0x00`	Escribe carácter en Stdout	r0: el carácter	
`swi 0x02`	Escribe cadena en Stdout	r0: dirección de una cadena ASCII terminada con el carácter NUL (0x00)	
`swi 0x07`	Dispara una ventana emergente con un mensaje y espera a que se introduzca un entero *Está documentada pero no funciona*	r0: dirección de la cadena con el mensaje	r0: el entero leído
`swi 0x11`	Detiene la ejecución		

Tabla A.1 Códigos de SWI

A.3. Otras herramientas

Este apartado contiene algunas indicaciones por si está usted interesado en avanzar un poco más en la programación en bajo nivel de ARM. Ante todo, recordar lo dicho en el apartado 4.10: ésta es una actividad instructiva, pero no pretenda programar aplicaciones en lenguaje ensamblador: espere a la asignatura «Análisis y diseño de software».

Veamos algunas herramientas gratuitas que sirven, en principio, para desarrollo cruzado, es decir, que se ejecutan en un ordenador que normalmente no está basado en ARM y generan código para ARM. Luego se puede simular la ejecución con un programa de emulación, como «qemu» (`http:qemu.org`), o con ARMSim#, o con el simulador incluido en la misma herramienta. Y también, como vimos en el apartado 4.10, se puede cargar el código ejecutable en la memoria de un dispositivo con procesador ARM.

Hay algunos ensambladores disponibles en la red que puede usted probar. Por ejemplo, FASM (flat assembler), un ensamblador muy eficiente diseñado en principio para las arquitecturas x86 y x86-64 que tiene una versión para ARM, y que funciona en Windows y en Linux, FASMARM:
`http://arm.flatassembler.net/`.

Pero lo más práctico es hacer uso de una «cadena de herramientas» *(toolchain)*: un conjunto de procesadores software que normalmente se aplican uno tras otro: ensamblador o compilador, montador, emulador, depurador, simulador, etc. Las hay basadas en el ensamblador GNU y en el ensamblador propio de ARM.

Herramientas GNU

- El proyecto de código abierto GNUARM («GNU ARM toolchain for Cygwin, Linux and MacOS», `http://www.gnuarm.com`) no se ha actualizado desde 2006, pero aún tiene miembros activos. Los programas pueden ejecutarse en UN*X y en Windows (Cygwin, `http://www.cygwin.com/`, es un entorno para tener la funcionalidad de Linux sobre Windows).
- Hay varios proyectos para incluir la cadena GNU en distribuciones de Linux. Por ejemplo, en Ubuntu (`https://wiki.ubuntu.com/EmbeddedUbuntu`) y en Debian (`http://www.emdebian.org/`). Esta última es la que se ha usado para generar los listados de este Tema.
- Actualmente, el software más interesante no es totalmente libre, pero sí gratuito. Es el desarrollado por la empresa «CodeSourcery», adquirida en 2011 por «Mentor Graphics», que le ha dado el nombre «Sourcery CodeBench». Tiene varias versiones comerciales con distintas opciones de entornos de desarrollo y soporte. La «lite edition» de la cadena GNU solamente contiene las herramientas de línea de órdenes, pero su descarga es gratuita. Está disponible (previo registro) para Linux y para Windows en:
  `http://www.mentor.com/embedded-software/sourcery-tools/`
  `sourcery-codebench/editions/lite-edition/`

  La documentación que se obtiene con la descarga es muy clara y completa.

Los procesadores de la cadena GNU se ejecutan desde la línea de órdenes. Todos tienen muchas opciones, y todos tienen página de manual. Resumimos los principales, dando los nombres básicos, que habrá que sustituir dependiendo de la cadena que se esté utilizando. Por ejemplo, `as` es el ensamblador, y para ver todas las opciones consultaremos `man as`. Pero «`as`» a secas ejecuta el ensamblador «nativo»: si estamos en un ordenador con arquitectura x86, el ensamblador para esa arquitectura. El ensamblador cruzado para ARM tiene un prefijo que depende de la versión de la cadena GNU que se esté utilizando. Para la de Debian es `arm-linux-gnueabi-as`, y para la «Sourcery», `arm-none-linux-gnueabi-as`. Y lo mismo para los otros procesadores (`gcc`, `ld`, etc.).

- `as` es el ensamblador. Si el programa fuente está en el fichero `fich.s`, con la orden `as -o fich.o <otras opciones> fich.s` se genera el código objeto y se guarda en `fich.o` (sin la opción `-o`, se guarda en `a.out`).
  Algunas otras opciones:
  - `-EB` genera código con convenio extremista mayor (por defecto es menor). Útil si solo se quiere un listado que sea más legible, como se explica en el último párrafo del apartado 4.1.
  - `-al` genera un listado del programa fuente y el resultado, como los del capítulo 4.
  - `-as` genera un listado con la tabla de símbolos del programa fuente. Se puede combinar con el anterior: `-als`
- `gcc` es el compilador del lenguaje C. Normalmente realiza todos los procesos de compilación, ensamblaje y montaje. Si los programas fuente están en los ficheros `fich1.c`, `fich2.c`, etc., la orden
  `gcc -o fich <otras opciones> fich1.c fich2.c...`
  genera el código objeto y lo guarda en `fich`.

Algunas otras opciones:

`-S` solamente hace la compilación y genera ficheros en ensamblador con los nombres `fich1.s`, `fich2.s`, etc. (si no se ha puesto la opción `-o`).

`-O`, `-O1`, `-O2`, `-O3` aplican varias operaciones de optimización para la arquitectura (en nuestro caso, para ARM).

- `ld` es el montador. `ld -o fich <otras opciones> fich1.o fich2.o...` genera un código objeto ejecutable.
  Algunas otras opciones:

  `-Ttext=0x1000` fuerza que la sección de código empiece en la dirección 0x1000 (por ejemplo).

  `-Tdata=0x2000` fuerza que la sección de datos empiece en la dirección 0x2000 (por ejemplo).

  `-oomagic` hace que las secciones de código y de datos sean de lectura y escritura (normalmente, la de código se marca como de solo lectura) y coloca la sección de datos inmediatamente después de la de texto.

- `nm <opciones> fich1 fich2...` produce un listado de los símbolos de los ficheros, que deben contener códigos objeto (resultados de `as`, `gcc` o `ld`), con sus valores y sus tipos (global, externo, en la sección de texto, etc.), ordenados alfabéticamente. Con la opción `-n` los ordena numéricamente por sus valores.
- `objdump <opciones> fich1 fich2...` da más informaciones, dependiendo de las opciones:

  `-d` desensambla: genera un listado en ensamblador de las partes que se encuentran en secciones de código.

  `-D` desensambla también las secciones de datos.

  `-r` muestra los diccionarios de reubicación.

  `-t` igual que `nm`, con otro formato de salida.

Android SDK

El SKD (Software Development Kit) de Android es una colección de herramientas con las que trabajará usted en la asignatura «Análisis y diseño de software». Se puede descargar libremente de `http://developer.android.com/sdk/`, y lo incluimos aquí porque en el apartado 4.10 hemos hecho referencia a uno de los componentes, el `adb` (Android Debug Bridge), con el que se pueden instalar y depurar aplicaciones en un dispositivo conectado al ordenador mediante USB.

Herramientas ARM

Keil es una de las empresas del grupo ARM, dedicada al desarrollo de herramientas de software. Todas están basadas en un ensamblador propio, que es algo distinto al de GNU que hemos visto aquí.

El «MDK-ARM» (Microcontroller Development Kit) es un entorno de desarrollo completo para varias versiones de la arquitectura ARM que solamente funciona en Windows. La versión de evaluación se puede descargar gratuitamente (aunque exige registrarse) de `http://www.keil.com/demo/`.

Bibliografía

El documento de Burks, Goldstine y von Neumann utilizado en el capítulo 1 se publicó en 1946 como un informe de la Universidad de Princeton con el título «Preliminary discussion of the logical design of an electronic computing instrument». Se ha reproducido en muchos libros y revistas, y se puede descargar de `http://www.cs.princeton.edu/courses/archive/fall10/cos375/Burks.pdf`
Es más claro y explícito que un documento previo de 1945 firmado solo por von Neumann, «First Draft of a Report on the EDVAC», que también se encuentra en la red:
`http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf`
(Estos URL se han comprobado el 30/05/2012).

Los libros de texto «clásicos» sobre arquitectura de ordenadores son los de Patterson y Hennessy:

- Computer Architecture: A Quantitative Approach, 5th ed. Morgan Kaufmann, 2011.
- Computer Organization and Design: The Hardware/Software Interface, rev. 4th ed. Morgan Kaufmann, 2011.

De ambos hay traducciones al español (pero de ediciones anteriores), publicadas por McGraw–Hill con los títulos «Arquitectura de computadores. Un enfoque cuantitativo» y «Organización y Diseño de Computadores. La Interfaz Hardware/Software».

De la arquitectura concreta del procesador ARM, la referencia básica son los «ARM ARM» (ARM Arquitecture Reference Manual). Estos documentos y otros muchos sobre ARM se pueden descargar de `http://infocenter.arm.com/`

También hay un libro específico sobre la programación de ARM (en el ensamblador de ARM): W. Hohl, «ARM Assembly Language: Fundamentals and Techniques», CRC Press, 2009

Sobre lenguajes de marcas y lenguajes de *script* se puede encontrar mucha información en la web. Un sitio recomendable es `http://http://www.w3schools.com/`: tiene «tutoriales» introductorios muy buenos (si no le importa que estén acompañados de bastante publicidad). Los estándares de HTML, XML, SVG y otros lenguajes de marcas se publican en el sitio del Consorcio WWW: `http://www.w3.org`. Y `http://www.ecmascript.org/` contiene el estándar de ECMAScript y varios foros de discusión.