# Elementos de Sistemas Operativos

DIT-UPM

© DIT-UPM, 2015. Algunos derechos reservados.
Este documento se distribuye con la licencia Creative Commons «by–sa» disponible en:
http://creativecommons.org/licenses/by-sa/4.0/legalcode

# Presentación

Este documento se ha elaborado a partir de unos apuntes diseñados para la asignatura «Arquitectura de computadores y sistemas operativos» («ARSO»), incluida en el plan de estudios del Grado en Ingeniería Biomédica que se imparte en la Escuela Técnica Superior de Ingenieros de Telecomunicación con la participación de otros cinco centros de la Universidad Politécnica de Madrid.

Tanto el contexto curricular como el entorno profesional de los futuros graduados hacían poco recomendable adoptar como material de estudio algunos de los magníficos libros de texto disponibles sobre estas materias. Se trata de una titulación multidisciplinar cuyo objetivo es la formación de ingenieros capaces de aplicar las tecnologías y desarrollar aplicaciones colaborando con otros profesionales en entornos médicos y sanitarios. Deben conocer los fundamentos de estos temas, pero sin la profundidad requerida por los informáticos, con los que, eventualmente, tendrán que trabajar.

Por otra parte, tanto en los textos como en los cursos sobre arquitectura y sobre sistemas operativos se supone que el estudiante tiene ya algunos conocimientos sobre la estructura y el funcionamiento de los ordenadores y sobre la representación de datos en binario. Normalmente, con alguna asignatura de nombre «Introducción a la informática», «Fundamentos de ordenadores»... Pero en el plan de estudios la única asignatura previa relacionada con la informática es «Fundamentos de programación», de primer curso; aunque incluye en su introducción algunos elementos superficiales, se centra, sobre todo, en la programación con el lenguaje Java y el entorno Eclipse. Igualmente superficial y sesgada por un punto de vista electrónico es la introducción a los procesadores digitales de la asignatura «Sistemas electrónicos», de segundo curso. Otras asignaturas relacionadas con la informática, que se imparten en el mismo curso que ARSO (tercero), son de un nivel de abstracción más alto: «Algoritmos y estructuras de datos» y «Bases de datos».

Esta situación nos motivó para elaborar unos materiales de apoyo a la docencia ajustados a los conocimientos previos de los estudiantes y a los objetivos de su formación. Creemos que el resultado, o parte de él, puede ser de utilidad para otras personas ajenas al contexto para el que ha sido concebido, y ese es el motivo por el que lo hacemos público. Para evitar malentendidos del eventual lector, sustituimos en el título el nombre de la asignatura por otro, más largo pero mejor ceñido al contenido.

El documento base que se entrega a los estudiantes es el formado por los capítulos 1 a 13 y los dos primeros apéndices. A lo largo del curso se proponen prácticas de laboratorio y ejercicios, y en esta edición se han incluido como apéndices algunos de los realizados en cursos pasados.

La asignatura se ha impartido durante tres cursos. Las profesoras Mercedes Garijo Ayestarán y Marifeli Sedano Ruiz no solamente han contribuido a la detección de errores y a la clarificación de muchos pasajes del texto, también han elaborado varias de las prácticas de laboratorio y de los ejercicios.

# Índice


**1 Introducción, terminología y objetivo . . . . . 1**
1.1 Datos, información y conocimiento . . . . . 1
1.2 Caudal y volumen . . . . . 3
1.3 Soportes de transmisión y almacenamiento . . . . . 4
1.4 Extremistas mayores y menores . . . . . 6
1.5 Ordenador, procesador, microprocesador, microcontrolador, SoC... . . . . 8
1.6 Niveles y modelos . . . . . 9
1.7 Arquitectura de ordenadores . . . . . 12
1.8 Objetivo y contenido . . . . . 13

**2 Sistemas operativos . . . . . 15**
2.1 ¿Qué es un sistema operativo? . . . . . 16
2.2 Tipos de sistemas operativos . . . . . 18
2.3 Recursos de hardware . . . . . 22
2.4 Servicios del sistema operativo (modelo funcional) . . . . . 29
2.5 Componentes del sistema operativo (modelo estructural) . . . . . 43
2.6 Funcionamiento del sistema operativo (modelo procesal) . . . . . 53
2.7 El *kernel*. Sistemas monolíticos, *microkernel* e híbridos . . . . . 66

**3 Representación de datos textuales . . . . . 71**
3.1 Codificación binaria . . . . . 71
3.2 Código ASCII y extensiones . . . . . 72
3.3 Códigos ISO . . . . . 74
3.4 Unicode . . . . . 75
3.5 Cadenas . . . . . 76
3.6 Texto con formato . . . . . 77
3.7 Hipertexto . . . . . 77
3.8 Documentos XML . . . . . 78

**4 Representación de datos numéricos . . . . . 79**
4.1 Números enteros sin signo . . . . . 79
4.2 Formatos de coma fija . . . . . 80
4.3 Números enteros . . . . . 80
4.4 Operaciones básicas de procesamiento . . . . . 81
4.5 Números racionales . . . . . 86

**5 Representación de [datos] multimedia . . . 91**
5.1 Digitalización . . . 92
5.2 Representación de sonidos . . . 95
5.3 Representación de imágenes . . . 96
5.4 Representación de imágenes en movimiento . . . 97

**6 Detección de errores y compresión . . . 99**
6.1 Bit de paridad . . . 100
6.2 Suma de comprobación *(checksum)* . . . 101
6.3 Códigos CRC . . . 102
6.4 Control de errores . . . 105
6.5 Compresión sin pérdidas *(lossless)* . . . 106
6.6 Compresión con pérdidas *(lossy)* . . . 109

**7 Almacenamiento en ficheros . . . 113**
7.1 Identificación del tipo de contenido . . . 113
7.2 Ficheros de texto, documentos, archivos y datos estructurados . . . 116
7.3 Ficheros multimedia . . . 117
7.4 Ficheros ejecutables binarios . . . 125

**8 La máquina de von Neumann y su evolución . . . 127**
8.1 Modelo estructural . . . 128
8.2 Modelo procesal . . . 131
8.3 Modelo funcional . . . 132
8.4 Evolución . . . 134
8.5 Modelos estructurales . . . 134
8.6 Modelos procesales . . . 135
8.7 Modelos funcionales . . . 138
8.8 Pilas y máquinas de pilas . . . 144
8.9 Comunicaciones con los periféricos e interrupciones . . . 147
8.10 Conclusión . . . 148

**9 El procesador BRM . . . 149**
9.1 Modelo estructural . . . 150
9.2 Modelo procesal . . . 152
9.3 Modelo funcional . . . 153
9.4 Instrucciones de procesamiento y de movimiento . . . 155
9.5 Instrucciones de transferencia de datos . . . 160
9.6 Instrucciones de transferencia de control . . . 162
9.7 Comunicaciones con los periféricos e interrupciones . . . 164

**10 Programación de BRM . . . 167**
10.1 Primer ejemplo . . . 167
10.2 Etiquetas y bucles . . . 169
10.3 El *pool* de literales . . . 171
10.4 Más directivas . . . 173
10.5 Menos bifurcaciones . . . 176

10.6 Subprogramas . . . 177
10.7 Módulos . . . 183
10.8 Comunicaciones con los periféricos e interrupciones . . . 185
10.9 Programando para Linux y Android . . . 190

**11 Paralelismo, arquitecturas paralelas y procesadores gráficos . . . 199**
11.1 Encadenamiento *(pipelining)* . . . 200
11.2 Paralelismo en los accesos a la memoria . . . 204
11.3 Memoria asociativa . . . 205
11.4 Arquitecturas paralelas . . . 207
11.5 Procesadores vectoriales . . . 208
11.6 Multiprocesadores . . . 210
11.7 Procesadores multihebrados *(multithreading)* . . . 211
11.8 Procesadores gráficos . . . 212

**12 Procesadores sofware . . . 219**
12.1 Ensambladores . . . 220
12.2 El proceso de montaje . . . 221
12.3 Tiempos de traducción, de carga y de ejecución . . . 223
12.4 Compiladores e intérpretes . . . 224
12.5 Compilación JIT y compilación adaptativa . . . 228
12.6 Virtualización, emulación y máquinas virtuales . . . 229

**13 Arquitectura de la máquina virtual Java . . . 233**
13.1 Visión de conjunto . . . 234
13.2 Modelo estructural . . . 238
13.3 Modelo funcional . . . 241
13.4 Modelo procesal: carga, montaje e iniciación . . . 252
13.5 Modelo procesal: interpretación . . . 254
13.6 Implementación de la JVM . . . 257

**A Sistemas de numeración posicionales . . . 263**
A.1 Bases de numeración . . . 263
A.2 Cambios de base . . . 263
A.3 Bases octal y hexadecimal . . . 265

**B El simulador ARMSim# y otras herramientas . . . 267**
B.1 Instalación . . . 267
B.2 Uso . . . 268
B.3 Otras herramientas . . . 270

**C Prácticas de laboratorio . . . 273**
Lab1: Trabajo con ficheros . . . 273
Lab2: Trabajo con directorios . . . 281
Lab3: Redirección, protección y procesos . . . 286
Lab4: Llamadas al sistema . . . 294
Lab5: Codificación de textos. Tipos de ficheros . . . 297

Lab6: ARMSim# . . . . . 304
Lab7: Llamadas en bajo nivel. Traducción y ejecución . . . . . 308

**D Ejercicios . . . . . 317**
Sobre representación . . . . . 317
Sobre detección de errores y compresión . . . . . 320
Sobre procesadores hardware . . . . . 322
Sobre BRM y procesadores software . . . . . 327

**E Soluciones de los ejercicios . . . . . 337**
Sobre representación . . . . . 337
Sobre detección de errores y compresión . . . . . 340
Sobre procesadores hardware . . . . . 343
Sobre BRM y procesadores software . . . . . 347

**Bibliografía . . . . . 357**

# Capítulo 1

# Introducción, terminología y objetivo

---

En la asignatura «Fundamentos de programación» usted ha estudiado los conceptos más básicos de la representación binaria de la información, y en «Sistemas electrónicos» los principios de los procesadores digitales. Este primer capítulo pretende profundizar en ambas cosas y proporcionar una visión unificada.

## 1.1. Datos, información y conocimiento

Con frecuencia utilizamos los términos «datos» e «información» como sinónimos, pero conviene diferenciarlos, y las definiciones «oficiales» no ayudan mucho. Las dos acepciones de «información» del Diccionario de la Real Academia Española más relevantes para nuestro propósito son:

> *5. f. Comunicación o adquisición de conocimientos que permiten ampliar o precisar los que se poseen sobre una materia determinada.*
>
> *6. f. Conocimientos así comunicados o adquiridos.*

La primera remite a los mecanismos por los que se adquieren los conocimientos, y la segunda parece identificar «información» con «conocimientos», concepto que define en singular:

> *Conocimiento:*
>
> *2. m. Entendimiento, inteligencia, razón natural.*

Y «dato» es:

> *1. m. Información sobre algo concreto que permite su conocimiento exacto o sirve para deducir las consecuencias derivadas de un hecho.*
>
> *3. m. Inform. Información dispuesta de manera adecuada para su tratamiento por un ordenador.*

¿Es la información conocimiento? ¿Son los datos información?

Para buscar definiciones más prácticas, operativas y orientadas a la ingeniería, podemos basarnos en un concepto de la rama de la Informática llamada «inteligencia artificial»: el concepto de *agente*[1]:

[1] Una explicación más detallada, con fuentes bibliográficas, de esta definición y de las siguientes puede encontrarse en el documento «Representación del conocimiento en sistemas inteligentes», disponible en `http://www.dit.upm.es/~gfer/ssii/rcsi/`.

Un **agente** es un sistema que percibe el entorno y puede actuar sobre él. Se supone que el agente es *racional*: tiene conocimientos sobre un dominio de competencia, mecanismos de razonamiento y objetivos, y utiliza los dos primeros para, en función de los datos que percibe, generar las acciones que conducen a los objetivos.

Debe usted entender «entorno», «percepción» y «actuación» en un sentido general: el agente puede ser una persona o un robot con sensores y actuadores sobre un entorno físico, pero también puede ser un programa que, por ejemplo, «percibe» el tráfico de paquetes de datos en una red (su entorno), los analiza y «actúa» avisando a otros agentes de posibles intrusiones.

En este contexto, las diferencias entre «datos», «información» y «conocimiento» se explican fácilmente:

- Los **datos** son estímulos del entorno que el agente percibe, es decir, meras entradas al sistema.
- La **información** está formada por datos *con significado*. El significado lo pone el agente receptor de los datos, que, aplicando sus mecanismos de razonamiento, los *interpreta* de acuerdo con sus conocimientos previos.
- El **conocimiento** es la información una vez asimilada por el agente en una forma sobre la que puede aplicar razonamientos.

Según estas definiciones, no tiene sentido decir que un libro o un periódico o la web, por ejemplo, contienen conocimiento, ni siquiera información. Sólo podría ser así si el libro, el periódico o la web tuviesen capacidad de razonamiento. Lo que contienen son datos, y lo que sí puede decirse de ellos es que son fuentes de información y conocimiento para los agentes capaces de asimilar esos datos. Seguramente ha oído usted hablar de la «web semántica»: los datos de las páginas, interpretables por agentes humanos, se complementan con más datos expresados en lenguajes interpretables por agentes artificiales. El nombre es un poco falaz, porque sin agentes no hay semántica (la semántica es la parte de la lingüística que se ocupa del significado, y éste lo pone el agente).

La definición de información como «datos con significado para un agente» es compatible con la clásica de la Teoría de la Información: la cantidad de información de un mensaje se mide por la reducción de incertidumbre en el agente receptor, es decir, depende del conocimiento previo de ese receptor. Si el periódico me dice que el Real Madrid ha ganado al Barça, la cantidad de información que me comunica es 1 bit, pero si yo ya lo sabía es 0 bits.

## ¿Qué es un bit?

Es posible que el final del párrafo anterior le haya sorprendido. Porque usted sabe interpretar el significado de «bit» en otros contextos. Por ejemplo, cuando una llamada telefónica irrumpe en su intimidad y una edulcorada voz le propone un servicio de «banda ancha» de «x megas». Usted entiende que «megas» quiere decir «millones de bits por segundo», y que lo que le está ofreciendo el inoportuno vendedor (o vendedora) es un *caudal de datos*, no de información. Si compra un *pecé* con un disco de un *terabyte* usted sabe que su *capacidad de almacenamiento de datos* es, aproximadamente, un billón de bytes, y que cada byte son ocho bits. Que ese caudal de datos o que los datos guardados en el disco contengan o no información para usted es otro asunto.

Ocurre que con el término «bit» designamos dos cosas que guardan cierta relación pero son diferentes:

- «Bit» es la unidad de medida de información: es la información que aporta un mensaje a un agente cuando este mensaje le resuelve una duda entre dos alternativas igualmente probables.
- «Bit» es la contracción de *«binary digit»:* es un *dígito binario* con dos valores posibles, «0» o «1». Éste es el significado que aplicaremos en adelante.

### Nivel de abstracción binario

Todos los datos que manejamos en telemática están *codificados en binario*, es decir, un dato está siempre formado por bits. Físicamente, un bit se materializa mediante algún dispositivo que puede estar en uno de dos estados posibles. Por ejemplo un punto en un circuito eléctrico que puede tener +5 voltios o 0 voltios, o una pequeña zona en la superficie de un disco magnetizada en un sentido o el opuesto. Nosotros hacemos abstracción de esos detalles físicos: todo está basado en componentes que pueden estar en uno de dos estados, y llamamos a esos estados «0» y «1».

### Resumiendo...

Tras esta introducción y tras un capítulo sobre sistemas operativos dedicaremos cinco capítulos a los convenios para representar, almacenar y transmitir distintos *tipos de datos:* textos, números, sonidos, imágenes, etc. Los datos forman parte de mensajes que envía un agente emisor a otro receptor a través de un *canal de comunicación*, y es necesario que ambos se atengan a los mismos convenios para que puedan entenderse. En un acto de comunicación, cada dato que llega al receptor puede aportarle información o no, dependiendo del estado de sus conocimientos, pero éste es un asunto en el que no entraremos.

En los seis capítulos siguientes (del 8 al 13) estudiaremos los principios de funcionamiento de las máquinas para el procesamiento de datos así representados. Pero antes es necesario detenerse en algunos convenios terminológicos y de notación.

## 1.2. Caudal y volumen

El **caudal** de un canal de comunicación es la **tasa de bits** *(bitrate)*, medida en bits por segundo. Se suele abreviar como «bps», y se utilizan los múltiplos habituales del Sistema Internacional: kilo, mega, etc.

Para un caudal constante, es trivial calcular el **volumen** de datos transmitido en un intervalo de tiempo mediante una multiplicación. Si, cediendo ante la insistencia de su convincente vendedor, contrata un servicio de 50 «megas» (50 Mbps), usted razonablemente espera que en una hora de conexión recibirá $50 \times 3.600 \times 10^6 = 180$ gigabits. Esto es correcto (si se cumple la promesa del vendedor), pero hay que matizar dos cosas, una que seguramente le es familiar, la otra quizás no tanto.

Sin duda le resulta familiar la palabra «byte». El volumen de datos normalmente está relacionado con la **capacidad de almacenamiento** de algún sistema capaz de guardarlos, lo que solemos llamar «memoria». En estos sistemas la unidad básica de almacenamiento es un conjunto de ocho bits, conocido como **octeto**, o **byte**, y ésta es también la unidad para expresar la capacidad. De modo que si usted pretende guardar en el disco de su ordenador los datos recibidos en el ejemplo anterior necesitará disponer de al menos $180/8 = 22,5$ GB (gigabytes). En ocasiones conviene referirse a la mitad de un byte, cuatro bits: en inglés se llama *«nibble»*; en español solemos decir «**cuarteto**».

### kilo y kibi, mega y mebi...

La segunda cosa a matizar es lo que queremos decir al hablar de una memoria de «x gigabytes». Más adelante veremos que la capacidad de almacenamiento de una memoria, medida en número de bytes, es siempre una potencia de 2. No tiene sentido una memoria de exactamente 1 KB. La potencia de 2 más próxima a 1.000 es $2^{10} = 1.024$. Coloquialmente solemos decir «kilobytes», «megabytes»,

etc., pero hablando de capacidad de almacenamiento de una memoria es incorrecto: los prefijos multiplicadores definidos en el Sistema Internacional de Medidas son potencias de diez, no de dos. Hay un estándar definido por la IEC (International Electrotechnical Commission) para expresar los multiplicadores binarios: «kibi» (kilo binary), «mebi» (mega binary), «gibi» (giga binary), etc. Es el resumido en la tabla 1.1, y el que utilizaremos en lo sucesivo.

| **Decimales (SI)** | | **Binarios (IEC)** | |
|---|---|---|---|
| **Valor** | **Prefijo** | **Valor** | **Prefijo** |
| $10^3$ | kilo (k) | $2^{10}$ | kibi (Ki) |
| $10^6$ | mega (M) | $2^{20}$ | mebi (Mi) |
| $10^9$ | giga (G) | $2^{30}$ | gibi (Gi) |
| $10^{12}$ | tera (T) | $2^{40}$ | tebi (Ti) |
| $10^{15}$ | peta (P) | $2^{50}$ | pebi (Pi) |
| $10^{18}$ | exa (E) | $2^{60}$ | exbi (Ei) |
| $10^{21}$ | zetta (Z) | $2^{70}$ | zebi (Zi) |
| $10^{24}$ | yotta (Y) | $2^{80}$ | yobi (Yi) |

**Tabla 1.1**: Prefijos multiplicadores.

### Binario y hexadecimal

| Hex. | Bin. |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| A | 1010 |
| B | 1011 |
| C | 1100 |
| D | 1101 |
| E | 1110 |
| F | 1111 |

Se supone que entre sus conocimientos previos están los que se refieren a los sistemas de numeración. Si no es así, acuda al apéndice A, en el que se resumen.

Todos los datos se codifican en binario para que puedan ser tratados por los sistemas digitales. Pero «hablar en bits», como hacen ellos (los sistemas), es prácticamente imposible para la mente humana. Sin embargo, a veces tenemos que referirnos al valor concreto de un byte, o de un conjunto de bytes. Por ejemplo, en el siguiente capítulo veremos que el carácter «Z» se codifica así: 01011010. Es casi inevitable equivocarse al escribirlo, y mucho más si hablamos de una sucesión de caracteres o, en general, de un contenido binario de varios bytes.

Una posibilidad es interpretar «01011010» como la expresión en base 2 de un número entero: $2^6+2^4+2^3+2^1 = 90$, y decir que la codificación de «Z» es 90. Pero más fácil es utilizar el sistema hexadecimal (base 16), porque la conversión de hexadecimal a binario o a la inversa es inmediata: basta con agrupar los bits en cuartetos y tener en cuenta la correspondencia entre dígitos hexadecimales y su expresión en binario, como indica la tabla al margen.

La codificación «binaria» (expresada en hexadecimal) de «Z» es: **0x5A**. «0x» es el prefijo más común para indicar que lo que sigue está expresado en hexadecimal.

## 1.3. Soportes de transmisión y almacenamiento

Para la transmisión de datos existe una variedad de medios físicos: cable de cobre trenzado, coaxial, fibra óptica, medios no guiados, etc., con distintas características de ancho de banda, coste, etc., como

conoce usted de la asignatura «Redes de comunicaciones», y no nos detendremos en ellos. Pero sí tenemos que dar algunos detalles sobre los soportes de almacenamiento.

Un **punto de memoria** es un mecanismo físico capaz de almacenar un bit. Hay diversas tecnologías (electrónicas, magnéticas, ópticas, etc.) que permiten implementar este mecanismo, dando lugar a diferentes **soportes** para almacenar bits. Los soportes tienen características variadas:

- **capacidad**: número de bits, medido en bytes, o KiB, etc.;
- posibilidad de **escritura** (grabación de un conjunto de bits) o de sólo **lectura** (recuperación de un conjunto de bits);
- **tiempos de acceso** para la lectura y la escritura: tiempo que transcurre desde que se inicia una operación hasta que concluye;
- **coste por bit**;
- **densidad de integración**: número de bits por unidad de superficie;
- **volatilidad**: el soporte es volátil si requiere alimentación eléctrica para conservar los contenidos almacenados.

Generalmente el subsistema de memoria de un sistema informático es una combinación de varios soportes, lo que conduce, como veremos en el apartado 2.3, al concepto de **jerarquía de memorias**.

### Biestables y registros

Un **biestable**, o **flip-flop**, es un circuito que conoce usted de la asignatura «Sistemas electrónicos». En todo momento, el biestable se encuentra en uno de dos estados posibles y permanece en él mientras no se le aplique un estímulo para cambiar de estado. Por tanto, sirve como punto de memoria. Veremos en el capítulo 4 que, acompañando al resultado de las operaciones aritméticas y lógicas, hay **indicadores** que toman uno de entre dos valores indicando si ese resultado fue negativo, si se ha producido desbordamiento, etc. Esos indicadores son biestables.

A partir del capítulo 8 estudiaremos los procesadores. Un procesador hardware contiene **registros**, que no son más que conjuntos de biestables. En cada registro puede almacenarse un conjunto de bits conocido como «palabra». La **longitud de palabra** es el número de bits que contienen los registros. Valores típicos son 8, 16, 32 o 64 (uno, dos, cuatro u ocho bytes), dependiendo del procesador.

Los registros son volátiles, pero son los soportes más rápidos dentro de la jerarquía de memorias. Los tiempos de acceso para la escritura (grabación de una palabra) y para la lectura (recuperación del contenido) son de pocos nanosegundos. Y los registros son también los soportes que tienen mayor coste por bit y menor densidad de integración. Los procesadores pueden incluir varias decenas de registros. Con 30 registros de una longitud de palabra de 64 bits resulta una capacidad de $30 \times 64/8 = 240$ bytes. Una capacidad del orden de KiB sólo con registros es, hoy por hoy, impracticable, tanto por su coste como por su tamaño.

### Memorias de acceso aleatorio

Las memorias llamadas «de acceso aleatorio» tienen capacidades de MiB o GiB. Para poder acceder a un byte determinado de la memoria es necesario identificarlo, y para ello cada byte tiene asignada una **dirección**, que es un número entero comprendido entre 0 y $M - 1$, donde $M$ es la **capacidad** de la memoria. Cuando tiene que leer (extraer) o escribir (grabar) un byte, el procesador genera su dirección y la pone en un registro (el «registro de direcciones») para que los circuitos de la memoria activen el byte (o la palabra) que corresponda. Si el registro tiene $n$ bits, las direcciones que puede contener están comprendidas entre 0 y $2^n - 1$. Por ejemplo, con un registro de 16 bits las direcciones posibles son

0x0000, 0x0001... 0xFFFF: en total, $2^{16}$ = 65.536 direcciones. En este caso, la máxima capacidad de memoria que puede direccionarse es $2^{16} = 2^6 \times 2^{10}$ bytes = 64 KiB. Con 32 bits resulta $2^{32}$ = 4 GiB.

«Acceso aleatorio» no significa «acceso al azar». Sólo quiere decir que el tiempo de acceso para la lectura o la escritura es el mismo para todos los bytes, independiente de la dirección. Estas memorias se conocen habitualmente como «**RAM**» *(Random Access Memory)*[2]. Conviene aclarar un malentendido bastante común. Es frecuente que una pequeña parte de la memoria esté construida con una tecnología que no es volátil, para conservar los programas de arranque y algunas rutinas básicas. Esta tecnología permite construir una memoria no volátil, pero de sólo lectura, o «**ROM**» *(Read Only Memory)*: una vez grabada por el fabricante no puede borrarse ni volver a escribirse[3]. Esto ha conducido a que en la jerga habitual se distinga entre RAM y ROM como si fuesen cosas excluyentes, cuando la ROM es también RAM. Sí son excluyentes «R/W» (*«Read/Write»:* lectura/escritura) y «ROM».

La memoria principal de un ordenador es siempre de acceso aleatorio. Si la longitud de palabra es, por ejemplo, cuatro bytes, una palabra se almacena en cuatro bytes de la memoria de direcciones consecutivas. Una palabra de la memoria puede contener una instrucción para el procesador, un dato o una dirección de memoria. Una palabra cuyo contenido se interpreta como una dirección se llama **puntero**, porque *apunta a* otra palabra.

### Memorias secundarias y terciarias

Diversas tecnologías (discos y cintas magnéticos, discos ópticos, *flash*, etc.) permiten construir soportes para el almacenamiento masivo de datos fuera de la memoria principal. De la gestión de esta **memoria secundaria** se ocupa automáticamente el sistema operativo, mientras que la **memoria terciaria** es aquella que requiere alguna intervención humana (como insertar o extraer un DVD).

En general, estos soportes se diferencian de los utilizados para implementar la memoria principal en que:

- no son volátiles,
- el acceso a los datos no es «aleatorio», y el tiempo medio de acceso es mayor, y
- no se comunican directamente con el procesador: los datos se leen o se escriben en la memoria principal en **bloques** de un tamaño mínimo de 512 bytes, mediante la técnica de acceso directo a memoria (DMA).

El convenio de almacenamiento en las memorias secundarias y terciarias se basa en estructurar los datos en **ficheros**. A los ficheros también se les suele llamar «**archivos**» pero en el apartado 6.5 (página 108) matizaremos: los archivos son ficheros, pero no todos los ficheros son archivos. En el apartado 2.3 ampliaremos algunos detalles sobre las memorias secundarias y en el capítulo 7 estudiaremos algunos convenios concretos para el almacenamiento en ficheros de los distintos tipos de datos.

## 1.4. Extremistas mayores y menores

Hemos dicho que cada dirección identifica a un byte dentro de la memoria principal. Y también que en la memoria se pueden almacenar palabras que, normalmente, ocupan varios bytes: dos, cuatro u ocho. ¿Cómo se identifica a una palabra, para poder escribirla o leerla? Por la dirección de uno de

[2] Aunque se sigue utilizando el nombre «RAM», las actuales «DRAM» (RAM dinámicas) no son, en rigor, de acceso aleatorio, porque en ellas no se accede a los datos byte a byte, o palabra a palabra, sino a ráfagas.

[3] En realidad, se usan tecnologías que no son estrictamente ROM, porque la memoria puede regrabarse, aunque a menor velocidad. Se llaman EEPROM *(Electrically Erasable Programmable ROM)*. Las bien conocidas memorias *flash* son un tipo de EEPROM.

sus bytes, pero ¿en qué direcciones se almacenan éstos? Parece obvio que en direcciones consecutivas, pero ¿en qué orden?

Gráficamente se entenderán mejor los posibles convenios para responder a esas preguntas. La representación gráfica de una palabra de $n$ bits es un rectángulo horizontal con una numeración de 0 a $n-1$ de los bits que componen la palabra, como muestra la figura 1.1 para el caso de $n = 16$. Esta numeración coincide con los pesos de los bits cuando el conjunto se interpreta como un número entero en binario. Al bit de peso 0 le llamamos «bit menos significativo» («bms» en la figura) y al de peso $n-1$ «bit más significativo» («bMs»).

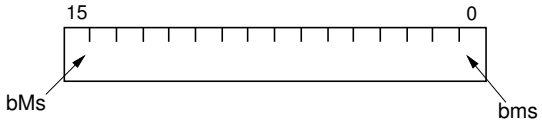



**Figura 1.1**: Representación gráfica de una palabra de 16 bits.

Por otra parte, para la memoria utilizaremos gráficos como el de la figura 1.2, con las direcciones en el margen izquierdo y empezando desde arriba con la dirección 0. Esta figura correspondería a una memoria de 16 MiB. En efecto, las direcciones tienen seis dígitos hexadecimales, y la dirección mayor (que está en la parte más baja de la figura) es 0xFFFFFF $= 16^6 - 1$. El número total de direcciones es: $16^6 = 2^4 \times 2^{20}$.

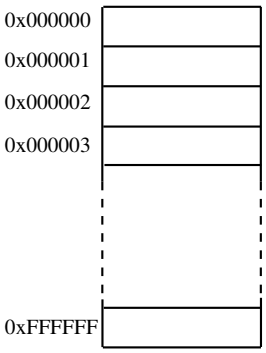



**Figura 1.2**: Representación gráfica de una RAM.

Volviendo a las preguntas anteriores sobre el almacenamiento de palabras, y suponiendo palabras de 16 bits, la figura 1.3 muestra los dos posibles convenios para almacenar sus dos bytes en las direcciones $d$ y $d+1$. El convenio de que al byte que contiene los bits menos significativos (del 0 al 7) le corresponda la dirección menor se le llama **extremista menor** *(little endian)*, y al opuesto, **extremista mayor** *(big endian)*.

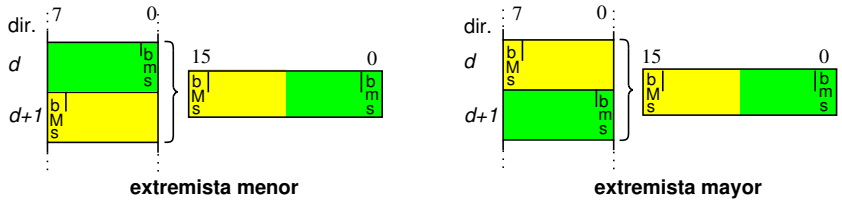



**Figura 1.3**: Convenios de almacenamiento para una palabra de dos bytes.

La generalización para otras longitudes de palabra es fácil. Así, una palabra de 32 bits ocupará cuatro direcciones, desde la $d$ a la $d+3$. Con el convenio extremista menor al byte menos significativo le corresponde la dirección $d$, mientras que con el extremista mayor le corresponde la $d+3$. En ambos casos, se considera que la dirección de la palabra es la más baja de las cuatro, $d$.

La diferencia de convenios afecta no sólo al almacenamiento, también al orden en que se envían los bytes en las comunicaciones. Algunos procesadores (por ejemplo, los de Intel) están diseñados con el convenio extremista menor, y otros (por ejemplo, los de Motorola) con el extremista mayor. El problema surge cuando uno de ellos interpreta una sucesión de bytes enviada por el otro. Por ejemplo, las direcciones IP (versión 4) se representan como un conjunto de 32 bits: cada una de las cuatro partes separadas por puntos es la traducción a decimal de un byte. Así, «138.4.2.61» en realidad es «0x8A04023D». Si una máquina con procesador Motorola genera esta dirección, envía los bytes por la red en el orden 8A-04-02-3D (primero lo más significativo) y lo recibe otra con procesador Intel, para ésta lo que llega primero es lo menos significativo, resultando que interpreta la dirección «61.2.4.138». Por cierto, los protocolos de la familia TCP/IP son extremistas mayores, por lo que es en el software de la máquina basada en Intel en el que hay que invertir el orden para que la comunicación sea correcta.

## 1.5. 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 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 este documento dirigido a lectores de España, utilizaremos siempre el término «**ordenador**»[4], entendiendo, además, que es de naturaleza *digital*.

«**Procesador**» se define así[5]:

> *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, 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**, o **CPU** *(Central Processing Unit)*.
- 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 de acceso aleatorio.
- **Procesador software:** Programa que transforma unos datos de entrada en unos datos de salida.

[4] 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. Es posible que esa preferencia esconda un afán elitista. En cualquier caso, utilizaremos el nombre más común.

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

«**Microprocesador**» tiene una definición fácil[6]: 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 8.6).

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 CPU, 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 CPU 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 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.

## 1.6. 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 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 veremos con detalle en los siguientes capítulos, 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**, de **circuito eléctrico** y de **dispositivo**, que se estudian en la Electrónica digital, la Electrónica analógica y la Electrónica física.

La figura 1.4 presenta gráficamente una jerarquía de niveles de abstracción en los que se pueden describir los procesadores.

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**.

[6] 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, y el procesador gráfico Nvidia GK110 más de siete millardos.

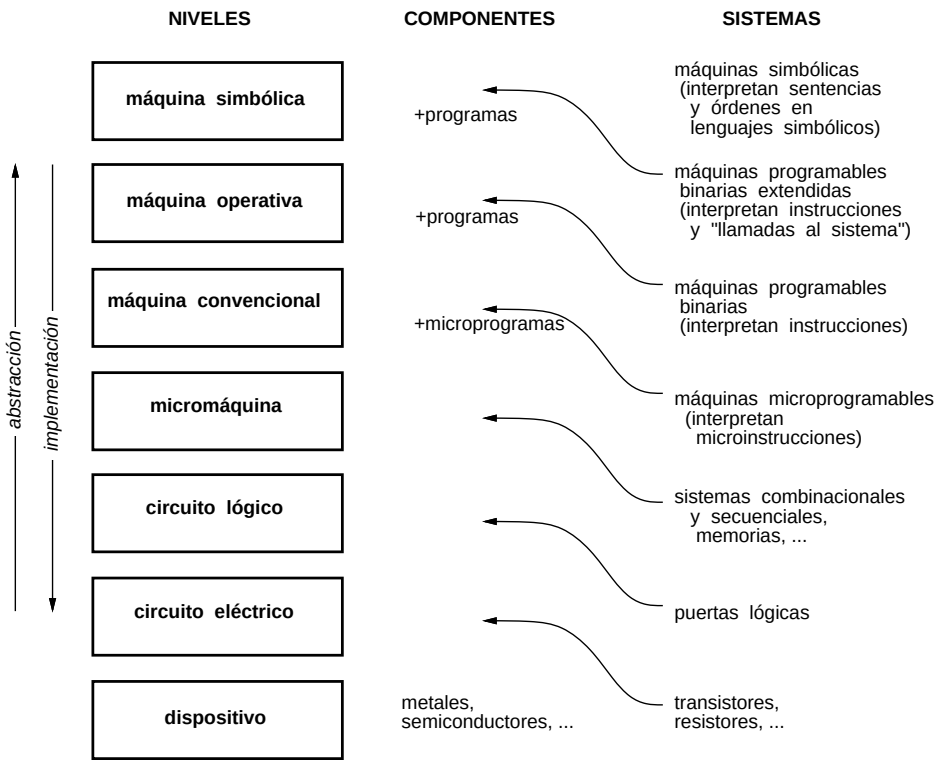



**Figura 1.4**: Niveles de abstracción para los procesadores.

## 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**[7], 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 abstracción 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 normalmente tiene un procesador Intel o AMD) y quiere que funcione también en un dispositivo móvil (que normalmente 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.

[7] «Ensamblador» es un término ambiguo: se refiere al lenguaje y al procesador (página 167).

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 `y` 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 que facilitan el uso de los recursos (las memorias, los procesadores y los periféricos), ofreciendo **servicios** al nivel de máquina simbólica. Éste es el nivel que estudiaremos en el capítulo 2.

## Modelos funcionales, estructurales y procesales

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 1.5). 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.

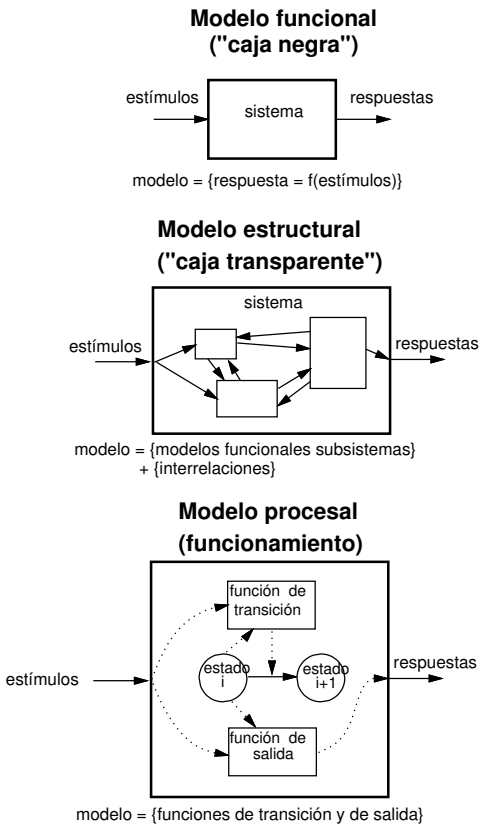



**Figura 1.5**: Tipos de modelos.

Un ejemplo:

Imagínese 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 los capítulos 2 y 8 presentaremos algunos modelos estructurales muy genéricos (figuras 2.7, 2.8 y 8.7, y en el capítulo 9 el de un procesador concreto (figura 9.1).

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.

## 1.7. 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. El diccionario de la R.A.E. define «arquitectura» en su acepción informática como «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, es decir, «ISA») de un procesador define los elementos y las funciones que debe conocer el programador que trabaje en el nivel de máquina convencional. 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 (el «nivel de micromáquina»), y la realización corresponde a los modelos en los niveles de circuito lógico e inferiores.

## 1.8. Objetivo y contenido

El objetivo de este documento es facilitar el estudio de la asignatura «Arquitectura de computadores y sistemas operativos». El programa publicado en la Guía de Aprendizaje[8] está estructurado en cuatro «temas»:

**Tema 1:** Sistemas operativos

**Tema 2:** Representación de la información

**Tema 3:** Arquitectura de procesadores hardware

**Tema 4:** Procesadores software

La asignatura comprende aspectos teóricos, que son los cubiertos aquí, y prácticos, que se detallan en otros documentos con ejercicios y prácticas de laboratorio.

A la parte teórica del Tema 1 se dedica el capítulo 2. A continuación se estudian los convenios de representación en bajo nivel (en binario) de datos de tipo textual (capítulo 3), numérico (capítulo 4) y multimedia (capítulo 5, donde introduciremos también algunas formas de representación simbólica, no binaria, de sonidos e imágenes).

Los datos de tipo audiovisual y los que proceden de sensores del entorno físico requieren para su representación en binario procesos de digitalización que usted debe conocer de la asignatura «Sistemas y señales», que recordaremos en el capítulo 5.

[8] `http://www.dit.upm.es/arso/`

Veremos los principios de la detección de errores y la compresión, parte también del Tema 2, en el capítulo 6. Y en el capítulo 7 resumiremos algunos convenios para el almacenamiento en ficheros de todos estos tipos de datos.

A partir del capítulo 8 entramos en la arquitectura de ordenadores y de los procesadores hardware y software (Temas 3 y 4). Empezamos 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 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 sólo 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, analizaremos otros procesadores hardware y estudiaremos los principios de los procesadores software. Finalmente, describiremos la arquitectura de un procesador software, la máquina virtual Java, en el capítulo 13.

En un apéndice se dan las instrucciones para instalar y utilizar un simulador con el que practicar la programación de BRM, y al que se dedica una de las pŕacticas de laboratorio. Y se sugieren otras herramientas más avanzadas por si a usted le gusta tanto este asunto que está interesado en explorarlo más a fondo; para este caso también se incluye una bibliografía recomendada.

# Capítulo 2

# Sistemas operativos

---

La Guía de Aprendizaje de la asignatura enuncia estos resultados de aprendizaje para el Tema 1:

- Comprender las funciones y la necesidad de los sistemas operativos.
- Conocer la estructura y el funcionamiento de los distintos tipos de sistemas operativos.
- Conocer los conceptos de organización de ficheros, procesos y recursos.
- Saber utilizar los comandos de Unix para manejo de ficheros, de procesos y de recursos.

El enfoque que se adopta para los dos últimos puntos en la impartición de la asignatura es eminentemente práctico, con sesiones de laboratorio que se describen en otros documentos. Este capítulo presenta las nociones teóricas y generales.

Usted ya dispone (aunque sólo sea por la asignatura «Fundamentos de programación») de algunas ideas básicas sobre la organización de un sistema informático: es una simbiosis de recursos de hardware (CPU, memorias y periféricos) y de software (programas que se ejecutan en la CPU). Y, desde el momento en que ha desarrollado programas, ha hecho uso de algún sistema operativo. Al estudiar esta otra asignatura comprenderá fácilmente la necesidad de los sistemas operativos y, en general, del «software de sistemas»: los recursos de hardware pueden programarse, como veremos en el capítulo 10, en bajo nivel (en binario), pero la tarea de desarrollar programas de aplicación sería inabordable en ese nivel. Ya ha comprobado que estos programas se desarrollan en algún lenguaje de alto nivel, y para que puedan ejecutarse es preciso que existan otros programas que los traduzcan a bajo nivel y que gestionen los recursos de hardware de manera transparente al programador.

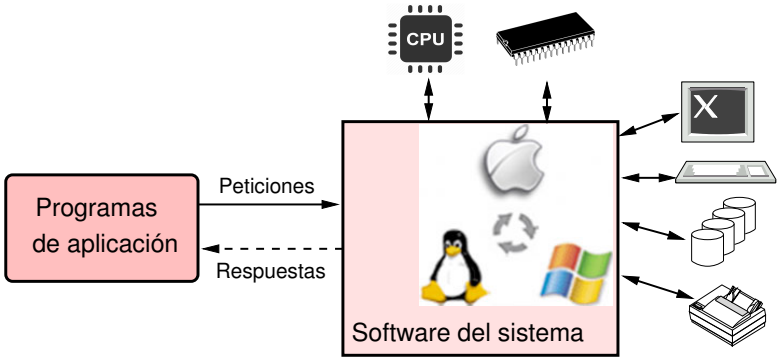



**Figura 2.1**: El sistema operativo (Windows, Linux, OS X...) como facilitador de las aplicaciones.

## 2.1. ¿Qué es un sistema operativo?

Comparemos dos definiciones de fuentes muy distintas. Según el Diccionario de la Lengua Española (DRAE), un sistema operativo es un:

> *Programa o conjunto de programas que realizan funciones básicas y permiten el desarrollo de otros programas.*

Es una definición imprecisa[1]. Por ejemplo, una función básica y que permite el desarrollo de programas es la que cumple un programa traductor como el que usted ha utilizado para procesar sus programas fuente en lenguaje Java. Sin embargo, no se considera que este tipo de programa forme parte del sistema operativo.

Una definición más técnica y más completa es la del FOLDOC[2], que traducida sería:

> *Software de bajo nivel que gestiona la interfaz con el hardware periférico, planifica tareas, asigna almacenamiento y presenta al usuario una interfaz por defecto cuando no se está ejecutando ningún programa de aplicación.*

El FOLDOC la complementa de esta manera:

> *El sistema operativo puede dividirse en un* kernel *que siempre está presente y varios programas del sistema que hacen uso de las facilidades proporcionadas por el* kernel *para realizar tareas rutinarias de nivel más alto, actuando frecuentemente en una relación cliente-servidor.*
>
> *Algunos incluyen una interfaz gráfica de usuario y un sistema de ventanas como parte del sistema operativo; otros no. El cargador del sistema operativo, la BIOS y otros requisitos de* firmware *para el arranque o la instalación del sistema generalmente no se consideran parte de él, aunque esta distinción no está muy clara en el caso de un sistema diseñado para residir en ROM* («rommable»)*, como RISC OS.*
>
> *Los servicios que proporciona un sistema operativo y su filosofía de diseño en general ejercen una gran influencia en el estilo de programación y sobre las culturas técnicas que afloran en torno a las máquinas en las que se ejecuta el sistema.*

Esta descripción pone de manifiesto que no hay una distinción clara sobre la pertenencia o no de ciertos programas al sistema operativo. Un poco mejor definida es la distinción clásica entre «software de sistemas» y «software de aplicaciones».

### Software de aplicaciones y software de sistemas

Forman el software de aplicaciones los programas que realizan tareas concretas para los usuarios: procesadores de textos y de documentos, hojas de cálculo, navegadores web, reproductores multimedia, gestores de bases de datos, etc. Estos programas se apoyan en otros cuya función no es ya una tarea concreta, sino proporcionar servicios generales. A estos otros se les llama «software de sistemas» (o «software del sistema»).

De este modo, el *programador de aplicaciones* (papel que usted ha practicado en las asignaturas «Fundamentos de programación» y «Algoritmos y estructuras de datos») se encuentra con una «máquina arropada», en el sentido de que dispone no sólo de un hardware, sino también de un conjunto de programas de utilidad que han sido previamente desarrollados para él por *programadores de sistemas*.

[1] Era algo mejor la definición de la edición anterior (la 22ª): «Programa o conjunto de programas que efectúan la gestión de los procesos básicos de un sistema informático, y permite la normal ejecución del resto de las operaciones».

[2] *Free On-Line Dictionary Of Computing,* `http://foldoc.org/operating+systems`

El sistema operativo es parte del software del sistema. Otros programas del software del sistema que *no suelen* considerarse parte del sistema operativo son:

- Los procesadores de lenguajes: traductores (ensambladores y compiladores) e intérpretes.
- Los entornos de desarrollo integrados (IDEs), que incluyen procesadores de textos y de lenguajes, depuradores, facilidades para el desarrollo de interfaces gráficas, etc.
- Las bibliotecas (o librerías), que contienen rutinas (programas que no son directamente ejecutables porque están diseñados para complementar otros) con funciones muy variadas.
- Los programas de respaldo *(backup)*.
- Los programas intermediarios *(middleware)* que dan servicios a aplicaciones distribuidas.
- Los programas que implementan interfaces de usuario, ya sean textuales (intérpretes de órdenes, o *shells*) o gráficas *(GUIs)*.

Pero no puede establecerse una frontera precisa para delimitar lo que forma parte y lo que no del sistema operativo. El motivo es práctico: un sistema operativo sin el acompañamiento de algunos otros programas sería tan inoperante como el hardware «desnudo». Qué programas son necesarios depende del uso del sistema:

- Para los *usuarios finales* la máquina tiene que estar dotada, junto con el sistema operativo, de una interfaz gráfica, de algunas utilidades (como un programa de respaldo) y, por supuesto, de los programas de aplicación necesarios para este tipo de usuarios. Es, por tanto, frecuente que bajo el nombre de «sistema operativo» se incluyan estos programas[3].
- Para los *desarrolladores de aplicaciones* es precisa, al menos, una interfaz de texto, bibliotecas y herramientas de desarrollo: procesadores de lenguajes, depuradores, etc.
- Para los *administradores del sistema*, además de la interfaz de texto, se necesitan utilidades de gestión de usuarios, diagnóstico, mantenimiento, auditoría, etc.
- En el caso de *servidores* (de web, de correo, etc.) el sistema tiene que incluir, obviamente, los programas necesarios para dar el servicio y para su gestión. Estos programas no forman parte del sistema operativo, pero éste tiene que dar las facilidades necesarias para el acceso a la red.
- En el caso de *sistemas para aplicaciones específicas*, en los que los usuarios finales hacen un uso indirecto del sistema operativo y ni siquiera son conscientes de su existencia (por ejemplo, los incorporados en televisores y otros electrodomésticos), el software integra, grabado en memoria ROM o *flash*, el sistema operativo, todos los programas necesarios y una interfaz muy específica controlada normalmente por botones, o por una pantalla táctil, o por un mando a distancia (a veces, como en los *routers*, también interfaces de texto y gráficas accesibles por la red).

[3] Esto ha provocado disputas legales. Por ejemplo, el «caso Estados Unidos contra Microsoft», litigio que se inició en 1998 con la denuncia de que integrar el navegador Internet Explorer con Windows violaba las leyes antimonopolio. El caso concluyó en 2004 (de manera controvertida) con el compromiso de la empresa de facilitar a terceros la información técnica necesaria para integrar otros productos.

## 2.2. Tipos de sistemas operativos

Se desprende de lo anterior que hay muchos tipos de entornos informáticos y telemáticos y, consiguientemente, muchos tipos de sistemas operativos. Podemos intentar clasificarlos desde dos puntos de vista: el tipo de trabajos a servir y los recursos hardware a gestionar. Al hacerlo iremos introduciendo algunos conceptos y términos que harán más fácil comprender los apartados siguientes.

### Naturaleza de los trabajos a los que da servicio el sistema

- **Procesamiento por lotes** *(batch processing)*.

Los programas que se procesan por lotes no requieren ninguna interacción directa con los usuarios finales. Es una forma de procesamiento tradicional que se introdujo en los ordenadores *(mainframes)*[4] de los años 1950: los usuarios preparaban sus programas y sus datos en máquinas independientes (perforadoras de tarjetas) y los entregaban a un operador, que los introducía en una máquina lectora, de donde pasaban a una cinta magnética o a un disco. Los trabajos se agrupaban en «lotes» que tenían necesidades similares (más uso de la CPU, o del disco, o de la impresora, etc.) y el sistema operativo optimizaba el uso de los recursos de manera que, por ejemplo, la CPU no quedase «congelada» porque el programa en ejecución no pudiese seguir hasta completar una lenta operación de entrada/salida (comunicación con un periférico). En este caso, el sistema cedía el uso de la CPU a otro programa de otro lote.

Este modo de funcionamiento, actualmente obsoleto en los términos descritos, introdujo un concepto fundamental en la evolución de los sistemas operativos: la **multiprogramación**. Los programas están almacenados en memoria secundaria, pero para poder ejecutar un programa éste tiene que estar en la memoria principal. Así, antes de su ejecución, hay que *cargar* el programa: copiarlo de la memoria secundaria a la principal. La monoprogramación consistía en cargar el primer programa, esperar a su finalización, cargar el siguiente y así sucesivamente. Esto era muy ineficiente, porque las operaciones de entrada/salida son mucho más lentas que la CPU, y ésta estaba mucho tiempo inactiva. En multiprogramación son varios los programas que se cargan y el sistema operativo se encarga de ceder la CPU a otro programa cuando el que se está ejecutando pide una operación de entrada/salida.

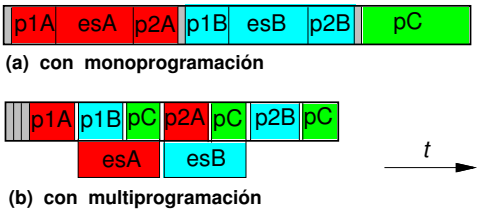



**Figura 2.2**: Diagramas de tiempos en la ejecución de tres programas.

La figura 2.2 ilustra gráficamente la idea en un ejemplo de ejecución de tres programas, A, B y C, dos de ellos idénticos (en cuanto a tiempos): tienen una parte de procesamiento (p1A, p1B) seguida de una de entrada/salida (esA, esB) de duración doble a la anterior y terminan con otra parte de procesamiento (p2A, p2B) de igual duración que la primera; el tercer programa sólo tiene procesamiento (pC). Las franjas sombreadas corresponden a la carga de los programas en la memoria, y las blancas (sólo en el caso de multiprogramación) a la sobrecarga de tiempo del sistema operativo para conmutar de uno a otro. Gracias a que las operaciones de entrada/salida se pueden realizar simultáneamente con las de procesamiento, el tiempo total para la ejecución de los tres programas se reduce. Además, las dimensiones de la figura son poco realistas: en la mayoría de los programas que tienen entrada/salida la duración de ésta no es el doble, sino desproporcionadamente mayor que la de procesamiento, y son cientos o miles los programas. La reducción de tiempo es mucho mayor de lo que sugiere la figura.

[4] El nombre proviene de *«main frame»*, el bastidor principal que contenía la CPU y la memoria principal. Luego, a partir de los años 1970, se ha utilizado para distinguir los grandes ordenadores de los miniordenadores y microordenadores.

Se llama **potencia** *(throughput)* al número de trabajos realizados por unidad de tiempo. El objetivo de la multiprogramación es alcanzar la máxima potencia con los recursos disponibles.

Nos hemos servido del origen histórico para explicar la multiprogramación porque se entiende mejor así, pero la esencia del concepto se mantiene en sistemas actuales. Por ejemplo, en los grandes sistemas para aplicaciones comerciales que trabajan dando servicio a terminales remotos hay aplicaciones no interactivas para actualizar información, generar informes, etc., que funcionan en modo de lotes. En los sistemas Unix hay también programas de comprobación y actualización *(«crons»)* que se ejecutan periódicamente de forma no interactiva. Y, como ahora veremos, es la misma idea que se aplica en los sistemas interactivos con el nombre de «multitarea».

- **Sistemas interactivos**

Los sistemas interactivos tienen un requisito adicional: responder en un tiempo razonable a las peticiones del o de los usuarios. Algunos están diseñados para atender a un solo usuario (puede haber varios usuarios registrados, pero en un momento dado sólo uno hace uso del sistema). Y algunos de éstos son «monotarea», es decir, monoprogramación: el usuario pide la ejecución de un programa (que puede ser interactivo porque pide datos) y hasta que no termine de ejecutarse no puede ejecutarse otro. Así era MS-DOS, y así son algunos sistemas para aplicaciones especiales, como «BareMetal OS», concebido para nodos de altas prestaciones en una red en la que cada nodo realiza una sola tarea. Pero la mayoría de los sistemas actuales, aunque sean monousuario (como los de los teléfonos móviles y las tabletas), son **multitarea**.

La multitarea es una forma de multiprogramación adaptada a los requisitos de los sistemas interactivos. Tal como la hemos descrito, la multiprogramación exigiría cargar todos los programas previamente a su ejecución. Un programa podría estar horas utilizando la CPU antes de cederla por necesitar una operación de entrada/salida. En una aplicación por lotes esto es aceptable, pero no en un entorno interactivo: el usuario debe poder lanzar la ejecución de cualquier programa mientras los otros se ejecutan. Además, tiene que dar la impresión de que los distintos programas se ejecutan simultáneamente: se puede estar escribiendo en un procesador de textos al tiempo que otro programa reproduce música y otro está descargando un fichero de la red. Si la CPU sólo puede ejecutar en cada instante un programa, esto únicamente puede conseguirse repartiendo su uso a intervalos regulares (y muy pequeños) de tiempo entre los distintos programas, como ilustra la figura 2.3.

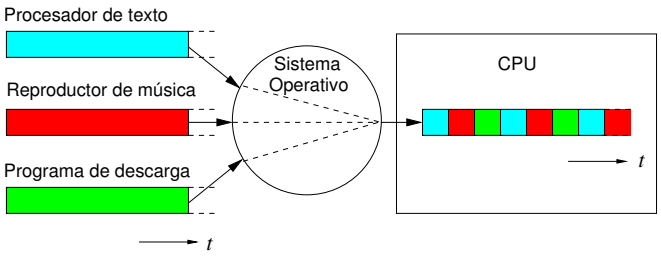



**Figura 2.3**: Multitarea con tres programas.

Hay dos formas de conseguir este comportamiento: en la **multitarea cooperativa** son los mismos programas los que periódica y voluntariamente, o bien cuando quedan bloqueados esperando una operación de entrada/salida, ceden el uso de la CPU, mientras que en la **multitarea apropiativa** *(preemptive)*[5] es el sistema operativo el que asigna en cada momento la CPU al programa que corresponda. Con la primera el sistema es más sencillo pero menos seguro: un programa mal diseñado puede bloquearlo todo. Se utilizaba en las versiones anteriores a Windows 95 y en las de Mac OS anteriores a OS X. Y se utiliza para sistemas que funcionan en entornos muy controlados, sin usuarios que puedan ejecutar programas arbitrariamente. Los demás sistemas actuales, incluyendo Android y

[5] «Apropiativa» es la traducción de «preemptive» más extendida (el sistema operativo *se apropia* de la CPU), aunque también se utilizan otras más literales como «preventiva» o «anticipativa».

iOS, funcionan con multitarea apropiativa.

La mayoría de los sistemas operativos de uso general actuales (las distintas versiones de Unix, Windows y OS X) son, además de multitarea, **multiusuario**. Esto significa que varios usuarios, de forma local o remota, pueden hacer uso de los recursos y ejecutar programas independientemente unos de otros mediante la técnica de **tiempo compartido** *(time sharing)*, que esencialmente es la misma de la multitarea.

- **Sistemas embebidos** *(embedded systems)* y **de tiempo real** *(RTOS: Real Time Operating Systems)*

  Se llama «embebido» a un sistema operativo (y a un hardware) diseñado para controlar el funcionamiento de un dispositivo que contiene otras partes mecánicas y eléctricas. Los incluidos en los teléfonos móviles son sistemas embebidos, pero también otros en los que la interacción con el usuario final es prácticamente nula o limitada a interfaces muy específicas: control de automóviles, electrodomésticos, etc.

  Como normalmente los recursos de hardware (velocidad de la CPU y capacidad de memoria) para estos sistemas son limitados, cuando no hay interacción significativa con el usuario y los programas están muy controlados se suele utilizar multitarea cooperativa.

  Hay versiones para estas aplicaciones de los sistemas operativos de uso general más conocidos. Por ejemplo, *Windows Embedded Standard*, utilizado en cajeros automáticos, terminales de punto de ventas, etc. TinyBSD es un subconjunto de freeBSD (un sistema operativo de tipo Unix, distinto de Linux y también libre). Y de Linux se han derivado varias distribuciones para sistemas embebidos: OpenEmbedded, LTIB, etc. La flexibilidad que ofrecen los sistemas de código abierto para su adaptación ha permitido a muchos fabricantes personalizarlos *(customize)* de manera muy específica para distintos dispositivos. Así, se encuentran implementaciones basadas en Linux para móviles (Android), televisores (webOS), *routers* (OpenWrt), navegadores GPS, etc.

  Los sistemas de **tiempo real** normalmente son también embebidos, pero tienen un requisito adicional: cumplir ciertas restricciones de tiempo de respuesta ante determinados estímulos. Esto es necesario en aplicaciones como control de máquinas, vehículos, instrumentos científicos e industriales, armamento bélico, etc.

### Distribución de los recursos hardware que gestiona el sistema

Los recursos de hardware a gestionar por un sistema operativo son la CPU, la memoria principal y los periféricos, incluyendo entre éstos a las memorias secundarias. En el apartado siguiente describiremos algunas características de estos recursos que es necesario conocer para comprender el funcionamiento del sistema operativo. Aquí nos detendremos en la distribución espacial de estos recursos.

- **Sistemas centralizados**

  Es el caso más familiar, en el que la CPU y la memoria están muy próximos y los periféricos normalmente se encuentran a poca distancia, aunque también pueden ser remotos, conectados a la unidad central mediante líneas dedicadas, una red local o internet.

  La **virtualización** es un concepto que desarrollaremos en el tema «Procesadores software» (apartado 12.6). Es la implementación sobre un determinado sistema operativo, o directamente sobre el hardware, de uno o varios sistemas operativos diferentes, que son sistemas virtuales (no «reales»). Esto tiene muchas aplicaciones. Por ejemplo, en un entorno personal uno puede trabajar habitualmente sobre Windows, pero esporádicamente necesitar alguna aplicación de Linux. Si tiene instalado sobre Windows el software de virtualización adecuado puede hacerlo sin necesidad de arrancar el ordenador. En un entorno corporativo es una manera de abordar la «consolidación de servidores»: reducir el número de servidores o ubicaciones de los servidores para optimizar el uso de los recursos.

- **Sistemas en red**

En una red de ordenadores cada una de las máquinas dispone de su propio sistema operativo independiente, pero estos sistemas tienen procedimientos de comunicación para que los usuarios y los programas puedan compartir recursos (ficheros, impresoras, etc.) conectados a la red.

Todos los sistemas operativos de uso general disponen de estos procedimientos[6]. En algunos sistemas embebidos (por ejemplo, en los *routers*) son esenciales.

- **Sistemas distribuidos**

En los sistemas distribuidos las máquinas también también están en red, pero con menos autonomía. Cada una dispone de una parte básica del sistema operativo (el *kernel* o el *microkernel*), y el resto de los componentes se encuentra en distintas máquinas. Para los usuarios es como si sólo existiese un sistema operativo, y la carga de trabajo de las aplicaciones se distribuye por las distintas máquinas.

Sus ámbitos de aplicación son mucho menores que las de los sistemas centralizados, y se han desarrollado principalmente en entornos académicos y de investigación. Pero estas investigaciones han facilitado el despliegue de otras arquitecturas, como el *grid computing* (combinación de máquinas en redes de área extensa para obtener capacidades similares a los supercomputadores) y la informática en la nube *(cloud computing)*, y ocasionalmente han dado lugar a otras aportaciones. Por ejemplo, el sistema «Amoeba» se diseñó a partir de 1980 por Andrew Tanenbaum en la Vrije Universiteit de Amsterdam y se abandonó en 2001. Pero el lenguaje Python, muy de actualidad, se creó inicialmente para este proyecto.

Uno de los pocos sistemas distribuidos que se comercializan es «Inferno». Es el sucesor de «Plan 9», que se desarrolló también en los años 1980 en Bell Labs. Desde 2000 se distribuye como software libre por una compañía británica, Vita Nuova, para *grid computing* y otras aplicaciones distribuidas.

- **Sistemas en la nube.**

Un entorno informático en la nube está basado en un gran número de recursos de hardware conectados a través de internet sobre el que se articulan tres tipos de servicios: de infraestructura *(IaaS: Infastructure as a Service)*, de plataforma *(PaaS: Platform as a Service)* y de aplicaciones *(SaaS: Software as a Service)*. El objetivo es suministrar bajo demanda recursos hardware *(IaaS)*, software de desarrollo *(PaaS)* y aplicaciones *(SaaS)* de manera flexible: el coste de los servicios está basado en el consumo que se realiza.

Los proveedores de *IaaS* más conocidos son Amazon EC2 *(Elastic Compute Cloud)* y *Google Compute Engine*. Ambos proporcionan almacenamiento, capacidad de proceso y sistemas operativos virtuales (Linux o Windows Server). El cliente puede elegir entre distintas configuraciones de CPU y memoria, y, en función de sus necesidades, demandar una o varias réplicas *(instances)* de las máquinas.

La gestión de los recursos físicos reales para poder servir a miles de peticiones es, naturalmente, muy compleja. El software que se ocupa de ella tiene funciones más elaboradas que las de un sistema distribuido tradicional: debe ser capaz de generar dinámica y rápidamente múltiples réplicas de las máquinas virtuales, ocuparse de la seguridad, de la contabilidad, etc. Este software se llama **gestor de infraestructura virtual** *(VIM: Virtual Infrastructure Manager)*, o también **sistema operativo de nube**.

[6] Si dispone usted de una red local de equipos Windows y desea integrar en ella un ordenador con Unix (Linux, OS X, *BSD...), puede instalar en éste «samba», un *middleware* que implementa los protocolos de Windows.

### Conclusión

A pesar su gran variedad, podemos identificar dos objetivos en todos los sistemas operativos:

- Junto con el resto de software del sistema, se trata de proporcionar a los programas de aplicación una *máquina virtual*, mediante gestión de los recursos de hardware: la CPU, las unidades de almacenamiento y los dispositivos periféricos. Estos recursos por sí solos son la «máquina real», y lo que permite el software del sistema es abstraer los detalles demasiado prolijos de su funcionamiento.
- Por otra parte, se trata de utilizar con *eficiencia* esos recursos desde el punto de vista global del conjunto de tareas a realizar. La eficiencia se mide por la *potencia* (número de trabajos realizados en una unidad de tiempo) y la *velocidad* (inversa del tiempo de ejecución de un programa), y depende tanto del hardware como de un buen diseño del sistema operativo. Dependiendo del uso del sistema es más importante una cosa o la otra.

En el resto de este capítulo vamos a considerar solamente sistemas centralizados, multitarea y multiusuario.

## 2.3. Recursos de hardware

### La CPU

La evolución de la tecnología ha hecho que el significado de esta sigla sea actualmente ambiguo. CPU *(Central Processing Unit)* es la expresión clásica para referirse al procesador hardware que ejecuta **instrucciones** (órdenes expresadas en lenguaje de máquina binario) de un programa almacenado en la memoria. El «corazón» de este procesador, la unidad de control, en un instante determinado sólo puede estar ejecutando una instrucción de un programa. Los **sistemas multiprocesador** contienen varios procesadores que funcionan en paralelo, pudiendo ejecutar varias instrucciones simultáneamente; actualmente en solo chip pueden integrarse varios procesadores. Algunos llaman «CPU» a cada uno de estos procesadores, y otros al conjunto. Aquí seguiremos el segundo convenio: la CPU es un conjunto de circuitos integrados en un chip que contiene uno o varios procesadores.

En el tema «Procesadores hardware» estudiaremos con todo detalle el funcionamiento de un procesador (capítulo 9) y veremos los principios de los sistemas multiprocesador (apartado 11.6). Para lo que sigue basta con saber dos cosas:

- Un procesador hardware extrae sucesivamente de la memoria principal las instrucciones que forman un programa y las va decodificando, interpretando y ejecutando.
- Un procesador hardware tiene, como mínimo, dos **modos de funcionamiento: modo supervisor y modo usuario**. Cuando está en modo usuario no permite ejecutar ciertas operaciones. Si un programa contiene una instrucción que implica alguna de esas operaciones y se ejecuta en modo usuario, al llegar a esa instrucción el procesador aborta la ejecución de ese programa y pasa al modo supervisor.

### Memorias

En el apartado 1.3 hemos visto las características básicas de los distintos soportes de almacenamiento. Concretando con datos, la tabla 2.1 resume las tecnologías de memoria de lectura y escritura más utilizadas. («Parcial» significa que la escritura es posible, pero a menor velocidad que la lectura).

| Tecnología | CMOS | SRAM[7] | DRAM | Flash NAND | Magnética | Óptica |
|---|---|---|---|---|---|---|
| Acceso aleat. | Sí | Sí | Sí | No | No | No |
| Lect. y escr. | Sí | Sí | Sí | Parcial | Sí | Parcial |
| T. acc. lect. ($\mu$s) | 0,0002–0,0005 | 0,0005–0,025 | 0,05–0,2 | 25–50 | 5.000–20.000 | > 100.000 |
| T. transf. (MB/s) | 20.000–100.000 | 5.000–50.000 | 1.000-5.000 | 200-500 | 500–1.000 | 1-5 |
| Coste (€/KiB) | $\approx 100$ | $\approx 10^{-2}$ | $\approx 10^{-5}$ | $\approx 10^{-6}$ | $\approx 10^{-7}$ | $\approx 10^{-7}$ |
| Dens. (bits/mm$^2$) | $\approx 10^2$ | $\approx 10^4$ | $\approx 10^6$ | $\approx 10^8$ | $\approx 10^9$ | $\approx 10^7$ |
| Volatilidad | Sí | Sí | Sí | No | No | No |
| Capacidad típica | < 1 KiB | < 64 MiB | < 16 GiB | < 1 TiB | < 10 TiB | < 100 GiB |
| Uso típico | Registros (M. local) | M. *cache* | M. principal | M. secundaria y terciaria | M. secundaria y terciaria | M. terciaria |
| Gestionado por | Hardware | Hardware | S. operativo | S. operativo/ Operador | S. operativo/ Operador | Operador |

**Tabla 2.1**: Comparación de tecnologías de memoria.

Lo importante no son los datos concretos, sino las proporciones entre las distintas tecnologías. Los datos son aproximados (y obtenidos de fuentes diversas), y evolucionan con el tiempo. Por ejemplo, hace unas décadas las tecnologías eran de componentes discretos para los registros, de núcleos de ferrita para la memoria principal y de discos más primitivos para la secundaria; si nos fijamos en la capacidad, un procesador típico podía tener ocho registros de 16 bits (es decir, en total $8 \times 16/8 = 16$ bytes), la memoria principal 1 MiB y el disco 500 MiB. Las proporciones son similares a las actuales. Esto explica que, aunque las tecnologías hayan evolucionado, los principios que se aplican para combinarlas sean esencialmente los mismos desarrollados hace 50 años.

## Memoria principal

La memoria principal es la que alberga los programas en ejecución, y para su implementación se hace necesario combinar varias tecnologías de memoria. Veamos por qué.

En el diseño de un sistema informático, como en el de cualquier sistema compuesto por partes, hay que adecuar entre sí las características de las partes para obtener el mejor funcionamiento posible con el menor coste. Ya sabemos que la CPU, si sólo tiene un procesador, va extrayendo (leyendo) sucesivamente instrucciones de la memoria y ejecutándolas (lo que puede implicar leer o escribir un dato en la misma memoria). Y esto lo hace a un ritmo constante. Para que el procesador no tenga que esperar, *la memoria principal debe ser obligatoriamente de acceso aleatorio* y, a ser posible, que tenga un tiempo de acceso compatible con la velocidad del procesador. Los procesadores más comunes actuales tienen velocidades del orden de 100 MIPS (y algunos llegan a 10.000 MIPS). Eso quiere decir que son capaces de leer y ejecutar 100 millones de instrucciones por segundo. Para proporcionar ese caudal y evitar así que el procesador espere, el tiempo de acceso de la memoria debe ser igual o inferior a $1/(100 \times 10^6)$ segundos, es decir, 0,01$\mu$s. Como vemos en la tabla 2.1, la tecnología adecuada sería la de SRAM, porque con una DRAM de 0,05$\mu$s sólo se llegaría a un caudal de $10^{-6}/(0{,}05 \times 10^{-6}) = 20$ MIPS.

Pero no hay que olvidar el factor coste[8], que depende de la capacidad de memoria necesaria. En un sistema con multiprogramación y de uso personal, actualmente es normal una capacidad de 4 GiB ($4 \times 2^{20}$ KiB) para albergar todos los programas en ejecución. De nuevo con los datos de la tabla, con

[7] Las SRAM *(Static Random Access Memory)*, DRAM *(Dynamic Random Access Memory)* y SDRAM *(Synchronous DRAM)* se implementan también con tecnología CMOS. La diferencia está en el número de transistores utilizados para un punto de memoria: uno en DRAM, varios en SRAM y muchos en los registros. De ahí las diferencias en densidad y coste.

[8] Tampoco hay que olvidar otros, como el tamaño, el consumo, el calentamiento... Si se consideran refuerzan aún más la argumentación.

SRAM resulta $0{,}01 \times 4 \times 2^{20} \approx 42.000$ €, un coste desorbitado en comparación con el de los demás componentes del sistema. Con DRAM es más razonable: 42 €.

Ahora bien, los patrones de ejecución de los programas tienen una propiedad que permite conseguir algo aparentemente imposible. Un programa puede contener decenas de miles de instrucciones, pero en el curso de su ejecución pasa largos períodos de tiempo ejecutando partes que sólo contienen unas decenas o centenas de instrucciones. El ardid consiste en conservar los programas en la memoria principal (DRAM), pero mantener copias de esas partes (una o varias de cada uno de los programas) en una memoria SRAM que es a la que realmente accede el procesador. Naturalmente, estas partes van cambiando, y tiene que controlarse mediante un hardware que para cada petición del procesador detecte si la parte que corresponde está copiada o no, y en caso necesario reemplazarla. En el tema «Procesadores hardware» (apartado 8.6) daremos más detalles.

De este modo, y siguiendo con el ejemplo, el procesador «ve» una memoria de 4 GiB, con un tiempo de acceso *medio* muy próximo al de la SRAM (algo inferior por la penalización que implica el que esporádicamente haya que sustituir alguna parte por otra) y con un coste sólo algo superior a de la DRAM (el de la SRAM y el hardware controlador). La memoria más rápida y más cercana al procesador, implementada actualmente con tecnología SRAM, se llama **memoria *cache***, o, simplemente, ***cache***[9]. Y como dentro de las tecnologías SRAM hay varias posibilidades de velocidad y coste, es normal que se combinen varios niveles: una pequeña *cache* (del orden de varios KiB), que es la más próxima al procesador (de hecho, suele estar integrada en el mismo chip) 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 memoria principal, el hardware controlador mira si lo que se pide está copiado en la L1, y generalmente resulta que sí, pero si no es así, el controlador accede a la L2 y, si aquí sí está se copia una parte en la L1, etc.

Por lo demás, tanto la memoria principal como la *cache* son volátiles, son de lectura y escritura y de acceso aleatorio y en ambas se puede acceder a un byte o a varios bytes (una «palabra») en una sola operación.

Se llama **espacio de direccionamiento** al conjunto de direcciones generado por un procesador al ejecutar un programa. Como luego veremos (página 45), no necesariamente coincide con la capacidad de la memoria principal.

### Memorias secundarias y terciarias

En la memoria secundaria se guardan todos los programas, estén en ejecución o no. Como pueden ser cientos de miles, se necesita una tecnología que permita una gran capacidad con un coste por bit reducido. Además, debe ser una tecnología de lectura y escritura y no volátil, para que los programas (y, por supuesto, los datos, que en muchas aplicaciones es lo verdaderamente importante a conservar) permanezcan sin necesidad de alimentación eléctrica. A la vista de nuevo de la tabla 2.1, la idónea es claramente la tecnología de discos magnéticos, aunque en algunos equipos se sustituyen por memorias *flash*[10] (son los llamados SSD, *Solid State Disks*).

La memoria terciaria es la de los soportes de almacenamiento que no están conectados permanentemente y que precisan de una intervención humana (insertar un CD o un «pincho» USB), o bien de un sistema robótico (en grandes archivos digitales). La tecnología más común para grandes volúmenes de datos es la de las cintas magnéticas.

[9] Una traducción obvia es «memoria oculta». O, más literal y más sugestivamente, «zulo». Pero el anglicismo está tan extendido, no sólo en la jerga técnica, también en el lenguaje cotidiano, que es mejor conservarlo.

[10] También, en algunos sistemas embebidos se utiliza *flash* de tipo NOR, que es de lectura y escritura y no volátil, para la memoria principal. Pero aquí nos estamos centrando en ordenadores de propósito general.

Para comprender el trabajo que tiene que realizar el sistema operativo a fin de ofrecer a los programas y a los usuarios una abstracción de los detalles del almacenamiento en las memorias secundarias es necesario conocer, aunque sea superficialmente, estos detalles. Sólo describiremos las cintas y discos magnéticos, sin entrar en otras tecnologías, como las ópticas o las *flash*.

## Cintas magnéticas

El almacenamiento en cintas magnéticas se basa en el mismo principio de las cintas de audio y vídeo: la magnetización de una película depositada sobre una banda delgada de material plástico, que se desplaza bajo una o varias cabezas de lectura y escritura. La diferencia es que no se graban señales analógicas, sino digitales: el campo magnético en cada punto de la superficie está orientado en un sentido o en el opuesto, y puede hacerse corresponder a cada sentido un valor binario, o, lo que es más frecuente (porque facilita el diseño de los circuitos detectores), a cada cambio de sentido un valor binario.

Las cintas «clásicas» (las que todavía pueden verse en algunos «centros de proceso de datos», y, sobre todo, en películas) tenían una anchura de media pulgada (1,27 cm) y una longitud variable, del orden de cientos de metros. Los datos se almacenaban en nueve **pistas** paralelas: ocho para un byte y la novena para un bit de paridad.

El **formato** de almacenamiento es el que muestra la figura 2.4. Los bytes se agrupan en **registros físicos**[11] y entre registro y registro se deja un intervalo de varios milímetros. En unos formatos los registros tienen un número fijo de bytes, y en otros el número es variable. En todo caso, el controlador reconoce en la lectura al intervalo como una zona sin cambios en la magnetización. Los intervalos son necesarios porque la cinta puede pararse al terminar de leer o escribir un registro (nunca en medio de uno), y cuando vuelve a arrancar tarda un tiempo en llegar a la «velocidad de crucero», por la inercia de las partes mecánicas. Los registros se agrupan en **ficheros**. Una codificación se reserva para la «marca» que indica el principio de un fichero, a la que le sigue el primer registro, que contiene, entre otros datos, el nombre del fichero. Se puede así localizar con relativa facilidad un fichero determinado: avanzando la cinta rápidamente hasta encontrar una marca de inicio de fichero, y leyendo su primer registro.

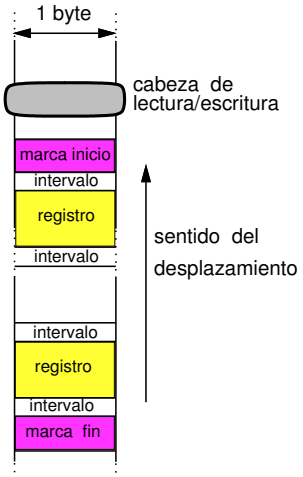



**Figura 2.4**: Formato de una cinta magnética.

Este formato primitivo ha ido evolucionando. Actualmente se utiliza un estándar complejo, el LTFS *(Linear Tape File System)*.

La capacidad de almacenamiento de las cintas más comunes actualmente es del orden de 10 TiB[12] y su tasa de transferencia, 160 MB/s. Su gran inconveniente radica en la naturaleza secuencial del acceso: acceder a un registro que se encuentra al final de la cinta cuando las cabezas están al principio puede requerir minutos. Con los discos magnéticos se consiguen capacidades y tasas de transferencia similares a las de las cintas, y los tiempos de acceso se reducen a los milisegundos; al evolucionar su tecnología se ha ido reduciendo su coste, y, así, han ido sustituyendo a las cintas. Éstas, no obstante, compiten con los discos y con tecnologías ópticas para el almacenamiento de **copias de respaldo** y de **archivos digitales**.

[11] Aunque en español se les dé el mismo nombre, no hay que confundir estos «registros» *(records)* con los registros de la CPU *(registers)*.

[12] Varios fabricantes han anunciado en 2015 cintas con capacidades superiores a 200 TiB sin compresión.

## Discos magnéticos

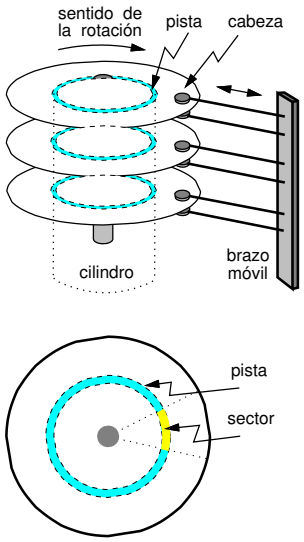



**Figura 2.5**: Disco magnético.

Los primeros discos duros comercializados para ordenadores personales, hace unos treinta años, tenían una capacidad de unos 8 MiB y un coste de unos 400 € . Actualmente (2015) se pueden encontrar discos de 8 TiB por el mismo precio. Es decir, la capacidad se ha multiplicado, aproximadamente, por un millón, y el el coste por bit se ha dividido por la misma cantidad. Sin embargo, los principios básicos de funcionamiento no se han alterado sustancialmente.

La configuración mecánica básica de una unidad de disco consta de un conjunto de discos recubiertos de un material magnetizable, que giran sobre un eje común. Cada disco tiene dos **superficies** (o **caras**), por lo que hay dos cabezas de lectura y escritura para cada disco. La organización del almacenamiento se basa en la división de la superficie en **pistas** concéntricas. Un conjunto de pistas de igual radio en todas las superficies se llama **cilindro**. El conjunto de las cabezas puede moverse solidariamente en sentido radial, de modo que en cada momento todas las cabezas están situadas sobre un determinado cilindro (figura 2.5). Las pistas se dividen en **sectores**, y cada sector contiene un conjunto de bytes.

El tamaño típico de un sector ha sido durante muchos años 512 bytes. Recientemente (desde 2010) se va imponiendo un nuevo estándar llamado *«Advanced Format»* con sectores de 4.096 bytes.

El número de cilindros, cabezas y sectores por pista, «**CHS**» (*Cylinders–Heads–Sectors*), determinan la llamada **geometría física** del disco. Para acceder a un determinado sector, el software debería comunicar al hardware controlador del disco los números de cilindro, de la superficie y del sector dentro de la pista.

Ahora bien, si la velocidad angular de rotación, el número de sectores por pista y el número de bytes por sector son constantes, como las pistas tienen distinto radio, la densidad de grabación en las pistas externas es menor que en las internas. Se puede aprovechar mejor la superficie magnética (y dotar de más capacidad al disco) haciendo que el número de sectores por pista sea mayor cuanto más externa sea la pista. Esta técnica, que desde el año 2000 utilizan todos los fabricantes de discos (y también de DVD) se llama «**ZBR**» *(Zone Bit Recording)*. El software puede acceder a un sector especificando simplemente su número «absoluto», entre 0 y $N - 1$, siendo $N$ el número total de sectores que puede contener el disco. El controlador se ocupa de hacer la traducción para averiguar en qué pista y superficie se encuentra. Esto se llama «**LBA**» *(Logical Block Addressing)*[13].

Los discos no tienen un modo de acceso aleatorio (como la memoria principal) ni secuencial (como las cintas), sino **cíclico**. Hay que distinguir entre el **tiempo de búsqueda** *(seek time)* que tardan las cabezas de lectura/escritura en desplazarse hasta la pista adecuada (y que depende de las características electromecánicas del sistema que mueve las cabezas) y el **tiempo de espera** (o «latencia»: *rotational latency*) que tarda el sector en llegar hasta donde está la cabeza (que depende de la velocidad de rotación del disco); la suma de ambos es, en valores medios, del orden de milisegundos. Sin embargo, una vez posicionada la cabeza, el flujo de lectura o escritura de datos puede proceder con una tasa de transferencia del orden de cientos de MB/s.

[13] ZBR y LBA tienen un efecto secundario: la tasa de transferencia es mayor en las pistas externas. Esto explica un fenómeno que si es usted observador quizás haya notado: un disco nuevo parece hacerse más lento conforme se utiliza. Es así porque en un disco vacío el sistema operativo empieza utilizando los sectores por su número, y estos sectores se numeran de afuera adentro. A medida que el disco se va llenando se va accediendo más a sectores de menor radio.

El tiempo de búsqueda tiene, para cada disco, unos valores máximo y mínimo bien determinados (lo que tardan las cabezas en desplazarse desde el cilindro más externo hasta el más interno, y lo que tardan en desplazarse entre dos cilindros contiguos), pero el valor medio es difícil de establecer (depende del programa: de cómo esté organizado el almacenamiento de los datos en el disco y de cómo sea el acceso a esos datos). Los valores medios anunciados por los fabricantes varían entre 4 y 15 ms. El tiempo medio de espera, sin embargo, es fácil de calcular: es lo que tarda el disco en dar media vuelta. Para uno que gire a 10.000 rpm (revoluciones por minuto) puede usted comprobar que resulta ser 3 milisegundos.

La tasa de transferencia depende de la velocidad de rotación y de la densidad de grabación[14]. En los discos actuales puede llegar a 1 GB/s. Pero esta es la tasa de transferencia «interna». Los controladores normalmente tienen una memoria *buffer*, y si los datos que se pretenden leer ya están en ella la transferencia es mucho más rápida. La transferencia externa, con la interfaz estándar SATA, puede llegar a 3 GB/s.

## Jerarquía de memorias

Todos estos elementos de almacenamiento forman una **jerarquía de memorias** (figura 2.6). La **brecha de velocidades** *(speed gap)* entre los procesadores y la memoria principal y la que hay entre eśta y la memoria secundaria ha permanecido durante décadas a pesar de la evolución de las tecnologías. La primera puede resolverse mediante hardware, como hemos visto, con las *caches*, pero para la segunda no hay una tecnología disponible. Un candidato actualmente es *flash*, pero sigue habiendo una brecha (menor) y tiene algunos inconvenientes (ser más lenta en escritura y tener un número máximo alto pero limitado de escrituras). La solución es mediante software: una ***cache* de disco**, que consiste en tener copiadas en zonas de la memoria principal las partes más utilizadas del disco.

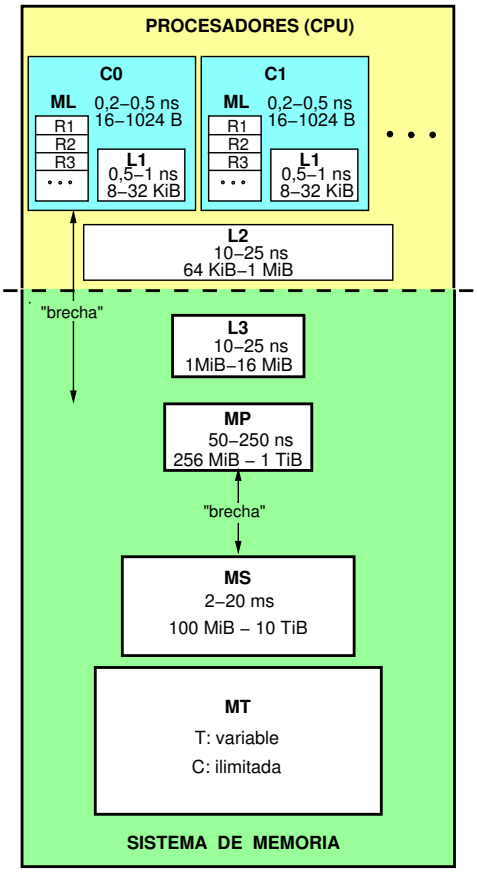



**Figura 2.6**: Jerarquía de memorias.

Podemos hacer una analogía entre esta jerarquía de memorias en un ordenador y la que se utiliza en ciertos modos de organización del trabajo intelectual. Por ejemplo, en el proceso de escribir un documento como éste es preciso consultar muchas fuentes: anotaciones y esquemas propios, artículos, libros, manuales, etc. En un momento dado, el autor siempre tiene rápidamente accesibles (en un tiempo, digamos, del orden de un segundo), al lado del teclado, las notas más relevantes. A veces debe hacer consultas rápidas (y en su caso renovar las notas de acuerdo con el resultado) en documentos que también están «a mano» (apilados en papeles, o en ficheros de trabajo) cuyo acceso puede requerir del orden de diez o veinte segundos. Esporádicamente ha de levantarse para buscar algo en una estantería próxima, o hacer una búsqueda

[14]En principio es fácil de calcular, pero para ficheros grandes es complicado: la lectura o la escritura puede necesitar el acceso a varios o a muchos sectores que pueden estar en distintas pistas. Una medida es IOPS *(Input/output Operations Per Second)*, que se obtiene experimentalmente comprobando el tiempo de transferencia de un conjunto de pequeños bloques de tamaño aleatorio.

en internet, lo que puede llevar varios minutos. En ocasiones necesita algo que ha de buscar en la biblioteca; en tal caso cambia de tarea (como en multiprogramación) y se dedica a otra cosa: el acceso a la biblioteca puede dejarse para dentro de unas horas. Finalmente, es posible que tenga que recurrir a solicitar un envío por correo, que, si es por medios tradicionales, puede tardar semanas o...

Los papeles al lado del teclado son análogos a los registros de los procesadores, los documentos «a mano» o en ficheros a la *cache*, los de la estantería o la búsqueda a la memoria principal, la biblioteca a los discos «en línea», y la información por la que tiene que esperar meses a la almacenada en un disco que hay que «montar» manualmente[15].

### Otros dispositivos periféricos

Hay un gran variedad de periféricos, no solamente en su funcionalidad, sino, por lo que respecta a sus conexiones con la CPU y la memoria, en su tasa de transferencia. La tabla 2.2 resume algunos datos, sólo para tener una idea de esa variedad.

| | **de entrada** | **de salida** | **de entrada y salida** |
|---|---|---|---|
| **de comunicación con personas** | teclado: <10 B/s<br>ratón: <20 B/s<br>micrófono: 20 KB/s<br>escáner: 100 KB/s<br>cámara *web*: 200 KB/s | impresora láser: 500 KB/s<br>altavoz: 20 KB/s<br>pantalla gráfica: 100 MB/s<br>GPU: 1 GB/s | pantalla táctil:<br>30 MB/s (salida)<br><20 B/s (entrada) |
| **de comunicación con artefactos** | ADC *(Analog to Digital Converter)*:<br>1 KB/s – 100 KB/s | DAC *(Digital to Analog Converter)*:<br>1 KB/s – 100 KB/s | xDSL: 32 KB/s – 6.600 KB/s<br>Ethernet:<br>10 MB/s – 100 MB/s |
| **de almacenamiento** | CD: X×150 KB/s<br>DVD: X×1.250 KB/s | | discos y cintas magnéticos:<br>200 MB/s – 1 GB/s |

**Tabla 2.2**: Tasas de transferencia típicas de algunos periféricos.

Las comunicaciones entre los componentes de hardware se realizan por medio 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. Hay *buses paralelos*, en los que se transmite un flujo de $n$ bits simultáneamente por $n$ líneas y *buses seriales*, en los que se transmite un flujo de bits en secuencia (USB), o $2n$ flujos de bits, si el bus es bidireccional y tiene $n$ enlaces (PCIe). En cualquier caso, un bus es un recurso compartido: en cada momento, sólo una de las unidades puede depositar un bit o un conjunto de bits en el bus, pero varias pueden leerlo simultáneamente. La propiedad principal que caracteriza a un bus es la tasa de transferencia (o ancho de banda, o velocidad): el número máximo de bits (o bytes) por segundo que puede transferir. Naturalmente, la velocidad del bus debe ser igual o superior a la tasa de transferencia de los componentes que se conectan a él.

Las transferencias entre la CPU y la memoria principal requieren un bus de alta velocidad, el **bus de memoria**. Los periféricos con tasas de transferencia elevadas, como los discos y las unidades de procesamiento gráfico *(GPU: Graphical Processing Units)* se adaptan también a la velocidad de este

[15] Como curiosidad, las relaciones de tiempos en esta analogía se conservan bastante bien con un factor de escala de $10^9$ (es decir, a un segundo «humano» le corresponde un nanosegundo «electrónico»).

bus y acceden directamente a la memoria principal *(DMA: Direct Memory Access)*. Pero la mayoría de periféricos tienen tasas de transferencia más reducidas, y pueden compartir un bus más lento, como muestra la figura 2.7.

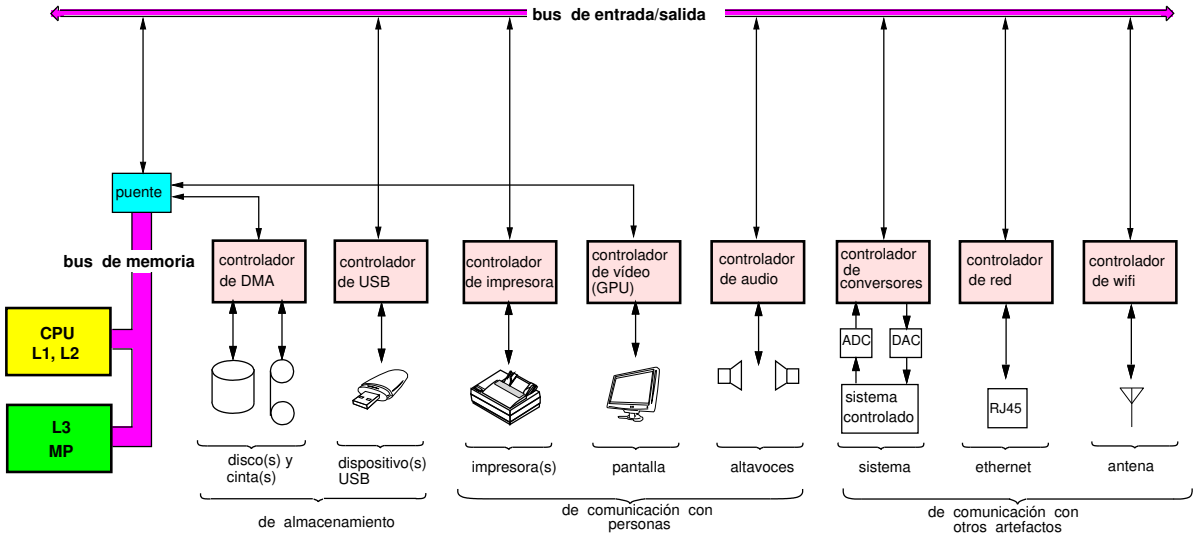



**Figura 2.7**: Conexión de periféricos mediante buses.

Todo periférico está acompañado de un hardware específico para él, el **controlador del periférico**, que interpreta las señales de control que recibe del bus para realizar la operación que corresponda (escribir un carácter en la pantalla, emitir un sonido, etc.) y genera señales de petición (leer un carácter del teclado, mover el cursor, etc.)

La figura 2.7 muestra una situación simplificada. En efecto, como entre los periféricos sigue habiendo una gran variedad, existe también una variedad de buses que normalmente se conectan formado una jerarquía en lugar de un único bus de entrada/salida. La figura 2.8 es un esquema del hardware de un ordenador personal actual. La CPU en este ejemplo contiene cuatro procesadores *(cores)* (C0-C3), cada uno de ellos con dos *caches* (Li, Ld), y una *cache* de nivel 2 común (L2). Los puentes *(bridges)* norte y sur forman parte del *«chipset»* del fabricante de la CPU, y los buses del sistema y de memoria son específicos de ese fabricante («propietarios»). Los demás buses son estándares: PCI *(Peripheral Component Interconnect)*, PCIe *(PCI Express)*, USB *(Universal Serial Bus)*, SATA *(Serial Advanced Technology Attachment)* y LPC *(Low Pin Count)*.

## 2.4. Servicios del sistema operativo (modelo funcional)

Un sistema operativo proporciona servicios a los programas e, indirectamente, a los usuarios, a través de interfaces: operaciones primitivas de comunicación. Los distintos sistemas operativos proporcionan estos servicios de maneras diferentes. En lo sucesivo, para concretar, nos referiremos a sistemas de tipo Unix *(Unix-like)* y «monolíticos» (en el sentido que explicaremos en el apartado 2.7). Dentro de esta categoría hay sistemas libres (Linux, freeBSD, openBSD, etc.) y privativos (AIX, HP-UX, Solaris, etc.) y, aunque no son idénticos, lo que aquí veremos es común a todos.

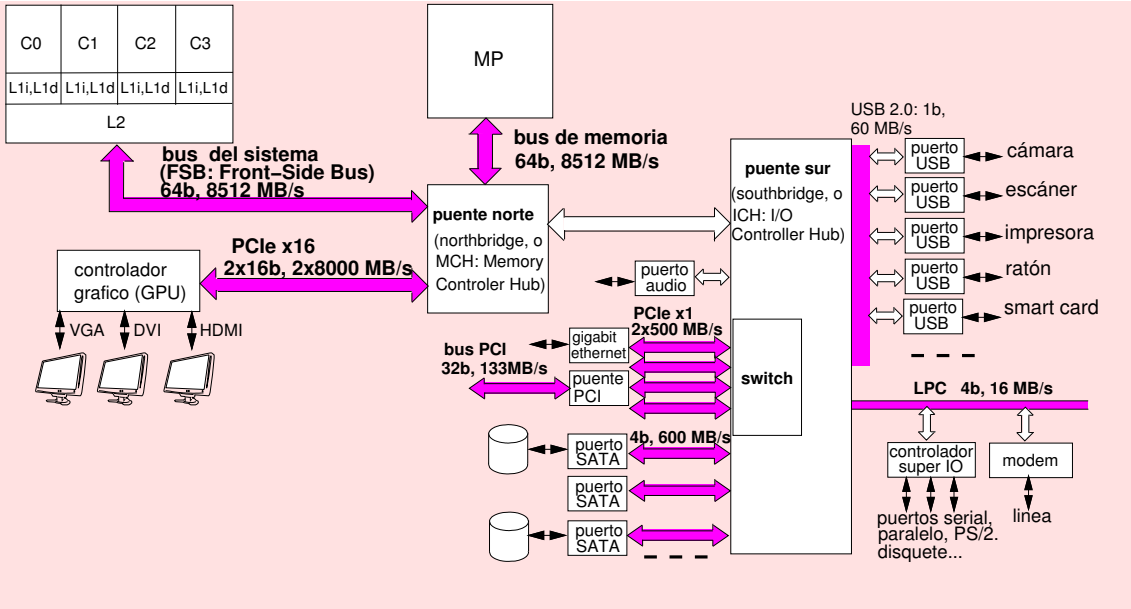



**Figura 2.8**: Buses en un ordenador personal.

## Interfaces: SCI, GUI, CLI, API y ABI

La *verdadera* interfaz de los sistemas operativos que estamos estudiando está formada por las **llamadas al sistema**. Son solicitudes que pueden hacerse desde un programa al sistema siguiendo unos convenios sobre el nombre de la petición y sus parámetros. El conjunto de las llamadas constituye la **SCI** *(System Call Interface)*.

Las interfaces para el usuario se implementan con software del sistema, mediante programas intérpretes. Usted ya habrá trabajado con alguna interfaz gráfica (**GUI**: *Graphical User Interface*), y en las prácticas de laboratorio tendrá ocasión de hacerlo con una interfaz de **línea de órdenes** (o **línea de comandos, CLI**: *Command Line Interface*) implementada con un programa **intérprete de órdenes**, o ***shell***. La *shell* interpreta directamente algunas órdenes *(builtin commands)*, pero la mayoría de las órdenes son en realidad nombres de programas del sistema que la *shell* invoca.

Los programas pueden hacer uso directamente de la SCI, pero las facilidades que ofrece son muy básicas y su uso engorroso. Por eso, los programas de aplicación, que normalmente se escriben en algún lenguaje de alto nivel, no hacen un uso directo de las llamadas al sistema, sino por medio de un conjunto de programas de utilidad que proporcionan funciones de un nivel similar al de estos lenguajes. Este conjunto se llama biblioteca (o librería), y ofrece otra interfaz a las aplicaciones, la **API** *(Application Programming Interface)*. Una llamada a una función de la API puede generar una o varias llamadas al sistema[16]. En sistemas Unix la biblioteca que ofrece la API se implementa en lenguaje C y se llama `libc` (o `glibc`). POSIX *(Portable Operating System Interface)* es un estándar del IEEE que define los nombres y convenios de las funciones de la API para Unix.

La ventaja de utilizar la API en lugar de la SCI, aparte de la mayor facilidad de programación, es que proporciona una abstracción del sistema operativo, lo que permite desarrollar programas más portables. Hay versiones de sistemas operativos que no son Unix pero que siguen el estándar POSIX; un mismo programa fuente sirve para cualquiera de estos sistemas, sin necesidad de adaptarlo.

[16]O ninguna, porque también contiene funciones de uso general. Por ejemplo, «atoi», que convierte una cadena de caracteres numéricos a su representación como número entero (el nombre proviene de *«ascii to integer»*).

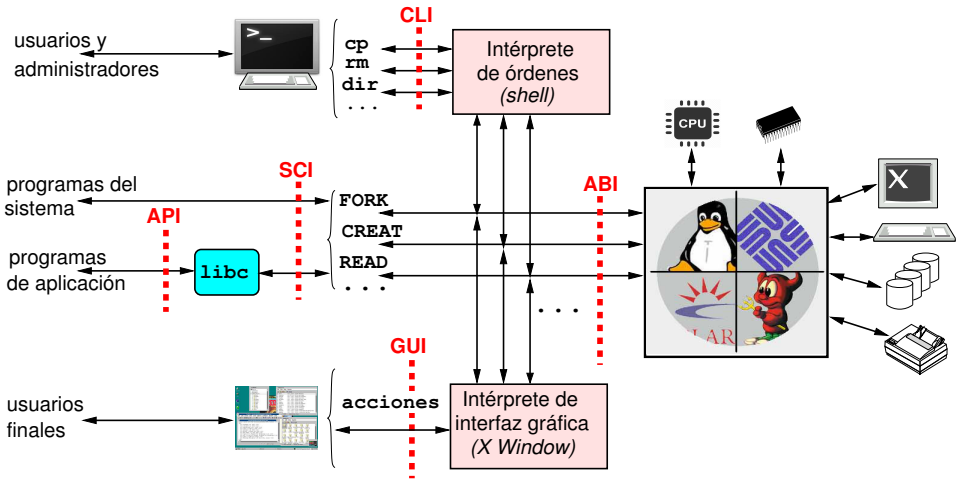



**Figura 2.9**: Interfaces de un sistema Unix.

La figura 2.9 resume las relaciones entre la SCI, la API y las interfaces de usuario[17]. Como estamos hablando del sistema operativo, veremos algunas ideas sobre la primera y no comentaremos nada más sobre las otras. Pero aún falta decir algo sobre la «ABI».

La SCI define operaciones primitivas de manera general, independientemente de los detalles de la CPU. Por ejemplo, `WRITE` es una llamada para escribir datos en un periférico; como luego veremos, tiene que ir acompañada de parámetros que localizan los datos en la memoria, el periférico y, si es de almacenamiento, la zona donde escribir, etc. Al ejecutarse en un programa esta llamada, esos parámetros (y también un número que identifica a `WRITE`) se introducen en registros de un procesador, el programa que se está ejecutando deja de hacerlo y se pasa a ejecutar el sistema operativo, que al leer esos registros sabe lo que se le está pidiendo. Pero los distintos procesadores (ARM, x86, MIPS, etc.) tienen distintos registros y distintas maneras de acceder a ellos. Por tanto, esa llamada, escrita de manera simbólica, se traduce de manera diferente para cada CPU. La **ABI** *(Application Binary Interface)* es específica: define las llamadas en función de las características del procesador o los procesadores.

Resumiendo, la SCI define las llamadas en lenguaje de alto nivel, mientras que la ABI las define en el nivel de código máquina, como muestra la figura 2.10 para el caso de la llamada `WRITE`. La API contiene una función genérica, de nombre también `write`, y otras particulares, como, por ejemplo, `printf`, que sirve para escribir por la salida estándar cadenas de caracteres y valores de variables (puede consultar los detalles en un terminal de Unix escribiendo `man 3 write` y `man 3 printf`: la sección 3 de las páginas de manual contiene las funciones de la API, y la 2 las llamadas), y genera una o varias `WRITE` (`man 2 write`)[18].

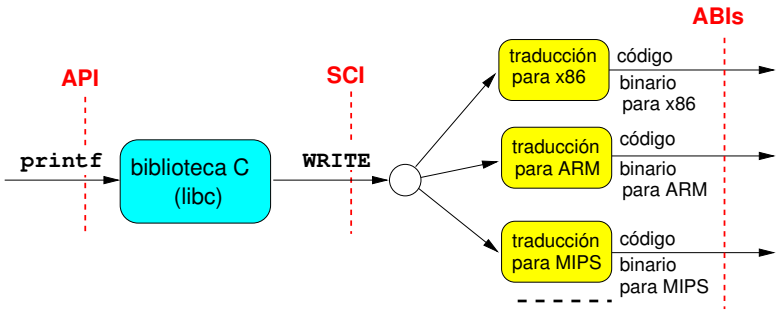



**Figura 2.10**: API, SCI y ABI.

[17] Para no complicar más la figura se han omitido algunos detalles. Por ejemplo, los intérpretes y otros programas del sistema también recurren a veces a las funciones proporcionadas por `libc`.

[18] Normalmente los nombres de las llamadas se escriben con minúsculas; aquí lo hacemos con mayúsculas para resaltar que son llamadas al sistema, no funciones de biblioteca.

## Sistema de ficheros

Una de las abstracciones más importantes que proporciona el sistema operativo es la de **fichero**. Hay ficheros «regulares» y ficheros «especiales».

- **Ficheros regulares**

  Un fichero «regular», también conocido como «archivo», es justamente aquello en lo que pensamos cuando se menciona «fichero»: unos datos en binario almacenados en memoria secundaria e identificados con un nombre. En Unix es, simplemente, una secuencia de bytes. Su contenido puede representar cualquier cosa: un programa binario ejecutable, una sinfonía, un vídeo, un texto... Para el sistema operativo sólo es una secuencia de bytes.

  Todo fichero tiene un nombre único en el sistema, pero éste puede albergar cientos de miles o millones de ficheros en memoria secundaria. Un programa, durante su ejecución, sólo trabajará con un número limitado. Para que desde ese programa se pueda operar con un fichero es necesario «abrirlo» previamente (con la llamada `OPEN` que veremos luego). Al hacerlo, el sistema devuelve un pequeño número entero positivo: el **descriptor del fichero**[19], que lo identifica para todas las operaciones y crea espacio en una estructura de datos, la *tabla de ficheros abiertos* (página 51). Hay tres descriptores predefinidos de ficheros que no es necesario abrir: 1 (la «entrada estándar»), 2 (la «salida estándar») y 3 (la «salida de error estándar»).

  Además del descriptor (que no cambia), los ficheros abiertos (salvo las entradas y salidas estándar) tienen asociado un número entero (que va cambiando conforme se lee del fichero o se escribe en él) que llamaremos **posición** *(offset)*. En efecto, si un fichero tiene 100 bytes, no es lo mismo escribir 10 bytes desde el principio (posición = 0), que sobrescribe los 10 primeros bytes, que desde la posición 100, que los añade al final.

  Hay varios tipos de ficheros que no son regulares: ficheros de dispositivos, directorios, enlaces simbólicos, *sockets*, *pipes*, etc. Veamos algo sobre los tres primeros.

- **Ficheros especiales: dispositivos**

  Los dispositivos periféricos se identifican con un nombre, y en Unix se integran en el sistema de ficheros. Por ejemplo, «`tty`» identifica a un terminal, y «`sda`» a un disco (que normalmente contendrá ficheros regulares).

  Hay dos tipos de ficheros de dispositivo: de bloques y de caracteres. Los primeros designan a periféricos rápidos, en los que se transfieren bloques de cientos o miles de bytes entre el controlador del periférico y la memoria principal. En un disco, por ejemplo, un bloque es un sector o un pequeño número de sectores que contiene entre 512 y 4.096 bytes[20]. El sistema de gestión de ficheros (una parte del sistema operativo, página 48) «ve» el fichero correspondiente a un disco como un conjunto de bloques accesibles de manera «aleatoria» (es decir, dando simplemente el número del bloque). Se puede tener el fichero especial «`sda1`» (partición 1 del disco `sda`), formado, por ejemplo, por $N = 10^9$ bloques de direcciones 0 a $N - 1$, e, independientemente, el usuario puede definir ficheros ordinarios sobre el mismo disco: el fichero «`carta_a_los_reyes`» podría ocupar los bloques 400,

[19] Llamarlo «descriptor» no parece muy adecuado (no «describe»). Pero se puede entender en un sentido amplio la segunda acepción de la R.A.E.: «Término o símbolo válido y formalizado que se emplea para representar inequívocamente los conceptos de un documento o de una búsqueda». En inglés es «a word o expression etc. used to describe or identify» (Concise Oxford).

[20] POSIX especifica 512, pero actualmente, con los discos de «Advanced Format» es más frecuente que el bloque coincida con el sector: 4.096 bytes. El tamaño y número de sectores del disco `sda` y el tamaño de bloque que usa el sistema se pueden ver con las órdenes (que requieren permisos de superusuario) «`fdisk -l /dev/sda`» y «`blockdev --getbsz /dev/sda`» respectivamente.

520 y 2.700.000 de esos mismos disco y partición.

Los ficheros especiales de caracteres identifican a periféricos como teclado, ratón, etc., en los que no tiene sentido el acceso arbitrario a un dato de un conjunto, y sólo cabe considerar un flujo secuencial de bytes que se transfieren entre el controlador y la CPU (en un periférico de este tipo no podemos leer o escribir «el byte número 600», o «el bloque número 400»).

- **Ficheros especiales: directorios**

Un **directorio** no contiene datos. Es una tabla con referencias a otros ficheros, que a su vez pueden ser directorios; éstos son subdirectorios (o «hijos») del directorio. Todo parte de un **directorio raíz**, que no tiene ningún «padre». Esto da lugar a una estructura jerárquica del sistema de ficheros que se puede representar en forma de árbol. Un árbol típico de Unix es el de la figura 2.11, en la que se representan los directorios con rectángulos y los ficheros que no son directorios (y que son las hojas del árbol) con óvalos. El nombre del directorio raíz es «/».

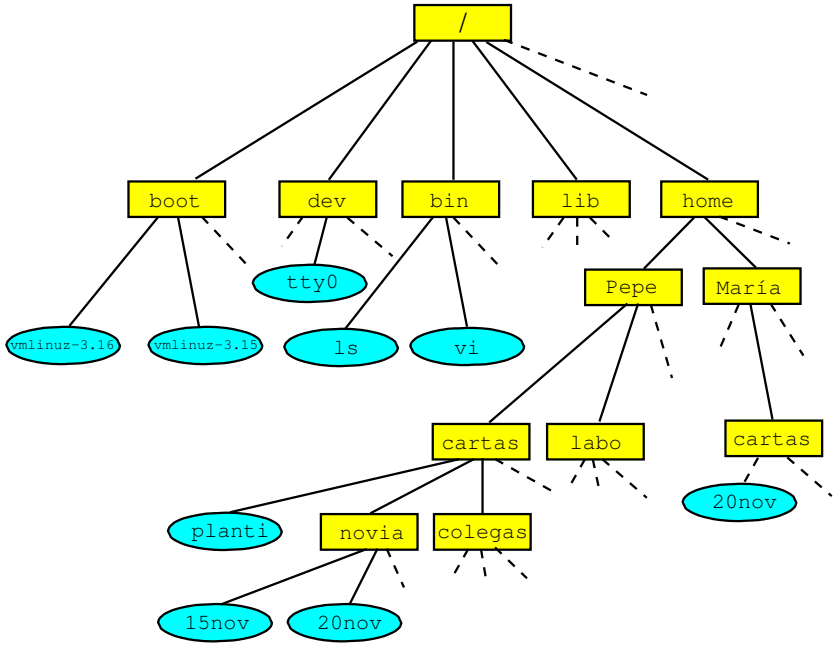



**Figura 2.11**: Árbol de directorios.

El convenio para denominar simbólica e inequívocamente a un fichero o a un directorio consiste en utilizar sucesivamente el símbolo «/» para determinar su **ruta** *(path)* absoluta desde el directorio raíz. En el ejemplo de la figura 2.11 hay dos directorios llamados «cartas» y dos ficheros distintos llamados «20nov»: uno «cuelga» del directorio `novia`, y otro de `cartas`, subdirectorio de `María`. El nombre del primero es `/home/Pepe/cartas/novia/20nov`, y el del segundo `/home/María/cartas/20nov`. Estas son **rutas absolutas**, pero también se pueden utilizar **rutas relativas** desde un directorio determinado. Así, María, desde su directorio personal (`/home/María/`) puede acceder a su fichero `20nov` con el nombre relativo `cartas/20nov`.

- **Ficheros especiales: enlaces simbólicos**

A veces es útil la posibilidad de definir «alias» para los nombres de los ficheros. Por ejemplo, en un sistema multiusuario resulta conveniente la facultad de que dos o más usuarios puedan *compartir* un mismo fichero y que cada uno lo «vea» como si estuviese en su directorio y con el nombre que desee.

Una forma sencilla y eficaz de hacerlo es creando un **enlace simbólico** *(symbolic link)* o «acceso directo». No es más que una entrada de un directorio que define un nombre de fichero sinónimo de otro que está en otra ruta[21].

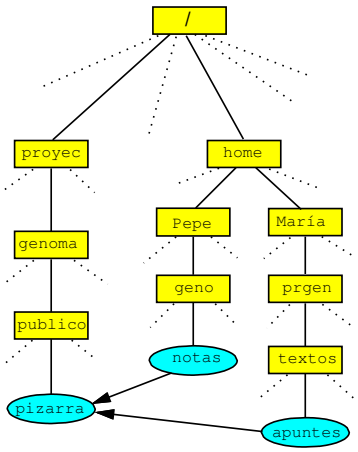



**Figura 2.12**: Grafo con enlaces simbólicos.

Por ejemplo, si existe un fichero cuya ruta absoluta es «`/proyec/genoma/publico/pizarra`» y el usuario Pepe quiere referirse a él con el nombre «`notas`» (dentro, quizás, de alguno de sus propios directorios), puede definir un enlace entre este nombre y el nombre completo, y a partir de ahí operar con el fichero como si estuviese en su directorio. Y del mismo modo, el usuario María puede conocerlo con otro nombre. El árbol de directorios deja de ser tal para convertirse en un *grafo acíclico*, como ilustra la figura 2.12.

## Montaje

Si el sistema contiene varios discos (y/o varias particiones en un disco), cada uno con su propio conjunto de ficheros y directorios y con un directorio raíz (`/`), se pueden «montar» unos árboles sobre otros de modo que el conjunto se vea como un sistema de ficheros único y los usuarios y los programas de aplicación no tengan necesidad de saber en qué disco se encuentra cada fichero. La figura 2.13 muestra el resultado de montar el disco `sdb` (que suponemos contiene los directorios de los usuarios) sobre el directorio `/home` del disco `sda`, que es el *punto de montaje*. Una vez montado, todos los ficheros y directorios aparecen en un árbol único, y el hecho de que unos estén físicamente en un periférico y otros en otro resulta transparente para los usuarios y las aplicaciones.

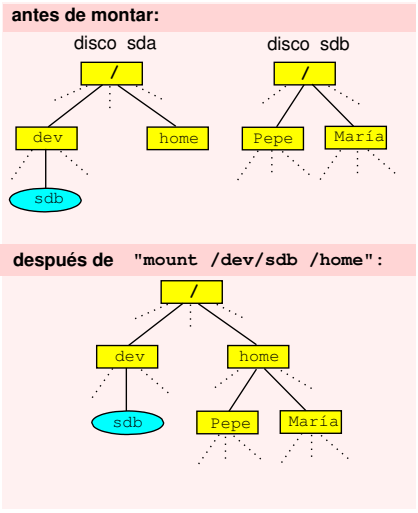



**Figura 2.13**: Montaje.

La operación de montaje es especialmente útil para los soportes extraíbles: memorias *flash*, CD, DVD, etc. Hay programas de utilidad que, cuando se inserta el soporte, realizan el montaje automáticamente en algún punto de montaje previsto (por ejemplo, `/media`).

## Permisos de acceso

En el laboratorio practicará usted el manejo del sistema de ficheros desde la interfaz de línea de comandos (CLI). Verá entonces que todo fichero tiene asociados datos sobre su creador o propietario (**uid**), el grupo de usuarios al que pertenece (**gid**) y los **permisos** que tienen el propietario, los miembros del grupo y los demás usuarios para leer, escribir o ejecutar el contenido del fichero. Así, si el árbol de directorios de la figura 2.11 fuese real, para ver las propiedades del fichero `20nov` de `Pepe` utilizaría la orden `ls` :

```
$ls -l /home/Pepe/cartas/novia/20nov
-rw-r----- 1 Pepe alumnos 3641 dic 20 09:26 /home/Pepe/cartas/novia/20nov
```

[21] Hay también «enlaces duros» *(hard links)*. Son diferentes porque no son referencias a un nombre de fichero, sino al fichero mismo. Podrá usted entender esta diferencia cuando comprenda lo que es un *inode* (página 49): un enlace simbólico es un fichero porque tiene su propio *inode* que apunta al *inode* del fichero original, mientras que uno duro no tiene *inode* y apunta directamente al original. Desde el punto de vista del uso, una de las diferencias es que si se mueve el fichero a otro directorio el enlace duro sigue funcionando, pero el simbólico no.

La primera línea es la orden, y la segunda la respuesta. El primer carácter (`-`) indica que es un fichero regular (`d` sería un directorio, `b` un dispositivo de bloques, etc.) A continuación, `rw-` significa que su propietario (`Pepe`) tiene permisos para leer y escribir pero no para ejecutar, los usuarios de su grupo (`alumnos`) tienen permiso sólo para leer (`r--`), y el resto de usuarios no puede hacer nada (`---`). También se muestra el tamaño del fichero en bytes y la fecha de modificación.

Si hacemos lo mismo con el de María y vemos:

```
$ls -l /home/María/cartas/20nov
-rwx------ 1 María alumnos 18435 dic 30 18:32
```

la diferencia en cuanto a permisos es que María no permite la lectura a los usuarios de su grupo. Seguramente María se ha equivocado, porque le ha puesto permiso de ejecución (`x`) y si efectivamente es una carta será un fichero de texto, no ejecutable.

En el laboratorio verá usted cómo se pueden modificar los permisos con la orden `chmod` y practicará muchas otras órdenes de la CLI. Cada vez que da una, el intérprete genera una o varias llamadas (normalmente, muchas). Veamos brevemente algunas (SCI).

### Llamadas para el sistema de ficheros

Las llamadas relacionadas con el sistema de ficheros son más de 60. Las más comunes son:

`OPEN`, que sirve para abrir un fichero.

La especificación (o «prototipo») de `OPEN` (puede verla completa con `man 2 open`) es:

```
int open(char *nombre, int flags);
```

Esto quiere decir que los parámetros que hay que pasar en la llamada son el nombre del fichero como cadena de caracteres y un entero (`flags`) que codifica si se abre sólo para escribir, o sólo para leer, o para las dos cosas, si se quiere acceder a él desde el primer byte (posición = 0) o desde el último (posición = *n*), etc. El sistema devuelve al programa el descriptor del fichero (o, como en todas, -1 se se produce un error).

Es importante observar que la operación de abrir un fichero no significa copiarlo en la memoria principal (al contrario de la operación de ejecución que veremos luego). El fichero sigue estando en memoria secundaria. Pero ésta puede contener miles o millones de ficheros, mientras que un programa sólo trabaja con un número reducido. Y hay datos sobre el fichero que sólo tienen sentido cuando se está trabajando con él. Por ejemplo, la posición. La apertura del fichero sólo implica generar un descriptor e introducirlo junto con esos datos (incluidos los «flags») como una fila en la *tabla de ficheros abiertos*, que ya hemos mencionado y que describiremos en el apartado 2.5 (página 51). Cuando el fichero se cierra se borra esa fila de la tabla.

`CREAT`, que crea un fichero nuevo:

```
int creat(char *nombre, int modo);
```

En «modo» se indican los permisos de acceso. Aparte de un nombre imperfecto[22], algunos la consideran obsoleta, porque lo mismo se puede hacer con

```
int open(char *nombre, int flags, int modo);
```

[22] A Ken Thompson, uno de los creadores de Unix, le preguntaron qué cambiaría si pudiese rediseñarlo, a lo que respondió «escribiría creat con una e al final». Parece que el motivo fue una limitación en los nombres de los símbolos del software que se utilizaba en el PDP-11 (el ordenador en el que se hicieron los primeros desarrollos).

`CLOSE`, para cerrar el fichero y liberar espacio en la tabla de ficheros abiertos. Devuelve 0 si tiene éxito, y -1 si hay algún error:

```
int close(int fd);
```

`fd` es el descriptor del fichero.

`READ`, para leer de un fichero:

```
int read(int fd, void *buf, int num);
```

`*buf` es la dirección de memoria («puntero») de la zona *(buffer)* en la que han de introducirse los bytes a leer del fichero identificado por `fd`, y `num` es el número de bytes a leer. El sistema devuelve el número de bytes realmente escritos (puede ser menor que `num` si se llega al final del fichero). El valor de «posición» se incrementa en el número de bytes leídos.

`WRITE`, para escribir en un fichero:

```
int write(int fd, void *buf, int num);
```

`*buf` es el puntero de la zona en la que se encuentran los bytes a escribir en el fichero identificado por `fd`, y `num` es el número de bytes a escribir. El sistema devuelve el número de bytes realmente escritos (puede ser menor si no queda espacio en el dispositivo de almacenamiento, o el sistema interrumpe la ejecución por alguna causa). El valor de «posición» se incrementa en el número de bytes escritos.

`LSEEK`, para modificar el valor de «posición»:

```
int lseek(int fd, int numero, int donde);
```

«`donde`» indica lo que se hace con «`numero`»: si se hace posición = número, o posición = posición actual + número, o posición = posición del final + número.

`STAT`, para obtener información de un fichero:

```
int stat(char *nombre, struct *buf);
```

Analiza el fichero `nombre` y devuelve los datos sobre él en `buf`: permisos, identificadores del propietario *(uid)* y del grupo *(gid)*, tamaño en bytes, tiempos del último acceso y última modificación, etc.

`IOCTL`, para controlar ficheros especiales de dispositivos:

```
int stat(int fd, int request,...);
```

Los parámetros («request») dependen de cada periférico particular.

`SOCKET`, que crea un punto de conexión de red y devuelve un descriptor:

```
int socket(int dominio, int tipo, int protocolo);
```

Junto con otras llamadas (`BIND`, `ACCEPT`, `SEND`, etc.) permite establecer y manejar las **interfaces de red**.

## Procesos y hebras

Para entender los servicios del sistema que se refieren a la carga y ejecución de programas es necesario recurrir a otra abstracción fundamental en los sistemas con multiprogramación: la de **proceso**.

Si la CPU sólo contiene un procesador, para hacer posible la multiprogramación el sistema operativo tiene que decidir en cada momento qué programa hace uso del procesador (y, por tanto, qué programas están «esperando»). Además de un procesador, un programa en ejecución necesita otros recursos: memoria, ficheros, etc. Si en un momento dado el sistema operativo suspende su ejecución para ceder el procesador a otro programa, es preciso conservar el estado de la ejecución (fundamentalmente, en qué instrucción se ha detenido y los valores de las variables que está utilizando) a fin de que cuando «le toque» pueda seguir su ejecución normalmente. Un programa es algo *estático:* una serie de instrucciones almacenadas en un fichero. Un proceso es *dinámico:* el programa más el estado de la ejecución, que se inicia cuando el programa se carga en memoria, va cambiando conforme se ejecuta y «muere» cuando termina la ejecución.

Como veremos en el apartado 2.6, cuando el sistema arranca se crean una serie de procesos para ejecutar programas. Entre ellos, un intérprete de órdenes para cada uno de los usuarios conectados. Si en uno de los terminales se da, por ejemplo, la orden «`cp fich1 fich2`» (copiar el contenido del fichero `fich1` en el fichero `fich2`), lo que se está pidiendo es la ejecución de un programa que se encuentra en `/bin/cp`. Esto da lugar a un nuevo proceso, que es «hijo» del proceso desde el que se ha pedido (el intérprete de órdenes). No es el caso de `cp`, pero el proceso creado podría a su vez crear nuevos procesos, que serían sus hijos, y así, en general, tendremos un árbol de procesos. Todo proceso «vivo» tiene asignado un número entero que lo identifica, el **pid**. Además, tiene un **uid** y un **gid** que normalmente son los que identifican al usuario que lo ha creado y que determinan los permisos de acceso. Más adelante, al entrar en los aspectos de seguridad, veremos que en realidad hay dos **uid** y dos **gid**.

El procedimiento para pedir la ejecución de programas en Unix consiste en crear primero el proceso y luego cargar el programa en la zona de memoria que se le ha asignado y comenzar a ejecutarlo.

En la asignatura «Algoritmos y estructuras de datos» ha estudiado usted el concepto de **hebra** *(thread)*, que hace posible la programación concurrente con programas de aplicación, concretamente en Java.

Las hebras son parecidas a los procesos: comparten el uso del procesador, pueden estar activas, preparadas o bloqueadas, pueden tener hijos... Pero, a diferencia de los procesos, que son independientes, un grupo de hebras existe dentro de un proceso y comparten con él la memoria y los ficheros abiertos.

No todos los sistemas operativos permiten crear y manejar hebras, y los que lo hacen utilizan distintos métodos (en Linux, por ejemplo, hay llamadas como `clone`, `exit_group` o `tkill` relacionadas con las hebras). Por este motivo, los programas, para ser portables, no utilizan directamente las hebras del sistema, sino a través de la API, que implementa funciones estandarizadas en POSIX: las *«pthreads»*.

## Llamadas para crear y terminar procesos y ejecutar programas

Las llamadas más importantes relacionadas con los procesos son `FORK`, `EXEC`, `WAIT` y `EXIT`.

`FORK` crea un proceso:

```
int fork();
```

Esta llamada crea un nuevo proceso que en principio ejecuta el mismo programa que ha hecho la llamada y que hereda del padre los identificadores de usuario y grupo y los descriptores de ficheros abiertos. El sistema operativo devuelve al proceso padre un número entero, el identificador del proceso hijo. Pero al programa del proceso hijo le devuelve 0.

Observe que a partir del momento en que el sistema operativo ejecuta `FORK` coexisten dos procesos idénticos (además de los que ya hubiese antes), con sendas copias del programa y de las variables en la memoria, ejecutándose en multitarea: el sistema irá cediendo a uno y otro el uso del procesador (o de uno de los procesadores). La única diferencia es que la variable *pid* en el proceso padre contiene el identificador del hijo, y en éste contiene 0. Normalmente queremos que el proceso hijo haga alguna otra cosa, por lo que el programa contendrá algo así:

```
pid = fork;
if (pid == 0) then {...}; /* lo que deba hacer el hijo */
else {...}; /* seguir ejecutando el programa del padre */
```

Con frecuencia, lo que se quiere es que el hijo ejecute otro programa, y para eso está `EXEC`.

`EXEC` es una familia de llamadas. La más utilizada es:

```
int execve(char *nombre, char argv[], char envp[]);
```

Los parámetros son el nombre del fichero que contiene el programa a ejecutar y argumentos y variables que se le pueden pasar a este programa. Si la llamada no genera ningún error, es decir, si tiene éxito, entonces no se devuelve nada al programa que la incluye, porque este programa queda sustituido por el nuevo en la memoria asignada al proceso. El proceso sigue siendo el mismo, con el mismo *pid*, y el programa hereda los descriptores de ficheros abiertos.

`EXIT` hace terminar al proceso en el que se está ejecutando:

```
void exit(int status);
```

En la variable «status» el proceso puede dejar una información para su padre sobre cómo ha terminado: normalmente, con error, etc..

`WAIT` **y** `WAITPID` hacen que el proceso espere a que termine alguno de sus hijos o un hijo determinado:

```
int wait(int *status);
```

Si este proceso tiene algún hijo que ya ha terminado el sistema devuelve el *pid* de este hijo y el proceso continúa. «`*status`» es un puntero a la variable donde el hijo ha dejado la condición de terminación. Pero si ninguno de sus hijos ha terminado el proceso queda bloqueado hasta que lo haga uno de ellos.

En esta otra:

```
int waitpid(int pid, int *status);
```

el proceso queda en espera (bloqueado), si es necesario, hasta que termine el hijo identificado por el primer parámetro (`pid`)

`KILL` envía una «**señal**» a otro proceso u otros procesos:

```
int kill(int pid, int sig);
```

Si `pid` $> 0$ la señal se envía al proceso pid. Si `pid` $= 0$, a todos los procesos del grupo del proceso actual (es decir, todos los de la «familia»: padres e hijos). Si `pid` $= -1$, a todos los procesos.

En `sig` se codifica la señal. Por defecto, la acción es «matar» al proceso (o los procesos), y de ahí su nombre, pero hay otros tipos de señales (un par de páginas más adelante comentaremos algo más sobre las señales).

La figura 2.14 puede ser útil para fijar las ideas. Después de la `FORK` el proceso hijo recién creado ejecuta el mismo programa que el padre, pero en el instante *t2* este programa hace una llamada `EXEC`. Observe que, aunque los programas en ambos procesos son idénticos, esta `EXEC` se ejecuta en el hijo, y no en el padre, por el procedimiento que hemos visto antes basado en los diferentes valores de *pid* que el sistema ha devuelto a los dos procesos. A partir de *t2* continúa el mismo proceso hijo, pero ejecutando otro programa. Entre *t1* y *t3* padre e hijo siguen ejecutándose «simultáneamente» (si lo hacen sobre el mismo procesador, el sistema operativo se encargará de ir cediéndolo alternativamente a uno y otro), pero en *t3* el proceso padre hace una llamada `WAIT`, y como el hijo aún no ha terminado, queda bloqueado. Cuando el hijo termina, en *t4*, el padre vuelve a estar activo (o, más precisamente, como veremos en el apartado 2.6, «preparado»).

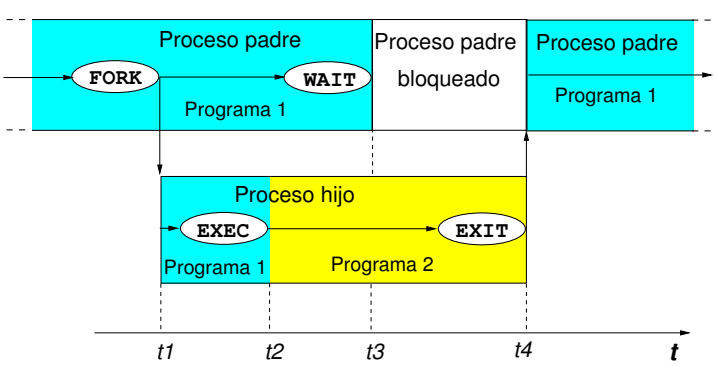



**Figura 2.14**: Ejecución de procesos padre e hijo.

## Otras llamadas

Las llamadas descritas son suficientes para comprender la descripción del funcionamiento del sistema que haremos en el apartado 2.6. Otras, cuyas funciones se pueden consultar en la sección 2 de las páginas de manual, son:

- Para la gestión de la memoria: `brk`, `sbrk`, `mmap`...
- Para la comunicación entre procesos, un asunto que veremos después de saber algo más sobre los procesos, en el apartado 2.6 (página 66): `pipe`, `msgsnd`, `msgrcv`...
- Para el mantenimiento de información: `getpid`, `getcwd`, `settimeofday`...
- Para la protección: `chmod`, `umask`, `chown`...

## Cómo «ver» las llamadas: `strace`

Un ejercicio interesante, si quiere usted comprobar la generación de llamadas desde un programa, es utilizar el programa de utilidad `strace`. Por ejemplo, como tendrá ocasión de practicar en el laboratorio, `/usr/bin/cp` es un programa para copiar un fichero en otro. Si desde el intérprete de órdenes escribe «`cp fich1 fich2`» se copiará el contenido de `fich1` (que tiene que existir) en `fich2` (que si no existe se crea, y si existe se sobreescribe). Si la orden la precede de `strace` con la opción de escribir el resultado (que es muy largo) en un fichero, es decir:

```
strace -o kkk.log cp fich1 fich2
```

la secuencia de llamadas queda registrada en el fichero `kkk.log`. Examinándolo verá que empieza con `EXECVE` para cargar `/usr/bin/cp`[23], luego hay una serie de llamadas para comprobaciones de

[23] No aparece `fork` porque `cp` no crea directamente el proceso, sino a través de una función de la API. Las llamadas a las funciones de biblioteca se pueden ver con otra utilidad: `ltrace`.

permisos, gestionar zonas de memoria, etc., y al final podrá ver las `OPEN`, `READ`, `WRITE` y `CLOSE` que hacen la operación pedida. Dependiendo del tamaño de `fich1` y del de la zona intermedia de memoria *(buffer)* que utilice el sistema bastará con una sola pareja de `READ` y `WRITE` o serán necesarias varias.

## Otros servicios

Además de la atención a las llamadas, el sistema operativo proporciona otros servicios:

- **Memoria virtual**
  Más allá de las llamadas, el principal servicio que proporciona un sistema operativo multiusuario y multitarea es el de ofrecer a los usuarios y a los procesos la impresión de disponer cada uno de una *máquina virtual* (página 22). Esto implica repartir equitativamente entre ellos el tiempo del procesador (o los procesadores), lo que se consigue con las técnicas de *multiprogramación* (página 18), pero también es necesario repartir la memoria de modo que cada proceso pueda ejecutarse como si fuese el único en la máquina. De este modo, los programas se escriben sin necesidad de tener en cuenta en qué parte de la memoria principal van a cargarse el código y los datos; cuando se crea un proceso el sistema se encarga de asignarle espacio libre. Además, la capacidad de la memoria principal es limitada y no puede albergar a cientos o miles de procesos, por lo que el sistema se encarga también de hacer uso de la memoria secundaria de manera transparente a los procesos. De este servicio de *memoria virtual* se ocupa el subsistema de gestión de memoria con técnicas que veremos en el apartado 2.5.
- **Demonios**
  Los demonios son programas que se ejecutan como procesos «en la sombra» o «entre bastidores» *(background)*, no bajo demanda de los usuarios. Normalmente se inician al arrancar el sistema y permanecen inactivos hasta que ocurre algún evento o hasta que pasa un tiempo. Por ejemplo, `syslogd` se activa cuando aparecen eventos, sean errores o no, y los registra en un fichero (`/var/log/syslog`); `crond` se «despierta» periódicamente (cada hora, o cada día, o cada semana...) para ejecutar tareas de mantenimiento programadas por el administrador del sistema; los servicios de red se activan al detectar eventos en las interfaces de red, etc.
  Pero esos ejemplos, muy importantes para el uso del sistema operativo, no son realmente servicios del sistema operativo, porque están implementados con software ajeno al sistema (tal como venimos considerando a éste: definido por «lo que hay detrás» de la SCI). Algunos demonios sí forman parte del sistema operativo. Por ejemplo, el que se ocupa periódicamente de vigilar el uso de la memoria y, si es necesario, intercambiar zonas con la memoria secundaria (página 46), o el planificador, que decide a qué proceso se le cede la CPU (página 45)[24].
- **Detección de errores y señales**
  En la ejecución de todas las llamadas (salvo `EXIT` y `EXEC` sin error) el sistema devuelve un número entero positivo, o cero. Pero si se ha producido algún error o violación de permisos lo que devuelve es $-1$, y, para que el programa pueda tomar la acción oportuna, pone un código que identifica al error en una variable llamada «errno» (accesible desde la biblioteca `libc`).
  Ahora bien, hay errores ajenos a las llamadas, sean de software (por ejemplo, intento de división por cero) o de hardware (por ejemplo, mal funcionamiento de un periférico). Para tratarlos están las **señales**, que son la inversa de las llamadas: no se dirigen del proceso al sistema operativo, sino al revés. Las llamadas son *síncronas* (si un programa se ejecuta repetidamente siempre

[24] Puede verse cuáles son con la orden `ps aux`, que lista todos los procesos. En la columna «VSZ» (o «SIZE») se muestra la memoria asignada a cada proceso. Aquellos en los que este valor es «0» son demonios del sistema, porque su memoria está incluida en la asignada al sistema operativo.

se generarán en el mismo punto del programa), pero las señales son *asíncronas* (un fallo del hardware aparece aleatoriamente, y la división por cero puede aparecer o no según los datos que maneje el programa).

Algunas de las señales que recibe un proceso están generadas por otro proceso. Por ejemplo, las procedentes de las llamadas `KILL` y `EXIT`, ya explicadas. Pero la mayoría las genera el sistema. En los sistemas de tipo Unix se definen unas 30 señales diferentes (pueden consultarse sus números y sus nombres simbólicos con «`man 7 signal`»). Cuando un proceso recibe una señal puede pedirle al sistema que ejecute un programa que el programador ha previsto para esa señal, llamado **manipulador de la señal** *(signal handler)*, o, lo más frecuente, dejar que el sistema ejecute un manipulador por defecto, que normalmente no hace más que abortar al proceso.

- **Contabilidad**

  La contabilidad de procesos consiste en registrar en ficheros (ficheros de *log*) los detalles de los procesos que se van ejecutando: programas que se cargan, usuario, uso de recursos, tiempo de CPU, etc., así como errores en la ejecución y eventos procedentes de la periferia. Hay programas de utilidad para extraer la información acumulada y presentar estadísticas: tiempo de uso de la CPU para cada usuario, programas que ha ejecutado, etc.

  Además de su utilidad para la **gestión administrativa** del sistema, una aplicación importante de la contabilidad es el **análisis forense**: tras un ataque a un sistema, la actividad registrada puede proporcionar muchas evidencias sobre las causas, los daños producidos y la identidad del atacante.

  Un tipo especial de contabilidad es la **contabilidad IP**, esencial para los proveedores de servicios de internet *(ISP: Internet Service Providers)*: el registro de las actividades de red permite facturar a los usuarios por uso de recursos. Igualmente importante son esos registros para tomar decisiones sobre ampliaciones de equipamiento (memoria, discos, servidores...) basadas en estadísticas de los distintos servicios ofrecidos.

- **Protección y seguridad**

  La diferencia entre protección y seguridad es consecuencia de un principio de diseño de los sistemas software: el de la **separación de mecanismo y política**. Se trata de que los mecanismos que controlan la autorización de ciertas operaciones y la asignación de recursos no deben restringir las políticas que definen esa autorización y esa asignación[25].

  El sistema operativo y el hardware proporcionan algunos mecanismos de protección para controlar el acceso de usuarios y procesos a los recursos. Las políticas de seguridad se implementan con otros programas del sistema.

  Los mecanismos de protección básicos del hardware son:

  - Los procesadores hardware tienen varios **modos de funcionamiento**, como ya adelantamos en el apartado 2.3 (página 22). Hay operaciones (como modificar el registro que contiene el modo de funcionamiento, o ejecutar ciertas instrucciones llamadas «privilegiadas») que el procesador sólo admite si está en modo supervisor. Si se intentan realizar en modo usuario el procesador detiene el proceso y, como veremos en el apartado 2.6, avisa al sistema operativo.
  - Los procesadores emiten una señal que indica el modo en el que se encuentran. El hardware externo puede vigilarla para detectar operaciones prohibidas. Por ejemplo, los procesos de

[25] Un ejemplo cotidiano lo aclara muy bien: el acceso a un local a través de una puerta con cerradura. El mecanismo clásico es el de la llave tradicional. Si se quiere cambiar la política de acceso, es decir, que a partir de un momento ciertas personas no puedan entrar, hay que cambiar la cerradura y las llaves. Es un mecanismo que restringe fuertemente la política. Un mecanismo con menos restricciones es el del acceso mediante DNI electrónico. El cambio de política es mucho más flexible: basta modificar la base de datos.

usuario tienen asignada una zona de memoria. Cuando piden un acceso para leer o escribir, la *unidad de gestión de memoria* (un subsistema hardware controlado por el *sistema de gestión de memoria*, página 47) comprueba que el acceso es correcto (dentro de su zona), y en caso contrario avisa al sistema operativo.

Los mecanismos de protección mínimos que se implementan en el sistema operativo son:

- Todo usuario registrado en el sistema está identificado por un *uid* y tiene una «cuenta» que incluye un registro en un fichero de contraseñas, `/etc/passwd` (donde, además de la contraseña cifrada[26], se guardan su nombre, *uid*, *gid*, nombre completo y datos de contacto), así como un subdirectorio de `/home` en el que puede libremente definir subdirectorios y ficheros y decidir sus permisos[27]. En toda operación solicitada por un proceso de usuario el sistema comprueba que es compatible con los permisos de los recursos.
- Como política general de seguridad se adopta el **principio de mínimos privilegios**: a cualquier agente (usuario o proceso) sólo se le permitirá acceder a la información o los recursos necesarios para sus objetivos legítimos. Así, los usuarios «normales» no tienen capacidad, por ejemplo, para leer, escribir o borrar ficheros de otros usuarios ni del sistema si los permisos lo impiden. Tampoco pueden ejecutar ciertos programas de utilidad como dar de alta o de baja a usuarios, cambiar la hora, instalar programas fuera de su directorio, apagar el sistema, etc. Pero estas actividades son necesarias para la administración del sistema. Hay un usuario especial, el que tiene *uid* = 0 y nombre normalmente «root»[28], que puede realizar cualquier operación sobre ficheros y procesos, independientemente de los permisos. Se le conoce, aparte de «root», como **superusuario** o **administrador**.
- Hay ocasiones en las que es necesario permitir una **escalada de privilegios** a fin de que un usuario obtenga permisos de *root* para alguna actividad de manera controlada. Un ejemplo es el cambio de contraseña. El programa que permite hacerlo es `/usr/bin/passdwd`, propiedad de *root*, que accede al fichero de contraseñas, protegido contra escritura por usuarios normales. Pero a estos usuarios hay que permitirles cambiar *su* contraseña. Para hacerlo posible, además de los permisos, los ficheros ejecutables pueden tener puesto un bit que se llama *setuid*:

  ```
  $ls -l /usr/bin/passwd
  -rwsr-xr-x 1 root root 54192 nov 20 23:03 /usr/bin/passwd*
  ```

  «`rws`» (y no «`rwx`») en los permisos del propietario indica que *setuid* está puesto. Esto hace que cuando el programa se ejecuta lo haga con *permiso real* del usuario que lo invoca, pero con *permiso efectivo* de superusuario *(root)*. De hecho, el sistema asigna a todo proceso cuatro números: UID real, UID efectivo, GID real y GID efectivo. Los UID y GID reales se usan para fines contables; los UID y GID efectivos para determinar permisos de acceso. Salvo en casos como éste, los números reales y efectivos son los mismos. Esta idea, acompañada de llamadas para cambiar los UID y GID (`setuid`, `setreuid`, etc.) facilitan el diseño de ciertos programas de utilidad de forma segura. Otro ejemplo es el de la escalada de privilegios para ejecutar programas de administración de manera controlada. El programa `su` sirve para hacerse pasar por otro usuario, particularmente *root*. Pero exige conocer la contraseña de *root*. Es más seguro y flexible `sudo`, que permite a los usuarios ejecutar ciertos programas.

[26] En muchos sistemas, para más seguridad, las contraseñas se guardan en un fichero asociado, `/etc/shadow`, que sólo puede leer el superusuario. El fichero `/etc/passwd` debe tener permisos de lectura para todos porque algunas utilidades lo necesitan para asociar el *uid* con el nombre del usuario.

[27] Se pueden definir «cuotas» para fijar un límite de uso de almacenamiento para los usuarios.

[28] El nombre podría ser cualquier otro. Se acostumbra llamar así porque al principio el directorio personal del superusuario era la raíz del árbol de directorios, `/` (actualmente es `/root`).

El administrador determina en un fichero (`/etc/sudoers`) qué usuarios y a qué programas pueden acceder, y el programa deja además un registro de las actividades.

- Todos los sistemas operativos de uso general incorporan los programas o módulos necesarios para el acceso a la red, y entre ellos algún procedimiento para inspeccionar las cabeceras de los paquetes. Sobre este procedimiento se implementan **cortafuegos** *(firewalls)* para filtrar el tráfico, que pueden formar parte o no del sistema operativo. Por ejemplo, *IPFilter* es un cortafuegos para varios sistemas operativos: freeBSD, netBSD, Solaris, AIX, etc. Linux contiene un conjunto de módulos, *netfilter*, en el que se apoyan cortafuegos que no forman parte del sistema operativo, como *iptables*.
- Los métodos «clásicos» de control de acceso basados en permisos, contraseñas y excepciones como *setuid*, son insuficientes en entornos corporativos o gubernamentales que exigen un grado muy alto de seguridad. Hay un conjunto de recomendaciones y estándares que establecen un **control de acceso imperativo, MAC** *(Mandatory Access Control)*, implementado como extensión de muchos sistemas operativos: SELinux, TrustedBSD (FreeBSD y OS X), Trusted Solaris, Mandatory Integrity Control (Windows), etc.[29]

## 2.5. Componentes del sistema operativo (modelo estructural)

Para proporcionar todos esos servicios, el sistema operativo es un sistema software complejo formado por miles de programas y estructuras de datos. Concretamente, la versión 3.16 de Linux contiene más de 47.000 ficheros y más de 12 millones de líneas de código (la mayoría en lenguaje C)[30].

A grandes rasgos, y atendiendo a los recursos que se gestionan, se suelen distinguir los cuatro subsistemas que indica la figura 2.15. Es una simplificación, porque faltan componentes importantes, como la gestión de red (que puede considerarse incluida en la gestión de periféricos o en la de ficheros), o la protección (que puede suponerse implícita en todos los bloques).

Veamos las funciones de cada subsistema y las principales estructuras de datos en las que se apoyan para cumplir estas funciones, sin entrar en los detalles del funcionamiento, objeto del siguiente apartado.

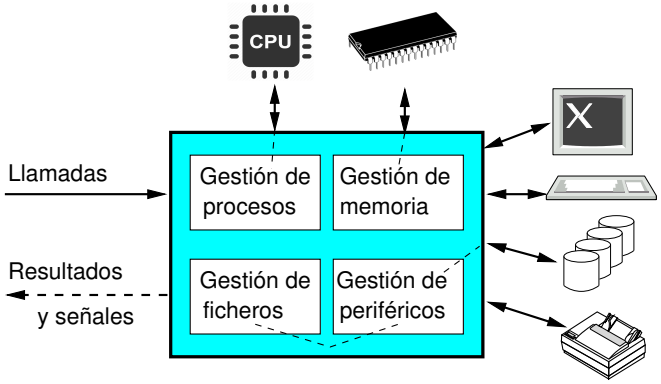



**Figura 2.15**: Componentes del sistema operativo.

[29] Todo esto forma parte de un marco general para la seguridad en las TIC en el que hay un estándar internacional de certificación, **Common Criteria** *(Common Criteria for Information Technology Security Evaluation)*. En España, el organismo de certificación es el Centro Criptológico Nacional (`https://www.oc.ccn.cni.es/`).

[30] Si dispone usted de un ordenador con los programas fuentes de Linux puede comprobar estos datos con los programas `find` y `sloccount`. Suponiendo que los fuentes se encuentran en el directorio `/usr/src/linux-source-3.16/`:

```
$ find /usr/src/linux-source-3.16/ -type f -print | wc -l
47382
$ sloccount /usr/src/linux-source-3.16/
12,582,321
```

(Las cifras pueden variar ligeramente porque dentro de una misma versión hay actualizaciones, y además las distribuciones pueden añadir sus propios «parches»).

## Gestión de procesos

El subsistema de gestión de procesos se ocupa de atender a las llamadas relacionadas con la creación, destrucción y comunicación entre procesos y de repartir la CPU entre los procesos.

Cuando se crea un proceso (con `FORK`) el sistema le asigna una zona de memoria exclusiva para él. Se llama **imagen en memoria** del proceso e incluye tanto una copia de las instrucciones del programa (podría llamarse «código», pero tradicionalmente se conoce como «texto») y espacio reservado para datos que crecen dinámicamente durante la ejecución.

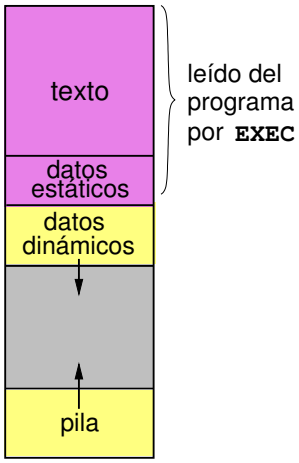



**Figura 2.16**: Imagen en memoria de un proceso.

En las primeras implementaciones de Unix se procesaba la llamada `FORK` como hemos sugerido en su descripción (página 37): haciendo una copia en la memoria del código del proceso padre. Pero es poco eficiente dedicar tiempo y espacio en memoria a esta operación si ninguno de los dos (ni el padre ni el hijo) van a modificar el código, o si el proceso hijo empieza con `EXEC`. Las implementaciones modernas utilizan la técnica «*copy-on-write*»: se crea el proceso hijo, pero no se hace copia de la imagen en memoria mientras él o su padre (o algún antepasado) no intenten escribir. La figura 2.16 muestra la imagen en memoria de un proceso después de ejecutarse una `EXEC` (la pila es un concepto que estudiaremos en el apartado 8.8).

Como el número de procesos normalmente es muy superior al número de procesadores de la CPU, una función esencial de la gestión de procesos es determinar, en cada momento, qué procesos hacen uso de los procesadores (procesos **activos**) y qué procesos están esperando, bien porque aún no tienen su «turno» (procesos **preparados**) o bien porque han pedido alguna operación de entrada salida, o una llamada `WAIT`, y no pueden continuar hasta que la operación termine o termine el hijo por el que espera `WAIT` (procesos **bloqueados**).

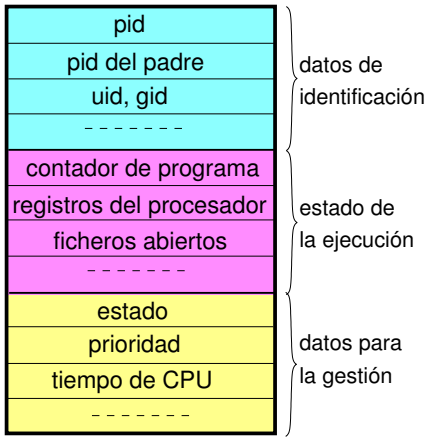



**Figura 2.17**: Bloque de control de proceso.

Un proceso que pasa de activo a preparado o bloqueado tiene que «recordar» en qué estado de la ejecución se encuentra para que luego pueda continuar cuando vuelva a estar activo. Para ello, a cada proceso se le asigna una estructura de datos, el **bloque de control del proceso, PCB** *(Process Control Block)* con los datos necesarios (figura 2.17). En el «estado de la ejecución» se conservan estos datos a recordar y a reponer cuando el proceso pase a estar activo: el «contador de programa» (que indica en qué instrucción del programa se encuentra la ejecución) los contenidos de los registros del procesador, los descriptores de ficheros abiertos, etc. Entre los datos para la gestión se encuentran el estado del proceso (preparado, bloqueado, etc.), la prioridad, el tiempo usado de CPU y de otros recursos a efectos de planificación y contabilidad, etc.

Los PCB de todos los procesos que están «vivos» (activo o activos, preparados y bloqueados) se colocan en una **tabla de procesos**. Cuando un proceso termina se libera su entrada en la tabla dejando sitio para un nuevo proceso. Como esta tabla tiene un tamaño limitado, también está limitado el número de procesos que pueden coexistir en un momento dado[31].

[31] Al menos en Linux se puede ver este número con `cat /proc/sys/kernel/pid_max`.

El **planificador** *(scheduler)* es la parte del sistema responsable de repartir la CPU entre los procesos preparados. Éstos se colocan en una cola, la **cola del procesador**, que el planificador ordena según la prioridad, como indica la figura 2.18 (suponiendo el convenio de que 0 representa la máxima prioridad, 1 la siguiente, etc.) Cuando el planificador decide activar un proceso asignándolo a un procesador saca de la cola el PCB del proceso que está en cabeza y cambia su estado a activo.

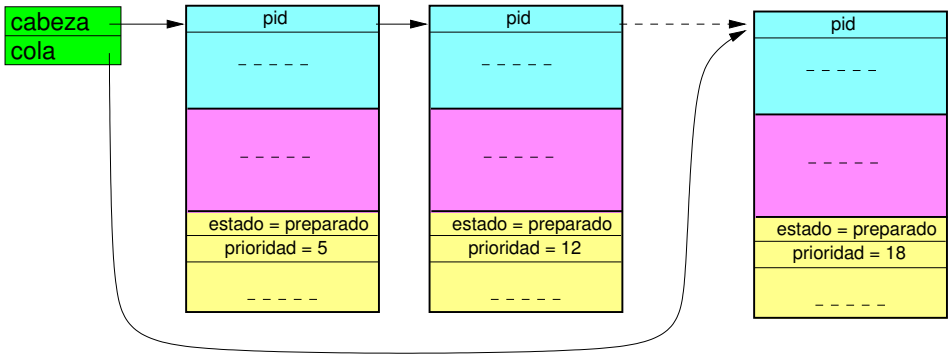



**Figura 2.18**: Cola de procesos preparados.

## Gestión de memoria

Los programas que forman el subsistema de gestión de memoria tienen la función de ofrecer a los procesos la abstracción de la *memoria virtual* (página 40). La multiprogramación implica, desde el punto de vista de la gestión de la memoria, llevar una cuenta de las zonas ocupadas y disponibles, desalojar temporalmente de la memoria (llevándolo a la memoria secundaria) algún proceso cuando no quede espacio en la memoria principal, reubicar procesos cuando sea necesario, y proteger el acceso de los procesos a zonas reservadas para otros procesos. Para estas tareas se utilizan varias técnicas que combinan hardware y software. Hagamos un resumen. Abreviaremos con «SGM» la expresión «sistema de gestión de memoria».

Una primera idea para asignar memoria a los procesos es ir ubicándolos uno detrás de otro en la zona de memoria que está libre. Normalmente el sistema operativo ocupa las direcciones más bajas, y una vez creados tres procesos, por ejemplo, la memoria quedaría como muestra la figura 2.19. Cada proceso tiene su espacio de direccionamiento, determinado por las direcciones *db* (dirección de base) y *dl* (dirección límite). Siempre que un proceso pide un acceso a la memoria el SGM, apoyado por algunos elementos de hardware (registro de base y registro límite) comprueba, antes de permitirlo, que está dentro de su espacio.

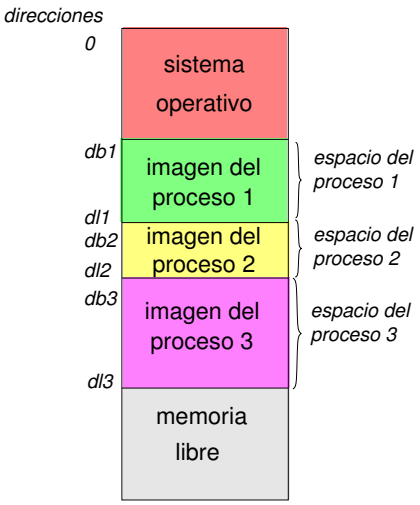



**Figura 2.19**: Asignación contigua.

Esta idea es sencilla de implementar y requiere poco hardware. Pero tiene un grave inconveniente: a medida que van terminando procesos y se van creando otros nuevos el SGM tratará de ubicar a los nuevos en las zonas que están libres. Supongamos que termina el «proceso 2». Un proceso nuevo sólo podrá utilizar el espacio que ha quedado libre si su imagen tiene un tamaño igual o inferior al que ocupaba ese proceso. Es fácil imaginar que al cabo del tiempo la memoria física estará llena de «huecos» demasiado pequeños para albergar nuevos procesos. A este fenómeno se le llama **fragmentación externa**.

¿Qué se hace cuando no queda espacio contiguo suficiente para un nuevo proceso? La solución es desalojar alguno de los procesos bloqueados, o últimos en la cola de procesos, copiando su imagen en la memoria secundaria y dejando libre su espacio. Cuando el proceso desalojado tenga que volver a estar activo se copia de la memoria secundaria a la principal (eventualmente, desalojando a otro). Esta técnica se llama **intercambio** *(swapping)*, y es costosa en tiempo si la imagen es grande.

Prácticamente todos los sistemas operativos utilizan un método mucho más elaborado y eficiente, pero que requiere la ayuda de un hardware adicional: la *unidad de gestión de memoria*. Se llama **paginación** y se basa en considerar el espacio de la memoria como una sucesión de «marcos de página», o «páginas físicas». Una página es un conjunto de bytes, normalmente 4.096 (4 KiB)[32], de direcciones contiguas. La página es la unidad mínima de asignación de memoria para un proceso. Es decir, a todo proceso se le asigna un número entero de páginas («páginas lógicas»).

La traducción entre las páginas lógicas del espacio que «ve» el proceso y las páginas físicas que le corresponden se realiza mediante una **tabla de páginas** para cada proceso. El PCB contiene la dirección de la memoria donde comienza la tabla de páginas del proceso. Supongamos un tamaño de página de 4 KiB y una memoria de 1 GiB, es decir, $2^{30}/4 \times 2^{10} = 2^{18} = 262.144$ marcos de página. Si la imagen de un proceso tiene algo menos de 20 KiB ocupará cinco páginas lógicas. La tabla de páginas de este proceso contiene los números de páginas físicas que corresponden a cada una de las páginas lógicas. Por ejemplo, si a la página lógica 0 le corresponde la página física 9, y si el proceso pide un acceso a una dirección de memoria que está en la página lógica 0, como la dirección 1.000, la tabla de páginas permite traducir a la dirección física que realmente le corresponde: $8 \times 4 \times 1.024 + 1000$. Si se crea otro proceso, a sus páginas lógicas se le asignarán páginas físicas que estén libres (figura 2.20).

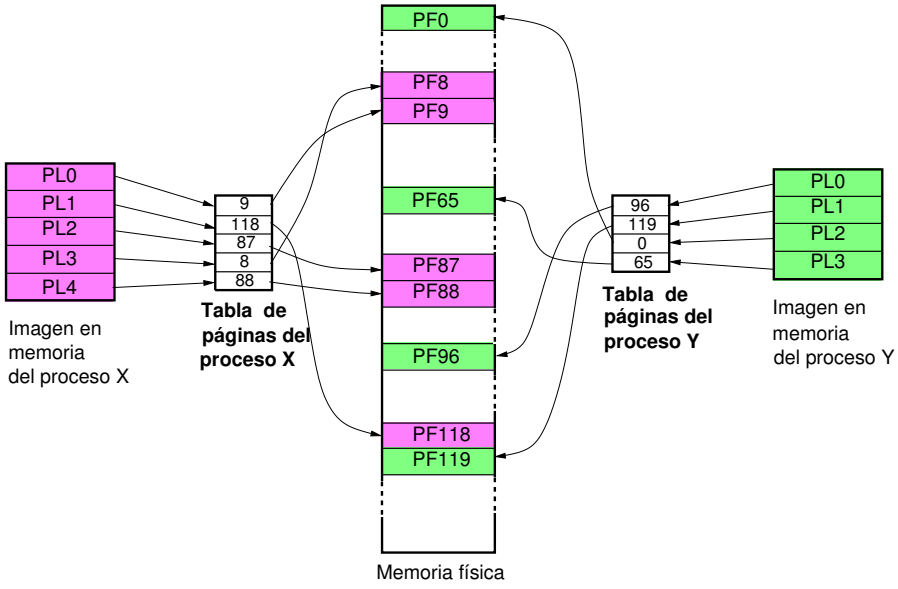



**Figura 2.20**: Tablas de páginas.

Con la técnica de paginación no hay fragmentación externa: se pueden usar todos los marcos de la memoria, sin «huecos» entre ellos. Sin embargo, hay **fragmentación interna**: si la imagen del proceso ocupa, por ejemplo, 4.097 bytes necesitará dos páginas, aunque de la segunda queden 4.095 bytes sin utilizar. En promedio, de la última página se ocupará la mitad, por lo que la fragmentación supone desperdiciar medio marco de memoria por proceso. Naturalmente, cuanto menor sea el tamaño de la

[32] Algunos sistemas tienen la posibilidad de definir tamaños de página mayores, de hasta 1 GiB *(«huge pages»)*. Se puede saber el tamaño de página que utiliza el sistema con la utilidad `getconf`: «`getconf PAGESIZE`».

página menor será la fragmentación interna, pero reducir el tamaño de página implica aumentar en la misma proporción el tamaño de todas las tablas de páginas.

Las tablas de páginas residen en la memoria principal. Esto supondría que para todas las peticiones de acceso a la memoria de un proceso serían necesarios en realidad dos accesos: uno para buscar en la tabla y traducir a la dirección física y otro para el acceso real. De ahí la necesidad de la **unidad de gestión de memoria, MMU** *(Memory Management Unit)*, que traduce mediante hardware[33]. Ahora bien, las tablas de páginas son demasiado grandes para mantenerlas en hardware, y se aplica una idea similar a la de la memoria *cache* (página 24): la MMU contiene una memoria muy rápida (pero de tamaño reducido, por su coste) con la que se obtiene la traducción de manera inmediata gracias a que es una *memoria asociativa*, cuyo funcionamiento veremos en el apartado 11.3. Se llama **TLB** *(Translation Look-aside Buffer)* y contiene las traducciones de páginas lógicas a páginas físicas más recientes. La dirección generada por el proceso tiene dos partes: el número de página lógica, PL, y la dirección relativa dentro de la página, *d*. Si la traducción de PL a la página física que le corresponde se encuentra en el TLB, la dirección física se obtiene de manera inmediata (figura 2.21). Si no es así hay un fracaso y es preciso recurrir a la tabla de páginas en la memoria.

El contenido de la tabla de páginas de un proceso no cambia (salvo algunas informaciones adicionales que comentaremos enseguida), pero el del TLB se tiene que actualizar cada vez que se cambia de proceso. Una indicación (un bit) en cada entrada (fila) del TLB la señala como válida o no.

Cuando un proceso deja de ser activo para que otro pase a serlo se marcan como inválidas todas las entradas del TLB, de modo que los primeros accesos a memoria del nuevo proceso necesitan recurrir a la tabla de páginas, pero rápidamente se van rellenando con los valores adecuados las nuevas entradas del TLB.

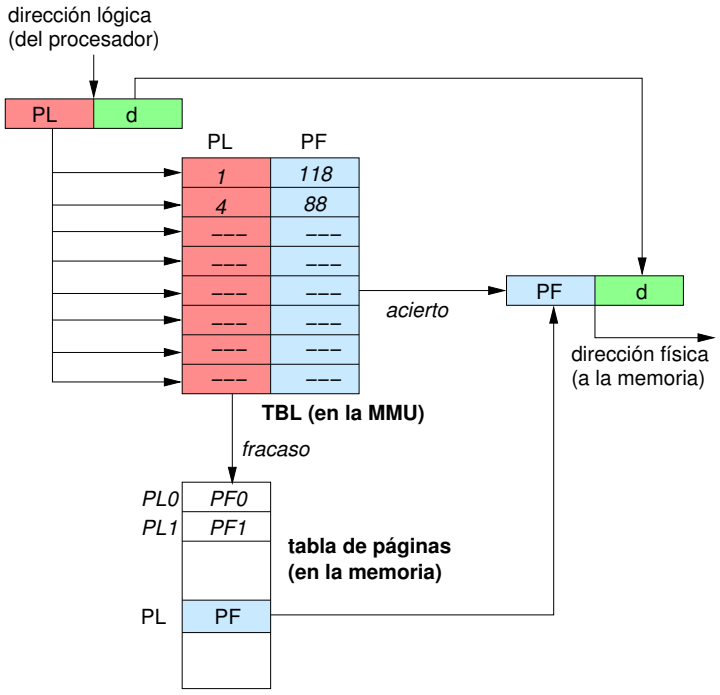



**Figura 2.21**: Traducción por medio de la TLB.

Con la paginación la protección de memoria de unos procesos frente a otros está garantizada: un proceso sólo puede acceder a las direcciones físicas que correspondan a sus páginas lógicas. Cada proceso dispone de su espacio de direccionamiento que, además, puede ser muy superior al espacio físico disponible en la memoria principal. En efecto, los números de páginas físicas que se almacenan en las tablas de páginas se representan típicamente en 32 bits, lo que permite generar $2^{32}$ marcos de página. Si éstos son de 4 KiB, resulta que la memoria física que puede direccionarse es $2^{32} \times 4 \times 2^{10} = 2^4 \times 2^{40} = 16$ TiB. No todas las páginas físicas asignadas a un proceso tienen que estar en la memoria principal: si no hay suficiente espacio se llevan a memoria secundaria, y un bit en la tabla de páginas indica si la página física en cuestión está o no en memoria principal. Al pedir un acceso, el SGM comprueba este bit y si la página no está se produce un **fracaso de página** *(page fault)*, que

[33] Actualmente la MMU suele estar integrada en el mismo chip que la CPU.

provoca el acceso a la memoria secundaria para copiarla en la principal (eventualmente desalojando a otra). Esta técnica se llama **paginación con demanda** *(demand paging)*. Otro bit de la tabla de páginas (el bit «sucio», *dirty bit*), señala si durante su estancia en la memoria principal el contenido de la página se ha modificado; si no se ha modificado y la página tiene que ser sustituida por otra no es necesario volver a copiarla en el disco.

En principio, la paginación con demanda hace innecesario el intercambio de procesos: si la memoria física está muy llena no es necesario desalojar a un proceso completo, solamente a algunas de sus páginas. Pero puede llegar un momento en que el sistema operativo tenga que dedicar más tiempo a estas operaciones de desalojo y copia de páginas que a trabajo realmente útil. El fenómeno se llama «**vapuleo**» *(thrashing)*. Por este motivo, en casos extremos de ocupación de memoria se recurre al intercambio de procesos completos[34].

Un esquema alternativo a la paginación es el de **segmentación**: en lugar de páginas, se utilizan **segmentos** cuyo tamaño no es fijo, sino que puede variar en el curso de la ejecución. Cada segmento define un espacio de direccionamiento independiente que puede acoger a un trozo de código más o menos largo, o a una estructura de datos (como una tabla, una lista, o una pila) de longitud variable. De este modo desaparece el problema de la fragmentación interna.

Una diferencia importante entre paginación y segmentación es que la primera es transparente en el nivel de máquina operativa, pero la segunda no. En efecto, con la paginación hay un solo espacio de direccionamiento virtual; los programas que hacen uso del sistema operativo utilizan ese espacio de direccionamiento único. Pero con la segmentación hay múltiples espacios de direccionamiento (tantos como segmentos definidos), y para cada problema concreto el programador decidirá los segmentos necesarios.

Los dos esquemas pueden combinarse de modo que cada segmento ocupe un número entero (y variable durante la ejecución) de páginas de longitud fija.

## Gestión de ficheros

Un **sistema de ficheros** *(filesystem)* es un conjunto de convenios sobre la organización de ficheros y directorios que incluye los aspectos visibles para los usuarios y para los programas de aplicación que hemos resumido en el apartado 2.4 (página 32) y también las estructuras de datos necesarias para su implementación. El **sistema de gestión de ficheros** (en adelante «SGF») es la parte del sistema operativo que se encarga de esa implementación: asignar zonas físicas de la memoria secundaria a los ficheros regulares, atender a las llamadas, etc.

Los distintos sistemas operativos disponen de sus propios sistemas de ficheros, con frecuencia más de uno. Por ejemplo, VFAT, NTFS, exFAT (para memoria *flash*)... en Windows; ext3, ext4, ReiserFS... en Linux. El SGF incluye programas para manejar varios de estos sistemas, incluso nativos de sistemas operativos diferentes. Así, `ntfs-3g` es un software de código abierto que puede incorporarse en el sistema de gestión de ficheros de un sistema Unix para manejar el sistema de ficheros NTFS.

La mayoría de los sistemas de ficheros están orientados a la organización de los datos en un disco, pero los hay también especializados para otros soportes: el ya mencionado exFAT y JFFS2 para *flash*, ISO 9660 para CD, DVD y Blu-ray, etc.

En general, la gestión de los sistemas de ficheros de disco en los sistemas de tipo Unix se basa en tres tipos de tablas almacenadas en la memoria principal cuyos contenidos van cambiando a medida

[34] Si dispone usted de un sistema Unix puede comprobar que normalmente en el disco hay una partición reservada para el intercambio de procesos y de páginas llamada «swap». Con la utilidad «`free`» puede ver las cantidades de memoria principal y de intercambio libres y ocupadas. Normalmente (salvo que el equipo sea antiguo y con poca memoria, o que esté muy cargado) observará que el área de swap está toda libre.

que se abren y se cierran ficheros y se opera con ellos:

- La tabla de ***inodes* copiados en la memoria**.
- La tabla de **ficheros abiertos**.
- Las tablas de **descriptores de ficheros** (una por cada proceso).

Ya hemos hablado de los descriptores y de los ficheros abiertos. Veamos lo que es un *«inode»*.

Todo fichero (sea regular o especial) tiene asociada una estructura de datos llamada ***inode*** *(index node)*, que contiene informaciones (metadatos) sobre el fichero. Su función, con respecto a su fichero, es la misma que cumple un PCB con respecto a su proceso. De hecho, en algunos documentos en lugar de *«inode»* (término clásico de Unix) se le llama FCB *(File Control Block)*. Una diferencia esencial (aparte de sus contenidos) es que los PCB «viven» en la memoria principal (salvo en las excepciones en las que un proceso, junto con su PCB, se desaloja temporalmente y se lleva al área de intercambio), mientras que el lugar de residencia permanente de los *inodes* es la memoria secundaria, igual que los ficheros. Cuando se abre un fichero se hace una copia de su *inode* en la memoria principal para que las eventuales modificaciones a sus datos sean más ágiles.

En la figura 2.22 se muestran los metadatos contenidos en un *inode* correspondiente a un fichero regular:

- El tipo de fichero (en este caso, regular, porque tiene asignados bloques del disco para los datos).
- Los permisos de acceso, *uid* y *gid* reales y efectivos, el número de enlaces al fichero (número de nombres en el directorio asociados a este fichero), el tamaño en bytes, los días horas, etc. de la creación, la última vez que se accedió al fichero y la última vez que se escribió en él.
- Números de los bloques asignados al fichero.

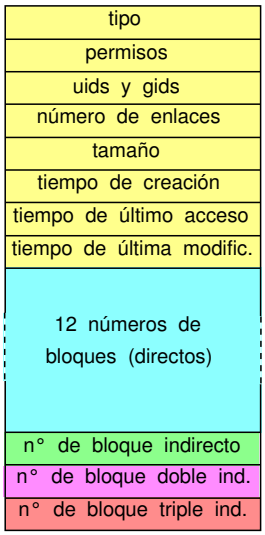



**Figura 2.22**: Un *inode*.

Este último punto requiere una explicación sobre cómo se asigna a los ficheros el espacio de almacenamiento del disco. En Unix se sigue un método («asignación indexada») similar al de la asignación de páginas para los procesos en la memoria principal (página 46): el espacio del disco se considera formado por un número entero de **bloques** de igual tamaño (página 26). El SGF mantiene una lista de los bloques que están libres en cada momento. Cuando se crea un fichero se le asignan los bloques que sean necesarios; si se escribe en él puede que al crecer necesite más bloques, que se toman de la lista de bloques libres. Y si se borra sus bloques se liberan y pasan a la lista de bloques libres[35].

Igual que en la paginación de la memoria principal, con este método no existe **fragmentación externa**[36], pero sí **fragmentación interna**: si un fichero no ocupa exactamente un número entero de bloques se malgasta una parte del último bloque. En particular, si el fichero necesita 4 o 1.000 bytes y el tamaño de bloque es 1 KiB ocupará, de todos modos, un bloque completo.

El tamaño del bloque y el número de bits con los que se representan los números de bloque determinan la capacidad máxima de un disco reconocible por el SGF. Con 32 bits para los números de bloque puede haber $2^{32}$ bloques, y si éstos son de 1 KiB la capacidad máxima resulta $2^{32} \times 2^{10} = 4 \times 2^{40}$ bytes = 4 TiB.

[35] Pero no se borra su contenido. De ahí que haya herramientas que permiten recuperar ficheros borrados (salvo que desde que se borró el fichero sus bloques hayan sido reutilizados).

[36] Esto explica que en los sistemas de tipo Unix no sea necesaria una operación de mantenimiento típica de otros sistemas que siguen otros métodos de asignación: «desfragmentar el disco».

Los directorios tienen también, como ficheros que son, sus propios *inodes*, y sus datos incluyen solamente una lista de los nombres de ficheros y los números de *inode* que corresponde a cada uno.

Volviendo al *inode* de la figura 2.22, vemos que contiene 12 números de bloques, por lo que el fichero puede tener asignados directamente hasta 12 bloques de datos; si son de 1 KiB, el fichero puede tener hasta 12 KiB. Para ficheros más grandes se recurre a un bloque indirecto: se trata de un bloque que no contiene datos, sino números de bloques. Si estos números se representan en 32 bits (cuatro bytes) el bloque indirecto puede albergar $n = 1.024/4 = 256$ números de bloque, y, por tanto, conducir a 256 bloques de datos que darán una capacidad de 256 KiB. Si no es suficiente también hay un bloque de doble indirección y otro de triple, como ilustra la figura 2.23, que aumentan la capacidad, respectivamente, en 64 MiB y 16 GiB. En este caso, el tamaño máximo de un fichero sería algo más de 16 GiB. Puede usted comprobar que si el tamaño del bloque se fija en 4 KiB los ficheros pueden llegar a 4 TiB.

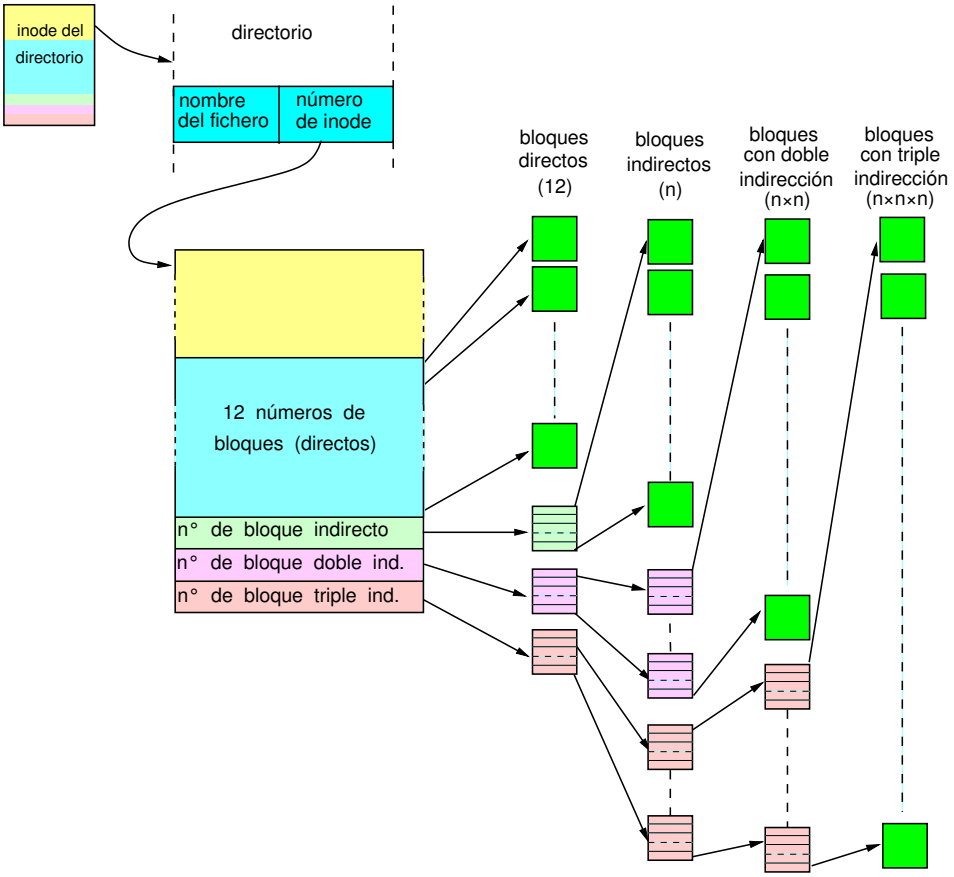



**Figura 2.23**: Asignación de bloques en el *inode*.

El SGF debe ocuparse de realizar las operaciones que piden las llamadas relacionadas con los ficheros (`OPEN`, `READ`, etc).

`OPEN` (página 35) busca el nombre del fichero en el directorio y en caso de éxito devuelve al proceso que la hace un descriptor que unívocamente identifica al fichero en ese proceso para todas las operaciones posteriores[37]. Cada PCB tiene asociada, a través del puntero «ficheros abiertos» (figura 2.17) una

[37]El motivo para utilizar el descriptor y no el nombre, aparte de que es más fácil tratar con un entero que con una cadena de caracteres, es que, debido a los enlaces (página 33), el mismo fichero podría tener varios nombres.

**tabla de descriptores de fichero**, en la que se guardan los descriptores con algunas propiedades *(flags)* definidas en `OPEN` (salvo el modo en que se ha abierto, lectura o escritura o ambos, que se pone en la tabla de ficheros abiertos). El mismo fichero puede ser abierto por distintos procesos; a cada uno se le devuelve un descriptor. Pero los distintos procesos pueden abrir el fichero con distinto modo de lectura y/o escritura, y cada proceso puede hacer operaciones (supuesto que tenga los permisos necesarios) que van modificando el valor de «posición» (página 32). De ahí que exista una estructura global (no asociada a ningún proceso en particular), la **tabla de ficheros abiertos**, en la que el SGF añade una entrada cada vez que ejecuta una `OPEN` de un proceso. Esta entrada tiene una referencia al proceso, una indicación del modo (lectura, escritura, etc.) y un puntero al *inode* del fichero. Como las modificaciones al *inode* son frecuentes (a medida que se va escribiendo o leyendo pueden ir variando el tamaño, los bloques asignados, las fechas, etc.), al abrir por primera vez un fichero se copia su *inode* en la memoria, formando una **tabla de *inodes* en memoria**. Cuando se hayan cerrado todas las operaciones el *inode* actualizado se escribe en el disco sustituyendo al primitivo.

De modo que si $n$ procesos abren un mismo fichero tendremos una copia del *inode* y $n$ entradas en la tabla de ficheros abiertos, cada una con el modo y el valor de posición. Ahora bien, a veces es necesario que dos o más procesos compartan las posiciones de sus ficheros abiertos. Así ocurre con la llamada `FORK`: el hijo hereda los ficheros abiertos por el padre, e interesa que cuando termine (con `EXIT`) el padre tenga las posiciones actualizadas. Por ejemplo, el padre abre un fichero y crea (con `FORK`) un hijo que (tras `EXEC`) escribe 500 bytes; termina el hijo y el padre crea otro hijo que vuelve a escribir 500 bytes en el mismo fichero. Si queremos que estos 500 bytes no se escriban borrando los primeros, sino a continuación de ellos, será necesario que la posición actualizada por el primer hijo (500) esté compartida con el padre y con el segundo hijo. En estos casos las tablas de descriptores de ficheros en los procesos deben apuntar a la misma entrada de la tabla de ficheros abiertos.

Si todo esto le parece complicado no se preocupe: lo es. La figura 2.24 puede ayudarle a comprenderlo. Muestra una situación en la que:

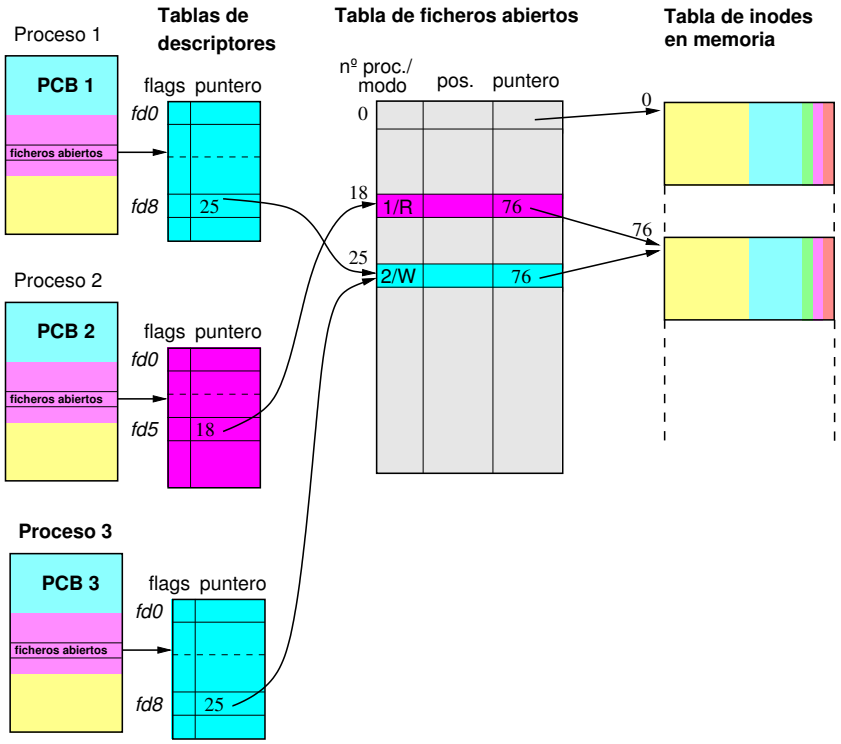



**Figura 2.24**: Las tablas de descriptores, la tabla de ficheros abiertos y la tabla de *inodes* en memoria.

- Los procesos 1 y 3 están emparentados (por ejemplo, 1 es padre de 3), y el proceso 2 es independiente de ambos. Los tres están operando sobre un mismo fichero (aquél cuyo *inode* ocupa la posición 76 en la tabla de *inodes* en memoria).
- El proceso 1 ha abierto el fichero para escritura y ha recibido como descriptor el número 8. Es decir, se ha ejecutado `OPEN("fichero",O_WRONLY)`, y el SGF ha devuelto 8. El SGF también ha asignado una entrada, la 25, en la tabla de ficheros abiertos y si el fichero no estaba ya abierto ha copiado su *inode* del disco a la memoria. En la octava componente (*fd8*) de la tabla de descriptores del proceso 1 el SGF ha puesto «25», que es el índice a la tabla de ficheros abiertos, y en la entrada 25 de ésta ha puesto «76» que es el índice al *inode* recién copiado en la tabla de *inodes* en memoria.
- El proceso 1 ha creado después, con `FORK`, al proceso 3. Por tanto, éste ha heredado todos los ficheros que su padre tenía abiertos, con sus índices. Concretamente, en *fd8* de su tabla de descriptores tiene, como su padre, «25». La entrada 25 de la tabla de ficheros abiertos contiene «2» en «número de procesos apuntando».
- El proceso 2 ha abierto el mismo fichero para lectura (`OPEN("fichero",O_RDONLY)`). El SGF ha devuelto en este caso *fd* = 5. Como el proceso es independiente de los otros dos, en el descriptor de fichero 5 tiene otro índice, concretamente «18», para que el proceso tenga su propia entrada en la tabla de ficheros abiertos. Esta entrada apunta también al índice 76 de la tabla de *inodes* en memoria, porque se trata del mismo fichero.

La función del campo «nº proc.» (número de procesos apuntando) de la tabla de ficheros abiertos es fácil de entender: cuando un proceso cierra el fichero su contenido se decrementa, y si llega a cero se libera la entrada de la tabla, pero no necesariamente la copia del *inode*. Cuando el SGF detecta que no queda ninguna entrada apuntando al *inode* entonces sí libera su espacio en la tablas de *inodes* en memoria.

## Gestión de periféricos

También conocida como **gestión de dispositivos** o **gestión de entrada/salida**, ésta es la parte que incluye el software necesario para las comunicaciones con los distintos dispositivos periféricos.

Como veíamos en la figura 2.7, cada periférico se conecta a un bus por medio de un controlador, un hardware específico para ese dispositivo. Paralelamente, para cada controlador de periférico hay un **gestor del periférico** *(device driver)*, un software que se ocupa de las operaciones de dar valores iniciales a los registros («puertos») del controlador y de «vigilar» si las comunicaciones se desarrollan correctamente.

Dada la gran variedad de periféricos a los que es posible tener que atender, el sistema operativo puede contener cientos o miles de gestores[38] y aún así no cubrirá todas las posibilidades: periféricos nuevos, o raros, para los que normalmente el fabricante proporciona el gestor (no siempre para todos los sistemas operativos). A diferencia de las otras partes del sistema, no tiene sentido mantener en la memoria todos estos gestores, solamente los que realmente sean necesarios para los periféricos instalados. Por este motivo, los gestores normalmente se implementan como módulos que pueden cargarse dinámicamente (página 68).

El gestor de un periférico incluye la rutina de servicio de sus interrupciones (veremos lo que es ésto en el apartado siguiente, página 59) y se ocupa de los detalles específicos, ocultándolos y ofreciendo un

[38] De los más de doce millones de líneas de código mencionados en la nota 30 (página 43) más de siete son para más de 6.000 gestores de periféricos.

modelo más abstracto del periférico a las otras partes del sistema operativo. El gestor y el controlador están íntimamente relacionados, y entre ambos proporcionan la interfaz entre el sistema operativo y el periférico. Dependiendo del diseño, algunas funciones puede asumirlas el software (el gestor) o el hardware (el controlador). Por ejemplo, un controlador de disco, para acceder a un determinado sector, puede necesitar que se le pasen los números de cilindro, cabeza y sector; en ese caso, el gestor debe ocuparse de traducir el número de bloque que genera el SGF a los datos que requiere el controlador. Pero los discos modernos utilizan LBA (página 26), y en ese caso la función la asume el controlador; el gestor sólo tendrá que encargarse, para esa función, de los ajustes necesarios en el caso de que el tamaño de bloque que maneja el SGF sea diferente del de los bloques físicos del disco.

Otra función de un gestor de periférico es poner en cola a los procesos que hagan peticiones cuando el periférico está ocupado. Para cada periférico hay una cola de procesos similar a la cola del procesador (figura 2.18). No obstante, hay un procedimiento para reducir la sobrecarga de tiempo de las colas de periféricos. Se llama **spooling** *(Simultaneous Peripheral Operations On-Line)* y consiste en que, en lugar de hacer la transferencia directamente con el periférico, se hace con un fichero del disco; estos ficheros actúan como *periféricos virtuales* para los procesos. Se utiliza típicamente para la impresora: el *spooler* de la impresora es el proceso que realmente escribe en ella. Cuando un proceso abre la impresora el sistema le asigna un fichero, y todas las transferencias hacia la impresora se dirigen a ese fichero; cuando la cierra, el fichero se pone en la cola formada por los ficheros creados anteriormente por otros procesos. El *spooler* es un *demonio*: un proceso que permanece inactivo hasta que algún acontecimiento lo despierta. Mientras la cola de impresión está vacía, `lpd` está «dormido» (bloqueado), y se «despierta» (pasa a preparado) en cuanto se pone algún trabajo en la cola. Como las transferencias con el disco son mucho más rápidas que con la impresora, esto mejora la potencia en las situaciones en que varios procesos están bloqueados porque necesitan imprimir.

## 2.6. Funcionamiento del sistema operativo (modelo procesal)

Hemos resumido las funciones de un sistema operativo de tipo Unix (proporcionar servicios en forma de llamadas y facilidades para el uso y la protección) y su estructura (las partes lo componen y las funciones y estructuras de datos de cada una). Veamos ahora unas ideas generales sobre su funcionamiento.

Para poder hacer una descripción suficientemente detallada pero comprensible y que al mismo tiempo no exceda de límites razonables, *en todo este apartado supondremos que la CPU sólo tiene un procesador*. En el caso de multiprocesador la gestión de procesos se complica: si hay $N$ procesos y $n$ procesadores con $N > n$ es preciso repartir en el tiempo equilibradamente los procesadores a los procesos. Pero si $n = 1$, como supondremos en adelante, la gestión es más sencilla.

### Arranque

Empecemos por lo primero: lo que ocurre cuando se «enciende» la máquina. La secuencia de pasos antes de que la máquina esté operativa depende del tipo de máquina: un sistema embebido, por ejemplo, es muy distinto de un ordenador personal. Nos centraremos en el caso más común de un ordenador personal o un servidor centralizado. Se suceden cuatro fases antes de que el sistema esté disponible para sus usuarios:

1. Arranque inicial desde la ROM.

2. Continuación del arranque desde la memoria secundaria.

3. Carga e iniciación del sistema operativo.

4. Carga e iniciación de servicios externos al sistema operativo.

### Arranque inicial desde la ROM

Casi toda la memoria principal es volátil, por lo que al encender la máquina no contiene nada (o su contenido no es significativo). La secuencia de actividades iniciales en las máquinas más comunes es la siguiente:

Lo primero que la CPU ejecuta es un programa grabado en la zona ROM (o EEPROM, nota 3, página 6), que consta de tres partes:

1. Comprobación del correcto funcionamiento del hardware: la CPU, toda la memoria principal (incluida la ROM), buses, periféricos, etc. En caso de errores informa mediante un código sonoro (con pitidos) y/o un código numérico (con un visualizador). También lee una pequeña memoria de tecnología CMOS alimentada por una batería que conserva, entre otras cosas, la hora. Esta parte se llama «POST» *(Power-On Self Test)*. Solamente si no hay errores se pasa a la segunda.

2. El cargador inicial, o cargador de arranque *(bootstrap loader)*[39], un programa cuya función es cargar en la memoria un sector del dispositivo de arranque. Pero antes, opcionalmente (pulsando alguna tecla en los primeros instantes), presenta una interfaz de texto en la que se pueden cambiar algunos parámetros. Entre ellos, determinar en qué orden se deben buscar los dispositivos que puedan contener un sector de arranque válido: disco, CD, memoria USB, red...

3. Algunos programas sencillos para servicios básicos de bajo nivel, como leer y escribir caracteres. Los sistemas operativos modernos no utilizan estos servicios, pero esto ha dado lugar al nombre con que se conoce al contenido de la ROM: «BIOS» *(Basic Input/Output System)*. La BIOS ha sido un estándar *de facto* desde los años 1980. Actualmente está siendo desplazado por otro, «UEFI» *(Unified Extensible Firmware Interface)*, que supera algunas limitaciones del diseño de la BIOS.

### Continuación del arranque desde la memoria secundaria

Para concretar, supondremos en adelante que el dispositivo de arranque es un disco magnético. Normalmente, el disco está «particionado»[40], es decir dividido lógicamente en un pequeño número de partes («particiones») con las que el sistema trabajará como si fuesen discos diferentes. Y cada partición está «formateada»[41] para un determinado sistema de ficheros.

En el caso de BIOS, lo único que carga en la memoria el cargador inicial (que podemos llamar «etapa 0» del arranque) es un sector del disco, el primero (el sector 0), llamado **MBR** *(Master Boot*

[39]Esta expresión tiene un origen interesante. «Bootstrap» es la pequeña lengüeta trasera que tienen algunas botas. «To pull oneself up by one's bootstraps» («levantarse uno mismo tirándose de los *bootstraps*») es una frase hecha que equivale a «arreglárselas por sí mismo», que era el problema que se presentaba antes de que se utilizasen memorias no volátiles: todo programa tiene que estar cargado en la memoria principal para poder ejecutarse, y para esto se necesita un cargador; pero el cargador es un programa. ¿Quién carga al cargador? La única forma de arrancar era introducir manualmente en la memoria, mediante conmutadores de la consola, un pequeño programa cargador. ¿Cómo hacer que el sistema «se las arregle por sí mismo»? Esto ha derivado en el verbo «boot» con el sentido de «arrancar automáticamente» (no de «dar una patada»). Y, en «espanglish», en «botar» y «rebotar».

[40]Con programas de utilidad, como `fdisk`, `cfdisk` o `gparted`.

[41]Con programas de utilidad como `mkfs.ext4`, `mkfs.ntfs`, o el mismo `gparted`.

*Record)* que contiene una **tabla de particiones** y un programa que continúa con el proceso de arranque («etapa 1»). La tabla de particiones tiene datos sobre los sectores en los que comienzan y terminan las particiones y una indicación sobre la partición en la que puede encontrarse la continuación del programa de arranque («etapa 2»).

La figura 2.25 es un ejemplo. El disco `/dev/sda` está dividido en cuatro particiones. La primera, en la que se montará el sistema de ficheros raíz, es conocida al sistema operativo como `/dev/sda1`. La segunda, `/dev/sda2`, que se montará en el subdirectorio `/boot`, está reservada para los programas que finalizan el arranque y para el sistema operativo «nativo» (otros sistemas operativos estarán en sus propias particiones). Hay también una partición `swap` que no se monta: es la que utiliza el SGM en las operaciones de intercambio (página 46). Finalmente, la partición `/dev/sda3` puede estar formateada para un sistema Windows (sistema de ficheros FAT o NTFS).

La figura también muestra la secuencia de etapas del arranque. El programa de arranque inicial grabado en ROM carga en la memoria el MBR. Se ejecuta la etapa 1, que esencialmente es un cargador que conoce la ubicación de la etapa 2 (en este caso, en `/dev/sda2`) y la carga en memoria. La ejecución de la etapa 2, si hay varios sistemas operativos, presenta un menú al usuario para que escoja uno (normalmente, con un límite de tiempo: si se sobrepasa, el programa opta por defecto por uno establecido en la configuración). A partir de ahí, se inicia la carga del sistema operativo.

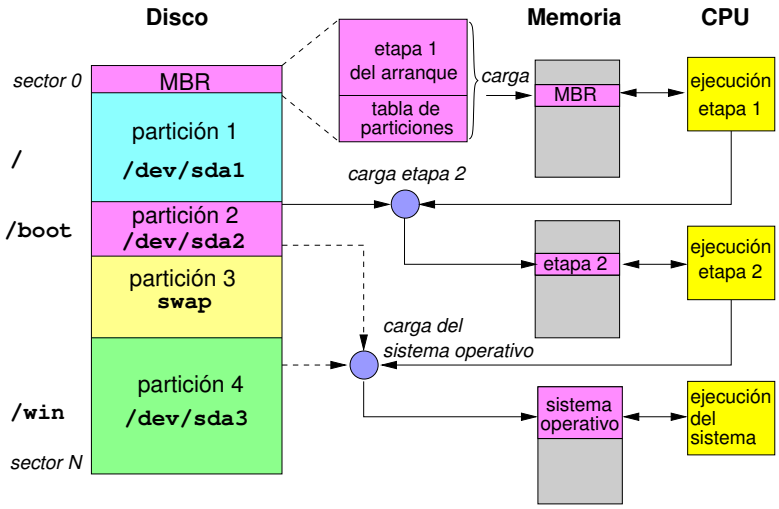



**Figura 2.25**: MBR, particiones y etapas del arranque.

Todos los sistemas de tipo Unix en una máquina con BIOS siguen estos pasos ejecutando un programa **gestor de arranque**. El más utilizado (al menos en los sistemas de código abierto y en Solaris) es **GRUB** *(GRand Unified Bootloader)*, que instala en el disco el MBR con la etapa 1 y la etapa 2.

El motivo para descomponer el programa de arranque en etapas *(chain loading)* tiene que ver con las limitaciones que por su diseño tiene el MBR. La tabla de particiones permite cuatro particiones «primarias» (aunque cada una puede incluir particiones «extendidas»). Esta tabla ocupa 16 bytes por partición. Sumando dos bytes para una «firma» (un código que identifica al sector como MBR), de los 256 bytes quedan libres 446 bytes para el cargador de arranque. Hace décadas, esto era suficiente para cargar directamente el sistema operativo (MS-DOS), que se encontraba en una localización fija en el disco, pero no para un gestor de arranque que da opciones para varios sistemas operativos y distintos sistemas de ficheros con esquemas de partición arbitrarios.

Otra de las limitaciones del MBR procede de que representa los números de sectores con 32 bits. Esto implica que como máximo puede direccionar $2^{32}$ sectores, y si los sectores son de 512 bytes la máxima capacidad del disco para poder acceder a todas sus particiones es $2^{32} \times 512 = 4 \times 2^{30} \times 10^{10}/2 = 2 \times 2^{40}$ bytes. Es decir, 2 TiB.

Asociado al nuevo estándar UEFI hay un nuevo diseño del MBR que se llama **GPT** *(GUID Par-*

*tition Table)*[42]. El primer sector del disco contiene un «MBR de protección» para que pueda también arrancarse con un sistema basado en BIOS. El GPT comienza en el sector 2 y ocupa varios sectores que pueden ser de 512 o de 4.096 bytes para discos con *«Advanced Format»* (página 26). GPT utiliza LBA para direccionar los discos (en MBR se sigue el sistema antiguo, CHS, página 26) y puede definir hasta 128 particiones en discos de hasta 18 EiB (18 exabytes = $18 \times 2^{20}$ TiB).

## Carga e iniciación del sistema operativo

El gestor de arranque termina cargando en la memoria el sistema operativo a partir de la dirección 0 (donde empiezan los «vectores de interrupción» que estudiaremos en el apartado 9.7).

Antes de que realmente esté «operativo» el sistema tiene que dar valores iniciales a sus estructuras de datos y copiar algunos datos del disco en la memoria. Por ejemplo, para las tablas de ficheros abiertos y de *inodes* en memoria hay espacio reservado; estas tablas deben estar inicialmente vacías, pero la memoria puede tener contenidos arbitrarios en estos espacios (por ejemplo, los dejados por la etapa 2 del arranque). Hay que rellenarlos con ceros, o algún código que indique que no hay nada.

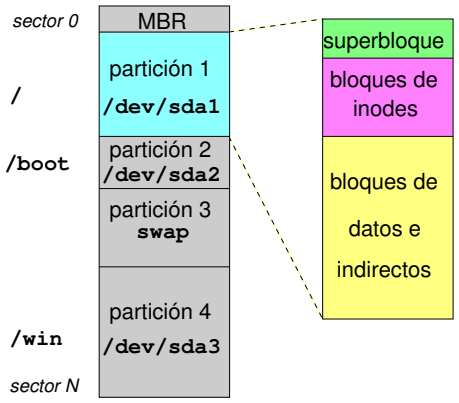



**Figura 2.26**: Metadatos y datos en una partición.

Los datos que hay que copiar del disco son metadatos sobre el sistema de ficheros, que se encuentran en uno o varios sectores del disco, al principio de cada una de las particiones: el **superbloque**. La figura 2.26 muestra, simplificadamente[43], la organización de una partición en un sistema de ficheros Unix. Después del superbloque hay espacio reservado para los *inodes* de los ficheros que se vayan creando[44] y el resto de la partición son bloques que pueden asignarse para los datos de los ficheros y para los bloques indirectos (página 50).

Los metadatos del superbloque son:

- **Informaciones generales sobre la partición**, como el número de bloques, el número de sectores por bloque y el número de *inodes*.
- El **mapa de bits de *inodes***, formado por tantos bits como *inodes* estén previstos. Cada bit se corresponde con un *inode* de la partición: si es «0» indica que el *inode* está libre, y si es «1», que está asignado a algún fichero. Los primeros *inodes* siempre estarán ocupados (corresponden a los ficheros y directorios existentes, a los periféricos reconocidos, etc.)
- El **mapa de bits de bloques**, con tantos bits como bloques tenga la partición, con idéntica función respecto de los bloques del disco: «0» indica que el bloque está disponible, y «1» que está asignado a algún fichero (o que es un bloque indirecto).

Estos metadatos se cargan inicialmente en la memoria, y se mantienen permanentemente en ella para que los programas del sistema de gestión de ficheros no tengan que acceder al disco y puedan

[42] «GUID» *(Globally Unique Identifier)* o «UUID» *(Universally Unique Identifier)* es un número de 128 bits que se utiliza para identificar de manera prácticamente inequívoca a un componente de software sin necesidad de una coordinación central.

[43] Para no complicar la figura no muestra una situación más real: en particiones grandes se repiten «grupos de cilindros», cada uno con un superbloque y bloques.

[44] Como este espacio es limitado, también es limitado el número máximo de *inodes* y, por tanto, de ficheros. Depende del sistema y de su configuración. Con la orden `df -i` puede usted ver este número y el número de *inodes* libres.

ejecutarse más rápidamente. El superbloque del disco se actualiza periódicamente (o dependiendo de la frecuencia con la que se modifica el superbloque en memoria)[45] y, en todo caso, al apagar el sistema.

Otra operación necesaria en la inicialización es montar el árbol de directorios. Se implementa mediante una lista encadenada, con un registro o nodo para cada directorio y para cada fichero que incluye punteros a los nodos que «cuelgan» de él. Cada partición tiene su propio árbol (en las particiones que contienen otros sistemas de ficheros distintos de Unix, implementado de otra forma), pero se necesita en memoria un árbol global. En el ejemplo de la figura 2.26, en la inicialización se debe montar la partición 2 sobre el punto de montaje (página 34) `/boot` y la partición 4 sobre `/win`. Esto se indica en un fichero, generalmente `/etc/fstab`, que podría tener este contenido[46]:

```
<sist. fich.>  <punto mont.>    <tipo>
/dev/sda1      /                ext4
/dev/sda2      /boot            ext4
/dev/sda3      /win             ntfs-3g
```

Significa que el árbol de la partición `/dev/sda1` debe montarse como raíz y que el sistema de ficheros es del tipo «ext4», etc.

## Carga e iniciación de servicios externos al sistema operativo

Terminadas las operaciones iniciales, el sistema operativo queda cargado en memoria y está disponible para atender a las llamadas y, como veremos enseguida, a las peticiones de los periféricos. Ahora bien, ¿de dónde proceden las llamadas? De algún proceso que ejecuta el programa que las contiene. Pero en este momento no hay ningún otro programa cargado en memoria, ni ningún proceso creado (salvo algunos demonios propios del sistema, como el planificador, página 45). Terminada la inicialización, el sistema operativo carga en memoria un programa que normalmente reside en `sbin/init`[47] y crea un proceso para él que va a ser el ancestro común de todos los procesos posteriores. Tiene *pid* = 1 y es un demonio que permanece en memoria (normalmente bloqueado) hasta que se apaga el sistema.

La función de `init` es lanzar nuevos servicios: servicios de red, acceso al sistema gráfico, etc. Concentrémonos en un caso sencillo, en el que el único servicio consiste en repartir inicialmente el sistema entre cuatro terminales con un intérprete de órdenes *(shell)* para cada uno.

La figura 2.27 resume gráficamente lo que ocurre. Con más detalle:

- `init` incluye cuatro llamadas `FORK` para crear cuatro procesos hijos. Cada uno de los hijos pide (con la llamada `EXEC`) la ejecución de un programa, `/sbin/getty`, que escribe en un terminal (cuyo número, entre 0 y 3, se le pasa como parámetro a `EXEC`) los caracteres «`login:`» y queda a la espera de que un usuario teclee algo. Si inicialmente no hay nadie en ninguno de los cuatro terminales el sistema se queda esperando, con cinco procesos bloqueados: `init` y los cuatro `getty`. Una situación más común es la que verá usted en el laboratorio: en lugar de `getty` se inicia un conjunto de procesos que proporcionan el entorno gráfico y un gestor de pantalla *(display manager)* que pide el nombre y la contraseña desde una ventana. Pero sigamos con la situación sencilla.

[45] Esta operación es importante, porque si «se cae» el sistema el superbloque del disco puede quedar en un estado inconsistente. Los sistemas de ficheros modernos (ext3, ext4, NTFS, etc.) utilizan *journaling*, una técnica que evita estas situaciones.

[46] Si examina usted el contenido de `/etc/fstab` verá que hay algunos otros datos que hemos omitido: cómo se monta (sólo lectura, etc.), qué hacer en caso de errores, etc.

[47] Éste es el enfoque «clásico». Hay varias alternativas para mejorar el tiempo de respuesta. En particular, muchas distribuciones de Linux están adoptando (con bastante controversia) «systemd».

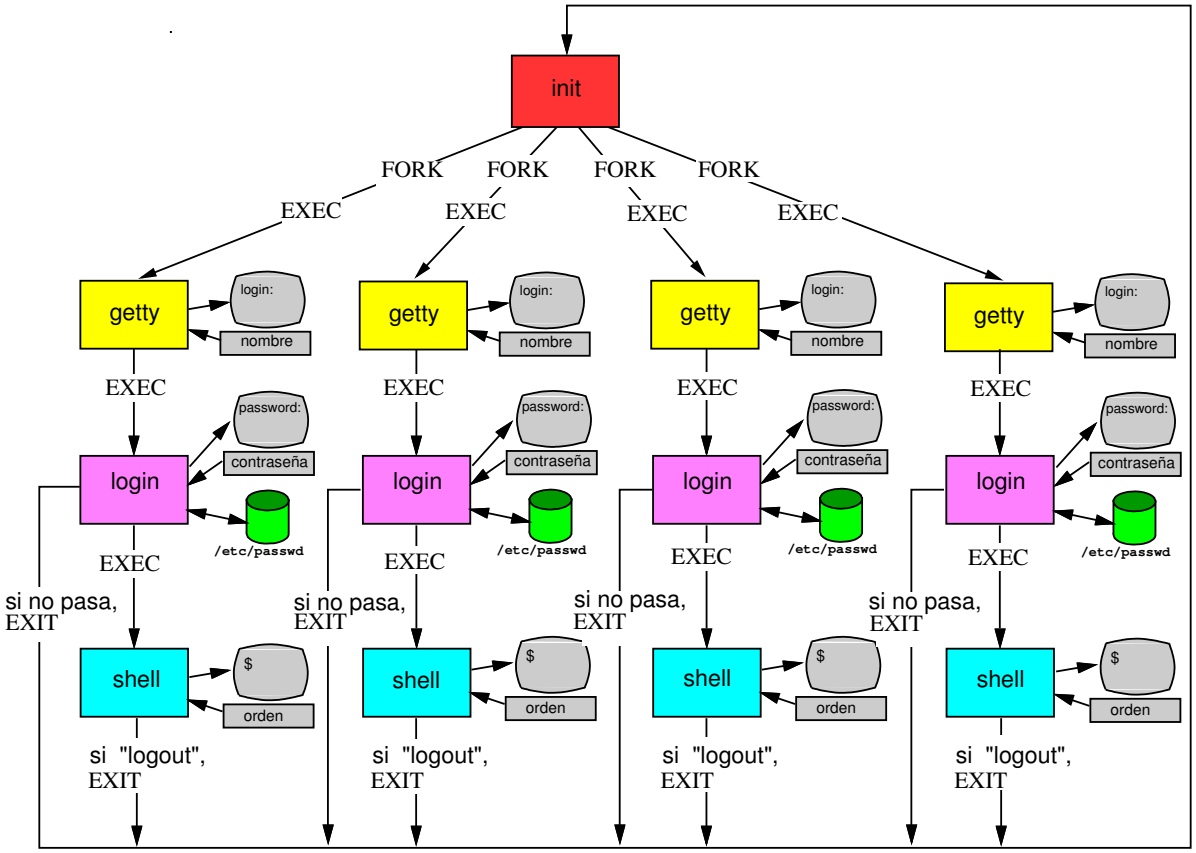



**Figura 2.27**: Creación de cuatro terminales.

- Cuando en un terminal se escribe un nombre se activa su proceso `getty` bloqueado, que sigue con una `EXEC` para ejecutar el programa `/bin/login`, pasándole como parámetro el nombre recibido del usuario. Este programa empieza haciendo algo parecido a `getty`: escribe «`password:`» en el terminal y queda esperando a que el usuario teclee su contraseña. Cuando tal cosa ocurre se comparan el nombre y la contraseña con los almacenados en el fichero de contraseñas (`/etc/passwd`, o `/etc/shadow`), y
  - si no hay coincidencia, el proceso termina con `EXIT` y, como muestra la figura 2.27, `init` (que «se despierta», porque había quedado bloqueado por una `WAIT`) vuelve a llamar a `getty` sobre ese mismo terminal;
  - si la hay, el usuario ha quedado autenticado, y `login` ejecuta una `EXEC` para cargar al intérprete, «`shell`» en la figura (como normalmente hay varios intérpretes a elegir, su nombre en realidad puede ser «`tcsh`» o «`bash`», etc). Además, el proceso, que hasta ahora tenía *uid* y *gid* de «root» cambia a los correspondientes del usuario.
- El intérprete escribe la invitación *(prompt)*[48] y queda a la espera de una orden.
- Cuando la orden sea «`logout`» el intérprete hará la llamada `EXIT` que «despierta» al padre, `init`.

Estos «terminales» pueden ser físicos o virtuales. Las distribuciones de Linux, por ejemplo, instaladas en un ordenador personal, que no tiene más que un terminal (pantalla y teclado), ofrecen hasta ocho terminales virtuales (`tty1` a `tty7`), pudiéndose cambiar de uno a otro con una combinación de teclas: Alt-F1, Alt-F2, etc.

[48] «`$`», «`#`» y «`%`» son invitaciones usuales, pero puede ser cualquier carácter, o una secuencia de caracteres, a elección del usuario, definidos en un fichero de configuración propio del usuario.

## Interrupciones y cambios de contexto

«Si inicialmente no hay nadie en ninguno de los cuatro terminales el sistema se queda esperando, con cinco procesos bloqueados». ¿Esperando qué? Obviamente, esperando que un usuario escriba algo. Pero ¿cómo se entera el sistema de que se ha escrito algo?

Es hora de hablar del invento más prodigioso en el campo de la arquitectura de ordenadores, comparable a lo que representa el invento de la rueda para los vehículos. Porque sin él nada de lo que estamos estudiando sería posible. Se trata de **la interrupción**.

En el tema «Procesadores hardware» estudiaremos en detalle las interrupciones, de manera general (apartado 8.9) y en un procesador concreto (apartado 9.7). Veamos ahora solamente las ideas imprescindibles para comprender la conmutación de procesos y la atención a las llamadas y a los periféricos por parte del sistema operativo.

Una interrupción es una señal del hardware que provoca que el procesador deje de ejecutar el programa en curso y pase a ejecutar a otro. La causa de esta señal puede ser interna del procesador o externa, procedente de la periferia (periféricos, MMU, etc.).

De las causas internas la que más nos interesa aquí es una instrucción especial para el procesador que se puede incorporar en cualquier programa. Su nombre simbólico depende del lenguaje ensamblador que se utilice (`SVC`, `SWI`, `INT`...) y su codificación binaria depende del diseño del procesador, pero su función siempre es la misma: provocar una interrupción desde el programa.

Los periféricos pueden generar la señal de interrupción a través de una línea (hardware) que entra en el procesador.

El procesador puede estar en un momento dado con las interrupciones permitidas o inhibidas. Si están inhibidas, cuando le llega una interrupción el procesador la ignora y la interrupción queda a la espera de que el procesador la acepte. Al aceptar la interrupción el procesador hace varias cosas:

1. Deja de ejecutar instrucciones por el momento, inhibe las interrupciones y pasa al *modo supervisor*. Recuerde lo que dijimos al hablar de la CPU: un procesador puede estar en modo usuario o en modo supervisor; cuando está en modo usuario no deja ejecutar ciertas instrucciones (por ejemplo, las de acceso a los periféricos, o las que permiten cambiar de modo)[49].
2. Guarda en la memoria el estado de la ejecución en que se encontraba el programa, concretamente en el PCB del proceso (página 44), para más adelante seguir con su ejecución como si nada hubiese pasado.
3. Averigua la causa de la interrupción.
4. Si la causa ha sido la instrucción de interrupción de programa es que se trata de una llamada al sistema, y pasa a ejecutar el programa necesario del sistema operativo.
5. Si ha sido una interrupción externa pasa a ejecutar una **rutina de servicio**: un programa específico para la interrupción que forma parte del gestor del periférico.

Este conjunto de operaciones se llama **cambio de contexto**. Se pasa del contexto de ejecución del programa de usuario (con las interrupciones permitidas y en modo usuario) al contexto de ejecución del sistema operativo (con las interrupciones inhibidas y en modo supervisor). Una vez que ha terminado la ejecución de la rutina de servicio, o que se ha satisfecho la petición de la llamada al sistema, se vuelve a hacer un cambio de contexto: se recupera el PCB del proceso de usuario (o del que esté en cabeza de la cola del procesador, página 45).

[49] Algunos procesadores ofrecen la posibilidad de tener más modos de funcionamiento, que se llaman «anillos de protección», con distintos privilegios. Por ejemplo, con cuatro, el anillo 0 es el modo supervisor, y el anillo 3 el modo usuario. El sistema operativo puede utilizar los intermedios para partes del sistema, como los gestores de periféricos, que no necesitan todos los privilegios del modo supervisor

Ahora podemos responder a la pregunta «¿cómo se entera el sistema de que se ha escrito algo?». En el momento en que se pulsa una tecla, el controlador del teclado genera una interrupción que provoca el cambio de contexto. Si el nombre que introduce el usuario tiene seis caracteres se provocan siete interrupciones: los caracteres y la tecla «retorno». La rutina de servicio va introduciendo los seis caracteres en una zona de la memoria reservada para el teclado *(buffer)* y volviendo tras cada uno al modo usuario. Si no hay otros procesos preparados, el sistema queda en espera hasta que la rutina detecta el «retorno». El proceso `gettty` había escrito (con una llamada `WRITE` sobre el fichero de descriptor 2, o con una función de biblioteca) los caracteres «`login:`» y luego había pedido leer caracteres (con una llamada `READ` sobre el fichero de descriptor 1, o con una función de biblioteca) y había quedado bloqueado esperando la vuelta de esta llamada. Cuando la rutina de servicio encuentra el «retorno» avisa al sistema de gestión de ficheros, que devuelve al proceso bloqueado el número de caracteres leídos (siete), el proceso se pone preparado y en la cola del procesador, de modo que vuelve a ejecutarse.

Antes de seguir es conveniente aclarar una confusión de términos que a veces se produce cuando se aborda este asunto por primera vez: «superusuario» y «supervisor» no tienen nada que ver. Todos los procesos del superusuario se ejecutan con $uid = 0$, pero con un $pid > 1$. $pid = 1$ identifica al proceso `init`, y $pid = 0$ está reservado para el sistema operativo, que es el único que se ejecuta en modo supervisor (aunque partes de él pueden hacerlo en modo usuario, como veremos en el apartado siguiente). Por tanto, los procesos propiedad del superusuario (o *root*) se ejecutan en modo usuario y sólo pueden acceder a la zona de la memoria (las páginas) que el SGM les ha asignado.

### Servicio de las llamadas

Todas las llamadas al sistema se traducen, al nivel de la ABI (figura 2.9), por una secuencia de instrucciones de máquina que termina con la instrucción de interrupción de programa. Las instrucciones previas de la secuencia introducen en registros del procesador los datos necesarios para que el sistema pueda satisfacerla, entre ellos, un número entero que, por convenio, identifica a la llamada.

Por ejemplo, la llamada `WRITE` para escribir «`login:`» en el terminal (salida estándar), según la especificación de la SCI (página 36) es:

```
write(2, *punt, 7);
```

«2» es el descriptor de la salida estándar, «`*punt`» es un puntero a una dirección de memoria en la que previamente se han puesto los caracteres a escribir, y «7» es el número de caracteres (incluido un «retorno»).

Su traducción al lenguaje de máquina de un procesador serían cinco instrucciones[50]. Las cuatro primeras introducen en cuatro registros del procesador los parámetros de la llamada:

- Un número que identifica a `WRITE`. Por ejemplo, 4.
- «2» para indicar la salida estándar
- Una dirección de memoria, la que corresponde al puntero
- El número de caracteres, «7»

Finalmente vendría la instrucción de interrupción de programa Cuando el sistema operativo atiende a la interrupción recoge de los registros los datos (la ABI define el convenio de uso de los registros) y, en este caso, llama al sistema de gestión de ficheros, que a su vez hace uso del gestor de `tty`, para ocuparse de los detalles de la transferencia.

[50] En el apartado 10.9 lo veremos en el caso concreto del procesador ARM.

## El planificador *(scheduler)*

El **planificador** es el componente más importante de la gestión de procesos. Ya hemos resumido su función antes (página 45): «es la parte del sistema responsable de repartir la CPU entre los procesos preparados». Y también hemos dicho que coloca estos procesos en una cola (figura 2.18). Los dos objetivos principales del planificador son aprovechar al máximo la CPU, manteniendo el procesador (o los procesadores) siempre ocupados, y conseguir tiempos de respuesta aceptables para todos los programas, especialmente los interactivos. Pero estos objetivos pueden entrar en conflicto. Por ejemplo si el número de procesos está al límite de la capacidad de memoria, para activar un proceso hay que recurrir al intercambio de otro proceso con el disco (página 2.5), y esto empeora el tiempo de respuesta.

Por su repercusión en el funcionamiento de todo el sistema se han diseñado muchas estrategias de planificación atendiendo a objetivos diversos. El planificador es un proceso que tiene *pid*=0 (de hecho, es el que se encarga en la inicialización de crear `init`, *pid*=1) y entra en juego periódica o esporádicamente. Vamos a resumir las estrategias respondiendo a tres cuestiones:

- ¿Con qué frecuencia se ejecuta?
- ¿Qué evento provoca que se ejecute?
- ¿Qué algoritmo utiliza?

◆ Por lo que respecta a la frecuencia, se distinguen tres tipos:

- Un planificador **a largo plazo** (o planificador de trabajos) es típico de los sistemas por lotes. Sólo se ejecuta cuando un proceso termina su ejecución en un procesador o queda bloqueado y el planificador debe activar a otro. Como funciona infrecuentemente, puede utilizar algoritmos elaborados para conseguir la máxima potencia (número total de trabajos ejecutados en plazo de tiempo grande) teniendo en cuenta la necesidad de recursos (CPU, memoria y periféricos) de los trabajos pendientes. Pero nos estamos centrando en sistemas interactivos y multiusuario, por lo que no comentaremos nada más sobre este tipo de planificación.
- Un planificador **a corto plazo** (o planificador de la CPU) se ejecuta con mucha frecuencia (entre 10 y 100 milisegundos) y debe ser muy eficiente: si se ejecuta cada diez milisegundos y tarda dos en ejecutarse se está gastando el 18 % del tiempo de la CPU para la planificación.
- Se suele llamar planificador **a medio plazo** al que entra en juego cuando la ocupación de memoria es tan alta que hay que hacer el intercambio. No tiene una frecuencia determinada de funcionamiento (depende de la carga del sistema en cada momento), pero normalmente está entre los dos casos anteriores.

◆ ¿Qué evento provoca que se ejecute el planificador? En principio, por lo que acabamos de decir, un planificador a corto plazo se ejecuta con una frecuencia determinada por una señal de reloj, pero hay otros acontecimientos:

- Las interrupciones. Una de las causas de interrupción es esa señal de reloj. El reloj tiene una frecuencia muy alta, de modo que la ejecución de una instrucción ocupa varios ciclos de reloj. Si su frecuencia es, por ejemplo, 1 GHz (período $10^{-9}$ segundos) y la frecuencia de planificación debe ser 100 milisegundos, un circuito temporizador generará la interrupción cada $0{,}1/10^{-9} = 10^8$ ciclos de reloj. Pero hay otras causas de interrupción que también «despiertan» al planificador. Por ejemplo, la generada por un controlador de periférico que anuncia el fin de la operación que había solicitado un proceso y que había quedado bloqueado.
- La entrada de un proceso nuevo como consecuencia de una llamada `FORK`.
- El bloqueo de un proceso que pide una operación como `WRITE`, `WAIT`, etc.
- El fin de un proceso por `EXIT` o `KILL`.

Ahora podemos definir con más precisión la diferencia entre «multitarea cooperativa» y «multitarea apropiativa» (página 19). Si el planificador sólo atiende a los dos últimos eventos (bloqueo o fin de un proceso) entonces hace multitarea cooperativa; si atiende a los cuatro es apropiativa.

◆ Se han propuesto, con mayor o menor grado de aceptación, muchos algoritmos de planificación. Los sistemas suelen incorporar varios planificadores y permiten asignar los programas a uno u otro y de ajustar parámetros de la planificación en función de sus necesidades de recursos[51]. Entre los algoritmos más conocidos están estos cuatro (los dos primeros ligados más a una estrategia cooperativa y los otros dos a una apropiativa):

- **Por orden de llegada**, o **FCFS** *(First-Come-First-Served)*. Es el algoritmo más sencillo, tanto en su concepción como en su implementación: basta con añadir a la cola del procesador los PCB de los procesos conforme se van creando o saliendo del estado bloqueado. Cuando un proceso activo se bloquea o termina se saca de la cola y se activa el siguiente. A menudo se implementa con **prioridades estáticas**: a cada nuevo proceso se le asigna un valor de prioridad, y hay tantas colas como prioridades diferentes. Para activar un proceso se elige el que está en cabeza de la cola de mayor prioridad.
- **Por elección del más corto**, o **SJF** *(Shortest Job First)*. Los programas tienen «ráfagas *(bursts)* de CPU» y «ráfagas de entrada/salida». «El más corto» significa «el que requiera menos tiempo en la siguiente ráfaga de CPU». Naturalmente, esto no se sabe, pero se puede estimar en función de la duración de las ráfagas anteriores. Si se analiza con una mezcla de programas se ve que globalmente se consigue mejor rendimiento en el uso de la CPU que con FCFS. Un problema es que puede dar lugar a que algunos procesos entren en **inanición** *(starvation)*: que nunca lleguen a ejecutarse porque hay muchos más cortos.
- **Por turno rotatorio**, o **RR** *(Round-Robin)*. Es una mejora sencilla de FCFS: se establece un período de tiempo máximo para cada proceso que se llama «cuanto»; si el proceso activo lo sobrepasa se pone al final de la cola de preparados (si hay varias colas, de la que le corresponda según su prioridad estática).
- **Por prioridades dinámicas**. Asociado o no a la existencia de «cuanto», evita la inanición y es el algoritmo más utilizado en los sistemas multiusuario. Se basa en ir reduciendo paulatinamente la prioridad del proceso activo y aumentando la de los preparados conforme pasa el tiempo. Veamos con un poco de detalle cómo funciona.

En la planificación por prioridades dinámicas todo proceso tiene asociado en todo momento un valor de prioridad. Es un número entero, $P$, cuyo rango de valores varía de unas implementaciones a otras. $P$ es la suma de dos componentes, una estática, $B$ (prioridad de base) y otra dinámica, $T$ (tiempo de CPU). Los valores de $B$ y $T$ están permanentemente registrados en el PCB (figura 2.17). El valor de $B$ se fija en la creación del proceso y el que va cambiando es $T$. Los rangos de valores de $B$ y $P$ varían entre implementaciones. Para concretar, supongamos que:

- el rango de $P$ es de 0 (prioridad máxima) a 19 (prioridad mínima),
- el rango de $B$ es de 0 (prioridad máxima) a 10 (prioridad mínima):
- se produce una interrupción de reloj cada 100 milisegundos y su rutina de servicio da paso al planificador.

Cada vez que se ejecuta, el planificador incrementa en una unidad el $T$ *del proceso activo*, pero una de cada diez veces (es decir, cada segundo) divide por dos los $T$ *de todos los procesos*. De esta manera,

[51] En Linux puede consultar «`man 7 sched`» y las páginas de la sección 2 a las que remite.

conforme pasan los segundos, en los procesos inactivos se va reduciendo $T$, que es así una medida del tiempo de CPU consumido «recientemente». Asimismo, cada segundo se calculan los nuevos valores de $P = B + T$ *para todos los procesos* y se reordena la cola de procesos por el valor de $P$ (en cabeza el que tiene el menor valor). Resumiendo, el algoritmo del planificador, que se ejecuta cada interrupción de reloj (y también cuando se da algunos de los otros eventos, pero entonces no se alteran los valores de $T$, sólo se reordena la cola de procesos), es:

```
para el proceso activo:
        T = T+1;
K = K+1;
si K = 10 entonces {
        K = 0;
        para todos los procesos:
            T = T/2;
            P = B+T
        }
actualizar la cola del procesador;
ejecutar el distribuidor
```

Observaciones y aclaraciones:

- En el arranque del sistema, a la variable `K` se le daría el valor inicial «0», y cuando se crea un proceso se le pone $T = 0$.
- Puede usted comprobar que el valor máximo que puede llegar a tomar $P$ es 19 (mínima prioridad).
- Un proceso creado con `FORK` hereda la prioridad del padre.
- Si hay un «empate» de prioridad entre dos procesos puede resolverse de manera arbitraria: transcurrido un segundo, si se ha activado uno de ellos en detrimento del otro su prioridad se habrá reducido, y dejará de ser el primero.
- La prioridad de un proceso bloqueado por una petición de entrada/salida crece rápidamente, de manera que en cuanto termina la operación (por ejemplo, la entrada de un dato desde el terminal) pasa a activarse inmediatamente.
- La operación que puede llevar más tiempo, dependiendo del número de procesos preparados, es la de ordenar la cola. El resto de operaciones son muy rápidas y se ejecutan con pocas instrucciones del procesador. En particular, «`T = T/2`» es una división entera, y se realiza con una sola instrucción (desplazamiento a la derecha, apartado 4.4).
- Cuando un proceso se crea se le da un valor por defecto a su prioridad de base, por ejemplo, $B = 5$. Hay llamadas al sistema que permiten, en cualquier momento, cambiar este valor (pero sólo un proceso de superusuario puede bajarlo de 5). Y también hay un programa de utilidad que se llama «`nice`». Antepuesto al nombre de cualquier orden permite incrementar la prioridad de base del proceso. Por ejemplo, «`nice -3 <orden>`» resta 3 a $B$. Esta facilidad le ofrece al usuario la oportunidad de ser *«nice»* (amable) con los demás: si ordena ejecutar un programa que requiere mucho tiempo de CPU y no tiene prisa en obtener los resultados puede reducir con ella la prioridad del proceso, dando así más tiempo de CPU a los procesos de otros usuarios[52].
- Finalmente, en la última línea del pseudocódigo anterior dice «ejecutar el distribuidor». Veamos de qué se trata.

[52] No cabe esperar que un usuario utilice esta orden directamente (desde el teclado), pero sí resulta muy útil para programas del sistema que no son urgentes. Estos programas pueden ser guiones *(scripts*, ficheros con órdenes) en los que se incluye directamente `nice`, o programas en ensamblador o lenguaje de alto nivel, en los que se utiliza la llamada correspondiente, `SETPRIORITY`.

### El distribuidor *(distpatcher)*

El distribuidor es el que se ocupa del trabajo final de activar al proceso que ha quedado en cabeza de la cola del procesador y cambiar del contexto de modo supervisor al modo usuario. Es decir:

1. Saca de la cola el PCB que está en la cabeza.
2. De este PCB recupera los datos que se refieren al estado de la ejecución del proceso que va a pasar a ser activo (figura 2.17). Concretamente, copia en los registros del procesador los valores guardados, para que el proceso continúe desde el punto del programa en el que se interrumpió, y los contenidos de la TBL (figura 2.21), también guardados en el PCB, que son los que asignan páginas de memoria a la imagen del proceso.
3. En un registro del procesador pone el modo usuario y permite interrupciones.
4. Por último introduce en el registro contador de programa el valor guardado en el PCB, lo que hace que inmediatamente se pase a ejecutar el proceso activado.

Algunos autores consideran estas operaciones incluidas en el planificador, identificando así «distribuidor» con «planificador a corto plazo». No tiene mayor importancia (sólo es una cuestión de convenio), pero la separación de funciones es interesante porque es otro ejemplo de la distinción entre «mecanismo» y «política» (página 41): el planificador determina la política, y el distribuidor implementa el mecanismo. Este mecanismo no restringe la política: el mismo distribuidor sirve para distintos planificadores.

De los pasos anteriores, el que más tiempo de CPU consume es el segundo. Como los cambios de contexto son muy frecuentes es necesario que sean lo más rápidos posible. Los procesadores hardware ayudan a conseguirlo duplicando registros o proporcionando instrucciones que guardan en memoria y recuperan los registros en la misma instrucción. Veremos algún ejemplo en el tema «Procesadores hardware».

### Ciclo de vida de los procesos

Ya sabemos que un proceso puede estar activo, preparado o bloqueado. Ahora podemos precisar las causas que hacen cambiar a un proceso de un estado a otro y presentar otros estados posibles.

Todo proceso nace como consecuencia de una llamada `FORK` desde otro proceso (salvo `init`, *pid* = 1, que es el «Adán» en el universo de los procesos; el planificador, *pid* = 0, sería el «dios»). Como siempre que hay una llamada (o cualquier otra interrupción), se produce un cambio de contexto al modo supervisor, se realiza el servicio que corresponda (en el caso de `FORK`, creación de la imagen en memoria y del PCB del nuevo proceso) y al final se ejecuta el planificador, que pone el PCB en la cola del procesador, y el distribuidor, que activa al que está en cabeza y vuelve al contexto del modo usuario.

Las interrupciones, en general, hacen que el proceso que está activo pase a preparado (pero si después del planificador sigue siendo el más prioritario seguirá siendo activo), salvo en dos ocasiones:

- las provocadas por llamadas «bloqueantes»: las que piden una operación con algún periférico y `WAIT`, con las que el proceso se saca de la cola del procesador y pasa al estado «bloqueado»;
- las provocadas por llamadas que hacen «morir» al proceso: `KILL` (que, en general, envía una señal a otro proceso; una de estas señales es «matar», página 38) y `EXIT`, que normalmente hacen desaparecer todo rastro del proceso (pero enseguida veremos que puede ser que el proceso no deba morir del todo).

Los procesos preparados y los bloqueados también pueden matarse con `KILL` desde otro proceso. El proceso activo también puede morir por algunas causas de interrupción. Por ejemplo, si intenta

ejecutar alguna operación no permitida en modo usuario, o si intenta acceder a una página que no le corresponde, en cuyo caso la MMU provoca una interrupción.

Los procesos bloqueados dejan de estarlo cuando desaparece la causa del bloqueo. Por ejemplo, cuando el controlador del disco avisa (con una interrupción) de que ha concluido la transferencia que se había hecho con `READ` o `WAIT`, o cuando termina el hijo (con `EXIT`) por el que un proceso había quedado bloqueado con `WAIT`.

La figura 2.28 resume gráficamente lo anterior, y muestra tres estados nuevos:

- «Preparado en memoria secundaria» y «bloqueado en memoria secundaria» se explican por su propios nombres: el intercambiador es el responsable, cuando interviene, de pasar de estos estados a sus equivalentes en memoria.

- «Zombi» tampoco es difícil de entender, aunque la explicación es algo más sutil. A veces, el objetivo de `FORK` es ejecutar otro programa (con `EXEC`) en el nuevo proceso, que no tiene nada que ver con su padre. Pero otras veces, entre el padre y el hijo se establecen vías de comunicación (por ejemplo, el hijo realiza cierto procesamiento cuyos resultados deja en un fichero, de donde los lee el padre). Cuando el hijo termina con `EXIT` deja en su PCB el «status» de terminación (página 38) y, si ha hecho alguna operación con ficheros cuyos descriptores ha heredado del padre, estos ficheros deben permanecer abiertos para el padre. Si este padre no puede seguir hasta que el hijo no haya acabado, utiliza una llamada `WAIT` que, como sabemos, provoca su bloqueo hasta que en el hijo se ejecute `EXIT`. Pero también puede ser que el hijo llegue a `EXIT` (o que otro proceso intente «matarlo») antes de que el padre haya llegado a `WAIT`. En tal caso, el sistema no debe desembarazarse del todo del proceso hijo (que es lo que normalmente hace con un proceso cuando en él aparece `EXIT`), porque tiene una información para el padre guardada en su PCB. Y sin embargo, a todos los demás efectos, está «muerto» (no tiene que volver a activarse, ni tiene por qué seguir ocupando espacio en la memoria). Por esto, se dice que está **zombi**. El sistema lo señala como tal, y cuando ejecuta la `WAIT` del padre (o cuando se hace terminar a éste con `KILL`), lo «mata» definitivamente eliminando su PCB de la tabla de procesos. Por tanto, un proceso zombi no es más que una entrada (su PCB) en la tabla de procesos que se libera cuando el padre ejecuta `WAIT` o cuando muere, pero su imagen en memoria (las páginas que ocupaba) ya ha quedado liberada.

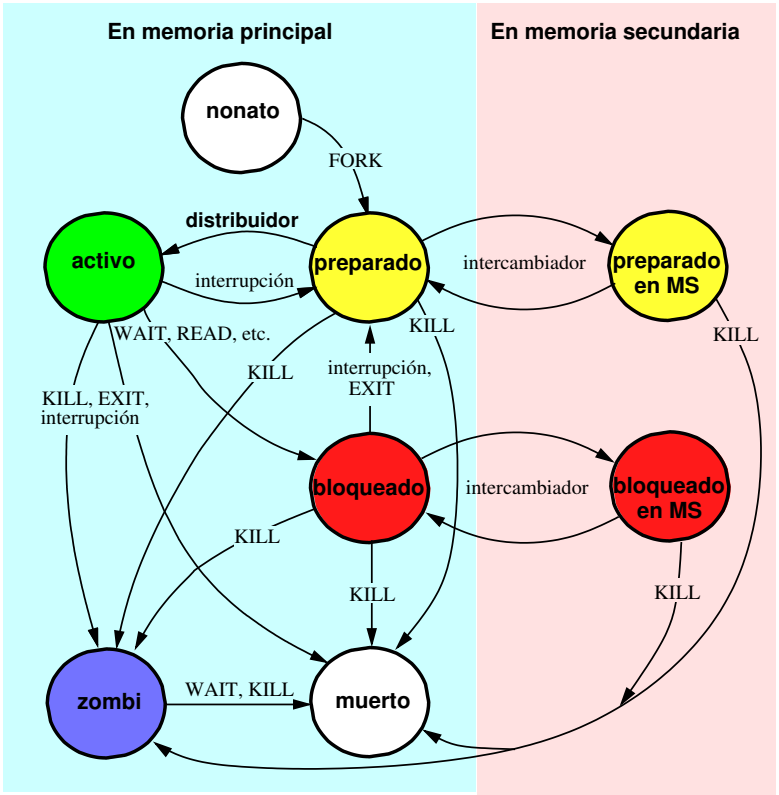



**Figura 2.28**: Estados y transiciones de los procesos.

Para terminar, una cuestión no menos sutil: si el padre termina antes que el hijo, ¿qué ocurre con éste? Porque todo proceso (salvo `init`) tiene un padre (cuyo *pid* está en su PCB, figura 2.17) al que

puede informar sobre su condición de terminación. Pues bien, estos procesos huérfanos son adoptados por `init`: en «pid del padre» de su PCB se pone «1». Como `init` está normalmente bloqueado con una `WAIT` (página 38), cuando este hijo adoptado termine informará a su padre adoptivo `init` (lo que normalmente es irrelevante para éste).

### Comunicación entre procesos

La comunicación entre procesos, o **IPC** *(Inter-Process Communication)* es algo esencial en redes de ordenadores, donde procesos en distintas máquinas se comunican siguiendo protocolos. Pero aun en sistemas operativos centralizados, que es lo que estamos estudiando, los procesos que se ejecutan concurrentemente pueden ser independientes o cooperantes. La finalidad de que dos o más procesos cooperen puede obedecer a varios motivos: separar distintas subtareas de una tarea global (por ejemplo, editar un documento y al mismo tiempo ejecutar un corrector ortográfico sobre el mismo documento), compartir información (por ejemplo, varios usuarios trabajando sobre un mismo fichero), facilitar la ejecución en paralelo cuando la CPU tiene varios procesadores, etc.

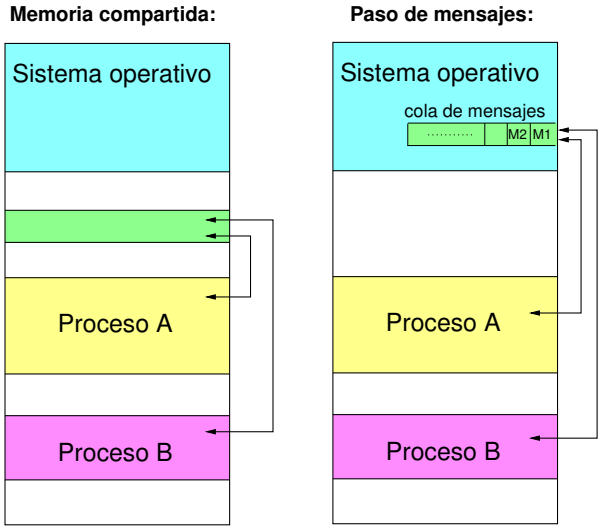



**Figura 2.29**: Modelos para IPC.

Hay dos modelos básicos para la comunicación entre procesos (figura 2.29):

- **Memoria compartida:** los procesos intercambian datos accediendo a una zona de memoria común.
- **Intercambio de mensajes:** los procesos se comunican enviándose mensajes que incluyen un *pid* del remitente y un *pid* del destinatario. Como se implementa haciendo uso de llamadas al sistema, es más lento que el uso de la memoria compartida, pero más adecuado para el caso de comunicaciones entre ordenadores.

La mayoría de los sistemas operativos tienen facilidades para utilizar ambos modelos.

## 2.7. El *kernel*. Sistemas monolíticos, *microkernel* e híbridos

Por poco que haya usted leído u oído sobre sistemas operativos alguna vez habrá encontrado la palabra *«kernel»*[53]. En este documento ya ha aparecido dos veces, en las páginas 16 y 21.

A veces se dice que el *kernel* es «la parte del sistema que siempre está en la memoria principal». Pero esto no es cierto: hay componentes (módulos) que pueden formar parte del *kernel*, como los gestores de dispositivos que se cargan sólo si son necesarios; además, hay procesos, como la mayoría de los demonios, que están siempre en la memoria y no son del *kernel*.

Otra definición incorrecta, por parcial, es: «la parte del sistema operativo que define una API para los programas de aplicación y algunos programas del sistema, con una interfaz para los gestores de dispositivos». Es parcial porque sólo incluye a los que se llaman «sistemas monolíticos».

[53] Puede traducirse (y en algunos documentos se hace) por «núcleo». Pero el término *«kernel»* está tan extendido que pasa lo mismo que con *«cache»* (nota 9, página 24): el empeño en mantener un purismo lingüístico no debe dificultar la comprensión.

Más correcta, pero imprecisa, es la definición del FOLDOC: «la parte esencial de Unix o de otros sistemas operativos, responsable de la asignación de recursos, interfaces del hardware en bajo nivel, seguridad, etc.»

Se puede definir de una manera mucho más precisa diciendo, simplemente:

*kernel:* el conjunto de programas que se ejecutan en modo supervisor.

Por eso no hemos hablado hasta ahora de *«kernel»*: hacía falta entender bien lo que significa «modo supervisor», también llamado **modo kernel**.

## *kernels* monolíticos

En un *kernel* monolítico todos los subsistemas del sistema operativo funcionan en modo supervisor, con las interrupciones inhibidas. Realmente, desde que al principio del apartado 2.4 dijimos que íbamos a concretar con sistemas monolíticos, podríamos haber sustituido «sistema operativo» por «*kernel*».

La mayoría de los sistemas de tipo Unix tienen, en principio, un diseño monolítico. Su implementación es un programa binario único (salvo los módulos, que ahora comentaremos) que da lugar a un único proceso con un único espacio de direccionamiento. Esto implica que, como las interrupciones están inhibidas durante este proceso único, hasta que no completa el servicio de una no puede ejecutarse ningún otro proceso. Se dice que es un *kernel no apropiativo* (página 62), en el sentido de que ninguna de sus tareas puede interrumpir a otra; obviamente, si el planificador lo permite, el *kernel* da un servicio apropiativo: puede suspender a los procesos que se ejecutan en modo usuario. En las aplicaciones de tiempo real este modo de funcionamiento no es admisible, porque ciertas tareas del *kernel* pueden tardar un tiempo excesivo. Muchos sistemas se han modificado para que estas tareas puedan interrumpirse, resultando así un *kernel apropiativo*.

La memoria en un sistema monolítico queda dividida en dos grandes zonas:

- El **espacio del *kernel***, donde está cargado el programa binario que lo forma.
- El **espacio de usuario**, donde se cargan los demás programas.

El *kernel* tiene acceso a toda la memoria cambiando los contenidos de su tabla de páginas (figura 2.20), cosa que no pueden hacer los procesos de usuario: no tienen acceso al espacio del *kernel*, y un intento de acceso a la MMU en modo usuario provoca una interrupción.

La implementación típica de un sistema monolítico tiene poca estructura. Un subprograma, el que, junto con los gestores de periféricos, está más cercano al hardware y se suele programar en lenguaje ensamblador, reconoce las interrupciones y activa a uno de los subsistemas (gestión de procesos, gestión de memoria, gestión de ficheros o gestión de periféricos). Hay rutinas con funciones comunes (manejo de colas, de tablas, etc.) de las que pueden hacer uso todos los subsistemas (figura 2.30). Tiene la ventaja, con respecto a otras implementaciones que veremos, de ser eficiente: una vez que se entra en el *kernel* todas las operaciones necesarias para satisfacer la demanda se ejecutan sin salir de él y las comunicaciones entre las partes son mucho más rápidas. Pero a medida que las funciones proporcionadas por el *kernel* aumentan crece la dificultad de implementar el sistema sin caer en problemas provocados por inconsistencias e interacciones indeseadas entre unos componentes y otros.

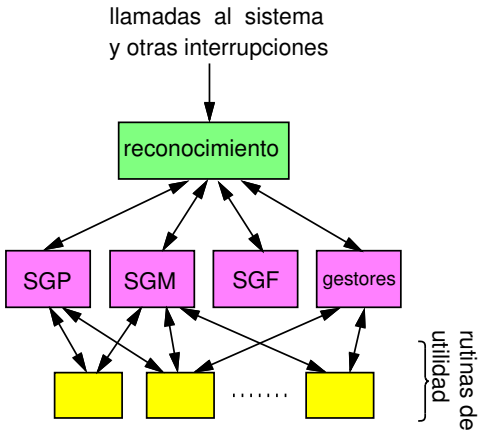



**Figura 2.30**: Estructura de un *kernel* monolítico.

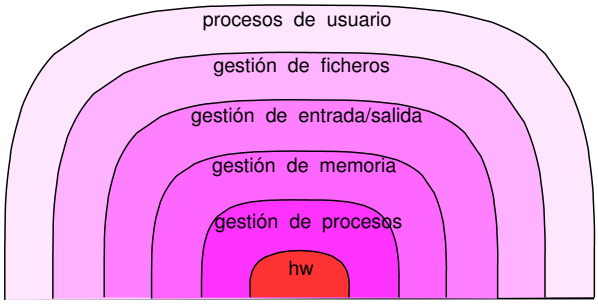



**Figura 2.31**: Estructura en capas.

Una propuesta de diseño estructurado, ya antigua (1968) pero origen de muchas innovaciones posteriores, fue el sistema THE de Dijkstra, diseñado en la **T**echnische **H**ogeschool **E**indhoven, que luego se adoptó en Multics *(Multiplexed Information and Computing Service)*, precursor de Unix. La idea es estructurar el sistema en capas (figura 2.31). Cada capa se implementa utilizando exclusivamente las facilidades proporcionadas por las capas inferiores, de manera que en cada capa quedan ocultos los detalles de su implementación, como los procedimientos y estructuras de datos que utiliza. Así, cuando un proceso de usuario solicita una operación sobre un fichero con una llamada al sistema, es la capa de gestión de ficheros la que se ocupa de pasarla a la siguiente, la de gestión de periféricos. Pero ésta no actúa inmediatamente: debe pasarla a la de gestión de memoria que a su vez la pasa a la que se ocupa de la planificación, y ésta al hardware.

Seguramente esto le recuerda algo: es similar a la idea que se aplica (con más éxito) en las redes de datos, con las capas de protocolos que usted ha estudiado en la asignatura «Redes de comunicaciones». El motivo de que no se haya seguido tan fielmente en los sistemas operativos es la eficiencia: cada paso de una capa a la otra tiene un tiempo de ejecución, lo que implica al final retardos considerables en el tratamiento de las llamadas. Pero la idea de la modularidad se ha aplicado de otras formas.

## Sistemas modulares

Gran parte de los componentes incluidos en un sistema operativo sólo se utilizan para un hardware determinado. Esto es especialmente cierto en el caso de los gestores de periféricos. Como son muchos los periféricos susceptibles de utilizarse, el *kernel* tiene que incluir a todos. Por ejemplo, hay docenas de controladores de gráficos diferentes, y para cada uno hace falta un gestor. Pero en una instalación sólo se utiliza uno de estos controladores. Lo mismo puede decirse de los discos, las impresoras, las tarjetas de red...[54] Si el *kernel* monolítico se implementa incluyendo todos estos componentes en un único programa binario se está utilizando sin necesidad mucho espacio de memoria.

Casi todas las implementaciones actuales de Unix siguen una idea más razonable: los componentes que, aun formando parte del *kernel*, pueden necesitarse en unas ocasiones pero no en otras, son programas ejecutables independientes, llamados **módulos**. El programa ejecutable principal sólo incluye las funcionalidades esenciales: gestión de procesos, de memoria y de ficheros. Los módulos se cargan bien en el arranque o bien dinámicamente durante la ejecución, cuando son necesarios. Incluso esas funcionalidades esenciales pueden también «modularizarse». Por ejemplo, para la gestión de procesos puede haber módulos que implementen distintos algoritmos de planificación de modo que se pueda cargar en la inicialización uno u otro. Otro ejemplo es la gestión de ficheros: el *kernel* puede reconocer y tratar con varios sistemas de ficheros (página 48); los detalles del procesamiento para cada sistema de ficheros se programan en módulos, y lo que se incluye en el programa principal es una funcionalidad genérica que se llama **VFS** *(Virtual Filesystem Switch)* que si, por ejemplo, se inserta un CD, automáticamente carga el módulo correspondiente al sistema de ficheros, ISO 9660.

Otra ventaja del enfoque modular es su flexibilidad para añadir a la máquina un periférico para el

[54] De ahí que en los programas fuente del *kernel* los gestores *(drivers)* representen más del 50 % de líneas de código, como vimos en la nota 38, página 52.

que no hay previsto un gestor: ese gestor se puede programar y compilar como módulo sin necesidad de volver a compilar todo el *kernel*.

Por todo esto, el antiguo debate entre los partidarios del *kernel* monolítico y los del *microkernel*[55] está prácticamente superado, al menos en la situación actual.

### *Microkernels*

El hecho de que un *kernel* tenga módulos que se cargan dinámicamente y no sea un único programa ejecutable no implica que deje de ser un *kernel* monolítico, puesto que todo se ejecuta en *modo kernel*[56].

Contrastando con el diseño monolítico, el principio del *microkernel* es reducir al mínimo las partes que se ejecutan en modo supervisor[57].

La gestión de ficheros, la implementación de la comunicación entre procesos de usuario (IPC), los gestores, se llaman «servidores» y se ejecutan como procesos en modo usuario. La función principal del *microkernel* se reduce al reconocimiento de interrupciones, a gestionar la memoria, y a proporcionar unos mecanismos básicos de planificación y de comunicación entre los servidores y los programas de aplicación (clientes) (figura 2.32).

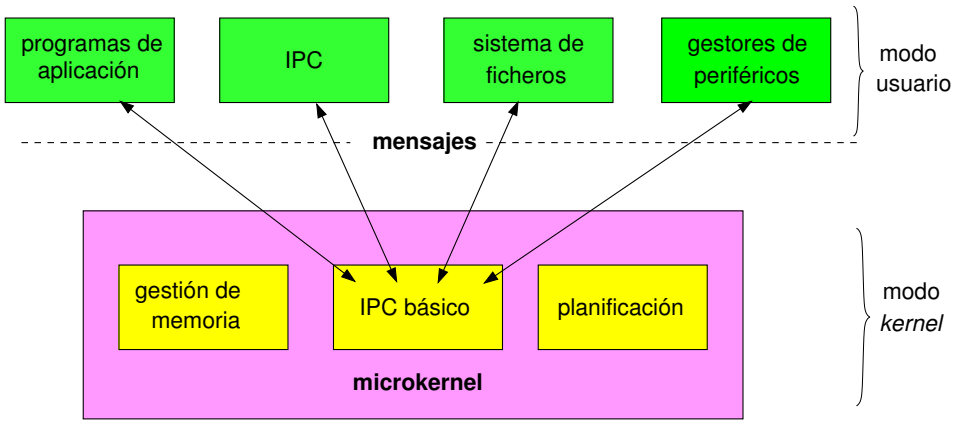



**Figura 2.32**: Estructura de un *microkernel*.

Como los servidores se ejecutan en procesos independientes, no es posible la comunicación directa ni por memoria compartida con ellos ni entre ellos, como se hace en un *kernel* monolítico. La comunicación se establece mediante paso de mensajes. Por ejemplo, si un cliente pide acceder a un fichero tiene que interactuar con el servidor de ficheros, pero no lo hace directamente, sino a través de mensajes con el *microkernel*. Esto tiene como consecuencia que el sistema es menos eficiente.

Sin embargo, los *microkernels* son más robustos: si un componente falla no falla todo el sistema, como sí puede ocurrir en un *kernel* monolítico.

El enfoque de *microkernel* es un caso claro de aplicación del principio de separación entre política y mecanismo (página 41). Los componentes que forman el *microkernel* proporcionan mecanismos básicos sobre los cuales se pueden implementar distintas políticas de asignación de recursos.

[55] Hay un documento histórico de lectura interesante: el intercambio de mensajes en 1992 entre Linus Torvals, a la sazón estudiante en la Universidad de Helsinki, y Andrew Tanenbaum, prestigioso profesor de la Universidad Libre (Vrije Universiteit) de Amsterdam y defensor del enfoque *microkernel*, que opinaba que Linux tenía un diseño «obsoleto»: `http://oreilly.com/catalog/opensources/book/appa.html`.

[56] Al ser Linux un sistema modular con esas características (salvo por algunas excepciones, como módulos en espacio de usuario) no tiene sentido decir «el *kernel* de Linux»: Linux *es un kernel.*

[57] MINIX 3 es un *microkernel* que tiene unas 9.000 líneas de código frente a los 12 millones de Linux (nota 30, página 43).

## Sistemas híbridos

Actualmente pocos sistemas son estrictamente monolíticos o estrictamente *microkernel*. Los diseñadores tratan de combinar las ventajas de los primeros (respuesta más eficaz al basarse en memoria compartida en el espacio del *kernel* y no en mensajes) con las de los segundos (mayor modularidad y robustez). La figura 2.33 ilustra la esencia de las diferencias generales entre los tres enfoques.

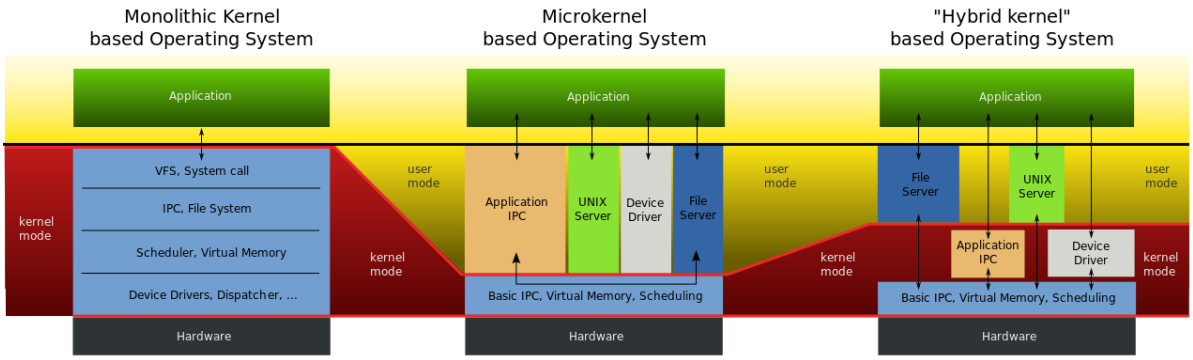



**Figura 2.33**: Kernel monolítico, microkernel e híbrido
Fuente: `http://en.wikipedia.org/wiki/Image:OS-structure.svg`.

Ejemplos de sistemas híbridos son Windows NT y XNU *(«X is Not Unix»)*, que es el *kernel* de Apple para los sistemas operativos OS X y iOS. Incluso en sistemas tradicionalmente monolíticos, como Linux, freeBSD y Solaris, ciertas funcionalidades se pueden implementar como módulos que se ejecutan en el espacio de usuario. Por ejemplo, FUSE *(Filesystem in Userspace)* es un módulo que permite implementar sistemas de ficheros virtuales en modo usuario.

# Capítulo 3

# Representación de datos textuales

---

Relacionados con el Tema 2 de la asignatura, la Guía de Aprendizaje establece estos resultados:

- Conocer los convenios de representación binaria, transmisión y almacenamiento de datos textuales, numéricos y multimedia.
- Conocer los principios de los algoritmos de detección de errores y de compresión.

En este capítulo veremos los convenios de representación de textos, en el capítulo 4 los de datos numéricos, y en el capítulo 5 los de datos multimedia. La detección de errores y la compresión se tratan en el capítulo 6. El capítulo 7 se dedica a los convenios de almacenamiento de los datos en ficheros.

Si vamos a tratar de «representación de textos» hay que empezar precisando qué es lo que queremos representar. Lo más sencillo es el *texto plano*: el formado por secuencias de **grafemas** (a los que, más familiarmente, llamamos *caracteres*), haciendo abstracción de su tamaño, color, tipo de letra, etcétera (que dan lugar a los **alógrafos** del grafema). Los grafemas son los caracteres visibles, pero el texto plano incluye también caracteres que no tienen una expresión gráfica y son importantes: espacio, tabulador, nueva línea, etcétera.

Empezaremos con la representación del texto plano. Para los distintos alógrafos los programas de procesamiento de textos utilizan convenios propios y muy variados (apartado 3.6), pero para el texto plano veremos que hay un número reducido de estándares. Y terminaremos el capítulo comentando una extensión de la representación textual muy importante en telemática: el hipertexto.

## 3.1. Codificación binaria

En un acto de comunicación se intercambian *mensajes* expresados en un lenguaje comprensible para los agentes que intervienen en el acto. Los mensajes de texto están formados por cadenas de caracteres. El conjunto de caracteres (o símbolos básicos) del lenguaje es lo que llamamos *alfabeto*. Para la comunicación entre personas los lenguajes occidentales utilizan un alfabeto latino. Pero si en el proceso de comunicación intervienen agentes artificiales los mensajes tienen que expresarse en su lenguaje, normalmente basado en un alfabeto binario.

La **codificación** es la traducción de un mensaje expresado en un determinado alfabeto, $\mathcal{A}$, al mismo mensaje expresado en otro alfabeto, $\mathcal{B}$. La traducción inversa se llama **descodificación** (o **decodificación**). Si los mensajes han de representarse en binario, entonces $\mathcal{B} = \{0,1\}$.

La codificación se realiza mediante un **código**, o correspondencia biunívoca entre los símbolos de los dos alfabetos. Como normalmente tienen diferente número de símbolos, a algunos símbolos de un alfabeto hay que hacerles corresponder una secuencia de símbolos del otro. Por ejemplo, si $\mathcal{A}$ = {a,b,c} y $\mathcal{B}$ = {0,1}, podríamos definir el código:

| $\mathcal{A}$ | $\mathcal{B}$ |
|---|---|
| a | 0 |
| b | 1 |
| c | 01 |

Ahora bien, tal código no sería útil, porque la descodificación es ambigua. Así, el mensaje codificado «011» puede corresponder a los mensajes originales «abb» o «cb». Si «a» se codifica como «00» el código ya no es ambiguo. Pero lo más sencillo es que el código sea de longitud fija, es decir, que el número de símbolos de $\mathcal{B}$ que corresponden a cada símbolo de $\mathcal{A}$ sea siempre el mismo. Podríamos establecer el código:

| $\mathcal{A}$ | $\mathcal{B}$ |
|---|---|
| a | 00 |
| b | 01 |
| c | 11 |

o bien:

| $\mathcal{A}$ | $\mathcal{B}$ |
|---|---|
| a | 01 |
| b | 00 |
| c | 10 |

o bien...

Los códigos BCD, ASCII y las normas ISO que veremos a continuación son de longitud fija. Pero UTF-8 (apartado 3.4) es de longitud variable. También veremos un ejemplo de codificación con longitud variable relacionado con la compresión en el apartado 6.5.

### Códigos BCD

Codificar en binario un conjunto de $m$ símbolos con un código de longitud fija requiere un número $n$ de bits tal que $2^n \geq m$. Por tanto, para codificar los diez dígitos decimales hacen falta cuatro bits. Estos códigos se llaman **BCD** (decimal codificado en binario). El más común es el BCD «natural» (o BCD, a secas), en el que cada dígito decimal se representa por su equivalente en el sistema de numeración binario.

| | **BCD natural** | **Aiken** | **exceso de 3** |
|---|---|---|---|
| 0 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0100 |
| 2 | 0010 | 0010 | 0101 |
| 3 | 0011 | 0011 | 0110 |
| 4 | 0100 | 0100 | 0111 |
| 5 | 0101 | 1011 | 1000 |
| 6 | 0110 | 1100 | 1001 |
| 7 | 0111 | 1101 | 1010 |
| 8 | 1000 | 1110 | 1011 |
| 9 | 1001 | 1111 | 1100 |

**Tabla 3.1**: Tres códigos BCD.

Como cuatro bits permiten codificar dieciséis símbolos de los que sólo usamos diez, hay $V_{16,10} = 16!/6! \approx 2{,}9 \times 10^{10}$ códigos BCD diferentes. En la tabla 3.1 se muestran tres: el BCD «natural», u «8-4-2-1» (los números indican los pesos correspondientes a cada posición, es decir, que, por ejemplo, $1001 = 1\times8+0\times4+0\times1+1\times1 = 9$) y otros dos que presentan algunas ventajas para simplificar los algoritmos o los circuitos aritméticos: el código de Aiken, que es un código «2-4-2-1», y el llamado «exceso de 3».

## 3.2. Código ASCII y extensiones

En nuestra cultura se utilizan alrededor de cien grafemas para la comunicación escrita contando dígitos, letras y otros símbolos: ?, !, @, $, etc. Para su codificación binaria son necesarios al menos siete bits ($2^6 = 64; 2^7 = 128$). Para incluir letras con tilde y otros símbolos es preciso ampliar el código a ocho bits ($2^8 = 256$).

El **ASCII** (American Standard Code for Information Interchange) es una **norma**, o **estándar**, para representación de caracteres con *siete bits* que data de los años 1960 (ISO/IEC 646) y que hoy está universalmente adoptada. Las treinta y dos primeras codificaciones y la última no representan grafemas sino acciones de control. Por tanto, quedan $2^7 - 33 = 95$ configuraciones binarias con las que se codifican los dígitos decimales, las letras mayúsculas y minúsculas y algunos caracteres comunes (espacio en blanco, «.», «+», «–», «<», etc.), pero no letras con tilde, ni otros símbolos, como «¿», «¡», «ñ», «ç», «€», etc.

| Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 032 | | 30 | 048 | 0 | 40 | 064 | @ | 50 | 080 | P | 60 | 096 | ‘ | 70 | 112 | p |
| 21 | 033 | ! | 31 | 049 | 1 | 41 | 065 | A | 51 | 081 | Q | 61 | 097 | a | 71 | 113 | q |
| 22 | 034 | " | 32 | 050 | 2 | 42 | 066 | B | 52 | 082 | R | 62 | 098 | b | 72 | 114 | r |
| 23 | 035 | # | 33 | 051 | 3 | 43 | 067 | C | 53 | 083 | S | 63 | 099 | c | 73 | 115 | s |
| 24 | 036 | $ | 34 | 052 | 4 | 44 | 068 | D | 54 | 084 | T | 64 | 100 | d | 74 | 116 | t |
| 25 | 037 | % | 35 | 053 | 5 | 45 | 069 | E | 55 | 085 | U | 65 | 101 | e | 75 | 117 | u |
| 26 | 038 | & | 36 | 054 | 6 | 46 | 070 | F | 56 | 086 | V | 66 | 102 | f | 76 | 118 | v |
| 27 | 039 | ’ | 37 | 055 | 7 | 47 | 071 | G | 57 | 087 | W | 67 | 103 | g | 77 | 119 | w |
| 28 | 040 | ( | 38 | 056 | 8 | 48 | 072 | H | 58 | 088 | X | 68 | 104 | h | 78 | 120 | x |
| 29 | 041 | ) | 39 | 057 | 9 | 49 | 073 | I | 59 | 089 | Y | 69 | 105 | i | 79 | 121 | y |
| 2A | 042 | * | 3A | 058 | : | 4A | 074 | J | 5A | 090 | Z | 6A | 106 | j | 7A | 122 | z |
| 2B | 043 | + | 3B | 059 | ; | 4B | 075 | K | 5B | 091 | [ | 6B | 107 | k | 7B | 123 | { |
| 2C | 044 | , | 3C | 060 | < | 4C | 076 | L | 5C | 092 | \ | 6C | 108 | l | 7C | 124 | \| |
| 2D | 045 | - | 3D | 061 | = | 4D | 077 | M | 5D | 093 | ] | 6D | 109 | m | 7D | 125 | } |
| 2E | 046 | . | 3E | 062 | > | 4E | 078 | N | 5E | 094 | ^ | 6E | 110 | n | 7E | 126 | ~ |
| 2F | 047 | / | 3F | 063 | ? | 4F | 079 | O | 5F | 095 | _ | 6F | 111 | o | 7F | 127 | DEL |

**Tabla 3.2**: Código ASCII.

La tabla 3.2 muestra las codificaciones ASCII a partir de 0x20 expresadas en hexadecimal y en decimal. El decimal es útil por la forma que hay en algunos sistemas para introducir caracteres que no están en el teclado: Alt+<valor decimal>.

Algunos de los caracteres de control son:

0x00, NUL: carácter nulo (fin de cadena)

0x08, BS (backspace): retroceso

0x0A, LF (line feed): nueva línea

0x0D, CR (carriage return): retorno

0x1B, ESC (escape)

0x7F, DEL (delete): borrar

## Extensiones

ASCII es «el estándar» de 7 bits, pero no es el único. Otro, también de 7 bits pero con cambios en los grafemas representados (introduce algunas letras con tilde), es el GSM 03.38, estándar para SMS en telefonía móvil.

La mayoría de las extensiones se han hecho añadiendo un bit (es decir, utilizando un byte para cada carácter), lo que permite ciento veintiocho nuevas configuraciones, conservando las codificaciones ASCII cuando este bit es 0. Se han desarrollado literalmente miles de códigos. Si no se lo cree usted,

pruebe el programa `iconv`, que traduce un texto de un código a otro. Escribiendo «`iconv -l`» se obtiene una lista de todos los códigos que conoce.

Un contemporáneo de ASCII, de ocho bits pero no compatible, que aún se utiliza en algunos grandes ordenadores *(mainframes)* es el EBCDIC (Extended Binary Coded Decimal Interchange Code).

Las extensiones a ocho bits más utilizadas son las definidas por las normas ISO 8859.

## 3.3. Códigos ISO

La ISO (International Organization for Standardization) y la IEC (International Electrotechnical Commission) han desarrollado unas normas llamadas «ISO 8859-X», donde «X» es un número comprendido entre 1 y 16 para adaptar el juego de caracteres extendido a diferentes necesidades:

- ISO 8859-1 (o «Latin-1»), para Europa occidental
- ISO 8859-2 (o «Latin-2»), para Europa central
- ISO 8859-3 (o «Latin-3»), para Europa del sur (Turquía, Malta y Esperanto)
- ...
- ISO 8859-15 (o «Latin-9»), revisión de 8859-1 en la que se sustituyen ocho símbolos poco usados por € y por algunos caracteres de francés, finés y estonio: Ž, Š, š, ž, Œ, œ, Ÿ
- ISO 8859-16 (o «Latin-10»), para Europa del sudeste (Albania, Croacia, etc.)

Todas ellas comparten las 128 primeras codificaciones con ASCII. La tabla 3.3 resume la norma ISO Latin-9. Las treinta y dos primeras codificaciones por encima de ASCII (0x80 a 0x8F) son también caracteres de control dependientes de la aplicación, y no se definen en la norma. «NBSP» (0xA0) significa «non-breaking space» (espacio inseparable) y su interpretación depende de la aplicación. En HTML, por ejemplo, se utilizan varios «` `» (o «` `», o «` `») seguidos cuando se quiere que el navegador presente todos los espacios sin colapsarlos en uno solo.

| Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | | Hex. | Dec. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A0 | 160 | NBSP | B0 | 176 | ° | C0 | 192 | À | D0 | 208 | Đ | E0 | 224 | à | F0 | 240 | ð |
| A1 | 161 | ¡ | B1 | 177 | ± | C1 | 193 | Á | D1 | 209 | Ñ | E1 | 225 | á | F1 | 241 | ñ |
| A2 | 162 | ¢ | B2 | 178 | ² | C2 | 194 | Â | D2 | 210 | Ò | E2 | 226 | â | F2 | 242 | ò |
| A3 | 163 | £ | B3 | 179 | ³ | C3 | 195 | Ã | D3 | 211 | Ó | E3 | 227 | ã | F3 | 243 | ó |
| A4 | 164 | € | B4 | 180 | Ž | C4 | 196 | Ä | D4 | 212 | Ô | E4 | 228 | ä | F4 | 244 | ô |
| A5 | 165 | ¥ | B5 | 181 | $\mu$ | C5 | 197 | Å | D5 | 213 | Õ | E5 | 229 | å | F5 | 245 | õ |
| A6 | 166 | Š | B6 | 182 | ¶ | C6 | 198 | Æ | D6 | 214 | Ö | E6 | 230 | æ | F6 | 246 | ö |
| A7 | 167 | § | B7 | 183 | · | C7 | 199 | Ç | D7 | 215 | × | E7 | 231 | ç | F7 | 247 | ÷ |
| A8 | 168 | š | B8 | 184 | ž | C8 | 200 | È | D8 | 216 | Ø | E8 | 232 | è | F8 | 248 | ø |
| A9 | 169 | © | B9 | 185 | ¹ | C9 | 201 | É | D9 | 217 | Ù | E9 | 233 | é | F9 | 249 | ù |
| AA | 170 | ª | BA | 186 | º | CA | 202 | Ê | DA | 218 | Ú | EA | 234 | ê | FA | 250 | ú |
| AB | 171 | « | BB | 187 | » | CB | 203 | Ë | DB | 219 | Û | EB | 235 | ë | FB | 251 | û |
| AC | 172 | ¬ | BC | 188 | Œ | CC | 204 | Ì | DC | 220 | Ü | EC | 236 | ì | FC | 252 | ü |
| AD | 173 | - | BD | 189 | œ | CD | 205 | Í | DD | 221 | Ý | ED | 237 | í | FD | 253 | ý |
| AE | 174 | ® | BE | 190 | Ÿ | CE | 206 | Î | DE | 222 | Þ | EE | 238 | î | FE | 254 | þ |
| AF | 175 | ¯ | BF | 191 | ¿ | CF | 207 | Ï | DF | 223 | ß | EF | 239 | ï | FF | 255 | ÿ |

**Tabla 3.3**: Códigos ISO 8859-15 que extienden ASCII.

Estos estándares son aún muy utilizados, pero paulatinamente están siendo sustituidos por Unicode. La ISO disolvió en 2004 los comités que se encargaban de ellos para centrarse en el desarrollo de Unicode, adoptado con el nombre oficial «ISO/IEC 10646».

## 3.4. Unicode

El Consorcio Unicode, en colaboración con la ISO, la IEC y otras organizaciones, desarrolla desde 1991 el código UCS (Universal Character Set), que actualmente contiene más de cien mil caracteres. Cada carácter está identificado por un nombre y un número entero llamado **punto de código**.

La primera versión (Unicode 1.1, 1991) definió el «BMP» (Basic Multilingual Plane). Es un código de 16 bits que permite representar $2^{16} = 65.536$ caracteres, aunque no todos los puntos de código se asignan a caracteres, para facilitar la conversión de otros códigos y para expansiones futuras. Posteriormente se fueron añadiendo «planos» para acomodar lenguajes exóticos, y la última versión (Unicode 6.2, 2012) tiene 17 planos, lo que da un total de $17 \times 2^{16} = 1.114.112$ puntos de código. De ellos, hay definidos 110.182 caracteres. Pero, salvo en países orientales, prácticamente sólo se utiliza el BMP.

Ahora bien, una cosa son los puntos de código y otra las codificaciones en binario de esos puntos. El estándar Unicode define siete formas de codificación. Algunas tienen longitud fija. Por ejemplo, «UCS-2» es la más sencilla: directamente codifica en dos bytes cada punto de código del BMP. Pero la más utilizada es «UTF-8», que es compatible con ASCII y tiene varias ventajas para el procesamiento de textos («UTF» es por «Unicode Transformation Format»; otras formas son UTF-16 y UTF-32).

### UTF-8

La tabla 3.4 resume el esquema de codificación UTF-8 para el BMP. «U+» es el prefijo que se usa en Unicode para indicar «hexadecimal»; es equivalente a «0x».

| Puntos | Punto en binario | Bytes UTF-8 |
|---|---|---|
| U+0000 a U+007F | 00000000 0xxxxxxx | 0xxxxxxx |
| U+0080 a U+07FF | 00000yyy yyxxxxxx | 110yyyyy<br>10xxxxxx |
| U+0800 a U+FFFF | zzzzyyyy yyxxxxxx | 1110zzzz<br>10yyyyyy<br>10xxxxxx |

**Tabla 3.4**: Codificación en UTF-8.

Los 128 primeros puntos de código se codifican en un byte, coincidiendo con las codificaciones ASCII. Los siguientes 1.920, entre los que están la mayoría de los caracteres codificados en las normas ISO, necesitan dos bytes. El resto de puntos del BMP se codifican en tres bytes. Los demás planos (puntos U+10000 a U+10FFFF, no mostrados en la tabla) requieren 4, 5 o 6 bytes.

Algunos ejemplos se muestran en la tabla 3.5. Observe que:

- «E» está codificado en un solo byte, como en ASCII.

| | Nombre Unicode | Punto de código | UTF-8 |
|---|---|---|---|
| E | LATIN CAPITAL LETTER E | U+0045 | 0x45 |
| ñ | LATIN SMALL LETTER N WITH TILDE | U+00F1 | 0xC3B1 |
| € | EUROSIGN | U+20AC | 0xE282AC |
| Pts | PESETA SIGN | U+20A7 | 0xE282A7 |
| $\mathcal{D}\rho$ | DRACHMA SIGN | U+20AF | 0xE282AF |
| $\Rightarrow$ | RIGHTWARDS DOUBLE ARROW | U+21D2 | 0xE28792 |
| $\infty$ | INFINITY | U+2210 | 0xE2889E |
| $\int$ | INTEGRAL | U+222B | 0xE288AB |

**Tabla 3.5**: Ejemplos de codificación en UTF-8.

- «ñ» no está en ASCII; el byte menos significativo de su punto de código coincide con la codificación en ISO Latin-9 (tabla 3.3), y, como puede usted comprobar, su codificación en UTF-8 se obtiene siguiendo la regla de la tabla 3.4.

- Sin embargo, «€», que también está en la tabla 3.3, tiene un punto de código superior a U+07FF y se codifica en tres bytes. El motivo es que los puntos U+00A0 a U+00FF son compatibles con ISO Latin-1, no con ISO Latin-9, y en el primero la codificación 0xA4 no está asignada al símbolo del euro, sino al símbolo monetario genérico, «¤».

## 3.5. Cadenas

Una «cadena» *(string)*, en informática, es una secuencia finita de símbolos tomados de un conjunto finito llamado alfabeto.

La representación de un texto plano, una vez elegido un código para la representación de los caracteres es, simplemente, la cadena de bytes que corresponden a la sucesión de caracteres del texto. A diferencia de otros tipos de datos, las cadenas tienen longitud variable. Un convenio que siguen algunos sistemas y lenguajes es indicar al principio de la cadena su longitud (un entero representado en uno o varios bytes). Pero el convenio más frecuente es utilizar una codificación de ASCII, 0x00 («NUL»), para indicar el fin de la cadena. Se les llama «cadenas C» *(C strings)* por ser el convenio del lenguaje C.

Como se trata de una sucesión de bytes, en principio no se plantea el dilema del «extremismo mayor o menor» (apartado 1.4): si el primer byte se almacena en la dirección $d$, el siguiente lo hará en $d + 1$ y así sucesivamente. Sin embargo:

- Esto es cierto para ASCII y para las normas ISO. Pero en UTF-16 el elemento básico de representación ocupa dos bytes (y en UTF-32, cuatro), y UTF-8 es de longitud variable, y el problema sí aparece. El consorcio Unicode ha propuesto un método que consiste en añadir al principio de la cadena una marca de dos bytes llamada «BOM» (Byte Order Mark) cuya lectura permite identificar si es extremista mayor o menor.

- Se suele ilustrar el conflicto del extremismo con el «problema NUXI», que tiene un origen histórico: en 1982, al transportar uno de los primeros Unix de un ordenador (PDP-11) a otro (IBM Series/1), un programa que debía escribir «UNIX» escribía «NUXI». La explicación es que para generar los cuatro caracteres se utilizaban dos palabras de 16 bits, y el problema surge al almacenar en memoria los dos bytes de cada palabra: el programa seguía el convenio del PDP-11 (extremista menor), mientras que el ordenador que lo ejecutaba era extremista mayor.

## 3.6. Texto con formato

Para los textos «enriquecidos» con propiedades de los caracteres (tamaño, tipo de letra o *«font»*, color, etc.) con el fin de representar los distintos alógrafos de los grafemas, así como para otros documentos que pueden incluir también imágenes y sonidos (presentaciones, hojas de cálculo, etc.), se han sucedido muchos estándares «*de facto*»: convenios para programas comerciales que durante algún tiempo se han impuesto en el mercado de la ofimática. Los ejemplos más recientes son los de la «suite» Microsoft Office, que incluye el programa de procesamiento de textos «Word» y su formato «.doc», el de presentaciones «PowerPoint» y su formato «.ppt» el de hojas de cálculo «Excel» y su formato «.xls», etc. Estos formatos son privativos (o «propietarios»), ligados a los programas comerciales que los usan. Recientemente, debido en parte a la presión de las administraciones europeas por el uso de formatos abiertos, se han extendido dos estándares oficiales:

- **ODF** (Open Document Format) es un estándar ISO (ISO/IEC 26300:2006) que incluye convenios para textos (ficheros «`.odt`»), presentaciones («`.odp`»), hojas de cálculo («`.ods`»), etc.

- **OOXML** (Office Open XML) es otro estándar (ISO/IEC IS 29500:2008) similar, promocionado por Microsoft. El convenio para los nombres de los ficheros es añadir la letra «x» a las extensiones de los antiguos formatos: «`.docx`», «`.pptx`», «`.xlsx`», etc.

Ambos comparten el basarse en un metalenguaje que mencionaremos luego, XML. Todo documento se representa mediante un conjunto de ficheros XML archivados y comprimidos con ZIP. Puede usted comprobarlo fácilmente: descomprima un documento cualquiera que esté en uno de estos formatos (aunque no tenga la extensión «.zip») y verá aparecer varios directorios («carpetas») y en cada directorio varios ficheros XML, ficheros de texto plano que puede usted leer (con `cat`, o con `more`, o con `less`...) y editar con cualquier editor de textos.

**PDF** *(Portable Document Format)* es otro formato muy conocido para distribución de documentos. En su origen también privativo, es actualmente un estándar abierto (ISO 32000-1:2008), y la empresa propietaria de las patentes (Adobe) permite el uso gratuito de las mismas, por lo que hay muchas implementaciones tanto comerciales como libres. Es un formato complejo, muy rico en tipografías y elementos gráficos matriciales y vectoriales.

## 3.7. Hipertexto

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».

Se trata de un concepto que actualmente es bien conocido por ser el fundamento de la web. Para su implementación se utiliza el lenguaje HTML (HyperText Markup Language). A los efectos de almacenamiento y de transmisión, un documento HTML (que se presenta como una página web) es texto plano. El intérprete del lenguaje incluido en el navegador reconoce las marcas y genera la presentación adecuada.

## 3.8. Documentos 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, como ODF y OOXML, o como SVG, un lenguaje cuyo objetivo es la representación de gráficos (apartado 5.3).

XML también está basado en marcas para definir las reglas sintácticas del lenguaje concreto y para construir aplicaciones con ese lenguaje. Y, al igual que HTML, un documento XML es un fichero de texto plano. En principio, la codificación es UTF-8, pero también admite otras codificaciones, señalándolo al principio del documento para hacérselo saber a la aplicación que va a procesarlo. Así, la primera línea de un documento XML podría ser:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Si se omite la información de «encoding» se entiende que es UTF-8.

La descripción de XML como lenguaje y la forma de definir lenguajes específicos sería muy interesante, pero desborda los objetivos de este capítulo, que se limita a los convenios de representación de textos, sin entrar en su contenido.

# Capítulo 4

# Representación de datos numéricos

Los datos de tipo numérico pueden codificarse en binario siguiendo distintos métodos. Los más usuales están basados en uno de dos principios: o bien codificar cada dígito decimal mediante un código BCD (de modo que un número de $n$ dígitos decimales se codifique con $n$ cuartetos, apartado 3.1), o bien representar el número en el sistema de numeración de base 2. En cualquier caso, hay que añadir convenios para representar números negativos y números no enteros.

La codificación en BCD actualmente sólo se usa en dispositivos especializados (relojes, calculadoras, etc.).

Por otra parte, se pueden representar números con *precisión arbitraria*, utilizando cuantos bits sean necesarios (con la limitación de la capacidad de memoria o del caudal del canal de transmisión disponibles, por eso no es «precisión infinita»). Hay lenguajes y bibliotecas de programas que permiten representar de este modo y realizar operaciones aritméticas *(«bignum arithmetic»)* para aplicaciones especiales, pero tampoco nos ocuparemos de este tipo de representación.

Nos vamos a centrar en los convenios normalmente adoptados en los diseños de los procesadores de uso general. Se basan en reservar un número fijo de bits (normalmente, una palabra, o un número reducido de palabras) para representar números enteros y racionales con una precisión y un rango (números máximo y mínimo) determinados por el número de bits. La figura 4.1 generaliza a $n$ bits otra que ya habíamos visto (figura 1.1). Como ya dijimos, el bit 0 (o «bit de peso cero») es el *bit menos significativo, «bms»* (o «LSB»: Least Significant Bit) y el de peso $n-1$ es el *bit más significativo, «bMs»* (o «MSB»: Most Significant Bit).

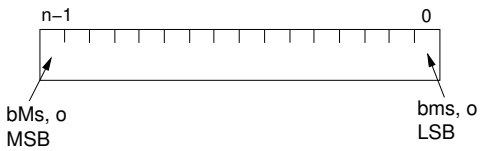



**Figura 4.1**: «Contenedor» de $n$ bits.

## 4.1. Números enteros sin signo

Algunos datos son de tipo entero no negativo, como las direcciones de la memoria, o el número de segundos transcurridos a partir de un instante dado. Su representación es, simplemente, su expresión en binario. Con $n$ bits el máximo número representable es $0b111\ldots1 = 2^n - 1$ («0b» indica que lo que sigue está en el sistema de numeración de base 2). El *rango* de la representación es $[0, 2^n - 1]$.

Por ejemplo, para expresar las direcciones de una memoria de 1 MiB = $2^{20}$ bytes necesitamos $n = 20$ bits. La primera dirección será 0x00000 y la más alta 0xFFFFF = $2^{20} - 1$. Otro ejemplo: las direcciones IPv4 (Internet Protocol versión 4) son enteros sin signo de 32 bits. La más pequeña es 0x00000000 (0.0.0.0) y la más grande 0xFFFFFFFF (255.255.255.255). En total, $2^{32}$ = 4.294.967.296 direcciones.

## 4.2. Formatos de coma fija

En el contexto de la representación de números, un **formato** es la definición del significado de cada uno de los $n$ bits dedicados a la representación. Se distinguen en él partes o **campos**, grupos de bits con significado propio. Los formatos más utilizados para la representación de datos numéricos son los de **coma fija** y **coma flotante** (o «punto fijo» y «punto flotante»).

Los formatos de coma fija sólo tienen dos campos. Uno es el bit más significativo o **bit de signo**, bMs = S: si S = 0, el número representado es positivo, y si S = 1, es negativo. El campo formado por los $n - 1$ bits restantes representa la magnitud del número. Se siguen distintos convenios sobre cómo se codifica esta magnitud.

## 4.3. Números enteros

Para un número entero no tiene sentido hablar de «coma». En todo caso, esta «coma» estaría situada inmediatamente a la derecha del bms.

Como el primer bit es el de signo, el máximo número representable (positivo) es $0b0111\ldots11 = 2^{n-1} - 1$. El mínimo depende del convenio para los números negativos.

### Convenios para números negativos

Hay tres convenios:

- **signo y magnitud** (o **signo y módulo**): después del signo (S = 1), el valor absoluto;
- **complemento a 1**: se cambian los «ceros» por «unos» y los «unos» por «ceros» en la representación binaria del correspondiente número positivo;
- **complemento a 2**: se suma una unidad al complemento a 1.

Por ejemplo, el número –58 = –0x3A = –0b111010, con $n = 16$ bits, y según el convenio que se adopte, da lugar a las siguientes representaciones:

- con signo y módulo: 1000000000111010 (0x803A)
- con complemento a 1: 1111111111000101 (0xFFC5)
- con complemento a 2: 1111111111000110 (0xFFC6)

Es fácil comprobar que si $X$ es la representación de un número con $n$ bits, incluido el de signo, y si $X1$ y $X2$ son las representaciones en complemento a 1 y complemento a 2, respectivamente, del mismo número cambiado de signo, entonces $X + X1 = 2^n - 1$, y $X + X2 = 2^n$. (Por tanto, en rigor, lo que habitualmente llamamos «complemento a 2» es «complemento a $2^n$», y «complemento a 1» es «complemento a $2^n - 1$»).

La tabla 4.1 muestra una comparación de los tres convenios para $n$ bits. Como puede usted observar, en signo y magnitud y en complemento a 1 existen dos representaciones para el número 0, mientras que en complemento a 2 la representación es única (y hay un número negativo más, cuyo valor absoluto no puede representarse con $n$ bits). El *rango*, o *extensión* de la representación (diferencia entre el número máximo y el mínimo representables) es una función exponencial de $n$.

En lo sucesivo, y salvo en alguna nota marginal, *supondremos siempre que el convenio es el de complemento a 2*, que es el más común.

| | configuraciones binarias | interpretaciones decimales | | |
|---|---|---|---|---|
| | | signo y magnitud | complemento a 1 | complemento a 2 |
| números positivos | 0000…0000 | 0 | 0 | 0 |
| | 0000…0001 | 1 | 1 | 1 |
| | … | … | … | … |
| | 0111…1110 | $2^{n-1}-2$ | $2^{n-1}-2$ | $2^{n-1}-2$ |
| | 0111…1111 | $2^{n-1}-1$ | $2^{n-1}-1$ | $2^{n-1}-1$ |
| números negativos | 1000…0000 | 0 | $-(2^{n-1}-1)$ | $-(2^{n-1})$ |
| | 1000…0001 | $-1$ | $-(2^{n-1}-2)$ | $-(2^{n-1}-1)$ |
| | … | … | … | … |
| | 1111…1110 | $-(2^{n-1}-2)$ | $-1$ | $-2$ |
| | 1111…1111 | $-(2^{n-1}-1)$ | 0 | $-1$ |

**Tabla 4.1**: Comparación de convenios para formatos de coma fija.

## 4.4. Operaciones básicas de procesamiento

En este capítulo estamos estudiando los convenios de representación de varios tipos de datos. Es posible que, a veces, alguno de esos convenios le parezca a usted arbitrario. Por ejemplo, ¿por qué representar en complemento los números negativos, cuando parece más «natural» el hacerlo con signo y módulo?

Pero tenga en cuenta que los convenios no se adoptan de forma caprichosa, sino guiada por el tipo de operaciones que se realizan sobre los datos y por los algoritmos o por los circuitos electrónicos que implementan estas operaciones en software o en hardware. Aunque nos desviemos algo del contenido estricto del tema («representación»), éste es un buen momento para detenernos en algunas operaciones básicas de *procesamiento*.

Para no perdernos con largas cadenas de bits pondremos ejemplos muy simplificados, en el sentido de un número de bits muy reducido: $n = 6$. Esto daría un rango de representación para números enteros sin signo de $[0, 63]$ y un rango para enteros de $[-32, 31]$, lo que sería insuficiente para la mayoría de las aplicaciones prácticas, pero es más que suficiente para ilustrar las ideas.

### Suma y resta de enteros sin signo

La suma binaria se realiza igual que la decimal: primero se suman los bits de peso 0, luego los de peso 1 más el eventual acarreo (que se habrá producido en el caso de 1+1=10), etc. Por ejemplo:

```
  13    001101   (0x0D)
+39   +100111  (+0x27)
  52    110100   (0x34)
```

Observe que se puede producir acarreo en todos los bits salvo en el más significativo, porque eso indica que el resultado no puede representarse en el rango, y se dice que hay un **desbordamiento**. Por ejemplo:

```
  60     111100   (0x3C)
+15    +001111  (+0x0F)
  75  1) 001011   (0x4B, pero el máximo es 0x3F: desbordamiento)
```

Para la resta también se podría aplicar un algoritmo similar al que conocemos para la base 10,

pero para el diseño de circuitos es más fácil trabajar con complementos. Como estamos tratando de enteros *sin signo* con seis bits, no tiene sentido el complemento a $2^6$, pero si sumamos al minuendo el complemento a $2^7$ del sustraendo obtenemos el resultado correcto. Por ejemplo:

$$\begin{array}{rrl} 60 & 111100 & (0x3C) \\ \underline{-15} & \underline{+1)110001} & (\underline{+0x71}) \text{ (complemento a } 2^7\text{de 15)} \\ 45 & 10)\ 101101 & (0x2D, \text{ olvidando el acarreo)} \end{array}$$

Fíjese que ahora *siempre hay un acarreo* de los bits más significativos que hay que despreciar, incluso si el resultado es 0 (puede comprobarlo fácilmente).

Parece que no tiene mucho sentido preguntarse por el caso de que el sustraendo sea mayor que el minuendo, ya que estamos tratando de números positivos, pero a veces lo que interesa es *comparar* dos números: saber si $X$ es mayor, igual o menor que $Y$, y esto se puede averiguar sumando a $X$ el complemento de $Y$. Por ejemplo:

$$\begin{array}{rrl} 15 & 001111 & (0x0F) \\ \underline{-60} & \underline{+1)000100} & (+0x44) \text{ (complemento a } 2^7\text{de 60)} \\ & 010011 & \end{array}$$

Lo importante ahora no es el resultado, obviamente incorrecto, sino el hecho de que *no hay acarreo de los bits más significativos*.

Conclusión: para comparar $X$ e $Y$, representados sin signo en $n$ bits, sumar a $X$ el complemento a $2^{n+1}$ de $Y$. Si se produce acarreo de los bits más significativos, $X \geq Y$; en caso contrario, $X < Y$.

## Suma y resta de enteros

La representación de los números negativos por su complemento conduce a algoritmos más fáciles para la suma y resta que en el caso de signo y módulo (y, consecuentemente, a circuitos electrónicos más sencillos para su implementación).

Como hemos visto antes, si $X$ es un entero positivo y $X2$ es su complemento a 2, $X + X2 = 2^n$.

Por ejemplo:
$$\begin{array}{rrl} 7 & 000111 & \\ \underline{+(-7)} & \underline{+111001} & \\ 0 & 1000000 & (2^6) \end{array}$$

Veamos cómo se aplica esta propiedad a la resta. Sean $X$ e $Y$ dos números positivos. La diferencia $D = X - Y$ se puede escribir así:
$D = X - Y = X - (2^n - Y2) = X + Y2 - 2^n$
donde $Y2$ es el complemento a 2 de $Y$.

Analicemos dos posibilidades: o bien $X + Y2 \geq 2^n$ ($D \geq 0$), o bien $X + Y2 < 2^n$ ($D < 0$).

- Si $D \geq 0$, restar $2^n$ de $X + Y2$ es lo mismo que quitar el bit de peso $n$ (acarreo de los bits de signo).
- Si $D < 0$ la representación de $D$ en complemento a 2 será $D2 = 2^n - (-D) = X + Y2$.

O sea, la suma binaria $X+Y2$, olvidando el posible acarreo de los bits de signo, nos va a dar siempre el resultado correcto de $X - Y$.

¿Y si el minuendo es negativo? La operación $L = -X - Y$ (siendo $X$ e $Y$ positivos) se escribe así en función de los complementos:

$L = -X - Y = -(2^n - X2) - (2^n - Y2) = X2 + Y2 - 2^{n+1}$

El resultado ha de ser siempre negativo (a menos, como veremos luego, que haya desbordamiento), es decir, $X2 + Y2 < 2^{n+1}$. La representación en complemento a 2 de este resultado es:

$L2 = 2^n - (-L) = X2 + Y2 - 2^n$.

En conclusión, *para restar dos números basta sumar al minuendo el complemento a 2 del sustraendo y no tener en cuenta el eventual acarreo de los bits de signo*[1]. Ejemplos:

- $\begin{array}{rrl} 25 & 011001 & \\ \underline{-7} & \underline{+111001} & \\ 18 & 1010010 & \rightsquigarrow \quad 010010 \text{ (representación de 18)} \end{array}$
- $\begin{array}{rrl} 14 & 001110 & \\ \underline{-30} & \underline{+100010} & \\ -16 & 110000 & \quad (-16 \text{ en complemento a 2}) \end{array}$
- $\begin{array}{rrl} -10 & 110110 & \\ \underline{-18} & \underline{+101110} & \\ -28 & 1100100 & \rightsquigarrow \quad 100100 \ (-28 \text{ en complemento a 2}) \end{array}$

## Desbordamiento

En todos los ejemplos anteriores (salvo en uno) hemos supuesto que tanto los operandos como el resultado estaban dentro del rango de números representables, que en el formato de los ejemplos es $-32 \leq X \leq 31$. Si no es así, las representaciones resultantes son erróneas. Por ejemplo:

- $\begin{array}{rrl} 15 & 001111 & \\ \underline{+20} & \underline{+010100} & \\ 35 & 100011 & \quad (-29 \text{ en complemento a 2}) \end{array}$
- $\begin{array}{rrl} -13 & 110011 & \\ \underline{-27} & \underline{+100101} & \\ -40 & 1011000 & \rightsquigarrow \quad 011000 \ (+24) \end{array}$

Este **desbordamiento** sólo se produce cuando los dos operandos son del mismo signo (se supone, por supuesto, que ambos están dentro del rango representable). Y su detección es muy fácil: cuando lo hay, el resultado es de signo distinto al de los operandos[2].

## Acarreo

A veces el bit de signo no es un bit de signo, ni el desbordamiento indica desbordamiento. Veamos cuáles son esas «veces».

Supongamos que tenemos que sumar 1075 (0x433) y 933 (0x3A5). El resultado debe ser 2008 (0x7D8). Para hacerlo en binario necesitamos una extensión de al menos doce bits:

$$\begin{array}{rcl} 0\text{x}433 & \rightsquigarrow & 010000110011 \\ 0\text{x}3\text{A}5 & \rightsquigarrow & \underline{001110100101} \\ & & 011111011000 \quad \rightsquigarrow \quad 0\text{x}7\text{D}8 \end{array}$$

Pero supongamos también que nuestro procesador solamente permite sumar con una extensión de seis bits. Podemos resolver el problema en tres pasos:

[1] Si se entretiene usted en hacer un análisis similar para el caso de complemento a 1, descubrirá que se hace necesario un paso más: el eventual acarreo de los bits de signo no se debe despreciar, sino sumar a los bits de peso 0 para que el resultado sea correcto.

[2] Una condición equivalente (y más fácil de comprobar en los circuitos, aunque su enunciado sea más largo) es que el acarreo de la suma de los bits de pesos inmediatamente inferior al de signo es diferente al acarreo de los bits de signo (el que se descarta).

1. Sumar los seis bits menos significativos como si fuesen números sin signo:
   $$\begin{array}{r} 110011 \\ +100101 \\ \hline 1\,011000 \end{array}$$
   (los seis bits menos significativos del resultado)
   Obtenemos un resultado en seis bits y un acarreo de los bits más significativos. Observe que esta suma es justamente la que provocaba desbordamiento en el ejemplo anterior de $-13-27$, pero ahora no estamos interpretando esos seis bits como la representación de un número entero con signo: los bits más significativos no son «bits de signo».
2. Sumar los seis bits más significativos:
   $$\begin{array}{r} 010000 \\ +001110 \\ \hline 011110 \end{array}$$
3. Al último resultado, sumarle el acarreo de los seis menos significativos:
   $$\begin{array}{r} 011110 \\ +1 \\ \hline 011111 \end{array}$$
   (los seis bits más significativos del resultado)

El ejemplo ha tratado de la suma de dos números positivos, pero el principio funciona exactamente igual con números negativos representados en complemento.

Veremos en el capítulo 9 que los procesadores suelen tener dos instrucciones de suma: suma «normal» (que suma operandos de, por ejemplo, 32 bits) y suma con acarreo, que suma también dos operandos, pero añadiendo el eventual acarreo de la suma anterior.

## Indicadores

Los circuitos de los procesadores (unidades aritmética y lógica y de desplazamiento) incluyen indicadores de un bit (materializados como biestables, apartado 1.3) que toman el valor 0 o 1 dependiendo del resultado de la operación. Normalmente son cuatro:

**C:** acarreo. Toma el valor 1 si, como acabamos de ver, se produce acarreo en la suma sin signo. Y, como vimos antes, en la resta sin signo $X - Y$ también se pone a 1 si $X \geq Y$.

**V:** desbordamiento. Toma el valor 1 si se produce un desbordamiento en la suma o en la resta con signo.

**N:** negativo. Coincide con el bit más significativo del resultado (S). Si no se ha producido desbordamiento, N = 1 indica que el resultado es negativo.

**Z:** cero. Toma el valor 1 si todos los bits del resultado son 0.

N y V, conjuntamente, permiten detectar el mayor de dos enteros *con signo* al hacer la resta $X - Y$:

- $X \geq Y$ si N = V, o sea, si N = 0 y V = 0 o si N = 1 y V = 1
- $X < Y$ si N $\neq$ V, o sea, si N = 0 y V = 1 o si N = 1 y V = 0

## Operaciones lógicas

Las principales operaciones lógicas son la *complementación* o *negación* (**NOT**), el *producto lógico* (**AND**), la *suma lógica* (**OR**), la *suma módulo 2* u *or excluyente* (**EOR**), el *producto negado* (**NAND**) y la *suma negada* (**NOR**). Se trata de las mismas operaciones del álgebra de Boole, aplicadas en paralelo a un conjunto de bits. Aunque usted ya las debe conocer por la asignatura «Sistemas electrónicos», las resumimos brevemente.

La primera se aplica a un solo operando y consiste, simplemente, en cambiar todos sus «ceros» por «unos» y viceversa. Si el operando se interpreta como la representación de un número entero, el resultado es su complemento a 1. Por ejemplo: NOT(01101011) = 10010100

Las otras cinco se aplican, bit a bit, sobre dos operandos, y se definen como indica la tabla 4.2, donde en las dos primeras columnas se encuentran los cuatro valores posibles de dos operandos de un bit, y en las restantes se indican los resultados correspondientes a cada operación.

| op1 | op2 | AND | OR | EOR | NAND | NOR |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

**Tabla 4.2**: Operaciones lógicas sobre dos operandos.

El **enmascaramiento** consiste en realizar la operación *AND* entre un dato binario y un conjunto de bits prefijado (**máscara**) para poner a cero determinados bits (y dejar los demás con el valor que tengan). Así, para extraer los cuatro bits menos significativos de un byte utilizaremos la máscara 0x0F = 00001111, y si la información original es «abcdefgh» (a, b,... ,h $\in \{0,1\}$) el resultado será «0000efgh».

## Operaciones de desplazamiento

Los desplazamientos, lo mismo que las operaciones lógicas, se realizan sobre un conjunto de bits, normalmente, una palabra. Hay dos tipos básicos: *desplazamientos a la derecha*, en los que el bit *i* del operando pasa a ocupar la posición del bit $i-1$, éste la del $i-2$, etc., y *desplazamientos a la izquierda*, en los que se procede al revés. Y hay varios subtipos (figura 4.2), dependiendo de lo que se haga con el bit que está más a la derecha (bms) y con el que está más a la izquierda (bMs):

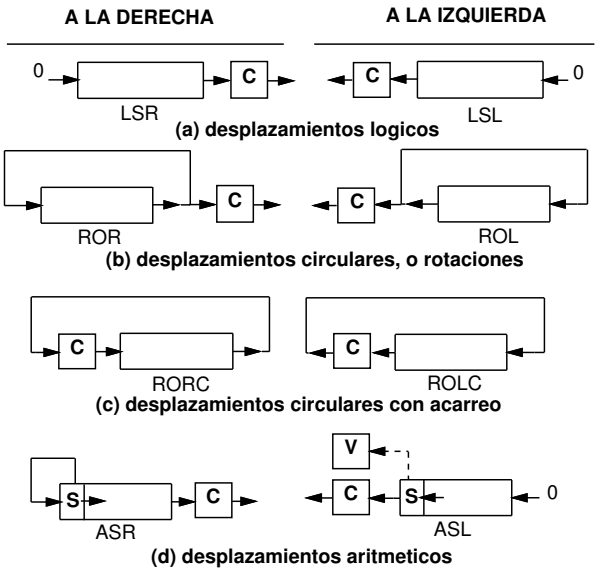



**Figura 4.2**: Operaciones de desplazamiento.

**Desplazamientos lógicos**

En un desplazamiento lógico a la derecha el bms se introduce en el indicador C (perdiéndose el valor anterior que éste tuviese), y por la izquierda se introduce un «0» en el bMs. Justamente lo contrario se hace en el desplazamiento lógico a la izquierda: el bMs se lleva a C y se introduce un «0» por la derecha, como ilustra la figura 4.2(a).

Las siglas que aparecen en la figura son nombres nemónicos en inglés: `LSR` es «Logical Shift Right» y `LSL` es «Logical Shift Left».

**Desplazamientos circulares**

En los desplazamientos circulares, también llamados **rotaciones**, el bit que sale por un extremo se introduce por el otro, como indica la figura 4.2(b). Los nemónicos son `ROR` («Rotate Right») y `ROL` («Rotate Left»).

Las rotaciones a través del indicador C funcionan como muestra la figura 4.2(c).

**Desplazamientos aritméticos**

En el desplazamiento aritmético a la izquierda, el bMs se introduce en el indicador de acarreo (C), y en el desplazamiento aritmético a la derecha se propaga el bit de signo. La figura 4.2(d) ilustra este tipo de desplazamiento, muy útil para los algoritmos de multiplicación. Es fácil comprobar que un desplazamiento aritmético de un bit a la derecha de un entero con signo conduce a la representación de la división (entera) por 2 de ese número. Y que el desplazamiento a la izquierda hace una multiplicación por 2.

Los nemónicos son `ASR` («Arithmetic Shift Right») y `ASL` («Arithmetic Shift Left»).

Aparentemente, el desplazamiento aritmético a la izquierda tiene el mismo efecto que el desplazamiento lógico a la izquierda, pero hay una diferencia sutil: el primero pone un «1» en el indicador de desbordamiento (V) si como consecuencia del desplazamiento cambia el bit de signo (S).

## 4.5. Números racionales

Volvamos al asunto principal de este capítulo, que es la representación de datos numéricos.

Para la representación de números racionales con una cantidad finita de bits hemos de tener en cuenta que no sólo habrá una **extensión** finita (un número máximo y un número mínimo), sino también una **resolución** (diferencia entre dos representaciones consecutivas) finita, es decir, hay una *discretización* del espacio continuo de los números reales.

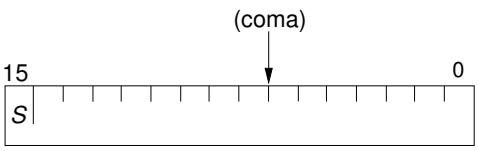



**Figura 4.3**: Formato de 16 bits para representación en coma fija de números racionales.

En principio, con un formato de coma fija podemos representar también números racionales: basta con añadir el convenio sobre «dónde está la coma». Si, por ejemplo, tenemos una longitud de dieciséis bits, podríamos convenir que los ocho que siguen al bit de signo representan la parte entera y los siete restantes la parte fraccionaria (figura 4.3).

En este ejemplo, el máximo número representable es 0b0 1111 1111,1111 111 = +0xFF,FE = +255,9921875 y el mínimo, suponiendo complemento a 2, es el representado por 1000000000000000, que corresponde a –0b1 0000 0000,0000 000 = –0x100 = –256. La resolución es: 0b0,0000 001 = 0x0,02 = 0,0078125.

En general, si tenemos $f$ bits para la parte fraccionaria, estamos aplicando un factor de escala $1/2^f$ con respecto al número entero que resultaría de considerar la coma a la derecha. El máximo

representable es $(2^{n-1} - 1)/2^f$, y el mínimo, $-2^{n-1}/2^f$. La diferencia máx − mín $= 2^{-f}$ es igual a la resolución.

Pero la rigidez que resulta es evidente: para conseguir la máxima extensión, en cada caso tendríamos que decidir dónde se encuentra la coma, en función de que estemos trabajando con números muy grandes o muy pequeños. Aun así, la extensión que se consigue puede ser insuficiente: con 32 bits, el número máximo representable (todos «unos» y la coma a la derecha del todo) sería $2^{31} - 1 \approx 2 \times 10^9$, y el mínimo positivo (coma a la izquierda y todos «ceros» salvo el bms) sería $2^{-31} \approx 2 \times 10^{-10}$

Por esto, lo más común es utilizar un formato de coma flotante, mucho más flexible y con mayor extensión.

## Formatos de coma flotante

El principio de la representación en coma flotante de un número $X$ consiste en codificar dos cantidades, $A$ y $E$, de modo que el número representado sea:

$$X = \pm A \times b^E$$

Normalmente, $b = 2$. Sólo en algunos ordenadores se ha seguido el convenio de que sea $b = 16$.

En un formato de coma flotante hay tres campos (figura 4.4):

- $S$: *signo* del número, 1 bit.
- $M$: *mantisa*, $m$ bits.
- $C$: *característica*, $e$ bits.

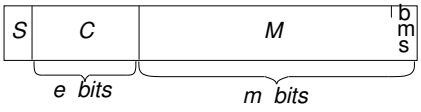



**Figura 4.4**: Formato de coma flotante.

Se pueden seguir varios convenios para determinar cómo se obtienen los valores de $A$ y $E$ en función de los contenidos binarios de $M$ y $C$

## Mantisa

Los $m$ bits de la mantisa representan un número en coma fija, $A$. Hace falta un convenio sobre la parte entera y la fraccionaria (o sea «dónde está la coma»). Normalmente la representación está «normalizada». Esto quiere decir que el exponente se ajusta de modo que la coma quede a la derecha del bit menos significativo de $M$ (normalización entera) o a la izquierda del más significativo (normalización fraccionaria). Es decir:

- Normalización entera: $X = \pm M \times 2^E$
- Normalización fraccionaria: $X = \pm 0, M \times 2^E$, o bien:
  $X = \pm 1, M \times 2^E$

Normalizar es trivial: por cada bit que se desplaza la coma a la izquierda o a la derecha se le suma o se le resta una unidad, respectivamente, a $E$.

Veamos con un ejemplo por qué conviene que la representación esté normalizada. Para no perdernos con cadenas de bits, supongamos por esta vez, para este ejemplo, que la base es $b = 16$. Supongamos también que el formato reserva $m = 24$ bits para la mantisa (24/4 = 6 dígitos hexadecimales), y que se trata de representar el número $\pi$ :

$X = \pi = 3{,}1415926\ldots = 0\text{x}3{,}243F69A\ldots$

Consideremos primero el caso de normalización entera; tendríamos las siguientes posibilidades:

| $M$ | $E$ | Número representado |
|---|---|---|
| 0x000003 | 0 | $0x3\times16^{0} = 0x3 = 3$ |
| 0x000032 | -1 | $0x32\times16^{-1} = 0x3{,}2 = 3{,}125$ |
| 0x000324 | -2 | $0x324\times16^{-2} = 0x3{,}24 = 3{,}140625$ |
| 0x003243 | -3 | $0x3243\times16^{-3} = 0x3{,}243 = 3{,}1413574$ |
| 0x03243F | -4 | $0x3243F\times16^{-4} = 0x3{,}243F = 3{,}1415863$ |
| 0x3243F6 | -5 | $0x3243F6\times16^{-5} = 0x3{,}243F6 = 3{,}141592$ |

La última es la representación normalizada; es la que tiene más dígitos significativos, y, por tanto, mayor precisión.

Con normalización fraccionaria y el convenio $X = \pm 0, M \times 16^{E}$:

| $M$ | $E$ | Número representado |
|---|---|---|
| 0x000003 | 6 | $0x0{,}000003\times16^{6} = 0x3 = 3$ |
| 0x000032 | 5 | $0x0{,}000032\times16^{5} = 0x3{,}2 = 3{,}125$ |
| ... | ... | ... |
| 0x3243F6 | 1 | $0x0{,}3243F6\times16^{1} = 0x3{,}243F6 = 3{,}141592$ |

y, como antes, la última es la representación normalizada.

El número «0» no se puede normalizar. Se conviene en representarlo por todos los bits del formato nulos.

Para los números negativos se pueden seguir también los convenios de signo y magnitud, complemento a 1 o complemento a 2 de la mantisa.

## Exponente

En los $e$ bits reservados para el exponente podría seguirse también el convenio de reservar uno para el signo y los $e-1$ restantes para el módulo, o el complemento a 1 o a 2 si $E < 0$. Pero el convenio más utilizado no es éste, sino el de la representación *en exceso de* $2^{e-1}$: el número *sin signo* contenido en esos $e$ bits, llamado **característica**, es:

$$C = E + 2^{e-1}$$

Dado que $0 \leq C \leq 2^{e} - 1$, los valores representables de $E$ estarán comprendidos entre $-2^{e-1}$ (para $C = 0$) y $2^{e-1} - 1$ (para $C = 2^{e} - 1$).

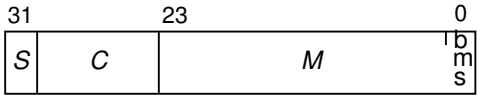



**Figura 4.5**: Un formato de coma flotante con 32 bits.

Como ejemplo, veamos la representación del número $\pi$ en el formato de la figura 4.5, suponiendo que el exponente está en exceso de 64 con base 16 y que la mantisa tiene normalización fraccionaria. De acuerdo con lo que hemos explicado antes, la mantisa será:
0x3243F6 = 0b0011001001000011111110110.
Para la característica tendremos: $C = 1 + 64 = 65 = 0x41 = 0b1000001$, y la representación resulta ser:
0 1000001 0011 0010 0100 0011 1111 0110

Observe que, al ser la base 16, la mantisa está normalizada en hexadecimal, pero no en binario, ni lo puede estar en este caso, porque cada incremento o decremento de $E$ corresponde a un desplazamiento de cuatro bits en $M$.

Si cambiamos uno de los elementos del convenio, la base, y suponemos ahora que es 2, al normalizar la mantisa resulta:
0x3,243F6A... =
0b11,001001000011111101101010... =
$0b0{,}11001001000011111101101010...\times2^{2}$

La característica es ahora $C = 2 + 64$. Truncando a 24 los bits de la mantisa obtenemos:
0 1000010 1100 1001 0000 1111 1101 1010
Vemos que al normalizar en binario ganamos dos dígitos significativos en la mantisa.

El mismo número pero negativo ($-\pi$), con convenio de complemento a 2, se representaría en el primer caso ($b = 16$) como:
1 1000001 1100 1101 1011 1100 0000 1010
y en el segundo ($b = 2$):
1 1000010 0011 0110 1111 0000 0010 0110

El inconveniente de la base 16 es que la mantisa no siempre se puede normalizar en binario, y esto tiene dos consecuencias: por una parte, como hemos visto, pueden perderse algunos bits (1, 2 o 3) con respecto a la representación con base 2; por otra, hace algo más compleja la realización de operaciones aritméticas. La ventaja es que, con el mismo número de bits reservados para el exponente, la escala de números representables es mayor. Concretamente, para $e = 7$ el factor de escala, con $b = 2$, va de $2^{-64}$ a $2^{63}$ (o, aproximadamente, de $5 \times 10^{-20}$ a $10^{19}$), mientras que con $b = 16$ va de $16^{-64}$ a $16^{63}$ (aproximadamente, de $10^{-77}$ a $10^{76}$).

## Norma IEEE 754

El estándar IEEE 754 ha sido adoptado por la mayoría de los fabricantes de hardware y de software. Define los formatos, las operaciones aritméticas y el tratamiento de las «excepciones» (desbordamiento, división por cero, etc.). Aquí sólo nos ocuparemos de los formatos, que son los esquematizados en la figura 4.6.[3]

Los convenios son:

- Números negativos: signo y magnitud.
- Mantisa: normalización fraccionaria.
- Base implícita: 2.
- Exponente: en exceso de $2^{e-1} - 1$

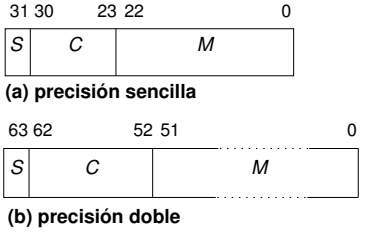



**Figura 4.6**: Formatos de la norma IEEE 754.

Como la mantisa tiene normalización fraccionaria y los números negativos se representan con signo y magnitud, el primer bit de la mantisa siempre sería 1, lo que permite prescindir de él en la representación y ganar así un bit. La coma se supone inmediatamente a la derecha de ese «1 implícito», por lo que, dada una mantisa $M$ y un exponente $E$, el número representado es $\pm(1, M) \times 2^E$.

El exponente, $E$, se representa en exceso de $2^{e-1} - 1$ (no en exceso de $2^{e-1}$). La extensión de $E$ en el formato de 32 bits, que tiene $e = 8$, resulta ser: $-127 \leq E \leq 128$, y en el de 64 bits ($e = 11$): $-1.023 \leq E \leq 1.024$. Pero los valores extremos de $C$ (todos ceros o todos unos) se reservan para representar casos especiales, como ahora veremos. Esto nos permite disponer, en definitiva, de un factor de escala que está comprendido entre $2^{-126}$ y $2^{127}$ en el formato de 32 bits y entre $2^{-1.022}$ y $2^{1.023}$ en el de 64 bits.

Resumiendo, dados unos valores de $M$ y $C$, el número representado, $X$, si se trata del formato de precisión sencilla (figura 4.6(a)), es el siguiente:

1. Si $0 < C < 255, X = \pm(1, M) \times 2^{C-127}$

[3] La norma contempla otros dos formatos, que son versiones extendidas de éstos: la mantisa ocupa una palabra completa (32 o 64 bits) y la característica tiene 10 o 14 bits. Pero los de la figura 4.6 son los más utilizados.

2. Si $C = 0$ y $M = 0, X = 0$
3. Si $C = 0$ y $M \neq 0, X = \pm 0, M \times 2^{-126}$
   (números más pequeños que el mínimo representable en forma normalizada)
4. Si $C = 255$ y $M = 0, X = \pm\infty$
   (el formato prevé una representación específica para «infinito»)
5. Si $C = 255$ y $M \neq 0, X =$ NaN
   («NaN» significa «no es un número»; aquí se representan resultados de operaciones como $0/0, 0\times\infty$, etc.)

Y en el formato de precisión doble (figura 4.6(b)):

1. Si $0 < C < 2.047, X = \pm(1, M) \times 2^{C-1.023}$
2. Si $C = 0$ y $M = 0, X = 0$
3. Si $C = 0$ y $M \neq 0, X = \pm 0, M \times 2^{-1.022}$
4. Si $C = 2.047$ y $M = 0, X = \pm\infty$
5. Si $C = 2.047$ y $M \neq 0, X =$ NaN

Una «calculadora» para pasar de un número a su representación y a la inversa se encuentra en `http://babbage.cs.qc.edu/IEEE-754/`

# Capítulo 5

# Representación de [datos] multimedia

---

El diccionario de la lengua de la Real Academia Española define el *adjetivo* «multimedia»:

> *1. adj. Que utiliza conjunta y simultáneamente diversos medios, como imágenes, sonidos y texto, en la transmisión de una información.*

El FOLDOC define «multimedia» como *sustantivo:*

> *<multimedia> Any collection of data including text, graphics, images, audio and video, or any system for processing or interacting with such data. Often also includes concepts from hypertext.*

En todo caso, es bien conocida la importancia actual de la representación de estos tipos de contenidos, que, además de textos (e hipertextos), incluyen sonidos (audio), imágenes (gráficos y fotografías) e imágenes en movimiento (animaciones y vídeos).

Salvo los textos e hipertextos, ciertas imágenes (las generadas por aplicaciones gráficas) y ciertos sonidos (los generados por sintetizadores digitales), la mayoría de las fuentes de datos multimedia son señales analógicas. Para almacenar, procesar y transmitir en formato digital una señal analógica es necesario *digitalizarla*, es decir, convertirla en una sucesión de datos numéricos representables con un número limitado de bits.

En la asignatura «Sistemas y señales» de segundo curso habrá usted estudiado los principios teóricos de este proceso de digitalización, en particular el teorema del muestreo y las conversiones de señales analógicas a digitales y viceversa. En el apartado siguiente los resumiremos, poniendo énfasis en las señales multimedia.

Por otra parte, dado el gran volumen de datos numéricos que resultan para representar digitalmente sonidos e imágenes, normalmente se almacenan y se transmiten comprimidos. El asunto de la compresión lo trataremos en el capítulo siguiente, limitándonos en éste a las representaciones numéricas sin comprimir. Pero hay que tener presente que en la práctica ambos procesos (digitalización y compresión) van unidos. Se llama **codec** (codificador/decodificador) al conjunto de convenios (frecuencia de muestreo, algoritmo de compresión, etc.) y al sistema hardware y/o software que se utiliza.

## 5.1. Digitalización

### La cadena analógico-digital-analógico

Cuando la fuente de los datos es una señal analógica y el resultado ha de ser también una señal analógica, el procesamiento y/o la transmisión digital requiere conversiones de analógico a digital y viceversa. La figura 5.1 ilustra el proceso para el caso particular del sonido. En este caso el procesamiento previo a la conversión a digital y el posterior a la conversión a analógico suelen incluir filtrado y amplificación.

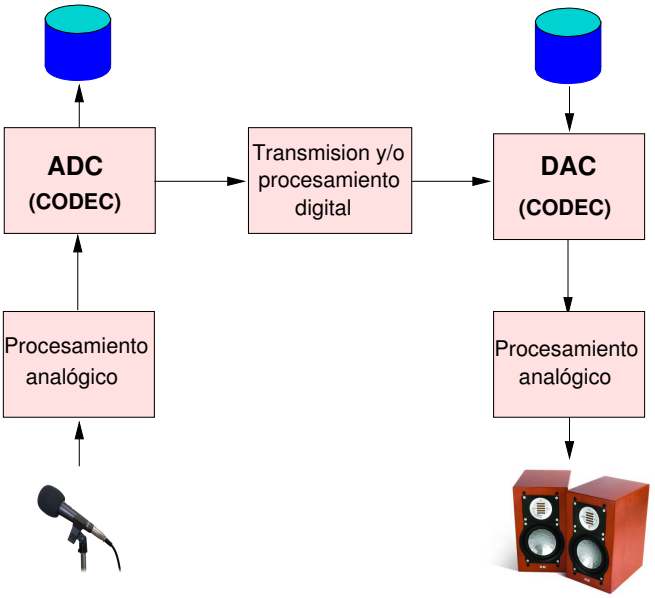



**Figura 5.1**: Procesamiento del sonido

El caso de las imágenes es similar si la fuente de tales imágenes es una cámara que genera señales analógicas. Las cámaras fotográficas y de vídeo actuales incorporan ya en sus circuitos los codecs adecuados, de manera que la imagen o el vídeo quedan grabados digitalmente en su memoria interna.

Los codecs, como ya hemos dicho, incluyen tanto los elementos que realizan las conversiones de analógico a digital (ADC: Analog to Digital Converter) y de digital a analógico (DAC: Digital to Analog Converter) como los que se ocupan de la compresión y la descompresión. Recordemos brevemente los principios de los primeros.

### Analógico a digital y digital a analógico

La conversión de una señal analógica en una señal digital se realiza en tres pasos (figura 5.2):

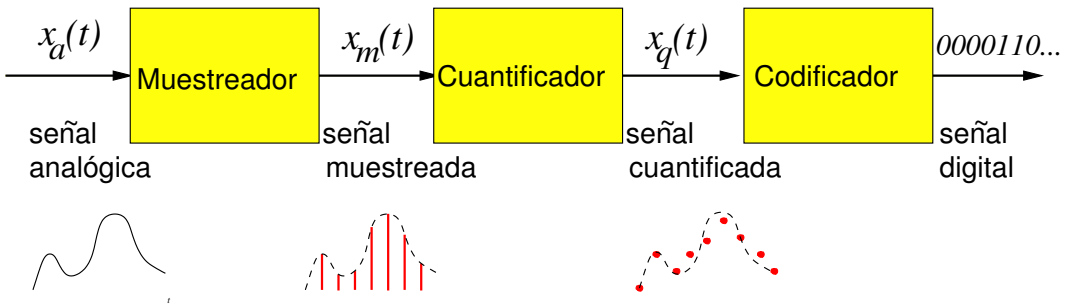
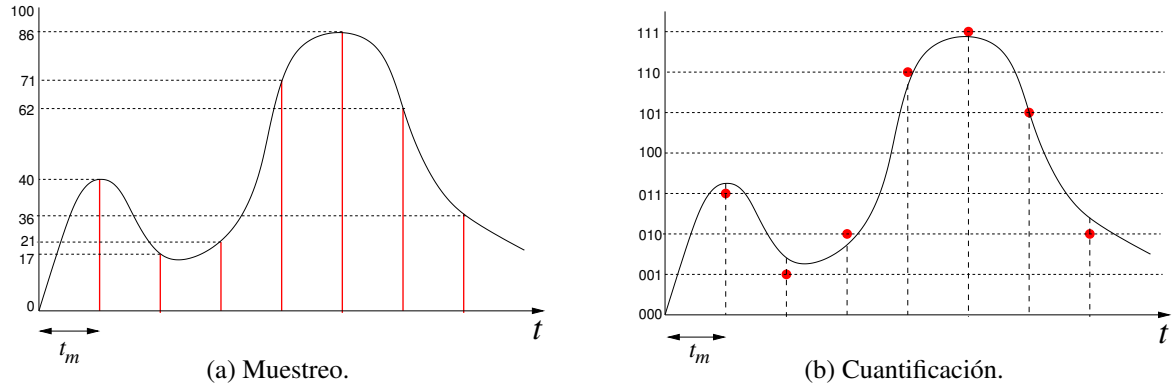



**Figura 5.2**: Conversión de analógico a digital.

1. El **muestreo** es una *discretización en el tiempo*[1]. El resultado es una sucesión temporal de **muestras**: valores de la señal original en una sucesión de instantes separados por el **período de muestreo**, $t_m$. En el ejemplo de la figura 5.3a, si $t_m = 1$, estos valores son 0 en el instante 0, 40 en el 1, 17 en el 2, 21 en el 3, etc.

2. La **cuantificación** es una *discretización de la amplitud*. Las muestras son valores reales que han de representarse con un número finito de bits, $n$. Con $n$ bits podemos representar $2^n$ valores distintos o **niveles**, y cada muestra original se representa por su nivel más cercano. El número de bits por muestra se llama **resolución**. La figura 5.3b indica el resultado de cuantificar las muestras del fragmento de señal del ejemplo con 8 niveles (resolución $n = 3$).




(a) Muestreo. (b) Cuantificación.

**Figura 5.3**: Muestreo y cuantificación.

3. En su función de **codificación**, un códec aplica (mediante software o mediante hardware) determinados algoritmos a las muestras cuantificadas y genera un flujo de bits que puede transmitirse por un canal de comunicación *(streaming)* o almacenarse en un fichero siguiendo los convenios de un formato. En el ejemplo anterior, y en el caso más sencillo (sin compresión), este flujo sería la codificación binaria de la secuencia de muestras: 000011001010...

La conversión inversa (de digital a analógico) se realiza con un descodificador acorde con el codificador utilizado y una interpolación para reconstruir la señal analógica. Lo ideal sería que esta señal reconstruida fuese idéntica a la original, pero en los tres pasos del proceso de digitalización y en la interpolación se pueden perder detalles que lo impiden, como ilustra la figura 5.4, en la que la función de interpolación es simplemente el mantenimiento del nivel desde una muestra a la siguiente *(zero-holder hold)*. Veamos cómo influye cada uno de los pasos del proceso.

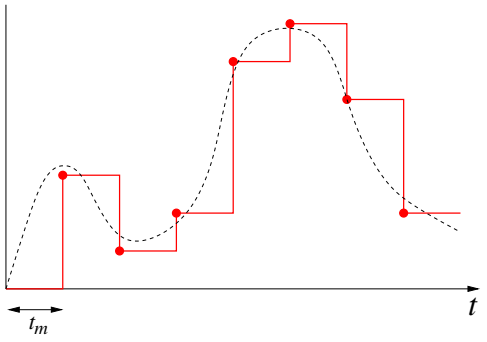



**Figura 5.4**: Reconstrucción con *zero-holder hold*.

[1] «Discretizar» una magnitud continua es convertirla en otra discreta. «Discreto» es (R.A.E.):

*«5. adj. Mat. Dicho de una magnitud: Que toma valores distintos y separados. La sucesión de los números enteros es discreta, pero la temperatura no.»*

## Muestreo

Si el período de muestreo es $t_m$, la frecuencia de muestreo es $f_m = 1/t_m$. Como debe usted saber, según el teorema del muestreo de Nyquist-Shannon [2], si la señal analógica tiene una frecuencia máxima $f_M$ basta con muestrear con $f_m \geq 2 \times f_M$ para poder reconstruir *exactamente* la señal original a partir de las muestras. A $f_m/2$ se le llama **frecuencia de Nyquist**. Si la señal original contiene componentes de frecuencia superior a la de Nyquist al reconstruirla aparece el fenómeno llamado ***aliasing***: cada uno de esos componentes genera otro espurio (un «alias») de frecuencia inferior a $f_M$ que no estaba presente en la señal original. Éste es el motivo por el que generalmente se utiliza un filtro paso bajo analógico antes del muestreo.

## Cuantificación

La figura 5.3b ilustra el caso de una cuantificación lineal: los valores cuantificados son proporcionales a las amplitudes de las muestras. Pero se utiliza más la *cuantificación logarítmica*, en la que los valores son proporcionales a los logaritmos de las amplitudes.

A diferencia del muestreo, la cuantificación introduce *siempre* una distorsión (**ruido de discretización**) tanto mayor cuanto menor es la **resolución** (número de bits por muestra). Depende de la aplicación, pero resoluciones de 8 (256 niveles) o 16 (65.536 niveles) son las más comunes.

## Codificación

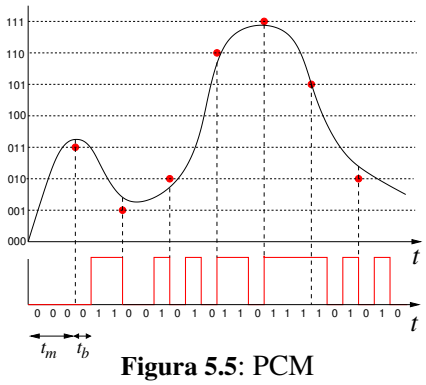



**Figura 5.5**: PCM

La forma de codificación más sencilla es **PCM** (Pulse-Code Modulation): en el intervalo de tiempo que transcurre entre la muestra $n$ y la muestra $n+1$ (período de muestreo, $t_m$) se genera un tren de impulsos que corresponde a la codificación binaria de la muestra $n$. En la figura 5.5 puede verse la sucesión de bits generados para el ejemplo anterior.

Con esta codificación el período del flujo de bits es $t_b = t_m/R$, donde $R$ es la resolución. Por tanto, el *bitrate* resultante es $f_m \times R$. En el caso de que la señal corresponda a una voz humana se puede aprovechar que hay una correlación entre muestras y codificar con DPCM (Diferencial PCM) o ADPCM (Adaptive DPCM), reduciéndose notablemente el *bitrate* a costa de una pequeña pérdida de calidad. Pero esto ya es una forma de compresión, asunto que trataremos en el capítulo siguiente.

## Interpolación

El *zero-holder hold* (figura 5.4) introduce bastante distorsión. En los DAC se suele utilizar interpolación lineal (*first-holder hold*) o polinómica. En teoría, si la frecuencia de muestreo es mayor que la de Nyquist (y obviando el ruido de discretización), se podría reconstruir perfectamente la señal original, pero la fórmula exacta es prácticamente irrealizable[3].

[2] «Theorem 1: If a function $f(t)$ contains no frequencies higher than $W$ cps, it is completely determined by giving its ordinates at a series of points spaced $1/(2W)$ seconds apart.» C. E. Shannon: Communication in the presence of noise. Proc. Institute of Radio Engineers, 37, 1 (Jan. 1949), pp. 10–21. Reproducido en Proc. IEEE, 86, 2 (Feb. 1998).

[3] En el artículo citado en la nota anterior se demuestra esta fórmula de interpolación: $f(t) = \sum_{n=-\infty}^{\infty} x_n \frac{\sin \pi(2Wt-n)}{\pi(2Wt-n)}$, donde $x_n$ es el valor de la muestra enésima.

## 5.2. Representación de sonidos

Ya sea la fuente del sonido analógica o un secuenciador digital, normalmente el sonido queda representado, a efectos de almacenamiento o transmisión digital, como una secuencia de bits. La secuencia resultante para una determinada señal analógica depende de la frecuencia de muestreo y de la resolución. Suponiendo que la codificación es PCM, el *bitrate* es:

$$f_b = f_m \times R \times C \quad \text{kbps}$$

donde $f_m$ es la frecuencia de muestreo en kHz (miles de muestras por segundo), $R$ la resolución en bits por muestra y $C$ el número de canales.

El límite superior de frecuencias humanamente audibles es aproximadamente 20 kHz, por lo que sería necesaria una frecuencia de muestreo de $f_m \geq 40$ kHz (40.000 muestras por segundo) y una resolución de 16 bits por muestra para permitir una reconstrucción prácticamente perfecta. No obstante, ciertas aplicaciones no son tan exigentes, y relajando ambos parámetros se pierde calidad pero se reduce la necesidad de ancho de banda en la transmisión y de capacidad de almacenamiento (que es el producto del *bitrate* por la duración de la señal). En la tabla 5.1 se resumen valores típicos para varias aplicaciones.

| **Aplicación** | $f_m$ **(kHz)** | $R$ **(bits)** | $C$ | $f_b$ **(kbps)** | **En 1 minuto...** |
|---|---|---|---|---|---|
| Telefonía | 8 | 8 | 1 | 64 | 480 kB (≈468 KiB) |
| Radio AM | 11 | 8 | 1 | 88 | 660 kB (≈644 KiB) |
| Radio FM | 22,05 | 16 | 2 | 705,6 | 5.292 kB (≈5 MiB) |
| CD | 44,1 | 16 | 2 | 1.411,2 | 10.584 kB (≈10 MiB) |
| TDT | 48 | 16 | 2 | 1.456 | 11.520 kB (≈11 MiB) |

**Tabla 5.1**: Tasas de bits y necesidades de almacenamiento para algunas aplicaciones de sonido.

En las aplicaciones que pretenden una reconstrucción perfecta la frecuencia de muestreo es algo superior a 40 kHz[4]. Esto se explica porque aunque la señal original se someta a un filtrado para eliminar las frecuencias superiores a 20 kHz, el filtro no es perfecto: una componente de 21 kHz aún pasaría, aunque atenuada, lo que provocaría el *aliasing*.

### Representación simbólica

La representación en un lenguaje simbólico de algunas características de los sonidos no es nada nuevo: los sistemas de notación musical se utilizan desde la antigüedad. Aquí nos referiremos a lenguajes diseñados para que las descripciones de los sonidos sean procesables por programas informáticos. Nos limitaremos a citar algunos ejemplos (si está usted interesado no le resultará difícil encontrar abundante información en Internet):

- La **notación ABC** es un estándar para expresar en texto ASCII la misma información que la notación gráfica de pentagrama.
- **MIDI** (Musical Instrument Digital Interface) es otro estándar que no solamente incluye una notación, también un protocolo e interfaces para la comunicación entre instrumentos electrónicos.
- **MusicXML** es un lenguaje basado en XML con mayor riqueza expresiva que MIDI.
- **VoiceXML** está más orientado a aplicaciones de síntesis y reconocimiento de voz.

[4]El que el estándar de audio CD determine precisamente 44,1 kHz se debe a una historia interesante. Si tiene usted curiosidad puede leerla en `http://www.cs.columbia.edu/~hgs/audio/44.1.html`.

## 5.3. Representación de imágenes

Como con el sonido, podemos distinguir entre la representación binaria de una imagen y la descripción de la misma mediante un lenguaje simbólico. En el primer caso se habla de «imágenes matriciales», y en el segundo de «gráficos vectoriales». Estos últimos tienen un interés creciente, sobre todo para las aplicaciones web, por lo que nos extenderemos en ellos algo más de lo que hemos hecho para los sonidos.

### Imágenes matriciales

Una imagen matricial *(raster image)* es una estructura de datos que representa una matriz de píxeles. Los tres parámetros importantes son el ancho y el alto en número de píxeles y el número de bits por píxel, **bpp**. A veces se le llama en general «*bitmap*», pero conviene distinguir entre:

- **bitmap** para imágenes en blanco y negro (1 bpp),
- **greymap** para imágenes en escala de grises ($n$ bpp; el número de tonos es $2^n$), y
- **pixmap** para imágenes en color ($n$ bpp es la **profundidad de color**; el número de colores es $2^n$)

La **resolución** de la imagen matricial mide la calidad visual en lo que respecta al grado de detalle que puede apreciarse en la misma. Lo más común es expresar la resolución mediante dos números enteros: el de columnas de píxeles (número de píxeles de cada línea) y el de líneas. O también, mediante el producto de ambos. Así, de una cámara con una resolución 10.320 por 7.752 se dice que tiene $10.320 \times 7.752 = 80.000.640 \approx 80$ megapíxeles.

A veces con el término «resolución» se designa a la **densidad de píxeles**, medida en píxeles por pulgada (ppi).

Los conceptos sobre digitalización resumidos antes se aplican también a las imágenes matriciales, cambiando el dominio del tiempo por el del espacio: la «pixelación» es un muestreo en el espacio. Pero hay un cambio en la terminología: lo que en la señal temporal muestreada se llama «resolución» (número de bits por muestra) aquí es «profundidad de color», y lo que en imágenes se llama «resolución» corresponde a la frecuencia de muestreo.

El teorema del muestreo normalmente se formula para funciones de una variable, el tiempo, pero es aplicable a funciones de cualquier número de variables, y, por tanto, a las imágenes digitalizadas. Y fenómenos como el *aliasing* aparecen también en estas imágenes: en la televisión convencional podemos observarlo (en alta definición menos) cuando una persona viste una camisa de rayas delgadas y muy juntas (frecuencia espacial grande).

### Gráficos vectoriales

La representación «vectorial» de las imágenes es un enfoque totalmente distinto al de la representación «matricial». En lugar de «ver» la imagen como una matriz de píxeles, se *describe* como un conjunto de objetos primitivos (líneas, polígonos, círculos, arcos, etc.) definidos matemáticamente en un lenguaje.

Para entenderlo, veamos un ejemplo concreto en el lenguaje SVG *(Scalable Vector Graphics)*, que es un estándar del W3C (Consorcio WWW). En la figura 5.6 puede usted ver el código en SVG y el resultado visual de una imagen sencilla. Aun sin saber nada del lenguaje (que está basado en XML), es fácil reconocer la correspondencia entre las sentencias y la imagen. Después de la cabecera (las cinco primeras líneas), aparecen tres declaraciones de rectángulos («`<rect... />`») con sus propiedades (posición, dimensiones y color).

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     width="100%" height="100%">
<rect x="0" y="0" width="240" height="50"
      stroke="red" stroke-width="1" fill="red" />
<rect x="0" y="50" width="240" height="70"
      stroke="yellow" stroke-width="1" fill="yellow" />
<rect x="0" y="120" width="240" height="50"
      stroke="red" stroke-width="1" fill="red" />
</svg>
```

**Figura 5.6**: Bandera.svg.

Salvo que se trate de un dibujo muy complicado, el tamaño en bits de una imagen descrita en SVG es mucho menor que el necesario para representarla como imagen matricial. El ejemplo de la figura, con una resolución 241 por 171 y una profundidad de color de 8 bits, necesitaría 241×171 = 41.211 bytes. Utilizando un formato comprimido (PNG) con la misma resolución y la misma profundidad de color (así se ha hecho en el original de este documento) ocupa solamente 663 bytes. Pero el fichero de texto en SVG tiene sólo 486 bytes (y además, se puede comprimir, llegando a menos de 300 bytes).

La representación vectorial tiene otra ventaja: es independiente de la resolución. Una imagen matricial tiene unos números fijos de píxeles horizontales y verticales, y no se puede ampliar arbitrariamente sin perder calidad (sin «pixelarse»). La imagen vectorial se puede ampliar todo lo que se necesite: la calidad sólo está limitada por el hardware en el que se presenta.

Sin embargo, la representación vectorial sólo es aplicable a los dibujos que pueden describirse mediante primitivas geométricas. Para imágenes fotográficas es absolutamente inadecuada.

## 5.4. Representación de imágenes en movimiento

La propiedad más importante de la visión humana aprovechable para la codificación de imágenes en movimiento es la de **persistencia**: la percepción de cada imagen persiste durante, aproximadamente, 1/25 seg. Esto conduce a lo que podemos llamar el *«principio de los hermanos Lumière»*: para conseguir la ilusión de movimiento continuo basta con presentar las imágenes sucesivas a un ritmo de 30 fps. «fps» es la abreviatura de «frames por segundo»; en este contexto, una traducción adecuada de «frame» es «**fotograma**»[5].

Observe que ahora hay un muestreo en el tiempo (añadido, en su caso, a la digitalización de cada fotograma), y el teorema del muestreo sigue siendo aplicable. Habrá visto, por ejemplo, las consecuencias del *aliasing* cuando la escena contiene movimientos muy rápidos con respecto a la tasa de fotogramas (por ejemplo, en el cine, cuando las ruedas de un vehículo parecen girar en sentido contrario).

El problema de aumentar la tasa de fotogramas es que aumenta proporcionalmente la tasa de bits. En animaciones en las que se puede admitir una pequeña percepción de discontinuidad la tasa puede bajar a 12 fps. En cinematografía son 24 fps, pero duplicados o triplicados ópticamente (la película

[5]Es importante la matización «en este contexto». En transmisión de datos «frame» se traduce por «trama», y en otros contextos por «marco»

avanza a 24 fps, pero con esa tasa se percibiría un parpadeo; el proyector repite dos o tres veces cada fotograma). En televisión analógica los estándares son 25 fps (PAL) o 30 fps (NTSC), pero en realidad se transmiten 50 o 60 campos por segundo entrelazados (un campo contiene las líneas pares y otro las impares). La HDTV en Europa utiliza 50 fps. Los monitores LCD suelen configurarse para 60 o 75 fps.

Sabiendo la duración de un vídeo, el número de fps y la resolución de cada fotograma es fácil calcular la capacidad de memoria necesaria para su almacenamiento y de ancho de banda para su transmisión. Por ejemplo, la película «Avatar» tiene una duración de 161 minutos. Si la digitalizamos con los parámetros de HD (1.980×1.080, 50 fps, 32 bpp), pero sin compresión, la tasa de bits resulta $1.980 \times 1.080 \times 50 \times 32 = 3.421{,}44 \times 10^6$ bps. Es decir, necesitaríamos un enlace de más de 3 Gbps para transmitir en tiempo real solamente el vídeo. Toda la película (sin sonido) ocuparía $3.421{,}44 \times 10^6 \times 161 \times 60/8 \approx 4{,}13 \times 10^{12}$ bytes. Es decir, $4{,}13 \times 10^{12}/2^{40} \approx 3{,}76$ TiB. Un disco de 4 TiB sólo para este vídeo. Es evidente la necesidad de compresión para estas aplicaciones.

# Capítulo 6

# Detección de errores y compresión

A diferencia del «mundo analógico», en el que una pequeña perturbación en una señal normalmente pasa desapercibida, en el digital la sola alteración de un bit entre millones puede cambiar totalmente el mensaje. En algunas aplicaciones (transferencia de ficheros, transacciones bancarias...) no puede admitirse ningún error. En otras (imágenes, comunicaciones de voz, *streaming* multimedia...) basta con reducirlos a un mínimo aceptable.

Los algoritmos para detectar si se ha producido algún error en la transmisión de un mensaje se basan, en general, en *añadir redundancia* al mensaje. Como ilustra la figura 6.1, el mensaje original se envía en bloques sucesivos, y a cada bloque se le añaden bits de comprobación, que no aportan información para el receptor, pero que sirven para verificar si se ha producido o no algún error.

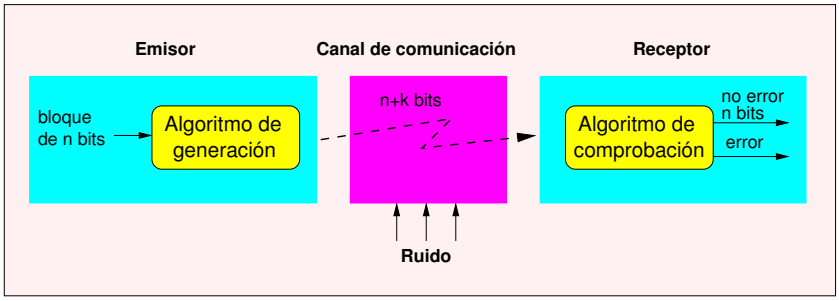



**Figura 6.1**: Detección de errores.

Hay algoritmos que permiten no sólo detectar errores, también pueden corregirlos. Veremos aquí el principio de tres algoritmos de detección comunes, y de las acciones posibles cuando se detecta un error. Están concebidos para errores ocasionales en bloques relativamente pequeños de datos e implementarse en hardware. No entraremos en otros, basados en técnicas *hash*, orientados a verificar la «integridad» de ficheros grandes (es decir, comprobar que el fichero no ha sido modificado), como MD5 y SHA.

Después estudiaremos un asunto en cierto modo «opuesto» y sin embargo complementario: si la detección de errores se basa en añadir redundancia, la compresión consiste en *reducir redundancia*. Si un mensaje de $N$ bits puede transformarse mediante un algoritmo en otro de $N_C$ bits de modo que $k = N/N_C > 1$ y que no haya pérdida en la información que el agente receptor interpreta (o que la pérdida sea aceptable), se dice que el algoritmo tiene un **factor de compresión** $f_c = k$:1, o que el **porcentaje de compresión** es $C = 100/k\,\%$.

Los fundamentos de la detección de errores y la compresión los ha estudiado usted también en la asignatura «Redes de comunicaciones». El contenido de este capítulo es, pues, redundante, pero complementario, al enfocarlo sobre la representación de los distintos tipos de datos.

## 6.1. Bit de paridad

El procedimiento más básico para detectar errores en un mensaje binario consiste en añadir un bit cada $n$ bits, de modo que el número total de «1» en los $n + 1$ bits sea o bien par («paridad par») o bien impar («paridad impar»).

Como ejemplo, consideremos un mensaje de texto que consiste solamente en la palabra «Hola». Los caracteres se codifican en ASCII (siete bits por carácter) y se transmiten en un orden extremista mayor: primero, «H» (0x48 = 1001000), etc. Si convenimos en añadir un bit de paridad impar, a la codificación de «H» se le añade en el extremo emisor un octavo bit, que en este caso deberá ser «1» para que el número total sea impar. Suponiendo que el convenio es que el bit de paridad se añade en la posición más significativa, para los cuatro caracteres del mensaje resulta esto:

| Carácter | ASCII (hex.) | ASCII (bin.) | | Resultado | Resultado (hex.) |
|---|---|---|---|---|---|
| H | 0x48 | 1001000 | $\rightsquigarrow$ | *1*1001000 | 0xC8 |
| o | 0x6F | 1101111 | $\rightsquigarrow$ | *1*1101111 | 0xEF |
| l | 0x6C | 1101100 | $\rightsquigarrow$ | *1*1101100 | 0xEC |
| a | 0x61 | 1100001 | $\rightsquigarrow$ | *0*1100001 | 0x61 |

El receptor, que, naturalmente, debe estar diseñado con los mismos convenios (en este ejemplo, grupos de ocho bits incluyendo el de paridad en la posición más significativa, y paridad impar) comprueba la paridad cada ocho bits. Si en algún grupo hay un número par de «1» señala el error. En caso contrario, quita el bit de paridad, con lo que recupera la codificación original.

La ventaja del procedimiento es su sencillez de implementación: el bit de paridad se calcula en el emisor haciendo una operación lógica «OR excluyente» (apartado 4.2) de todos los bits del grupo, y en el receptor se comprueba del mismo modo. Y esta operación es fácil de realizar con componentes electrónicos, por lo que la incorporan algunos componentes de hardware, como el bus PCI.

Por ejemplo («$\oplus$» significa «or excluyente»):

- El emisor, E, tiene que enviar 0x48 = 1001000 (ASCII de «H»)
- E calcula el bit de paridad: $1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$
- Si el convenio es de paridad impar, el bit de paridad es $\overline{0} = 1$
- E envía 11001000
- Hay un error en el bit menos significativo, y el receptor, R, recibe 11001001
- R calcula el bit de paridad: $1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$
- Como el convenio es de paridad impar, debería haber resultado 1, por lo que R detecta que ha habido un error.

El inconveniente es que si se alteran dos bits (o, en general, un número par de bits), el algoritmo no lo detecta.

El procedimiento del bit de paridad par (con el bit de paridad en la posición menos significativa) es un caso particular y degenerado de otros más robustos y muy utilizados en redes y en las transferencias con discos: los CRC (Cyclic Redundancy Check), que veremos en el apartado 6.3.

## 6.2. Suma de comprobación *(checksum)*

Los procedimientos basados en sumas de comprobación son algo más robustos y más adecuados para mensajes largos que el del bit de paridad. Se usan, por ejemplo, en los protocolos TCP/IP.

En general, una suma de comprobación, o de prueba, o «suma *hash*», es un conjunto de bits que se obtiene aplicando una determinada función a un bloque de datos. El mensaje original se descompone en bloques, a cada uno se le aplica la función y el emisor envía el bloque y la suma de comprobación. El receptor aplica la misma función sobre cada bloque recibido y compara el resultado con la suma enviada.

Hay muchos modos de definir la función, y, por tanto, muchos algoritmos de detección de errores. De hecho, como hablamos de «función» y no de «suma», aquí estarían incluidos los procedimientos que describiremos en el siguiente apartado. Veamos uno de los más sencillos: el de la **suma modular**:

- Cada bloque tiene $m$ grupos de $n$ bits cada uno.
- En cada bloque se suman los $m$ grupos como números sin signo y descartando los bits de desbordamiento (es decir se hace una suma módulo $2^n$) y se envía, junto con el bloque, el complemento a 1 del resultado.
- En el receptor se suman módulo $2^n$ los $m$ grupos y el complemento a 1 recibido.
- El resultado tiene que ser cero. En caso contrario se ha producido un error en ese bloque.

Veamos un ejemplo con un mensaje de texto corto: «Hola ¿qué tal?» codificado en ISO-8859-15, con bloques de ocho grupos y ocho bits por grupo ($m = 8, n = 8$):

El primer bloque contiene las codificaciones de los ocho primeros caracteres, es decir, «Hola ¿qu» (Esto es porque estamos suponiendo un código ISO; si fuese UFT-8 tendría un carácter menos, porque «¿» se codificaría con dos bytes). El emisor envía los caracteres, y a medida que lo hace va calculando la suma modular de los grupos. Tras los ocho bytes, envía el complemento a 1 de la suma. El receptor, a medida que recibe bytes va realizando la suma, y tras recibir ocho bytes, hace el complemento a 1, y comprueba que es igual a cero. Concretamente:

| **Carácter** | **ISO Latin-9 (hex)** | **ISO Latin-9 (bin)** | | **En el receptor** |
|---|---|---|---|---|
| H | 48 | 01001000 | ⇝ | 01001000 |
| o | 6F | 01101111 | ⇝ | 01101111 |
| l | 6C | 01101100 | ⇝ | 01101100 |
| a | 61 | 01100001 | ⇝ | 01100001 |
| | 20 | 00100000 | ⇝ | 00100000 |
| ¿ | BF | 10111111 | ⇝ | 10111111 |
| q | 71 | 01110001 | ⇝ | 01110001 |
| u | 75 | 01110101 | ⇝ | 01110101 |
| **Checksum:** | (3)*49* | *01001001* | | |
| **Complemento a 1:** | | *10110110* | ⇝ | *10110110* |
| | | | **Suma:** | *11111111* |

Si el resultado es cero, como aquí, la probabilidad de que haya habido alteraciones en uno o varios bits es pequeña.

Para completar el ejemplo, el segundo bloque, que sólo contiene seis caracteres («é tal?»), se completa con dos caracteres 0x00 («NUL»):

| Carácter | ISO Latin-9 (hex) | ISO Latin-9 (bin) | | En el receptor |
|---|---|---|---|---|
| é | E9 | 01100001 | ⇝ | 01100001 |
| | 20 | 00100000 | ⇝ | 00100000 |
| t | 74 | 10111111 | ⇝ | 10111111 |
| a | 61 | 01100001 | ⇝ | 01100001 |
| l | 6C | 01101100 | ⇝ | 01101100 |
| ? | 3F | 00111111 | ⇝ | 00111111 |
| NUL | 00 | 00000000 | ⇝ | 00000000 |
| NUL | 00 | 00000000 | ⇝ | 00000000 |
| **Checksum:** | (2)*89* | *10001001* | | |
| **Complemento a 1:** | | *01110110* | ⇝ | *01110110* |
| | | | **Suma:** | *11111111* |

## 6.3. Códigos CRC

Los CRC (códigos de comprobación, o de verificación, por redundancia cíclica) son algo más complejos, pero detectan muchos más errores que las sumas de comprobación. Tienen una fundamentación matemática: la teoría algebraica de anillos. Sin entrar en estos fundamentos, resumamos la forma de aplicarlos.

La comprobación se basa en un *polinomio generador* de grado $k$, $G(x)$, que se aplica en el emisor y en el receptor. Por ejemplo, $G(x) = x^{16} + x^{12} + x^5 + 1$ es el código CRC-16-CCITT, que se usa en muchos protocolos de comunicaciones, como X.25 y Bluetooth.

La esencia del procedimiento es la siguiente:

- El mensaje se envía en bloques sucesivos de $n + k$ bits: $n$ bits de datos y $k$ bits de comprobación. Los $n$ bits de datos de cada bloque se consideran coeficientes (binarios) de un polinomio, $M(x)$, de grado $n - 1$.
  Por ejemplo, si $n = 8$, el polinomio para los datos 0x9B = 10011011 es $M(x) = x^7+x^4+x^3+x+1$, y si $n = 32$, el polinomio para 0xA0048001 = 10100000 00000100 10000000 00000001 es $M(x) = x^{31} + x^{29} + x^{18} + x^{15} + 1$

- A cada bloque se le añaden $k$ bits, de modo que el polinomio resultante, $T(x)$, de grado $n - 1 + k$, sea divisible por $G(x)$.

- El receptor comprueba que los $n + k$ bits recibidos corresponden a un polinomio divisible por $G(x)$; de no ser así se ha producido algún error.

Todas las operaciones se hacen en módulo 2 ($1 + 1 = 0$, sin acarreo; $0 - 1 = 1$, sin acarreo), lo que hace muy fácil su implementación en hardware mediante puertas XOR (OR excluyente) y un registro de desplazamiento. Más detalladamente, el segundo paso se puede realizar así:

1. Añadir $k$ ceros a la derecha de los $n$ bits de datos; el polinomio correspondiente resulta ser $M(x) \times x^k$
2. Dividir (módulo 2) $M(x) \times x^k$ entre $G(x)$, obteniendo un resto (o residuo) $R(x)$.
3. Calcular $T(x) = M(x) \times x^k - R(x)$ (módulo 2); esta operación se realiza simplemente añadiendo los $k$ bits de $R(x)$ a la derecha de los $n$ bits de datos originales (puesto que en módulo 2 la resta es igual que la suma).

Veamos un ejemplo concreto. Supongamos que el mensaje a enviar es 0x35B = 001101011011, al que corresponde el polinomio $M(x) = x^9 + x^8 + x^6 + x^4 + x^3 + x + 1$, y que se utiliza como polinomio generador $G(x) = x^4 + x + 1$ (10011, $k = 4$).

1. Añadir $k = 4$ ceros: `0011010110010000`

2. Dividir módulo 2 por $G(x)$ (`10011`):

```
0 0 1 1 0 1 0 1 1 0 1 1 0 0 0 0 |1 0 0 1 1
  - 1 0 0 1 1                    1 1 0 0 0 0 1 0 1 0
    0 1 0 0 1 1                  (Cociente: x^9 + x^8 + x^3 + x)
    - 1 0 0 1 1
      0 0 0 0 0 1 0 1 1 0
            -1 0 0 1 1
             0 0 1 0 1 0 0
               - 1 0 0 1 1
                 0 0 1 1 1 0   (Resto: x^3 + x^2 + x)
```

   Puede usted comprobar que al sumar el resto al producto del cociente por el divisor, $G(x)$ (y operando en módulo 2, es decir, por ejemplo, $1 \cdot x^9 + 1 \cdot x^9 = 0 \cdot x^9$), resulta el dividendo, es decir, $M(x) \times x^4$.

3. Añadir los cuatro bits de redundancia: `0010011011100`

Ahora bien, observe que solamente se utiliza el resto de la división: el cociente es irrelevante. Por tanto, basta con realizar las operaciones indicadas en la parte izquierda del ejemplo, que se reducen a una sucesión de operaciones XOR y de desplazamientos sobre los bits del dividendo.

Si al receptor le llegan estos 12+4 bits, hace la comprobación de que corresponden a un polinomio divisible por $G(x) = x^4 + x + 1$:

```
0 0 1 1 0 1 0 1 1 0 1 1_1 1 1 0
    1 0 0 1 1
    0 1 0 0 1 1 1 0 1 1_1 1 1 0
      1 0 0 1 1
      0 0 0 0 0 1 0 1 1_1 1 1 0
                1 0 0 1 1
                0 0 1 0_0 1 1 0
                    1 0 0 1 1
                    0 0_0 0 0 0 (Resto: 0)
```

Como decíamos en el apartado 6.1, el procedimiento del bit de paridad par (o «CRC-1») es un caso particular, con $G(x) = x + 1$. Puede usted comprobarlo como ejercicio.

El análisis de la capacidad de un código CRC para detectar errores es fácil, teniendo en cuenta que un error en el bit $t$ (contando desde el menos significativo, el primero en llegar al receptor) equivale a sumar (siempre módulo 2) un término $x^t$ al mensaje transmitido, $T(x)$. Y un número arbitrario de errores de bit equivale a sumar un polinomio $E(x)$ de grado igual o inferior al mensaje original, $M(x)$. Los errores pasarán desapercibidos en el extremo receptor únicamente si $T(x) + E(x)$ da como resultado un polinomio múltiplo de $G(x)$.

Ahora bien, por la construcción de $T(x)$ éste es un múltiplo de $G(x)$, es decir, $R(T(x)/G(x)) = 0$. Como consecuencia, el resto de la división por $G(x)$ del mensaje recibido, $T(x) + E(x)$, sólo depende de $E(x)$:

$$R(x) = R(\frac{T(x) + E(x)}{G(x)}) = R(\frac{E(x)}{G(x)})$$

independientemente del mensaje original, $M(x)$.

Cualquier polinomio generador $G(x)$ que contenga el término $x^0$ (es decir, «1») detectará un error de un solo bit ($E(x) = x^t$). Esto ya ocurre en el caso del bit de paridad par, puesto que $x^t$ no es múltiplo de $G(x) = x + 1$. Sabemos que el bit de paridad no detecta un número par de errores. En efecto, si hay dos errores contiguos, $E(x) = x^{t+1} + x^t = x^t(x + 1)$, que es múltiplo de $x + 1$. Y si los dos errores están separados por $l$ bits correctos, el polinomio de error es $E(x) = x^{t+l} + x^t = x^t \cdot (x^l + 1)$. En módulo 2 es fácil comprobar[1] que $x^l + 1$ es múltiplo de $x + 1$ para cualquier valor de $l$, por lo que el bit de paridad tampoco lo detecta. La condición para que un polinomio generador $G(x)$ detecte dos errores separados por un máximo de $l$ bits, además de que contenga el término $x^0 = 1$, es que $x^i + 1$ no sea múltiplo de $G(x)$ para ningún valor de $i$ hasta $i = l$. Los polinomios que se utilizan están elegidos de modo que cumplen esta condición para valores muy elevados de $l$.

Pero la gran virtud de CRC es su capacidad para detectar **ráfagas de errores** *(burst errors)*, muy frecuentes en la transmisión serial de datos binarios. Porque la modificación ocasional de uno o dos bits es posible, pero lo más normal es que la alteración afecte a toda una secuencia. Piense, por ejemplo, en el efecto de una partícula de polvo sobre la superficie magnética de un disco o de una hendidura microscópica en un CD: con las altas velocidades de rotación y densidades de grabación que se utilizan normalmente, la «sombra» afecta a muchos bits. O en el efecto sobre una transmisión inalámbrica de una perturbación electromagnética producida al accionar un interruptor cercano: con una tasa de 54 Mbps (WiFi 802.11g), un ruido de un microsegundo puede afectar a 54 bits.

Una ráfaga se define como una secuencia de bits en la que el primero y el último son erróneos y no contiene ninguna subsecuencia de $m$ bits correctos, de modo que el último bit de la ráfaga y el primero de la siguiente están separados por al menos $m$ bits. Por ejemplo, en este caso:

```
                          ↓←    ráfaga    →↓
Mensaje original:  ...1 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0...
Mensaje recibido:  ...1 0 0 1 0 x x x x x x x x 1 0 0 1 0...
```

tenemos una ráfaga de 10 bits. No todos los bits interiores de la ráfaga («x») tienen por qué ser erróneos: en un caso extremo pueden ser todos correctos, pero en ese caso los $m = 8$ bits anteriores y posteriores a la ráfaga deben ser correctos (en caso contrario debería ser considerada parte de una ráfaga más larga; $m$ se llama «intervalo de guarda», *«guard band»*).

En este ejemplo, si todos los bits interiores de la ráfaga son correctos, $E(x) = x^t \cdot (x^9 + 1)$. En general, para una ráfaga de longitud $f$ que tenga su bit menos significativo en la posición $t$, $E(x) = x^t \cdot E_{f-1}(x)$, donde $E_{f-1}(x)$ es un polinomio de grado $f - 1$ que tiene un término $x^0 = 1$. El resto de la división por $G(x)$ es:

$$R(x) = R(\frac{E(x)}{G(x)}) = R(\frac{x^t \cdot E_{f-1}(x)}{G(x)})$$

Si $G(x)$ contiene el término $x^0 = 1$ no puede tener $x^t$ como factor, por lo que no puede reducir el grado del denominador $G(x)$. Para que el resto fuese cero $E_{f-1}(x)$ debería ser múltiplo de $G(x)$, lo que es imposible si $f$ (longitud de la ráfaga) es menor o igual que $k$ (grado de $G(x)$).

[1] $x^l + 1 = (x + 1) \cdot \sum_{i=0}^{l-1} x^i$

Así, un código como el CRC-16-CCITT de 16 bits mencionado antes detectará cualquier ráfaga que tenga 16 bits o menos.

Si la longitud de la ráfaga es $f = k + 1$, el resto sólo puede resultar cero en el caso muy especial de que $E_{f-1}(x) = G(x)$, es decir, que la configuración de todos sus bits sea exactamente la misma que la de $G(x)$. Como el primero y el último bit de $E_{f-1}(x)$ son «1», esto depende de los $f - 2$ bits intermedios. Suponiendo equiprobables todas las combinaciones, la probabilidad de que se dé una combinación concreta es $2^{-(f-2)} = 2^{-(k-1)}$. Para $k = 16$ el resultado es que el porcentaje de ráfagas de 17 bits que se detectan es $100 \cdot (1 - 2^{-15}) = 99{,}997\,\%$.

Un análisis más detallado demuestra que con un polinomio generador de grado $k$ y con el término $x^0 = 1$ la probabilidad de que una ráfaga de $f > k + 1$ bits quede sin detectar (es decir, dé un resultado $R(E(x)/G(x)) \neq 0$) es $2^{-k}$. Para $k = 16$ resulta que el algoritmo detecta el 99,998 % de todas las ráfagas de más de 17 bits.

En las redes locales de datos como Ethernet y WiFi, y también en los accesos a medios de almacenamiento, se utilizan polinomios generadores de 32 bits: los datos se transfieren en bloques sucesivos llamados «tramas» *(frames)*, y cada trama incluye, al final, sus 32 bits de CRC. Esto permite detectar cualquier ráfaga de hasta 32 bits, la probabilidad de que una ráfaga de 33 bits no sea detectada es $2^{-31}$, y la probabilidad de que no sea detectada una ráfaga de más de 33 bits es $2^{-32}$ (uno entre cuatro miles de millones). La generación de estos 32 bits en el extremo emisor y la comprobación en el receptor se realiza con hardware de manera muy rápida para no influir en la tasa de bits. Varios algoritmos de compresión, como ZIP y gzip y bzip2, utilizan también códigos CRC de 32 bits, aunque en este caso, normalmente, la generación y la comprobación se hacen mediante software.

## 6.4. Control de errores

Detectado el error, es preciso corregirlo. Los dos enfoques principales son los que se conocen con las siglas «ARQ» y «FEC»:

### Solicitud de repetición automática, ARQ *(Automatic Repeat Request)*

Los métodos de ARQ datan de los primeros tiempos de las comunicaciones de datos. Ya en el código ASCII se introdujeron dos caracteres de control para aplicarlos: ACK (acknowledge, 0x006) y NAK (negative acknowledge, 0x025).

Tras cada comprobación, el receptor envía un mensaje de reconocimiento positivo (si no ha detectado error) o negativo (si lo ha detectado). En el caso negativo, o si no recibe el mensaje tras un cierto tiempo *(«timeout»)*, el emisor retransmite las veces que sean necesarias.

La versión más sencilla es la de **parada y espera**: el emisor envía un bloque de bits y no envía el siguiente hasta no recibir respuesta del receptor; si ésta es positiva envía el bloque siguiente, y en caso contrario reenvía el mismo bloque. Requiere que sea posible transmitir en los dos sentidos, pero no simultáneamente: es lo que se llama «semidúplex».

Es relativamente fácil de implementar, pero el inconveniente de la parada y espera es su poca eficiencia. Dependiendo de la aplicación, los retrasos pueden ser inaceptables.

Mucho más eficientes, pero también más complejos, son los métodos de **transmisión continua**: el emisor no espera, pero va analizando los mensajes de respuesta, y si alguno es negativo retransmite desde el bloque en el que se produjo el error. Esto implica dos cosas: por una parte, que la comunicación tiene que ser simultánea en los dos sentidos *(«full duplex»)*, y por otra que los bloques tienen que ir

numerados. Para esto último, en cada bloque, a los bits de datos y de comprobación se les añade una «cabecera» con un número de bloque en binario .

### Corrección de errores en destino, FEC *(Forward Error Correction)*

En algunas aplicaciones no es posible aplicar métodos de ARQ: aquellas en las que no hay un canal de retorno, o en las que los retardos son inaceptables. En esos casos se puede recurrir a métodos de FEC, que se basan en códigos correctores. Estos códigos introducen la redundancia suficiente para que el receptor pueda, no sólo detectar un error en un bloque de bits, sino también recuperar los datos originales. Se utilizan, por ejemplo, en las transmisiones vía satélite, en los módems y en los dispositivos de CD y DVD.

Los códigos correctores tienen también una fundamentación matemática. Como decíamos antes, si en algunas asignaturas le enseñan temas que parecen (y son) abstractos, por ejemplo, modelos de Markov, sepa que éstos son la base de algoritmos como el de Viterbi, que permiten implementar la corrección de errores en la transmisión digital.

Hay métodos que combinan las dos técnicas. Son los de **ARQ híbrido, HARQ** *(Hybrid ARQ)*: si el receptor no puede por sí solo recuperar los datos originales recurre a ARQ con un mensaje de reconocimiento negativo.

## 6.5. Compresión sin pérdidas *(lossless)*

La detección y corrección de errores es de aplicación en la transmisión de datos, ya sea a distancias más o menos largas (redes locales, Internet, comunicaciones por satélite...) o cortas (transferencias por buses, lectura y escritura en un DVD, copia de las fotos de una cámara a un disco...). La compresión de datos es asimismo útil en la transmisión (enviar un flujo de datos por un medio con ancho de banda limitado), pero también para el almacenamiento (guardar un volumen de datos en menos espacio del que en principio requiere).

Por este motivo, si en los apartados anteriores hablábamos de «emisor–mensaje–receptor», en éste y el siguiente, para generalizar, diremos «codificador–datos (comprimidos)–descodificador». Cuando se trata de transmisión el codificador se encuentra en el emisor, los datos comprimidos forman el mensaje y el descodificador está en el receptor. En el caso de almacenamiento, el codificador es un programa que genera un fichero con los datos comprimidos, y el descodificador es otro programa que extrae los datos originales del fichero. Naturalmente, los algoritmos del codificador y del descodificador tienen que ser complementarios. Por eso, particularmente cuando se trata de datos multimedia, a la pareja se le llama **códec**.

Se dice que la compresión de datos es sin pérdidas cuando a partir de los datos comprimidos es posible recuperar exactamente los datos originales. Ya dijimos en la introducción de este capítulo que el principio básico de todos los algoritmos de compresión es reducir la redundancia. Hay muchos métodos para hacerlo, que además se suelen combinar.

### Codificación de secuencias largas o RLE (Run-Length Encoding)

Si en la secuencia de símbolos que forman los datos originales hay subsecuencias de símbolos idénticos, en lugar de codificar uno a uno estos símbolos idénticos se puede codificar sólo uno e indicar cuántas veces se repite. Como ilustración, el mensaje «`AAABBBBBBBBBCCBBFFFDDDDDDDDEEEEGGGGG`» se podría codificar así: `3A9B2C2B3F8D4E5G`, reduciéndose el número total de signos de 36 a 16.

En este ejemplo trivial, como $k = 36/16 = 2,25$, el factor de compresión es 2,25:1. En las aplicaciones reales las secuencias son mucho más largas y los factores de compresión mucho mayores.

Por ejemplo, de la digitalización de una página para fax resultan, aproximadamente, 2.300×1.700 píxeles en blanco y negro, lo que requiere una capacidad de cerca de 4 Mbits. Suponga que la página tiene lo que muestra la figura 6.2: las franjas negras están separadas por 300 píxeles, y cada una tiene 50 píxeles de ancho. En lugar de representar cada línea mediante 1.700 bits podemos crear un código en el que «0» seguido de un número binario de 10 bits indique un número de ceros seguidos, y «1» seguido de otro número indique un número de unos seguidos. Como hay cuatro franjas negras y cinco blancas, cada línea se representaría como una sucesión de $4 \times 11 + 5 \times 11 = 99$ bits, obteniéndose un factor de compresión de 1.700/99:1≈17:1. El código que se utiliza realmente para fax no es éste, pero el principio es el mismo.

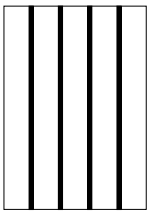


**Figura 6.2**: Una hipotética página de fax.

## Codificación con longitud variable, o codificación Huffman

El método de RLE está limitado porque no analiza los datos como un todo: sólo tiene en cuenta secuencias, olvidando las anteriores. Así, en el último ejemplo está claro que no se aprovecha la redundancia de que todas las líneas son iguales.

Otro enfoque consiste en considerar todo el conjunto de datos y codificar los distintos símbolos con más o menos bits según que el símbolo aparezca con menos o más frecuencia. La idea es antigua: ya la aplicó intuitivamente Samuel Morse para su famoso código (circa 1850). El esquema de codificación y el algoritmo correspondiente más conocido es el que elaboró David Huffman en 1951 (siendo estudiante en el MIT, y como trabajo de una asignatura llamada «Teoría de la información»).

No presentaremos aquí el algoritmo, que se basa en unas estructuras llamadas «árboles binarios». Solamente un sencillo ejemplo para comprender su principio: la secuencia de 36 caracteres `AAABB...` que veíamos antes.

Si contamos el número de veces que aparece cada símbolo, podemos codificar el más frecuente como «1», el siguiente como «01», el siguiente como «001», etc., como indica la tabla 6.1.

| | N | Cod. |
|---|---|---|
| B | 11 | 1 |
| D | 8 | 01 |
| G | 5 | 001 |
| E | 4 | 0001 |
| F | 3 | 00001 |
| A | 3 | 000001 |
| C | 2 | 0000001 |

**Tabla 6.1**: Ejemplo de código Huffman.

Con este código, el dato original se codificaría como 000001000001... El número total de bits es 11×1+8×2+5×3+4×4+3×5+3×6+2×7 = 105 en lugar de los $36 \times 8 = 288$ que resultan codificando todos los caracteres en ISO Latin9. Observe que, tal como se han elegido las codificaciones, no hay ambigüedad en la descodificación.

Como el código se elabora en el codificador de manera *ad-hoc* para unos datos originales concretos, quizás se esté usted preguntando «¿cómo sabe el descodificador cuál es el código?» Y la respuesta, efectivamente, es la que también estará pensando: los datos codificados tienen que ir acompañados del código. En este ejemplo trivial la tabla con el código ocuparía más bits que el mismo dato, pero en los casos reales los datos originales son muchos más voluminosos, y los datos del código son proporcionalmente mucho más pequeños.

## Codificación por diccionario

En la codificación Huffman cada símbolo, o cada secuencia de bits de longitud fija, se codifica con un conjunto de bits de longitud variable. En la codificación por diccionario es al revés: se identifican

secuencias de bits de longitud variable que se repiten y a cada una se le asigna una codificación de longitud fija. Por ejemplo en esta cadena:

```
010-10100-1000001-10100-1000001-1000001-010-1000001-010-10100-010
```

| Secuencia | Índice | Cód. |
|---|---|---|
| 010 | I0 | 00 |
| 10100 | I1 | 01 |
| 1000001 | I2 | 10 |

**Tabla 6.2**: Ejemplo de diccionario.

podemos identificar secuencias que se repiten (los guiones sólo están puestos para visualizarlas, no forman parte de los datos) y construir el «diccionario» de la tabla 6.2. Los datos codificados se construyen con las codificaciones sucesivas de los índices («entradas» del diccionario), en este caso, I0-I1-I2-I1-I2-I2-I0-I2-I0-I1-I0, que dan lugar a $2 \times 11 = 22$ bits en lugar de los 56 originales (en este ejemplo trivial el índice se codifica con sólo dos bits).

En principio puede usted pensar que, como el diccionario se construye *ad-hoc* para los datos originales, sería necesario, como en la codificación Huffman, adjuntar este diccionario a los datos codificados para poder descodificarlos. Y así sería si el algoritmo fuese como sugiere el ejemplo. Pero el ejemplo está trivializado: el diccionario incluye también subsecuencias, y lo maravilloso del método es que el algoritmo que construye incrementalmente el diccionario conforme se van codificando los datos originales tiene una pareja: otro algoritmo que permite ir reconstruyendo el diccionario en el proceso de descodificación.

Los algoritmos más conocidos son LZ77 y LZ78 (por sus autores, Lempel y Ziv, y los años de sus publicaciones). De ellos han derivado otros, como el LZW, que es el que se utiliza para comprimir imágenes en el formato GIF, con el que se obtienen factores de compresión entre 4:1 y 20:1.

## Deflate y otros

Deflate es un algoritmo complejo que, esencialmente, combina los métodos de LZ77 y Huffman. Se desarrolló en 1993 para el formato ZIP y el programa PKZIP (el nombre es por su autor, Phil Katz). Luego se adoptó oficialmente como estándar en Internet (RFC 1951, 1996) y es la base de dos formatos muy conocidos: gzip (genérico) y PNG (para imágenes). El factor de compresión de PNG es, en media, entre un 10 % y un 30 % mejor que el de GIF.

Se han elaborado otros algoritmos derivados de LZ77, y algunos han dado lugar a productos patentados. Uno de los más conocidos es RAR (Roshal ARchive), desarrollado por el ruso Eugene Roshal en 1993 y muy utilizado en entornos Windows.

Hay otros métodos que utilizan varias de las técnicas anteriores y otras que no hemos mencionado. Por ejemplo, bzip2 y 7-Zip, muy eficientes en factor de compresión, pero que exigen más recursos (fundamentalmente, tiempo de ejecución del algoritmo de compresión). 7-Zip es, realmente, un formato con arquitectura abierta que permite acomodar distintos métodos de compresión: Deflate, bzip2, etc.

## Compresores y archivadores

En los párrafos anteriores hemos aludido a algunos algoritmos de compresión (RLE, Huffman, LZ77, LZ78, LZW, Deflate) y algunos formatos de ficheros para imágenes (GIF, PNG) o genéricos (ZIP, gzip, RAR, bzip2, 7-Zip) en los que se aplican esos algoritmos. Ahora bien, entre estos últimos hay dos tipos: gzip y bzip2 son, simplemente, *compresores*. ZIP, RAR y 7-Zip son también *archivadores*.

Aquí conviene explicar la diferencia entre **fichero** *(file)* y **archivo** *(archive)*. Son cosas distintas que suelen confundirse debido al empeño de cierta empresa dominante en Informática de traducir «file» por «archivo»[2].

[2]Como tampoco debería traducirse «directory» por «carpeta»: ésta sería la traducción de «folder».

Un fichero es un conjunto de datos codificados en binario (comprimidos o no), almacenados en un soporte no volátil con un nombre asociado, y disponible para los programas que pueden interpretar esos datos. (Ese conjunto de datos puede ser, en sí mismo, un programa).

Un archivo es un fichero que contiene varios ficheros junto con datos sobre estos ficheros («metadatos») que permiten extraer los ficheros mediante un programa adecuado.

Pues bien, gzip y bzip2 sólo son compresores: su entrada es un fichero y su salida es otro fichero comprimido. ZIP, RAR y 7-Zip son, también, archivadores: pueden recibir como entrada varios ficheros y generar un archivo comprimido. Además, incluyen, opcionalmente, una función en cuyos detalles no entramos (pero que es bien conocida): el cifrado (a veces llamado «encriptación»).

No es casualidad que gzip y bzip2 tengan su origen en el «mundo Unix», mientras que los otros proceden del «mundo Windows». Uno de los principios de la «filosofía Unix» es que los programas deben hacer una sola cosa (pero hacerla bien) y otro, que han de diseñarse de modo que se puedan encadenar fácilmente. Los programas archivadores en Unix son «`ar`», «`shar`» y el más utilizado, «`tar`». Por ejemplo, para crear un archivo comprimido de todos los ficheros contenidos en el directorio `dir/` se puede utilizar esta orden para el intérprete:

```
tar cf - dir/ | bzip2 > dircompr.tar.bz2
```

- «`c`» significa «archivar» (para extraer, «`x`», y para listar, «`t`»).
- «`f`» significa que a continuación se da el nombre de un fichero para el resultado.
- «`-`» significa que el resultado no vaya a un fichero, sino a la «salida estándar».
- «`dir/`» es el directorio que contiene los ficheros a comprimir (aquí podrían ponerse varios ficheros y/o directorios).
- «`|`» establece, como ya sabe usted por las prácticas, una tubería (*pipe*): la salida estándar de lo que hay atrás se utiliza como entrada para lo que sigue.
- «`bzip2`» comprime lo que le llega y envía el resultado a la salida estándar.
- Finalmente, como también conoce, «`>`» hace una redirección: dirige la salida estándar hacia el fichero que hemos llamado `dircompr.tar.bz2`.

(El programa `tar` incluye una facilidad con la que se puede hacer lo mismo escribiendo menos: `tar cjf - dir/ > dircomp.tar.bz2`. El argumento «`j`» hace que automáticamente se cree la tubería y se ejecute `bzip2`. Con «`z`» se ejecutaría `gzip`).

## 6.6. Compresión con pérdidas *(lossy)*

En la compresión con pérdidas se sacrifica el requisito de que a partir de los datos comprimidos se puedan reconstruir *exactamente* los originales. Típicamente se aplica a datos audiovisuales (o sensoriales, en general), tanto para su almacenamiento como para su transmisión, porque el agente destinatario final de los datos, normalmente humano, es capaz de recuperar la información siempre que la distorsión producida por la pérdida de datos no sea muy grande.

La elección de un algoritmo de compresión es muy dependiente de la aplicación y de los recursos disponibles. En general (sea con pérdidas o sin ellas) es necesario un equilibro entre

- Factor de compresión: cuanto mayor sea, menos memoria se requiere para almacenar los datos, y menos ancho de banda para su transmisión.
- Recursos disponibles: podemos tener un algoritmo muy eficiente (que consiga un factor de compresión muy grande) pero que no sirva porque la memoria disponible para el programa sea insuficiente, o el tiempo necesario para ejecutarlo sea inaceptable.

Con los algoritmos de compresión con pérdidas se consiguen factores de compresión mayores que sin pérdidas, pero aparece un tercer elemento a considerar:

- Grado de distorsión aceptable.

El problema es que este grado es subjetivo. Un ejemplo que se entenderá fácilmente es el de la música codificada en MP3. Para la mayoría de las personas, con el factor de compresión habitual en MP3 (11:1, a 128 kbps), el resultado es más que aceptable. Pero para ciertos oídos exquisitos no (normalmente son los mismos a los que les suena mejor la música grabada en vinilo que en CD).

Ahora bien, como ya sabemos, los datos procedentes de las señales acústicas y visuales se obtienen mediante un proceso de muestreo y cuantificación (apartado 5.1), y en este sentido ya han sufrido «pérdidas». Si se reduce la frecuencia de muestreo, o el número de niveles de cuantificación, para una misma señal se generan menos datos, pero la reconstrucción de la señal es menos fiel. En principio, la «compresión» se refiere al proceso posterior, en el que se aplica un algoritmo sobre los datos ya digitalizados. Sin embargo, aunque teóricamente la diferencia entre los dos procesos está clara, en la práctica es difícil separarlos, porque ambos se combinan para optimizar el resultado. Es por eso que se utilizan los términos más generales «codificación» y «descodificación», entendiéndose que engloban tanto la discretización (conversión de analógico a digital) y la reconstrucción (conversión de digital a analógico) como la compresión y la descompresión. Un **códec**, como hemos dicho al principio del apartado 6.5, es una pareja de codificador y descodificador diseñados de acuerdo con unos convenios sobre discretización y algoritmo de compresión.

Aunque hay muchos tipos de códecs para datos de audio, de imágenes y de vídeo, casi todos comparten dos principios de diseño:

- Se basan en **codificación perceptual**, es decir, el esquema de codificación se diseña teniendo en cuenta características fisiológicas de la percepción. Normalmente, esto implica una transformación del dominio temporal de las señales originales a un dominio en el que se expresa mejor la información que contienen. Por ejemplo, en lugar de representar los sonidos como una sucesión de valores de amplitud en el tiempo, se transforman para representarlos como variaciones en el tiempo de componentes de frecuencias, lo que se aproxima más a los modelos de la percepción humana.

- Esas transformaciones de dominios de representación están perfectamente estudiadas matemáticamente. La más utilizada en los algoritmos de compresión con pérdidas es la **DCT** (transformada discreta del coseno), que es una variante de la transformada de Fourier que ha estudiado usted en la asignatura «Sistemas y señales».

Para concretar, aunque sin entrar en los algoritmos, veamos cómo se aprovechan algunas de esas propiedades de la percepción de distintos tipos de medios.

## Sonidos

El rango de frecuencias audibles es, aproximadamente, de 20 Hz a 20.000 Hz, pero para aplicaciones en las que sólo interesa que la voz sea inteligible (como la telefonía) basta un rango de 300 Hz a 3.500 Hz (tabla 5.1). Como hemos visto en el apartado 5.1, el teorema del muestreo de Nyquist-Shannon permite calcular la frecuencia de muestreo mínima para poder reconstruir la señal. Reduciendo esta frecuencia reducimos el número de bits necesario para codificar la señal, a costa de perder calidad del sonido.

Esta característica de la percepción se aplica al proceso de discretización, pero hay otras que sirven también para la compresión. Por ejemplo, el *enmascaramiento:* dos sonidos de frecuencias próximas que se perciben bien separadamente, pero sólo se percibe el más intenso cuando se producen conjuntamente (enmascaramiento simultáneo) o cuando uno precede a otro (enmascaramiento temporal).

Todo esto se tiene en cuenta para diseñar procedimientos que facilitan la compresión. Por ejemplo, SBC *(Sub-Band Coding)* consiste en descomponer la señal en bandas de frecuencias y codificar cada banda con un número diferente de bits. Forma parte de los algoritmos utilizados en muchos formatos de audio: MP3, AAC, Vorbis, etc.

Hay características que son específicas de la voz humana, no de la percepción, y en ellas se basan algoritmos optimizados para aplicaciones de voz, como CELP *(Code Excited Linear Prediction)*, o AMR *(Adaptive Multi-Rate audio codec)*, que se utiliza en telefonía móvil (GSM y UMTS).

## Imágenes

Una propiedad importante de la percepción visual es que el ojo humano es menos sensible a las variaciones de color que a las de brillo.

Para conseguir una representación «perfecta» del color *(«truecolor»)*, en cada píxel se codifican los tres colores básicos (rojo, verde y azul) con ocho bits cada uno (256 niveles para cada color). Es decir, se utilizan 24 bits (tres bytes) por píxel. Para almacenar una imagen de, por ejemplo, 2.000×2.600 píxeles se necesitan $3 \times 2.000 \times 2.600/2^{20} \approx 14{,}9$ MiB. Obviamente, se puede reducir el tamaño reduciendo la resolución (el número de píxeles), pero también reduciendo los bits necesarios para codificar el color.

En el caso de dibujos en los que no es necesario reproducir un número tan elevado de colores como en la fotografía se utiliza un solo byte para el color. Es común en aplicaciones de dibujo tener una **paleta de colores**: un conjunto de 256 colores que se pueden seleccionar previamente. Cada píxel se codifica con un solo byte, que es un índice a la paleta. Es lo que se hace en el formato GIF (recuerde que en este formato, después de esta discretización, se aplica un algoritmo de compresión sin pérdidas, el LZW).

Para imágenes reales 256 colores son insuficientes, y se aplica un procedimiento más sutil para aprovechar esa propiedad de menor sensibilidad a las variaciones de color («croma») que a las de brillo («luma»). En la representación de un píxel por las intensidades de sus tres colores básicos se mezclan la información de color y la de brillo. Se puede hacer una transformación matemática del espacio de representación RGB (rojo, verde, azul) en un espacio en el que una dimensión corresponda al brillo y las otras dos al color. Hay varias maneras de definir esa transformación. En televisión analógica, las normas NTSC definían el espacio llamado «YIQ», aunque más tarde adoptaron el de PAL, «YUV». En codificación digital se utiliza otro espacio, el «**YCbCr**». Esencialmente, «Y» es el luma, un valor proporcional a R+G+B, y «Cb» y «Cr» proporcionan el croma: son proporcionales a B–Y y R–Y, respectivamente.

La mayoría de los algoritmos de compresión de imágenes y de vídeos (JPEG, MPEG, DivX, Theora, etc.) se apoyan en esta transformación para aplicar el método de **submuestreo del color** *(chroma subsampling)*. Consiste, simplemente, en utilizar más muestras para Y que para Cb y Cr. Hay una notación para expresar la proporción de muestras de cada componente: «(4:*a*:*b*)» significa que en una región de cuatro píxeles de ancho y dos de alto se toman *a* muestras de (Cb,Cr) en la primera línea y *b* en la segunda. En el estándar DV (Digital Video) se usa (4:1:1); en JPEG, en MPEG y en DVD, (4:2:0).

Otro procedimiento común en los algoritmos de compresión es el **muestreo de altas frecuencias**. Se aplica a los coeficientes que resultan de aplicar la DCT: los correspondientes a frecuencias altas se codifican con menos bits.

Los programas que aplican estos algoritmos suelen tener una opción para definir un parámetro de calidad, entre 0 y 100. Cuanto menor es el parámetro mayor es el factor de compresión (y mayores las pérdidas perceptibles).

### Imágenes en movimiento

Al final del capítulo 5 hemos comprobado con un ejemplo concreto la necesidad de comprimir las señales de vídeo. Obviamente, la primera solución es comprimir, uno a uno, los fotogramas. Pero aunque los fotogramas estén comprimidos, basta hacer unos sencillos cálculos para comprobar que con tasas de 30 o más fps resultan unas tasas de bits que exceden la capacidad de los medios de transmisión normales y harían imposible, por ejemplo, el *«streaming»* por Internet. A esa *compresión intrafotograma*, que aprovecha la *redundancia espacial* se le puede añadir una *compresión interfotograma*, que aprovecha la *redundancia temporal*. Esta redundancia procede, por una parte, de otra propiedad fisiológica: el sistema visual no aprecia distorsiones si en la secuencia de fotogramas algunos no son «reales», sino «promedios» de los próximos. Por otra, es muy frecuente que fotogramas sucesivos sólo difieran en algunos detalles (por ejemplo, en una secuencia en la que un objeto se mueve, pero el fondo de la imagen permanece prácticamente constante).

La duplicación óptica del cine aprovecha esa redundancia temporal, pero en los códecs se utiliza de manera más elaborada. En todos los estándares MPEG se aplica lo que se llama **compensación de movimiento**, que, resumidamente, consiste en lo siguiente:

- Una secuencia de fotogramas se estructura en grupos. Cada grupo se llama «GOP» *(group of pictures)*. No todos los fotogramas de un GOP se codifican (comprimidos) con toda la resolución, sólo algunos.
- Los *«I-frames»* (intra) se comprimen normalmente con algoritmos similares a JPEG.
- Los *«P-frames»* (predichos) contienen solamente las diferencias con el fotograma anterior y se comprimen mucho más.
- Los *«B-frames»* (bi-predichos) se obtienen del anterior y del siguiente y se comprimen aún más.

La figura 6.3, tomada de la Wikipedia[3], ilustra muy claramente la idea de este método de compresión.

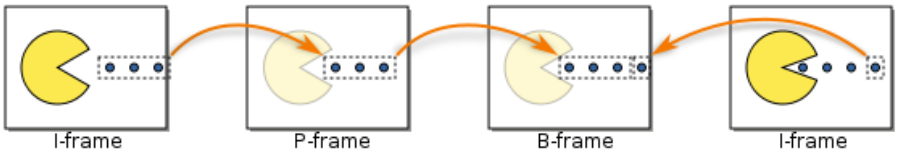



**Figura 6.3**: Fotogramas I, P y B.

[3] http://en.wikipedia.org/wiki/File:I_P_and_B_frames.svg

# Capítulo 7

# Almacenamiento en ficheros

---

En el capítulo 2 introdujimos el concepto de fichero y sus distintos tipos (regular, directorio, etc.), y en el capítulo anterior lo hemos mencionado para matizar la diferencia entre «fichero» y «archivo», que es un tipo de fichero regular (página 108).

Y vimos con bastante detalle las estructuras de datos del sistema de gestión de ficheros (SGF) de los sistemas de tipo Unix, entre ellas los *inodes* (página 49). Decíamos que un *inode* contiene metadatos sobre el fichero: tipo (regular, especial, etc.), permisos, etc. Pero también decíamos que para el sistema operativo un fichero regular no es más que una secuencia de bytes y no sabe nada de su contenido. Los metadatos del *inode* se refieren a propiedades generales del fichero, no a su contenido. También hay metadatos sobre el contenido dentro del mismo contenido.

## 7.1. Identificación del tipo de contenido

Para el sistema de ficheros todos los ficheros regulares son del mismo tipo, pero para los programas que trabajan con ellos es necesario conocer lo que contienen: texto, una imagen, un código ejecutable... Hay varias maneras de hacerlo.

### Extensión del nombre

Una forma muy sencilla de identificar el tipo de contenido es indicarlo en el mismo nombre, con el convenio de que este nombre termine siempre en un punto seguido de varios caracteres que se llaman «**extensión**» y que definen el tipo. El sistema DOS y los primeros Windows tenían la limitación de que la extensión no podía tener más de tres caracteres, pero esta limitación ya no existe. Por ejemplo, los documentos en HTML tienen un nombre terminado en `.htm` o `.html`, las imágenes en JPEG, `.jpg` o `.jpeg`, los programas fuente en lenguaje C, `.c`, en Java, `.java`, etc.

### Metadatos internos

Un método más seguro es incluir los datos sobre el tipo de contenido dentro del mismo fichero, en una posición predeterminada, normalmente al principio y antes de los datos propiamente dichos, formando la **cabecera** *(header)*.

La cabecera puede ser más o menos larga y compleja, desde los ficheros de texto, en los que no existe (salvo en UTF-16 y UTF-32, que pueden llevar dos bytes al principio con el BOM, apartado 3.5), hasta los ficheros de multimedia, que normalmente tienen muchos metadatos. En cualquier caso, la cabecera, si existe, empieza con unos pocos bytes que identifican el tipo de contenido, y se les llama «número mágico».

### Números mágicos y el programa `file`

Algunos ejemplos de números mágicos:

- Los ficheros que contienen imágenes GIF empiezan con un número mágico de seis bytes que son las codificaciones en ASCII de «GIF89a» o de «GIF87a». A continuación, dos bytes con el número de píxeles horizontales y otros dos con el número de píxeles verticales.
- Los que contienen imágenes TIFF empiezan con cuatro bytes. Los dos primeros son las codificaciones ASCII de «II» o de «MM». «II» (por «convenio Intel») indica que el convenio de almacenamiento de los caracteres o palabras de todo el fichero es extremista menor, y «MM» (por «convenio Motorola») indica que es extremista mayor. Los dos siguientes son el carácter «*» codificado en 16 bits, con el valor nulo (0) en el byte más significativo.
- Los que contienen imágenes JPEG empiezan todos con 0xFFD8 y terminan con 0xFFD9 (al final del fichero). Pueden contener más metadatos, según las variantes. Por ejemplo, con el formato «Exif» *(Exchangeable image file format)*, común en las cámaras fotográficas, los metadatos (fecha y hora, datos de la cámara, geolocalización, etc.) forman ya una cabecera relativamente grande.
- Los PNG empiezan con ocho bytes: 0x89504E470D0A1A0A.
- Los PDF, con « %PDF» (0x25504446).
- Los ficheros de clases Java compiladas (.class), con 0xCAFEBABE
- Los archivos comprimidos ZIP, con «PK» (0x504B).
- Los ficheros ejecutables de Windows, con «MZ» (0x4D5A).
- Los ficheros ejecutables de Unix, con «.ELF» (0x7F454C46).
- Los ficheros ejecutables de Mac OS X («Mach-O») pueden empezar con 0xFEEDFACE (PowerPC), con 0xCEFAEDFE (Intel) o con 0xCAFEBABE (Universal, ver apartado 7.4).
- A veces el número mágico no está en los primeros bytes. En los ficheros de archivo `tar` son los caracteres «ustar» a partir del byte 0x101 de la cabecera.

«`file`» es un programa de utilidad para sistemas Unix que identifica el tipo de contenido de un fichero. En principio, se basa en comparar los primeros bytes del fichero con los números mágicos guardados en un fichero de nombre «`magic`» (su localización depende del sistema: puede estar en `/etc/magic`, o en `/usr/share/file/magic`, o en otro directorio). En realidad, hace tres pruebas:

1. Prueba con los metadatos del SGF (contenidos en el *inode*) para ver si es un fichero especial: directorio, dispositivo, etc. Si es así informa de ello y si no, es que es un fichero regular y pasa a la segunda prueba
2. Prueba con los números mágicos de `magic`. Si encuentra coincidencia informa y, dependiendo de lo que ha identificado, continúa leyendo bytes de la cabecera para dar más información. Por ejemplo, para un GIF mira las dimensiones. Si no logra identificar ningún número mágico pasa a la tercera prueba.
3. Prueba de lenguaje. Como los ficheros de texto plano no tienen cabecera ni número mágico, el programa examina la secuencia de los primeros bytes. Normalmente acierta, identificándolo como ASCII, ISO Latin1, UTF-8, etc. Otras pruebas de naturaleza más heurística le permiten identificar tipos particulares de ficheros de texto. Por ejemplo, los documentos HTML tienen al principio «`html`» o «`HTML`», los programas fuente del lenguaje Java normalmente contienen declaraciones como «`public class...`» y sentencias como «`import java.util.*;`», etc.

En cualquier caso, el programa `file` no es infalible, y es fácil «engañarlo». Un ejercicio interesante que puede usted hacer (y así repasa la representación de caracteres y de números): escriba un fichero de texto plano con los caracteres ASCII «GIF89aABCD», guárdelo con un nombre cualquiera (por ejemplo, «ppp») y déselo a `file`. Obtendrá esto:

```
$ file ppp
ppp: GIF image data, version 89a, 16961 x 17475
```

Habiendo identificado el número mágico, ha seguido mirando bytes y ha interpretado los siguientes como las dimensiones. El ejercicio consiste en comprobar que `file` ha interpretado las codificaciones ASCII de «ABCD» como dos números de 16 bits con el convenio extremista menor y de ahí han resultado esas «dimensiones».

### Metadatos externos y tipos MIME

Ya hemos dicho que normalmente los sistemas operativos no mantienen metadatos sobre el contenido de los ficheros en estructuras del SGF, que serían metadatos externos al propio fichero. Pero hay otra forma de codificar metadatos que está relacionada con los protocolos de comunicación. Aunque raramente se utiliza para el almacenamiento en ficheros, que es de lo que estamos tratando, tiene que ver con la identificación del tipo de fichero y es interesante conocerla. Veamos brevemente de qué se trata.

Se trata de MIME (Multipurpose Internet Mail Extensions). En su origen se definió como un modo de extender el correo electrónico (que inicialmente era sólo para texto ASCII) con textos no ASCII, imágenes, sonidos, etc. Se usa en los protocolos de correo (SMTP), y también en los de la web (HTTP) y en otros protocolos.

Es un sistema de identificadores estandarizado por la IANA (Internet Assigned Numbers Authority). Cada identificador consta de un «tipo» y un «subtipo» separados por el símbolo «/». En la cabecera que precede al fichero que contiene los datos se indica el tipo y el subtipo: text/plain, text/html... audio/mp4, audio/ogg... video/mpeg, video/quicktime... De este modo, el receptor (el lector de correo, o el navegador) sabe qué programa tiene que utilizar para interpretar los datos, sin necesidad de fiarse de la extensión ni de abrir el fichero para ver su cabecera.

La IANA tiene definidos nueve tipos (application, audio, example, image, message, model, multipart, text y video) y para cada uno, muchos subtipos.

### Formatos de ficheros

Para facilitar la interoperabilidad (es decir, que, por ejemplo, un vídeo grabado con un sistema Windows pueda reproducirse en Linux) hay convenios sobre cómo deben estructurarse los datos para las distintas aplicaciones. Estos convenios se llaman **formatos**.

En el capítulo 4 hemos estudiado formatos para tipos de datos simples, como son los números, que se representan con pocos bits: una o varias palabras. Ahora hablamos de datos que pueden ocupar un volumen grande, de MiB o GiB, y como estos bits pueden representar datos de naturaleza muy diversa, hay una gran variedad de formatos.

Los detalles de cada formato son muy prolijos y sólo tiene sentido estudiarlos si se va a trabajar, por ejemplo, en el diseño o en la implementación de un procesador de textos o de un códec. En los apartados siguientes veremos algunos aspectos generales de los formatos más conocidos para los distintos tipos de ficheros.

## 7.2. Ficheros de texto, documentos, archivos y datos estructurados

Hay que distinguir entre texto plano y texto con formato (apartado 3.6). En el primer caso, los datos están formados por la cadena de bytes correspondiente a las codificaciones de los caracteres (apartado 3.5), y el contenido del fichero no es más que esa secuencia de bytes, sin más formato. Solamente varían pequeños detalles de un sistema a otro. Por ejemplo, la manera de señalar el final de una línea. En Windows se hace con una combinación de dos caracteres de control: retorno (CR, 0x0D) y nueva línea (LF, 0x0A), mientras que en Unix y en Mac OS X es simplemente LF. En algunos sistemas primitivos se señalaba el fin del texto mediante otro carácter de control: «Control-Z» (0x1A). Ya no es necesario (aunque aún se encuentra en algunos ficheros de texto), puesto que la longitud en bytes del fichero siempre está incluida en los metadatos del SGF (en Unix, en el *inode*, figura 2.22).

La semántica del texto depende del agente que lo lee o lo escribe. Puede ser muchas cosas: un mensaje para ser leído por personas, unos datos de configuración de una aplicación, un programa escrito en un lenguaje de programación, una página web escrita en HTML, una lista de teléfonos...

Si nos referimos a texto con formato, o a otros tipos de documentos (presentaciones, hojas de cálculo, etc.), hay que distinguir, a su vez, entre formatos privativos y estándares.

Los formatos privativos siguen convenios particulares elegidos por su diseñador para representar los detalles de presentación del documento (tipo y tamaño de letra, color, etc.). Estos convenios normalmente se basan en códigos binarios y el contenido del fichero es totalmente ilegible si lo abrimos con una aplicación para leer textos (como `cat`, o `less`). Además, no están documentados públicamente (o lo están bajo licencias muy restrictivas), y los programas de aplicación que escriben y leen los ficheros (por ejemplo, el procesador de textos Word) están ligados al fabricante propietario del formato.

Por el contrario, los formatos de los estándares ODF y OOXML son públicos. Como hemos dicho en el apartado 3.6, un fichero que contenga un documento en uno de estos formatos es un archivo comprimido formado por ficheros XML en los que se codifican todas las particularidades del documento siguiendo las especificaciones del estándar.

### Ficheros de archivos

El formato de un archivo «puro», es decir, sin comprimir, es muy sencillo: es una concatenación de los ficheros, con algunos metadatos. Por ejemplo, en un fichero (archivo) `tar` cada uno de los ficheros que lo componen va precedido de una cabecera con el nombre y los demás metadatos (propietario, permisos, fechas, etc.). Normalmente el archivo se comprime, como hemos visto en un ejemplo en el apartado 6.5 (página 109), formando lo que se llama un «**tarball**». Un fichero comprimido tiene también su cabecera antes de los datos comprimidos. En `gzip`[1] son diez bytes con el número mágico, versión y fecha, y, tras los datos, cuatro bytes para comprobación de errores con CRC-32 (apartado 6.3) y otros cuatro bytes con el tamaño del original (sin comprimir) módulo $2^{32}$.

El formato de los ficheros ZIP, al tratarse de una combinación de archivo y compresión y, opcionalmente, cifrado, es bastante más complejo.

### Ficheros de datos estructurados

Muchas aplicaciones de la Informática (de hecho, las más «tradicionales») tienen que ver con conjuntos de objetos o individuos, cada uno con sus propiedades, o atributos. Los datos correspondientes a esos atributos se reparten en distintos ficheros, y cada fichero contiene un conjunto de valores de atributos para un tipo de objeto determinado.

[1] `https://tools.ietf.org/html/rfc1952`

Por ejemplo, una universidad tiene que mantener datos sobre alumnos, centros, titulaciones, proyectos, etc. Uno de los ficheros puede ser el de centros. Para cada centro tendrá registrado el nombre, las titulaciones que ofrece, las asignaturas que imparte, etc. Los profesores pueden estar en otro fichero, y para cada profesor tendrá también sus propiedades.

Un fichero de este tipo contiene, para cada uno de los objetos o individuos, un conjunto de datos que se llama **registro lógico** *(«logical record»)*[2]. Los registros deben tener una determinada forma, o **esquema**: definición de las partes o **campos** que lo forman. Cada campo corresponde a un atributo, y el esquema define el **tipo** de ese atributo (valores posibles que puede tomar). Por ejemplo, el esquema puede definir el campo «Nombre» de tipo *«string»* (cadena de caracteres), el campo «Fecha de nacimiento» de tipo *«date»*, etc.

La aplicación final deberá facilitar el acceso controlado a esos datos para los distintos tipos de usuarios. Así, el administrador podrá insertar, modificar y borrar registros, mientras que un alumno no podrá hacerlo, pero sí consultar sus calificaciones.

La implementación de estas aplicaciones en «bajo nivel», es decir, programando los detalles de cómo se accede a un campo de un registro, cómo se busca, etc., teniendo en cuenta que se trata de múltiples ficheros interrelacionados, sería muy laboriosa. Existen herramientas genéricas que facilitan su diseño, su construcción y su uso. Son los **sistemas de gestión de bases de datos**, que se estudian en la asignatura «Bases de datos».

## 7.3. Ficheros multimedia

Los ficheros multimedia pueden almacenar una variedad de tipos de datos, y generalmente se agrupan en un fichero compuesto llamado «contenedor». Veamos separadamente los distintos tipos de ficheros que, además de los de texto y documentos, pueden agruparse en un contenedor.

### Ficheros de sonidos

Los algoritmos que aplica un códec (apartado 5.1) y los formatos de los ficheros generados están íntimamente relacionados, y dependen de la aplicación. Los podemos clasificar según el grado de compresión (apartados 6.5 y 6.6) que aplican:

- Sin compresión. La mayoría están basados en PCM (apartado 5.1). De este tipo es el formato estándar de Audio-CD, así como WAV (Microsoft/IBM), AIFF (Apple) y AU (Sun). El primero establece unos parámetros fijos (2 canales muestreados a 44.100 Hz y cuantificados con 16 bits por muestra). Los otros códecs permiten seleccionar los parámetros para adaptar el resultado a diferentes necesidades (ancho de banda, capacidad de almacenamiento, calidad...).
- Con compresión sin pérdidas. Pueden llegar a un porcentaje de compresión de 60-50 % (entre 1,66:1 y 2:1), y, como se puede recuperar íntegramente el original, son muy adecuados para archivos sonoros. Un ejemplo es FLAC *(Free Lossless Audio Codec)*, que utiliza para la compresión RLE (apartado 6.5) y algoritmos de predicción lineal. Otro, WMA Lossless, sólo para sistemas Windows (no está documentado).
- Con compresión con pérdidas. El formato más conocido actualmente es el de MP3 (MPEG-1 Audio Layer 3), que se define como una secuencia de «marcos» *(frames)*. Cada marco tiene

[2]Como ya advertimos en la nota 11 de la página 25, aunque en español los llamemos igual, estos registros son estructuras de datos que no tienen nada que ver con los registros hardware *(«registers»)*, componentes hardware de almacenamiento (apartado 1.3). Por otra parte, los registros físicos para el almacenamiento en cintas (apartado 2.3) pueden agrupar varios registros lógicos.

una cabecera con bits de sincronización y metadatos: la frecuencia de muestreo, la tasa de bits *(bitrate)*, las etiquetas ID3, etc., seguida de datos. Las etiquetas ID3 codifican metadatos como título, artista, número de pista, etc.

Otros formatos importantes de ficheros de sonido con compresión con pérdidas son:

- AAC (Advanced Audio Coding), sucesor de MP3. Forma parte de los estándares MPEG-2 y MPEG-4. Se utiliza en muchos de los últimos dispositivos móviles.
- Vorbis y Opus, libres de patentes (a diferencia de MP3 y AAC).
- WMA (Windows Media Audio), tecnología privativa que forma parte del «Windows Media Framework».

Hay mucha controversia sobre la calidad, pero los resultados prácticos son bastante parecidos. El factor de compresión está entre 10:1 y 12:1 para una tasa de bits de 128 kbps (aunque los códecs permiten aplicar la técnica VBR, *Variable Bit Rate*, con la que para la misma compresión se obtiene mayor calidad).

El factor de compresión no es el único parámetro importante, también hay que considerar la velocidad de ejecución del algoritmo (compresión y descompresión), la latencia en el caso de *streaming*, la calidad percibida subjetivamente, el soporte en hardware y en sistemas operativos, etc.

## Ficheros de imágenes matriciales

Uno de los formatos más sencillos para imágenes matriciales en blanco y negro es el **PBM** *(portable bitmap)*. En la versión ASCII, todo el contenido del fichero es texto plano. En la primera línea, el número mágico, «P1». En la segunda la resolución (número de píxeles): el ancho, *n*, y el alto, *m*, en decimal. Y a continuación, *m* líneas. Cada línea tiene *n* caracteres ASCII que tienen que ser «0» (0x30) para «blanco» o «1» (0x31) para «negro», y pueden ir seguidos o separados por espacios (0x20). El símbolo «#» en cualquier línea indica que lo que sigue hasta el final de la misma es un comentario.

La figura 7.1 muestra el contenido de un fichero en formato PBM y el resultado de su visualización con un programa de gráficos. En realidad, este resultado está ampliado, porque si analiza usted el fichero verá que los trazos de las letras sólo tienen un píxel, y las dimensiones de la imagen (27×9) son muy pequeñas.

```
P1     # Número mágico
27 9   # 27x9 píxeles
#  A  -   R  -   S  -   O
011110011111000111110011110
100001010000101000000100001
100001010000101000000100001
100001010000101000000100001
111111011111000111100100001
100001010100000000010100001
100001010010000000010100001
100001010001000000010100001
100001010000101111100011110
```

ARSO

**Figura 7.1**: ARSO.pbm.

Observe que los «1» y «0» no son bits, sino caracteres. Es decir, aunque sea un «bitmap», cada píxel ocupa realmente ocho bits.

Un formato similar para *pixmaps* es el **PPM** *(portable pixmap)*. En lugar de «1» y «0» contiene, para cada píxel, una tripla de números decimales, «R G B», que representan los niveles de rojo, verde y azul. En la cabecera, después del número mágico «P3» y de las dimensiones, en otra línea se indica el valor máximo, M, de la escala en la que se expresan estos niveles, de 0 a M. Normalmente M = 255, pero en el ejemplo mostrado en la figura 7.2 se ha utilizado M = 5.

```
P3    # Número mágico
27 9  # 27x9 píxeles
5     # Máxima intensidad
#                     A               --                R                     --   \
#                     S               --                O
5 5 5  5 3 0  5 3 0  5 3 0  5 3 0  5 5 5  5 5 5  5 5 5  5 0 5  5 0 5  5 0 5  5 0 5  5 5 5  5 5 5 \
5 5 5  0 5 0  0 5 0  0 5 0  0 5 0  0 5 0  0 5 0  5 5 5  5 5 5  5 0 0  5 0 0  5 0 0  5 0 0  5 5 5
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 5 5  5 5 5  5 0 5  5 5 5  \
0 5 0  5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 5 5  5 5 5  5 0 5  5 5 5  \
0 5 0  5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 5 5  5 5 5  5 0 5  5 5 5  \
0 5 0  5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 3 0  5 3 0  5 3 0  5 3 0  5 3 0  5 5 5  5 0 5  5 0 5  5 0 5  5 0 5  5 0 5  5 5 5  5 5 5  \
5 5 5  0 5 0  0 5 0  0 5 0  0 5 0  5 5 5  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 0 5  5 5 5  5 5 5  5 5 5  \
5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  0 5 0  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 0 5  5 5 5  5 5 5  5 5 5  \
5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  0 5 0  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 0 5  5 5 5  5 5 5  5 5 5  \
5 5 5  5 5 5  5 5 5  5 5 5  5 5 5  0 5 0  5 5 5  5 0 0  5 5 5  5 5 5  5 5 5  5 5 5  5 0 0
5 3 0  5 5 5  5 5 5  5 5 5  5 5 5  5 3 0  5 5 5  5 0 5  5 5 5  5 5 5  5 0 5  5 5 5  5 0 5  5 5 5  \
0 5 0  0 5 0  0 5 0  0 5 0  0 5 0  5 5 5  5 5 5  5 5 5  5 0 0  5 0 0  5 0 0  5 0 0  5 5 5
```

**Figura 7.2**: ARSO.ppm.

Analizando el contenido del fichero se puede ver que los píxeles de la A están codificados con RGB = 530, lo que da un resultado naranja, los de la R con RGB = 505 (magenta), los de la S con RGB = 050 (verde), los de la O con RGB = 500 (rojo) y los píxeles blancos con RGB = 555. El símbolo «\» indica que la línea no acaba ahí, sino que continúa en la siguiente (en el fichero original no aparecen, cada una de las líneas de la imagen está representada en una sola línea de texto).

La única ventaja de estos dos formatos es que todo el contenido está codificado en texto plano (ASCII), lo que los hace adecuados para la comunicación entre sistemas que sólo están preparados para ese modo (como ocurría en los primeros tiempos del correo electrónico). Es obvio que se consigue una representación mucho más compacta codificando en binario. Los dos formatos son parte de una «familia» de formatos y programas de conversión de código abierto que se llama **Netpbm** y que incluye los resumidos en la tabla 7.1.

| Número mágico | Tipo | Extensión | bpp |
|---|---|---|---|
| P1 (0x5031) | bitmap en ASCII | `.pbm` | $8 \times 1$ |
| P2 (0x5032) | graymap en ASCII | `.pgm` | entre $8 \times 1$ y $8 \times 3$ |
| P3 (0x5033) | pixmap en ASCII | `.ppm` | entre $3 \times 8 \times 1$ y $3 \times 8 \times 3$ |
| P4 (0x5034) | bitmap binario | `.pbm` | 1 |
| P5 (0x5035) | graymap binario | `.pgm` | 8 |
| P6 (0x5036) | pixmap binario | `.ppm` | 24 |

**Tabla 7.1**: Formatos Netpbm.

Normalmente se aplican las versiones binarias. Por ejemplo, el programa `pngtopnm` convierte una imagen PNG a PBM si la imagen está en blanco y negro, a PGM si tiene grises y a PPM si tiene colores, pero siempre a los formatos binarios. Sin embargo, `pnmtopng` convierte de cualquiera de los formatos

de la tabla 7.1 a PNG («PNM», o *«portable anymap»*, es el nombre común en Netpbm para PBM, PGM y PPM).

Netpbm es especialmente útil para operaciones de conversión entre otros formatos, como un formato intermedio. No se suele utilizar como formato final para la distribución de imágenes, porque hay otros, como PNG y GIF, más eficientes (ocupan menos espacio). El paquete contiene más de 220 programas de conversión y de manipulación. Está incluido en todas las distribuciones de las distintas variantes de Unix y tiene versiones para los demás sistemas operativos.

Los formatos de imágenes matriciales más conocidos son BMP, TIFF, GIF, PNG, y JPEG. De los cuatro últimos ya hemos comentado algo sobre los algoritmos de compresión que utilizan (apartados 6.5 y 6.6) y sus números mágicos (apartado 7.1). Algunos otros aspectos generales son:

- **BMP** *(Bitmap)* es el formato matricial de los sistemas Windows. Los detalles de los convenios de almacenamiento dependen de la profundidad de color, que se indica en la cabecera: 1, 2, 4, 8, 16 o 24 bpp. Cuando $n \leq 8$ bpp incluye una tabla que define una paleta de $2^n$ colores. En estos casos la imagen puede estar comprimida con RLE o Huffman, pero normalmente los ficheros BMP no están comprimidos.

- **TIFF** *(Tagged Image File Format)* es el más conocido en el mundo profesional fotográfico. Se utiliza sin compresión, o con compresión sin pérdidas (LZW), para archivos de imágenes y para aplicaciones que requieren alta resolución y fidelidad, como las imágenes médicas.

- **GIF** *(Graphics Interchange Format)* es el formato más extendido en la web para imágenes sencillas. Su poca profundidad de color (8 bpp: 256 colores) lo hace poco adecuado para fotografías. Utiliza para la compresión LZW, que tenía el problema de estar patentado, pero la patente expiró en 2004. En un mismo fichero se pueden incluir varias imágenes con metadatos para presentarlas secuencialmente, lo que permite hacer animaciones sencillas («GIFs animados»).

- **PNG** *(Portable Network Graphics)* surgió como alternativa a GIF por el problema de la patente (utiliza Deflate en lugar de LZW, apartado 6.5) y tiene más opciones, como la posibilidad de añadir un nivel de transparencia a los píxeles («canal alpha»). El factor de compresión es algo mejor que con GIF, pero no siempre (depende del programa que genera las imágenes y de la imagen misma). No permite animaciones, aunque hay dos extensiones, MNG *(Multiple-image Network Graphics)* y APNG *(Animated Portable Network Graphics)*, poco utilizadas.

- **JPEG** *(Joint Photographic Experts Group)* es el formato más utilizado para la distribución de fotografías, por su flexibilidad y la eficiencia de los algoritmos de compresión (apartado 6.6). Las herramientas que generan o manipulan los ficheros con este formato suelen ofrecer un parámetro global de calidad, Q, cuyo valor puede elegirse entre 0 y 100. Para Q = 0 se obtiene la máxima compresión (mínimo tamaño, mínima calidad), y para Q = 100 la mínima (máximo tamaño, máxima calidad).

### Ficheros de imágenes vectoriales

En el apartado 5.3 vimos el principio de los formatos gráficos vectoriales, ilustrándolo con un ejemplo sencillo en el formato SVG (figura 5.6). Hay otros formatos (algunos muy extendidos, como PostScript y PDF, que permiten combinar gráficos vectoriales con imágenes matriciales, texto, etc.), pero, conocidos ya los algoritmos de compresión, es un buen momento para volver a comparar SVG con los formatos matriciales comprimidos.

Decíamos que los gráficos sencillos sencillos normalmente ocupan menos espacio en SVG que en formatos matriciales, aun comprimidos. Concretamente, con el programa `convert` del paquete «ImageMagick» se pueden generar a partir del fichero de la figura 5.6 otros con la misma imagen en otros formatos, y así comparar los tamaños. Estos son los resultados para GIF y PNG con resolución de 241×171:

```
$ls -l --sort=size bandera.*
-rw-r--r-- 1 gfer gfer 836 jul 21 20:26 bandera.gif
-rw-r--r-- 1 gfer gfer 663 jul 21 20:37 bandera.png
-rw-r--r-- 1 gfer gfer 486 jul 21 20:24 bandera.svg
```

Para imágenes no tan simples el código es, por supuesto, más largo, y el tamaño del fichero puede ser mayor que el matricial. La figura 7.3 procede de un fichero SVG que ocupa 16 KiB. Las conversiones a GIF y a PNG con resolución 341×378 y 16 bpp dan lugar a ficheros de 15 KiB y 60 KiB, respectivamente. Como se ve, PNG no siempre ocupa menos que GIF: depende de la imagen, los colores, etc. En el fichero SVG unos 2 KiB son de la cabecera con metadatos sobre el autor, la licencia (dominio público), etc.

**Figura 7.3**: Beastie (Logo de freeBSD).

El fichero SVG completo es demasiado grande para incluirlo aquí; la figura 7.4 muestra una pequeña parte: la que dibuja el cuerno derecho del demonio.

```
 ...
 <path d="M125.64,39.465
    c-3.15-3.979-5.49-6.699-5.22-11.58
    c1.3-11.46,21.03-20.449,18.58-26.39
    c-1.77-4.29-18.9,7.28-26.54,13.21
    c-21.54,16.73-39.66,35.03-23.63,62.37
    C92.14,66.315,111.47,37.095,125.64,39.465z"
    fill="#CB1009" stroke="#000000" stroke-width="0.5"/>
...
```

**Figura 7.4**: El cuerno derecho del demonio.

La sentencia «`<path d=.../>`» especifica las coordenadas de un punto de origen (125.64,39.465) seguidas de las coordenadas relativas de otros puntos que se unen cada uno al anterior mediante curvas de Bézier cúbicas («`c`») y termina con las mismas coordenadas del punto de origen, indicando el color de relleno y el grosor y el color del contorno.

Generalmente, el tamaño del fichero con la imagen vectorial es similar o menor que el del matricial, como se puede apreciar en los datos de los ejemplos anteriores. Pero al tratarse de texto plano se puede comprimir bastante. Veamos los resultados antes y después de aplicarles `gzip`:

```
$ls -l --sort=size freebsd-daemon.*
-rw-r--r-- 1 gfer gfer 61242 jul 26 10:57 freebsd-daemon.png
-rw-r--r-- 1 gfer gfer 16531 jul 26 00:09 freebsd-daemon.svg
-rw-r--r-- 1 gfer gfer 15780 jul 26 10:57 freebsd-daemon.gif
$gzip freebsd-daemon.*
$ls -l --sort=size freebsd-daemon.*
-rw-r--r-- 1 gfer gfer 55847 jul 26 10:57 freebsd-daemon.png.gz
-rw-r--r-- 1 gfer gfer 15239 jul 26 10:57 freebsd-daemon.gif.gz
-rw-r--r-- 1 gfer gfer  7055 jul 26 00:09 freebsd-daemon.svg.gz
```

Los ficheros GIF y PNG apenas se han reducido (los formatos originales ya están comprimidos), pero para el SVG ha resultado un factor de compresión aproximado de 2,34:1 ($\approx 43\,\%$).

La conclusión es que depende de la aplicación el que sea mejor trabajar con formatos y herramientas matriciales o vectoriales, o una combinación de ambos. Observe la palabra «herramientas». Como el formato vectorial es texto plano, puede componerse una imagen con un editor de textos normal. Así se ha hecho la imagen de la figura 5.6. Para dibujos no triviales sería demasiado laborioso, y es imprescindible utilizar un programa (herramienta). La imagen de la figura 7.3 (procedente del proyecto freeBSD) está compuesta con «Adobe Illustrator». Hay programas de dibujo de código abierto que tienen SVG como lenguaje «nativo», como Inkscape (`http://www.inkscape.org/`), o que pueden exportar los dibujos a SVG, como xfig (`http://www.xfig.org`).

Volviendo al formato del fichero SVG (no comprimido), como es una aplicación XML su contenido es texto plano y no tiene ninguna cabecera. La utilidad `file` lo reconoce como SVG:

```
$file freebsd-daemon.svg
freebsd-daemon.svg: SVG Scalable Vector Graphics image
```

Utiliza la «prueba de lenguaje» (apartado 7.1): al explorar el contenido encuentra en la primera línea «`<?xml...>`», que indica que utiliza el metalenguaje XML (apartado 3.8), y en la siguiente «`<!DOCTYPE svg...>`», que indica que el lenguaje concreto es SVG.

### Ficheros de vídeo

Algunos formatos para ficheros de vídeo se limitan a hacer una compresión intrafotograma (apartado 6.6), con la que consiguen factores de compresión comprendidos entre 5:1 y 10:1. Por ejemplo:

- **DV** *(Digital Video)* se propuso en 1996 por un consorcio de 60 fabricantes para los mercados industrial y doméstico, y para almacenamiento en cinta. Se ha ido modificando para otros medios y para videocámaras, con diversas variantes según los fabricantes.

- **MJPEG** *(Motion JPEG)* comprime cada fotograma en JPEG. No es un estándar, y no hay documentos de especificaciones. Se utiliza en aplicaciones de edición: al no haber compresión interfotograma se pueden manipular las imágenes individualmente con herramientas de JPEG.

Los formatos más conocidos para la transmisión de vídeo (y audio) y para su almacenamiento en ficheros son los que especifican las normas del MPEG (*Moving Picture Experts Group*, un grupo de trabajo de ISO e IEC), y sus derivados:

- **MPEG-1**. El vídeo se comprime con los procedimientos que mencionamos al final del apartado 6.6 para una calidad VHS con un factor de compresión 26:1 y una tasa de bits de 1,5 Mbps, y se utiliza, por ejemplo, en Video CD (VCD), un formato para almacenar vídeo en CD de audio. La parte de audio es la popularmente conocida como MP3.

- **MPEG-2** introduce algunas mejoras sobre MPEG-1 en vídeo, y, sobre todo, en audio, con AAC, ya comentado antes en los ficheros de sonidos. Es la base de otros estándares, como DVB-T *(Digital Video Broadcasting - Terrestrial)* (el utilizado para lo que en España se llama TDT) y DVD-Video.

- **MPEG-4**, que en principio estaba orientado a comunicaciones con ancho de banda limitado, se ha ido extendiendo a todo tipo de aplicaciones con anchos de banda desde pocos kbps a decenas de Mbps. Aparte de mejorar los algoritmos de compresión de MPEG-2 (se obtiene la misma calidad con la mitad de tasa de bits), incluye otras características, como **VRML** *(Virtual Reality Modeling Language)* que permite insertar objetos interactivos en 3D. La parte de vídeo, H.264/MPEG-4 Part 10, se conoce como **H.264** o **AVC** *(Advanced Video Coding)* y es actualmente uno de los formatos más utilizados para compresión, almacenamiento y distribución de vídeo de alta definición, por ejemplo, en HDTV, en los discos Blu-ray y en la web.

- **H.265**, o **HEVC** (High Efficiency Video Coding), sucesor de H.264, que duplica su tasa de compresión.

Hay varios formatos derivados de o relacionados con los estándares MPEG:

- **WMV** *(Windows Media Video)*, inicialmente basado en MPEG-4, evolucionó a un formato propietario que ha sido estandarizado por SMPTE *(Society of Motion Picture and Television Engineers)* con el nombre «VC-1». Se usa también en Blue-ray y en la Xbox 360.

- **DivX**, que tiene una historia algo tortuosa: su origen fue una copia (ilegal) de WMV llamada «DivX ;-)»; luego se comenzó a reelaborar completamente como software libre («Proyecto Mayo»), pero los líderes del proyecto terminaron creando una empresa, DivX, Inc., que comercializa códecs privativos. Paralelamente, algunos colaboradores del proyecto continuaron con la versión libre, que actualmente se llama «Xvid».

- **F4V**, que está basado en H.264, es el formato más reciente del muy extendido «Flash Player» (el formato FLV está basado en un estándar anterior, el H.263).

- **Theora**, desarrollado por la Fundación Xiph.org, es, a diferencia de los anteriores, libre: sus especificaciones son abiertas y los códecs son de código abierto. Desarrollado por Xiph.org Foundation. Es comparable en tecnología y eficiencia a MPEG-4 y WMV.

- **VP8**, creado por una empresa (On 2 Technologies) que fue comprada por Google y actualmente lo distribuye como software libre.

- **VP9**, que duplica la tasa de compresión de VP8.

## Contenedores multimedia

Un formato contenedor, o «envoltorio» *(wrapper)* es un conjunto de convenios para definir cómo se integran distintos tipos de datos y de metadatos en un solo fichero. Es especialmente útil en multimedia, para distribuir y almacenar conjuntamente ficheros de naturaleza diferente pero relacionados: varios canales de sonido, vídeos, subtítulos, menús de vídeo interactivos, etc.

Algunos de los formatos de vídeo mencionados antes, como FLV y F4V, son en realidad contenedores. En la tabla 7.2 se resumen algunas características de otros, incluyendo también las extensiones que suelen ponerse en los nombres de los ficheros.

| Contenedor | Origen | Comentarios | Ext. |
|---|---|---|---|
| QuickTime | Apple | Uno de los más antiguos. Puede contener diversos formatos de audio y de vídeo. MPEG-4 se basó en él. | .mov |
| MP4 (MPEG-4 Part 14) | MPEG | Contenedor oficial para los formatos de audio (AAC) y vídeo (AVC) de MPEG-4, pero admite otros formatos. | .mp4, .m4a, .m4v |
| 3GP (3rdGeneration Partnership Project) | 3GPP | Adaptación de MP4 para servicios multimedia UMTS. Vídeo AVC y varios formatos de audio. | .3gp |
| AVI (Audio Video Interleave) | Microsoft | Contenedor habitual de Windows desde 1992. Obsolescente. | .avi |
| ASF (Advanced Streaming Format) | Microsoft | Reemplazo de AVI. | .asf |
| Matroska | matroska.org | Formato universal para contener todo tipo de contenido multimedia. Especificaciones abiertas e implementaciones libres. | .mkv |
| Ogg | Xiph.org | Contenedor para los códecs de audio Vorbis y Opus y el de vídeo Theora. Todos ellos, software libre. | .ogg |
| WebM | Google | Basado en Matroska, con códecs Vorbis para audio y VP8/VP9 para vídeo, también es de código abierto. Orientado a la web (HTML5). | .webm |

**Tabla 7.2**: Algunos contenedores de multimedia

## Multimedia en la web

Habiendo mencionado varios formatos de audio, vídeo y contenedores, es interesante comentar, como nota marginal, una controversia actual sobre la distribución de contenido multimedia en la web. (Es «marginal» porque es efímera: en unos años la situación habrá cambiado).

El formato de vídeo más extendido en las páginas web es H.264/MPEG-4 AVC, que está sujeto a patentes. Por otra parte, la última versión del lenguaje HTML, HTML5, incluye elementos para insertar directamente objetos multimedia sin recurrir a complementos *(plugins)* privativos (como «Flash Player»). Las especificaciones iniciales recomendaban que todos los «agentes de usuario» (navegadores) deberían incluir soporte para un formato de vídeo libre de patentes. Concretamente, Theora para vídeo y Vorbis para audio, con Ogg como contenedor. Varias empresas se opusieron con el argumento de que esos formatos vulneraban patentes, y en diciembre de 2007 se eliminó ese requisito. La confusión aumentó al presentar Google el formato VP8 en 2010 (y su sucesor VP9 en 2013) y el contenedor WebM y su decisión (de momento no cumplida) de abandonar el soporte para H.264 en Chrome, puesto que las mismas empresas le acusaron de violar patentes.

Actualmente (2015) los navegadores Internet Explorer y Safari tienen soporte «nativo» (es decir, sin necesidad de instalar complementos) para H.264, pero no para Ogg ni WebM, mientras que en Chromium (versión libre de Chrome) es lo contrario. Firefox, Opera, Android browser y Chrome (de momento) aceptan los tres.

En cualquier caso, esta controversia no tiene prácticamente ninguna repercusión para el usuario final, puesto que los complementos, aunque privativos y sujetos a patentes, se distribuyen gratuitamente.

## 7.4. Ficheros ejecutables binarios

Los ficheros ejecutables contienen programas, es decir, órdenes, o instrucciones, para ser ejecutadas por un procesador, a diferencia de los ficheros estrictamente de datos, cuyo contenido «alimenta» a los programas.

Ahora bien, como veremos en el capítulo 12, hay varios tipos de lenguajes de programación. En algunos, esas «órdenes o instrucciones» se escriben en un formato de texto plano, para que un programa llamado **intérprete** las vaya leyendo *una tras otra* y generando, para cada una, las acciones adecuadas. Programas de ese tipo son los intérpretes de órdenes (o *shells*, apartado 2.4). Un fichero puede contener un programa formado una sucesión de tales órdenes (lo que se suele llamar un *«script»*), y podemos decir de él que es «ejecutable».

Pero cuando decimos «ejecutables binarios» nos referimos a otra cosa. El único lenguaje que «entiende» el procesador es un lenguaje binario, el **lenguaje de máquina**. Cada procesador tiene el suyo, definido por las operaciones básicas o **instrucciones** que, expresadas en binario, es capaz de ejecutar. Hay lenguajes de programación diseñados para escribir programas inteligibles para las personas que han de ser *traducidos* al lenguaje de máquina del procesador. Los programas escritos en estos lenguajes se llaman **programas fuente** y se guardan en ficheros de texto plano. A diferencia del intérprete, el programa **traductor** lee un programa fuente y *de una sola vez* genera el equivalente en el lenguaje de máquina del procesador. Este programa generado se llama **programa objeto**, o **código objeto**. Tres observaciones:

- «Binario» se refiere a «ejecutable», no a «fichero», aunque a veces se diga también «ficheros binarios» para referirse a los que no son de texto plano. Esta expresión es algo falaz, porque parece implicar que los ficheros de texto no son binarios, y, sin embargo, aunque su contenido se interprete como una sucesión de codificaciones de caracteres, siempre será binario.
- La distinción entre «programa» y «datos» es relativa: un programa fuente es, obviamente, un programa, pero para el programa traductor son datos de entrada.
- Hay una diferencia esencial entre este tipo de fichero (ejecutable binario) y los anteriores (documentos, multimedia, etc.), y es que su formato no puede independizarse del entorno de ejecución: está íntimamente ligado al procesador y al sistema operativo en los que va a ejecutarse.

El traductor guarda el código objeto en un fichero para que, en cualquier momento posterior, el sistema lo cargue en la memoria y se pueda ejecutar[3]. No intente usted leer (con `cat`, `less` etc.) un fichero de este tipo (ni de ningún tipo que no sea texto plano), porque no verá nada más que símbolos raros (y eventualmente perderá el control del terminal, al aparecer accidentalmente la codificación de algún carácter de control). Sí puede «verlo» con un programa como «`hd`» (o «`hexdump`»), que muestra en hexadecimal los contenidos binarios. Pruebe a hacerlo con cualquier fichero del directorio `/bin` de un sistema Unix. Por ejemplo, con la orden «`hd /bin/ls | less`». Verá cómo al principio aparece el número mágico (codificaciones ASCII de «.ELF»).

Como cada procesador tiene su lenguaje, el traductor tiene que estar adaptado al procesador concreto para el que traduce. El binario resultante, además del número mágico, tiene una cabecera que informa de qué procesador se trata, y un determinado formato. Y como los programas de aplicación hacen un uso sistemático de las llamadas al sistema (apartado 2.4) para acceder a los recursos de hardware, ese formato está especialmente ligado al sistema operativo. Cada uno tiene sus convenios. No tiene sentido entrar aquí en los detalles, pero sí citar los tres formatos más comunes actualmente:

[3] Como veremos en el capítulo 12, lo que genera el traductor no es directamente ejecutable, tiene que pasar por otro proceso, llamado «montaje».

- **PE** (Portable Executable) en los sistemas Windows
- **ELF** (Executable and Linkable Format) en los sistemas Unix.
- **Mach-O** (Mach Object) en los sistemas Darwin y Mac OS X.

### Binarios gordos y *blobs*

Cuando usted instala un programa nuevo en su ordenador tiene dos opciones:

- La del «experto»: si dispone del programa fuente, lo traduce (normalmente, con un compilador del lenguaje en el que está escrito el programa) y genera un código objeto adaptado a su ordenador y a su sistema operativo. Pero esta opción no siempre es posible, ni siendo experto, porque muchos programas son privativos, y no se dispone del código fuente.
- La «normal»: ejecuta un programa instalador proporcionado por el distribuidor del programa, que automáticamente extrae el código objeto ya adaptado a su sistema y lo guarda en un fichero, añadiendo datos al SGF para que pueda localizarlo.

La segunda opción implica que el distribuidor debe hacer versiones para los distintos procesadores y sistemas operativos. Un enfoque para reducir esta variedad es que el código objeto incluya versiones para varios procesadores. Naturalmente, tiene un tamaño mayor que el generado para un procesador concreto, y de ahí el nombre de *«fat binary»*, en el que «fat» tiene su significado literal en inglés (nada que ver con la tabla de asignación de ficheros, página 48). Dos ejemplos de este enfoque son:

- El formato *«Universal Binary»* de Apple, para programas que pueden ejecutarse tanto en los microprocesadores PowerPC como en los Intel (Apple cambió de unos a otros en 2005).
- El formato *«FatELF»* de Linux y otros Unix, que es una extensión del formato ELF.

«**Blob**» es un acrónimo de *binary large object* y en principio es un término ligado a las bases de datos. El FOLDOC lo define así:

> *Un gran bloque de datos almacenado en una base de datos, como un fichero de sonido o de imagen. Un BLOB no tiene una estructura que pueda ser interpretada por el sistema de gestión de la base de datos, éste lo conoce únicamente por su tamaño y localización.*

Pero el término se utiliza también con un sentido más general, para referirse a código binario cuyo programa fuente no está disponible. Este uso está ligado a la cultura del software libre. Por ejemplo, los núcleos de sistemas operativos como Linux, FreeBSD, NetBSD, etc., son «libres» (o de «código abierto») lo que significa, entre otras cosas, que todos los programas fuente se distribuyen libremente. El problema es que algunos fabricantes de controladores de periféricos (tarjetas gráficas, de red, etc.) facilitan el código binario de los gestores (*drivers*, apartado 2.5, página 52), pero no los fuentes ni la información técnica necesaria para poder programar esos fuentes. En algunas distribuciones de estos sistemas operativos se incluyen esos gestores en binario, y se dice de ellos (con un cierto toque despectivo) que son «blobs».

# Capítulo 8

# La máquina de von Neumann y su evolución

Del Tema 3 del programa de la asignatura, según la Guía de aprendizaje, deben obtenerse estos resultados:

- Conocer los principios básicos de la arquitectura de ordenadores.
- Comprender el funcionamiento de los procesadores en el nivel de máquina convencional

Ya en el capítulo 1 hemos analizado los distintos significados de la expresión «arquitectura de ordenadores» (apartado 1.7) y hemos situado el «nivel de máquina convencional» en la jerarquía de niveles de abstracción (apartado 1.6). En este capítulo veremos cómo ha evolucionado la arquitectura de los procesadores hardware desde sus orígenes, resumiendo los avances más importantes e insistiendo en algunos aspectos que ya introdujimos en el capítulo 2. En los dos siguientes estudiaremos detalladamente un subconjunto de la arquitectura de un procesador hardware moderno, y en el capítulo 11 resumiremos los conceptos más importantes de las arquitecturas paralelas y de una clase importante de procesadores especializados: los procesadores gráficos.

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 tienen que estar guardadas, o *almacenadas*, en una memoria de acceso aleatorio o, lo que es lo mismo, 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, aunque al hacerlo se repitan algunos conceptos que ya hemos adelantado en el capítulo 2.

## 8.1. Modelo estructural

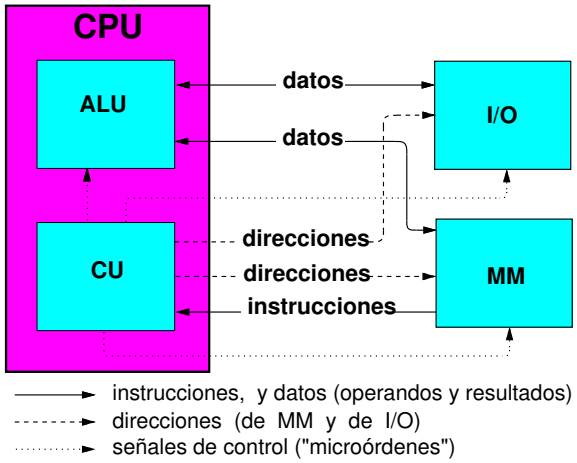



**Figura 8.1**: La «máquina de von Neumann».

*«... 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...»*

Aquí el documento introduce el *modelo estructural básico* en el nivel de máquina convencional. Los «órganos» son los subsistemas llamados «ALU» *(Arithmetic and Logic Unit)*, «MM» *(Main Memory)*, «CU» *(Control Unit)* e «I/O» *(Input/Output)* en el diagrama de la figura 8.1. El conjunto de la CU y la ALU es la «CPU» *(Central Processing Unit)*. Como hemos dicho en el apartado 1.5, 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 sólo 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 memoria, 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». El hilo es una reliquia histórica; las memorias secundarias actuales son preferentemente discos, o de estado sólido, y en ciertas aplicaciones, cintas (apartado 2.3).

> *«... 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 $\mu$s 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** que ya conocemos (página 27).

La figura 8.2 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 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 (salvo en las ROM y EEPROM, nota 3, página 6).

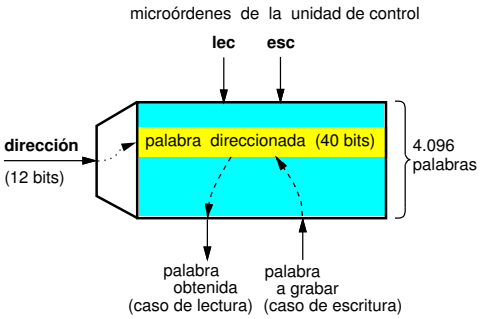



**Figura 8.2**: Lectura y escritura en la MM.

Dos propiedades importantes de la memoria 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 posición; la escritura sí lo es: el contenido anterior desaparece y queda sustituido por el nuevo.

## 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 ALU 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).

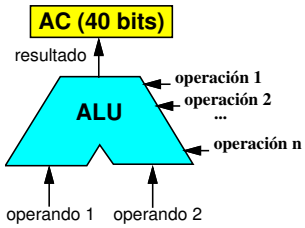



**Figura 8.3**: Diagrama de la ALU.

En resumen, la unidad aritmética y lógica, ALU, es un subsistema que puede tomar dos operandos (o sólo 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 ALU por el diagrama de la figura 8.3. El resultado, en este modelo primitivo, se introducía en un **registro acumulador** (AC). Se trataba de una memoria de acceso mucho más rápido que la memoria, pero que sólo podía 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** o **dispositivos**), representadas en la figura 8.1 como un simple bloque (I/O), en aquella época eran muy básicas: simples terminales electromecánicos. Actualmente son muchas y variadas (tabla 2.2, página 28).

## Unidad de control

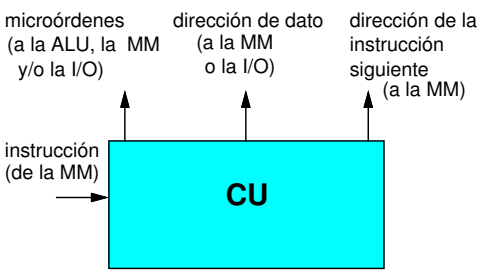



**Figura 8.4**: Unidad de control.

*«... 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 ...»*

La unidad de control 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 memoria o de entrada/salida oportunas (figura 8.4).

> *«... 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).

> *«... Debe ser posible transferir datos de la memoria al órgano aritmético, y viceversa. En la transferencia de la información desde el órgano aritmético a la memoria hay que distinguir dos tipos: transferencias de números como tales y transferencias de números que forman parte de órdenes. El primer caso es bastante obvio y no necesita mayor explicación. El segundo es más sutil y sirve para explicar la generalidad y sencillez del sistema. Consideremos, a modo de ilustración, el problema de la interpolación. Supongamos que hemos formulado las instrucciones necesarias para realizar una interpolación de orden* n *en una secuencia de datos. La posición exacta dentro de la memoria de las* (n+1) *cantidades que pueden corresponder al valor funcional deseado es función de un argumento. Y este argumento es, probablemente, el resultado de una computación en la máquina. Así pues, necesitamos una orden que pueda sustituir un número dentro de una determinada orden – en el caso de la interpolación, la posición del argumento o del grupo de argumentos más próximo en nuestra tabla al valor deseado –. Por medio de tal orden, los resultados de un cálculo se pueden introducir en las instrucciones que realizan esa u otra computación diferente. Esto hace posible que una secuencia de instrucciones se pueda utilizar con diferentes conjuntos de números localizados en diferentes partes de la memoria...»*

> *«... En resumen, las transferencias a la memoria serán de dos tipos:* sustituciones totales*, con las que la cantidad previamente almacenada queda borrada y sustituida por un nuevo número, y* sustituciones parciales *con las que la parte de una orden que contiene un* número de posición de memoria *– suponemos que las distintas posiciones de memoria están numeradas consecutivamente por números de posición de memoria – queda sustituida por un nuevo número de posición de memoria...»*

Estos dos párrafos aluden a una técnica a la que había que recurrir en las primeras máquinas para recorrer zonas de memoria: la *modificación de instrucciones*. Posteriormente se introdujeron otros elementos de hardware, los *registros de índice*, con los que se resuelve el mismo problema de forma más segura y cómoda, pero la posibilidad de modificar las instrucciones de un programa en el curso de su ejecución es consecuencia de la misma idea de programa almacenado.

Los «números de posiciones de memoria» son lo que ahora llamamos *direcciones*, definidas más arriba al hablar de la memoria.

## 8.2. 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 memoria la instrucción e introducirla en la unidad de control.
2. En la fase de **decodificación** la unidad de control analiza los bits que la componen y «entiende» lo que «pide» la instrucción.
3. En la fase de **ejecución** la unidad de control genera las microórdenes para que las demás unidades completen la operación. Esta fase puede requerir un nuevo acceso a la memoria para extraer un dato o para escribirlo.

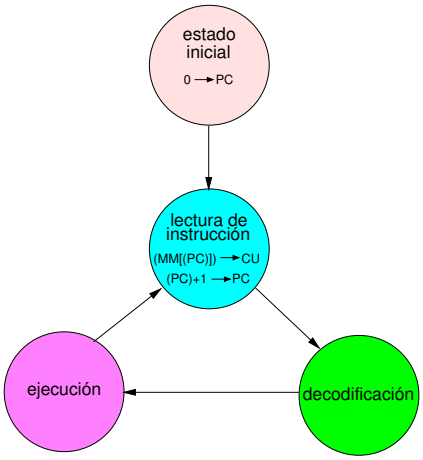



**Figura 8.5**: Modelo procesal.

Concretando un poco más, la figura 8.5 presenta gráficamente el modelo procesal. «PC» es el **contador de programa** *(Program Counter)*, un registro de doce bits dentro de la unidad de control que contiene la dirección de la instrucción que se lee, e inmediatamente se incrementa para apuntar a la instrucción siguiente. Hay un estado inicial en el que se introduce en PC la dirección de la primera instrucción: la expresión «0 → PC» significa «introducir el valor 0 (doce ceros en binario) en el registro PC» (se supone que cuando arranca la máquina el programa ya está cargado a partir de la dirección 0 de la memoria). Después se pasa cíclicamente por los tres estados: lectura, decodificación y ejecución. En la lectura de instrucción «(MM[(PC)]) → CU» significa «transferir la palabra de dirección indicada por PC a la unidad de control», y «(PC)+1 → PC», «incrementar en una unidad el contenido del contador de programa».

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 por la propia instrucción. En la fase de ejecución de estas instrucciones se introduce un nuevo valor en el registro PC.

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 sólo hacía falta una lectura para ejecutar dos instrucciones seguidas.

## 8.3. 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)...»*

O sea, el **formato de instrucciones** propuesto es el que indica la figura 8.6, con dos instrucciones en cada palabra. En muchos diseños posteriores (CISC, ver apartado 8.7) 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.

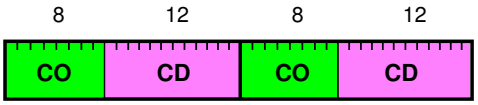



**Figura 8.6**: Formato de instrucciones.

En el formato de la figura 8.6 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 memoria a la que **hace referencia** la instrucción.

Por ejemplo, tras leer y decodificar 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 memoria, luego llevaría este contenido a una de las entradas de la ALU y el contenido actual del acumulador a la otra, generaría la microorden de suma para la ALU 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...»*
>
> *«... Frecuentemente, dos secuencias distintas de órdenes terminan en una rutina común. Por tanto, es preciso indicar al control que en ambos casos siga desde el punto de comienzo de la rutina común. Esta transferencia incondicionada puede llevarse a cabo bien por el empleo artificial de una transferencia condicionada o bien mediante la introducción de una orden explícita para tal transferencia...»*

Es decir, los programas constan de una secuencia de instrucciones que se almacenan en direcciones consecutivas de la memoria. 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). Las bifurcaciones pueden ser incondicionadas o condicionadas a algún resultado previo.

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 9, ya con un procesador concreto y moderno.

## 8.4. 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 (figura 1.4, página 10), que usted debe conocer por la asignatura «Fundamentos de Electrónica». Pero también en este nivel de máquina convencional se han ido produciendo innovaciones. Veamos, de manera general, las más importantes.

## 8.5. Modelos estructurales

La figura 8.7 es un modelo estructural de un sistema monoprocesador muy básico, pero más actual que el de la figura 8.1. Se observan varias diferencias:

- En lugar de un solo registro acumulador (figura 8.3), 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 memoria: 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, SR** *(Status Register)*, que, entre otras cosas, contiene los indicadores Z, N, C y V, cuya función hemos visto en el apartado 4.4, y un bit que indica el **modo** en que se encuentra el procesador: usuario o supervisor (página 22).

[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 $2.300 \times 2^{20} \approx 2.400$ millones. El Intel Xeon Westmere-EX de 10 núcleos tiene más de 2.500 millones.

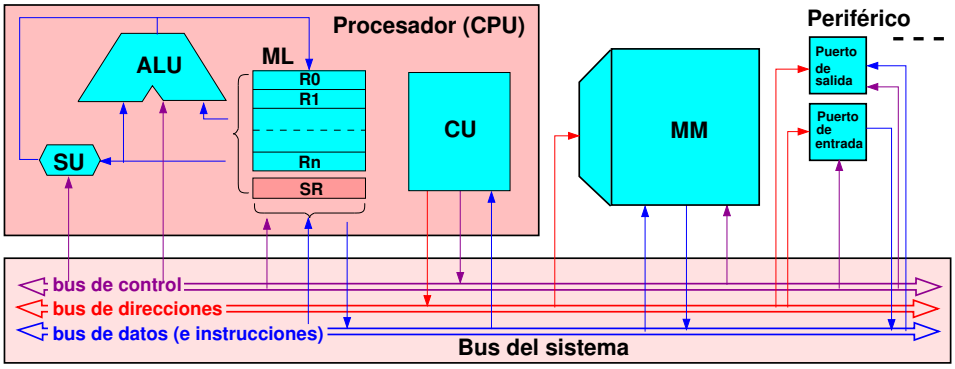



**Figura 8.7**: Modelo estructural de un sistema basado en procesador

- Hay un nuevo componente, la **unidad de desplazamiento, SU** *(Shift Unit)*, que permite realizar sobre cualquier registro las operaciones de desplazamientos y rotaciones que también hemos estudiado en el apartado 4.4.
- Los datos y las instrucciones se transfieren de una unidad a otra a través de buses. Ya hemos adelantado en el apartado 2.3 (página 28) lo que es un bus. En este caso hay un bus único.
- Los periféricos se conectan al bus a través de registros incluidos en los controladores, que se llaman **puertos**.

Sólo es cuestión de convenio considerar que hay un único bus, el **bus del sistema**, formado por tres subuses, o que realmente hay tres buses:

- El **bus de direcciones** transporta direcciones de la memoria, o de los puertos, generadas por la unidad de control. Su ancho está relacionado con la capacidad de direccionamiento: si tiene $n$ líneas puede transportar $2^n$ direcciones diferentes.
- Por el **bus de datos** viajan datos leídos de o a escribir en la memoria y los puertos, y también las instrucciones que se leen de la memoria y van a la unidad de control.
- El **bus de control** engloba a las microórdenes que genera la unidad de control y las que proceden de peticiones que hacen otras unidades a la unidad de control. 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 8.7 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 8.2) es un caso especial: en algunas arquitecturas es también transparente, pero en otras (como la que estudiaremos en el capítulo 9) es uno de los registros de la memoria local.

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. Normalmente se utiliza una jerarquía de buses, como ya vimos en la figura 2.8.

## 8.6. 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 *cache*.

## 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.

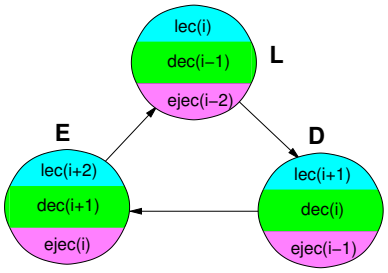



**Figura 8.8**: Modelo procesal con encadenamiento.

El modelo procesal de la figura 8.5 supone que hay tres fases, o etapas, que se llevan a cabo una detrás de otra, y que en cada una se realiza *una sola operación sobre una sola instrucción*. En la primera se hace uso de la memoria, pero no así en la segunda ni (salvo excepciones) en la tercera. En la segunda se decodifica y se usa la memoria local para leer registros, y en la tercera se usa la ALU y la memoria local para escribir. Suponiendo que en la memoria local se pueden leer dos registros y escribir en otro al mismo tiempo, en cada fase se usan recursos distintos, por lo que pueden superponerse, resultando el modelo procesal de la figura 8.8: en la etapa **L** se lee la instrucción **i**, pero al mismo tiempo se decodifica la anterior (**i-1**) y se ejecuta la anterior a la anterior (**i-2**); en la fase **D**, al mismo tiempo que se decodifica la **i** se lee la siguiente (**i+1**) y se ejecuta la **i-1**, etc.

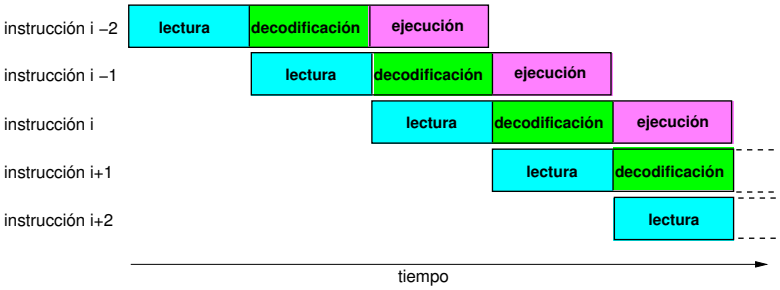



**Figura 8.9**: Ejecución de instrucciones en una cadena de tres etapas.

El resultado, por lo que se refiere al tiempo de ejecución de una secuencia de instrucciones, es el que ilustra la figura 8.9. Con respecto a un modelo procesal estrictamente secuencial, la velocidad de ejecución se multiplica (teóricamente) por tres.

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.

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

- Si la instrucción opera con un dato de la memoria, la fase de ejecución no puede solaparse con la de lectura (a menos que la memoria tenga dos vías de acceso, una para instrucciones y otra para datos). Es un ejemplo de un *conflicto estructural*, que ocurre cuando dos o más etapas de la cadena necesitan un recurso que no puede compartirse.
- Si se trata de una instrucción de bifurcación, esto no se sabe hasta la etapa de decodificación, cuando ya se ha leído, inútilmente, la instrucción que está en la dirección siguiente. Es un *conflicto de control*.
- En el caso de la cadena de tres etapas que estamos utilizando como ejemplo no ocurre, pero en cadenas de mayor profundidad aparecen *conflictos de datos* cuando una instrucción opera sobre el resultado de otra previa que aún está en la fase de ejecución.

Estos conflictos se resuelven de diversas maneras en el nivel de micromáquina. Volveremos sobre ellos en el apartado 11.1.

### *Caches*

El concepto de memoria *cache* ya lo hemos introducido en el capítulo 2 (página 24). Vimos también que la misma idea se aplica, implementada mediante software, para mejorar los tiempos de respuesta en los accesos a disco (página 27). E incluso en aplicaciones: la *cache* de un navegador web es una zona de la memoria en la que se guardan las páginas más visitadas.

Aquí nos referimos a la cache de memoria principal, implementada en hardware, de la que vamos a ampliar algunos detalles relacionados con el encadenamiento, en particular, con los conflictos estructurales.

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 memoria. Es lo que en la figura 2.6 llamábamos «brecha de velocidades». Como toda instrucción tiene que ser leída de la memoria, 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 8.9 para el caso de que la etapa de lectura tenga una duración cien veces superior a las de decodificación y ejecución, la velocidad del procesador, medida en instrucciones por segundo, no se multiplicaría por tres, sino por un número ligeramente superior a uno[2].

Como ya dijimos, la *cache* es una RAM de capacidad y tiempo de acceso muy pequeños con relación a los de la memoria (pero no tan pequeños como los de la memoria local incluida en el procesador). En la *cache* se mantiene una *copia* de *parte del contenido* de la memoria. Siempre que el procesador hace una petición de acceso a una dirección de memoria primero se mira en la *cache* 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 memoria. 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 memoria y un tiempo de acceso *medio* que, idealmente, debe acercarse al de la *cache*.

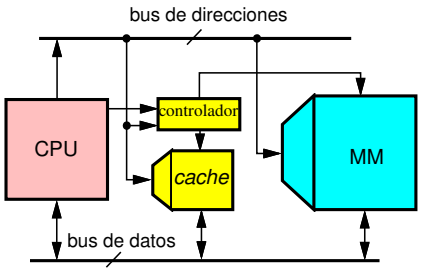



**Figura 8.10**: Memoria *cache*.

El buen funcionamiento del sistema de memoria construido con *cache* 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 memoria a las que se accede durante un período de tiempo en el proceso suelen estar localizadas en zonas pequeñas de la memoria. 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 *cache* 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 *cache*: 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 memoria, se direcciona en la L1, y generalmente resulta un acierto, pero si hay fracaso, el controlador accede a la L2, etc.

En las CPU 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

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

Harvard, apartado 8.1): L1d es la *cache* de datos y L1i es la *cache* de instrucciones (figura 2.8). De esta manera, es posible que la lectura de la instrucción **i** sea simultánea con la lectura o escritura de un dato en la ejecución de la instrucción **i-2**, reduciéndose así los conflictos estructurales 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[3], cada uno de ellos acompañado de una *cache* de datos y una *cache* de instrucciones, de 32 KiB cada una, y que hay una *cache* 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, las *caches* son transparentes en el nivel de máquina convencional. Es decir, las instrucciones de máquina hacen referencia a direcciones de la memoria, y el programador que las utiliza no necesita ni siquiera saber de la existencia de tales *caches*. 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 memoria, y la única consecuencia «visible»[4] será que la ejecución de esa instrucción tardará más tiempo.

## 8.7. 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 1.7) 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 de un procesador concreto.

### Direcciones y contenidos de la memoria

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 memoria 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 (capítulo 3) 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**.

[3] A los procesadores, o «cores», el programa `lscpu` les llama «CPU». Aquí llamamos CPU al conjunto formado por los cuatro.

[4] 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.

En todas las arquitecturas modernas la memoria 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[5]. 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 memoria, 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 en el 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 8.11(a)), etc. En este segundo caso, se dice que las direcciones de palabra deben estar **alineadas**.

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 dieciséis líneas el espacio es de $2^{16} = 64$ KiB, etc.

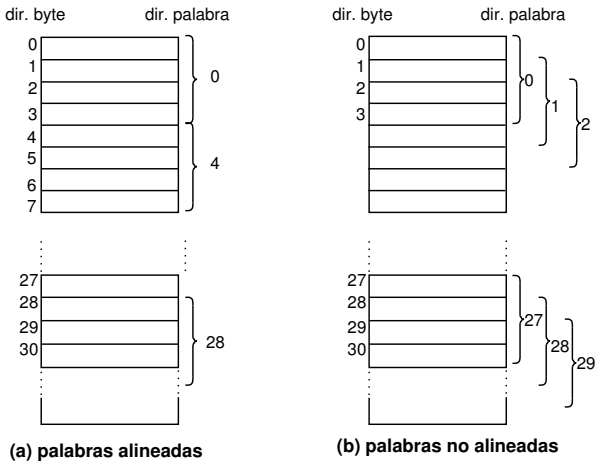



**Figura 8.11**: Direcciones alineadas y no alineadas en una arquitectura de 32 bits.

Observe que hablamos de «espacio de direccionamiento», no de «capacidad de la memoria». La memoria no necesariamente alcanza la capacidad que es capaz de direccionar el procesador. Si las direcciones son de 32 bits, el espacio de direccionamiento es $2^{32} = 4$ GiB, pero la capacidad real de la memoria puede ser menor. Como hemos visto en el capítulo 2, la **memoria virtual** permite que que los programas «vean» ese espacio de direccionamiento como si realmente tuviesen disponible una memoria de esa capacidad, ayudándose de memoria secundaria y de una combinación de hardware (la unidad de gestión de memoria, MMU) y software (el sistema de gestión de memoria, página 45).

## 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

[5] 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 unidad de control 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.

cantidad (varias centenas) como en modos de acceder a la memoria. Esto obedecía a razones prácticas[6]: cuantas más posibilidades ofrezca el nivel de máquina convencional más fácil será desarrollar niveles 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», figura 1.4). Como esta técnica representaba algo intermedio entre el hardware y el software, se inventó una palabra nueva: **firmware**[7].

Ahora bien, en los años 1970 y 1980 se hicieron muchos estudios sobre estadísticas de uso de instrucciones durante la ejecución de programas reales. Estos estudios demostraban que gran parte de las instrucciones apenas se utilizaban. En media, resultaba que alrededor del 80 % de las operaciones se realizaba con sólo 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 6, 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». Lo cual nos conduce a recordar una alternativa de diseño que ya conoce usted de la asignatura «Sistemas Electrónicos»: RISC vs. CISC.

## CISC, RISC y MISC

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 memoria 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.

[6] Se ha demostrado teóricamente que un procesador con sólo 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 memoria infinita, tendría la misma potencia computacional que una máquina de Turing).

[7] Actualmente se sigue utilizando, pero dado que los microprogramas quedan integrados en el chip, sólo accesibles al fabricante, el significado de «firmware» ha derivado hacia otra cosa: el software grabado en memorias ROM o *flash*.

La mayoría de los procesadores RISC actuales tienen más de cien instrucciones. Algunos vuelven a la idea original y tienen un número mucho menor. Normalmente son máquinas de pilas (apartado 8.8), diseñadas para implementarse con (relativamente) pocos transistores y tener un consumo muy reducido, pensando en su uso para dispositivos embebidos (apartado 1.5). Se llaman «**MISC**» *(Minimal Instruction Set Chip)*.

## Modos de direccionamiento

En la máquina de von Neumann, las instrucciones que hacen referencia a memoria (almacenar el acumulador en una dirección de la memoria, sumar al acumulador el contenido de una dirección, bifurcar a una dirección...) indican la dirección de memoria *directamente* en el campo CD. Pero normalmente 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. Tres de estos **modos de direccionamiento** son:

- El **direccionamiento indexado**, en el que el contenido del campo CD se suma con el contenido de un **registro de índice** para obtener la dirección efectiva.
- El **direccionamiento indirecto** consiste, en principio, en interpretar el contenido de CD como la dirección de memoria 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 memoria, puesto que primero hay que leer la palabra que contiene la dirección efectiva. A esta palabra se le llama **puntero** (página 6): *apunta a* el operando o a la instrucción siguiente. Como normalmente el procesador tiene una memoria local, es más eficiente que el puntero sea un registro. En tal caso, la instrucción no tiene campo CD, sólo 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 los modos indirecto y relativo: sumar los cien elementos de un vector que están almacenados en cien bytes consecutivos de la memoria, 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 memoria *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 10:

```
        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 8.9), 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 sólo 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 software) 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 haría que el programa sólo funcionase cargándolo en la memoria a partir de la dirección $I$. Sin embargo, con el direccionamiento relativo, se puede 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 unidad de control 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 memoria 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 memoria 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 deben cumplirse dos condiciones:

- **Pasar los parámetros** (o «argumentos»): antes de saltar a esa secuencia (a la que en adelante llamaremos **subprograma**) hay que indicarle los datos sobre los que ha de operar («parámetros de entrada»), y después de su ejecución es posible que el subprograma tenga que 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.

El mapa de memoria de la figura 8.12 corresponde a una situación en la que hay un programa cargado a partir de la dirección $p$ 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 («retorno») para volver a la instrucción siguiente a la de llamada, que es distinta cada vez.

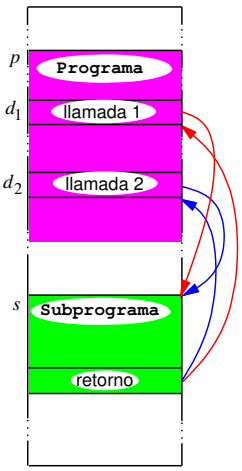



**Figura 8.12**: Llamadas y retornos.

Para el paso de parámetros, lo más sencillo es colocarlos en registros antes de la llamada, de donde los puede recoger el subprograma (otra forma es mediante la pila, como veremos en el siguiente apartado). 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 8.12 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 memoria *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 8.3: hacen que la instrucción siguiente a ejecutar no sea la que está en la dirección ya preparada en el contador de programa (PC), 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 PC 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?

La mayoría de los procesadores tienen un registro especial, llamado **registro de enlace**. La instrucción de llamada, antes de poner el nuevo valor en PC, 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 PC. 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.

## 8.8. Pilas y máquinas de pilas

Una **pila** es un tipo de memoria en la que sólo 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 8.13. Por tanto, *en una memoria de pila no tiene sentido hablar de «direcciones»*. La primera posición es el *fondo* de la pila. Si la pila está vacía, no hay cima; si sólo contiene un elemento, la cima coincide con el fondo.

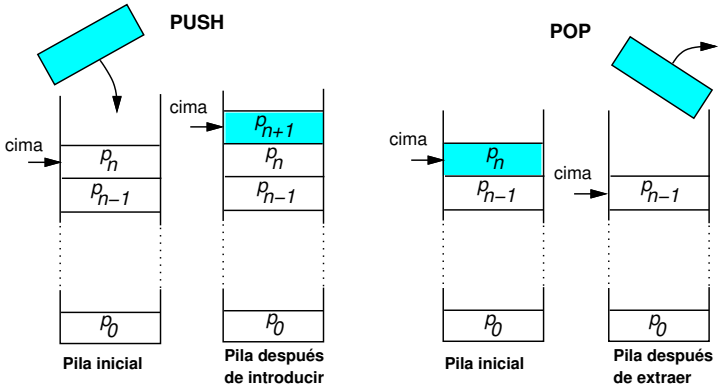



**Figura 8.13**: Memoria de pila.

Este tipo de almacenamiento, también conocido como «memoria LIFO» *(Last-In First-Out)*, se puede simular con una memoria de acceso aleatorio, y, como vamos a ver, es la forma más utilizada en los procesadores para conseguir 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).

## Pilas en la RAM

Todos los procesadores hardware tienen un mecanismo para simular una pila en una parte de la memoria, que consiste en lo siguiente:

En el procesador hay un registro especial llamado **puntero de pila, SP** *(Stack Pointer)*, que contiene en todo momento la primera dirección de la memoria libre por encima de la pila (figura 8.14) (o bien la dirección de la cima). En la instrucción de llamada a subprograma, la unidad de control 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 SP y luego se decrementa el contenido de SP en tantas unidades como bytes ocupe una dirección (si el convenio es que SP apunta a la cima, se invierten los pasos).

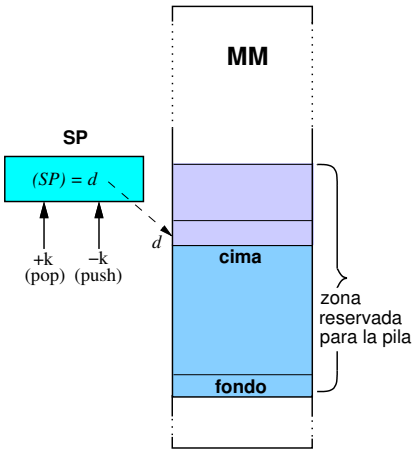



**Figura 8.14**: Una pila en una RAM.

La ejecución de la instrucción de retorno se hace con un *pop* para extraer de la pila la dirección de retorno y ponerla en el contador de programa. La operación *pop* consiste en incrementar el contenido de SP y luego leer de la dirección apuntada por SP (si el convenio es que SP apunta a la cima, se invierten los pasos).

Repare en tres detalles de esta solución:

- Las operaciones *push* y *pop* mencionadas pueden ser 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 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. Para ello, el procesador dispone de instrucciones específicas para introducir o extraer de la pila cualquier registro. Por ejemplo, el efecto de ejecutar `PUSH R5` sería que el contenido del registro R5 se introduce en la pila y el contenido de SP se decrementa en tantas unidades como bytes tenga R5; el de `POP R5`, que el contenido de la cima se copia en R5 y SP se incrementa. Antes de `CALL` (o `PUSH PC` seguido de bifurcación, que es lo mismo) el programa utiliza instrucciones `PUSH` para introducir los parámetros en la pila, y el subprograma puede acceder a ellos con un modo de direccionamiento que sume una distancia al puntero de pila. Algunos procesadores disponen de una instrucción que permite guardar en la pila varios registros. Al espacio ocupado por los parámetros y la dirección de retorno en un subprograma se le llama **marco de pila**.

## Máquinas de pilas

Recuerde que los RISC se caracterizan, entre otras cosas por tener una arquitectura «*load/store*»: todas las operaciones aritméticas y lógicas se realizan en registros del procesador. Son **máquinas de registros**. En las máquinas de pilas no hay registros visibles al programador, y esas operaciones se realizan sobre operandos que están en una memoria de pila implementada con hardware dentro del procesador.

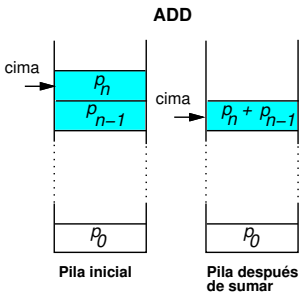



**Figura 8.15**: Suma en una pila.

Por ejemplo, la instrucción «sumar» (`ADD`) en una máquina de pila 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 (figura 8.15).

Esta arquitectura tiene cierta ventaja a la hora de evaluar eficientemente expresiones aritméticas. Con un ejemplo sencillo se entenderá: la evaluación de la expresión $5 \times 3 + 12/2$ con una máquina de pila que tiene instrucciones `PUSH`, `ADD`, `MUL` y `DIV` se puede hacer con esta secuencia:

```
PUSH  #5
PUSH  #3
MUL
PUSH  #12
PUSH  #2
DIV
ADD
```

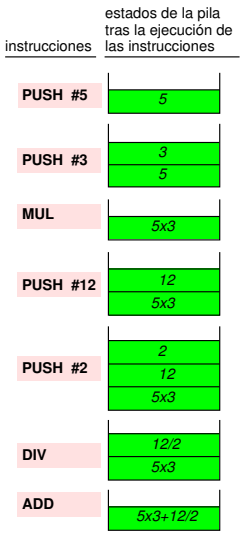



**Figura 8.16**: Ejecución de operaciones en una pila.

En la figura 8.16 puede seguirse, paso a paso, el estado de la pila (suponiéndola inicialmente vacía) a medida que se ejecutan las instrucciones.

Las operaciones para evaluar expresiones aritméticas en una pila se corresponden, formalmente, con la llamada **notación polaca inversa**. La notación aritmética habitual es la infija: el operador se escribe entre los dos operandos, y, así, para expresar la suma de $A$ y $B$ escribimos «$A + B$». En la notación polaca el operador es prefijo («$+AB$»), y en la notación polaca inversa, sufijo («$AB+$»). En ambas notaciones puede prescindirse del uso de paréntesis, que en la notación habitual son necesarios para eliminar ambigüedades. Por ejemplo, las expresiones «$(A + B) \times C$» y «$A + (B \times C)$» se escriben en notación polaca inversa «$AB + C\times$» y «$ABC \times +$» respectivamente.

El procedimiento para interpretar una expresión aritmética en notación polaca inversa es:

- explorar la cadena de izquierda a derecha;
- al encontrar un operando, apilarlo;
- al encontrar un operador, realizar la operación con los dos operandos inmediatamente anteriores; eliminar esos dos operandos y el operador e introducir en su lugar el resultado;
- seguir explorando la cadena hasta que no queden operandos.

Si aplicamos este procedimiento al caso «$5 \times 3 + 12/2$», que, escrito en notación polaca inversa es «$5 \cdot 3 \times 12 \cdot 2/+$» (el punto entre operandos sólo tiene la función de separador), el resultado es:

1. apilar 5: `PUSH #5`;
2. apilar 3: `PUSH #3`;
3. ahora aparece «×»: `MUL` (elimina 5 y 3 y apila $5 \times 3$));
4. apilar 12: `PUSH #12`;
5. apilar 2: `PUSH #2`;
6. aparece «/»: `DIV` (elimina 12 y 2 y apila 12/2);
7. aparece «+»: `ADD` (elimina 12/2 y $5 \times 3$ y apila la suma);

Es decir, la misma secuencia de instrucciones escrita anteriormente.

En lugar de una memoria local con registros accesibles individualmente, como tienen las máquinas de registros (fig 8.7), las máquinas de pilas suelen tener una o varias pilas. Por ejemplo, la figura 8.17 muestra que puede haber una pila de datos en la que se apilan los marcos (que también son pilas) de los subprogramas que se llaman sucesivamente y otra pila para las direcciones de retorno. Cada marco contiene una pila para los parámetros y las variables locales del correspondiente subprograma. Algunos procesadores hardware siguen esta arquitectura, pero su uso más extendido es como modelo para *máquinas virtuales*, concepto que veremos en el apartado 12.6. En el capítulo 13 estudiaremos la arquitectura de una máquina de pilas muy conocida: la JVM (Máquina Virtual Java).

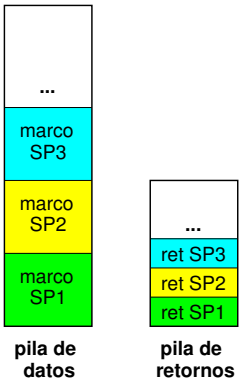



**Figura 8.17**: Pilas en una máquina de pilas.

## 8.9. 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 cada periférico, y, 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», página 32):

- **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 juegan un papel importante las interrupciones.

### Interrupciones

Como dijimos en el tema de sistemas operativos (página 59), todos los procesadores hardware de uso general actuales tienen un mecanismo de interrupciones. Este mecanismo permite intercambiar datos entre el procesador y los periféricos, implementar llamadas al sistema operativo (mediante instrucciones de interrupción) 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 cuidadosamente 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 memoria.
- 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. Es lo que en el apartado 2.6 (página 59) hemos llamado **cambio de contexto**.
- En muchos procesadores estos contenidos se guardan en la pila. En el que estudiaremos en el siguiente capítulo se guardan en registros dedicados para esa función.
- 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.

## 8.10. Conclusión

En este capítulo hemos visto los conceptos más importantes del nivel de máquina convencional que se implementan en los procesadores hardware actuales. 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.

Hay, además, otra evolución importante en el nivel de máquina convencional que hasta ahora no hemos mencionado: las arquitecturas que integran a varios procesadores hardware funcionando simultáneamente («en paralelo»). Estudiaremos algunas en el capítulo 11.

# Capítulo 9

# 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, 8,7 millardos en 2012 y 10 millardos en 2013[1]. Estos procesadores se encuentran, sobre todo, en dispositivos móviles (el 95 % de los teléfonos móviles contienen al menos un SoC basado en ARM), pero también en controladores de discos, en televisores, en reproductores de música, en lectores de tarjetas, en automóviles, 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 esta asignatura, 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 y con un sistema basado en ARM («raspberry pi»).

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 1.6). 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, sólo 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_FullReport_2013.pdf`
`http://ir.arm.com`

## 9.1. Modelo estructural

La figura 9.1 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 memoria 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]

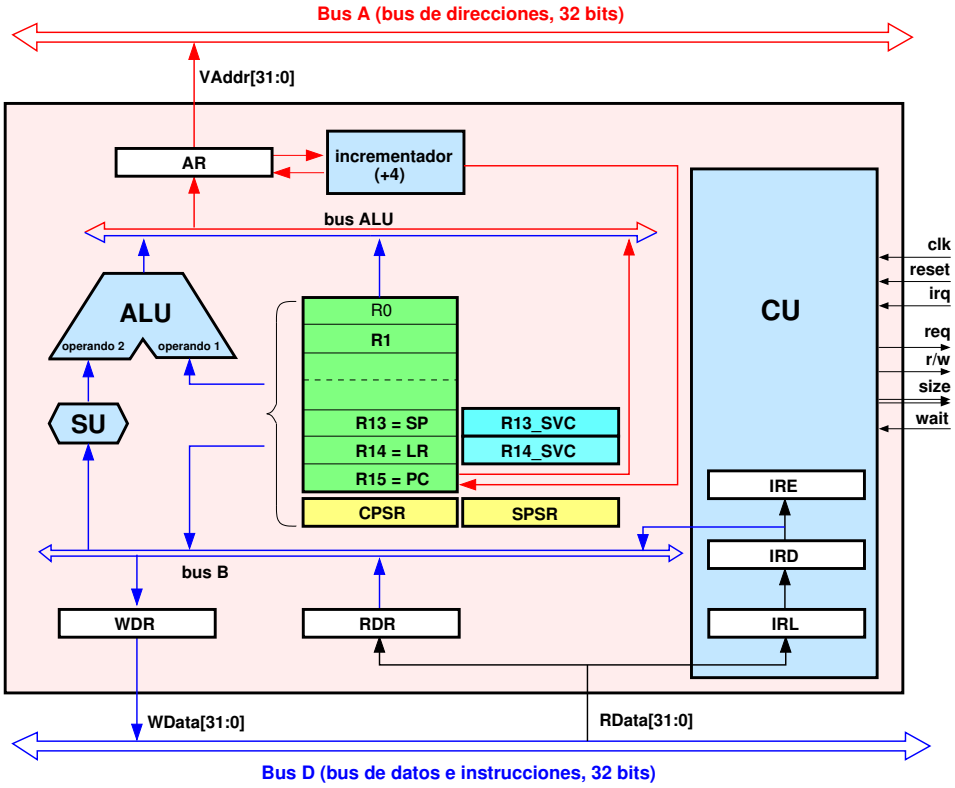



**Figura 9.1**: Modelo estructural de BRM.

Ya sabemos lo que son la ALU, la SU y la CU (no se han dibujado las microórdenes que genera ésta para controlar a las demás unidades, lo que complicaría innecesariamente el diagrama), que tienen las mismas funciones que en la figura 8.7 (página 135).

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.

[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.

### Registros de comunicación con el exterior

En la parte superior izquierda, el *registro de direcciones*, AR *(Address Register)*, sirve para contener 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 memoria 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 memoria y los periféricos a los buses el que, mediante circuitos descodificadores, determine qué direcciones no son de la memoria sino de puertos.

Abajo a la izquierda hay dos **registros de datos**. En RDR *(Read Data Register)* se introducen los datos que se leen del exterior, y en WDR *(Write Data Register)* los que se van a escribir.

Si hay un registro, RDR, en el que se introducen los datos leídos (de la memoria o de los puertos periféricos), hace falta otro, el **registro de instrucciones**, IR *(Instruction Register)* para introducir las *instrucciones* que se leen de la memoria. Pero no sólo hay uno, sino tres: IRL, IRD e IRE. 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 8.9, página 136): mientras una se ejecuta (la que está en IRE) la siguiente (en IRD) se está descodificando y la siguiente a ésta (en IRL) se está leyendo.

Cuando la CU tiene que 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 9.5), pone la dirección en el registro AR, si es una salida de datos (escritura) pone también el dato en WDR, y activa las señales **req**, **r/w** y **size**. Si ha sido una petición de lectura (entrada de dato o de instrucción) después de un ciclo de reloj la CU introduce el contenido del bus D en RDR o IRL, 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**.

### Modos usuario y supervisor

Los **modos de funcionamiento** de un procesador, que ya hemos definido en el capítulo 2 (página 22), *no tienen nada que ver* con los **modos de direccionamiento** de la memoria, que hemos explicado de manera general en el apartado 8.7 y que veremos para el caso particular de BRM en el apartado 9.5.

Hay dos R13 y dos R14. El motivo es que el procesador BRM 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 memoria 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 el puntero de pila, R13 (e incrementará éste), 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.

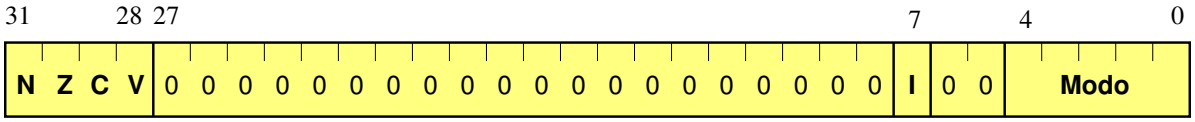



**Figura 9.2**: Registro de estado, CPSR.

Las informaciones que contiene CPSR son (figura 9.2):

- El **modo**[3], 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 dependen del resultado de la última operación realizada en la ALU. Recuerde (apartado 4.4) 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.

## 9.2. Modelo procesal

La unidad de control sigue el modelo procesal de tres fases de la figura 8.5 con encadenamiento (figura 8.8). Pero hay que matizar un detalle con respecto a esas figuras: como las instrucciones ocupan cuatro bytes, si 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 8.7 referente a la suma de las componentes de un vector (página 143). Las instrucciones del bucle, en pseudocódigo, eran:

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

| Dirección | Instrucción |
|---|---|
| $I + 12$ | Copiar en R3 el contenido de la palabra de la memoria *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 9.3). Es en la fase de ejecución de esa instrucción, en el intervalo de $t + 2$ a $t + 3$, cuando la unidad de control calcula la dirección efectiva, $I + 32 - 20 = I + 12$, y la introduce en el contador de programa. Pero la unidad de control, 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. Por tanto, en el instante $t + 3$ se «vacía» la cadena y empieza un nuevo proceso de encadenamiento, habiéndose perdido dos ciclos de reloj[4].

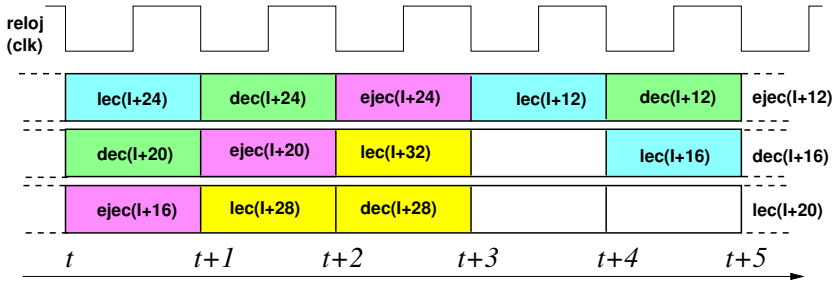



**Figura 9.3**: Bifurcación en la cadena.

Así pues, las instrucciones de bifurcación retrasan la ejecución, tanto más cuanto menos instrucciones tenga el bucle. En el apartado 10.5 veremos que en BRM es posible, en ciertos casos, prescindir de instrucciones de bifurcación, gracias a que todas las instrucciones son condicionadas.

## 9.3. 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 memoria deben estar *alineadas* (figura 8.11(a)): un número representado en una palabra (32 bits) tiene que tener su byte menos significativo en una dirección múltiplo de cuatro.

[4]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. La unidad de control 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 una bifurcación va a cumplirse o no, y actúan en consecuencia. Pero esto es propio del nivel de micromáquina, que no estudiamos aquí.

## Representación de instrucciones

Todas las instrucciones ocupan treinta y dos bits y también deben estar almacenadas en la memoria 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 sólo las bifurcaciones pueden ser condicionadas.

Hay cuatro tipos de instrucciones, y cada tipo tiene su formato. Pero los cuatro formatos comparten dos campos (figura 9.4):

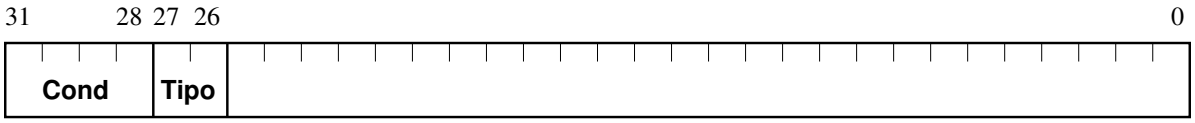



**Figura 9.4**: Formato de instrucciones: campos «condición» y «tipo».

- En los cuatro bits más significativos, «**Cond**» codifica la condición para que la instrucción se ejecute. La tabla 9.1 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 ALU 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 4.4.
- 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.

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 8.5: 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,

| 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 9.1**: Códigos de condición.

sin pretender asimilar todos los detalles, para pasar al capítulo siguiente y volver a éste cuando sea necesario.

## 9.4. 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 9.5.

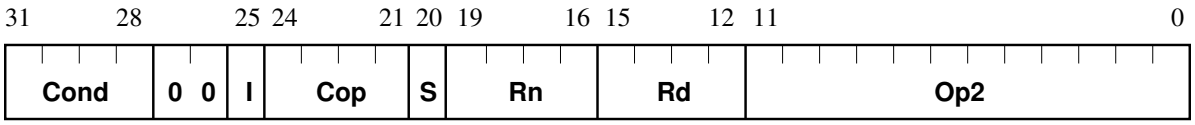



**Figura 9.5**: Formato de las instrucciones de procesamiento y movimiento.

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 9.2.

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 9.1). Para indicar una condición se añade el sufijo correspondiente según la tabla 9.1. Así, `ADDPL` significa «sumar si N = 0».

| 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 9.2**: Códigos de operación de las instrucciones de procesamiento y movimiento.

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 9.2): 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?[5]

Para empezar, la mayoría de las constantes que se usan en un programa son números pequeños. Según el formato de la figura 9.5, disponemos de doce bits en el campo Op2, 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.

[5]Recuerde que BRM es un RISC (apartado 8.7). En los CISC no existe este problema, porque las instrucciones tienen un número variable de bytes.

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 9.5 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 4.095? El convenio de BRM es que sólo 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 9.6). Conviene volver a mirar la figura 9.1: la entrada 2 a la ALU (en este caso, procedente del registro IRD 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 4.2(b)). 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.

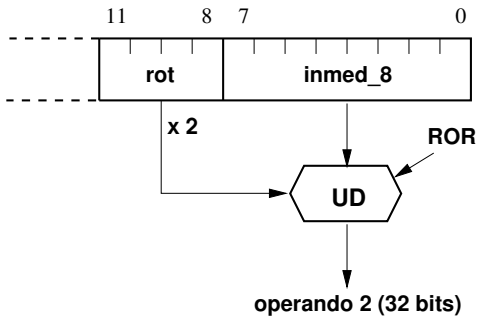



**Figura 9.6**: Operando inmediato.

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 $32-4\times2=24$. 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 10.3.

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; por ejemplo, `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`[6]. 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 9.2, `MVN` (*move negative*), produce el complemento a 1. Y el complemento a 1 de 4 es igual que el complemento a 2 de 5.

## Operando en registro

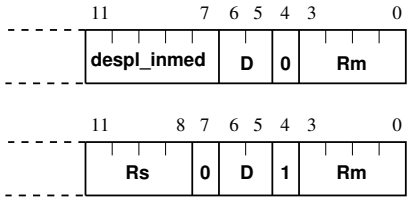



**Figura 9.7**: Operando en registro.

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 9.7):

- 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.
- bits 5 y 6 (D): tipo de desplazamiento (recuerde el apartado 4.4 y la figura 4.2):
  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.

[6] 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`).

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 9.1), 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 instrucciones 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 sólo para completar la descripción de BRM. La figura 9.8 muestra los formatos.

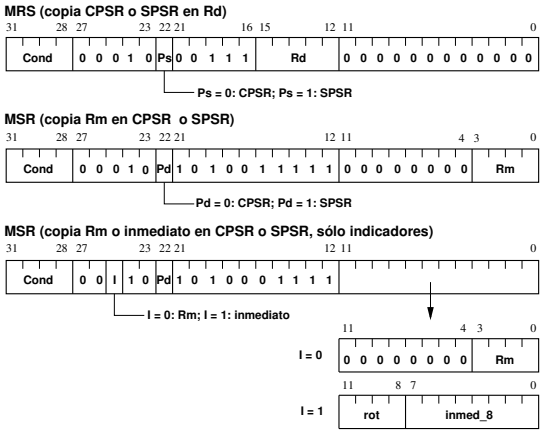



**Figura 9.8**: Instrucciones `MRS` y `MSR`

`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 (sólo en modo supervisor)
```

Para copiar en un registro de estado hay dos versiones. En una se modifica el registro completo (pero en modo usuario sólo pueden modificarse los indicadores, o «flags»: bits N, Z, C y V, figura 9.2), y en otra sólo 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
```

## 9.5. 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 memoria o en un puerto, o a la inversa. Su formato es el de la figura 9.9

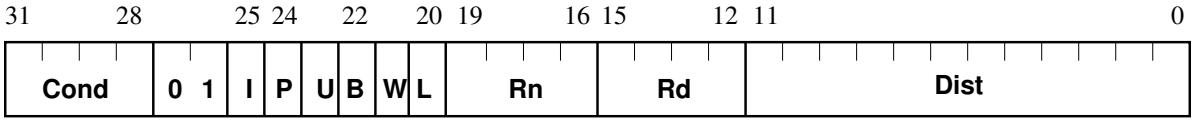



**Figura 9.9**: Formato de las instrucciones de transferencia de datos.

A pesar de la aparente complejidad del formato, hay sólo cuatro instrucciones, que se distinguen por los bits 20 (L: *load*) y 22 (B: *byte*). En la tabla 9.3 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]<br>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 9.3**: Instrucciones de transferencia de datos.

«DE» es la *dirección efectiva* (apartado 8.7), 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 9.7). 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 8.7):

```
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, sólo 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 9.7.

- 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  LDREQB 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
```

## 9.6. Instrucciones de transferencia de control

Puesto que el contador de programa es uno más entre 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 sólo 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 9.10. El bit 24 (L) distingue entre `B` (L = 0) y `BL` (L = 1).

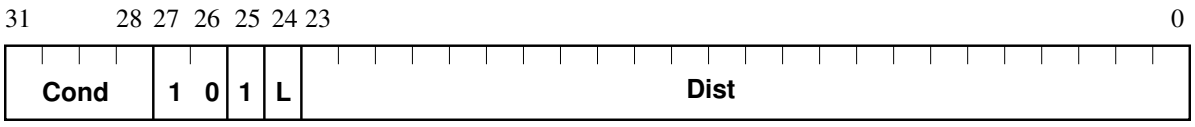



**Figura 9.10**: Formato de las instrucciones de bifurcación.

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 10.2 y 10.6, 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` («software interrupt») o `SVC` («supervisor call»), tiene un formato muy sencillo, el de la figura 9.11.

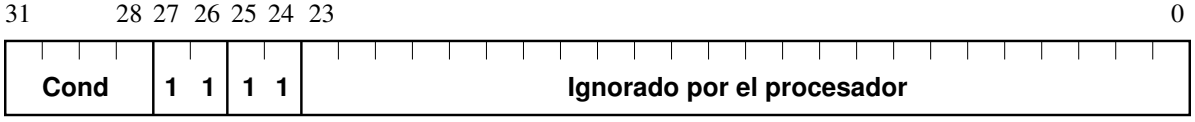



**Figura 9.11**: Formato de la instrucción `SWI`.

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.

## 9.7. Comunicaciones con los periféricos e interrupciones

En el apartado 8.9 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 10.8.

También nos referíamos a las interrupciones en términos generales en el apartado 8.9: 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 9.1 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 9.2).
- 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[7]. 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 9.4 se indican los orígenes de las causas y las direcciones de los vectores[8].

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, grabada en ROM, consiste en realizar las operaciones de arranque inicial (página 54).

[7] 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.

[8] 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.

| Interrupción | Causa | Dirección del vector |
|---|---|---|
| *Reset* | Activación de la señal *reset* (figura 9.1) | 0x00 |
| *Instrucción desconocida* | El procesador no puede descodificar la instrucción | 0x04 |
| *de programa* | Instrucción `SWI` | 0x08 |
| *IRQ* | Activación de la señal *irq* | 0x18 |

**Tabla 9.4**: Tabla de vectores de interrupción.

Las otras causas sólo 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 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 9.4 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 9.5)? 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:* su efecto es que, además de la función propia de la instrucción, hace una 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)

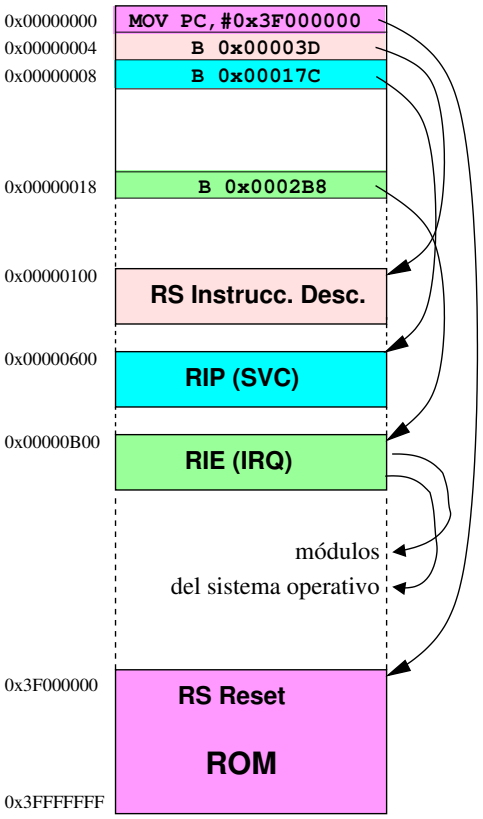



**Figura 9.12**: Mapa de memoria de vectores y rutinas de servicio.

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 9.12, 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 10.8 comentaremos lo que pueden hacer las rutinas de servicio.

# Capítulo 10

# Programación de BRM

---

Ya hemos dicho en el apartado 1.6 que un lenguaje ensamblador, a diferencia de uno de alto nivel, está ligado a la arquitectura (ISA) de un procesador. Pero no sólo es que cada procesador tenga su lenguaje, es que para un mismo procesador suele haber distintos convenios sintácticos para codificar las instrucciones de máquina 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 B.3 del apéndice B 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 B.3): un programa ensamblador que se ejecuta bajo un sistema operativo (Windows, Linux, Mac OS X...) 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. También se pueden ensamblar directamente y ejecutar en un sistema con arquitectura ARM, como se explica en el apartado 10.9. Debería usted no solamente 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.

## 10.1. 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 los 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 (apartado 7.4).

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 memoria. 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 9.6). 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 B.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 en un fichero binario), nos dé un listado (en el apartado B.3 se explica cómo hacerlo), obtenemos lo que muestra el programa 10.1.

[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 «`;`».

```
                        /*********************************************
                        *                                            *
                        *               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 10.1**: 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 B.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 B.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 B.2).

## 10.2. Etiquetas y bucles

En el apartado 8.7, 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.

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:

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

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

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 10.2 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 10.2**: 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 9.10). 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 × 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 × (−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 9.4 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 al tratar de ensamblar.

## 10.3. 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 9.5).

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 9.9). 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 sólo muestra el código, pero los contenidos de la memoria se pueden ver como se indica en el apartado B.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 9.6). 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 9.5, 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 $2 \times 4$ 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 9.2), 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 10.2 cambiando la instrucción `mov r3,#Num` por `ldr r3,=Num` verá que el ensamblador ya no genera errores para ningún número.

## 10.4. 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 sólo 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 10.7 volveremos sobre los módulos, y en el 12.2 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 para introducir valores en la sección de datos son: `.word`, `.byte`, `.ascii`, `.asciz` y `.align`. Las dos primeras 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 10.3 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 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` | (instrucción `SWI`) |
| 0000101C | `00001024` | (dirección en la que está `AAAAAAAA`) |
| 00001020 | `00001028` | (dirección en la que está `BBBBBBBB`) |
| 00001024 | `AAAAAAAA` | |
| 00001028 | `BBBBBBBB` | |

```
                  /************************************************
                  *     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 10.3**: Intercambio de palabras.

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 10.4 es la implementación de la suma de las componentes de un vector (apartado 8.7). En el simulador puede verse que el resultado de la suma 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.
- Otra desilusión provisional: pruebe usted a ejecutar el programa poniendo algún número negativo. El resultado es incorrecto. En el apartado 10.6 veremos una solución general con un subprograma, pero de momento le invitamos a pensar en arreglarlo con sólo 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 10.5 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 B.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.

```
                    /***********************************************
                    *  Suma de las componentes de un vector        *
                    *  (suma bytes, pero sólo 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 10.4**: Suma de las componentes de un vector.

```
                    /***********************************************
                    *   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 10.5**: Copia una cadena de caracteres

## 10.5. Menos bifurcaciones

Una instrucción de bifurcación, cuando la condición se cumple, retrasa la ejecución en dos ciclos de reloj (apartado 9.2). 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 9.2) puede hacerse así:

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

## 10.6. Subprogramas

La instrucción BL (apartado 9.6), que, además de bifurcar, guarda la dirección de retorno en el registro de enlace (LR = R14) permite llamar a un subprograma (apartado 8.7), 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 10.6, que incluye dos llamadas para probarlo.

```
                    /*************************************************
                    *                                                *
                    * 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 10.6**: Subprograma MCD y llamadas.

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 10.4)? 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.[3]

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 veinticuatro 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 veinticuatro 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
```

[3]Entre las instrucciones de ARM de las que hemos prescindido 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. Pero se trata de hacerlo sin esa instrucción.

Basta modificar el programa 10.4 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 parámetros 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 sólo 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 10.5) 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 8.7). 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 9.5). 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 |
|---|---|
| (M[(sp)+4]) → R0 | `ldr r0,[sp,#4]` |
| (M[(sp)+8]) → 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 10.7 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 sólo 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] @ coge el segundo parámetro
00000048 E59D1010        ldr  r4,[sp,#16] @ aoje 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 10.7**: Subprograma MCD y llamadas con paso por pila

## Anidamiento

Ya decíamos en el apartado 8.7 que si sólo 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 sobrescribiría 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 $\mathrm{MCD}(x, y, z) = \mathrm{MCD}(x, \mathrm{MCD}(y, z))$.

Como no cabe en una sola página, se ha partido en dos (programa 10.8 y programa 10.9) 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: recoge 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: recode el resultado
                                         @  MCD(9,81,156) = 3 → R9
00000048 E28DD008       add sp,sp,#8     @ para dejar la pila como estaba
0000004C EF000011       swi 0x11
                  .end
```

**Programa 10.8**: 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] @ coge el tercer parámetro
00000064 E59D2018        ldr  r2,[sp,#24] @ coge el segundo parámetro
00000068 E59D101C        ldr  r1,[sp,#28] @ coge 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
                                @ M[(SP)+4] y los parámetros segundo
                                @ y primero en M[(SP)+8] y M[(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] @ coge el segundo parámetro
000000B8 E59D4010        ldr  r4,[sp,#16] @ coge 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 M[(SP)+4] y el primer
                                             @ parámetro en M[(SP)+8]
                        .end
```

**Programa 10.9**: 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.

## 10.7. 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 de líneas de código.

Cualquier programa que no sea trivial se descompone en módulos, no sólo 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. En el apartado 12.2 estudiaremos cómo es este proceso de montaje.

Para que el ensamblador entienda que ciertos símbolos están definidos en otros módulos se puede utilizar una directiva, «`extern`», pero en el ensamblador GNU no es necesaria: entiende que cualquier símbolo no definido en el módulo es «externo». Sí es necesario indicar, mediante la directiva «`global`», los símbolos del módulo que son necesarios para otros. 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 no definidos y 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. Sólo hay que escribir, en tres ficheros diferentes, los tres módulos «`pruebaMCD3`» (programa 10.8), «`MCD3`» y «`MCD`» (programa 10.9), añadiendo las oportunas `global`. Le invitamos a hacerlo como ejercicio.

Veamos mejor otro ejemplo que exige 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 la necesidad de 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 10.10 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 10.11. Como necesita multiplicar y BRM no tiene instrucción para hacerlo[4], hemos añadido otro módulo (programa 10.12) con un algoritmo trivial para multiplicar[5].

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.

[4] ARM sí la tiene. Puede probar a sustituir la llamada a `mult` por la instrucción `mul r2,r1,r2` y verá que funciona igual.

[5] 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
                                .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 10.10**: Llamada a la función para calcular el factorial de 4

```
                /* Subprograma factorial recursivo */
                                  .text
                                  .global fact
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 10.11**: 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 10.12**: Subprograma para multiplicar

La figura 10.1 muestra los estados sucesivos de la pila para $N = 4$, suponiendo que el módulo con la función (programa 10.11) 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 10.10) 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.

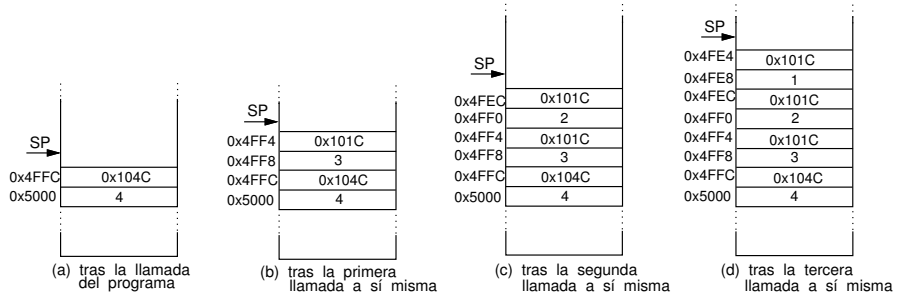



**Figura 10.1**: Estados de la pila en el cálculo del factorial.

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`»).

## 10.8. Comunicaciones con los periféricos e interrupciones

Decíamos en el apartado 9.7 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 ASCII 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 ASCII del carácter a presentar |

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

Cuando se pulsa una tecla, el controlador del teclado 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 9.2). 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[6]. 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 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 interrupciones permite esa «simultaneidad» de la ejecución de un programa cualquiera y de la transferencia de caracteres. Como decíamos en el apartado 8.9, el mecanismo es una combinación de hardware y software:

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

En el hardware, el procesador, al atender a una interrupción, realiza las operaciones descritas en el apartado 9.7. 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.

El procesador y los controladores están conectados como indica la figura 10.2. 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 sólo 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.[7] Inicialmente, el hardware, antes de ejecutarse el programa de arranque en ROM (figura 9.12), inhibe las interrupciones de los controladores (IT = 0) y el permiso general de interrupciones (I = 0 en el CPSR, figura 9.2)

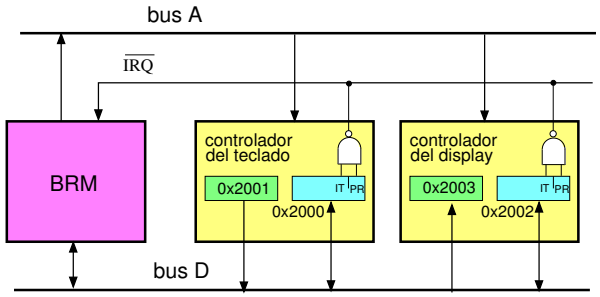



**Figura 10.2**: Conexión de controladores de teclado y *display*.

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 0xB00 (suponiendo el mapa de memoria de la figura 9.12). 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 9.7. 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 sólo 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 usuario 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

[7] El motivo de que sea NAND y no AND y de que la línea se active con «0» y no con «1» es que puede implementarse electrónicamente la función OR uniendo las salidas de las NAND. $\overline{\text{IRQ}}$ es una «línea de colector abierto».

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. Además, las rutinas deben salvaguardar los registros que usen y restaurarlos al final. En pseudocódigo:

**Rutina de servicio del teclado:**

```
Guardar registros en pila
Leer car en puerto 2001 (0 → PR)
car → buftecl(xtecl)
Si xtecl=80 o car=<ret>ir a fin
xtecl+1 → xtecl
Restaurar registros y volver
fin: 1 → fintecl
0 → IT (en puerto 2000)
Restaurar registros y volver
```

**Rutina de servicio del *display*:**

```
Guardar registros en pila
Leer car en bufdisp(xdisp)
car → puerto 2003 (0 → PR)
Si xdisp=80 o car=<ret>ir a fin
xdisp+1 → xdisp
Restaurar registros y volver
fin: 1 → findisp
0 → IT (en puerto 2002)
Restaurar registros y volver
```

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 9.12). 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*.

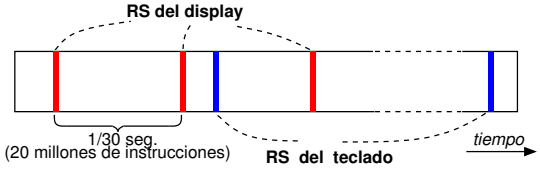



**Figura 10.3**: Tiempos de ejecución de las rutinas.

En una aplicación en la que ambos periféricos están trabajando resultaría un reparto de tiempos como el indicado en la figura 10.3. Se supone que el *display* tiene una tasa de transferencia de 30 caracteres por segundo, y que la del teclado es menor. Las zonas coloreadas representan los tiempos de ejecución de las rutinas de servicio, y las blancas todo el tiempo que puede estar eje-

cutándose el programa (o los programas) de aplicación. Si la velocidad del procesador es 600 MIPS (millones de instrucciones por segundo), en 1/30 segundos puede ejecutar 20 millones de instrucciones. 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 11.4).

## Consulta de interrupciones

En los ejemplos anteriores sólo 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 8.9, 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 identificación 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) sólo tiene que leer ese puerto para inmediatamente bifurcar a la rutina de servicio que corresponda.

La figura 10.4 muestra las conexiones entre el procesador, el controlador de interrupciones y varios periféricos. Un de ellos es un *controlador de DMA*, que, como vamos a ver, es necesario para los periféricos rápidos.

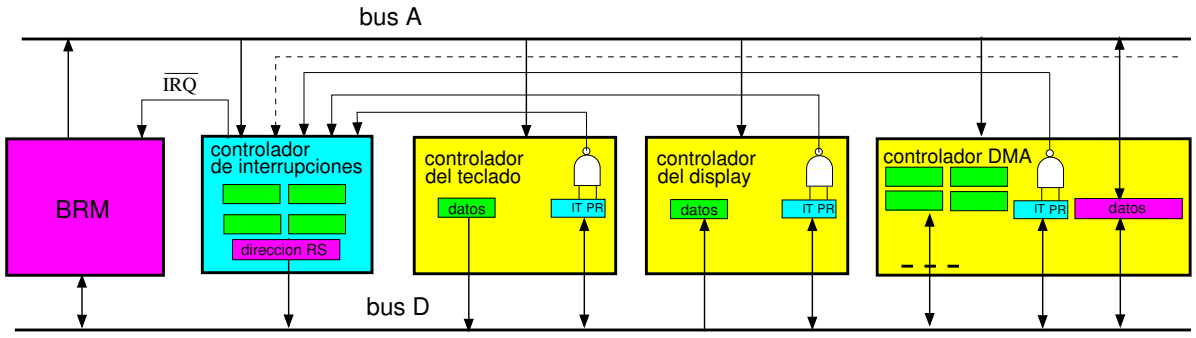



**Figura 10.4**: Controlador de interrupciones.

## Periféricos de bloques

Imagínese, mirando la figura 10.3, 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 $\mu$s. Suponiendo que

el procesador ejecuta 600 MIPS, el número de instrucciones que pueden ejecutarse en ese intervalo se reduce de 20 millones a veinte mil. 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 8.9). Para realizar una transferencia, el programa inicializa al controlador de DMA con los datos necesarios para que sea el controlador, y no la CPU, quien 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 9.1) 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 sólo 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.

## 10.9. Programando para Linux y 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 algunos ejemplos sencillos, que es fácil adaptarlos para que hagan uso de las llamadas al sistema en un dispositivo basado en ARM y un sistema operativo Linux o Android.

El ensamblador GNU admite indistintamente los códigos `SWI` y `SVC`. En los últimos documentos de ARM se prefiere el segundo, pero en los ejemplos anteriores hemos utilizado `SWI` porque es el único que entiende ARMSim#. En los siguientes pondremos `SVC`.

### Hola mundo

En un primer curso de programación es tradicional empezar con el «Hola mundo»: un programa (normalmente en un lenguaje de alto nivel) que escribe por la pantalla (la «salida estándar») esa cadena. El programa 10.13, escrito en el ensamblador GNU para ARM, hace eso mismo.

Como vimos en el tema de sistemas operativos (apartado 2.4) `WRITE` es una llamada de la SCI 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 la ABI de ARM, 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 `SVC`, forman la llamada al sistema.* Cuando el programa se ejecute, al llegar a `SVC` se produce una interrupción. La rutina de servicio de `SVC` (que ya forma parte del sistema operativo), al ver

```
                .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   svc 0x00000000
00000014 E3A00000   mov r0, #0      @ R0=0: código de retorno normal
00000018 E3A07001   mov r7, #1      @ R7=1: número de "EXIT"
0000001C EF000000   svc 0x00000000
                .data
                 cadena:
00000000 C2A1486F   .asciz "¡Hola mundo!\n"
         6C61206D
         756E646F
         210A00
                .end
```

**Programa 10.13**: Programa «Hola mundo».

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, y éste 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 la llamada `EXIT`, 1, se introduce en R7 antes de ejecutarse la instrucción `SVC`. Ahora, la rutina de servicio llama al sistema de gestión de memoria, que libera el espacio ocupado por el programa y el planificador se encarga de 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 10.14 pregunta al usuario por su nombre, lo lee, y le saluda con ese nombre. (Observe que en ARMSim# no podemos simular programas como éste debido a que no admite entrada del teclado).

## 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 B.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 10.2 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»*, no el número entero 233. Por tanto, hace falta un subprograma que convierta esa cadena 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

```
                .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   svc 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   svc 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   svc 0x00000000
0000003C E3A00001   mov r0, #1
00000040 E59F102C   ldr r1, =nombre
00000044 E3A02019   mov r2,#25
00000048 E3A07004   mov r7, #4
0000004C EF000000   svc 0x00000000
00000050 E3A00001   mov r0, #1
00000054 E59F1020   ldr r1, =cad3
00000058 E3A02002   mov r2,#2
0000005C E3A07004   mov r7, #4
00000060 EF000000   svc 0x00000000
00000064 E3A00000   mov r0, #0      @ R0 = 0: código de retorno normal
00000068 E3A07001   mov r7, #1      @ R7 = 1: número de "exit"
0000006C EF000000   svc 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 10.14**: Programa «Saludo».

sencilla, pero laboriosa (especialmente si, como debe ser, se incluye la detección de errores: caracteres no numéricos, etc.). Pero hay una manera más fácil: subprogramas de ese tipo están disponibles como funciones en la *biblioteca estándar de C*. Al final de este apartado veremos cómo puede utilizarse.

### Ensamblando, montando, cargando y ejecutando en Linux

Si usted dispone de un sistema con un procesador ARM y una distribución de Linux (por ejemplo, «Raspberry Pi»[8] o «BeagleBone Black»[9]) le resultará muy fácil comprobar los ejemplos. Entre otras muchas utilidades, la distribución incluirá un ensamblador («`as`») y un montador («`ld`»).

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: `as -o saludo.o saludo.s`
2. *Montar* el programa (ya hemos comentado en el apartado 10.4 que este paso es necesario aunque sólo haya un módulo): `ld -o saludo saludo.o`
3. Comprobar que `saludo` tiene *permiso de ejecución*, y si no es así, ponérselo: `chmod +x saludo`
4. Dar la orden de *carga y ejecución*: `./saludo` (si `saludo` está en un directorio contenido en la variable de entorno `PATH` basta escribir `saludo` desde cualquier directorio).

Y si tiene usted interés en seguir experimentando puede probar el programa `gdb` (GNU debugger), que permite seguir paso a paso la ejecución, poner puntos de ruptura, ir examinando los contenidos de los registros y de la memoria, etc. (Para esto es necesario que añada la opción «`-gstabs`» al ensamblador; lea `man as` y `man gdb`).

Este procedimiento no funciona si el ordenador está basado en otro procesador (normalmente, uno con arquitectura x86). El motivo es que en este caso el ensamblador `as` está diseñado para esa arquitectura, y, al no entender el lenguaje fuente, no puede traducir y sólo dará errores. Pero en este caso también pueden comprobarse los programas para ARM utilizando un ensamblador cruzado, como se explica en el apéndice B (apartado B.3) y un emulador, como `qemu`. El procedimiento ahora sería:

1. *Ensamblar* el programa fuente: `arm-linux-gnueabi-as -o saludo.o saludo.s`
2. *Montar* el programa: `arm-linux-gnueabi-ld -o saludo saludo.o`
3. *Simular* la ejecución para comprobar que funciona correctamente: `qemu-arm saludo`[10]

### Ensamblando, montando, grabando y ejecutando en Android

Android es un sistema operativo basado en Linux, y utiliza las mismas llamadas al sistema. Entonces, «¿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 realizarlo, pero si está usted pensando en aprender a desarrollar aplicaciones para este tipo de dispositivos es recomendable (por no decir imprescindible) utilizar entornos de desarrollo para lenguajes de alto nivel.
- Estos dispositivos están concebidos para la ejecución de aplicaciones, no para su desarrollo, por lo que la construcción del programa ejecutable hay que hacerla en un ordenador que disponga de las herramientas (`as`, `ld`, etc.).

[8] `http://www.raspberrypi.org`

[9] `http://beagleboard.org/`

[10] En algunas instalaciones es necesario decirle a `qemu` dónde se encuentran algunas bibliotecas. Por ejemplo, en Debian: `qemu-arm -L /usr/arm-linux-gnueabi saludo`

- El procedimiento que vamos a describir se ha probado en dos dispositivos Samsung: Galaxy S con Android 2.2 y Galaxy Tab 10.1 con Android 4.0, 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 dispositivo con una distribución de Linux (como Raspberry Pi), o bien 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 B (apartado B.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.

El procedimiento es:

1. *Generar* el programa ejecutable como se ha indicado antes y comprobar que funciona (ejecutándolo, si se ha generado en un sistema ARM, o con `qemu`, en caso contrario).
2. *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` (un componente de «Android SDK», apartado B.3), pero para este sencillo ejercicio se puede conseguir de otro modo sin necesidad de instalar más herramientas en el ordenador:
3. Conectar el dispositivo al ordenador y transferir el fichero.
4. 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.
5. Abrir el terminal (en el dispositivo) y adquirir privilegios de administrador (`su`).
6. Comprobar que existe `/data/local/bin`, y si no es así, crearlo (`mkdir`).
7. Si el fichero que hemos transferido es, por ejemplo, `saludo`, 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/sdcard/saludo > /data/local/bin/saludo`.
8. Asegurarse de que el fichero tiene permiso de ejecución, y si es necesario ponérselo con `chmod`.
9. 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).
10. Ya puede ejecutarse el programa (como usuario normal) escribiendo «`saludo`» en el terminal. (Si no aparecen los caracteres no-ASCII «¿» y «ó», configure el terminal para que acepte UTF-8).

La figura 10.5 muestra esas operaciones realizadas en el terminal (se ha ocultado el teclado virtual para que se vean todas las líneas), y en la figura 10.6 puede verse el resultado de la ejecución. La figura 10.7 es el resultado del programa que veremos a continuación

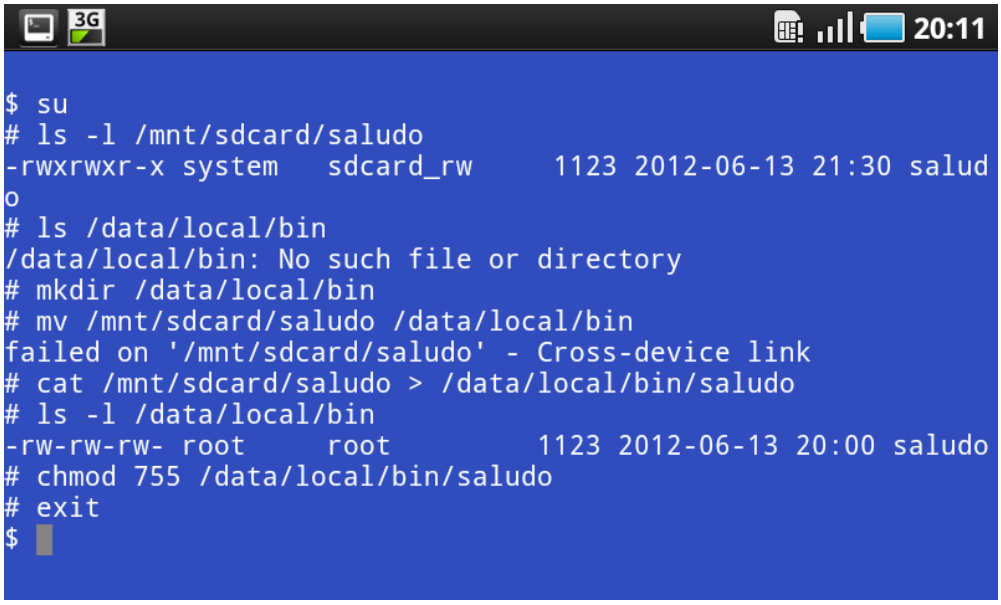



**Figura 10.5**: Órdenes para copiar el programa ejecutable en un directorio con permisos de ejecución.

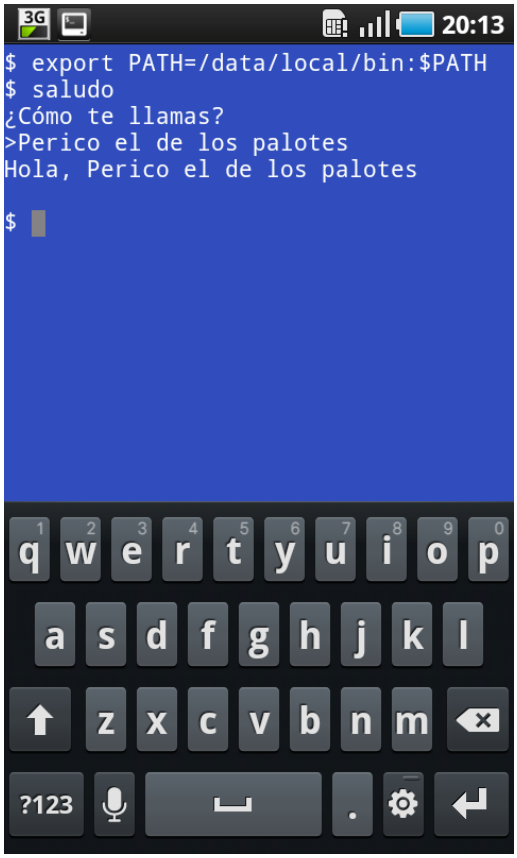



**Figura 10.6**: Ejecución de `saludo`.

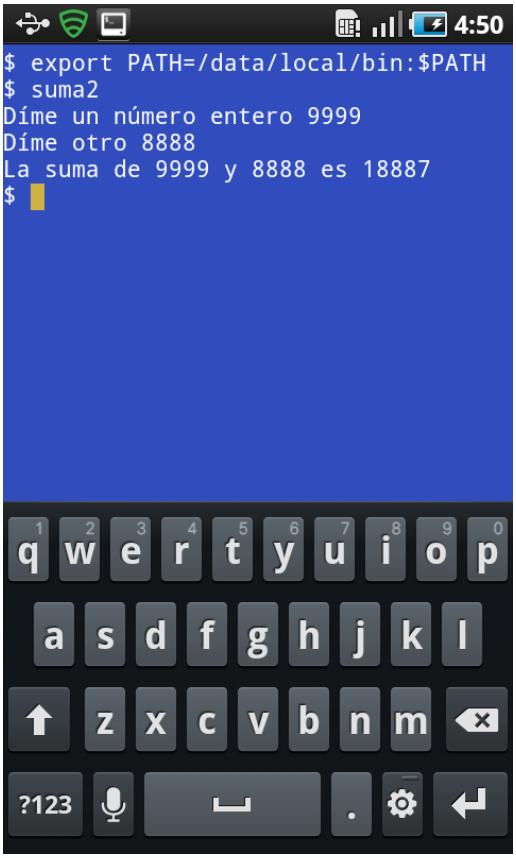



**Figura 10.7**: Ejecución de `suma2`.

### Usando la biblioteca estándar de C (libc)

Un programa tan sencillo como el de sumar dos números enteros (programa 10.1) se complica enormemente si queremos convertirlo en interactivo, es decir, que lea los operandos del teclado y escriba el resultado en la pantalla. El motivo es, como ya hemos dicho, que hay que acompañarlo de un subprograma que convierta las cadenas de caracteres leídas en números y otro que convierta el resultado numérico en una cadena de caracteres, a fin de poder utilizar las llamadas `READ` y `WRITE`.

Ya vimos en el tema de sistemas operativos que el software del sistema tiene bibliotecas que incluyen esos subprogramas y muchas otras cosas de utilidad (página 30). Concretamente, en la biblioteca llamada «libc» hay dos funciones, `scanf` y `printf`, que, llamándolas con los parámetros adecuados, leen y escriben en una variedad de formatos, y se encargan también de las llamadas necesarias al sistema. Para llamarlas desde un programa en ensamblador basta utilizar la instrucción normal de llamada a subprograma, `bl scanf` y `bl printf`. El programa 10.15 es un ejemplo de su uso para el caso que nos ocupa. Los comentarios incluidos en el listado explican cómo se utilizan estas funciones y los parámetros que se les pasan en este ejemplo. (Si quiere usted saber más sobre estas funciones, puede hacer como siempre: `man scanf` y `man printf`).

Son necesarias algunas observaciones sobre este programa y su procesamiento:

- El programa es sintácticamente correcto (enseguida explicaremos por qué que el símbolo de entrada global es `main`, y no `_start`), y se puede ensamblar sin errores. De hecho, el listado se ha obtenido con `arm-linux-gnueabi-as -al -EB suma2.s > suma2.list`, orden a la que podríamos añadir `-o suma2.o` para generar un módulo objeto.

- Pero si se intenta montar `suma2.o` el montador (`ld`) sólo genera errores, porque no encuentra los módulos objeto correspondientes a `scanf` y `printf`.

- El motivo es que, como indica su nombre, la biblioteca que incluye a `bl scanf` y `bl printf` es un conjunto de funciones *en lenguaje C*. Sólo se puede generar un programa ejecutable combinando esas funciones con nuestro programa fuente mediante el compilador de C, `gcc`.

- `gcc` admite programas fuente tanto en lenguaje C como en ensamblador. Pero el símbolo «`_start`» está definido internamente en una de sus bibliotecas, y el que hay que utilizar como símbolo de entrada es «`main`».

- La ejecución de `gcc` encadena los pasos de traducción y de montaje. Por tanto, para generar el ejecutable basta con `gcc -static -o suma2 suma2.s`
  (o `arm-linux-gnueabi-gcc -static -o suma2 suma2.s` si se hace en un sistema basado en un procesador que no sea ARM).

- La opción «`-static`» sólo es necesaria si el resultado (`suma2`) se va a transferir a un dispositivo como un móvil o una tableta. Si no se pone, el ejecutable generado es más pequeño, pero requiere cargar dinámicamente las bibliotecas, cosa imposible en esos dispositivos.

- Para programar aplicaciones, ya lo hemos dicho, es preferible utilizar algún lenguaje de alto nivel. Si se compila el programa 10.16, que está escrito en lenguaje C, se obtiene un ejecutable que realiza la misma función. Y, además, el mismo programa fuente se puede compilar para distintas arquitecturas.

```
              .text
              .global main
              main:

              /* En esta llamada a printf sólo se le pasa la
                 dirección del texto a escribir: */
0000 E59F0054    ldr r0, =texto1
0004 EBFFFFFE    bl printf
              /* (Podría haberse hecho directamente con la llamada
                     al sistema, como en los ejemplos anteriores) */

              /* A scanf se le pasan en esta llamada dos parámetros:
                   en r0 la dirección de una cadena que indica el formato, y
                   en r1 la dirección donde tiene que poner lo leído */
0008 E59F0050    ldr r0, =formato @ pone la dirección de formato como primer
                                  @  parámetro ("%d" indica número decimal)
000C E59F1050    ldr r1, =n1      @ y la dirección donde dejar el resultado
                                  @  como segundo
0010 EBFFFFFE    bl scanf

              /* Repetimos para el segundo número: */
0014 E59F004C    ldr r0, =texto2
0018 EBFFFFFE    bl printf
001C E59F003C    ldr r0, =formato
0020 E59F1044    ldr r1, =n2
0024 EBFFFFFE    bl scanf

              /* Ahora llevamos los números a r0 y r1: */
0028 E59F0034    ldr r0, =n1
002C E5900000    ldr r0, [r0]
0030 E59F1034    ldr r1, =n2
0034 E5911000    ldr r1, [r1]

              /* Hacemos la suma: */
0038 E0802001    add r2, r0, r1

              /* Llamamos a printf con cuatro parámetros: */
003C E1A03002    mov r3, r2   @ el resultado (último en escribirse)
0040 E1A02001    mov r2, r1   @ el segundo número
0044 E1A01000    mov r1, r0   @ el primero
0048 E59F0020    ldr r0, =formato_salida @ la cadena que indica el formato
004C EBFFFFFE    bl printf

              /* Llamada para salir:*/
0050 E3A00000    mov r0,#0
0054 E3A07001    mov r7,#1
0058 EF000000    svc 0x0

              .data
0000 256400      formato: .asciz "%d"
0003 44C3AD6D    texto1:  .asciz "Díme un número entero "
     6520756E
     206EC3BA
     6D65726F
     20656E74
001C 44C3AD6D    texto2:  .asciz "Díme otro "
     65206F74
     726F2000
0028 4C612073    formato_salida: .asciz "La suma de %d y %d es %d\n"
     756D6120
     64652025
     64207920
     25642065
0042 00000000    n1: .skip 4 @ donde se guarda el primer número
0046 00000000    n2: .skip 4 @ donde se guarda el segundo
              .end
```

**Programa 10.15**: Programa «Suma2».

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int n1, n2;
  printf("Díme un número entero ");
  scanf("%d", &n1);
  printf("Díme otro ");
  scanf("%d", &n2);
  printf("La suma de %d y %d es %d\n", n1, n2, n1+n2);
  exit(0);
}
```

**Programa 10.16**: Versión en C del programa «Suma2».

Después de la última observación, y como conclusión general, quizás se esté usted preguntando: «Entonces, ¿a qué ha venido todo este esfuerzo en comprender y asimilar todos estos ejemplos en ensamblador?». La respuesta es que el objetivo de este capítulo no era aprender a programar en ensamblador, sino comprender, mediante casos concretos, los procesos que tienen lugar en el funcionamiento de los procesadores hardware y software.

# Capítulo 11

# Paralelismo, arquitecturas paralelas y procesadores gráficos

En arquitectura de ordenadores el adjetivo «paralelo» se aplica en un sentido temporal: «arquitecturas paralelas» se refiere a sistemas cuyo modelo procesal contiene actividades que se realizan simultáneamente. Es lo contrario de «secuencial».

La arquitectura de von Neumann es secuencial, porque las instrucciones se ejecutan una tras otra, y hasta que no ha terminado una no se comienza con la siguiente (apartado 8.2). Pero esta afirmación puede matizarse: depende del nivel de detalle con que se considere, ya que si descendemos al nivel de bit es claramente falsa: las operaciones en la memoria se realizan sobre un determinado número de bits simultáneamente, es decir, en paralelo; igualmente ocurre con las operaciones aritméticas y lógicas. Por otra parte, los procesos en el nivel de microarquitectura implican que en todo momento puede haber muchas microórdenes activas, realizando simultáneamente varias acciones (por ejemplo, lectura de una instrucción y preparación de la dirección de la siguiente, figura 8.5).

Al afirmar la naturaleza secuencial de la arquitectura básica se dan por supuesto esos modos de funcionamiento paralelos subyacentes. Lo que se pretende expresar es que se sigue una *secuencia* en los ciclos «extracción-ejecución»: en cada momento se está o bien leyendo una instrucción en la memoria o bien procediendo a su ejecución.

Sin embargo, en la arquitectura de BRM ya hay un funcionamiento paralelo debido al encadenamiento (apartado 8.6) que hace que se solapen en el tiempo distintas fases de la ejecución de las instrucciones. Es un «paralelismo en el nivel de instrucción», o **ILP** *(Instruction Level Paralelism)*.

Pero «arquitecturas paralelas» se refiere a un paralelismo más «agresivo»: acceso y procesamiento en paralelo a datos *(Data Level Paralelism)* o ejecución en paralelo de varios flujos de instrucciones *(Process Level Paralelism* o *Thread Level Paralelism)*.

En este capítulo volveremos sobre el encadenamiento porque no sólo se aplica a un flujo de instrucciones, también a operadores aritméticos que se utilizan en las arquitecturas paralelas y a los accesos a la memoria. También trataremos de algunas formas de organización de la memoria para conseguir accesos en paralelo.

En cuanto a las arquitecturas paralelas, veremos solamente sus principios básicos y su aplicación a los procesadores vectoriales y gráficos, porque hay una gran variedad y los detalles son muy prolijos.

## 11.1. Encadenamiento *(pipelining)*

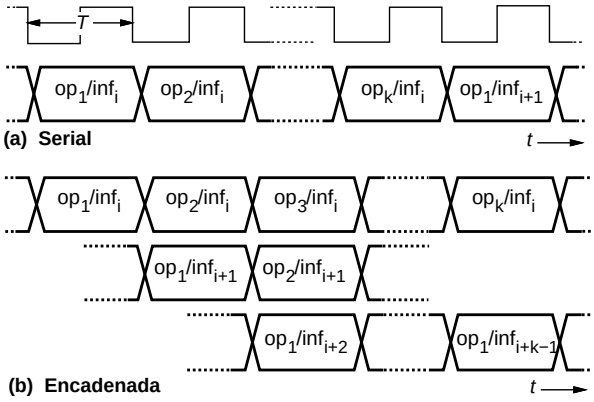



**Figura 11.1**: Encadenamiento de operaciones.

En el apartado 8.6 vimos el principio del encadenamiento en el flujo de instrucciones. La misma idea es aplicable a los flujos de datos. Supongamos que existe un flujo de informaciones y que cada una de ellas debe someterse a una secuencia de operaciones. La figura 11.1(a) muestra una manera de proceder puramente serial: mientras no se termina con un elemento (aplicación del operador «$op_k$» sobre el elemento «$inf_i$») no se empieza con el elemento siguiente.

Pero si disponemos de operadores independientes para las distintas operaciones podemos hacerlos funcionar simultáneamente, como ilustra la figura 11.1(b): una vez que el primer operador («$op_1$») ha procesado «$inf_i$», y mientras el segundo se ocupa de ella, puede ya empezar con «$inf_{i+1}$».

### Encadenamiento en los operadores aritméticos

En muchas aplicaciones es frecuente la repetición de operaciones aritméticas sobre flujos de datos que se presentan en paralelo. Por ilustrarlo con un ejemplo sencillo, supongamos que se trata de realizar la operación $a_i \times b_i + c_i \times d_i + e_i \times f_i$ un determinado número de veces sobre datos que se presentan secuencialmente (es decir, en un instante se presentan $a_1, ..., f_1$, en el instante siguiente $a_2, ..., f_2$, y así sucesivamente), y que esta operación es tan frecuente que resulta rentable disponer de un operador específico para ella, implementado completamente en hardware.

Un posible diseño para este operador podría estar basado en *un* operador de multiplicación y *otro* de suma. Supongamos que el tiempo de respuesta de ambos operadores es $T$. En el instante $t$=0 se cargarían seis registros de entrada con los primeros datos y empezaría la multiplicación $a_1 \times b_1$, en $t$=$T$ el resultado se dejaría en un registro intermedio y se empezaría la multiplicación $c_1 \times d_1$, y así sucesivamente. Como son necesarias tres multiplicaciones y dos sumas, el tiempo para una operación completa sería $5T$, y hasta transcurrido este tiempo no puede empezarse a operar sobre los siguientes datos ($a_2, ..., f_2$).

Frente a ese diseño serial, un operador encadenado para esta operación requeriría *tres* multiplicadores y *dos* sumadores, conectados como indica la figura 11.2 ; en la parte inferior se muestra un cronograma de su comportamiento. Obsérvese que los datos de entrada se presentan cada $T$ unidades de tiempo (y no cada $5T$) y que una vez que la cadena está llena (a partir de $t = 3T$) se obtiene también un resultado cada $T$ unidades de tiempo, a pesar de que cada operación individual ahora tarda $3T$ (desde que entran los datos hasta que se obtiene el resultado).

En este ejemplo, la cadena tiene tres **etapas** o **secciones**, y cada una de ellas se comunica con la siguiente mediante registros. En general, si el operador encadenado tiene $k$ secciones se dice que la cadena tiene **profundidad** $k$; cada cálculo completo requiere $k \times T$ unidades de tiempo, pero, una vez llena la cadena, se obtiene un resultado cada $T$ unidades de tiempo.

Una aplicación del encadenamiento de datos se encuentra, como ya sugiere el ejemplo, en los cálculos sobre *estructuras vectoriales* de datos. Otra, en las operaciones que de manera natural se

descomponen en fases; por ejemplo, se utiliza con frecuencia en el diseño de *procesadores de coma flotante*. Y también se utiliza habitualmente en los procesadores de gráficos (apartado 11.8).

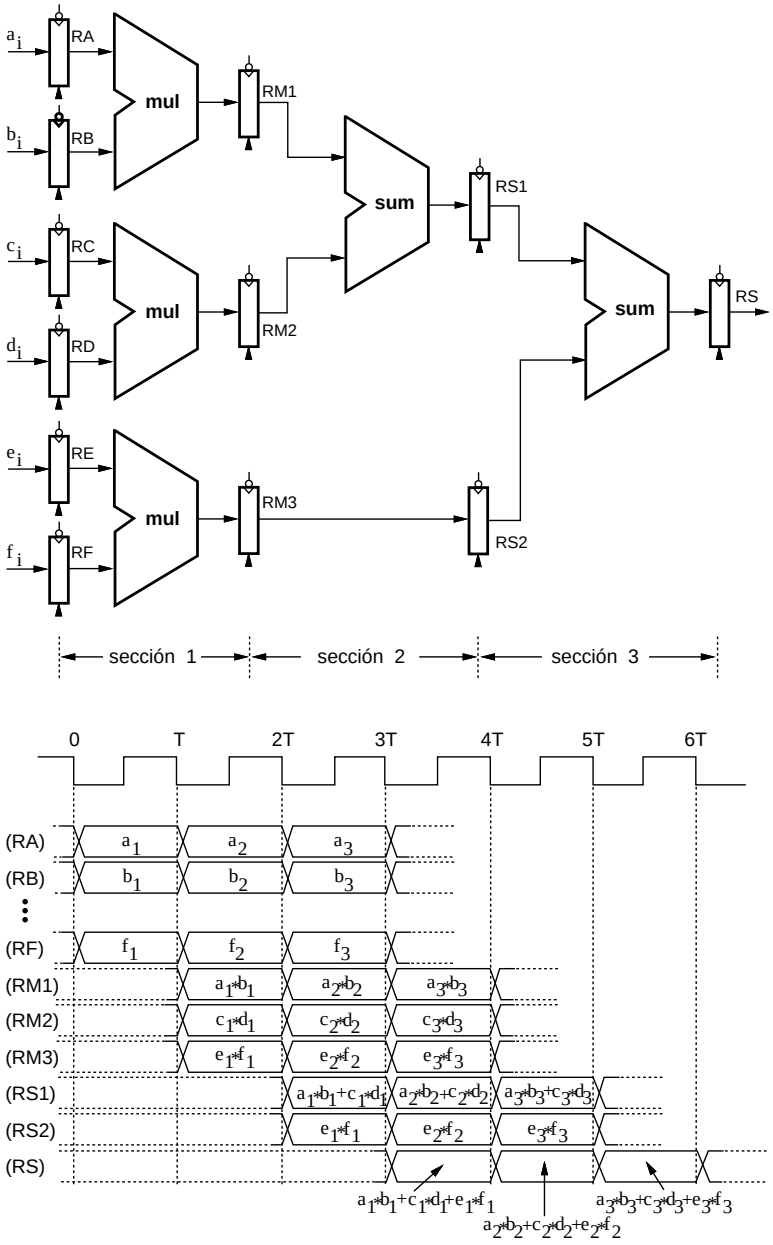



**Figura 11.2**: Esquema y cronograma de funcionamiento de un operador aritmético encadenado.

## Encadenamiento en los accesos a memoria

El mismo principio puede aplicarse en los accesos a la memoria. En el nivel de circuito, cada acceso requiere una sucesión de fases (decodificación de fila, decodificación de columna, lectura o escritura, eventualmente recuperación) que pueden encadenarse. Las memorias *cache*s (apartado 8.6) más comunes son del tipo llamado PBC *(Pipeline Burst Cache)*, en el que se combina el encadenamiento de

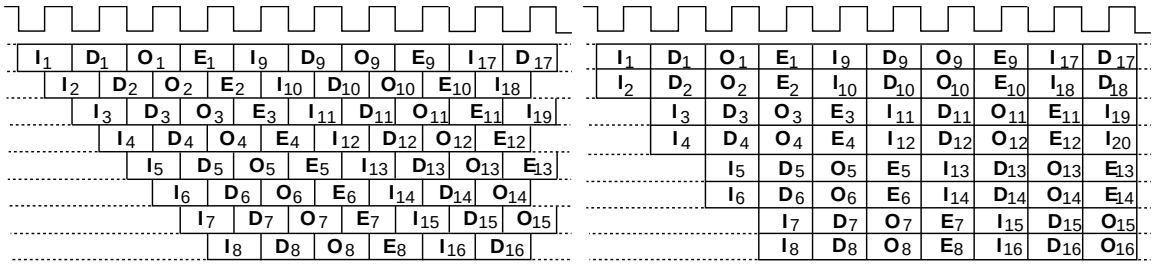
accesos a la memoria y la *cache* con un funcionamiento «a ráfagas»: no solamente se lee la dirección solicitada por el procesador sino también las siguientes. Esto es muy útil para los procesadores superescalares que veremos enseguida.

## Procesadores superencadenados, superescalares y VLIW

Muy brevemente, y sin entrar en los conflictos, que son más numerosos que los del encadenamiento sencillo del flujo de instrucciones (apartado 8.6), comentaremos tres refinamientos que están incorporados en prácticamente todos los microprocesadores.

Supongamos que son cuatro las etapas para cada instrucción: **I** (lectura de la instrucción), **D** (decodificación), **O** (acceso a los operandos) y **E** (ejecución).

El **superencadenamiento** consiste en adelantar las etapas aun cuando no se haya concluido la misma etapa con la instrucción anterior (figura 11.3a), y la organización **superescalar** (figura 11.3b) en hacer paralelas las mismas etapas sobre instrucciones sucesivas (ésta última es la que tienen los procesadores ARM recientes, pero con cadenas de profundidad mayor que 10)[1].




(a) Ejecución superencadenada. (b) Ejecución superescalar.

**Figura 11.3**: Superencadenado y superescalar.

Ambos requieren, a efectos de acceso a la memoria, que esta memoria esté organizada de modo que pueda escribirse o leerse por dos o más vías independientes (lo que puede conseguirse con esquemas que veremos en el apartado siguiente), o bien que el número de bits que se leen al acceder a la memoria sea lo suficientemente grande como para acoger a dos o más instrucciones. Y esta última sugerencia nos conduce a un tercer enfoque: si se dispone de varios procesadores de datos independientes es posible «empaquetar» varias instrucciones en una sola con campos para las distintas operaciones[2]. Por ejemplo, una sola instrucción puede contener varias operaciones de coma fija, varias de coma flotante y una bifurcación. El formato de estas instrucciones puede llegar a tener cientos de bits, y se habla entonces de «**VLIW**» *(Very Long Instruction Word).*

## Ejecución reordenada *(out-of-order)* y ejecución especulativa

A fin de reducir las paradas *(stalls)* de la cadena originados por los conflictos de datos o estructurales, la mayoría de los procesadores reorganizan el orden de ejecución de las instrucciones. En los procesadores encadenados y superescalares hay que modificar el modelo procesal básico (figura 8.5) en la etapa de ejecución: si los operandos requeridos para la ejecución de la instrucción están disponibles en los registros entonces ejecutarla; si no es así (porque algún operando sea el resultado de alguna

[1] Algunos autores consideran como ejemplos de **MISD** (apartado 11.4) a los procesadores diseñados para este modo de funcionamiento. Pero «MISD» implica varios flujos de instrucciones independientes unos de otros y ejecutados por distintas unidades de control todos operando sobre un mismo flujo de datos, lo que es verdaderamente raro.

[2] Volviendo así, curiosamente, al tipo de formato propuesto inicialmente en la «máquina de von Neumann» (figura 8.6).

operación anterior que aún no ha terminado) entonces «parar» la cadena (en realidad, introducir una «burbuja» en la cadena: un intervalo de tiempo en el que no se hace nada). Pero esto es muy ineficiente. La ejecución reordenada se apoya en elementos de hardware adicionales que modifican el modelo procesal de este modo:

1. Se lee la instrucción y se descodifica
2. Si los operandos están disponibles en la cola de resultados se ejecuta la instrucción; en caso contrario, se mete en una cola
3. Se ejecuta la instrucción que está en cabeza de la cola si sus operandos están disponibles, y el resultado se envía a una cola de resultados

Un sencillo ejemplo, con instrucciones de BRM:

```
ldr r0,[r1],#4
add r0,r0,r2
sub r3,r4,r5
...           @ otras instrucciones que no dependen de R0
```

El que el contenido de R0 esté disponible cuando se ejecuta `ADD` depende de la profundidad de la cadena, pero en cualquier caso si en `LDR` se produce un fracaso en el acceso a la memoria *cache* transcurren varios ciclos de reloj hasta que se lee de la memoria. El procesador no ejecuta las instrucciones en ese orden, sino en este otro:

```
ldr r0,[r1],#4
sub r3,r4,r5
...           @ otras instrucciones que no dependen de R0
add r0,r0,r2
```

Los conflictos de control son también muy determinantes para el rendimiento, como vimos en el apartado 9.2. En aquél ejemplo, con una cadena muy sencilla de tres etapas, veíamos que el hecho de que se siguiese alimentando la cadena con instrucciones como si la bifurcación no fuese efectiva suponía una penalización de dos ciclos de reloj. Ahora bien, en los bucles lo más frecuente es que la bifurcación *sí* sea efectiva. En ese ejemplo no tiene sentido, pero en cadenas de mayor profundidad cabe adoptar otra táctica: predecir que *sí* va a ser efectiva. Y un método más eficiente que requiere de nuevo algunos elementos de hardware adicionales ya lo señalábamos en una nota en el mismo apartado (nota 4, página 153): *predecir* si la bifurcación va a efectuarse o no. Basándose en esta predicción, ejecutar las instrucciones siguientes provisionalmente, pero no confirmar (escribir resultados en los registros) hasta no tener constancia de que la predicción ha sido correcta. Esto se llama *ejecución especulativa*.

### Implicaciones en el software

El objetivo del encadenamiento es multiplicar el número de instrucciones por segundo ejecutadas, o, lo que es lo mismo, reducir al mínimo el número de *ciclos de reloj por instrucción (CPI)*. En teoría, con el encadenamiento simple se podría tener CPI = 1, y con los de la figura 11.3, CPI = 1/2. Pero los conflictos hacen esto imposible. Aunque el procesador resuelva muchos mediante la ejecución reordenada, para obtener el máximo rendimiento de la ILP los programas en el nivel de máquina convencional tienen que abandonar la suposición de que las instrucciones se ejecutan secuencialmente y ordenarlas

adecuadamente. Ahora bien, normalmente se programa en algún lenguaje de alto nivel, y es el compilador el que contiene el conocimiento necesario para adaptarse al modelo funcional, resultando el encadenamiento transparente al programador en el nivel de máquina simbólica.

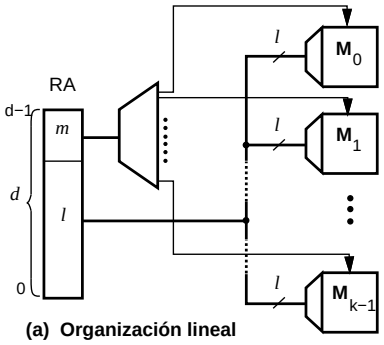
Por otra parte, los procesadores superencadenados y superescalares normalmente se utilizan en un entorno de software *multihebra* (apartado 11.7). Por ejemplo, en la figura 11.3 el flujo de instrucciones I1-I3-I5... pertenecería a una hebra de ejecución, y el flujo I2-I4-I6... a otra.

## 11.2. Paralelismo en los accesos a la memoria

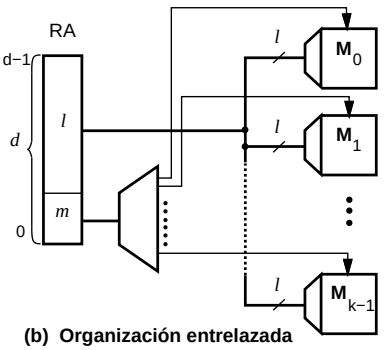
El «cuello de botella» originado por la naturaleza secuencial de la memoria (cada ciclo de memoria dura varios ciclos de reloj, y en cada ciclo sólo es posible realizar una operación de acceso) puede mitigarse organizando la memoria como un conjunto de módulos o **bancos**, cada uno con sus propias vías de acceso, que puedan funcionar en paralelo. Si las direcciones tienen $d$ bits (capacidad direccionable: $2^d$ bytes) podemos tener $k$ bancos de $2^l$ bytes cada uno, de modo que $k \times 2^l = 2^d$. Por ejemplo, direcciones de dieciséis bits (espacio de direccionamiento $2^{16}$ = 64 KB) y dieciséis bancos de $2^{12}$ = 4 KB. Una dirección de $d$ bits se descompone en dos partes: la dirección del banco (cuatro bits en el ejemplo) y la dirección dentro del banco (doce bits en el ejemplo). Una vez iniciada una operación de lectura o escritura en un banco se puede inmediatamente comenzar otra en otro banco.

La correspondencia concreta entre la dirección global, el número del banco y la dirección dentro de él determina la forma de la organización. Hay dos posibilidades, que llamaremos «lineal» y «entrelazada». En la figura 11.4 aparecen los esquemas básicos de ambas organizaciones, que vamos a comentar seguidamente.

### Organización lineal




(a) Organización lineal




(b) Organización entrelazada

**Figura 11.4**: Memoria modular.

En la organización lineal los bits de mayor peso de la dirección dan el número o dirección del banco y los de menor peso la dirección dentro del banco. En la figura 11.4(a) puede verse su esquema. Los $d$ bits de la dirección se descomponen en dos partes: los $m$ más significativos direccionan al banco (si hay $k$ bancos, $2^m = k$); los $l$ menos significativos se introducen en paralelo en la entrada de dirección de todos los bancos. Siguiendo con el ejemplo anterior, $d = 16$, $m = 4$ y $l = 12$; la dirección 0xBFFF selecciona el byte (o la palabra) de dirección 0xFFF=4095 dentro del banco 0xB=11 de entre los dieciséis posibles.

Se puede disponer inicialmente de una memoria con capacidad inferior a la máxima posible, cuya ampliación con más bancos es muy fácil. Si en lugar de los dieciséis posibles ponemos ocho bancos de 4 KB tendremos disponible una capacidad de 32 KB en lugar de 64 KB; lo único que ocurrirá es que la última dirección válida es 0x7FFF, y de 0x8000 a 0xFFFF («1» en el bit más significativo) son direcciones que no tienen físicamente asignada memoria. Basta «enchufar», por ejemplo, un banco más para disponer de las direcciones 0x8000 a 0x8FFF.

El inconveniente es que el acceso simultáneo a dos posiciones es imposible si estas posiciones son contiguas (salvo en el caso excepcional de que una corresponda al final de un banco y otra al

comienzo del siguiente). En la extracción de instrucciones precisamente nos interesa que se puedan ir adelantando las lecturas de direcciones consecutivas. Sin embargo, sí es posible que dos o más procesadores de datos o de instrucciones accedan simultáneamente a bloques distintos. Es, así, una organización adecuada para algunas estructuras de tipo SIMD o MIMD que veremos en el apartado 11.4.

### Organización entrelazada *(interleaved)*

Con la organización que muestra la figura 11.4(b), los bytes (o palabras) de direcciones consecutivas no se encuentran en el mismo banco, sino en bancos consecutivos, puesto que la dirección de banco viene fijada por los bits de menor peso de la dirección global. Todos los bancos posibles deben estar presentes, porque, de no ser así, en el espacio de direccionamiento quedarían «huecos»: con el mismo ejemplo anterior, si faltara el último banco (el que tiene dirección 15 = 0xF), las direcciones 0x000F, 0x001F, ..., 0xFFFF no serían válidas.

Este esquema resulta más útil que el anterior para anticipar la lectura de instrucciones, normalmente almacenadas en direcciones consecutivas. Iniciada la lectura de una instrucción, inmediatamente puede introducirse la dirección siguiente e iniciarse otra lectura, porque estará en el banco siguiente. De esta manera, tenemos una **memoria encadenada**, en la que pueden leerse varias instrucciones casi simultáneamente, como exigen los procesadores superencadenados y superescalares (figura 11.3). Es la organización usada en la memorias SDRAM.

Si el tiempo de acceso dentro de cada banco es $t_M$ y los $k$ bancos están continuamente ocupados, el tiempo de acceso del sistema de memoria será $t_{MM} = t_M/k$. En la práctica, y dependiendo del programa concreto que se esté ejecutando, se consigue un factor de reducción del tiempo de acceso más o menos grande, comprendido entre 1 y $1/k$.

## 11.3. Memoria asociativa

El principio funcional de una memoria de **acceso asociativo**, también llamada memoria **direccionable por el contenido**, o **CAM** *(Content Addressable Memory)*, es completamente diferente del «acceso aleatorio» que venimos considerando. En efecto, en una memoria asociativa los datos no se identifican por ninguna dirección, sino por ellos mismos.

La figura 11.5 puede servirnos para ilustrar el modelo funcional de una memoria asociativa. Los datos se almacenan en palabras de $n$ bits. Cada palabra contiene un grupo de $k$ bits que forma una **clave** asociada con los datos almacenados en los $n-k$ bits restantes. Las operaciones posibles en la memoria son tres: lectura, escritura y borrado.

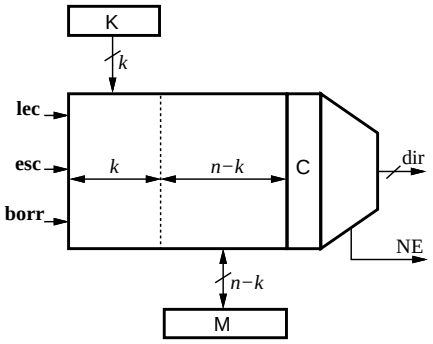



**Figura 11.5**: Memoria asociativa.

### Lectura

Para una operación de lectura se proporciona una clave en el registro K y se obtienen en el registro M los datos asociados. En la mayoría de las aplicaciones las claves almacenadas no pueden estar repetidas (la «clave» es algo que identifica de manera única a los datos asociados), pero si pudieran estarlo la memoria incluiría los mecanismos para obtener sucesivamente (o en paralelo, en varios registros M) todos los datos asociados a la clave dada. En lo sucesivo supondremos que las claves no pueden repetirse dentro de la memoria.

C es un **registro de coincidencia**, con tantos bits como palabras tiene la memoria; en la operación de lectura se pone a «1» el bit que corresponde a la palabra en la que está la clave. Como indica la

figura, un circuito codificador puede dar la dirección binaria de esta palabra, lo que resulta necesario, como veremos en el apartado siguiente, para algunas aplicaciones. Este mismo circuito puede generar una señal «NE» («no encontrado») cuando todos los bits de C son «0».

Lo más notable de este tipo de memoria es que, una vez introducida una clave en K, la «búsqueda» de los datos asociados se realiza en paralelo en el interior de la memoria. Esta operación de «búsqueda en tablas» es muy común en las aplicaciones, y en un ordenador convencional (con memoria de acceso aleatorio) exige la ejecución de un programa que, siguiendo un algoritmo u otro, tiene que acceder repetidamente a la memoria hasta encontrar la «clave». Una estructura basada en memoria asociativa acelera considerablemente este tipo de operaciones.

### Escritura

Para la operación de escritura se presentan la clave en K y los datos asociados en M, y los circuitos se encargan de almacenarlos en una palabra que esté libre. Estos circuitos pueden generar dos señales de error (no indicadas en la figura): una para el caso de que la memoria esté llena, y otra para el caso de que la clave ya exista.

### Borrado

La operación de borrado consiste en que los circuitos liberan la palabra cuya clave se da en K. Para esto (y también para permitir la escritura), cada palabra tiene internamente asociado un bit que indica si la palabra está libre u ocupada.

### Aplicaciones

Ya hemos sugerido antes la aplicación más inmediata de la memoria asociativa: la búsqueda paralela en datos organizados de forma tabular. Un **procesador asociativo** contiene una memoria asociativa y circuitos que permiten que en cada posición se puedan realizar algunas operaciones básicas (comparación, desplazamiento, etc.), resultando una estructura de tipo SIMD (apartado siguiente) en la que el procesador de datos incluye también funciones de almacenamiento. Con estos procesadores pueden conseguirse velocidades muy grandes para ciertos tipos de aplicaciones. La dificultad de su materialización se debe a la tecnología disponible, que se adapta muy bien a la construcción de memorias de acceso aleatorio, pero muy mal a la de memorias asociativas: tanto el coste por bit como la densidad de integración y el elevado consumo de energía que pueden alcanzarse hacen prohibitivo actualmente disponer de memorias asociativas de capacidad superior a unos pocos MiB. En cualquier caso, se trata de procesadores especializados, no de propósito general[3].

Tres aplicaciones muy comunes en las que se utilizan pequeñas memorias asociativas como elemento auxiliar son:

- Las TLB de las unidades de gestión de memoria (página 47).
- Los circuitos de control de las memorias *caches*, en los que, cuando se produce un fracaso, es necesario buscar en una tabla la dirección de la memoria que corresponde a la dirección solicitada.
- Los dispositivos de conmutación de redes *(routers* y *switches)*, en los que hay que consultar tablas de encaminamiento *(routing)*.

[3] Un ejemplo es el *Silicon Vertex Tracker* utilizado en el LHC *(Large Hadron Collider)* del CERN, que contiene varios cientos de «AMChips».

## 11.4. Arquitecturas paralelas

Como decíamos al principio de este capítulo, por «arquitecturas paralelas» se entienden modelos estructurales, funcionales y procesales en los que varios flujos de datos y/o de instrucciones se procesan simultáneamente. Es clásica la «clasificación de Flynn»: SISD, SIMD, MIMD.

### SISD

Las arquitecturas **SISD** (un Solo flujo de Instrucciones y un Solo flujo de Datos) son las del modelo básico. La figura 11.6 repite su principio (haciendo abstracción de los detalles de buses, registros, etc.), con un pequeño cambio en la terminología para comparar con las otras alternativas: «PI» es el *procesador de instrucciones*, correspondiente a lo que hasta ahora hemos llamado «unidad de control», y «PD» es el *procesador de datos*, que contiene la ALU y los registros aritméticos. El bloque «M» puede ser la memoria principal, o una *cache*. No hay funcionamiento en paralelo (salvo el ILP, si el PI tiene encadenamiento). La vía única de acceso a la memoria es el «cuello de botella de von Neumann» (apartado 8.6) que, como sabemos, se puede mitigar con una arquitectura Harvard (apartado 8.1).

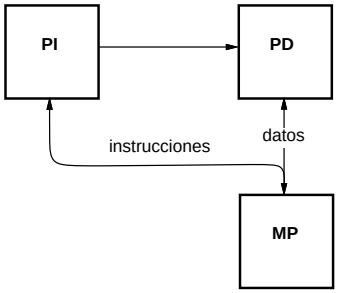



**Figura 11.6**: Arquitectura SISD.

### SIMD

Las arquitecturas **SIMD** (un Solo flujo de Instrucciones y Múltiples flujos de Datos) mantienen un único procesador de instrucciones (unidad de control) que ejecuta instrucciones secuencialmente (o encadenamiento), pero permiten hacer operaciones en paralelo sobre los datos. En la figura 11.7 se ilustra su principio: varios procesadores de datos trabajan en paralelo, y pueden acceder simultáneamente a la memoria a través de una «red de interconexión» (RI); para que tal cosa sea posible, la memoria debe disponer de varias vías de acceso, con una organización modular.

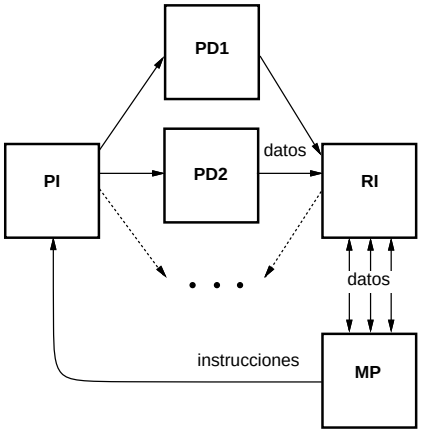



**Figura 11.7**: Arquitectura SIMD.

En estas arquitecturas SIMD, al conservarse un flujo único de instrucciones (y, por tanto, un único contador de programa), el modelo funcional en el nivel de máquina (es decir, la arquitectura ISA) sigue siendo del mismo tipo que en las SISD.

### MIMD

En las arquitecturas **MIMD** (Múltiples flujos de Instrucciones y Múltiples flujos de Datos) ya no existe una única «unidad de control» (o «procesador de instrucciones»), sino varias que pueden estar interpretando instrucciones simultáneamente. Hay una gran variedad de esquemas, pero se pueden clasificar en dos tipos (figura 11.8): multiprocesadores y multiordenadores.

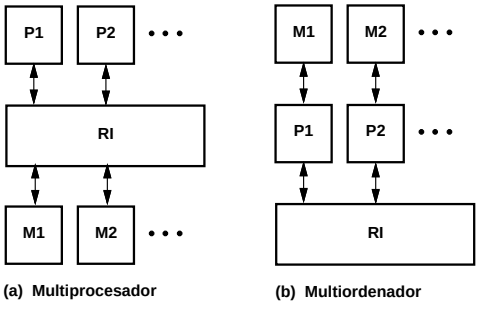



**Figura 11.8**: Arquitecturas MIMD.

Los **multiprocesadores** comparten un mismo espacio de direccionamiento, aunque cada procesador puede (y suele) estar acompañado de su propia *cache* y a veces de un módulo de memoria local, o privada. En la figura 11.8(a), «P1», «P2»... son, en general, procesadores de instrucciones o de datos, y «M1», «M2»... módulos de memoria (puede haber sólo uno). La «RI» (red de interconexión) puede ser simplemente un bus, pero también puede ser una matriz de circuitos de conmutación.

Los **multiordenadores** no comparten memoria. Mediante la red de interconexión, que normalmente es también un bus, los procesadores se comunican mediante **paso de mensajes**. Las **agrupaciones de ordenadores**, o **COW** *(Clusters of Workstations)* están formadas por cientos o miles de ordenadores de bajo coste. Cada nodo es o bien un ordenador completo (como en las redes de área local, pero con un software que permite el trabajo cooperativo entre los nodos), o bien sólo procesador y memoria, de modo que el conjunto es como una sola máquina. De este segundo tipo son los «Beowulf clusters», muy extendidos para trabajo científico por su relativamente bajo coste y la facilidad de implementación con el software disponible en muchas distribuciones de Unix.

Finalmente, para completar la clasificación, habría que mencionar a las arquitecturas **MISD** (Múltiples flujos de Instrucciones y uno Solo de Datos), pero en la práctica han sido escasas y poco significativas las propuestas de máquinas con este tipo de arquitectura.

## 11.5. Procesadores vectoriales

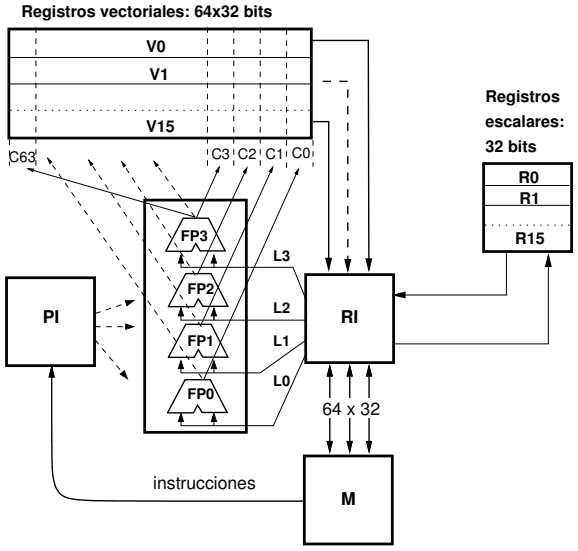



**Figura 11.9**: Componentes para un BRM vectorial.

Los **procesadores vectoriales** tienen una arquitectura SIMD con instrucciones y unidades aritméticas de coma fija y flotante para operar sobre vectores almacenados en grandes registros (cada uno puede contener entre 64 y 256 elementos de 32 o 64 bits). Las operaciones de transferencia entre la memoria y estos registros pueden hacerse en paralelo, gracias a una organización modular de la memoria. De este modo, operaciones que requieren un bucle en un procesador normal pueden realizarse con una sola instrucción en uno vectorial. Muchos de los llamados **supercomputadores**[4] están basados en un procesador vectorial.

Para concretar con un ejemplo ficticio, supongamos que al modelo estructural de BRM añadimos los componentes que muestra la figura 11.9. Además de los 16 registros normales («escalares») hay 16 **registros vectoriales**, V0, V1... V15. Cada uno de ellos puede contener un vector de 64 componentes, donde cada componente tiene 32 bits. Además de la UAL para el procesamiento normal de datos de 32 bits (no mostrada en la figura) tiene cuatro unidades de coma fija y flotante para operar sobre vectores. El modelo funcional se amplía con instrucciones para estas operaciones: `LDRV` (cargar vector), `STRV` (almacenar vector), `ADDVI` (sumar vectores en coma fija), `ADDVF` (sumar vectores en coma flotante), etc.

[4] «Supercomputador» no es lo mismo que «gran ordenador» (o *mainframe*). El primero tiene una arquitectura optimizada para la ejecución de cálculos matemáticos, mientras que el segundo está diseñado con otro objetivo: facilitar la multiprogramación y ejecutar programas que normalmente requieren más comunicaciones con los periféricos que cálculos. Lo que prima en un supercomputador es la *velocidad* (número de instrucciones ejecutadas por segundo), mientras que en un gran ordenador es la *potencia* (número de trabajos terminados por unidad de tiempo).

En la figura sólo se muestran los operadores para coma flotante. Los cuatro pueden funcionar en paralelo. La operación de sumar dos vectores, V1 y V2, y dejar el resultado en un tercero, V3, se podría realizar en 16 fases, trabajando simultáneamente en cada una con componentes de los vectores (Vi(Cj) es el componente Ci de Vj; cada componente son 32 bits):

1. FP0 suma V1(C0) y V2(C0) y deja el resultado en V3(C0);
   al mismo tiempo, FP1 suma V1(C1) y V2(C1) y deja el resultado en V3(C1);
   al mismo tiempo, FP2 suma V1(C2) y V2(C2) y deja el resultado en V3(C2);
   al mismo tiempo, FP3 suma V1(C3) y V2(C3) y deja el resultado en V3(C3)
2. FP0 suma V1(C4) y V2(C4) y deja el resultado en V3(C4);
   al mismo tiempo, FP1 suma V1(C5) y V2(C5) y deja el resultado en V3(C5);
   al mismo tiempo, FP2 suma V1(C6) y V2(C6) y deja el resultado en V3(C6);
   al mismo tiempo, FP3 suma V1(C7) y V2(C7) y deja el resultado en V3(C7)
3. ...

16. FP0 suma V1(C60) y V2(C60) y deja el resultado en V3(C60);
    al mismo tiempo, FP1 suma V1(C61) y V2(C61) y deja el resultado en V3(C61);
    al mismo tiempo, FP2 suma V1(C62) y V2(C62) y deja el resultado en V3(C62);
    al mismo tiempo, FP3 suma V1(C63) y V2(C63) y deja el resultado en V3(C63)

Cada una de estas fases puede ejecutarse en un ciclo de reloj. Obviamente, con 16 operadores, por ejemplo, serían cuatro fases en lugar de 16, y con 64 sólo sería una. Al no disponer de los operadores suficientes (64) para transportar simultáneamente todo el tráfico de 64×32 bits, se encauza este tráfico en **carriles** *(lanes)*. En el caso de la figura 11.9 hay cuatro carriles: L1, L2, L3 y L4.

Veamos cómo se programaría una operación vectorial típica: $\vec{V_3} = k \times \vec{V_1} + \vec{V_2}$, donde $\vec{V_1}$, $\vec{V_2}$ y $\vec{V_3}$ son vectores con 64 componentes y cada componente (32 bits) representa un número en coma flotante. Los vectores están almacenados en la memoria a partir de las direcciones apuntadas por R1, R2 y R3.

En primer lugar, sin los operadores ni los registros vectoriales, en BRM, suponiendo que tuviese las instrucciones para sumar y multiplicar en coma flotante, tendríamos que escribir este programa:

```
       mov   r0, #64     @ contador
       ldr   r4, =k
bucle: ldr   r5,[r1], #4
       mulf  r6,r4,r5
       ldr   r7,[r2], #4
       addf  r8,r6,r7
       str   r8,[r3],#4
       subs  r0,r0, #1
       bne   bucle
```

En total se ejecutan $2 + 7 \times 64 = 450$ instrucciones. Con la arquitectura vectorial el programa sería:

```
       ldr    r4, =k
       ldrv   v1,[r1]
       mulvf  v6,r4,v1
       ldrv   v7,[r2]
       addvf  v8,v6,v7
       strv   v8,[r3]
```

con el que solamente se ejecutan seis instrucciones. El tiempo de ejecución depende del número de carriles, pero aun cuando sólo hubiese uno (y, por tanto, las instrucciones aritméticas tardasen 64 ciclos) sería menor que con la versión no vectorizada.

En el apartado 11.8 veremos que el mismo principio de realizar operaciones en paralelo con datos manteniendo un solo flujo de instrucciones, es decir, SIMD, es aplicable al procesamiento de gráficos.

## 11.6. Multiprocesadores

Un multiprocesador es una arquitectura MIMD «fuertemente acoplada»: los procesadores comparten memoria, y, como son independientes, hacen posible un paralelismo completo en la ejecución de los procesos o las hebras (apartado 11.7).

Un **multiprocesador simétrico**, o **SMP** *(Symmetric MultiProcessor)*, es aquél en el que el tiempo de acceso a una dirección de memoria es el mismo para todos los procesadores. También se llama de **acceso uniforme a la memoria**, **UMA** *(Uniform Memory Access)*, para distinguirlo del caso general, **NUMA**, o **DSM** *(Distributed Shared Memory)*.

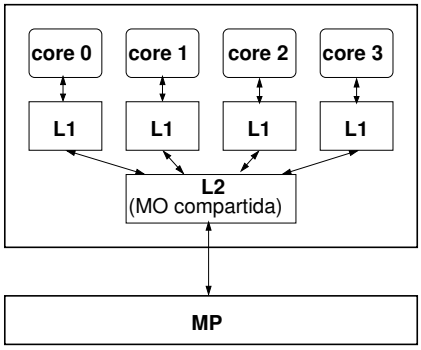



**Figura 11.10**: Chip *multicore*.

Un «chip multicore» es un multiprocesador integrado, normalmente del tipo SMP. Como hemos adelantado en el apartado 11.4, aunque comparten memoria, cada procesador tiene su propia memoria *cache*. En la figura 11.10 «L1» son *caches* de nivel *(level)* 1, privadas de cada procesador *(core)*, y «L2» es la de nivel 2, compartida. Cuando se combinan varios (o muchos) *multicore* para formar otro multiprocesador normalmente se conectan mediante una red de interconexión. El resultado es de tipo NUMA (figura 11.11), puesto que el acceso de un *multicore* a la memoria de otro pasa por la red de interconexión, y es más lento que el acceso a su propia memoria.

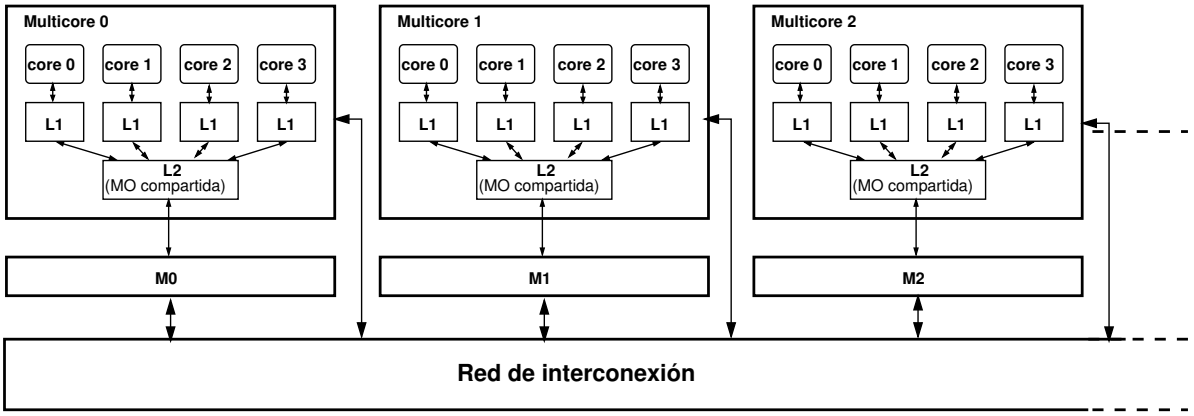



**Figura 11.11**: Multiprocesador con *multicores*.

Hay arquitecturas MIMD «**masivamente paralelas**», llamadas **MPP** *(Massively Parallel Procesors)* o **LSM** *(Large-Scale Multiprocessors)*, con cientos o miles de procesadores. «Masivamente paralelo» se aplica también a sistemas que no son propiamente multiprocesadores sino multiordenadores, como los *clusters* mencionados antes o el *grid computing*, en el que miles ordenadores dispersos geográficamente contribuyen a una tarea común (un ejemplo muy interesante es BIONIC, red de voluntarios que se está utilizando en más de 80 proyectos).

Aunque su estudio desborda lo que tradicionalmente se entiende por «arquitectura de ordenadores», conceptualmente cabe encuadrar también en la categoría de multiprocesadores a las **redes neuronales**, en las que la desviación del modelo SISD es radical: los elementos de procesamiento son también elementos de memoria, y entre todos forman una memoria global de tipo asociativo. Los modelos son

totalmente diferentes de los que venimos estudiando, y aunque es frecuente implementarlas mediante simulación (y se utilizan con éxito en aplicaciones como reconocimiento de caracteres manuscritos, de voz, de imágenes, etc.), se está avanzando en varios laboratorios para su implementación en hardware. Lo más interesante es que estos modelos permiten materializar de manera muy natural un tipo de **aprendizaje en la máquina**.

## 11.7. Procesadores multihebrados *(multithreading)*

Recapitulando lo visto hasta ahora sobre el paralelismo en los procesadores, hay tres tipos:

- Paralelismo a nivel de instrucciones (ILP), que se consigue con el encadenamiento.
- Paralelismo a nivel de datos (DLP), en las arquitecturas SIMD, por ejemplo, los procesadores vectoriales.
- Paralelismo a nivel de hebras (TLP), en las arquitecturas MIMD y, como vamos a ver, también en las otras.

Lo que caracteriza al TLP es que requiere un compromiso mayor del sofware. Aunque el hardware puede facilitar la tarea, es un planificador el que debe decidir en cada momento qué hebra pasa a ser activa en cada procesador, y esto es tarea, normalmente, del sistema operativo (no obstante, veremos más adelante que las modernas GPU incluyen planificadores implementados en hardware).

Pero no es necesaria la presencia de varios procesadores completos para que varias hebras se ejecuten concurrentemente. De la misma manera que un sistema operativo multiproceso (o multitarea) asigna en cada momento el procesador a alguno de los procesos preparados, también puede ir asignándolo a lo largo del tiempo a distintas hebras de distintos procesos. La conmutación de hebras es más ligera que la de procesos, y los procesadores multihebrados incluyen algunos componentes de hardware que agilizan esta conmutación.

Dejando aparte los multiprocesadores (cuyos procesadores pueden ser también multihebrados), hay dos formas de facilitar la concurrencia de hebras en un procesador:

- **Multihebrado entrelazado** *(interleaved multithreading)*, también llamado «multihebrado temporal», en el que el procesador se va asignando a distintas hebras a lo largo del tiempo. Dependiendo de la frecuencia con la que se conmuta de una hebra a otra hay dos tipos:
  - De grano grueso *(coarse-grain)*: la hebra en ejecución continúa hasta que termina o queda bloqueada (por una operación de entrada/salida, por un fracaso en el acceso a la *cache*, etc.) y el planificador activa a alguna de las preparadas. Es similar a la «planificación cooperativa» clásica de los sistemas operativos de multiprogramación (página 19). El hardware del procesador, si permite la concurrencia de dos hebras, tiene duplicados los registros (los visibles al programa y otros, como el contador de programa y el registro de estado), y conmuta automáticamente.
  - De grano fino *(fine-grain)*: la conmutación de una hebra a otra se produce automáticamente cada pocos ciclos de reloj, dando a cada hebra una «rodaja de tiempo». Es similar a la «planificación apropiativa» *(preemptive scheduling)*. Además de los registros, el hardware tiene que incluir elementos para identificar en cada etapa de la cadena la hebra a la que corresponde.

- **Multihebrado simultáneo** *(SMT: Simultaneous MultiThreading)* en el que varias hebras se ejecutan simultáneamente en el mismo *procesador escalar* (figura 11.3b). Aquí también el procesador tiene que incluir registros para cada hebra. Aparte de un mayor rendimiento tiene la ventaja de que la vista que ofrece al software es la de varios (normalmente dos en los microprocesadores actuales) *procesadores lógicos*, de modo que se adapta fácilmente a los sistemas operativos diseñados para multiprocesamiento simétrico (SMP). Aunque el principio ya se había aplicado en algunos diseños desde los años 1960, fue Intel quien lo introdujo en los microprocesadores y registró la tecnología con el nombre de «hyperthreading».

De todas las combinaciones posibles no está claro cuál es la mejor. AMD refina el SMT duplicando no sólo los registros, también algunas unidades funcionales de procesamiento de datos, y llama a esta técnica (de forma bastante equívoca) CMT *(Cluster MultiThreading)*. Por su parte, ARM utiliza también en sus diseños procesadores superescalares, pero sin SMT y sí con ejecución reordenada y especulativa.

## 11.8. Procesadores gráficos

Las aplicaciones multimedia se han beneficiado de los avances en paralelismo, y, al mismo tiempo, los han impulsado. Particularmente, los controladores de pantalla, que inicialmente se limitaban a generar las señales adecuadas para el monitor a partir de la información que la CPU había generado en una zona de la memoria llamada **memoria de fotograma** *(framebuffer)*, han ido asumiendo funciones de procesamiento (para descargar a la CPU) hasta llegar, actualmente, a convertirse en complejos procesadores paralelos, las **GPU** *(Graphical Processing Units)*.

Antes de llegar a las GPU conviene tener algunas ideas generales sobre la generación y presentación de gráficos y de algunas modificaciones que se han hecho a la arquitectura de los procesadores convencionales para apoyar las aplicaciones multimedia. Nos centraremos en el procesamiento de gráficos, aunque la mayoría de los subsistemas incorporan también funciones para la presentación de vídeos, particularmente, descodificadores de flujos MPEG.

### Controladores de gráficos

Los controladores de gráficos más comunes en los ordenadores de sobremesa y portátiles se llaman «integrados», no porque sus circuitos estén en el mismo chip que la CPU (que pueden estarlo o no), sino porque comparten con ella la memoria del sistema. Sus componentes principales son:

- **Memoria de fotograma.** La capacidad de memoria necesaria para un fotograma es $(R_h \times R_v \times n)/8$ bytes, donde $R_h$ y $R_v$ son las resoluciones (número de píxeles) horizontal y vertical y $n$ es la profundidad de color en bpp (apartado 6.6). En esta memoria el programa escribe los contenidos necesarios para presentar un fotograma y los circuitos de control del controlador la leen periódicamente y generan las señales adecuadas para el monitor. Una tecnología de memoria que se desarrolló para este modo de funcionamiento es la VRAM (RAM de vídeo), que tiene un puerto de escritura con acceso aleatorio y puertos de lectura formados por registros de desplazamiento de gran longitud (los bits suficientes para codificar una línea de la imagen). Los registros, uno para cada color básico, se cargan en paralelo y se leen en serie, generando así la secuencia de píxeles que se van dibujando. Actualmente se utilizan más las memorias SDRAM con encadenamiento.

- **Memorias internas.** La presentación directa de los contenidos de la memoria de fotograma al mismo tiempo que se está escribiendo en ella conduce a un efecto visual de parpadeo *(flickering)* inadmisible. Para evitarlo se recurre a la técnica de *double buffering*: se duplica la memoria de fotograma, F1 y F2. Mientras el procesador (la CPU o la GPU) escribe en F1 se envía a los puertos de lectura el contenido de F2. Al terminar de enviar el último píxel de la imagen los circuitos de control invierten las funciones: se envía a los puertos de lectura F2 y el procesador prepara la siguiente imagen en F1.
  Además, el controlador puede tener otras memorias para funciones auxiliares. Por ejemplo, una ROM que contiene los mapas de bits de los caracteres: la inserción de un texto en la imagen no necesita que se generen sus píxeles componentes cada vez que ha de presentarse un carácter, basta colocar los mapas de bits de cada uno de los caracteres en las coordenadas adecuadas. Similarmente, puede haber pequeñas RAM que contengan mapas de bits de fragmentos de imagen: con frecuencia, la diferencia entre un fotograma y el siguiente es solamente el desplazamiento de fragmentos pequeños, independientes del resto de la imagen. Se llaman *sprites*, que podemos traducir libremente como «monos» (recuerde, si tiene edad suficiente, el juego del «comecocos»).
- **Generadores de señal.** Para salida analógica (como VGA y vídeo compuesto o por componentes) se utiliza un «RAMDAC» *(Random Access Memory Digital-to-Analog Converter)*, que es una combinación de una pequeña memoria con tres conversores analógico-digitales. Para salida digital (como DVI y HDMI) la parte «DAC» no existe y se utiliza la tecnología «TMDS» *(Transition-Minimized Differential Signaling)*, concebida para la transmisión de datos a alta velocidad minimizando el ruido electromagnético.
- **Interfaz con el sistema.** La tendencia es a integrar el controlador en el mismo chip de la CPU; en este caso también se integra la memoria: se llama «eDRAM» *(embedded DRAM)*. El fabricante AMD introdujo el nombre **APU** *(Accelerated Processing Unit)* para los chips que integran una CPU y una GPU. Si el controlador, pese a ser «integrado», no está en el mismo circuito integrado de la CPU puede estar soldado en la placa base o en una tarjeta que se conecta mediante un bus de entrada/salida (antiguamente, ISA, luego PCI).
- **Circuitos de control.** En el caso más sencillo sólo generan las señales adecuadas para el funcionamiento de los demás componentes. A medida que ha ido avanzando la tecnología se han ido incorporando funciones de procesamiento, y resulta difícil determinar cuándo han pasado de ser meros «circuitos de control» a «acelerador de gráficos» y finalmente convertirse en una GPU.

Los controladores «dedicados» tienen su propia memoria de vídeo y se conectan al sistema (procesador y memoria) mediante un bus de alta velocidad: PCIe (figura 2.8) o AGP *(Accelerated Graphics Port)*. Actualmente se utilizan para aplicaciones de alto rendimiento, y, por tanto, lo que contienen es realmente una GPU.

## Extensiones multimedia

En el procesamiento de imágenes es común realizar una determinada operación sobre muchos datos. Por ejemplo, para cambiar el brillo de una imagen representada en la memoria como un *pixmap* (apartado 7.3) hay que leer de la memoria, píxel a píxel, los valores de las componentes de color (R, G, B), sumar o restar una cantidad a cada una y escribir el resultado en la memoria. La operación es mucho más rápida si en lugar de ejecutarla píxel a píxel secuencialmente disponemos instrucciones del tipo «leer n píxeles», «sumar k a los n píxeles», etc., y este tipo de instrucciones es característica del procesamiento SIMD.

La aplicación del principio de SIMD aquí es diferente a la que se hace en los procesadores vectoriales: no se trata de operar con vectores de muchas componentes con muchos bits, sino al contrario: los operandos (intensidad de cada color, valor de una muestra de sonido, etc.) suelen ser de 8 o 16 bits. Esto condujo a introducir en los microprocesadores algunos componentes nuevos para implementar ese paralelismo a nivel de datos pequeños *(subword paralelism)*.

Así, en 1997 la arquitectura de los microprocesadores de Intel incluía registros para coma flotante de 64 bits. La adición de pequeñas unidades aritméticas permitía reutilizarlos para operar en paralelo con ocho operandos de ocho bits, o cuatro de dieciséis, mediante nuevas instrucciones de suma, resta, máximo, mínimo, desplazamientos, etc. Intel las llamó **MMX** *(MultiMedia eXtensions)* y luego fue perfeccionando la idea: en 1999 **SSE** *(Streaming SIMD extension)* evitaba reusar los registros añadiendo otros especializados de 128 bits y 70 instrucciones nuevas (algunas pensadas para la decodificación de MPEG2) con sucesivas mejoras en número y longitud de los registros y en instrucciones (SSE2, SSE3, SSE4) y más recientemente (2011) **AVX** *(Advanced Vector Extensions)*.

Otros fabricantes de microprocesadores también han adoptado ese principio de dotar a la arquitectura del procesador de hardware e instrucciones para el paralelismo con datos pequeños: AMD con las extensiones «3DNow!», ARM con la tecnología «NEON», etc.

### GPU

La generación de imágenes realistas implica muchas más operaciones de las que pueden concebirse con «extensiones multimedia». Llegar a una materialización como una matriz de píxeles a partir de un modelo 3D exije una sucesión de actividades que se llama **cadena de generación** *(rendering pipeline)*. El modelo 3D es una representación de la escena, las fuentes de luz y la posición de la cámara, o punto de vista (o dos puntos de vista, para simular la sensación de relieve). La escena está compuesta por objetos. Cada objeto tiene una forma, una ubicación y un tamaño y se describe en su propio sistema de coordenadas como un conjunto de poliedros cuyos vértices no sólo están asociados con las tres coordenadas, sino con otros datos necesarios para visualizar correctamente el objeto: color, iluminación, propiedades de reflexión, textura, etc.

La cadena de generación consta de una secuencia de pasos: procesamiento de los vértices, transformaciones de los sistemas de coordenadas, sombreado, eliminación de caras ocultas, iluminación, tramado *(rasterización)*, procesamiento de píxeles, en cuyos detalles que no tiene sentido entrar aquí (es asunto propio de *computer graphics* no de *computer architecture*), pero del análisis de esos pasos se extraen conclusiones interesantes para su procesamiento:

- Se trata de una *cadena* de muchas transformaciones, y en cada una se realizan operaciones independientes de los otros pasos. Es natural aplicar operadores hardware encadenados.
- En cada paso, las operaciones se ejecutan idénticamente sobre muchas estructuras de datos. Es natural realizarlas mediante una arquitectura SIMD.
- Esas estructuras de datos son realmente numerosas: representan cientos de miles o millones de poliedros, vértices, píxeles, etc. Como no hay tantos procesadores de datos, se agrupan en carriles *(lanes)* como en los procesadores vectoriales, y el proceso se descompone en hebras, una para cada carril.
- Las sucesivas instrucciones de máquina necesarias para la generación pueden ejecutarse en paralelo. Es natural aplicar una arquitectura MIMD.

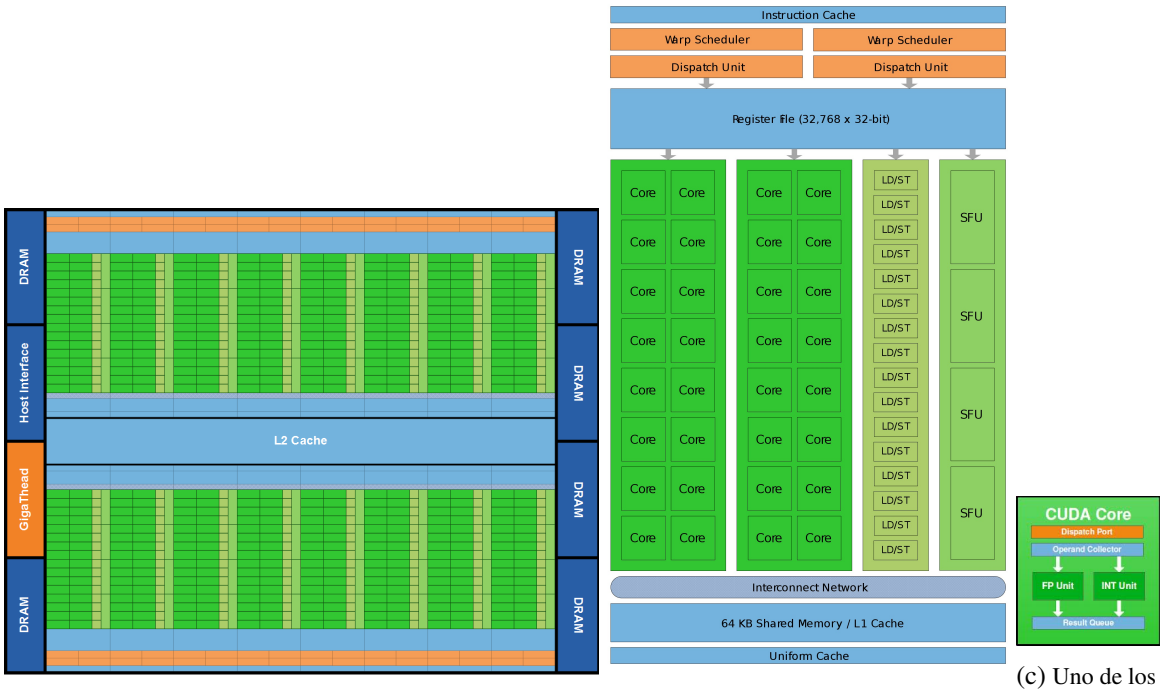
El resultado es que en una GPU aparecen combinados todos los tipos de paralelismo: a nivel de instrucción (ILP), a nivel de datos (DLP, SIMD), a nivel de hebras (TLP) y multiprocesador (MIMD).

Aunque la GPU, según hemos visto, es realmente, una parte del controlador gráfico, esta parte se ha desarrollado de tal modo que es, con mucho, la dominante en recursos (transistores) del controlador.

Los fabricantes de microprocesadores que integran la GPU con la CPU (fundamentalmente, Intel con «HD Graphics» y AMD, con «Accelerated Processing Units») no proporcionan más que los detalles imprescindibles sobre su arquitectura y su implementación. Se extienden, naturalmente, en los aspectos de rendimiento. Lo mismo ocurre con los fabricantes de SoC. Por ejemplo, la «Raspberry Pi» utiliza un chip de Broadcom (BCM2835) que integra tres procesadores: un ARM, una GPU «VideoCore IV» y un procesador especializado para varias etapas de la cadena de generación. De la GPU se conoce poco: Broadcom sólo facilita información a compañías que tengan un mercado grande, y bajo un acuerdo de confidencialidad (NDA: *Non-Disclosure Agreement*)[5].

Los fabricantes de GPU dedicadas, que son más programables y se aplican no solamente al procesamiento gráfico, sí ofrecen más información. Los principales son Nvidia, con una variedad de productos, fundamentalmente las tarjetas de las series «GeForce» y «Quadro» y AMD con productos similares en funcionalidad (pero distintas arquitecturas), las tarjetas «Radeon» y «FireStream»

Para dar una idea de la complejidad de las GPU dedicadas actuales, la figura 11.12a muestra un esquema de la distribución de componentes en la GeForce GTX 480 de Nvidia, una implementación de la «arquitectura Fermi», realizada con más de 3.000 millones de transistores. Contiene 16 procesadores SIMD multihebrados que en la terminología de Nvidia se llaman «SM» *(streaming multiprocessors)*. Comparten una memoria *cache* común L2 (cada SM tiene una L1 propia), hasta seis módulos de memoria DRAM de 64 bits y 1 GiB, un «GigaThread» (terminología de Nvidia para un planificador de bloque de hebras) y una interfaz para el bus PCIe.




(a) Distribución de componentes (b) Uno de los 16 *streaming processors* (c) Uno de los 512 *cores*

**Figura 11.12**: Implementación en la tarjeta GTX 480 de la arquitectura Fermi.
Fuente: `http://www.nvidia.com`

[5]Hay un grupo de desarrolladores trabajando en ingeniería inversa para obtener el máximo partido de la Raspberry Pi: `https://github.com/hermanhermitage/videocoreiv`.

Cada uno de los 16 SM contiene 32 *cores*, como muestra la figura 11.12b (el número total de *cores* es, por tanto, 16×32 = 512, aunque en las implementaciones en tarjetas concretas puede ser menor). Cada *core* tiene una unidad de coma flotante y otra de coma fija de 64 bits, ambas con encadenamiento, los circuitos necesarios para distribuir los operandos a la unidad oportuna y una cola para guardar resultados (figura 11.12c). Los operandos y los resultados se encuentran en una memoria local de 32.768 registros de 32 bits. LD/ST son los circuitos de acceso a la *cache* L1. «SFU» *(Special Function Units)* son operadores para funciones trigonométricas, raíces cuadradas, logaritmos, etc.

Para gestionar y ejecutar cientos de hebras éstas se organizan jerárquicamente. El programa de aplicación, o el compilador, agrupa las hebras en grupos *(grids)* y bloques *(blocks)*. Un bloque es un conjunto de hebras que pueden cooperar mediante el acceso a una memoria privada del bloque. Un grupo es un conjunto de bloques que pueden ejecutarse independientemente. Esta jerarquía se empareja con la jerarquía de procesadores en la GPU: La GPU ejecuta uno o más grupos; el «GigaThread» planifica y distribuye estos grupos (figura 11.12a). Cada SM ejecuta uno o más bloques; en la arquitectura Fermi los SM son duales (figura 11.12b): pueden ejecutar concurrentemente dos bloques mediante los correspondientes planificadores, que trabajan sobre conjuntos de 32 hebras llamados *wraps*. Y los *cores* y las SFU ejecutan las hebras.

Es interesante observar que es la GPU, no las aplicaciones ni el sistema operativo, quien se ocupa de la ejecución paralela y de la gestión de hebras.

La arquitectura del conjunto de instrucciones (ISA) se llama «PTX» *(Parallel Thread eXecution)* y está diseñada para la ejecución concurrente y superescalar de hebras. Tiene instrucciones típicas de RISC: aritmética entera y de coma flotante en doble precisión sobre registros, conversiones, transferencia de control, acceso a la memoria y otras especializadas para procesamiento de gráficos. Un detalle es que, como en ARM, todas las instrucciones son condicionadas.

| Año | Arquitectura | *Cores* | Transistores | Tecnología |
|---|---|---|---|---|
| 2008 | Tesla | 240 | 1.400 millones | 65 nm |
| 2010 | Fermi | 512 | 3.000 millones | 40 nm |
| 2012 | Kepler | 1.400 | 7.000 millones | 28 nm |
| 2014 | Maxwell | 2.880 | 5.000 millones | 20 nm |
| 2016? | Pascal | | | |

**Tabla 11.1**: Algunos datos de las GPU de Nvidia.

Nvidia introduce una nueva generación de GPU cada dos años. Cada generación da lugar a la implementación de una serie de productos (tarjetas). La tabla 11.1 resume algunos datos aproximados, aunque hay que tener en cuenta que, como cada arquitectura se materializa en varios productos, los datos son los teóricamente alcanzables. Por ejemplo, de la arquitectura Maxwell existen varias tarjetas en el mercado, pero aún se fabrican con tecnología de 28 nm. En particular es interesante observar que se sigue cumpliendo la ley de Moore, al menos hasta la arquitectura Kepler. La reducción en el número de transistores de Maxwell (el dato concreto se refiere a una GPU, la GM204) no la desmiente del todo: la superficie de chip se ha reducido de 550 $mm^2$ a 400 $mm^2$.

## Implicaciones en el software

Las primeras GPU eran configurables pero no programables. Las actuales tienen, como hemos visto en el ejemplo anterior, una arquitectura ISA bien definida. Pero si ya es difícil programar en bajo nivel un procesador normal, intentarlo con una GPU sería un disparate. Se utilizan dos herramientas para la programación de aplicaciones gráficas: interfaces de programación (API) y lenguajes de alto nivel.

Las API gráficas son capas de abstracción del hardware. Hay dos competidoras: **OpenGL**, que es un estándar abierto y **Direct3D**, de Microsoft. Ambas definen una cadena de generación lógica (procesamiento de vértices, procesamiento geométrico, etc.) que se hace corresponder en cada GPU con los

componentes de hardware, así como modelos de programación y lenguajes para las distintas etapas de la cadena. Los fabricantes de GPU facilitan gestores *(drivers)* para ambas cadenas. Direct3D se utiliza más para la programación de juegos; OpenGL para aplicaciones profesionales (Adobe Photoshop, Blender, Google Earth, etc.)

Los lenguajes de alto nivel más conocidos están basados en C: **Cg** *(C for graphics)* de Nvidia, **HLSL** *(High-Level Shading Language)*, de Microsoft, y **GLSL** *(OpenGL Shading Language)*. El compilador de Cg puede generar programas que cumplen las especificaciones de OpenGL o de Direct3D. Los otros dos generan código para sus correspondientes API.

### GPGPU y computación GPU

En los primeros años 2000, cuando ya la capacidad de procesamiento y la facilidad de programación de las GPU era notable, se empezó a pensar en la explotación de este hardware para resolver problemas que se adaptaban a una computación en paralelo: cálculo científico, dinámica molecular, plegamiento de proteínas, imágenes médicas, algoritmos de aprendizaje, redes neuronales... En general, todas aquellas que necesitan procesar en paralelo muchos conjuntos de datos con pocas dependencias entre esos datos. Se acuñó entonces el acrónimo **GPGPU** *(General-Purpose computing on GPU)*.

El desarrollo de estas aplicaciones, aunque factible, era laborioso porque utilizaban herramientas (OpenGL, Cg, etc.) concebidas para la cadena de generación gráfica: no es fácil resolver problemas generales expresándolos en términos de vértices, polígonos, transformaciones de espacios, etc.

A partir de 2006 aparecieron nuevas herramientas adaptadas a la tarea de programación paralela genérica, y actualmente se prefiere la expresión **GPU Computing** para diferenciarse de esa primera etapa de GPGPU. Las herramientas más utilizadas son:

- **CUDA** *(Compute Unified Device Architecture)*, un lenguaje, compilador y bibliotecas desarrollados por Nvidia para facilitar la programación. El procesador de CUDA genera programas en C y C++ para la CPU y para la GPU.
- **CAL** *(Compute Abstraction Layer)*, lenguaje de nivel ensamblador para las GPU de AMD.
- **OpenCL** *(Open Computing Language)*, de código abierto, que tiene características similares a CUDA, pero que no está ligado a ningún fabricante.

Un ejemplo concreto de aplicación cuyos resultados ilustran claramente la potencialidad de la computación GPU es la generación de imágenes de tomosíntesis digital. Se trata de una técnica de mamografía (también llamada «mamografía 3D») que se suele realizar al mismo tiempo y en el mismo escáner que una mamografía convencional (2D) pero que puede llegar a desplazarla: aparte de mejor visualización y generar menos falsos positivos, requiere menos presión y menos dosis de radiación.

Observe que, aunque se trata de generar una imagen, el problema no tiene que ver con la cadena de generación a la que nos venimos refiriendo, que está concebida para juegos, animaciones, simulaciones, etc.

Aunque el concepto es antiguo, no fue hasta los años 1990 que se dispuso de detectores digitales suficientemente sensibles para hacerlo realizable. Un centro pionero en la técnica es el Massachusetts General Hospital. Inicialmente, la potencia de cálculo necesaria conducía a unos retrasos inadmisibles para obtener resultados (a menos que se utilizase un supercomputador, lo que no parece razonable en la práctica clínica). Los primeros ensayos en un ordenador de sobremesa tardaban cinco horas en procesar los datos de un paciente. Experimentando con un *cluster* de 34 ordenadores el tiempo se reducía a 20 minutos, pero tampoco parecía factible utilizar este equipamiento en un servicio de radiología. En 2006,

con un ordenador de sobremesa y una GPU Nvidia Quadro FX se consiguió reducir el tiempo a cinco minutos. En 2011, con una GTX 285 se obtenía en unos dos segundos. Ese mismo año la FDA aprobaba el primer equipo: Selenia Dimensions 3D System, de la empresa Hologic.

# Capítulo 12

# Procesadores sofware

Sobre el último Tema del programa, el 4, la Guía de aprendizaje enuncia estos resultados:

- Conocer los niveles y tipos de lenguajes de programación.
- Conocer los procesadores de lenguajes.

Hasta ahora hemos tratado de los principios de los procesadores hardware. Decíamos en el apartado 1.5 que los procesadores software son aquellos programas que transforman unos datos de entrada en unos datos de salida. Pero esto es lo que hacen prácticamente todos los programas. Nos referimos aquí a los programas que sirven para procesar programas escritos en lenguajes simbólicos.

En el apartado 1.6 introducíamos las ideas básicas de los lenguajes simbólicos de bajo y alto nivel. Decíamos allí que los programas fuente escritos en estos lenguajes necesitan ser traducidos al lenguaje de máquina del procesador antes de poderse ejecutar. Recuerde, en particular, que el traductor de un lenguaje de bajo nivel, o lenguaje ensamblador, se llama **ensamblador**, y el de uno de alto nivel, **compilador**.

Pero 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 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 la asignatura «Fundamentos de programación» ha estudiado usted un lenguaje compilado, Java. En este capítulo estudiaremos los principios de los procesos de traducción, montaje e interpretación de una manera general (independiente del lenguaje). En el capítulo siguiente veremos con detalle la estructura y el funcionamiento de un procesador software concreto: la máquina virtual Java, que interpreta los programas que previamente han sido traducidos a un lenguaje intermedio *(bytecodes)* por un compilador de Java.

## 12.1. 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 10.2 (página 170):

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 9.5. Como no tiene condición (es `MOV`, no `MOVEQ`, ni `MOVNE`, etc.), los bits de condición, según la tabla 9.1, son 1110 = 0xE. El operando es inmediato, por lo que el bit 25 es `I = 1`, y el código de operación de `MOV` (tabla 9.2) 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 10.2: `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 9.10, 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 12.1**: Tabla de símbolos para el programa 10.2.

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

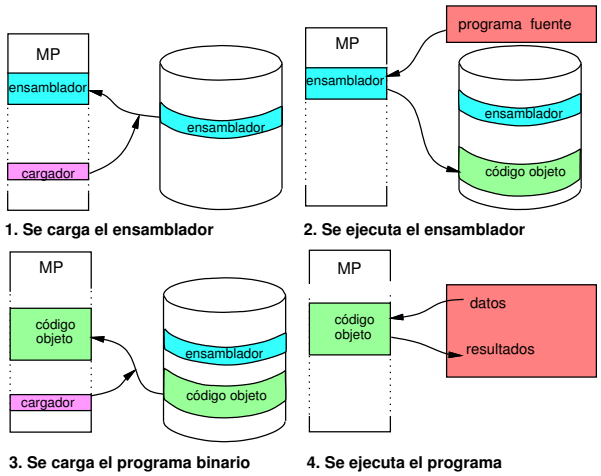



**Figura 12.1**: Procesos para el ensamblaje y la ejecución (incompleto).

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 operaciones que puede usted imaginar fácilmente, como las comprobaciones de errores sintácticos en el primer paso (códigos de operación que no existen, instrucciones mal formadas, etc.).

Es importante tener una idea clara de la secuencia de actividades necesarias para ejecutar un programa escrito en lenguaje ensamblador. La figura 12.1 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»).

Los cuatro pasos que muestra la figura son:

1. Se carga el programa ensamblador mediante la ejecución de otro programa, el cargador, que tiene que estar previamente en la memoria[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.
3. Se carga el código objeto en la memoria. 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.

¿Qué es lo que falta? Ya lo hemos avanzado en el apartado 10.4: 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.

## 12.2. 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 (o que no esté definido en el módulo), 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 10.7. Probablemente no habrá usted reparado en la curiosa forma que ha tenido el ensamblador de traducir la instrucción `bl fact` del programa 10.10 (felicitaciones si lo ha hecho): `0xEBFFFFFE`. Los ocho bits más significativos, 0xEB = 0b11101011, corresponden efectivamente, según el formato de la figura 9.10, 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 + $(-4 \times 2)$ = 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 no figura en el módulo, 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 10.11 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 10.11 (a partir de la dirección 0) y a continuación el 10.10 (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.

[1] Si el cargador está en la memoria, tiene que haber sido cargado. Recuerde del capítulo 2 (página 54) que este círculo vicioso se rompe gracias al cargador inicial *(bootstrap loader)*.

Por si analizando los detalles ha perdido usted la idea general, la figura 12.2 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`.

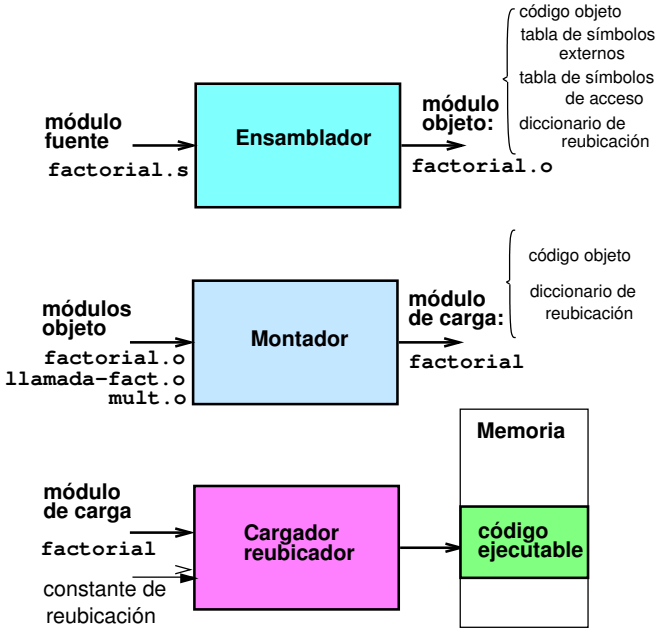



**Figura 12.2**: Procesos para el ensamblaje, el montaje y la ejecución.

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, apartado 2.5). 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, sin embargo, el programa 10.3. 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 10.3. 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 10.3.

## 12.3. Tiempos de traducción, de carga y de ejecución

Los procesos de traducción, montaje, carga y ejecución se realizan secuencialmente. Hay, pues, cuatro «tiempos», y es importante saber lo que se hace en cada uno: muchos errores de programación, tanto en bajo como en alto nivel, son consecuencia de su ignorancia.

- Las operaciones en *tiempo de traducción* las realiza el ensamblador o compilador una sola vez. Por ejemplo, la asignación de direcciones de memoria a las etiquetas que aparecen en el programa fuente.
- Las operaciones en *tiempo de montaje* las realiza el montador. Por ejemplo, la asignación de direcciones a símbolos externos.
- Las operaciones en *tiempo de carga* las realiza el cargador. Por ejemplo, la introducción de valores definidos en «`.data`»), también una sola vez (siempre que se carga el programa).
- Las operaciones en *tiempo de ejecución* se realizan cuando se ejecutan las instrucciones.

La figura 12.3 resume gráficamente esta secuencia de procesos y añade algunos detalles: El montador recibe como entradas los módulos objeto (cada uno de éstos incluye el código objeto, las tablas de símbolos y el diccionario) generados por el traductor, y también **módulos de biblioteca**. Éstos son programas o rutinas de uso general, residentes en disco, de los que pueden hacer uso los demás programas. Como indica la figura, los módulos de biblioteca los puede incorporar el montador o el cargador. La salida del cargador es ya un programa binario ejecutable. Para reducir su tamaño, ciertas funciones de biblioteca pueden cargarse dinámicamente, es decir, en tiempo de ejecución (cuando el programa las reclama).

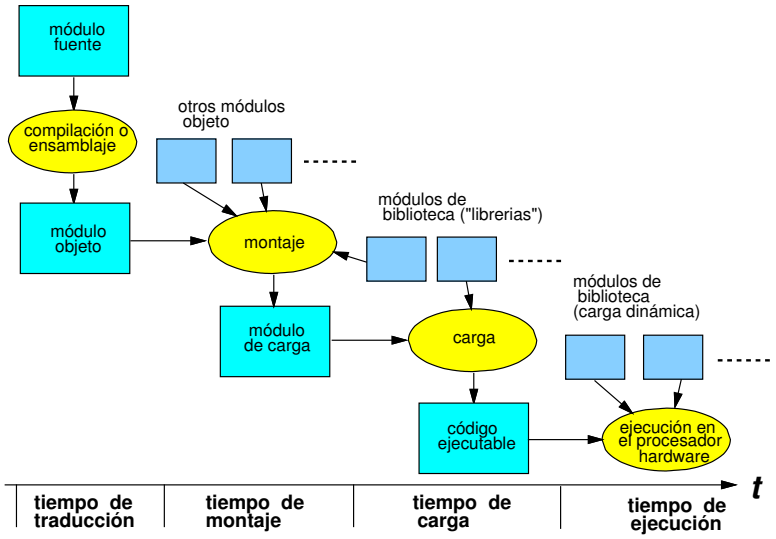



**Figura 12.3**: Traducción, carga y ejecución.

## 12.4. 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 10.5) 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 12.4.

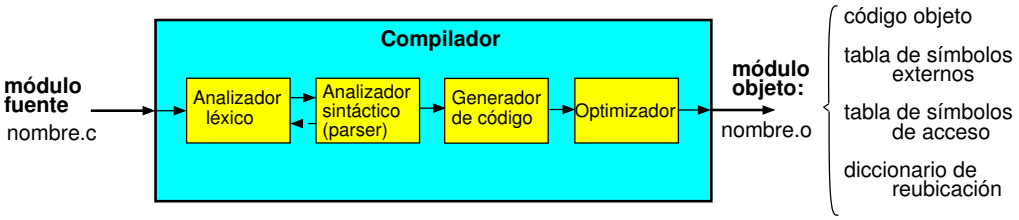



**Figura 12.4**: Componentes de un compilador.

Sin entrar en los detalles de implementación, veamos las funciones esenciales de estas fases y estos componentes.

### 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 12.5, que corresponden a las tres reglas anteriores.

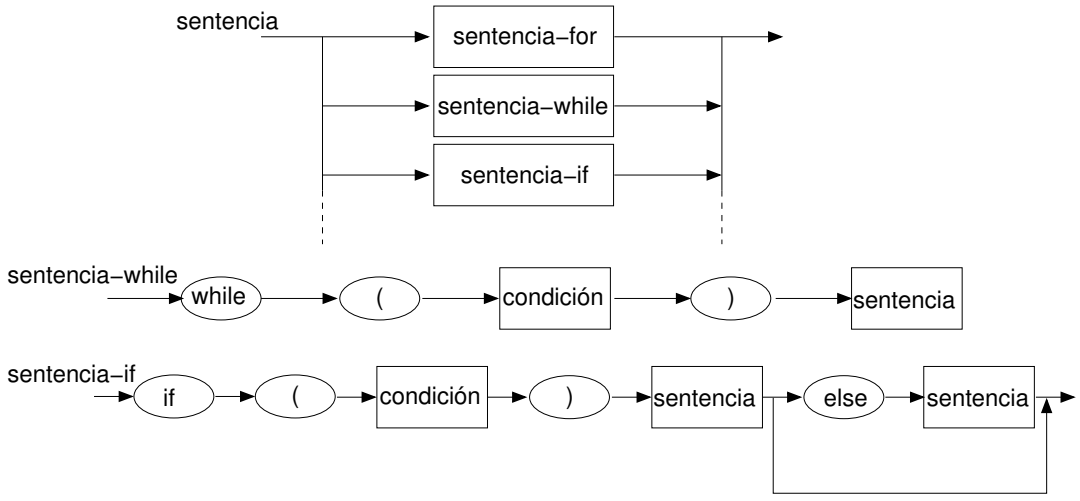



**Figura 12.5**: Diagramas sintácticos.

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 12.6 presenta el correspondiente a la función MCD. Esta representación gráfica es para «consumo humano». Realmente, se implementa como una estructura de datos.

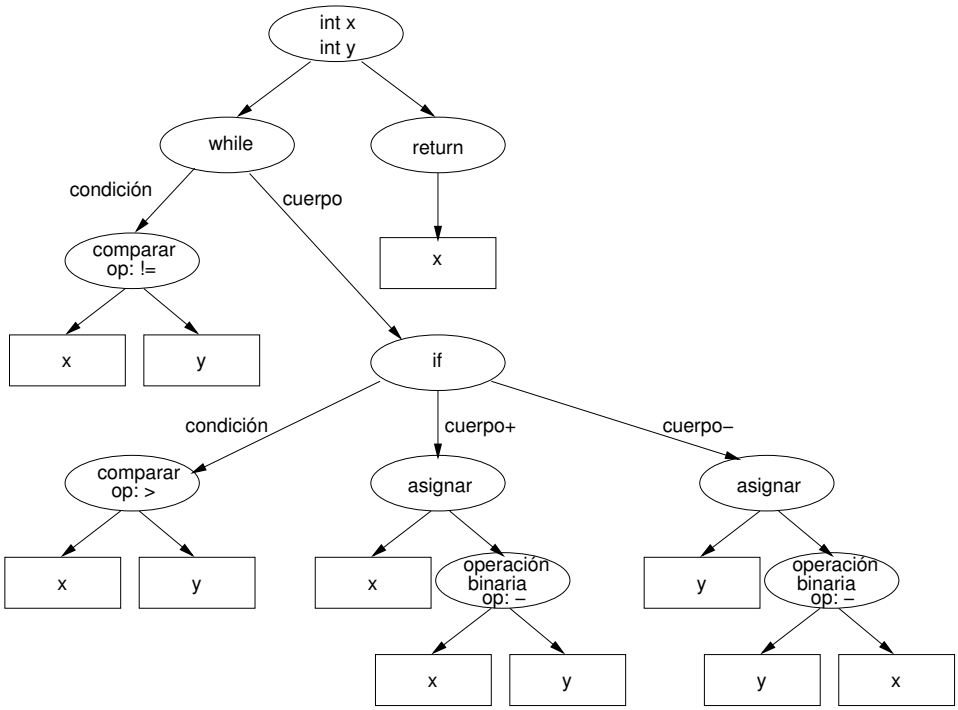



**Figura 12.6**: Árbol sintáctico de la función MCD.

## 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 10.6) 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 escrita en lenguaje C (con el código dado al principio de este apartado) 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 10.5. Pidiéndole al

compilador que optimice y que entregue 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 10.5. 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 procesador de Java, lenguaje que usted conoce de la asignatura «Fundamentos de 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, como veremos enseguida (apartado 12.6) normalmente no se implementa en hardware. El programa traducido a ese lenguaje (llamado *«bytecodes»*) normalmente se ejecuta mediante un programa intérprete (o bien se ejecuta directamente en una implementación hardware o firmware de la JVM, apartado 13.6).
- 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 12.7): así como el *parser* le va pidiendo *tokens* al analizador léxico, el ejecutor le va pidiendo sentencias al *parser*.

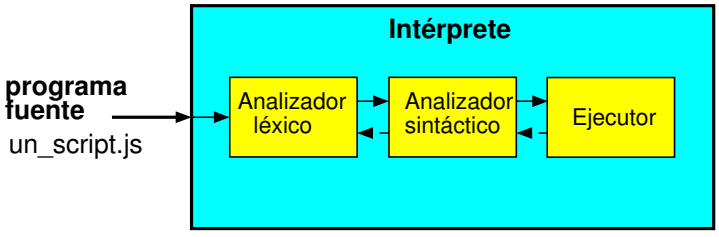



**Figura 12.7**: Componentes de un intérprete.

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

El modelo procesal de un intérprete puede resumirse en este pseudocódigo:

```
do {
  leer(S)
  explorar(S)  // análisis léxico
  analizar(S)  // análisis sintáctico
  ejecutar(S)
while (S no sea fin)
```

- la función `ejecutar(S)` implica, además de la traducción de la sentencia `S` a las instrucciones de máquina y de su ejecución, determinar la siguiente sentencia a interpretar, y
- lo normal es que la función `analizar(S)` llame repetidamente a `explorar(S)`, y que `ejecutar(S)` lo haga con `analizar(S)`.

Por ejemplo, en JavaScript la sintaxis de las sentencias `while` e `if` es la misma de C y de Java. Para la función MCD, el *parser* irá generando el mismo árbol sintáctico de la figura 12.6, pero a medida que se lo pide el ejecutor. Cuando éste 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.

## 12.5. Compilación JIT y compilación adaptativa

La ejecución de un programa mediante interpretación tiene varias ventajas sobre la compilación:

- El programa puede distribuirse en código fuente (para los preocupados por la propiedad intelectual, hay técnicas de ofuscación y cifrado) y ejecutarse en cualquier sistema (siempre que disponga del intérprete adecuado). Sin embargo, si se distribuye el código binario compilado sólo es ejecutable en máquinas que tengan la arquitectura para la que ha sido compilado.
- Durante el desarrollo, en el caso de la compilación, cualquier modificación del programa fuente requiere volver a compilar y ejecutar el nuevo código binario, mientras que con la interpretación basta con volver a ejecutar el intérprete.

Sin embargo, la interpretación tiene un inconveniente frente a la compilación, y es el tiempo de ejecución. Imagine un programa que contiene un bucle dentro del cual hay sentencias. Si se compila, estas sentencias se traducen una sola vez a binario, pero si se interpreta, las sentencias tienen que interpretarse cada vez que se pasa por ellas.

Recuerde la figura 12.3. En el caso de la interpretación no existen los tiempos de traducción, montaje y carga, pero la duración del tiempo de ejecución puede ser mucho mayor que con la compilación.

Una técnica más eficiente combina los dos procesos (traducción e interpretación). Es la **compilación justo a tiempo**, o **compilación JIT** *(Just-In-Time)*. En el bucle de interpretación, la primera vez que se ejecuta una parte del programa (en Java, un método) un compilador traduce los *bytecodes* al lenguaje de máquina del ordenador, y este código binario «nativo» se guarda en memoria, para que su posterior ejecución sea mucho más rápida. Observe que se trata de una *compilación dinámica*, que se realiza en la máquina en la que se está ejecutando el programa, a diferencia de la *compilación estática* tradicional (para lenguajes como C, C++, etc.), que se hace en la máquina en la que se desarrolla la

aplicación dando lugar a un código ejecutable «nativo» que sólo puede ejecutarse en una determinada plataforma.

Un inconveniente de esta técnica es que requiere más memoria. Y otro, que la compilación también requiere tiempo (y ahora es tiempo de ejecución), y si el código se ejecuta una sola vez o pocas veces puede ocurrir que el tiempo total sea superior al de la simple interpretación.

En la mayoría de los programas el tiempo se gasta ejecutando repetidamente ciertas partes del código, mientras que otras partes se ejecutan una o pocas veces. Esto conduce a la idea de la **compilación adaptativa**: a medida que el programa se ejecuta el procesador va observando las partes (subprogramas, o métodos) que se usan más de unas pocas veces, y sólo compila y optimiza esas partes, que se llaman *hot spots*; las demás quedan en su forma original. El resultado es que el procesador está del orden del 90 % del tiempo ejecutando binario (los *hot spots*), mientras que sólo ha sido necesario compilar y optimizar del orden del 10 % del programa fuente (la parte correspondiente a los *hot spots*). Los tiempos de ejecución que se consiguen son ya comparables a los de programas nativos (compilados para una máquina específica).

Actualmente, y en particular al referirse a la implementación de la máquina virtual Java, al decir «compilación JIT» se sobreentiende que es adaptativa.

## 12.6. Virtualización, emulación y máquinas virtuales

El concepto de virtualización, que podemos definir como «crear con apariencia de real algo que no lo es», se aplica en distintos niveles de la jerarquía de abstracciones. Por ejemplo, en el capítulo 2 hemos comentado la virtualización en el nivel de máquina operativa (página 20), una técnica que permite que uno o varios sistemas operativos se ejecuten sobre otro sistema operativo diferente. Y también estudiamos la *memoria virtual* (página 40), otra técnica que permite, en el nivel de máquina convencional, trabajar con una memoria principal de mucha mayor capacidad de la que realmente existe.

### Emulación

En este contexto, la emulación es una forma de virtualización. El término es más antiguo: se acuñó en los años 1960 para referirse a cambiar el comportamiento, mediante microprogramas (firmware, apartado 8.7), de un procesador de modo que su modelo funcional fuese el de otro, y así poder reutilizar todo el software desarrollado para un ordenador en otro nuevo[3]. La diferencia con «simulación» es que ésta se utiliza con propósitos de análisis de un sistema, mientras que la emulación tiene el objetivo de reemplazar el original para un uso real.

Actualmente es la misma idea, pero realizada mediante software. Aunque también se aplica a otros dispositivos (hay emuladores de terminales, de impresoras, de consolas de videojuegos, etc.), aquí nos referimos a emuladores de procesadores programables. Es decir, un emulador es un programa que se ejecuta sobre una arquitectura (ISA, apartado 1.7) para convertirla, virtualmente, en otra arquitectura diferente. De este modo, si tenemos un programa compilado, por ejemplo, para la arquitectura ARM, mediante un emulador podemos ejecutarlo en un procesador con arquitectura x86. Naturalmente, la ejecución en la arquitectura emulada será más lenta que en la original.

Es posible que la máquina emulada no exista físicamente, y entonces decimos que es una máquina virtual. La emulación suele implementarse mediante un proceso de interpretación, y, conceptualmente,

[3] Concretamente, fue cuando IBM introdujo en el mercado los ordenadores de la serie «IBM/360». Su modelo funcional era diferente a ordenadores anteriores, para los cuales había ya muchas aplicaciones desarrolladas. La emulación permitía ejecutar en los nuevos esas aplicaciones sin necesidad de volverlas a compilar.

muchos lenguajes interpretados definen máquinas virtuales. Podemos, por ejemplo, decir que un intérprete de JavaScript define la máquina virtual JavaScript, cuyo lenguaje de máquina estaría definido por el lenguaje JavaScript.

Estas máquinas virtuales definidas mediante emulación son arquitecturas ISA (nivel de máquina convencional). Pero el concepto de *máquina virtual* es más general. Se refiere a la implementación en software de uno o varios sistemas operativos o procesadores hardware sobre otro sistema u otro procesador. En el caso de virtualizar sistemas operativos se habla de **máquinas virtuales de sistema**, y si se trata procesadores, de **máquinas virtuales de proceso**.

## Máquinas virtuales de sistema

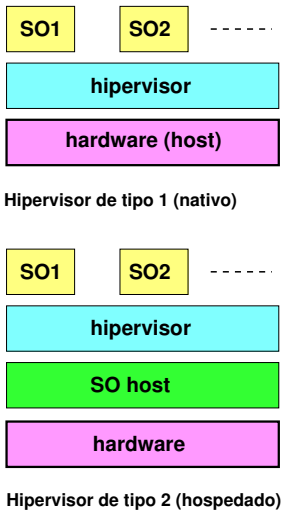



**Figura 12.8**: Tipos de hipervisores

El software que sirve para implementar máquinas virtuales de sistema se llama **hipervisor**[4], o **VMM** *(Virtual Machine Monitor)*[5]. El sistema sobre el que se ejecuta el hipervisor es el **sistema anfitrión** *(host)*, y el sistema virtualizado es el **sistema huésped** *(guest)*. Aunque la clasificación no es perfecta, se distinguen dos tipos de hipervisores (figura 12.8):

- Los del **Tipo 1**, también llamados «nativos», o «sobre máquina desnuda» *(bare metal)*, se ejecutan directamente sobre un hardware (procesador o procesadores, memorias y periféricos), permitiendo que varios sistemas operativos huésped se ejecuten concurrentemente compartiendo recursos. Ejemplos comerciales de este tipo son Oracle VM Server, VMware ESX y Microsoft Hyper-V. En software libre, el más conocido es Xen, de la Universidad de Cambridge. Todos ellos permiten virtualizar varios sistemas operativos sobre hardware de naturaleza muy diversa.
- Los del **Tipo 2**, también llamados «hospedados» *(hosted)*, se ejecutan sobre un sistema operativo anfitrión. Estos hipervisores son los que se utilizan en los ordenadores personales para poder cambiar rápidamente de un sistema operativo a otro sin tener que rearrancar. Ejemplos: VMware (comercial, aunque con una versión gratuita, VMware Player), VirtualBox (software libre) y QEMU (también libre). Este último tiene dos modos de funcionamiento: en *User mode emulation* se puede ejecutar un programa escrito para un procesador en un sistema operativo que se ejecuta sobre otro procesador (es decir, implementa máquinas virtuales de proceso), mientras que en *System mode emulation* permite emular no sólo procesadores, sino también periféricos, e implementar así máquinas virtuales de sistema.

Normalmente los hipervisores del segundo tipo (salvo QEMU) no incluyen emulación, es decir tanto el anfitrión como el o los huéspedes tienen la misma arquitectura ISA.

En general, la virtualización de sistemas tiene un inconveniente obvio: dado un hardware y una aplicación concreta, esta aplicación siempre se ejecutará con mayor eficiencia sobre un sistema operativo diseñado específicamente para ese hardware que sobre uno virtualizado. Sin embargo, sus ventajas en reducción de costes para las organizaciones (y en comodidad para los usuarios de ordenadores personales) la hacen una solución muy extendida (tanto, que a veces se escribe con el numerónimo «v12n»).

[4] Recuerde que al *kernel* de un sistema operativo ordinario también se le llama «supervisor».
[5] «Virtual Machine Manager» es un producto concreto para un hipervisor del Tipo 2 (`http://virt-manager.org/`).

### Máquinas virtuales de proceso

El objetivo de las máquinas virtuales de proceso es diferente. Normalmente, se trata de arquitecturas abstractas que se implementan con un intérprete específico para cada arquitectura real. Es decir, son lo que antes hemos llamado emuladores.

En el siguiente capítulo veremos la arquitectura de una máquina virtual de proceso, la JVM, una máquina de pilas (apartado 8.8) que normalmente se implementa en software (aunque partes de ella pueden implementarse en hardware, apartado 13.6) mediante un intérprete con compilación JIT.

Otras máquinas virtuales de proceso son:

- La máquina que interpreta CLI (Common Language Infrastructure) para C# y otros lenguajes compatibles con el entorno «.NET»
- Parrot, máquina de registros, para los lenguajes Perl y Python
- Dalvick, otra máquina de registros, integrada en el sistema operativo Android, también para el lenguaje Java y diseñada como alternativa a la JVM para sistemas con recursos de hardware limitados.

### El caso de Wine

No deben entenderse las anteriores definiciones de una manera rígida. La terminología cambia de unos autores a otros, y hay muchos matices. Por poner un ejemplo paradigmático, terminemos con el caso de Wine.

Wine es una aplicación muy popular en Linux con la que un programa compilado para Windows puede ejecutarse en un sistema de tipo Unix. Inicialmente se presentó como «Windows emulator». Sin embargo, ahora los desarrolladores dicen que es un acrónimo recursivo: «*Wine Is Not an Emulator*». ¿Es o no es un emulador?

Si «emular» significa «imitar las acciones de otro procurando igualarlas e incluso excederlas» (R.A.E.), Wine *sí emula* a Windows (de hecho, para algunas aplicaciones, especialmente las desarrolladas para versiones más antiguas de Windows, tiene mejor rendimiento que el propio Windows).

Pero la definición que hemos dado antes dice que un emulador es «un programa que se ejecuta sobre una arquitectura ISA para convertirla, virtualmente, en otra arquitectura diferente». Hemos dicho lo que hace Wine (modelo funcional). Pero ¿como lo hace (modelo procesal)? Pues bien, no toca la arquitectura ISA. Lo que hace es sustituir las bibliotecas de Windows por otras que hacen llamadas al sistema Unix y ejecutar un proceso que sustituye al *kernel* de Windows. En este sentido, *no es un emulador*. Podemos decir que es un emulador en el nivel de máquina operativa, pero no en el nivel de máquina convencional.

¿Es una máquina virtual? En cierto sentido sí: el usuario ejecuta un programa que está compilado para un sistema operativo en otro sistema diferente. Pero hemos dicho que una máquina virtual de sistema implementa todo un sistema operativo sobre otro, y no es esto lo que hace Wine.

# Capítulo 13

# Arquitectura de la máquina virtual Java

---

Normalmente, al mencionar «Java» pensamos en el lenguaje de programación. Pero no hay que olvidar que el lenguaje está indisolublemente unido a otras tres cosas:

- El formato definido para los ficheros de clases que contienen los resultados de la compilación (ficheros `.class`).
- La API *(Application Programming Interface)* Java, conjunto de programas ya compilados (con el formato «`.class`» o con el formato «nativo» del sistema operativo), de los que se puede servir un programa escrito en Java.
- La máquina virtual Java, que carga, verifica e interpreta los *bytecodes* contenidos en los ficheros de clases, a la que en adelante llamaremos **JVM** *(Java Virtual Machine)*. Su implementación en software, junto con la API, es lo que conocemos como **JRE** *(Java Runtime Environment)*

En sus orígenes, la JVM se concibió para la ejecución independiente de la arquitectura de programas escritos en Java, pero actualmente se aplica también a otros lenguajes. Por ejemplo, Jython y JRuby son implementaciones de procesadores para los lenguajes Python y Ruby que generan código de *bytecodes* para la JVM. De esta manera, los programas fuente pueden hacer uso de las bibliotecas disponibles para Java.

En principio, la JVM es una especificación abstracta (una arquitectura) que se puede implementar de diversas maneras. Pero también se le da ese nombre a una implementación concreta (por ejemplo, «este navegador incluye una JVM»), o a un proceso en esa implementación: toda aplicación o *applet* se ejecuta en «una JVM», un proceso de una determinada implementación.

Este capítulo se centra en la especificación de la JVM. Empezaremos con una visión global de la compilación y la ejecución de programas en Java. Resumiremos también el formato de los ficheros de clases, que es necesario conocer para comprender ciertos aspectos procesales. Luego describiremos un modelo estructural abstracto de la JVM y sus modelos funcional y procesal. Esta descripción no será completa (la especificación «oficial» está contenida en un libro de 608 páginas, el «libro azul», disponible también en línea[1]). Nos limitaremos a un subconjunto del repertorio de instrucciones y a algunos conceptos procesales. Como indica su título, el capítulo se dedica a la arquitectura de la JVM, no a su implementación (apartado 1.7). No obstante, es insoslayable ir mencionando algunos detalles de la implementación para bien comprender la arquitectura. Y al final veremos algunas ideas sobre las distintas implementaciones posibles.

[1] `http://docs.oracle.com/javase/specs/jvms/se8/html/`

## 13.1. Visión de conjunto

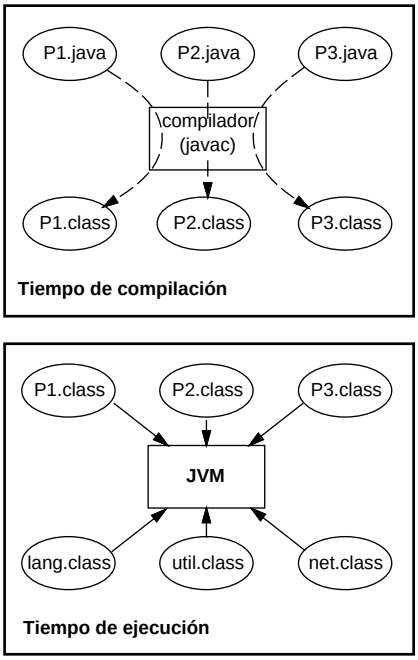



**Figura 13.1**: Ejecución de programas Java.

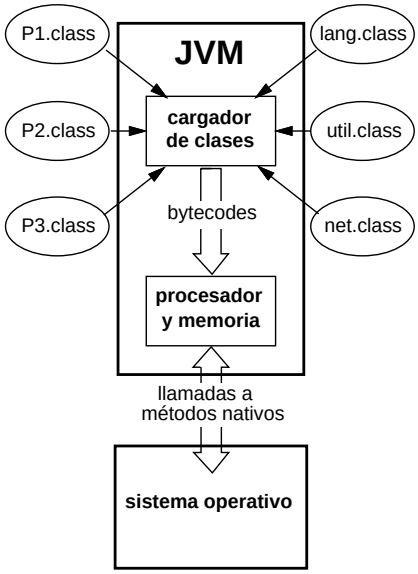



**Figura 13.2**: Componentes de la JVM.

Los programas fuentes, almacenados en ficheros con extensión `.java`, se compilan al lenguaje de la JVM *(bytecodes)*, guardándose en ficheros de clases, con la extensión `.class`, uno por cada clase definida en los fuentes. Con frecuencia, varios ficheros de clases se archivan en un único fichero comprimido (`.jar`). Estos ficheros, quizá después de viajar por la red, son cargados e interpretados por la JVM. Normalmente, los programas fuentes contienen referencias a objetos de las clases de la API. La JVM lo detecta y para poder hacer la interpretación carga las clases necesarias. En la figura 13.1, las clases `lang`, `util` y `net`; normalmente, muchas más.

### Carga, montaje e iniciación

Un **proceso JVM** arranca cuando la JVM recibe la orden de carga de una clase inicial. Por ejemplo, al escribir «`java P1`» en la línea de órdenes. Como debe usted saber, la clase `P1.java` tiene que contener un método `main()`, declarado `public`, `static` y `void`, con un solo argumento, un *array* de `String`. Antes de empezar con la ejecución de los *bytecodes* que han resultado de la compilación de este método, el proceso JVM tiene que completar tres fases: carga de la clase `P1` y de las que sean precisas, montaje e iniciación. En la figura 13.2 se supone que las tres se realizan en el bloque llamado «cargador de clases».

La intención de este capítulo es centrarse en los modelos del procesador que interpreta los *bytecodes*, no en el cargador de clases, cuyos detalles son muy prolijos debido al gran número de comprobaciones de seguridad que realiza. Sin embargo, por los motivos que veremos en el apartado 13.4, en la JVM es imposible separar en el tiempo las operaciones de montaje de las operaciones de ejecución, operaciones que en los procesadores de lenguajes tradicionales se realizan en distintos tiempos (apartado 12.3). Por esto, al explicar el modelo procesal tendremos que comentar algunos detalles de lo que ocurre en carga y el montaje.

### La API y los métodos nativos

Los programas de usuario pueden hacer llamadas a «métodos nativos». No es que hayan nacido en ninguna parte, se llaman así los programas escritos en otros lenguajes y dependientes del sistema operativo concreto en el que se ejecuta la JVM. Pero si se pretende que los programas Java sean *portátiles* (no ya los programas fuente, sino los resultantes de la compilación) sólo deben hacer uso de la API. Por su parte, los programas de la API sí hacen uso

de métodos nativos. La API tiene una implementación específica para cada sistema, pero su modelo funcional (la *interfaz de programación*) es único, independiente de la implementación.

La implementación de la JVM y de la API para un determinado ordenador y un determinado sistema operativo, o para un determinado navegador, es un **entorno de ejecución de Java (JRE)**. Si un programa de usuario no hace uso de llamadas a métodos nativos, los ficheros de clase resultantes de su compilación se ejecutarán sin cambiar nada en cualquier sistema que tenga instalado su entorno de ejecución Java.

Cuando se da la orden de ejecución de un programa Java compilado, o bien cuando el navegador llama a su entorno de ejecución para ejecutar un *applet*, el cargador de clases carga los ficheros de clases de ese programa y los ficheros de clases de la API a las que pueda hacer referencia el programa. El conjunto que resulta de cargar todos los ficheros de clases (los compilados a partir del programa fuente y los de la API), así como las bibliotecas dinámicas (que contienen los métodos nativos) que haya sido necesario cargar, es el programa que ejecuta el procesador de la JVM. No es necesario que este programa esté completo en memoria antes de empezar la ejecución: normalmente se carga la clase principal (la que contiene el método `main()`), comienza la ejecución y se van cargando clases a medida que son necesarias.

### Los procesos JVM y las hebras

En la asignatura «Algoritmos y estructuras de datos» ha estudiado usted el concepto de **hebra** *(thread)* y ha visto su utilidad, en el nivel de máquina simbólica, para la programación concurrente.

Cuando se ejecuta una aplicación (o un *applet*) se crea un **proceso JVM** que muere cuando la aplicación termina. Si la implementación lo permite, se puede pedir la ejecución de una nueva aplicación antes de que hay terminado la primera, y así, si lanzamos $n$ aplicaciones en un JRE tendremos $n$ procesos JVM en ejecución (y $n$ réplicas de la JVM).

Tras la operación de carga de la clase que contiene el método `main()` (o, en el caso de un *applet*, la subclase de `java.applet.Applet`), el proceso JVM arranca una hebra inicial en la que se interpretan los *bytecodes* de este método, es decir, las instrucciones de máquina de la JVM. Desde esta hebra pueden arrancarse otras: en una clase que herede de `java.lang.Thread`, creando un nuevo objeto de esa clase y llamando al método `start()`, o bien en una clase que implemente la interfaz `java.lang.Runnable`, creando un objeto de la clase `java.lang.Thread` y llamando a su método `start()`. En la JVM hay dos tipos de hebras: *demonios* y *no demonios*. La hebra inicial es del segundo tipo y las arrancadas por ella, normalmente, también, aunque el programa de aplicación puede señalar (con `setDaemon(true)`) como demonio a cualquier hebra. Los demonios son generalmente hebras creadas por la propia JVM, como la que se ocupa del reciclado, es decir, el recogemigas (apartado 13.6).

Por tanto, dentro de un mismo proceso JVM puede haber múltiples hebras en ejecución. A lo largo de él pueden irse creando nuevas hebras y terminando otras. El proceso termina cuando no queda ninguna hebra de tipo no demonio.

Así como a cada aplicación le corresponde un proceso JVM (que contiene una copia del procesador y de los elementos de memoria), a cada hebra (dentro de un mismo proceso) le corresponde un «caso de ejecución» *(runtime instance)* del procesador. Cada hebra tiene su propio contador de programa, que contiene la dirección de la instrucción que está siendo ejecutada por esa hebra.

La implementación en bajo nivel de las hebras depende de las facilidades que ofrezca el sistema operativo (apartado 13.6).

## Los ficheros de clases

El cargador de clases coloca en la memoria de la JVM los contenidos de los ficheros de clases, y estos contenidos son lo que luego interpreta el procesador. Para entender el funcionamiento de este procesador es preciso conocer el convenio de almacenamiento en los ficheros de clases, aunque no sea de manera exhaustiva. Este convenio está esquematizado en la figura 13.3. Es una sucesión de bytes en la que se codifican todos los datos que definen una clase. Los datos que ocupan dos o más bytes se almacenan con el convenio extremista mayor.

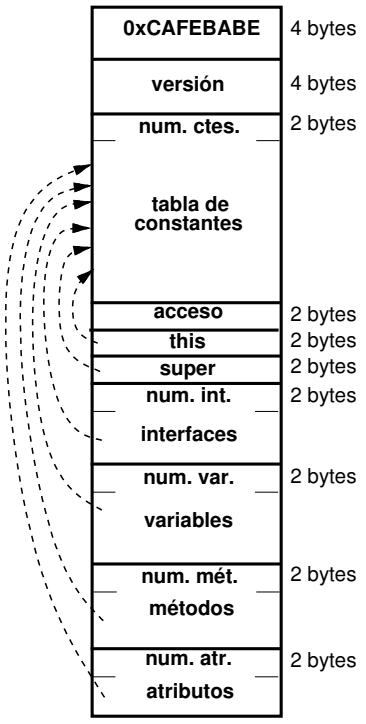



**Figura 13.3**: Formato de los ficheros de clases.

- Primero está 0xCAFEBABE, que es el **número mágico** (apartado 7.1) asociado a los ficheros de clases, y a continuación el **número de versión** de la especificación. Una JVM está diseñada para trabajar con una cierta versión de la especificación (y con las anteriores), y si el fichero procede de un compilador que genera una versión posterior el cargador lo rechaza.

- La **tabla de constantes** *(constant_pool)* es importante para el montaje dinámico, como veremos en el apartado 13.4. Es equivalente a las tablas de símbolos de los ensambladores y compiladores (apartado 12.2), pero más compleja. Debe entenderse por «constantes» no sólo las así declaradas en el programa Java (con el modificador `final`), con valores conocidos en tiempo de compilación, sino todos los nombres simbólicos que aparecen en el programa y que han de resolverse en tiempo de ejecución: nombres de clases (incluidos el de ésta y el de su superclase), de interfaces, de variables y de métodos, con sus firmas (la firma de un método es el tipo del retorno y el número y tipos de sus argumentos). A cada constante se le asocia una estructura de datos, accesible mediante un índice, en esta tabla. En los dos primeros bytes está el número de estructuras contenidas en la tabla. Luego detallaremos el formato de algunas estructuras en función del tipo de constante. Terminemos antes con los componentes que muestra la figura 13.3.

- En **acceso** se codifica (de acuerdo con un convenio en el que no entramos) si se trata de una clase o de una interfaz, y si está declarada `public`, `abstract` o `final`.

- En **this** y **super** hay sendos índices de la tabla de constantes. Las constantes en esos índices deben ser del tipo «CONSTANT_Class», y, como explicaremos más adelante, conducen a los nombres de esta clase y su superclase.

- La lista de **interfaces**, precedida del número de interfaces que esta clase implementa, es simplemente un *array* de índices a la tabla de constantes, donde figuran sus nombres.

- La sección de **variables** también está precedida del número de variables (tanto de objeto como de clase) declaradas en esta clase. Está compuesta por una sucesión de estructuras, una para cada variable, que contienen datos como un índice a la tabla de constantes donde se encuentra el nombre, el tipo, los modificadores (`public`, `private`, `protected`, `final`...) y, en caso de que sea una variable de clase (`static`), su valor.

- En la sección de **métodos**, tras el número de métodos declarados, se incluyen los datos de cada método: índice a la tabla de constantes, modificadores, firma del método, número máximo de palabras

de la pila requeridas para las variables locales y, salvo que el método sea abstracto (o nativo), su código, es decir, los *bytecodes*. Si el código del método incluye manejadores de excepciones hay también una *tabla de excepciones*. Cada entrada de esta tabla contiene las direcciones de las instrucciones primera y última de cada uno de los manejadores (es decir, de los *bytecodes* resultantes de la compilación de las sentencias protegidas por una cláusula `catch`) y un índice a la tabla de constantes donde se encuentran los datos de la clase correspondiente a esta excepción.

• Finalmente, hay una lista de **atributos** que contiene propiedades generales de la clase o interfaz, pero que no vamos a tener en cuenta aquí.

Pasemos a la representación de constantes. Cada una de las estructuras de la tabla de constantes tiene un primer byte llamado «etiqueta» que identifica el tipo de constante y los siguientes contienen datos que dependen del tipo. Hay once tipos diferentes. Algunos son:

• «CONSTANT_Utf8» corresponde a cadenas de caracteres. Tras la etiqueta (de valor 1), dos bytes indican el número de bytes de la cadena, y a continuación van los caracteres codificados en UTF-8.

• Las estructuras «CONSTANT_Integer» y «CONSTANT_Float» contienen, tras la etiqueta (3 y 4 respectivamente) el valor de un entero o de un real en coma flotante, en cuatro bytes.

• «CONSTANT_Class» (etiqueta 7) sólo contiene un índice de esta misma tabla en donde se encuentra el nombre de una clase (en una estructura CONSTANT_Utf8).

• Las estructuras «CONSTANT_Fieldref» y «CONSTANT_Methodref» (etiquetas 9 y 10) son para variables y métodos, y contienen dos índices de la tabla. En el primero debe haber una estructura del tipo CONSTANT_NameAndType, y en el segundo una CONSTANT_Utf8 con el nombre de la clase o de la interfaz en la que está declarada esta variable o este método.

• «CONSTANT_NameAndType» (etiqueta 12) contiene dos índices a estructuras del tipo CONSTANT_Utf8: en la primera está el nombre de la variable o del método, y en la segunda, su «descriptor». No tiene nada que ver con el descriptor de un fichero en los sistemas operativos: aquí un descriptor es una cadena de caracteres que representa el tipo de una variable o de un método.

Supongamos, por ejemplo, que en el programa fuente se ha declarado la variable «`int veloc`» en una clase llamada «`Coche`». El compilador crea para esta variable una estructura del tipo CONSTANT_Fieldref que contiene un índice a la estructura de `Coche` y otro índice a una estructura CONSTANT_NameAndType. En la figura 13.4 puede verse la situación que podría resultar. El descriptor para el tipo `int` es, simplemente, «I». Y en la tabla de variables, en el lugar correspondiente a `veloc` el compilador pondrá el índice 2.

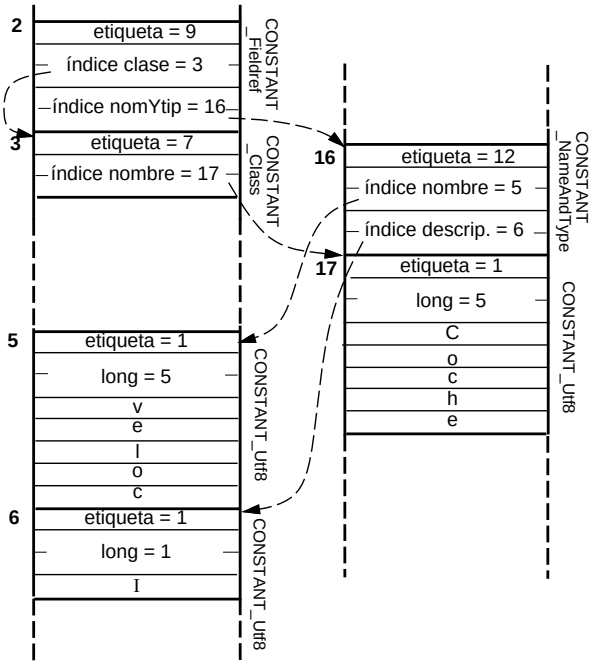



**Figura 13.4**: Estructuras en la tabla de constantes ligadas a una declaración de variable.

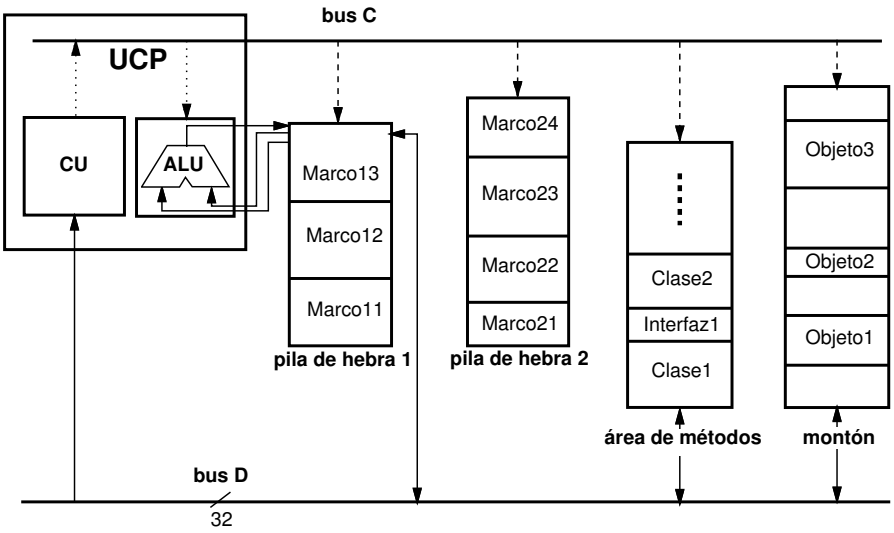



**Figura 13.5**: Modelo estructural de la JVM.

## 13.2. Modelo estructural

La figura 13.5 es un modelo estructural de la JVM[2] en el nivel de máquina convencional. Las diferencias más importantes con los modelos que hemos estudiado anteriormente (figuras 8.7 y 9.1) se deben a que ésta no es una máquina de registros, sino una *máquina de pilas* (apartado 8.8) diseñada para facilitar la programación concurrente con múltiples *hebras:*

- No existen registros accesibles al «programador»[3]: todas las operaciones aritméticas y lógicas se hacen sobre una pila de operandos.
- No hay (para el programador) «direcciones de memoria». De ahí que no aparezca el bus A. En la implementación, los distintos elementos de memoria se materializarán en RAM, pero esto sólo se aprecia en el nivel de microarquitectura, y ahora estamos hablando de la arquitectura (nivel de máquina convencional).
- No hay elementos para las comunicaciones con el exterior. La especificación de la JVM no los contempla. Ésta es una de las funciones que se implementan con métodos nativos. Aunque el procesador tiene que incluir las llamadas al sistema operativo para implementar los métodos nativos (figura 13.2), no entraremos en esto.
- Como indica la figura en el bus D, la longitud de palabra se supone de treinta y dos bits, aunque en la implementación puede ser distinto.
- En lugar de una «memoria principal» hay varios módulos de memoria cuya función vamos a explicar.

Pero antes señalemos otras dos diferencias, éstas más conceptuales:

- En los modelos de las figuras figuras 8.7 y 9.1 se supone que los bloques (CPU, memoria, etc.) se materializan físicamente (implementándose en el nivel de máquina convencional o en el de micromáquina), pero ahora se trata de un *modelo abstracto*. Aunque los bloques pueden realizarse

[2]Por brevedad, seguiremos hablando de «la JVM», aunque, como hemos dicho, no nos ocuparemos del cargador de clases, sólo del procesador (CPU) y la memoria.

[3]El hipotético programador que trabajase codificando directamente los *bytecodes*.

físicamente, lo más frecuente es que no sean más que procedimientos y estructuras de datos de un programa que se ejecuta en una máquina «normal» (von Neumann).

- Aquellos modelos son «estáticos»: los componentes son los que son, no cambian con el tiempo. Pero la figura 13.5 corresponde a un *estado* de la máquina, en el que hay dos hebras en ejecución: una ha llamado a tres métodos y otra a cuatro. En otro momento puede haber otras hebras, y en ese momento habría un número mayor o menor de pilas de hebra con distintos números de marcos.

Los bloques de memoria son: el *área de métodos*, donde se cargan las clases y las interfaces, y que es accesible para todas las hebras, el *montón*, donde se crean objetos, también accesible para todas las hebras, y las *pilas*, una para cada hebra (y no accesible desde las otras)[4].

Podríamos abandonar aquí la explicación del modelo estructural y pasar al funcional (tipos de datos y repertorio de instrucciones). El lector apresurado puede hacerlo, saltándose el resto de este apartado. Si lo hace, se dará cuenta de que faltan algunos detalles sobre la organización de estos bloques de memoria. Veamos esos detalles, pero le advertimos sobre dos cosas:

- Para comprenderlos habrá que ir adelantando algo sobre el modelo procesal e, incluso, sobre la implementación. Por ejemplo, tendremos que hablar de punteros, cosa invisible en el nivel de máquina convencional.
- Los detalles completos son algo enrevesados. Intentaremos dar ideas generales suficientes para que pueda usted imaginar su concreción y, si le interesa, facilitarle el estudio de los documentos oficiales de la especificación.

### El área de métodos

El área de métodos es análoga a la zona de la memoria ocupada por un módulo de carga en una máquina convencional. Pero este nombre, «área de métodos», que se da en la especificación de la JVM, es demasiado restrictivo. Debería llamarse «memoria de clases», porque en ella se almacena toda la información necesaria sobre las clases e interfaces cargadas. Siempre que se carga una clase o una interfaz se crea para ella espacio en este área en el que se transcriben los datos contenidos en su fichero de clase.

Las estructuras de representación en el área de métodos, a diferencia de las de los ficheros de clases, no están prescritas por la especificación: cada implementación puede elegir las más convenientes. Por ejemplo, el **almacén de constantes** *(runtime constant pool)* es una versión interna de la tabla de constantes contenida en el fichero de clase.

Hay otros datos en el área de métodos que no están originalmente en los ficheros de clases:

- Como veremos, al cargar una clase se crea en el montón un ejemplar *(instance)* de la clase `Class` que la representa, y un puntero a ella debe incluirse entre sus datos en el área de métodos.
- La implementación puede incluir otras estructuras de datos convenientes para el rápido acceso a los distintos elementos. Por ejemplo, para cada clase, una *tabla de métodos*, que es un *array* de punteros a las direcciones de comienzo de los *bytecodes* de cada método, incluyendo los heredados de las superclases.

### El montón (*«heap»*)

La JVM asigna a cada objeto el espacio necesario en una memoria llamada **montón**, a la que tienen acceso todas las hebras. No sólo están los objetos que se crean en tiempo de ejecución al interpretar

[4] En realidad, la JVM mantiene dos pilas para cada hebra: la «pila Java», que es a la que nos referimos, y otra pila para los métodos nativos. Pero ya hemos dicho que vamos a prescindir de los métodos nativos en esta presentación resumida.

`new`, o `clone()`, o cualquier otro recurso de los que dispone el lenguaje Java para crear un ejemplar *(instance)* de una clase. En tiempo de carga, para toda clase o interfaz cargada se crea en el montón un ejemplar de la clase `java.lang.Class` a través del cual se puede obtener la información sobre la clase almacenada en el área de métodos (por ejemplo, con `obj.getClass().getName()`). Y a la inversa, un programa puede usar referencias a objetos de la clase `Class` (por ejemplo, con `forName()`, que recibe un nombre de clase y devuelve la referencia a esa clase), y de ahí la necesidad mencionada antes de que en el área de métodos se incluyan punteros, uno para cada clase, a los correspondientes ejemplares de la clase `Class`.

Los *arrays* son objetos, y se almacenan también en el montón. Por ejemplo, para la sentencia `long[] vector = new long[100]` la JVM crea en el montón espacio para los cien componentes (800 bytes), una indicación de la longitud (100) y un puntero a la clase que representa a los *arrays* de una dimensión con el tipo `long`, que el cargador de clases debe haber puesto en el área de métodos. Hay un convenio para formar los nombres de estas clases que se puede resumir con algunos ejemplos: para `int[]` el nombre es «`[I`», para `int[][]`, «`[[I`», para `double[][][]`, «`[[[D`», para `Thread[]`, «`[Ljava/lang/Thread`», para `long[][][][]`, «`[[[[J`».

## Las pilas

Para cada hebra hay una pila asociada que contiene *marcos de pila* (apartado 8.8), uno para cada método que haya sido llamado desde esa hebra y que no haya terminado. Los marcos contienen espacio para los parámetros del método, para las variables locales y para una pila de operandos, e informaciones para acceder a constantes y variables de los objetos y las clases y para retornar del método (extrayendo el marco de la pila). Aunque no se muestre en la figura 13.5, porque es algo propio de la microarquitectura, cada hebra tiene su *contador de programa*, que contiene la dirección en el área de métodos de la instrucción en ejecución.

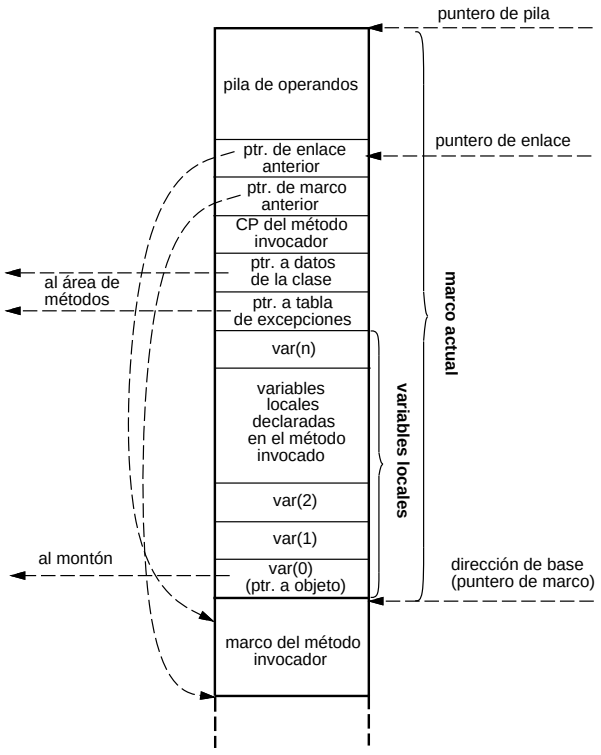



**Figura 13.6**: Un marco de pila.

Si la JVM está implementada sobre un sistema monoprocesador, en cada momento sólo hay una hebra activa. (Su contador de programa coincide con el contenido del registro PC de la CPU). Cuando desde esta hebra se llama a un método, la JVM crea un nuevo marco y lo mete en la pila sobre el marco del método que ha originado la llamada. El último marco metido es el *marco actual* (en el estado de la figura 13.5, sería el M13). Cuando este método termina (ya sea normalmente o debido a una excepción) el marco del método que lo llamó pasa a ser el marco actual.

Un marco tiene tres partes: espacio para variables locales (parámetros pasados al método y variables declaradas en él), un conjunto de punteros y una pila de operandos (figura 13.6).

El acceso a los operandos es de tipo pila, pero a las variables locales se accede mediante un *índice* contenido en la instrucción correspondiente, de modo que se puede hablar de «la variable 0», o «var(0)», «la variable 1», o «var(1)» etc. Las variables de los tipos `long` y `double` ocupan dos

índices. De este modo, a cada índice le corresponden cuatro bytes, y en la implementación puede dedicarse un registro que apunte a la dirección de base, y obtener la dirección efectiva sumando a este puntero el índice multiplicado por cuatro.

En la parte de variables locales están primero los parámetros pasados al método y luego las variables locales declaradas en él. Si es un método de clase (`static`) los parámetros se introducen en variables consecutivas a partir de la variable 0. Si es un método de objeto la variable 0 se utiliza para una referencia al objeto sobre el cual se ha llamado al método (lo que en el lenguaje Java se llama «`this`»), y los parámetros se pasan a partir de la variable 1.

El marco contiene un puntero a los datos de la clase en la que está declarado el método, lo que permite resolver en tiempo de ejecución las referencias que en el programa se hagan a clases para crear objetos, variables a las que acceder o métodos a los que llamar, y otro puntero a la tabla de excepciones. Además, están también los datos necesarios para retornar del método: valor que tenía el contador de programa cuando se ejecutó la instrucción de llamada (para volver al método invocador, es decir, al que hizo esa llamada), puntero al marco anterior y un puntero de enlace.

Por último, la parte superior de cada marco es una *pila de operandos*. Como en toda máquina de pilas (apartado 8.8), se puede hacer una operación entre los dos elementos que están en cabeza extrayendo *(pop)* estos dos elementos, enviándolos a la ALU y metiendo *(push)* el resultado en la pila. También se pueden hacer operaciones *push* y *pop* con otros módulos de memoria, como sugiere la figura 13.5 (con la línea de flechas entre el Marco13 y el bus D) y como veremos al explicar las instrucciones.

Igual que para el área de métodos y el montón, la especificación de la JVM no prescribe cómo deben organizarse internamente las pilas. Los marcos podrían irse asignando en un montón (el mismo de los objetos u otro), pero lo normal es que se implementen, efectivamente, como una pila, como indica la figura 13.6, en donde se supone que el marco actual corresponde a un método no `static` (puesto que var(0) es un puntero al objeto).

El espacio necesario para las variables locales y los punteros y el tamaño máximo de la pila de operandos se determina en tiempo de compilación y se registra en el fichero de clase (en la tabla de métodos), y al cargar la clase se conserva entre los datos correspondientes a esta clase en el área de métodos. De este modo, cuando procesa una llamada a un método, la JVM sabe cuánto espacio en la pila tiene que asignarle al marco.

## 13.3. Modelo funcional

### Tipos de datos

Las instrucciones de la JVM manejan los mismos tipos de datos definidos en el lenguaje Java: `byte`, `short`, `int`, `float`, etc., salvo `boolean`, que se maneja como `int`: 0 para `false`, 1 para `true`. Hay un tipo que no está en el lenguaje, y que se utiliza para las instrucciones de llamada a subprogramas y retornos: `returnAddress`.

La representación interna de un dato numérico o booleano ocupa siempre una o dos palabras de treinta y dos bits (a menos que la implementación se haga sobre una máquina de sesenta y cuatro o más bits): los valores de los tipos `int`, `float` y `reference` ocupan una palabra, los de los tipos `byte`, `short` y `char` se convierten a `int` antes de ponerse en las variables locales o en la pila y los de los tipos `long` y `double`, dos palabras. Para `int` y `long` se utilizan formatos de coma fija de treinta y dos bits, y para `float` y `double` los formatos de la norma IEEE 754 (apartado 4.5).

Como hemos adelantado en el apartado 13.1 al describir la tabla de constantes, los caracteres se

codifican en UTF-8. Los nombres simbólicos de clases e interfaces incluyen el nombre del paquete. En el lenguaje Java se utiliza el separador «.» para construir estos nombres completos (por ejemplo, `java.lang.Thread`); aquí el convenio es utilizar «/» (por ejemplo, `java/lang/Thread`).

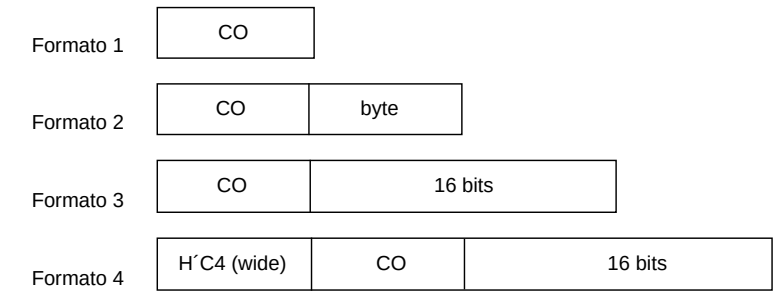
A los datos almacenados en variables locales se accede, como ya hemos dicho, mediante un *índice* contenido en la instrucción. Si el dato es del tipo `long` o `double`, el índice es el de la primera de las palabras. Por ejemplo, si ocupa las palabras de índices 10 y 11 (var(10) y var(11)), la instrucción contendrá el índice 10.

El orden de almacenamiento definido para los ficheros de clases es siempre extremista mayor (si los programas han de ser portátiles, el convenio debe ser único). Sin embargo, en la implementación en software puede cambiarse para adaptarse al convenio de la máquina concreta.

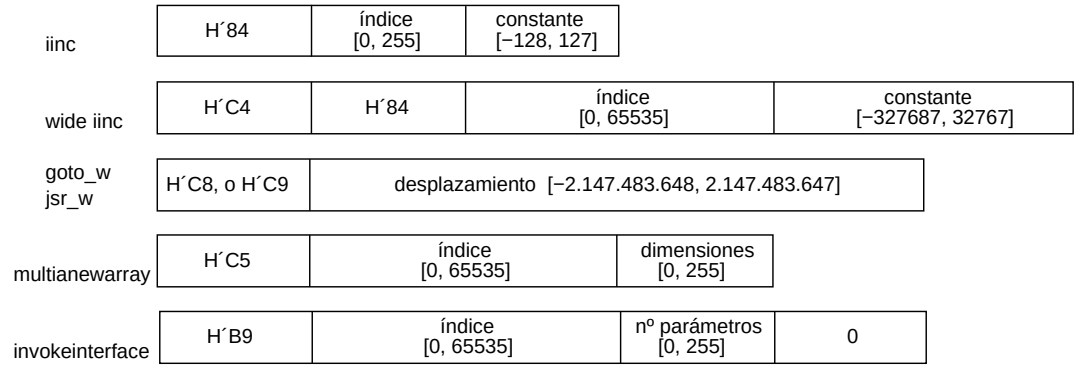
## Formatos de instrucciones

Las instrucciones de la JVM tienen longitud variable. Muchas se codifican en un solo byte, que contiene el código de operación. En otras, este byte va seguido de uno o más bytes que pueden contener un operando, un número de variable local, un índice del almacén de constantes, un desplazamiento en bytes (para las instrucciones de transferencia de control), etc.

En la figura 13.7 aparecen todos los formatos. La mayoría de las instrucciones tiene alguno de los tres primeros. En el formato 2 el byte que sigue al primero contiene un índice o un código numérico (entre 0 y 255) o una constante de tipo `byte` (entre –128 y 127). En el formato 3 los dos bytes que siguen al código de operación contienen un índice (entre 0 y 65.535), una constante de tipo `short` (entre –




**(a) Formatos comunes**




**(b) Formatos específicos**

**Figura 13.7**: Formatos de instrucciones de la JVM.

32.768 y 32.767) o un desplazamiento (entre –32.768 y 32.767) para instrucciones de transferencia de control.

Las instrucciones que acceden a variables locales utilizan, normalmente, el formato 2, poniendo en el segundo byte el índice de la variable. Con 256 palabras suele ser suficiente para almacenar las variables locales, pero si no es así, estas instrucciones no sirven. Hay un código de operación específico, 0xC4 («`wide`» en el ensamblador) que se antepone al código de operación propiamente dicho para indicar que el índice que le sigue está codificado con dieciséis bits (formato 4).

Y hay cinco formatos que son específicos para algunas instrucciones de las que hablaremos más adelante.

### Modos de direccionamiento

¿Modos de direccionamiento? ¿No hemos dicho que para el «programador» (el hipotético programador que escribiese en *bytecodes* o en ensamblador, es decir, con el modelo funcional en el nivel de máquina convencional) no hay «direcciones de memoria»? Sí, es cierto, pero nos referimos a direcciones «abstractas», no a direcciones de la memoria, como en las máquinas de tipo von Neumann. Concretamente, en las instrucciones de la JVM encontramos:

- Direccionamiento inmediato: hay instrucciones que incluyen un operando en el byte o los bytes siguientes al del código de operación.
- Direccionamiento a pila (apartado 8.8).
- Direccionamiento indexado: las instrucciones que acceden al almacén de constantes de una clase, a una variable local de un marco o a una variable de un objeto lo hacen dando un índice que identifica la posición de la constante o la variable.
- Direccionamiento relativo, para las instrucciones de bifurcación dentro de un método.

### Repertorio de instrucciones

Las instrucciones de la JVM están especializadas para los distintos tipos de operandos. Por ejemplo, `iload #n` mete en la pila de operandos el contenido de la variable local `n` (que debe ser de tipo `int`), donde `n` es un número: el índice desde la primera de las variables locales. `float #n` hace lo mismo, pero con una variable que debe ser del tipo `float`. Otras no tienen operandos con tipo, como `goto`, `return` o `nop`. «Debe ser» significa que la JVM tiene que generar una excepción en tiempo de ejecución si no se cumple la condición. En las descripciones que resumiremos a continuación aparecen estos «debe ser», pero no detallaremos las excepciones concretas que se generan en caso de no cumplirse las condiciones.

Utilizaremos los códigos nemónicos definidos en la especificación de la JVM. El convenio es que la primera letra del código indica el tipo del operando (o los operandos): `i` (`int`), `l` (`long`), `s` (`short`), `b` (`byte`), `c` (`char`), `f` (`float`), `d` (`double`) o `a` (`reference`, o *address*). El tipo `boolean` no existe en la JVM, puesto que no hay instrucciones que operen con él. Como ya hemos dicho, las variables de este tipo se representan como enteros. Además, la mayoría de las operaciones sobre variables de tipos `byte`, `short` y `char` se realizan con instrucciones «`i`», puesto que estas variables también se representan internamente como enteros.

A continuación se presentan algunas de las 201 instrucciones incluidas en la especificación. Para cada una se da su código nemónico y en hexadecimal. En los ejemplos se expresa el resultado de la ejecución con la misma notación que hemos utilizado para BRM (es decir, $(A) \rightarrow B$ significa copiar el

contenido de $A$ en $B$, apartado 9.3), y cuando afecta a la pila de operandos se da una descripción del cambio de estado de esa pila con esta notación:

$\ll \ldots, x_3, x_2, x_1 \Rightarrow \ldots, y_3, y_2, y_1 \gg$

donde $x_1$ es el elemento que está en la cabeza antes e $y_1$ el que queda en cabeza después. En lo sucesivo diremos simplemente «la pila» para referimos a la *pila de operandos del marco actual*.

### Movimientos con la pila

Para transferir valores entre variables locales y la pila o, simplemente, manejar la cabeza de la pila, existen, entre otras, las siguientes instrucciones:

- Meter una variable local en la pila (formato 2):

`iload #n`, `lload #n`, `fload #n`, `dload #n` y `aload #n` (códigos binarios 0x15 a 0x19).
`lload` y `dload` meten dos palabras.
Ejemplo:

`iload #5`: (var(5)) → pila
… ⇒ … (var(5))

- Sacar de la pila y llevar a una variable local (formato 2):

`istore #n`, `lstore #n`, `fstore #n`,
`dstore #n`, `astore #n` (0x36 a 0x3A).
`lstore` y `dstore` sacan dos palabras.
Ejemplo:

`lstore #5`: x → var(5); y → var(6)
… x, y ⇒ …

- En las instrucciones anteriores `n` es el índice de la variable local. Es un número comprendido entre 0 y 255 que se codifica en el segundo byte de la instrucción. Ya hemos dicho que, anteponiendo el código `wide`, el valor de `n` puede estar comprendido entre 0 y 65.535 (en dos bytes, formato 4).

Ejemplo:

`wide iload #400`: (var(400))→ pila
… ⇒ … (var(400))

- Pero el caso más común es el contrario: acceder a una variable local con un índice pequeño. Para esto, la JVM dispone de las instrucciones:

`iload_0` a `iload_3` (0x1A a 0x1D)
`lload_0` a `lload_3` (0x1E a 0x21)
`fload_0` a `fload_3` (0x22 a 0x25)
`dload_0` a `dload_3` (0x26 a 0x29)
`aload_0` a `aload_3` (0x2A a 0x2D)
`istore_0` a `istore_3` (0x3B a 0x3E)
`lstore_0` a `lstore_3` (0x3F a 0x42)
`fstore_0` a `fstore_3` (0x43 a 0x46)
`dstore_0` a `dstore_3` (0x47 a 0x4A)
`astore_0` a `astore_3` (0x4B a 0x4D)

cuyos efectos son los mismos que `iload #0`, etc, pero sólo ocupan un byte (formato 1).

Las instrucciones `pop`, `dup` y `swap` tienen también el formato 1:

- Sacar de la pila:

`pop` (0x57), `pop2` (0x58). Ejemplo:

`pop2`: … x, y ⇒ …

- Duplicar la cabeza de la pila:

`dup` (0x59), `dup2` (0x5C) (la segunda duplica dos palabras). Ejemplo:

`dup2`: ... x, y ⇒ ... x, y, x, y

- Intercambiar en la cabeza:

`swap` (0x5F):

`swap`: ... x, y ⇒ ... y, x

- Meter una constante de tipo `byte` o `short` (contenida en la instrucción) en la pila:

`bipush k` (0x10), `sipush k` (0x11).

En la primera, `k` es la constante, comprendida entre −128 y 127 (formato 2), y en la segunda, entre −32.768 y 32.767 (formato 3).

Ejemplo:

`bipush 27`: ... ⇒ ... 27

También hay instrucciones específicas, que ocupan un solo byte, para el caso frecuente de meter constantes pequeñas:

`iconst_m1`, `iconst_0` a `iconst_5` (0x02 a 0x08)

cuyos efectos son los mismos que `bipush -1` `bipush 0`, etc. (pero con un solo byte).

En cualquier caso, lo que se mete en la pila es una palabra de treinta y dos bits, es decir, se convierte el tipo `byte` o `short` a `int`, extendiendo el bit de signo (apartado 10.6, nota al pie).

- Meter una constante (del almacén de constantes) en la pila:

`ldc #n`, `ldc_w #n`, `ldc2_w #n` (0x12 a 0x14)

Para operandos que ocupan una palabra (cuatro bytes), `ldc #n` mete en la pila el valor de una constante del almacén de constantes. En el índice `n` del almacén tiene que haber una estructura del tipo «CONSTANT_Integer», «CONSTANT_Float» o «CONSTANT_String». En este último caso lo que se mete en la pila es la referencia a un objeto que representa al `String`.

`lcd` tiene el formato 2. Para índices (`n`) mayores que 255, `ldc_w`, tiene el formato 3, y también tiene este formato `ldc2_w`, que es para meter, en dos palabras, un operando de tipo `long` o `double`.

- `getstatic #n` (0xB2) y `putstatic #n` (0xB3) permiten acceder a variables de clase (que no están en un objeto del montón, sino en el área de métodos). Tienen el formato 3. En el índice `n` del almacén de constantes debe haber una estructura del tipo «CONSTANT_Fieldref» que conduce al nombre de la variable y al nombre de la clase en la que está declarada. La primera mete en la pila el valor de la variable, y la segunda extrae la cabeza de la pila y la pone como valor de la variable.

### Operaciones aritméticas y lógicas

Todas las operaciones aritméticas y lógicas, salvo una, `iinc`, se realizan con la pila y las instrucciones tienen un solo byte. Para las que tienen dos operandos el resultado es:

... x, y ⇒ ... op(x,y)

y para las unarias:

... x ⇒ ... op(x)

Hay instrucciones para operandos enteros (`int` y `long`) y reales (`float` y `double`), no para operandos de los tipos `byte`, `short`, `char` y `boolean`, que se tratan con las instrucciones para el tipo `int`.

- Suma: `iadd`, `ladd`, `fadd`, `dadd` (0x60 a 0x63)
- Resta: `isub`, `lsub`, `fsub`, `dsub` (0x64 a 0x67)
- Multiplicación: `imul`, `lmul`, `fmul`, `dmul` (0x68 a 0x6B)

- División: `idiv`, `ldiv`, `fdiv`, `ddiv` (0x6C a 0x6F)
- Resto: `irem`, `lrem`, `frem`, `drem` (0x70 a 0x73)
- Complemento a 2: `ineg`, `lneg`, `fneg`, `dneg` (0x74 a 0x77)
- Desplazamiento: `ishl`, `lshl`, `ishr`, `lshr`, `iushr`, `lushr` (0x78 a 0x7D). Las cuatro primeras son desplazamientos aritméticos (llamados `ASL` y `ASR` en la figura 4.2), y las dos últimas desplazamientos lógicos *(«u»: unsigned)*.
- Operaciones lógicas: `iand`, `land`, `ior`, `lor`, `ixor`, `lxor` (0x7E a 0x83)
- Incremento de una variable local: `iinc #n k` (0x84). `n` es un índice de variable local (entre 0 y 255, en un byte) y `k` es un número entre −128 y 127 (en un byte). Ejemplo:

  `iinc #12 -3`: (var(12)) – 3 → var(12)

También existe la versión «`wide`», como puede verse en la figura 13.7, para el caso de que el índice sea mayor que 255 y/o `k` requiera dieciséis bits.

### Creación y manipulación de objetos

- Creación de un objeto:

`new #n` (0xBB),
donde `n` es un índice al almacén de constantes de la clase actual (es decir, la clase en la que está declarado el método correspondiente al marco actual), comprendido entre 0 y 32.767 (la instrucción tiene el formato 3). El contenido en ese índice debe ser una estructura del tipo CONSTANT_Class. Como veremos al explicar el modelo procesal, la ejecución de esta instrucción implica primero resolver la referencia simbólica (para lo cual puede ser necesario cargar la nueva clase) y luego asignar espacio en el montón para el objeto y meter en la pila un puntero a ese objeto:

`new #1`: ... ⇒ ... ref_objeto

Esta instrucción tiene que ir seguida de otra: `invokespecial`, que llama a uno de los constructores de la clase, como veremos más adelante.

- Acceso a variables:

`getfield #n` (0xB4) y `putfield #n` (0xB5)
La referencia (puntero) al objeto debe estar preparada en la pila. Tienen el formato 3, y en el índice `n` del almacén de constantes debe haber una estructura del tipo «CONSTANT_Fieldref» que conduce al nombre de la variable y al nombre de la clase en la que está declarada. Con estos datos, la JVM puede acceder a la variable en la zona del montón reservada para el objeto. Después de la ejecución de `getfield` queda el valor de la variable en la cabeza de la pila: ... ref_obj ⇒ ... valor. Para `putfield` la pila debe contener previamente la referencia y el valor a poner: ... ref_obj, valor ⇒ ...

Hay instrucciones específicas para crear *arrays* y acceder a sus elementos:

- Creación de un `array` unidimensional[5]:

`newarray tipo` (0xBC)
«`tipo`» es un código numérico de un byte que identifica el tipo de los elementos del array.

Por ejemplo, para `new long[100]` (`long` se codifica con 11) el compilador puede generar estas tres instrucciones:

```
bipush 100
newarray 11
astore_1
```

[5]Para *arrays* multidimensionales y *arrays* de referencias hay otras instrucciones, `anewarray` y `multianewarray`.

Al ejecutarse en la JVM:

`bipush 100`: ... ⇒ ... 100

`newarray 11`: ... 100 ⇒ ... ref_array

donde ref_array es un puntero a un bloque de 202 palabras (figura 13.8) que la JVM ha reservado (y puesto a cero) en el montón.

`astore_1`: ref_array → var(1)

... ref_array ⇒ ...

Es decir, se saca la referencia de la pila y se lleva a la variable local 1.

- Acceso a un elemento de un *array:*

`iaload` (0x2E) y `iastore` (0x4F)

Estas dos instrucciones, que sólo tienen un byte, permiten, respectivamente, meter en la pila un elemento de un *array* de `int` y sacarlo para ponerlo en el *array*. Con la primera, el efecto sobre la pila es:

... ref_array, índice ⇒ ... valor

y con la segunda:

... ref_array, índice, valor ⇒ ....

Para `arrays` de otros tipos existen instrucciones análogas: `laload` (0x2F) y `lastore` (0x50) para un *array* de `long`, `aaload` (0x32) y `aastore` (0x53) para uno de `reference`, etc.

- Longitud de un *array:* `arraylength` (0xBE)

También tiene un solo byte. En la pila:

... ref_array ⇒ ... longitud,

es decir, previamente se ha metido el puntero al *array*, y tras la ejecución queda en cabeza la longitud (con el formato de `int`).

### Transferencias de control

- Bifurcaciones incondicionadas:

`goto d` (0xA7), `jsr d` (0xA8), `goto_w d` (0xC8), `jsr_w d` (0xC9), `ret #n` (0xA9)

Donde `d` es un desplazamiento (número con signo) de dieciséis bits para las dos primeras (que tienen el formato 3) o treinta y dos bits para las dos siguientes, que tienen su propio formato (figura 13.7). `ret #n` tiene el formato 2.

`jsr` (0xA8) y `jsr_w` son para bifurcar localmente a un subprograma declarado en el cuerpo del método. Se utilizan, junto con `ret`, para la implementación de la cláusula `finally`. Después de su ejecución la dirección de retorno (que es del tipo `returnAddress`) queda en la cabeza de la pila. Sin embargo, `ret` toma la dirección de retorno de una variable local cuyo índice se da en la instrucción (`ret` no debe confundirse con `return`, que veremos enseguida).

- Bifurcaciones condicionadas a la cabeza de la pila:

`ifeq d`, `ifne d`, `iflt d`,
`ifge d`, `ifgt d`, `ifle d` (0x99 a 0x9E)

Tienen, como `goto`, el formato 3 (y, como ella, van acompañadas de un desplazamiento de dieciséis bits en el segundo byte) y la condición de bifurcación depende del valor que tenga la cabeza de la pila ($= 0, \neq 0, < 0, \geq 0, > 0 \leq 0$), que debe ser de tipo `int` y se saca de la pila:

... valor ⇒ ...

`ifnull d` (0xC6) e `ifnonnull d` (0xC7) son similares, pero lo que hay en cabeza de la pila debe ser del tipo `reference`.

• Bifurcaciones condicionadas a la comparación de dos valores en la cabeza de la pila:
`if_icmpeq d`, `if_icmpne d`, `if_icmplt d`,
`if_icmpge d`, `if_icmpgt d`, `if_icmple d`
(0x9F a 0xA4)
También tienen el formato 3 (desplazamiento de dieciséis bits). La condición es el resultado de comparar los dos valores que están en la cabeza, que deben ser del tipo `int` y se sacan de la pila:

... v1, v2 ⇒ ...

La condición de bifurcación se cumple, respectivamente, si (v1 = v2), (v1 ≠ v2), (v1 < v2), etc.

### Llamadas a y retornos de métodos

• `invokevirtual #n` (0xB6) tiene el formato 3. En el índice `n` del almacén de constantes debe haber una «CONSTANT_Methodref», que conduce al nombre del método y al nombre de la clase en la que se encuentra. En la cabeza de la pila de operandos debe haber un puntero al objeto sobre el que se ejecuta el método y los valores de los argumentos, que la JVM saca de la pila y pone en las variables locales del marco que se crea para el nuevo método (que pasa a ser el marco actual):

ref_obj → var(0)
arg1 → var(1)
arg2 → var(2)
...
argn → var(n)
... ref_obj, arg1, arg2 ... argn ⇒ ...

• `invokespecial #n` (0xB7) es igual que `invokevirtual`, pero se utiliza para métodos de iniciación. Todo constructor que aparezca en el programa Java se traduce por un método de iniciación de ejemplares al que el compilador le da el nombre especial «`<init>`». Si en el programa no se ha declarado ninguno, el compilador construye uno por defecto que llama sin argumentos al iniciador de la superclase. Estos métodos sólo pueden ser llamados mediante la instrucción `invokespecial` sobre objetos no iniciados.

• `invokestatic #n` (0xB8) es similar, pero no existe puntero a objeto, porque es para llamar a métodos de clase (`static`):

arg1 → var(0)
arg2 → var(1)
...
argn → var(n)
... arg1, arg2 ... argn ⇒ ...

• `invokeinterface #n` (0xB9) es igual que `invokevirtual`, pero se utiliza para métodos declarados en una interfaz, que requieren un procedimiento distinto para localizarlos.

• Hay varias instrucciones de retorno, según el tipo del valor que se devuelve:
`ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn` y `return` (0xAC a 0xB1)
La primera se usa para retornar con valores de los tipos `boolean`, `byte`, `char`, `short` e `int`. La última, para el caso de `void` y para métodos de iniciación. Salvo en este último caso, el valor que se devuelve se extrae de la cabeza de la pila y se mete en la pila de operandos del marco del método que llamó a este. En cualquier caso, la JVM ajusta sus punteros internos para que el marco actual pase a ser el del método que llamó (figura 13.6).

## Un programa

Naturalmente, nadie programa en el lenguaje de *bytecodes* (salvo que otorguemos al compilador la categoría de «alguien»). Consideremos este sencillo programa Java:

```
class Coche {
  private int veloc;
  Coche(int inic) {
    veloc = inic;
  }
  int acelera(int a) {
    veloc += a;
    return veloc;
  }
}
class PruebaCoche {
  public static void main(String[] args) {
    int vel0 = 0, vel1;
    int a = 40;
    if (args.length >=1)
       vel0 = Integer.parseInt(args[0]);
    Coche c1 = new Coche(vel0);
    vel1 = c1.acelera(a);
  }
}
```

Al compilar (`javac PruebaCoche.java`) se generan `PruebaCoche.class` y `Coche.class`. El desensamblador (`javap`) es un programa que a partir de un fichero de clase nos da (con la opción `-c`) la codificación en el lenguaje ensamblador de la JVM. Veamos y analicemos el resultado de la orden `javap -c Coche`:

```
Compiled from "PruebaCoche.java"
class Coche extends java.lang.Object{
Coche(int);
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: aload_0
   5: iload_1
   6: putfield #2; //Field veloc:I
   9: return
int acelera(int);
  Code:
   0: aload_0
   1: dup
   2: getfield #2; //Field veloc:I
   5: iload_1
   6: iadd
   7: putfield #2; //Field veloc:I
   10: aload_0
   11: getfield #2; //Field veloc:I
   14: ireturn
}
```

Aunque no se muestra en el listado que genera `javap`, el compilador ha construido la tabla de constantes, y el cargador, a partir de ella, el almacén de constantes en la parte del área de métodos reservada para la clase `Coche`.

En los *bytecodes* que resultan de traducir el constructor `Coche(int)` hay primero una llamada al constructor de la superclase (`Object`). En el índice 1 de la tabla de constantes hay una referencia a este constructor[6], y por eso `invokespecial #1`. La ejecución de esta instrucción supone que previamente se ha puesto el puntero al objeto en la cabeza de la pila. Pero este constructor de `Coche` se estará ejecutando como consecuencia de otra `invokespecial` (resultado de traducir `new Coche(vel0)` en la clase `PruebaCoche`) que habrá creado un marco en cuyas variables locales habrá metido el argumento y el puntero al objeto previamente creado. De ahí que la primera instrucción sea `aload_0`, para meter en la cabeza de la pila ese puntero (que está en var(0)).

Después de la ejecución del constructor de `Object()` se vuelve al método `Coche(int)`, en el *bytecode* de número 4. Lo que dice el constructor en el programa Java es poner el argumento (`inic`) en la variable `veloc`, y esto se realiza con `putfield #2` que contiene un índice, en este caso 2, a la tabla de constantes (al almacén de constantes, cuando el programa se esté ejecutando) en donde está la referencia simbólica a la variable, y saca de la pila el puntero al objeto y el valor a poner. Por eso, previamente se meten en la pila el puntero, que está en la variable local 0 (`aload_0`), y el valor, que está en la variable local 1 (`iload_1`). Finalmente, `return` hace que el marco actual pase a ser el del método que llamó a éste, `main()`.

Los números que aparecen a la izquierda son los de los *bytecodes* relativos al primero de cada método: `invokespecial #1` ocupa, en el método constructor `Coche(int)`, los bytes 1, 2 y 3.

El método `acelera` empieza también con `aload_0` para poner en la cabeza de la pila la referencia al objeto (`Coche`). En su marco estarán las variables locales que se han pasado en la llamada: var(0) (referencia al objeto) y var(1) = `a` (el parámetro que se le ha pasado). Sigamos paso a paso los movimientos en la pila de operandos (que inicialmente está vacía, []):

`aload_0`: [] ⇒ ref_obj

`dup`: ref_obj ⇒ ref_obj, ref_obj

`getfield #2`: ref_obj, ref_obj ⇒ ref_obj, `veloc`

`iload_1`: ref_obj, `veloc` ⇒ ref_obj, `veloc`, `a`

`iadd`: ref_obj, `veloc`, `a` ⇒ ref_obj, `(veloc+a)`

`putfield #2`: ref_obj, `(veloc+a)` ⇒ []; `(veloc+a)` → `veloc` (en el objeto `Coche`)

`aload_0`: [] ⇒ ref_obj

`getfield #2`: ref_obj ⇒ `veloc`

`ireturn`: `veloc` ⇒ [], y en la pila del método que llamó: ... ⇒ ... `veloc`

Tras la última instrucción, el marco del método que llamó vuelve a ser el marco actual, y en la cabeza de su pila está el valor devuelto por `acelera`.

[6] Decir que «en el índice hay una referencia a...» es una simplificación. En realidad, lo que hay en el índice 1 es una CONSTANT_Methodref que contiene los índices 4 y 15. En el índice 4 hay una CONSTANT_Class que contiene el índice 18, y en el índice 18 una CONSTANT_Utf8 con la cadena `java/lang/Object`. En el índice 15 una CONSTANT_NameAndType con los índices 7 y 19, en el índice 7 otra UTF-8 con la cadena <init> (nombre del método a ejecutar) y en el índice 19 otra UTF-8 con la cadena `()V` (convenio para indicar que no tiene argumentos y que el tipo de retorno es `void`). Esta información se ha obtenido con otro desensamblador, JReversePro, que permite ver la tabla de constantes.

Con `javap -c PruebaCoche` se obtiene:

```
Compiled from "PruebaCoche.java"
class PruebaCoche extends java.lang.Object{
PruebaCoche();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return
public static void main(java.lang.String[]);
  Code:
   0: iconst_0
   1: istore_1
   2: bipush 40
   4: istore_3
   5: aload_0
   6: arraylength
   7: iconst_1
   8: if_icmplt 18
   11: aload_0
   12: iconst_0
   13: aaload
   14: invokestatic #2; //Method java/lang/Integer.parseInt:(Ljava/lang/String;)I
   17: istore_1
   18: new #3; //class Coche
   21: dup
   22: iload_1
   23: invokespecial #4; //Method Coche."<init>":(I)V
   26: astore 4
   28: aload 4
   30: iload_3
   31: invokevirtual #5; //Method Coche.acelera:(I)I
   34: istore_2
   35: return
}
```

Y si lo desea puede usted proseguir el análisis, poniendo en relación las instrucciones que ha generado el compilador con las descripciones que hemos dado para ellas y las sentencias del programa fuente. Luego volveremos sobre estos *bytecodes* para analizar cómo se realiza su interpretación. De momento, observe tres cosas:

- Como en el programa fuente no se declara ningún constructor el compilador ha generado uno por defecto que se limita a llamar al de la superclase `Object`, aunque en este caso no se usa (no se crean ejemplares de esta clase).
- La instrucción `if_icmplt 18` ha resultado de traducir `if (args.length >= 1)`. ¿Por qué no es `if_icmpge`? La respuesta es que el compilador trata de optimizar el código que genera. Invirtiendo los términos de la comparación se ahorran dos instrucciones, como puede comprobarse fácilmente.
- Con respecto a la misma instrucción, ¿no le sorprende que el número que le acompaña sea «18», el número del *bytecode* al que efectivamente hay que bifurcar? Porque, según la especificación funcional de las instrucciones de bifurcación, en el segundo byte hay un *desplazamiento*, que en este caso debería ser «10». La explicación es que, para facilitar la lectura, el desensamblador

traduce los desplazamientos a direcciones relativas al comienzo del método. Si se mira el contenido hexadecimal correspondiente a esa instrucción se ve que es 0xA10A: 0xA1 es el código de operación de `if_icmplt` y 0x0A = 10 es el desplazamiento.

## 13.4. Modelo procesal: carga, montaje e iniciación

Profundicemos un poco sobre lo dicho en el apartado 13.1: antes de empezar con la interpretación de los *bytecodes* que han resultado de la compilación del método `main()` contenido en una clase `C`, el proceso JVM tiene que completar tres fases: carga, montaje e iniciación de `C` *en este orden*.

¿En este orden? ¿No hay una contradicción con lo que habíamos dicho en el apartado 12.3? Allí explicábamos que el montador, a partir de las tablas de símbolos y de los códigos objeto resultantes de la compilación de los distintos módulos, genera un programa binario ejecutable que *luego* el cargador introduce en la memoria (figura 12.3). Y esto es lo que se hace en la compilación tradicional. Pero el resultado es un programa binario que sólo puede ejecutarse en una plataforma (un hardware y un sistema operativo) determinada, lo que está en contra de uno de los objetivos del diseño de Java. Los «módulos» que genera el compilador de Java son los *ficheros de clase*, que tienen que poder ejecutarse en cualquier máquina que tenga implementada una JVM específica para ella. No se puede hacer un montaje previo a la ejecución porque resultaría un código binario específico para una máquina determinada. De aquí que tales módulos se cargan tal como están, sin sustituir los nombres simbólicos, y la JVM, ya *en tiempo de ejecución*, se ocupa de *resolver* estas referencias simbólicas.

Así pues, las fases de carga, montaje e iniciación deben sucederse antes de que una clase pueda ser usada. También deben hacerse para las interfaces, con algunas diferencias, pero lo que vamos a decir se refiere únicamente a las clases.

### Carga

Un proceso JVM empieza con la **carga** de una clase inicial. Luego de su montaje e iniciación se ejecuta el método `main()` incluido en la clase, lo que normalmente provoca la carga, montaje e iniciación de más clases e interfaces. La clase inicial normalmente es la nombrada en la línea de órdenes (por ejemplo, `java PruebaCoche`), pero la implementación (por ejemplo, en un navegador) también puede definir una clase inicial por defecto.

Hay un cargador de clases inicial *(bootstrap class loader)*, que forma parte de la implementación de la JVM, y cargadores de clases definidos por el usuario (incluidos en el programa fuente), que son objetos de subclases de `java.lang.ClassLoader`. En cualquier caso, las funciones de un cargador de clases consisten en buscar el fichero (localmente o en la Red), comprobar que tiene el formato `.class`, crear en el área de métodos la representación interna de la clase (que depende de la implementación), si tiene una superclase y ésta no ha sido cargada aún, cargarla (esto puede implicar una ejecución recursiva del cargador hasta llegar a la clase `java.lang.Object`), y crear en el montón un objeto de la clase `java.lang.Class` que representa a la clase.

### Montaje

La especificación de la JVM atribuye tres funciones a la fase de montaje: verificación, preparación y (opcionalmente) resolución.

En la *verificación* se comprueban los *bytecodes:*

- Que los códigos de operación binarios son correctos.

- Que las instrucciones de transferencia de control bifurcan al comienzo de otra instrucción.
- Que los accesos a la tabla de constantes son coherentes. Por ejemplo, en el método `acelera(int)` del ejemplo anterior hay `getfield #2` y `putfield #2`; en el índice 2 de la tabla de constantes debe haber una estructura CONSTANT_Fieldref.
- Que en las operaciones con la pila y las variables locales se respetan los tipos. Por ejemplo, en el mismo método, para la instrucción `iadd` se tiene que comprobar que previamente la pila va a contener dos enteros.
- Que la clase no es subclase de otra declarada `final`.
- ...

Esta verificación puede requerir la carga de nuevas clases, por ejemplo, si hay una instrucción que asigna a una variable de la clase `C2` una referencia a un objeto de la clase `C1`, hay que cargar `C2` para comprobar que `C2` es una subclase de `C1`.

En la *preparación* se ponen valores por defecto en las variables de clase, es decir, las declaradas `static`, y se construyen estructuras de datos internas que permiten posteriormente, en tiempo de ejecución, un acceso eficiente a los métodos y variables.

La *resolución* es la conversión de las referencias simbólicas en referencias directas. Enseguida volvemos sobre esto.

### Iniciación

Por último (antes de empezar la ejecución), en la fase de iniciación se ejecutan los métodos iniciadores estáticos de clase y los iniciadores de variables de clase. Un iniciador de variable de clase es, por ejemplo: `static int n = (int)(Math.random()*100.0);`
Esto implica la carga de la clase `java.lang.Math` y la ejecución de un método para calcular `n`, que el compilador habrá traducido (y le habrá dado el nombre especial `<clinit>`). Los iniciadores estáticos de clase (los que en el programa fuente aparecen en un bloque de sentencias con la palabra `static`) se traducen igualmente en un método `<clinit>`. Esta fase no se realiza necesariamente tras la carga, sino cuando la clase vaya a ser usada, es decir, cuando se intente crear un objeto, o cuando se intente acceder a un método o variable de clase. En ese momento deberá iniciarse la clase y, recursivamente, las superclases que no estén iniciadas.

### Resolución

Es en el último paso del montaje, el de resolución, en donde más se diferencia el modelo procesal de la JVM del tradicionalmente seguido en el procesamiento de lenguajes. Si se está usted preguntando qué es eso de «convertir las referencias simbólicas en referencias directas», se trata simplemente de calcular direcciones absolutas o relativas de memoria a partir de los nombres simbólicos que aparecen en el programa fuente y que han quedado almacenados en el almacén de constantes de la clase. Normalmente, las referencias directas a clases, interfaces, variables y métodos de clase (`static`) son simplemente punteros al área de métodos. Las referencias directas a variables y métodos de objetos son desplazamientos: para una variable, el desplazamiento, sumado al puntero que corresponde al objeto en el montón, da la dirección absoluta de la variable, y para un método el desplazamiento se suma al puntero que da la dirección de comienzo de la tabla de métodos de la clase a la que pertenece el objeto.

Pero como hemos señalado más arriba, hacer la resolución antes de empezar a ejecutar los *bytecodes* de la clase es *opcional*.

La JVM puede proceder al cálculo de todas las referencias directas antes de empezar a ejecutar los *bytecodes* de `main()`, lo que implica una ejecución recursiva de las fases de carga y montaje. En efecto, normalmente, la clase cargada hará referencia a variables y métodos de otras clases. A esto se le llama **resolución temprana** (*early*, o *eager resolution*)[7]. Es más frecuente que la implementación siga una **resolución tardía**, también llamada **perezosa** (*late*, o *lazy resolution*): sólo cuando, ya en tiempo de ejecución, una instrucción necesite acceder a una variable o a un método declarado en esta u otra clase se pone en marcha el mecanismo de resolución, buscando a través de las referencias simbólicas almacenadas en el almacén de constantes, y llamando, si es necesario, a un cargador de clases.

La ejecución de la resolución es muy costosa en tiempo, porque además de tener que calcular la referencia directa, también se realizan durante ella comprobaciones. Por ejemplo, si se trata de acceder a una variable de otra clase y esta variable es privada la JVM debe lanzar una excepción. Por esto, cuando se resuelve una referencia simbólica debe conservarse el resultado de modo que si posteriormente otra instrucción requiere el acceso a esa referencia no sea necesario resolver de nuevo. La especificación de la JVM deja este aspecto abierto a la implementación.

## 13.5. Modelo procesal: interpretación

La ejecución de los *bytecodes* se realiza, en principio, en un bucle de interpretación (apartado 12.4):

```
int PC = tabla_métodos[método_actual];
do {
     byte CO = Área_Métodos [PC];
     switch (CO) {
       case 0x00:      // 0x00 = nop
       //...
       case 0x03:      // 0x03 = iconst_0
         0 → pila;
       //...
       case 0x15:      // 0x15 = iload
          PC = PC + 1;
          byte N = Área_Métodos[PC];
          int OP = Variables_Locales[N];
          OP → pila
       //...
     }
     PC = PC + 1;
   }
while (instrucciones por ejecutar)
```

«PC» es el registro contador de programa en el caso de una implementación cableada, o una variable que inicialmente apunta al primer *bytecode* del método actual.

Pero este bucle se multiplica por el número de hebras que hay en cada momento, cada una con su contador de programa y su pila. Ya hemos dicho en el apartado 12.5 que para soslayar la ineficiencia inherente al proceso de interpretación cuando se realiza con software se suele implementar con compilación JIT (y adaptativa).

Las instrucciones que hemos puesto en el bucle anterior son de las más fáciles. Las más difíciles de interpretar son las que hacen referencias simbólicas al almacén de constantes, que requieren resolu-

[7] En términos matemáticos: la resolución temprana calcula el cierre transitivo de la clase inicial.

ción: `invokevirtual`, `invokespecial`, `getfield`, `putfield`, etc. Como no se trata aquí de describir todas, tomemos el ejemplo anterior de la clase `PruebaCoche` y sigamos el proceso JVM.

### Un ejemplo: `PruebaCoche`

Al arrancar el proceso (como consecuencia de la orden `java PruebaCoche`) se carga la clase `PruebaCoche.class`, se extrae la información binaria contenida en el fichero y se escribe en el área de métodos, construyendo el almacén de constantes. En este caso el almacén queda como indica la tabla 13.1 (en la que se han omitido algunos elementos que no son relevantes para la explicación).

El cargador completa los datos de esta clase en el área de métodos, incluyendo los *bytecodes* del constructor y del método `main()`, y carga también `java.lang.Object` (la superclase de ésta). Supongamos que la JVM hace resolución tardía, por lo que tras la verificación y la preparación pasa a la iniciación (en la que no hay que hacer nada, no hay iniciadores) antes de comenzar la interpretación del método `main()`.

La JVM crea la hebra de `main()` con su contador de programa apuntado al primer *bytecode* y con su pila, que contiene el primer marco. En la sección de variables locales de éste no hay puntero a objeto, puesto que es un método estático, únicamente los parámetros pasados (en este caso sólo uno: una referencia a un *array* de `String`) y espacio para las variables locales:

var(0) = referencia al *array* de `String`
var(1) = `vel0` (tipo `int`)
var(2) = `vel1` (tipo `int`)
var(3) = `a` (tipo `int`)
var(4) = `c1` (tipo `reference`)

Supongamos que en la línea de órdenes se ha escrito «`java PruebaCoche 50`». La JVM crea en el montón un *array* con un único elemento: una referencia a un objeto de tipo `String` con el valor «50», y pone la referencia al *array* en la variable local var(0). Sigamos la ejecución de la hebra, para lo cual hemos de tener presentes tanto el código Java de `main()` como su traducción a *bytecodes* dados una páginas más atrás y los índices y contenidos del almacén de constantes, en la tabla 13.1.

| | **etiqueta** | **índices o nombre** |
|---|---|---|
| 1: | Methodref | 7, 16 |
| 2: | Methodref | 17, 18 |
| 3: | Class | 19 |
| 4: | Methodref | 3, 20 |
| 5: | Methodref | 3, 21 |
| 6: | Class | 22 |
| 7: | Class | 23 |
| 8: | Utf-8 | `<init>` |
| 9: | Utf-8 | `()V` |
| 10: | Utf-8 | `Code` |
| 11: | Utf-8 | `LineNumberTable` |
| ... | | |
| 16: | NameAndType | 8, 9 |
| 17: | Class | 24 |
| 18: | NameAndType | 25, 26 |
| 19: | Utf-8 | `Coche` |
| 20: | NameAndType | 8, 27 |
| 21: | NameAndType | 28, 29 |
| 22: | Utf-8 | `PruebaCoche` |
| 23: | Utf-8 | `java/lang/Object` |
| 24: | Utf-8 | `java/lang/Integer` |
| 25: | Utf-8 | `parseInt` |
| 26: | Utf-8 | `(Ljava/lang/String;)I` |
| 27: | Utf-8 | `(I)V` |
| 28: | Utf-8 | `acelera` |
| 29: | Utf-8 | `(I)I` |

**Tabla 13.1**: Almacén de constantes resultante para la clase `PruebaCoche`.

Las dos primeras sentencias del programa fuente son las asignaciones `vel0 = 0` y `a = 0`, para las cuales el compilador ha generado:

```
iconst_0  // mete 0 en la pila
istore_1  // saca 0 y lo lleva a var(1)
bipush 40 // mete 40 en la pila
istore_3  // saca 40 y lo lleva a var(3)
```

Para `if (args.length >= 1)`:

```
aload_0      // mete var(0) en la pila
arraylength  // saca var(0) y mete longitud
iconst_1     // mete 1
if_icmplt 18 // si long-1 < 0, a 18; la pila queda vacía
```

Si había argumentos en la línea de órdenes, la instrucción `arraylength` devuelve en la pila un valor mayor que 0, y la interpretación continúa a partir del *bytecode* 11, con las instrucciones que han resultado de traducir la sentencia `vel0 = Integer.parseInt(args[0]);`:

```
aload_0    // mete var(0) en la pila
iconst_0   // mete 0 (índice 0 del array)
aaload     // saca lo anterior y mete referencia a args[0]
invokestatic #2 // llamada a parseInt
istore_1   // saca de la pila y pone en var(1); la pila queda vacía
```

Con las tres primeras instrucciones queda en cabeza de la pila la referencia al `String` («50» en el caso de `java PruebaCoche 50`). `invokestatic #2` provoca una llamada al método `parseInt()` de la clase `java.lang.Integer`, que devuelve 50 en la cabeza de la pila en formato de entero, y la instrucción siguiente lo introduce en var(1) = `vel0`.

A continuación, las instrucciones que resultan de traducir `c1 = new Coche(vel0)`:

```
new #3           // crea objeto y mete puntero en la pila
dup
iload_1          // mete var(1)
invokespecial #4 // llamada a Coche(var(1))
astore 4
```

La instrucción `new #3` crea en el montón un objeto de la clase cuyos datos se encuentran a partir del índice 3 del almacén de constantes. La JVM encuentra (siguiendo la cadena de índices) que el nombre de la clase es «`Coche`». Buscando en la memoria de métodos comprueba que aún no está cargada (recuerde que estamos suponiendo resolución tardía), por lo que, con el cargador de clases, procede como antes: lee `Coche.class` y coloca la información, incluyendo su almacén de constantes y los *bytecodes* de su método, en la memoria de métodos. De alguna manera (dependiente de la implementación, como veremos en el apartado siguiente) modifica el contenido del índice 3 para que si se vuelve a hacer referencia a esa clase se acceda más rápidamente a sus datos. Luego usa estos datos de la clase `Class` para asignar la memoria necesaria en el montón para el objeto `c1`. Pone a cero (valor por defecto) en la palabra asignada a la variable `velocidad` y termina poniendo en la pila un puntero al objeto creado. Pero aún quedan operaciones: llamar al constructor de `Coche` y poner la referencia al objeto creado en `c1` (var(4)).

`dup` duplica la cabeza de la pila, e `iload_1` mete el valor de var(1), que en este momento es el entero 50. La pila queda así: ref_obj, ref_obj, 50. Después, `invokespecial #4` llama al constructor de la clase `Coche` cuyos *bytecodes* hemos analizado en el apartado anterior. Por lo que dice el contenido del índice 27 del almacén de constantes, sólo hay un argumento a pasar, de tipo `int`. De acuerdo con la especificación de `invokespecial`, se crea un marco nuevo y se ponen en sus variables locales 50 (en var(1)) y ref_obj (en var(0)). La pila del nuevo marco (que pasa a ser el actual) queda vacía, pero la de éste queda con un solo elemento, ref_obj. Como vimos en el apartado anterior, cuando termina el constructor `Coche(int)`, con `return`, nuestro método vuelve a ser el actual, y su pila no se ha modificado. La instrucción siguiente, `astore 4`, saca la referencia al objeto y la pone en var(4), completando así la sentencia de asignación del programa fuente.

Por último, el compilador ha generado las siguientes instrucciones como traducción de la sentencia `vel1 = c1.acelera(a)`:

```
aload 4
iload_3
invokevirtual #5
istore_2
return
```

`aload 4` vuelve a meter en la pila la referencia al objeto, `iload_3` mete el contenido de var(3) = `a` e `invokevirtual #5` llama al método `acelera(int)` de la clase `Coche` pasándole en el marco que crea para él el valor de var(3) y la referencia, y sacando estos elementos de la pila, que vuelve a quedar vacía. Este método, cuyos bytecodes hemos analizado en el apartado anterior, calcula y pone el nuevo valor de la variable `veloc` en el objeto y devuelve ese valor en la pila, que se lleva a var(2) = `vel1` con la instrucción `istore_2`.

## 13.6. Implementación de la JVM

### Implementaciones totalmente programadas

En las versiones más sencillas, el procesador de la JVM es un programa que se ejecuta sobre una máquina convencional y que tiene un bucle de interpretación como el esbozado al final del apartado 12.4, y concretado en el caso de la JVM en el apartado 13.5. Pero el hecho de que las referencias a las clases, los métodos y las variables conserven en el fichero de clases y en el almacén de constantes su forma simbólica obliga a perfeccionar el proceso, porque de lo contrario resultarían tiempos de ejecución inaceptables.

Las explicaciones dadas hasta ahora son quizás demasiado escuetas para percibir la cantidad de operaciones a realizar cada vez que se resuelve una referencia simbólica. Piense, por ejemplo, en lo que la JVM tiene que hacer para ejecutar la instrucción `getfield #2` contenida en el método `acelera(int)` del ejemplo visto anteriormente. En primer lugar, algo que no hemos dicho antes pero es necesario: como las estructuras contenidas en el almacén de constantes tienen longitud variable, la implementación debe incluir una tabla que haga corresponder a cada índice su posición relativa dentro del almacén (esta tabla se construye cuando se carga la clase y se guarda junto con los demás datos de la clase en el área de métodos). Usando esta tabla, la JVM comprueba que en el índice 2 del almacén de constantes hay una estructura CONSTANT_Fieldref. En ella encuentra un índice a la estructura de la clase en la que está declarada y un índice a una estructura que da el nombre y el tipo (figura 13.4). A través del primer índice (3) encuentra el nombre de la clase (en 17, comprobando previamente que es una CONSTANT_Class), comprueba que ya está cargada (claro, es esta misma clase; si fuese una variable de otra clase podría ser necesario cargarla en este momento) y, teniendo en cuenta los modificadores de acceso (incluidos entre los datos de la clase, que, igual que el almacén de constantes, están en el área de métodos), que este método tiene permiso para acceder a esa variable (en este caso lo tiene, porque está declarado en la misma clase). Entre los datos de la clase hay una tabla de variables con índices del almacén de constantes. A partir de ahí encuentra la posición relativa de la variable que tiene el índice 2. Comprueba que tiene el mismo tipo, `int`, que figura en el almacén de constantes (índice 6). Y con esta posición relativa y el puntero al objeto (que previamente se habrá puesto en la cabeza de la pila) puede ya acceder al valor de la variable `veloc` dentro de la zona del montón reservada para el objeto. El número de operaciones sería aún mayor si la variable estuviese declarada en otra clase, o en una superclase.

Las comprobaciones que se realizan para realizar la resolución de las referencias simbólicas se justifican porque permiten garantizar la integridad del código (si alguna de las comprobaciones falla la JVM lanza una excepción y hace abortar la ejecución del método), algo especialmente importante para programas que se mueven por la Red y que podrían ser «hostiles». Pero una vez realizadas no tiene sentido repetirlas. Entre los *bytecodes* que resultaron de la traducción del método `acelera()` puede verse que poco después de la primera `getfield #2` hay otra `getfield #2`. Si tras realizar por primera vez las operaciones y comprobaciones descritas la resolución ha dado como resultado, sin error, una referencia directa (un desplazamiento para acceder a la variable `veloc` en la zona del montón dedicada al objeto en cuestión) lo razonable es guardar ese resultado para usarlo directamente en cualquier referencia posterior.

Al no tener que rehacer una y otra vez el proceso de resolución el ahorro de tiempo de ejecución puede ser muy grande, especialmente si la instrucción se ejecuta repetidamente en un bucle. Esto conduce a una primera idea para la optimización de la JVM: la *reescritura del código*. Veamos en qué consiste.

### Modificación de los programas en tiempo de ejecución

Un método muy sencillo y eficiente: sustituir, en tiempo de ejecución y tras ejecutarlas por primera vez, las instrucciones que requieren resolución de referencias simbólicas por otras instrucciones que acceden a través de una referencia directa, y que se llaman «rápidas», o «ligeras» *(quick)*. Estas instrucciones se ejecutan mucho más rápidamente porque con ellas la JVM no necesita rehacer todas las operaciones y comprobaciones ya hechas con las instrucciones «pesadas».

Por ejemplo, la versión rápida de `getfield` es `getfield_quick`. Su código de operación es uno de los no definidos en la especificación de la JVM, y contiene no ya un índice al almacén de constantes, sino un desplazamiento en el objeto referenciado (recuerde que en la cabeza de la pila debe estar previamente el puntero al objeto) que permite acceder *directamente* a la variable. Cuando la JVM ejecuta por primera vez la instrucción `getfield #2` sustituye, en todos los lugares del método en los que aparece, sus tres *bytecodes* por los de `getfield_quick #n`, donde `n` es el desplazamiento que ha tenido que calcular, en ocho bits.

Las instrucciones que mueven entre las variables locales y la pila, las aritméticas y lógicas y las de transferencias de control no necesitan resolver referencias simbólicas. Todas las que crean objetos o arrays, acceden a sus variables o métodos o elementos, así como las llamadas a métodos tienen una versión «quick». Estas versiones tienen el mismo número de bytes que las instrucciones originales a las que sustituyen (de lo contrario, la sustitución de una afectaría a todo el código). Así, `getfield #n` tiene el formato 3 de la figura 13.7 (tres bytes), porque `n` es un índice al almacén de constantes; `getfield_quick #n` sólo necesita dos bytes, porque `n` es un desplazamiento de ocho bits, pero se codifica con tres bytes, no usándose el tercero.

Es notable que esta técnica implica una modificación de los programas en tiempo de ejecución, pero no es el programa el que se modifica a sí mismo, es la JVM quien lo hace.

Como la especificación incluye 201 instrucciones, hay 55 códigos no definidos que se pueden usar para las instrucciones «quick». La elección de éstos queda abierta a la implementación, ya que no afecta al formato de los ficheros de clases, y, por tanto, a que los programas sean portátiles, pero en la primera edición de la especificación se definieron veinticuatro instrucciones «quick» que se siguen utilizado en algunas implementaciones actuales. Algunas son:

- `ldc_quick #n` (H'CB). Es como `ldc #n` (mete en la pila una constante), pero en el índice `n` del almacén de constantes estará ya la constante, no la estructura que conduce a ella.

- `getfield_quick #n` (H'CE) pone en la pila el valor de una variable de un objeto. Como `getfield #n`, tiene el formato 3, pero en el segundo byte hay un desplazamiento en la memoria reservada para el objeto (cuya referencia ha de estar en la cabeza de la pila), y el tercer byte no se usa.
- `putfield_quick #n` (H'CF) pone en una variable de objeto un valor sacado de la pila. Como `putfield #n`, tiene el formato 3, pero en el segundo byte hay un desplazamiento en la memoria reservada para el objeto (cuya referencia ha de estar en la cabeza de la pila, antes del valor a poner) y el tercer byte no se usa.
- `getstatic_quick #n` (H'D2). Con el formato 3, como `getstatic #n`, pone en la pila el valor de una variable de clase. En el índice `n` del almacén de constantes debe estar, ya resuelta, esta variable.
- `putstatic_quick #n` (H'D3). Pone en el índice `n` del almacén de constantes, ya resuelto para contener a una variable de clase, el valor que figura en la cabeza de la pila,
- `invokevirtual_quick #n k` (H'D6) llama a un método de objeto cuya referencia, junto con los valores de los parámetros, se han puesto en la pila. Tiene tres bytes: el segundo se interpreta como un índice a la tabla de métodos de la clase del objeto, y el tercero es el número de parámetros del método.
- `invokestatic_quick #n` (H'D9) llama a un método de clase. En el índice `n` (de dos bytes) del almacén de constantes hay una referencia directa al comienzo del método.
- `new_quick #n` (H'DD) es como `new #n`, pero en el índice `n` del almacén de constantes de la clase actual hay una referencia directa a los datos de la clase del objeto a crear.

## Compilación en tiempo de ejecución

Prácticamente todas las implementaciones programadas tanto de la JVM como de otras máquinas virtuales de proceso actuales (apartado 12.6) utilizan compiladores adaptativos (apartado 12.5). La implementación de referencia es la llamada «HotSpot», que forma parte del OpenJDK (Open Java Development Kit), software libre mantenido por la empresa Oracle. Si tiene usted instalado en su ordenador personal alguna versión de Java, seguramente será ésta la JVM que tenga.

Nada mejor para comprobar la eficiencia que se consigue con esta implementación que hacer un pequeño ejercicio. Este programa Java:

```
class Bucles {
 public static void  main(String[] args) {
  int k;
  for (int i = 0; i < 10000; i++)
   for (int j = 0; j < 1000; j++)
    for (int l = 0; l < 10; l++)
      k = 10*i - 100*j + 1000*l;
 }
}
```

no tiene nada de particular. Solamente que requiere realizar algunas operaciones aritméticas cien millones de veces. ¿Cuánto tiempo tardará? En Unix es fácil averiguarlo con la orden `time`: después de compilarlo, en lugar de «`java Bucles`» escriba «`time java Bucles`». La primera línea del resultado muestra el número de segundos transcurridos desde que se da la orden hasta que termina el programa. Recuerde o anote este dato.

Se puede forzar a la JVM para que funcione en modo intérprete puro (es decir, con el compilador inhibido) poniendo la opción `-Xint`. Compruebe el tiempo de ejecución escribiendo esto en la línea de órdenes: «`time java -Xint Bucles`». Observará que es del orden de treinta veces mayor (los resultados concretos varían, naturalmente, en función del ordenador, y, en el mismo, de una vez a otra, puesto que dependen de los procesos activos en cada momento).

Pero aún hay algo mucho más sorprendente. Un código máquina nativo, resultado de compilar un programa en lenguaje de alto nivel tradicional, debería ser más eficiente. Sin embargo, pruebe a compilar este programa en C (`gcc -o Bucles Bucles.c`), que es completamente equivalente al anterior:

```
int main () {
  int i, j, k, l;
  for (i = 0; i < 10000; i++)
   for (j = 0; j < 1000; j++)
    for (l = 0; l < 10; l++)
      k = 10*i - 100*j + 1000*l;
}
```

y a ejecutarlo midiendo el tiempo (`time ./Bucles`). Observará que ¡es mayor que el utilizado por la JVM! (del orden del que resulta inhibiendo la compilación). Ahora bien, esto tiene una explicación: el compilador `gcc` no optimiza por defecto el código para el procesador concreto. Para mitigar la sorpresa, vuelva a compilar añadiendo la opción «`-O3`» y vuelva a medir el tiempo de ejecución del resultado. Naturalmente, lo que ocurre es que la implementación de la JVM optimiza por defecto.

### Las hebras verdes y las hebras nativas

En las primeras implementaciones de la JVM pocos sistemas operativos daban facilidades para el uso de hebras, y además es más eficiente crear y gestionar las hebras en la aplicación (la JVM) que depender del sistema operativo. Esas implementaciones incluían un planificador, y a las hebras gestionadas de este modo se les llamó *green threads*[8].

La JVM arranca con una sola hebra, la correspondiente a `main()`. Cuando se crea una nueva hebra (con el método `Thread.start()`) se le asigna espacio en la memoria para su pila (figura 13.5) y variables (palabras de la memoria) para mantener la dirección del fondo, el puntero de pila y su contador de programa, y la hebra se pone en el estado preparada. A partir de ese momento es el planificador quien decide qué hebra pasa en cada momento al estado activa (en ejecución), en función de su prioridad (que se puede fijar con el método `Thread.setPriority()`).

Los sistemas operativos actuales incluyen facilidades para la creación y planificación de hebras. Para Unix hay un estándar, *pthreads* (POSIX threads, página 37), que define una API con todo tipo de funciones para ello, y funciones similares se encuentran en la API de Windows. Las implementaciones actuales de la JVM normalmente hacen uso de estas «hebras nativas», fundamentalmente por dos motivos:

- Generalmente, el planificador en las *green threads* sólo cambia la hebra activa a otro estado cuando esta hebra cede explícitamente el control (con `Thread.yield()`, `Object.wait()`, etc.) o realiza una operación que la bloquea (`InputStream.read()`, `ServerSocket.accept()`, etc.). Además, cuando la hebra se bloquea todas las demás también lo hacen. Por el contrario, las hebras nativas son gestionadas por el planificador del *kernel*, que normalmente tiene un algoritmo apropiativo (*preemptive*, página 19).

[8] El lenguaje Java se presentó en 1995. Se gestó a partir de los trabajos de un pequeño grupo de investigadores de la empresa Sun Microsystems formado en 1991 que se autodenominaba el «green team».

- Más importante aún, en un sistema multiprocesador (y prácticamente ya lo son todos) todas las *green threads* del proceso JVM se ejecutan el mismo procesador hardware. Sin embargo, el *kernel* puede distribuir las hebras nativas entre los distintos procesadores.

## Las pilas, el montón y el recogemigas

La implementación de las pilas de las hebras en una memoria de acceso aleatorio es conceptualmente sencilla, siguiendo la idea que vimos en el apartado 8.8.

Un poco más complicada es la implementación del montón, donde, como hemos dicho (apartado 13.2), la JVM reserva espacio para los objetos que se van creando, para los *arrays* y para ejemplares representativos de las clases. El espacio asignado a un objeto contiene sus variables (las declaradas en su clase y en sus superclases) y una referencia para localizar los datos de su clase. El asignado a un *array* contiene, además del número de componentes y de sus valores, una referencia a la clase que representa al tipo de *array*.

El convenio concreto puede ser el de la figura 13.8: espacio para las variables de objeto (no `static`) declaradas en la clase y en sus superclases (para poder acceder a sus valores a partir de una referencia al objeto), y un puntero a la tabla de métodos (para poder ejecutar llamadas a esos métodos). En realidad, la representación de objetos debe incluir también variables para implementar la sincronización del acceso de hebras así como datos para poder liberar el espacio que ocupa cuando ya no se use.

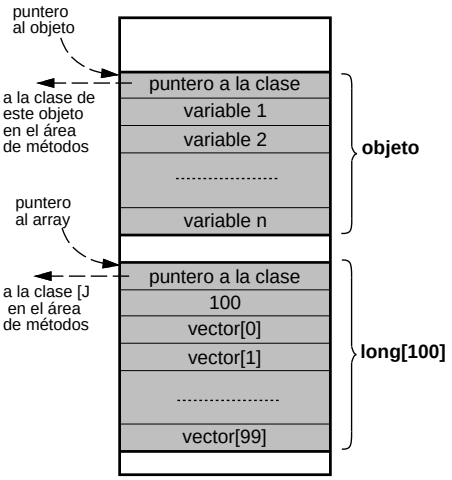



**Figura 13.8**: Un objeto y un *array* en el montón.

Pero más complicada que la representación en el montón es su gestión. La JVM dispone de una instrucción para la operación de asignar memoria a nuevos objetos, pero no hay ninguna instrucción para liberar memoria. Normalmente, la implementación incluye un **recogemigas** *(garbage collector)* que se ejecuta en su propia hebra y gestiona el montón, detectando los objetos que han dejado de ser referenciados y liberando el espacio ocupado por ellos, y moviendo los demás objetos para evitar la «fragmentación externa» (es decir, que queden fragmentos de memoria libres pero de tamaño insuficiente para nuevos objetos).

## Implementaciones parciales en hardware

Si la arquitectura de la JVM es la de una máquina de pilas parece razonable diseñar un procesador hardware con esa arquitectura en lugar de traducir los programas a una máquina de registros. En efecto, desde la publicación de la primera especificación de la JVM en 1995 se ha trabajado en eso. Sun Microsystems anunció en 1998 una serie de procesadores cableados: MicroJava, PicoJava y UltraJava. Del segundo se publicaron bastantes detalles: 341 instrucciones (las 201 de la JVM, más las 25 «quick», más 115 para entrada/salida, operaciones del software del sistema y ejecución de resultados de la compilación de programas en C y C++), una memoria de pila de 64 registros, 25 registros de 32 bits adicionales para funciones específicas, memorias *cache* opcionales de entre 1 KB y 16 KB, y una unidad de control con encadenamiento en la que la mayoría de las instrucciones se ejecutan en un solo ciclo de reloj. Pero estos «chips Java», salvo por algunos prototipos, no llegaron a utilizarse.

Otros fabricantes han adoptado desde entonces parte de las ideas para diseñar procesadores Java, o coprocesadores «aceleradores» que trabajan conjuntamente con un microprocesador convencional. Su aceptación en el mercado ha sido baja, y gran parte de la culpa ha sido la progresiva mejora de los compiladores JIT y el aumento de la capacidad de memoria disponible en los dispositivos pequeños, a los que iban destinados estos procesadores.

Un caso paradigmático lo representa la evolución de las innovaciones de ARM en esta línea. En BRM hemos estudiado la arquitectura ISA básica, pero, entre otras cosas, hemos prescindido de un segundo «modo de ejecución», llamado *Thumb*, que tienen los procesadores ARM. Es en realidad otra arquitectura ISA en el mismo chip en la que la mayoría de las instrucciones son de 16 bits en lugar de 32, para conseguir programas más compactos. Un bit en el registro de estado permite pasar de un modo a otro. Pues bien, en 2003 ARM presentó procesadores con un tercer modo de ejecución en el que el hardware ejecuta directamente los *bytecodes*. Se llamaba «tecnología Jazelle DBX» *(Direct Bytecode eXecution)* y se anunciaba como un gran avance para aplicaciones en dispositivos móviles: permitía acelerar el 95 % de las instrucciones de la JVM y prescindir de los recursos de memoria que exige la compilación JIT. Sólo dos años después se presentó «Jazelle RCT» *(Runtime Compilation Target)*, en la que ya no se interpretaban en hardware las instrucciones de la JVM, pero el nuevo estado de ejecución, que se llamó «ThumbEE» *(Thumb Execution Environment)*, era una nueva arquitectura ISA optimizada para que los compiladores JIT (no sólo de Java, también de C#, Perl y Python) puedan generar un código más compacto. Finalmente, en 2011 ARM ha presentado una nueva arquitectura de 64 bits, la ARMv8, en la que el modo «ThumbEE» ha desaparecido.

# Apéndice A

# Sistemas de numeración posicionales

## A.1. Bases de numeración

El sistema de numeración decimal utiliza diez cifras o dígitos y un convenio que consiste en atribuir a cada dígito un valor que depende de su posición en el conjunto de dígitos que representa al número. Así, por ejemplo:
$3.417 = 3 \times 10^3 + 4 \times 10^2 + 1 \times 10^1 + 7 \times 10^0$;
$80,51 = 8 \times 10^1 + 0 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$

Decimos que este sistema tiene base 10. La base, $k$, es el cardinal del conjunto de símbolos (o *alfabeto*) utilizados para la representación del número. Si $k > 10$, entonces los diez dígitos decimales no son suficientes; es convenio generalizado el utilizar las letras sucesivas, a partir de la A. Así, para $k = 16$ (sistema hexadecimal), el alfabeto es:

$$\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\},$$

y el número AF1C, por ejemplo, tendrá la siguiente representación decimal:
$10 \times 16^3 + 15 \times 16^2 + 1 \times 16 + 12 = 44.828$

En general, si $a_n a_{n-1} \ldots a_1 a_0, a_{-1} \ldots a_{-m}$ está escrito en base $k$, entonces su valor en base 10 puede calcularse así:
$a_n \times k^n + a_{n-1} \times k^{n-1} + \ldots + a_1 \times k + a_0 + a_{-1} \times k^{-1} + \ldots + a_{-m} \times k^{-m}$

## A.2. Cambios de base

El problema general, considerando de momento sólo números enteros, consiste en que, dado un número expresado en base $k$:

$$N_{(k)} = a_p a_{p-1} \ldots a_1 a_0,$$

se desea hallar su representación en base $h$:

$$N_{(h)} = b_q b_{q-1} \ldots b_1 b_0$$

La forma más sencilla de resolverlo (porque estamos acostumbrados a operar en decimal) es pasar por el intermedio de la base 10. Así, de $N_{(k)}$ calcularemos $N_{(10)}$ mediante las potencias sucesivas. Para obtener luego $N_{(h)}$ a partir de $N_{(10)}$ tendremos en cuenta que:
$N_{(10)} = b_0 + b_1 \times h + b_2 \times h^2 + \ldots + b_{q-1} \times h^{q-1} + b_q \times h^q = b_0 + N1_{(10)} \times h$
con

$N1_{(10)} = b_1 + h \times N2_{(10)}$;
$N2_{(10)} = b_2 + h \times N3_{(10)}$, etc.

De aquí,

$b_0 = N_{(10)}$ módulo $h_{(10)}$ (resto de la división $N_{(10)}/h$)
$b_1 = N1_{(10)}$ módulo $h_{(10)}$, etc.

Es decir, el procedimiento consiste en dividir sucesivamente el número y los cocientes que vayan resultando por la base a la que queremos pasar, y los dígitos de la representación en esta base serán los restos que vayan resultando.

Por ejemplo, para representar en base 12 (alfabeto: {0,1,2,3,4,5,6,7,8,9,A,B}) el número $4432_{(5)}$ escrito en base 5 (alfabeto: {0,1,2,3,4}) operaremos así:

$N_{(10)} = 4 \times 5^3 + 4 \times 5^2 + 3 \times 5 + 2 \times 5^0 = 617$
$617 \div 12 = 51 + 5/12; b_0 = 5;$
$51 \div 12 = 4 + 3/12; b_1 = 3;$
$4 \div 12 = 0 + 4/12; b_2 = 4;$

Luego $4432_{(5)} = 435_{(12)}$

Otro ejemplo. Conversión de $4664_{(7)}$ a base 12:

$N_{(10)} = 4 \times 7^3 + 6 \times 7^2 + 6 \times 7 + 4 = 1712$
$1712 \div 12 = 142 + 8/12; b_0 = 8;$
$142 \div 12 = 11 + 10/12; b_1 = 10_{(10)} = \mathrm{A}_{(12)};$
$11 \div 12 = 0 + 11/12; b_2 = 11_{(10)} = \mathrm{B}_{(12)};$

Luego $4664_{(7)} = \mathrm{BA8}_{(12)}$

El paso intermedio por el sistema decimal no sería necesario si hiciésemos todas las divisiones en la base $k$ de partida, pero resultaría más incómodo, al no estar acostumbrados a operar en otra base que la decimal.

El procedimiento para la parte fraccionaria es parecido, pero con multiplicaciones sucesivas en lugar de divisiones. En efecto, sea un número fraccionario $M$, entre 0 y 1, expresado en la base $k$:

$M_{(k)} = 0, a_{-1}a_{-2}\ldots$

y queremos hallar su representación en base $h$:

$M_{(h)} = 0, b_{-1}b_{-2}\ldots$

Primero lo pasamos a base 10, y luego tenemos en cuenta que

$M_{(10)} = b_{-1} \times h^{-1} + b_{-2} \times h^{-2} + \ldots$

Para hallar $b_{-1}$ haremos:

$h_{(10)} \times M_{(10)} = b_{-1} + M'_{(10)}$

Por tanto, $b_{-1}$ = (parte entera de $h_{(10)} \times M_{(10)}$).

Análogamente,

$b_{-2}$ = (parte entera de $h_{(10)} \times M_{(10)}$),

y así sucesivamente.

Si queremos, por ejemplo, escribir en base 12 el número $M = 0,01_{(5)}$ operaremos de este modo:

$M_{(10)} = 1 \times 5^{-2} = 1/25 = 0,04$
$12 \times 0,04 = 0,48 = 0 + 0,48; b_{-1} = 0$
$12 \times 0,48 = 5,76 = 5 + 0,76; b_{-2} = 5$
$12 \times 0,76 = 9,12 = 9 + 0,12; b_{-3} = 9$
$12 \times 0,12 = 1,44 = 1 + 0,44; b_{-4} = 1$
$12 \times 0,44 = 5,28 = 5 + 0,28; b_{-5} = 5$
$12 \times 0,28 = 3,36 = 3 + 0,36; b_{-6} = 3$
$12 \times 0,36 = 4,32 = 4 + 0,32; b_{-7} = 4$

Resultando: $0,01_{(5)} = 0,0591534\ldots_{(12)}$.

## A.3. Bases octal y hexadecimal

La conversión de una base a otra resulta muy fácil si una de ellas es potencia de la otra, pues en ese caso basta con reagrupar los dígitos.

Sea por ejemplo un número expresado en binario,

$N_{(2)} = a_p a_{p-1} a_{p-2} \ldots a_1 a_0$

del que queremos hallar su representación en base 8, o **representación octal** ($2^3 = 8$):

$N_{(8)} = b_q b_{q-1} b_{q-2} \ldots b_1 b_0$

Si hacemos la siguiente agrupación:

$N_{(10)} = a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 +$
$a_3 \times 2^3 + a_4 \times 2^4 + \ldots =$
$= (a_0 + 2 \times a_1 + 4 \times a_2) +$
$(a_3 + 2 \times a_4 + 4 \times a_5) \times 2^3 +$
$(a_6 + 2 \times a_7 + 4 \times a_8) \times 2^6 + \ldots$

y llamamos

$b_0 = (a_0 + 2 \times a_1 + 4 \times a_2)_{(10)} = (a_2 a_1 a_0)_{(2)}$
$b_1 = (a_3 + 2 \times a_4 + 4 \times a_5)_{(10)} = (a_5 a_4 a_3)_{(2)}$
$b_2 = (a_6 + 2 \times a_7 + 4 \times a_8)_{(10)} = (a_8 a_7 a_6)_{(2)}$

etc., podemos escribir:

$N_{(10)} = b_0 + b_1 \times 2^3 + b_2 \times 2^6 + \ldots$
$= b_0 \times 8^0 + b_1 \times 8^1 + b_2 \times 8^2 + \ldots$

Vemos así que los dígitos de la representación octal de un número binario pueden obtenerse directamente, agrupando los bits de tres en tres, y lo mismo ocurre con los números fraccionarios. He aquí un par de ejemplos:

$\underbrace{100}_{4}\underbrace{110}_{6} = 46_{(8)}$

$\underbrace{11}_{3}\underbrace{111}_{7}\underbrace{000}_{0}, \underbrace{010}_{2}\underbrace{001}_{1}\underbrace{1}_{4} = 370,214_{(8)}$

Por esta razón, la base octal se puede utilizar como una manera compacta de escribir números binarios, o cualquiera otra información codificada en binario.

Como la unidad por encima del bit es el byte, generalmente es más cómodo utilizar la base 16, o **hexadecimal**. Un desarrollo similar al anterior nos conduciría a que la equivalencia se obtiene agrupando los bits de cuatro en cuatro. Así, los dos números anteriores se escribirían del siguiente modo en hexadecimal:

$\underbrace{10}_{2}\underbrace{0110}_{6} = 26_{(16)}$

$\underbrace{1111}_{F}\underbrace{1000}_{8}, \underbrace{0100}_{4}\underbrace{011}_{6} = \mathrm{F8},46_{(16)}$

La representación hexadecimal es más compacta que la octal; tiene el inconveniente de necesitar, además de los diez dígitos decimales, seis «dígitos» más, A, B, C, D, E, F, cuyos valores decimales respectivos son 10, 11, 12, 13, 14 y 15. Con un poco de práctica, sin embargo, no es difícil interpretar expresiones como «efemil acientos benta y ce» (FABC). Más laborioso, absolutamente inútil y sólo interesante para adictos a pasatiempos extravagantes, sería aprender a operar aritméticamente con esta base.

Para eludir el uso de subíndices se pueden seguir distintas notaciones. La más común (derivada de Unix y el lenguaje C) es utilizar el prefijo «0x» para indicar hexadecimal, «0b» para binario y «0» para octal. Si el primer carácter es un dígito entre 1 y 9 se entiende que es decimal. He aquí dos ejemplos de interpretación decimal de un número binario pasando por la representación hexadecimal:

$$0b\underbrace{10}_{2}\underbrace{1001}_{9}\underbrace{1110}_{E}\underbrace{1111}_{F} = 0x29EF = 2\times16^3 + 9\times16^2 + 14\times16 + 15 = 10735$$

$$0b\underbrace{111}_{7}\underbrace{1010}_{A}\underbrace{0000}_{0}\underbrace{1100}_{C}\underbrace{1101}_{D} = 0x7A0CD = 7\times16^4 + 10\times16^3 + 12\times16^1 + 13 = 499917$$

# Apéndice B

# 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`.

## B.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 Unix

«Mono» es un proyecto de código abierto que implementa .NET. Permite ejecutar el programa ARMSim# en un sistema operativo de tipo Unix: 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 un editor de textos, 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).

## B.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 B.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 Unix: 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 sólo 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 instrucción `SWI` leyendo el código (número) que la acompaña. En la tabla B.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 B.1**: Códigos de SWI

## B.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 10.9: ésta es una actividad instructiva, pero no pretenda programar aplicaciones en lenguaje ensamblador.

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 10.9, 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 Unix y también en Windows, por ejemplo, con Cygwin (`http://www.cygwin.com/`), que 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`
  - en Debian: `http://www.emdebian.org/`

  Esta última es la que se ha usado para generar los listados del capítulo 10.
- 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 sólo se quiere un listado que sea más legible, como se explica en el último párrafo del apartado 10.1.

  `-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. Se puede descargar libremente de `http://developer.android.com/sdk/`, y lo incluimos aquí porque en el apartado 10.9 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/`.

# Apéndice C

# Prácticas de laboratorio

Las prácticas están concebidas para realizarlas en sesiones de entre una y dos horas[1]. El laboratorio dispone de equipos conectados en red y a Internet con sistema operativo Linux, distribución Ubuntu. Se utiliza la plataforma Moodle para que al final de la sesión el alumno suba un fichero de texto con las órdenes tecleadas (sesiones Lab1 y Lab2) o una plantilla en formato ODT rellena con los resultados y las respuestas a las cuestiones que se plantean (en las demás sesiones).

Se recomienda al alumno que previamente trate de realizar cada práctica con sus propios medios o en el mismo laboratorio, en salas y horario de libre acceso, de modo que la sesión de laboratorio tenga la función de una clase de tutoría para resolver dudas.

Se reproduce a continuación lo esencial de la documentación que se entrega a los estudiantes previamente a cada sesión, eliminando algunos detalles específicos del entorno del laboratorio. En particular, se han editado los documentos de las sesiones Lab3 a Lab7 omitiendo las «plantillas» e incluyendo en su lugar las preguntas (en negrita) en la misma descripción de la práctica.

## Lab1: Trabajo con ficheros

### Objetivos

- Familiarizarse con la línea de comandos (órdenes) de Unix (CLI, apartado 2.4).
- Conocer los comandos básicos para trabajar con ficheros.
- Saber utilizar un editor de texto.
- Aprender a consultar las páginas del manual.

### Inicio

Como ya ha comprobado al darse de alta, los ordenadores del laboratorio están configurados para que al entrar en su cuenta le ofrezcan un entorno gráfico («gnome») desde el cual se puede abrir en una ventana un terminal de texto. Todas las actividades de esta práctica se pueden hacer en ese terminal de texto, pero, para que las facilidades que ofrece el entorno gráfico no le distraigan, es mejor que realice las actividades de las primeras sesiones de laboratorio desde un terminal de texto «puro». Para ello:

[1] Por limitaciones de disponibilidad de recursos, algunas de las prácticas que aquí se proponen como separadas, concretamente Lab1+Lab2 y Lab3+Lab4, se realizan en una sola sesión de dos horas.

1. Pulse *simultáneamente* tres teclas: Ctrl, Alt y F1[2]. La pantalla se convertirá en una consola de texto, con letras blancas sobre fondo negro, pidiéndole de nuevo su nombre de *login* y contraseña. En cualquier momento puede volver al entorno gráfico pulsando simultáneamente Alt y F7, y retornar de éste a la consola con Ctrl+Alt+F1.
2. Una vez identificado le saldrá la invitación («prompt») de Unix: `su_login$`

En lo sucesivo, el icono significa que debe usted escribir la orden indicada.

Antes de empezar a practicar con las órdenes básicas, escriba:

`history -c`

Hay que dejar al menos un espacio de separación entre el nombre de la orden (`history`) y la opción (`-c`), y pulsar la tecla «Intro» (⟵) al final (todas las órdenes se terminan con esta tecla). Observará que no hay ninguna respuesta (vuelve a salir la invitación), pero esta orden borra (`c` = clear) el historial de órdenes anteriores, de modo que al final de esta sesión de laboratorio se habrá generado un historial de las órdenes que usted ha tecleado.

Escriba su nombre y apellidos:

`Nombre Apell1 Apell2`

El intérprete de órdenes («shell») le dirá que no conoce ninguna que se llame como su nombre (salvo que usted tenga un nombre extraño que coincida con el de alguna orden). Pero la (falsa) orden ha quedado registrada. Escriba a continuación:

`history`

y fíjese en el resultado.

## Observaciones generales

- A continuación deberá ir escribiendo una serie de órdenes que quedarán registradas en el historial. Si se equivoca al teclear y el intérprete de órdenes le dice que no existe esa orden no se preocupe y escríbala correctamente. No es necesario que borre todo el historial anterior y repita lo que ya ha hecho.

- En algún momento puede ser que el intérprete se quede «bloqueado» y sin volver a dar la invitación (esto ocurre si, por ejemplo, ha olvidado completar la orden con algún parámetro). Para conseguir que vuelva a la invitación (y así anular dicha orden) pulse `^c` (pulsación de la tecla «c» mientras se mantiene pulsada la «Ctrl»).

- Para repetir una orden ejecutada previamente, sin necesidad de volver a escribirla, puede utilizar la teclas «arriba» (↑) y «abajo» (↓): con pulsaciones sucesivas de ↑ se le irán mostrando la última

[2]Puede ser F1, F2... F6: se pueden utilizar hasta seis consolas, y se puede cambiar de una a otra con Alt+Fi. En algunas distribuciones de Linux la consola «F1» no está disponible, porque queda bloqueada por los programas que lanzan el entorno gráfico: en lugar de una sola línea pidiendo el *login*, la pantalla está llena de mensajes de texto. En ese caso, pulse Alt y F2 para cambiar a la siguiente consola.

orden, la anterior, etc., y con ↓ se moverá en sentido inverso. Puede confirmar la repetición de una orden con «Intro» (⟵), y, si es necesario, editar antes sus opciones o parámetros con las teclas «←», «→», «Supr» y «Retroceso».

## Listar ficheros

Compruebe los resultados de estas órdenes (tenga cuidado de dejar al menos un espacio de separación entre la orden, `ls`, y la opción que le sigue, que empieza por «-»):

| Orden | Resultado |
|---|---|
| `ls` | Lista los nombres de los ficheros y directorios que hay en el directorio actual. |
| `ls -l` | Listado largo en el que, además de los nombres, aparecen datos asociados a los ficheros y directorios: permisos, número de enlaces, nombres del propietario y del grupo, tamaño en bytes y fecha y hora de la última modificación. |
| `ls -a` | Incluye ficheros y directorios ocultos: aquellos cuyo nombre empieza por un punto. |
| `ls -l -a`<br>o bien:<br> `ls -la` | Combina las dos opciones anteriores. |

De momento, y si no ha creado aún ningún fichero en su cuenta, `ls` y `ls -l` no dan ningún resultado. (En los ordenadores del laboratorio aparece un directorio llamado «Escritorio» que contiene ficheros y/o enlaces relacionados con el entorno gráfico).

## Crear y editar ficheros de texto

Vamos a utilizar un editor de texto sencillo que se llama «`nano`»:

`nano prueba`

(En este caso, `prueba` no es una opción, sino un **parámetro**, o **argumento**, para `nano`. Las opciones se distinguen porque empiezan con «-»).

Si el fichero `prueba` ya existía se muestra su contenido. Supuesto que no, casi toda la ventana estará en blanco, y el cursor en la parte superior, esperando alguna entrada de teclado. Las dos últimas líneas resumen las acciones más comunes. El símbolo «^» representa a la tecla «Ctrl». Por ejemplo, «`^G`» significa «pulsar simultáneamente las teclas `Ctrl` y `G`» (lo que lleva a mostrar una ayuda más extensa).

Escriba una palabra de cuatro letras, guarde el fichero (`^O`, pide confirmación, que se le da con «⟵») y salga del editor (`^X`).

`ls -l`

Verá que el fichero `prueba` tiene cinco bytes[3]. Esto es porque al guardarlo se ha añadido un carácter invisible: el «retorno» (pero si usted ha pulsado la tecla «⟵» antes de guardar el fichero se habrá introducido otro carácter de retorno adicional).

Vuelva a abrir `nano` para crear y editar otro fichero de texto:

[3] Si la palabra contenía letras con tilde, eñe, etc., puede que tenga más bytes. En el tema de representación de la información veremos por qué.

```
nano prueba1
```

Inserte en él, con ^R (read), el fichero prueba. Vamos a «engordar» prueba1 de modo que contenga al menos cien líneas. Empiece por escribir varias líneas con cualquier texto (con la tecla Intro se introduce un carácter «retorno», que salta al comienzo de una línea nueva). Compruebe que puede mover el cursor a cualquier parte con las teclas de flecha, y que puede insertar texto, así como modificarlo con las teclas «←» (borra el carácter anterior al cursor) y «Supr» (o «Del») (borra el carácter que está bajo el cursor).

Como se trata de obtener un fichero con texto arbitrario, para no perder el tiempo escribiendo cien líneas, haga uso de otra facilidad del editor: Sitúese en cualquier punto y pulse «^K». Esto «corta» la línea actual, pero la conserva en memoria. Pulse luego «^U»: esto «pega» esa línea delante de aquella en la que se encuentre el cursor. Por tanto, si mantiene dos o tres segundos pulsadas las teclas «^» y «U», rápidamente se insertarán más de cien líneas.

Salve el fichero (pulse «^O» y confirme que desea guardarlo con el nombre prueba1), salga de nano («^X»), y con ls -l compruebe el número de bytes de prueba1.

Otra orden útil para conocer el tamaño de los ficheros de texto es wc, que da el número de líneas, de palabras y de bytes. Compruébelo con estas dos órdenes:

```
wc prueba
wc prueba1
```

## Ver el contenido de un fichero de texto

Abriendo el fichero con un editor, como nano, se puede ver su contenido, pero si sólo se trata de leerlo (no editarlo) se pueden utilizar estas órdenes:

```
cat prueba
```

le mostrará el contenido del fichero prueba.

Sin embargo, observe que con

```
cat prueba1
```

únicamente se pueden visualizar las últimas líneas que caben en la pantalla. Para ver todo, desde el principio, se puede utilizar un *paginador* como more:

```
more prueba1
```

muestra las primeras líneas y pulsando la barra espaciadora salta a las siguientes, y con la tecla b (back) a las anteriores. Termina cuando se avanza hasta el final o con la tecla q (quit). También se puede, con ciertas teclas, avanzar o retroceder línea a línea o un determinado número de líneas, pero las teclas de flechas no funcionan. Generalmente es preferible otro paginador, less, en el que sí lo hacen, y es más versátil:

```
less prueba1
```

Para avanzar (y retroceder) línea a línea utilice las teclas de flecha abajo (y arriba); para avanzar una página, la barra espaciadora, y para retroceder, la tecla b. Para salir, la tecla q.

Salvo que se trate de un fichero pequeño, `cat` no tiene mucha utilidad para ver su contenido. Pero su función principal es otra: dándole como parámetros varios nombres de fichero, los concatena uno tras otro antes de enviarlos a la salida. En el Lab3 veremos cómo esa salida puede *redirigirse* a otro fichero en lugar de la pantalla.

### Copiar, mover y borrar ficheros

La orden `cp` (copy) se utiliza con dos parámetros, un *origen* y un *destino*:

```
cp prueba1 prueba2
```

hace una copia del fichero origen `prueba1` en el fichero destino `prueba2`. El fichero origen debe existir. Si el fichero destino no existe, lo crea, y si existe, lo sobrescribe.

La orden `diff` también se utiliza con dos nombres de fichero, y se utiliza para mostrar las diferencias, línea a línea, entre los contenidos de los ficheros. Haga una copia de `prueba1` (o de `prueba2`, que contiene lo mismo) en `prueba3`, edite (con `nano`) `prueba3`, modificando dos o tres líneas cualesquiera, y analice los resultados de:

```
diff prueba1 prueba2
diff prueba1 prueba3
```

La orden `mv` (move) con dos nombres de ficheros como parámetros renombra el primer fichero sin cambiar su contenido:

```
mv prueba pp
```

simplemente cambia el nombre de `prueba` para que pase a llamarse `pp`

Finalmente, `rm` (remove) borra uno o varios ficheros. Compruébelo con:

```
ls
rm pp prueba2
ls
```

Observe que el mismo resultado de la orden para cambiar el nombre de `prueba` (`mv prueba pp`) se puede conseguir con «`cp prueba pp`» seguida de «`rm prueba`».

Esta orden (`rm`) debe utilizarse con cuidado, porque los ficheros borrados no pueden recuperarse y, además, no pide confirmación: para que lo haga está la opción «`-i`». En algunas configuraciones está redefinida para que por defecto se utilice con esta opción. En otras se define una orden «`del`», equivalente a «`rm -i`».

### Buscar en el contenido de ficheros de texto

Hay varias posibilidades para buscar una cadena de caracteres en un fichero de texto:

- Con un editor:

  ```
  nano prueba1
  ```

  Vaya al final del fichero (con la flecha, o con `^V`, que avanza página a página) e inserte una palabra nueva. Luego retroceda al principio (con la flecha, o con `^Y`, que retrocede página a página). Para buscar la palabra pulse `^W`. Salve el fichero (`^O`) y salga (`^X`).

- Dentro de `less`:

```
less prueba1
```

Escriba «/» seguido de la palabra que insertó antes y un retorno y observe el resultado. Salga de `less` (`q`).

- `grep` es un programa que tiene muchas opciones para buscar cadenas de caracteres en ficheros de texto. Por ejemplo, `grep -n <cadena> <fichero>` (donde «`<cadena>`» es una cadena de caracteres y «`<fichero>`» es un nombre de fichero) escribe las líneas del fichero que contienen esa cadena precedidas de sus números de línea (opción `-n`). Si la palabra que insertó antes es, por ejemplo, «Pepe», compruébelo con

```
grep -n Pepe  prueba1
```

## Uso de comodines

Los caracteres «*» y «?» tienen un significado especial para el intérprete de órdenes, y se llaman «comodines» (wildcards).

«*» casa con cualquier secuencia de caracteres. Véalo con:

```
ls p*
```

«?» se empareja con un solo carácter:

```
ls p?
```

Si no hay ningún fichero que empiece por «p» y tenga exactamente dos caracteres no se obtiene nada. Haga algunos:

```
cp prueba1 pp
cp prueba1 p1
cp prueba1 pa
cp prueba1 pab
```

y observe el resultado de:

```
ls p?
```

Finalmente, pruebe esto:

```
rm -i p?
ls p*
```

(la opción `-i` hace que pida confirmación antes de borrar) y luego esto otro:

```
rm -i p*
ls p*
```

## Moverse por la historia de órdenes

En este momento usted ha tecleado unas 40 órdenes que han quedado guardadas en una zona de memoria reservada por el intérprete de órdenes. Puede listarlas con

```
history
```

Pero observe que sólo puede ver las últimas órdenes, las que caben en la pantalla. En los siguientes laboratorios aprenderá cómo verlas todas[4].

Puede repetir la orden que tiene el número n escribiendo «`!n`». Pero es más conveniente el uso de las teclas de flechas arriba y abajo («↑» y «↓»): como ya hemos adelantado en las «observaciones generales» (página 274), con ellas puede usted moverse por la historia de órdenes. Puede confirmar la repetición de una orden con «⟵», y, si es necesario, editar antes sus opciones o parámetros (con las teclas «←», «→» y «Supr») .

## Las páginas de manual

Casi todas las órdenes de Unix tienen un nombre corto y admiten muchas opciones. Una orden muy útil para obtener ayuda sobre cualquier orden es `man`. Así, para obtener toda la información sobre `ls` escriba:

```
man ls
```

`man` utiliza normalmente `less` para paginar, lo que resulta útil para localizar informaciones en las páginas de manual, a veces muy extensas: escriba «`/size`» y «⟵» y verá que le lleva a la primera aparición de «size»[5]. Pulsando «n» irá a la siguiente, y con «N» a la anterior.

Igual que con `less`, la tecla `q` hace salir de `man` y devuelve a la invitación del intérprete de órdenes.

## Entrega de resultados

Escriba esto:

```
history -w Apell1-Apell2-L1.txt
```

Esto hará que la historia se grabe en el fichero `Apell1-Apell2-L1.txt`. Compruébelo con

```
less Apell1-Apell2-L1.txt
```

Un truco: cuando, después de «`less`», haya escrito los dos o tres primeros caracteres del nombre del fichero pulse la tecla «Tab» (⇆).

Pues bien, se trata de subir este fichero al Moodle. Lo más sencillo es que vuelva al entorno gráfico (recuerde: Alt+F7) y lo haga desde el navegador Firefox. Pero antes de volver a ese entorno gráfico, cierre su sesión abierta en este terminal de texto con:

```
logout
```

(De lo contrario, y aunque luego cierre la sesión desde el entorno gráfico, el terminal de texto queda abierto y cualquiera podría acceder a su cuenta con Ctrl+Alt+F1, o Ctrl+Alt+F2, etc.)

[4] Si quiere hacerlo ya, escriba «`history | less`». El símbolo «|» se obtiene pulsando «AltGr» y «1».

[5] Esto, suponiendo que su sistema tiene las páginas de manual en inglés. Si las tiene en español, escriba «tamaño».

## Resumen

| Orden | Uso |
|---|---|
| `ls` | Lista nombres de ficheros y directorios<br>*Algunas opciones:*<br>`-l`: Listado largo<br>`-a`: Incluye ficheros ocultos |
| `nano [<fichero>]` | Editor de texto |
| `wc <fichero>` | Cuenta de líneas, de palabras y de bytes<br>*Algunas opciones:*<br>`-w`: Cuenta palabras; `-l`: Cuenta líneas<br>`-c`: Cuenta bytes; `-m`: Cuenta caracteres |
| `cat <f1> <f2> ...` | Concatena ficheros y envía el resultado a la salida |
| `more <fichero>` | Paginador para ver el contenido de ficheros de texto |
| `less <fichero>` | Otro paginador, con más facilidades que `more` |
| `cp <f1> <f2>` | Copia el fichero `<f1>` en el `<f2>`, creando éste si no existe<br>*Algunas opciones:*<br>`-i`: si `<f2>` existe, pregunta antes de sobre escribirlo<br>`-b`: si `<f2>` existe, hace una copia (backup) previa con nombre `<f2>~` |
| `diff <f1> <f2>` | Lista las diferencias en los contenidos de `<f1>` y `<f2>` |
| `mv <f1> <f2>` | `<f1>` pasa a llamarse `<f2>` |
| `rm <f1> <f2> ...` | Borra los ficheros<br>*Opción:* `-i`: pide confirmación antes de borrar |
| `grep <cad> <f>` | Escribe las líneas de `<f>` que contienen la cadena de caracteres `<cad>`<br>*Algunas opciones:*<br>`-i`: Ignora la caja (mayúsculas = minúsculas)<br>`-v`: Escribe las líneas que *no* contienen la cadena<br>`-n`: Escribe también los números de las líneas |
| `man <orden>` | Página de manual de la orden<br>*Algunas opciones:*<br>`-a`: Si hay varias páginas para la orden las presenta secuencialmente<br>`-k`: Seguida de una palabra (y sin nombre de orden) responde con todas las órdenes que mencionan esa palabra |

**Notas:**

- Los símbolos «<» y «>» en los nombres de parámetros no deben escribirse: «`<orden>`» significa «un nombre cualquiera de orden». «`[<fichero>]`» significa que el nombre de fichero es opcional.
- Las opciones deben ir antes que los parámetros.
- No olvide separar con espacios el nombre de la orden, las opciones y los parámetros.

# Lab2: Trabajo con directorios

## Objetivos

- Conocer las órdenes básicas para crear y borrar directorios y moverse por el árbol de directorios.
- Entender la forma de identificar ficheros y directorios mediante rutas absolutas y relativas.
- Realizar un ejercicio completo para comprobar que ha asimilado las funciones de las órdenes y el concepto de rutas y cómo se utilizan.

## Inicio

Como en la práctica anterior, abra una consola de texto (Ctrl, Alt, F1).

El árbol del sistema de ficheros parte del directorio raíz (`/`). De este directorio cuelgan varios subdirectorios (figura 2.11). Uno de ellos es el directorio `/home`, del que cuelgan a su vez los directorios personales de los usuarios. Y de cada directorio personal cuelgan los ficheros y directorios de ese usuario.

El nombre del directorio personal es diferente para cada usuario y se corresponde con el nombre de usuario asignado. Por ejemplo, «`/home/juan.garcía`».

En esta práctica veremos cómo moverse por el árbol de directorios. En un momento determinado se encontrará situado en un directorio al que llamaremos *directorio de trabajo actual*. Cuando un usuario entra en su cuenta, el sistema sitúa su directorio de trabajo actual en su directorio personal.

Así pues, si después de entrar en su cuenta teclea usted

`ls`

observará que conserva los ficheros que no borró de la sesión anterior (en la que no se movió de su directorio personal).

En lo sucesivo sustituiremos el icono del teclado por la invitación por defecto del sistema, «$».

## Creación de directorios

Para crear subdirectorios se utiliza la orden `mkdir`:

```
$ mkdir Lab2
```

crea el subdirectorio `Lab2`, que cuelga del directorio de trabajo actual, que en este momento, si no ha ejecutado ninguna orden diferente a las anteriores, es su directorio personal. Por tanto, se crea el directorio `Lab2`, que cuelga de su directorio personal.

Compruébelo tecleando:

```
$ ls -l
```

## Cambiar de directorio

Con la orden `cd` se cambia el directorio de trabajo, es decir se mueve en el árbol del sistema de ficheros. Cambie el directorio de trabajo actual a `Lab2`, cambiando por tanto su posición actual en el árbol del sistema de ficheros:

```
$ cd Lab2
```

Mire el contenido de ese directorio, que debe estar vacío:

```
$ ls -l
```

En Unix «.» significa el directorio actual y «..» el directorio inmediatamente superior, es decir, el padre del actual directorio de trabajo. Suba al directorio padre del actual, que, si no hay tecleado nada más, es su directorio personal, tecleando:

```
$ cd ..
```

Independientemente de dónde se encuentre en un momento dado en el árbol del sistema de ficheros, podrá cambiarse a su directorio personal tecleando `cd` sin opciones. Esto es muy útil cuando se está perdido en el árbol del sistema de ficheros.

La orden «`cd /`» le lleva al directorio raíz (lo que normalmente no tiene mucha utilidad; para ver lo que cuelga de la raíz basta hacer «`ls -l /`»).

## Rutas

Se identifica a un fichero o directorio con la ruta o camino (*path*) de ese fichero o directorio en el árbol del sistema de ficheros. La ruta puede ser absoluta o relativa. Una ruta absoluta señala la localización exacta de un fichero o directorio en el árbol del sistema de ficheros desde la raíz. Una ruta relativa señala la localización exacta de un fichero o directorio en el árbol del sistema de ficheros desde el directorio de trabajo actual.

Por ejemplo, la identificación del directorio `Lab2` que ha creado previamente se puede hacer mediante su ruta absoluta: `/home/nombre-cuenta/Lab2` o mediante su ruta relativa. Si, por ejemplo, su directorio de trabajo actual es su directorio personal la ruta relativa es `Lab2`; si su directorio de trabajo actual es `/home` la ruta relativa es `nombre-cuenta/Lab2`.

Mediante la orden `pwd` puede saber dónde se encuentra en cada momento en el árbol del sistema de ficheros. Si no se ha cambiado de directorio y sigue en su directorio personal, tecleando `pwd` conocerá la ruta absoluta de su directorio personal:

```
$ pwd
```

Utilice las órdenes `cd`, `ls` y `pwd` para explorar el sistema de ficheros. Recuerde que con la orden `cd` sin opciones se cambia a su directorio personal.

## Entendiendo las rutas

Se puede referir a su directorio personal con el carácter «~»[6]. Para cambiarse al directorio `Lab2` que ha creado antes, independientemente de dónde se encuentre actualmente en el árbol del sistema de ficheros, escriba:

```
$ cd ~/Lab2
```

Abra con un editor de texto un fichero nuevo que se llame `prueba`:

```
$ nano prueba
```

Escriba una línea con un texto cualquiera en dicho fichero y salga del editor.

Ahora teclee sucesivamente:

```
$ cd ..
$ ls -l
$ ls -l prueba
```

[6] «`cd`» es equivalente a «`cd ~`» y a «`cd ~/`».

Mediante estás órdenes se ha movido al directorio padre de `Lab2`, que es su directorio personal, ha listado los nombres de ficheros y directorios que se encuentran de él, y por último ha ordenado generar un listado largo del fichero `prueba`.

Como resultado de la última orden habrá obtenido un mensaje que le indica que no existe el fichero o directorio `prueba`. La razón es que `prueba` no se encuentra en el directorio actual de trabajo, su directorio personal, sino que cuelga del directorio `Lab2`.

Para identificar el fichero o directorio que va a utilizar una orden, por ejemplo la orden `ls -l` sobre el fichero `prueba`, hay que especificar la ruta absoluta o relativa ese fichero o directorio. La ruta relativa depende del directorio actual de trabajo. Por tanto, hay varias alternativas:

- Con ruta absoluta desde cualquier directorio:

  ```
  $ ls -l /home/nombre-cuenta/Lab2/prueba
  ```

- Con ruta relativa desde el directorio personal:

  - Si ya se encuentra en este directorio,

    ```
    $ ls -l Lab2/prueba
    ```

  - Desde cualquier directorio,

    ```
    $ ls -l ~/Lab2/prueba
    ```

Pruebe a volver a utilizar todas las órdenes anteriores con otros ejemplos para familiarizarse con ellas, y muévase por los directorios del sistema para comprender el árbol del sistema de ficheros y cómo moverse en el mismo.

### Borrar directorios

Con la orden `rmdir` se borran directorios. En concreto, si se teclea `rmdir nombre-directorio`, se borra `nombre-directorio`, pero únicamente si está vacío.

Como hemos visto en el laboratorio anterior, `rm` sirve, en principio, para borrar ficheros. Pero si se utiliza con la opción `-r` y un nombre de directorio, borra, además de ese directorio, todos los ficheros y directorios que cuelgan de él, de forma recursiva. Esta opción es muy peligrosa ya que puede, sin desearlo, borrar mucha información si se equivoca al indicar el nombre del directorio que quería borrar.

Si `rm` se utiliza con la opción `-i`, se pide confirmación antes de proceder al borrado

Sitúese en su directorio personal y borre el directorio `Lab2` y los ficheros que cuelgan de él:

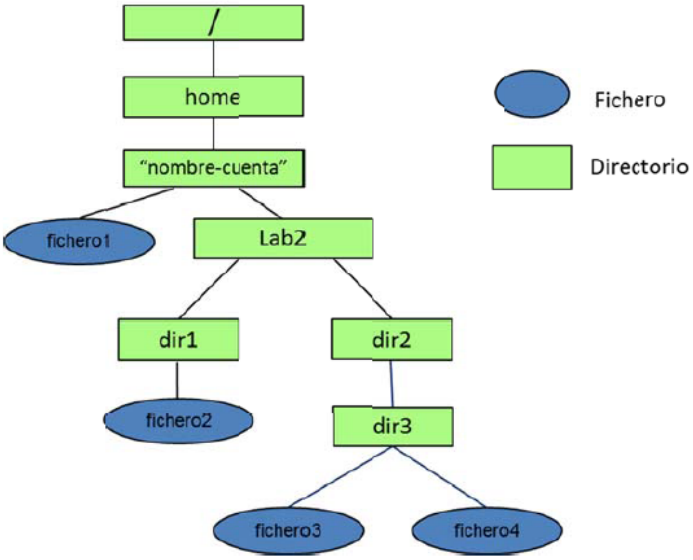
```
$ cd
$ rm -ri Lab2
```

## Ejercicio a realizar

Borre la historia de las órdenes que ha tecleado previamente con `history -c` para que, como en la primera sesión de laboratorio, se genere un historial de las órdenes que deberá subir al Moodle al final de ésta.

1. Escriba su nombre y apellidos, para que queden registrados en la historia.

2. Utilizando `mkdir` y el editor `nano`, cree la siguiente estructura de directorios y ficheros:




Todos los ficheros deben tener dos líneas con este contenido:

Su nombre y dos apellidos
Nombre del fichero

Cambie su directorio de trabajo actual a su directorio personal y compruebe, con la orden `tree`, que ha creado la estructura anterior. Observe que además de los ficheros y directorios creados aparece un directorio «`Escritorio`» con los enlaces de su entorno gráfico.

3. Sin moverse de su directorio personal y **utilizando rutas absolutas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias para:
   - Listar en la pantalla el contenido del `fichero1` (sin utilizar un editor).
   - Ver un listado largo con las características (nombre, permisos, tamaño...) de los ficheros que hay en el directorio `dir3`.
   - Editar el `fichero4` y añadir el nombre de esta asignatura.
   - Sacar por pantalla el contenido del `fichero2` (sin utilizar un editor).
   - Cambiar el nombre del `fichero3` a `fichero5`.
4. Sin moverse de su directorio personal y **utilizando rutas relativas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias para:
   - Listar en la pantalla el contenido del `fichero1` (sin utilizar un editor).
   - Ver un listado largo con las características (nombre, permisos, tamaño...) de los ficheros que hay en el directorio `dir3`.
   - Editar el `fichero4` y añadir la fecha actual.
   - Sacar por pantalla el contenido del `fichero2` (sin utilizar un editor).
   - Cambiar el nombre del `fichero4` a `fichero6`.
5. Compruebe con la orden `tree` que los cambios de nombre de los ficheros se han realizado correctamente.

6. Sitúese en el directorio `dir1`. Sin moverse de él, y **utilizando rutas absolutas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias para:
   - Crear un nuevo fichero llamado `fichero7` que cuelgue del directorio `dir2` y cuyo contenido sea el mismo que el de `fichero1` (sin utilizar un editor).
   - Borrar todos los ficheros cuyo nombre comience por `fich` y se encuentren en el directorio `dir3`.

7. Sin salir de `dir1`, y **utilizando rutas relativas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias para:
   - Crear un nuevo fichero llamado `fichero8` que cuelgue del directorio `dir3` y cuyo contenido sea el mismo que el de `fichero1` (sin utilizar un editor).
   - Borrar todos los ficheros cuyos nombres comiencen por `fich` y que se encuentren en el directorio `dir2`.

8. Cambie su directorio de trabajo actual a su directorio personal y compruebe con la orden `tree` que los cambios realizados se han ejecutado correctamente.

   Grabe la historia de todas las órdenes que ha ejecutado para resolver este ejercicio en un fichero de texto: `history -w Apell1-Apell2_L2.txt`

   Compruebe el contenido del fichero con `less` y suba al Moodle el fichero generado.

9. Para que le quede clara la estructura de ficheros y directorios con la que ha trabajado durante el laboratorio en la consola de texto, abra en el entorno gráfico un explorador de archivos (Nautilus) y muévase por el árbol de ficheros y directorios.

## Resumen

| **Orden** | **Uso** |
|---|---|
| `mkdir <dir>` | Crea el directorio `<dir>` |
| `cd <dir>` | Cambia el directorio de trabajo actual a `<dir>` |
| `cd` | Cambia el directorio de trabajo actual al directorio personal |
| `cd ~` | Cambia el directorio de trabajo actual al directorio personal |
| `cd ..` | Cambia el directorio de trabajo actual al directorio padre |
| `cd /` | Cambia el directorio de trabajo actual al directorio raíz |
| `pwd` | Muestra la ruta absoluta del directorio de trabajo actual |
| `rmdir <dir>` | Borra un directorio vacío |
| `rm <dir>` | Borra ficheros. *Algunas opciones:*<br>`-r`: borra de forma recursiva todos los ficheros y directorios que cuelgan del directorio a borrar<br>`-i`: pide confirmación antes de borrar |
| `tree` | Lista recursivamente en forma de árbol los ficheros, directorios y contenidos de los directorios |

# Lab3: Redirección, protección y procesos

## Objetivos

- Aprender a redirigir la entrada y la salida y a encadenar comandos.
- Saber interpretar y establecer los permisos de acceso a ficheros y directorios.
- Conocer los comandos básicos que permiten gestionar los procesos.

## Inicio

A partir de esta sesión, para mayor comodidad, trabajaremos con el entorno gráfico (gnome). Abra un terminal de texto con los menús de la barra superior: «Aplicaciones/Accesorios/Terminal».

Aplicando los conocimientos que adquirió en la práctica anterior, cree un directorio nuevo de nombre `Lab3`, subdirectorio de su directorio personal, y muévase a él. Todas las actividades de esta sesión se realizarán dentro de `~/Lab3`.

## Redirección

Muchos de los procesos iniciados al ejecutar una orden leen los datos que necesitan de la entrada estándar, escriben en la salida estándar y envían los mensajes de error a la salida de errores. Por defecto, la entrada estándar es el teclado, y las salidas estándar y de errores es la pantalla.

En el primer laboratorio usó la orden `cat prueba` para ver el contenido del fichero `prueba` en la pantalla. Al ir acompañada de un nombre de fichero, `cat` lee los datos del fichero y los envía a la salida estándar, es decir la pantalla. Si únicamente escribimos `cat` sin indicar un nombre de fichero, lee los datos no ya de un fichero, sino de la entrada estándar, es decir del teclado, y los sigue sacando por la pantalla. Pruébelo:

```
$ cat
```

A continuación escriba palabras en el teclado, por ejemplo nombres de frutas, todas ellas separadas por Intro, es decir cada una en una línea. Finalmente teclee `^D` (Ctrl-D, que indica el fin de fichero) para indicar que se termina la entrada de texto por teclado. Observe que a medida que introduce una palabra seguida de Intro, esa palabra se saca inmediatamente por la pantalla, viéndose duplicada.

Se puede redirigir tanto la entrada como la salida de las órdenes. En el ejemplo anterior, en el primer caso, la orden `cat` lee los datos del fichero que se pone a continuación y en el segundo caso, al no indicársele la entrada, lee los datos de la entrada estándar, es decir del teclado. En ambos casos la salida de datos se produce por la salida estándar, es decir la pantalla.

## Redirección de la salida

Se utiliza el símbolo «>» para redirigir la salida de una orden. Por ejemplo, para crear un fichero de nombre `frutas` con una lista de nombres de fruta que se van introduciendo por el teclado, escriba:

```
$ cat > frutas
Pera
Manzana
Plátano
Melocotón
^D
```

Esta orden lee los datos de la entrada estándar, el teclado, y como la salida está redirigida («>») al fichero `frutas`, en lugar de sacarlos por la salida estándar, es decir la pantalla, los redirige al fichero `frutas`. Si `frutas` no existe lo crea, y si ya existe borra previamente su contenido.

Compruebe el resultado:

```
$ cat frutas
```

Otro ejemplo: `sort` ordena alfabéticamente o numéricamente una lista. Escriba:

```
$ sort > verduras
Puerro
Coliflor
Lechuga
^D
```

Con orden se leen los datos de la entrada estándar (el teclado) se ordenan y se escriben en el fichero `verduras`.

Si ahora queremos añadir más frutas al fichero `frutas` y para ello ejecutamos `cat >frutas`, los nuevos nombres que escribamos en el teclado se escribirán en el fichero `frutas`, pero perderemos los que habíamos introducido antes. Para poder añadir datos a un fichero sin perder los que ya tiene utilizamos «>>»:

```
$ cat >> frutas
Cereza
Melón
Sandía
^D
```

Hemos añadido los nuevos nombres de frutas al fichero `frutas` sin borrar su contenido previo. Compruébelo:

```
$ cat frutas
```

Ahora ya tenemos dos ficheros: `frutas` con siete frutas y `verduras` con tres verduras ordenadas alfabéticamente. Podemos utilizar la redirección para unir (concatenar) los dos ficheros y crear uno nuevo con el contenido de ambos ficheros:

```
$ cat frutas verduras > comida
```

Lo que estamos haciendo es leer el contenido de los ficheros frutas y verduras y redirigir la salida al fichero comida. Compruébelo:

```
$ cat comida
```

Si queremos que el fichero comida tenga su contenido ordenado alfabéticamente podemos hacer:

```
$ sort frutas verduras > comida
```

Lo que hemos hecho es dar a la orden `sort` el contenido de dos ficheros, `frutas` y `verduras`; `sort` ordena los datos y la salida es redireccionada con «>» al fichero comida. Compruébelo:

```
$ cat comida
```

## Tuberías

Para contar el número de ficheros y directorios que hay en su directorio de trabajo actual puede teclear:

```
$ ls > nombres
$ wc -w < nombres
```

Con la primera orden se crea el fichero `nombres` con los nombres de los ficheros y directorios que cuelgan de su directorio de trabajo actual. En la segunda orden, «<» es una redirección de la entrada estándar; la orden cuenta el número de palabras del fichero `nombres`, obteniendo así el número de ficheros y directorios que hay en su directorio de trabajo actual. Tenga en cuenta que la primera orden crea primero el fichero `nombres` y luego ejecuta `ls`, por lo que `wc` cuenta también ese fichero `nombres`.

Pero ejecutar estás dos órdenes seguidas hace que el proceso sea más lento y que nos tengamos que acordar de borrar el fichero de trabajo, `nombres`, cuando hayamos terminado.

Para evitar estos problemas se pueden utilizar las tuberías *(pipes)* con el símbolo «`|`». Las tuberías conectan directamente la salida de una orden a la entrada de otra. Por ejemplo, podemos conseguir lo mismo tecleando:

```
$ ls | wc -w
```

La salida de la orden `ls` es la entrada de la orden `wc`, consiguiendo como resultado el número de ficheros y directorios que cuelgan del directorio de trabajo actual. En este caso no se cuenta el fichero `nombres`, ya que no se ha creado.

## Permisos de acceso en el sistema de ficheros

Habrá observado que con

```
$ ls -l
```

la primera información que se obtiene para cada fichero o directorio es algo como:

```
-rwxrw-r--
```

Es una cadena de diez símbolos que pueden tomar, entre otros, los valores `-`, `d`, `r`, `w` o `x`. El primer símbolo indica el tipo de fichero[7]:

- `-`: fichero regular.
- `d`: directorio.

Los nueve símbolos siguientes determinan los permisos de acceso y se agrupan así:

- El primer grupo de tres símbolos indica los permisos para el usuario propietario del fichero o directorio. En el ejemplo, `rwx`.
- El segundo grupo de tres símbolos indica los permisos para el grupo al que pertenece el propietario del fichero o directorio. En el ejemplo, `rw-`.

[7] Otros símbolos corresponden a otros tipos de ficheros especiales: `l` (enlace simbólico), `b` (fichero especial de bloques), `c` (fichero especial de caracteres), `s` (*socket*).

- El último grupo de tres símbolos indica los permisos para el resto de usuarios. En el ejemplo, `r--`.

Los símbolos `r`, `w` y `x` tienen diferentes valores en función de que sea un fichero o directorio. Si es un fichero indican:

- `r`: permiso de lectura, permiso para leer o copiar el fichero. Si en su lugar hay un `-`, no tiene permiso de lectura.
- `w`: permiso de escritura, permiso para modificar el fichero. Si en su lugar hay un `-`, no tiene permiso de escritura.
- `x`: permiso de ejecución, permiso para ejecutar el fichero. Si en su lugar hay un `-`, no tiene permiso de ejecución.

Si es un directorio los permisos de acceso indican:

- `r`: permiso para visualizar los ficheros en el directorio. Si en su lugar hay un `-`, no tiene permiso para visualizar los ficheros.
- `w`: permiso para borrar, crear o mover ficheros en el directorio. Si en su lugar hay un `-`, no tiene permiso para borrar, crear o mover los ficheros.
- `x`: permiso de acceso a los ficheros del directorio. Si en su lugar hay un `-`, no tiene permiso de acceso.

En el ejemplo anterior, `-rwxrw-r--` indican:

- El primer `-`, que es un fichero.
- `rwx`, que el propietario del fichero puede leerlo, escribirlo y ejecutarlo.
- `rw-`, que el grupo del propietario del fichero, puede leerlo y escribir sobre él, pero no ejecutarlo.
- `r--`, que el resto de usuarios sólo pueden leerlo.

### Cómo cambiar los permisos de acceso

Con `chmod` se pueden cambiar los permisos de un fichero. Los permisos de un fichero únicamente pueden ser cambiados por su propietario[8]. La orden `chmod` tiene que ir acompañada de dos parámetros:

- Tres dígitos en octal que indican los nuevos permisos que se le dan al fichero. El primer dígito corresponde a los permisos del usuario, el segundo dígito a los permisos del grupo y el último dígito a los permisos del resto de usuarios.
- El nombre del fichero cuyos permisos se cambian.

Por ejemplo: `chmod 764 fichero`. Cada dígito en octal, pasado a binario se representa con tres dígitos binarios (0 o 1):

[8] O por el superusuario, «root», apartado 2.4.

- El primer dígito binario indica si tiene (1) o no (0) permisos de lectura
- El segundo dígito binario indica si tiene (1) o no (0) permisos de escritura
- El tercer dígito binario indica si tiene (1) o no (0) permisos de ejecución

Los tres dígitos en octal que hemos puesto a la derecha de `chmod`, `764`, pasados a binario serían `111 110 100`. El 7 (111) indica que el propietario del fichero puede leerlo, escribirlo y ejecutarlo. El 6 (110), que el grupo puede leerlo, escribirlo pero no ejecutarlo. El 4 (100), que el resto de usuarios sólo pueden leerlo.

Ejecute:

```
$ chmod 440 frutas
$ ls -l
$ nano frutas
```

e intente modificar el nombre de una fruta.

Con la primera orden hemos cambiado los permisos al fichero `frutas` de forma que el propietario y el grupo sólo pueden leerlo y el resto de usuarios no pueden hacer nada con el fichero. Con la segunda orden comprobará el cambio de los permisos y con la tercera orden comprobará que el fichero no puede modificarse.

Si mira la descripción de `chmod` en el manual, observará que hay una alternativa a la hora de indicar los permisos que se cambian a un fichero. Consiste en utilizar símbolos como `+` (añadir), `-` (quitar), `r`, `w` o `x` en lugar de los tres dígitos en octal.

## Procesos

Como ya hemos estudiado (apartado 2.4), un proceso es un programa en ejecución. Todo proceso tiene un identificador único, el PID, así como identificadores de su propietario, UID, y del grupo de éste, GID.

La orden `ps` nos da información sobre los procesos de usuario que se están ejecutando en el terminal donde se ejecuta esa orden. En concreto, proporciona esta información para cada proceso: su PID, el terminal asociado al proceso (TTY), el tiempo de CPU acumulado y el nombre del programa.

Teclee:

```
$ ps
```

para ver los procesos de usuario que se están ejecutando en el terminal. Observe que hay dos, el propio `ps` y el intérprete de órdenes *(bash)*.

Para obtener datos de todos los procesos que se están ejecutando en el sistema:

```
$ ps -e
```

Al comando `ps` se le puede añadir la opción `-f` para que nos muestre más información de los procesos: el identificador del propietario del proceso (UID), el PID del proceso padre (PPID), el porcentaje de uso del procesador (C) y la hora de comienzo de ejecución del proceso en el sistema (STIME).

```
$ ps -f
& ps -ef
```

A veces es necesario matar un proceso, por ejemplo cuando se ha metido en un bucle sin fin. La orden `kill`, en general, sirve para enviar una señal a un proceso (lo que hace es generar la llamada `KILL`, apartado 2.4). La señal de matar se identifica con el parámetro «`-9`». Para comprobarlo, vamos a iniciar un proceso en un terminal y matarlo desde otro:

En el entorno gráfico, abra otro nuevo terminal de texto, con los menús de la barra superior: «Aplicaciones/Accesorios/Terminal» y teclee:

```
sleep 1000
```

La orden `sleep` espera el número de segundos que se le indica antes de terminar, en nuestro ejemplo 1000 segundos. Vuelva al primer terminal de texto y ejecute `ps -ef` para ver los procesos que se están ejecutando en el sistema. Entre ellos estarán el del nuevo terminal (`bash`) y el del `sleep` que ha lanzado desde él. Por lo tanto verá dos `bash`: el correspondiente al terminal en el que está trabajando en este momento y el del nuevo, donde se está ejecutando el proceso `sleep`. Puede saber qué PID tiene asignado el terminal en el que está situado en un momento dado con:

```
$ ps | grep bash
```

Para saber los PID de todos los terminales abiertos:

```
$ ps -e | grep bash
```

Apunte los PID del proceso `sleep` y del correspondiente al terminal donde se está ejecutando y mátelos (primero `sleep` y luego `bash`) con las órdenes:

```
$ kill -9 <PIDsleep>
$ kill -9 <PIDbash>
```

## Ejercicio a realizar

1. Cree un directorio de nombre `procesos` que cuelgue del directorio `Lab3`
2. En el entorno gráfico abra tres terminales de texto adicionales al terminal de texto con el que está trabajando actualmente. En cada uno de esos tres terminales elija una de estas tres órdenes (no repita órdenes):

   ```
   $ sleep 10000
   $ cat
   $ sort
   ```

   Con esto tendrá tres procesos: un `sleep`, un `cat` y un `sort`, todos ellos esperando, y cuatro terminales de texto.
3. Vuelva al terminal de texto primero, donde no tiene ningún proceso corriendo.

A partir de este momento, cada uno de los puntos que siguen debe realizarse *con una sola línea de comandos* (salvo el último), utilizando, si es necesario, tuberías y redirección.

1. Cambie su directorio de trabajo actual al directorio `procesos` que acaba de crear.
   **Línea:**

2. Sin cambiarse de directorio de trabajo (`procesos`), *utilizando rutas relativas* y las órdenes `ps` y `wc`, cree un fichero llamado `numero-procesos` en el directorio `Lab3`, cuyo contenido sea número de procesos que se están ejecutando en este momento en el sistema.
   **Línea:**

Si mira el número de procesos contados observará que aparecen dos más de los que se ven con `ps`. Uno es debido a que la orden `wc` cuenta la línea correspondiente a las cabeceras de la orden `ps` y otro a que hay un proceso extra, el correspondiente a la tubería (`|`), que no se ve.

3. Sin cambiarse de directorio de trabajo (`procesos`), y *utilizando rutas relativas*, cambie los permisos del fichero `numero-procesos` para que únicamente pueda ser leído y escrito por el propietario, y el resto de usuarios, incluidos los de su grupo, no puedan ni leerlo, ni escribirlo, ni ejecutarlo.
   **Línea:**

4. Sin cambiarse de directorio de trabajo (`procesos`), *utilizando rutas relativas*, añada al fichero `numero-procesos` la lista de procesos (en formato extendido) que se están ejecutando en este momento en el sistema, ordenados alfabéticamente.
   **Línea:**

5. Sin cambiarse de directorio de trabajo (`procesos`), *utilizando rutas absolutas*, añada al fichero `numero-procesos` el número de procesos que se están ejecutando en el sistema de los cuales es usted propietario (tienen su UID).
   **Línea:**

   Compruebe que el contenido del fichero `numero-procesos` es correcto y que sus permisos son los deseados.

6. Sin cambiarse de directorio de trabajo (`procesos`), *utilizando rutas absolutas*, cree el fichero `otros-procesos` con la lista de procesos que se están ejecutando en el sistema cuyo propietario no es *root*.
   **Línea:**

   Compruebe que el contenido del fichero `otros-procesos` es correcto.

7. Mate los procesos `sleep`, `cat` y `sort` y a continuación los tres terminales de texto que abrió previamente. *Tenga mucho cuidado para no matar también el terminal en el que está introduciendo las órdenes de la práctica* (para ello mire el PID de dicho terminal con `ps | grep bash` y asegúrese de que no es uno de los terminales que mata).
   **Líneas:**

## Resumen

| Orden | Uso |
|---|---|
| `orden > fichero` | Redirecciona la salida estándar a un fichero |
| `orden >> fichero` | Añade la salida estándar a un fichero |
| `orden < fichero` | Redirecciona a la entrada estándar el contenido de un fichero |
| `orden1 \| orden2` | Conecta la salida de la `orden1` a la entrada de la `orden2` |
| `cat fich1 fich2 > fich0` | Concatena `fich1` y `fich2` y escribe el resultado en `fich0` |
| `sort` | Ordena alfabética o numéricamente los datos |
| `chmod [3díg] fichero` | Cambia los permisos de acceso del fichero |
| `ps` | Lista los procesos de usuario en el terminal |
| `ps -ef` (o `ps ax`) | Lista los procesos en el sistema, en formato largo |
| `kill -9 [PID]` | Mata el proceso con identificador PID |
| `sleep [n]` | Espera `n` segundos para terminar |

# Lab4: Llamadas al sistema

## Objetivos

- Comprender la relación entre las funciones de biblioteca (API) y las llamadas al sistema (SCI) (figura 2.10).
- Ver la utilidad de la función `strace`.
- Observar las llamadas al sistema que se generan al ejecutarse un programa.

## Inicio

Analice el siguiente programa escrito en C:

```
/* Directivas para incluir declaraciones de funciones de biblioteca
   necesarias para las llamadas al sistema: */
#include <sys/types.h> /* para open y creat */
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>     /* para printf */
#include <unistd.h>    /* para read y write */
#include <stdlib.h>    /* para exit */

void main(int argc, char *argv[]) {
  char palabra[10];
  int  descr, n1, n2;
  if (argc !=2) { /* si no se ha introducido un nombre de fichero */
    printf("\nUso:  %s fichero\n\n", argv[0]);
    exit(1);
  }
  /* Si ha pasado la comprobación anterior... */
  if ((descr = open(argv[1], O_WRONLY)) == -1) /* Intenta abrir el fichero */
       descr = creat(argv[1], 0666);
      /* Si no existe, lo crea con permisos 666 */
  else   lseek(descr, 0, SEEK_END);    /* si existe, se va al final */
  printf("\nEscribe algo\n\n");
  n1 = read(1, palabra, 10);       /* lee la línea de stdin */
  n2 = write(descr, palabra, n1);  /* y la escribe en el fichero */
  exit(0);
}
```

El programa utiliza estas funciones de biblioteca:

- `printf`: escribe en la salida estándar de acuerdo con un formato.
- `open`: recibe el nombre de un fichero para abrir y el modo; si el fichero existe devuelve un descriptor (número natural), y si no devuelve -1.
- `creat`: crea un fichero y devuelve su descriptor.

- `lseek`: mueve la posición del puntero de lectura/escritura en un fichero («`0,SEEK_END`» significa que se vaya al final).
- `read`: lee de un fichero.
- `write`: escribe en un fichero.

Si quiere saber más sobre el uso correcto de estas funciones puede consultar el manual. Por ejemplo, «`man 3 printf`» (el «3» es necesario, porque algunos nombres están sobrecargados; así, «`printf`», además de una función de biblioteca, que se documenta en la sección 3 del manual, es también el nombre de un programa de utilidad que se documenta en la sección 1).

## Descripción de la práctica

A partir de este momento, responda a todas las preguntas que se van planteando.

1. Abra un terminal de texto y cree un subdirectorio de su directorio personal con nombre `Lab4`. Muévase a él y haga una copia del programa en un fichero de nombre `Lab4.c`.

   Compile el programa con la orden «`gcc -o Lab4 Lab4.c`»

   El fichero resultante (`Lab4`) debe tener permisos de ejecución para todos. Compruébelo.
   **Orden:**

2. Ejecute `Lab4` con la orden «`./Lab4`» y observe el resultado.
   **¿Qué sentencias se han ejecutado?**

3. Vuelva a ejecutar `Lab4` con la orden «`./Lab4 fichero`», donde «`fichero`» es cualquier nombre de un fichero que no exista previamente.

   El programa le solicita que escriba algo.
   **¿Qué sentencias se han ejecutado hasta este momento?**

   Escriba una palabra.
   **¿Qué sentencias se ejecutan desde el momento en que termina de escribir y pulsa el retorno, hasta el final?**

4. Compruebe (con `cat`, o con `less`), el contenido de `fichero`.
   **Contenido:**

5. Vuelva a ejecutar el programa y compruebe el nuevo contenido de `fichero`.
   **Contenido:**

6. Resuma en pocas líneas lo que hace el programa.
   **Resumen:**

Las funciones de biblioteca son las que generan las llamadas al sistema. Se puede seguir la generación de estas llamadas mediante el programa `strace`. Ejecute «`strace ./Lab4`». Observará que hay una sucesión de llamadas, empezando por `execve`. Puede mirar en la sección 2 del manual lo que hacen las llamadas (pero es difícil entenderlo sin conocer detalles del funcionamiento interno de Unix).

Nos vamos a fijar en las tres últimas llamadas: `write`, `write` y `exit_group`.

7. ¿A qué sentencias del programa fuente corresponden estas tres llamadas?
   **Respuesta:**

8. Mire en el manual (`man 2 write`, etc.) la descripción de esas llamadas. ¿Qué correspondencias hay entre los argumentos que hemos puesto en las sentencias y los argumentos que aparecen en las llamadas?
   **Respuesta:**

Ejecute ahora «`strace ./Lab4 fichero`». Verá que el programa se detiene haciendo la llamada `read`, que ha puesto el descriptor de la entrada estándar (1) y está esperando una cadena.

9. Escriba una palabra cualquiera terminada con retorno. ¿Cómo completa el sistema la `read`?
   **Respuesta:**

10. ¿Cuál es el descriptor del fichero?
    **Respuesta:**

Ejecute de nuevo «`./Lab4 fichero`» y cuando le pida que escriba algo, teclee varias palabras, que en conjunto tengan más de 10 caracteres.

10. ¿Qué ha ocurrido al introducir una cadena de más de 10 caracteres? ¿Por qué?
    **Respuesta:**

11. Analice qué pasaría si después de los 10 caracteres se escribe «`rm -rf /`» (en el caso de que lo ejecute un usuario normal y en el caso de que lo haga el superusuario).
    **ATENCIÓN: ¡NO SE LE OCURRA PROBARLO!**
    **Respuesta:**

No es necesario que lo haga ahora, pero investigue en Internet lo que es un *«buffer overflow»*.

# Lab5: Codificación de textos. Tipos de ficheros. Metadatos

## Objetivos

- Analizar algunas diferencias en los códigos más comunes para representación de textos.
- Aprender a examinar el contenido binario de un fichero y deducir su tipo a partir de sus metadatos internos
- Conocer el formato y practicar el uso de herramientas que permiten cambiar metadatos y manipular ficheros de audio MP3.

## Inicio

Abra un terminal de texto desde el entorno gráfico y cree un nuevo subdirectorio de su directorio personal con nombre `Lab5`. Todas las actividades se realizarán dentro de `~/Lab5`.

## Primera parte: codificación de textos

Cree un fichero con el nombre `apell` que contenga los caracteres correspondientes a su primer apellido (no ponga tildes y, si tiene alguna ñ, cámbiela por n) seguidos de un espacio y el símbolo €. Lo puede hacer con un editor de texto, o con el comando `echo`:

```
$ echo <su-apellido> € > apell
```

Tenga en cuenta para los siguientes apartados que los editores de texto y el comando `echo` añaden el carácter 0x0A (salto de línea) al final del fichero.

Una vez creado `apell`, y como comprobación, visualice su contenido en la pantalla.

1. Utilice la orden `wc -c` para contar el número de bytes del fichero `apell`.
   **Numero de bytes:**

2. A la vista del número de bytes y teniendo en cuenta el contenido del fichero, ¿puede deducir si está codificado en ISO Latin9 o en UTF-8?
   **Respuesta:**

3. Compruebe la codificación del fichero `apell` con la utilidad `file`.
   **Resultado de `file apell`:**

El programa `iconv` convierte un fichero de texto de una codificación a otra. Así:

```
$ iconv -f <codificación-de-fich> -t <codificación-result> fich
```

convierte el contenido del fichero `fich`, codificado como se indica a continuación de `-f` (from) a la codificación que se indica a continuación de `-t` (to) y da el resultado por la salida estándar.

Se pueden ver todas las codificaciones posibles con la orden `iconv -l`. Dos de ellas son LATIN9 (ISO 8859-15) y UTF-8[9].

4. Si su fichero `apell` está en UTF-8, conviértalo a LATIN9 y guárdelo en otro fichero de nombre `apell-cod`:

[9] Puede comprobar cuál es la codificación por defecto en su sistema con `echo $LANG`.

```
$ iconv -f UTF-8 -t LATIN9 apell > apell-cod
```

Si está en LATIN9 invierta los valores de los parámetros `-f` y `-t` para convertirlo a un fichero codificado en UTF-8.

Compruebe, con `file`, que se ha cambiado la codificación del fichero `apell-cod`.
**Resultado de `file apell-cod`:**

5. Mire con `wc -c` el número de bytes del nuevo fichero `apell-cod`

   **Explique las diferencias que se aprecian en ambos ficheros**

El programa `hd` («hexadecimal dump») permite ver los contenidos binarios (representados en hexadecimal) de un fichero, mostrando en la salida estándar tres columnas: en la primera columna se indican las direcciones relativas de los bytes (empezando por 0x0), en la segunda columna se muestra la expresión en hexadecimal de cada byte (16 bytes en cada línea) y en la tercera su posible interpretación como ASCII (si corresponde a un carácter de control, muestra un punto).

6. Compruebe con `hd` los contenidos de los dos ficheros (`apell` y `apell-cod`), observando la correspondencia entre caracteres y codificaciones en ambos ficheros y analizando sus diferencias.

   **Salida de `hd apell`:**

   **Salida de `hd apell-cod`:**

   **Explique el por qué de las diferencias.**

Los ficheros que se han manipulado en los anteriores apartados son ficheros de texto «plano». Pero cuando se edita un fichero con un procesador de texto se generan «ficheros enriquecidos» que, además de los caracteres, incluyen propiedades (color, tamaño, tipo de letra). Dos extensiones estándares son bien conocidas: .odt (usada en OpenOffice) y .docx (en Microsoft Word) que realmente corresponden a archivos, es decir ficheros que contienen un conjunto de directorios y ficheros de texto plano (en XML) comprimidos con ZIP (apartado 3.6).

7. Busque en su ordenador algún fichero que tenga extensión .odt. La orden

   ```
   $ locate "*.odt"
   ```

   le devolverá todos los que encuentre (en caso de no encontrar ninguno, búsquelo en Internet). Copie uno de ellos en su directorio de trabajo poniéndole como nombre `docu.odt` y observe los ficheros y directorios contenidos en el archivo:

   ```
   $ unzip -l docu.odt
   ```

   **Resultado:**

## Segunda parte: identificación del tipo de contenidos: metadatos

Como hemos estudiado (apartado 7.1), los metadatos internos de un fichero tienen información sobre su contenido, y la utilidad `file` puede extraerla.

1. Copie en su directorio de trabajo el fichero `Lab4` que generó en la sesión de laboratorio anterior compilando `Lab4.c`. Mire con `file` qué tipo de contenido tiene el fichero.
   **Resultado de `file Lab4`:**

Compruebe que `Lab4` tiene permiso de ejecución para el usuario (si no lo tuviese póngaselo) y ejecútelo (`./Lab4 fichero`), para ver que se ejecuta correctamente.

Cambie los permisos de `Lab4` para que no pueda ser ejecutado ni por el propietario, ni por el grupo ni por el resto de usuarios, manteniendo los permisos de lectura.

2. Compruebe con `file` si `Lab4` sigue siendo ejecutable e intente ejecutarlo de nuevo.
   **Resultado de `file Lab4`:**
   **Resultado de `./Lab4 fichero`:**

Copie en su directorio de trabajo un fichero que contenga una imagen codificada con el formato GIF (puede buscarlo, como antes, con `locate`), poniéndole como nombre `imag.gif`.

3. Mire el tipo de su contenido:

   **Resultado de `file imag.gif`:**

4. Haga una copia de `imag.gif` que se llame `imag.exe` y mire también el tipo:
   **Resultado de file `imag.exe`:**

5. Compruebe con `file` si el fichero `imag.exe` es de tipo ejecutable.
   **¿Es ejecutable?**
   **¿Qué relación hay entre el tipo de un fichero y su extensión?**

6. Cambie los permisos de fichero `imag.exe` con `chmod`, de forma que pueda ser ejecutado por el propietario, el grupo y el resto de usuarios, manteniendo los permisos de lectura e intente ejecutarlo.
   **Resultado de `./imag.exe`:**
   **¿Qué relación hay entre el tipo de un fichero y sus permisos?**

La cabecera de un fichero de tipo GIF empieza con tres bytes (la «firma») que representan las codificaciones ASCII de los caracteres «GIF», y sigue con otros tres bytes (la «versión»), que pueden ser la codificación de los caracteres «87a» o la de los caracteres «89a» (apartado 7.1, página 114)[10].

Con la orden `hd` puede comprobar la información de cabecera del fichero `imag.gif` pero, como el listado resulta ser muy largo, para ver sólo las primeras líneas escriba:

```
$ hd imag.gif | less
```

o bien:

```
$ hd imag.gif | head
```

[10] La especificación completa está en `http://www.w3.org/Graphics/GIF/spec-gif89a.txt`.

Compruebe que los doce primeros dígitos hexadecimales de la primera línea corresponden a lo que hemos dicho, y fíjese en los ocho dígitos siguientes.

7. De esos ocho dígitos, interprete los cuatro primeros como un número entero representado en formato de coma fija de 16 bits y almacenado con convenio extremista menor.
   **¿Cuál es el número representado?**

8. Haga lo mismo con los cuatro dígitos siguientes.
   **¿Cuál es el número representado?**
   **¿Qué relación guardan los números obtenidos con el resultado que muestra la orden** `file imag.gif`**?**

9. Genere un fichero de texto plano de nombre `raro.gif` que contenga solamente «GIF89a2014» (sin las comillas, y respetando mayúsculas y minúsculas) y compruebe su tipo:

   ```
   $ file raro.gif
   ```

   ¿Por qué el resultado indica que el fichero contiene una imagen de tamaño 12338 x 13361 cuando en realidad no contiene ninguna imagen?
   **Explique de dónde sale ese tamaño de imagen**.

Descargue de la página `http://www.fileformat.info/format/tiff/sample/` los ficheros `MARBLES.TIF` y `MARBIBM.TIF`.

Para las preguntas que siguen tenga en cuenta la explicación sobre números mágicos del apartado 7.1 (página 114).

10. Examine los ocho primeros bytes de ambos ficheros:

    ```
    $ hd MARBLES.TIF | head
    $ hd MARBIBM.TIF | head
    ```

    **Justifique el contenido de esos ocho bytes en cada uno de los ficheros**.
    **¿Por qué el contenido de los bytes quinto al octavo en uno de los ficheros es 0x2A00 y en el otro es 0x002A?**

11. Observe el resultado de `file` sobre cada uno.
    **¿Cómo obtiene** `file` **los dos primeros metadatos que muestra?**

12. Mire el número mágico de un fichero ejecutable en Unix. Por ejemplo el fichero correspondiente a la orden `cat`:

    ```
    $ hd /bin/cat | head
    ```

    y vea el resultado de `file /bin/cat`
    **¿Cómo obtiene** `file` **el primero de los metadatos?**

Compruebe que el fichero `Lab4`, al ser un ejecutable, tiene el mismo número mágico.

## Tercera parte: manipulación de contenidos de ficheros MP3

El objetivo de esta parte de la práctica es aprender el uso de dos herramientas muy útiles para manipular los contenidos de ficheros de audio:

- `avconv`[11], conversor de audio y vídeo con muchas facilidades: conversión de formatos, cambio de frecuencias de muestreo, escalado de vídeo en tiempo real, captura de una fuente de audio o vídeo, etc[12].
- `id3`, herramienta de edición de las etiquetas informativas (metadatos) que se incluyen en los ficheros MP3 para facilitar su catalogación, siguiendo el estándar ID3[13].

Descargue un fichero MP3 disponible en Internet bajo licencia «Creative Commons» desde la página `http://creativecommons.org/wired/`. Por comodidad, cambie el nombre original por otro más corto que no contenga espacios en blanco, manteniendo la extensión mp3 y utilizando una orden similar a ésta (las comillas son necesarias por los espacios en blanco):

```
$ mv "Beastie Boys - Now Get Busy.mp3" prueba.mp3
```

1. Una de las funciones de `avconv`, como su nombre sugiere, es la conversión de formatos de audio y vídeo. Convierta `prueba.mp3` al formato WAV:

   ```
   $ avconv -i prueba.mp3 prueba.wav
   ```

   (si `prueba.wav` no existe, lo crea; si existe pide confirmación antes de sobrescribirlo). Compruébelo[14], y compare los tamaños:

   ```
   $ file prueba.*
   $ ls -l prueba.*
   ```

   **Explique la diferencia de tamaños**.

2. Pero `avconv` tiene muchas opciones para ajustar diversos parámetros en el proceso de conversión (consulte `man avconv`). Por citar sólo dos:

   ```
   $ avconv -i prueba.mp3 -ar 16000 -ac 1 prueba-pobre.wav
   ```

   «`-ar 16000`» pone una frecuencia de muestreo de 16 kHz (en el original de «Beastie Boys» es 44,1 kHz) y hace el sonido monoaural (en el original es estéreo). Compruébelo con `file` y mire también el tamaño de `prueba-pobre.wav`.

   **Explique la diferencia de tamaños**.

[11] `avconv` forma parte del proyecto `libav`, que es un «fork» de `ffmpeg`. Las distribuciones Debian y su derivada Ubuntu utilizan ahora `avconv`, pero otras distribuciones siguen con `ffmpeg`. Todo lo que sigue puede hacerse exactamente igual sustituyendo «`avconv`» por «`ffmpeg`».

[12] Más información en `http://www.libav.org/`.

[13] Más información en `http://id3.org/`.

[14] Obviamente, también puede comprobar con un reproductor de audio que la salida sonora es la misma.

Otra de las funcionalidades de `avconv` es la de extraer fragmentos de ficheros de audio y vídeo:

```
$ avconv -i <entrada> -ss <inicio> -t <duración> -acodec copy <salida>
```

en donde:
`-i <entrada>` indica el fichero audio origen.
`-ss <inicio>` indica el instante en segundos en el que se quiere empezar a extraer.
`-t <duración>` es la duración en segundos del fragmento a extraer.
`-acodec copy` indica que el audio o el vídeo debe copiarse sin ninguna transformación.
`<salida>` es el nombre dado al fichero resultante de la extracción.

Ejecute `avconv` poniendo los valores adecuados de las opciones para que «corte» el fichero `prueba.mp3` a partir de algún momento distinto del de inicio y obtenga un fragmento de duración 10 segundos y deje el resultado en un nuevo fichero, `prueba-corta.mp3`.

3. En este momento tiene usted cuatro ficheros de audio: `prueba.mp3`, `prueba.wav`, `prueba-pobre.wav` y `prueba-corta.mp3`. Compruebe el tamaño de los cuatro y ejecute

   ```
   $ avconv -i <fich>
   ```

   sobre los cuatro. Verá que, aunque avisa al final de que hay que dar un fichero de salida, previamente muestra metadatos, entre ellos la duración, la tasa de bits, la frecuencia de muestreo y el número de canales.

| fichero | tamaño | duración | bitrate | frec. muestreo | canales |
|---|---|---|---|---|---|
| `prueba.mp3` | | | | | |
| `prueba.wav` | | | | | |
| `prueba-pobre.wav` | | | | | |
| `prueba-corta.mp3` | | | | | |

Habrá observado que entre los metadatos aparecen también etiquetas *(tags)* con sus valores. Vamos a practicar desde la línea de comandos[15] con la utilidad `id3` para edición de etiquetas de ficheros MP3. Esta utilidad sigue la versión 1 del estándar ID3 (ID3v1), que establece las siguientes etiquetas:

- Título (30 caracteres)
- Artista (30 caracteres)
- Álbum (30 caracteres)
- Año (4 caracteres)
- Comentario (30 caracteres) (28 en ID3v1.1)
- Pista (sólo en ID3v1.1, 2 bytes)
- Género (1 byte: 0x00 Blues, 0x01 Classic Rock... 0x4F Hard Rock, 0xFF Sin valor)[16]

[15] Hay muchos editores gráficos, por ejemplo, «EasyTAG», para manipular etiquetas ID3. No los vamos a utilizar en esta práctica, pero es interesante experimentar con alguno.

[16] `http://id3.org/id3v2.3.0#Appendix_A_-_Genre_List_from_ID3v1`

cuyos valores se guardan en un bloque de 128 bytes al final del fichero MP3[17]. Este bloque empieza con las codificaciones ASCII de «TAG» y le siguen los 125 bytes correspondientes a los valores de las etiquetas en el orden indicado.

La versión 2 de ID3 añade más etiquetas al comienzo del fichero MP3 siguiendo un convenio más estructurado y con muchas más posibilidades: puede incluir hasta imágenes y llegar a ocupar hasta 256 MiB. La utilidad correspondiente a esta versión 2 es `id3v2`.

Practicaremos sólo con id3v1. El programa `id3` tiene trece opciones para manipular las etiquetas, que puede ver consultando `man id3` (o, simplemente, tecleando «`id3`», que muestra un resumen de las trece). Por ejemplo `id3 -a "Pepe Perez" <fich>` cambia el nombre de la etiqueta «Artist» por «Pepe Perez». El valor de la etiqueta debe ir entre comillas siempre que haya espacios en blanco y no debe incluir caracteres no representables en ASCII (como vocales con tilde o ñ).

5. Ejecute `id3` con opciones `-l` o `-lR` para obtener los valores de las etiquetas ID3 del fichero `prueba.mp3`

   **Resultado de** `id3 -l prueba.mp3`**:**

   **Resultado de** `id3 -lR prueba.mp3`**:**

Con `hd` puede ver los contenidos del principio y del final de un fichero MP3 y averiguar qué versión de ID3 tienen las etiquetas. Con

```
$ hd <fich>| head
```

verá si el fichero comienza con «0xFFFB», en cuyo caso se tratará de ID3v1, o comienza con «ID3» y, en tal caso, se trata de ID3v2.

Con

```
$ hd <fich>| tail
```

verá que hacia el final aparece «TAG», luego el título, etc.

6. Borre las etiquetas que hayan podido quedar en el fichero cortado (`prueba-corta.mp3`) utilizando la herramienta `id3v2`, que elimina tanto las etiquetas ID3v1 como las ID3v2:

   ```
   $ id3v2 -D prueba-corta.mp3
   ```

   Añada nuevas etiquetas ID3 personalizadas al fichero que contiene el fragmento. En concreto, dé valores a las etiquetas: *Artist* (poniendo las iniciales de su nombre), *Year* (el año actual) y *Comment* («Modificado por» seguido de su nombre). Una vez hechas las modificaciones, compruebe con `id3` los valores resultantes:

   **Resultado de** `id3 -lR prueba-corta.mp3`**:**

   Compruebe también con `hd` que los últimos 128 bytes del fichero empiezan con «TAG» y le siguen las codificaciones de los valores de las etiquetas que ha añadido en las posiciones que les corresponden.

[17] El motivo de incluir estos metadatos al final y no al principio es que en el diseño de los primeros reproductores de MP3 no se había previsto su existencia, por lo que si estuviesen al principio generarían un pequeño ruido al comenzar la reproducción.

# Lab6: ARMSim#

## Objetivos

- Comprobar el funcionamiento del ensamblador en sus funciones de traducción de instrucciones y pseudoinstrucciones (`ldr`) y de traducción de directivas de carga de datos (`.asciz`).
- Comprobar los resultados de la ejecución de instrucciones de movimiento y procesamiento (`mov`, `add`, `cmp`).
- Comprobar el funcionamiento de instrucciones de bifurcación condicionada (`bne`).
- Comprobar el funcionamiento de instrucciones de acceso a la memoria (`ldrb`, `srtb`).

## Programa objeto de la práctica

```
        .text
        .equ      NUL, 0
        .global   _start
_start:   ldr   r0, =nombre
          ldr   r1, =erbmon+7
          mov   r3, #0
bucle:    ldrb  r2, [r0], #1
          strb  r2, [r1], #-1
          cmp   r2, #NUL
          add   r3, r3, #1
          bne   bucle
          swi   0x11
       .data
nombre: .asciz  "Nombre "  @ sustituya esto con su nombre en 7 bytes
erbmon: .skip 8
      .end
```

## Actividades previas

Lo mismo que en los laboratorios anteriores, antes de la sesión que corresponda a su grupo deberá usted leer este documento e intentar realizar las actividades que en él se describen.
Además, **es necesario que:**

1. Estudie, analice y comprenda el programa 10.5 (página 175).
2. Haga lo mismo con el programa objeto de la práctica.
3. Escriba un fichero de texto con el programa, como se indica en el punto 1 de la descripción, y deje grabado este fichero en su cuenta en un directorio `~/Lab6`, o bien en una memoria USB, para utilizarlo durante la sesión de laboratorio.
4. Haga los cálculos necesarios para averiguar la traducción a binario de la instrucción `bne bucle`. Anote estos cálculos y el resultado (en hexadecimal) como se indica en el punto 4 de la descripción. Para realizar esta actividad deberá hacer estudiado y comprendido el apartado «Instrucciones de bifurcación» del apartado 9.6.

## Descripción de la práctica

1. Utilizando un editor de texto plano (puede ser `nano`, o `gedit`, que es más cómodo y se encuentra en Aplicaciones > Accesorios > Editor de textos) escriba en un fichero el programa, sustituyendo la cadena «`Nombre `» por su nombre de pila entre comillas, y con exactamente 7 caracteres ASCII, es decir, sin tildes. (Si tiene más, sólo los 7 primeros; si tiene menos, complete con blancos, por ejemplo: «`Eva  `»). Sálvelo. El fichero puede tener cualquier nombre, pero su extensión debe ser «`.s`».

2. Ejecute el simulador ARMSim# (apéndice B)

3. Cargue (File > Load) el fichero en el simulador. Observe que, al decirle «Load» al simulador, éste automáticamente ensambla el programa antes de cargarlo. Si le da algún error al cargarlo es porque ha tecleado algo mal. Vuelva al editor de texto, corrija lo necesario, salve y vuelva a cargarlo (File > Reload, o icono «Reload» en la barra de herramientas)[18].

4. **Compruebe que la traducción de la instrucción «`bne bucle`» coincide con el resultado de estos cálculos:**

   Campo Cond (binario y hexadecimal): _____

   Bits 27-24 (binario y hexadecimal): _____

   Campo Dist:

   Cálculos:

   Resultado del campo Dist (hexadecimal): _____

   Resultado final de la instrucción(hexadecimal): _____

5. Antes de empezar la ejecución, visualice los contenidos de memoria:

   *a*) En el menú superior, «View > Memory»

   *b*) En la parte inferior, a la izquierda, en la casilla para indicar la dirección de comienzo, dirección de comienzo del programa (00001000) e «Intro»

   *c*) A la derecha, ponga, de momento, «Word Size > 32 Bit». Observará que los contenidos de palabras sucesivas se corresponden con los del listado que ha generado el ensamblador [19] .

   *d*) Cambie a «Word Size > 8 Bit». Observará que cada cuatro bytes aparecen los mismos contenidos, pero en orden inverso.
   **¿Por qué cree que aparecen en ese orden?**

6. Observe los contenidos a partir de la dirección 0000102C (poniendo ese valor en la casilla de la izquierda y con Word Size de 8 bits).
   **¿A qué corresponden esos contenidos?**

[18] El simulador pone por defecto un color gris para el fondo. Puede cambiarlo a blanco o a cualquier otro con el botón derecho del ratón.

[19] En las direcciones de memoria en las que no se ha cargado nada (por ejemplo, a partir de la siguiente a 0x103B, que es el último byte reservado por «`.skip 8`») el simulador pone un valor arbitrario, que suele ser 0x81. Este valor puede modificarse en «File > Preferences > Memory Fill Pattern». En principio es indiferente el valor que se ponga, pero influye si se quiere responder con detalle a la última pregunta.

| Dirección | Contenido (hex.) | Corresponde a |
|---|---|---|
| 0x102C | | |
| 0x102D | | |
| 0x102E | | |
| 0x102F | | |
| 0x1030 | | |
| 0x1031 | | |
| 0x1032 | | |
| 0x1033 | | |

7. En la vista de registros (a la izquierda) seleccione «Hexadecimal». Compruebe que todos los registros están inicialmente a cero, salvo R13 (SP), que tiene el valor 0x5400 (fondo de la pila) y R15 (PC), que tiene el valor 0x1000 (dirección de comienzo de la ejecución del programa cargado). Ejecute, una a una (icono «Step Into» en la barra de herramientas, arriba a la izquierda), las tres primeras instrucciones del programa[20]. Los contenidos de los registros que cambian en cada instrucción se van resaltando en rojo.
**Explique brevemente a qué corresponden estos nuevos contenidos.**

| Registros | Nuevo valor | Corresponde a |
|---|---|---|
| R0 | 0x102C | Dirección del primer byte de «nombre» |
| | | |
| | | |

8. Observe, en la traducción que ha hecho el ensamblador (en la ventana central), que la última instrucción está en la dirección 0x1020. Sin embargo, los datos no empiezan en 0x1024, sino en 0x102C. Mirando el mapa de la memoria (mejor con 32 bits) puede comprobar qué es lo que el ensamblador ha puesto en ese «hueco».
**¿Qué contenidos tienen las palabras de direcciones 0x1024 y 0x1028, a qué corresponden, y en qué instrucciones se utilizan?**

9. Ejecute la instrucción siguiente («`bucle`»).
**¿Qué registros han cambiado y porqué?**

| Registros | Nuevo valor | ¿Por qué? |
|---|---|---|
| R0 | 0x102D | Se le ha sumado una unidad |
| | | |
| | | |

10. Ejecute la instrucción siguiente a «`bucle`» (`strb`).
**¿Qué registros han cambiado y porqué?**

| Registros | Nuevo valor | ¿Por qué? |
|---|---|---|
| | | |
| | | |

11. **¿Ha cambiado algo en la memoria?**

[20] Si en algún momento se equivoca y en lugar de «Step Into» pulsa «Run», el programa se ejecutará sin parar hasta el final. En tal caso, vuelva a cargarlo (icono «Reload», a la derecha).

12. Continúe la ejecución del programa hasta su finalización (icono «Run»). Los contenidos de los registros y de la memoria que han cambiado aparecen resaltados en rojo.
**Explique el por qué de estos nuevos contenidos, tanto de los registros como de la memoria.**

**Cambios en los registros:**

| Registros | Nuevo valor | ¿Por qué? |
|---|---|---|
| R0 | 0x1034 | Ha avanzado hasta el byte siguiente al último de «nombre» (que es NULL, porque se ha definido con «.asciz») |
| | | |
| | | |
| | | |
| | | |

**Cambios en la memoria:**

Observe los mensajes que aparecen en la ventana inferior al seleccionar «OutputView – Console».

13. Vuelva al editor de texto y modifique el programa quitando «`#1`» y «`#-1`» de las instrucciones de carga y almacenamiento (o sea, quedarán «`ldrb r2, [r0]`» y «`strb r2, [r1]`». Vuelva a cargar el programa, ejecútelo (icono «Run») y observe los mensajes de la ventana inferior («OutputView – Console»).
**¿Qué está ocurriendo y por qué?**

14. Pulse sobre el icono «Stop». Los contenidos de los registros que han cambiado aparecen resaltados en rojo.
**¿Qué valores tienen?**

| Registros | Nuevo valor |
|---|---|
| | |
| | |
| | |
| | |
| | |

15. **Explique el por qué de estos nuevos contenidos.**

16. Restituya con el editor de texto las instrucciones `ldrb r2, [r0], #1` y `strb r2, [r1], #-1`, pero haga otra modificación: cambie `add r3, r3, #1` por `adds r3, r3, #1`. Cargue el nuevo programa y ejecútelo (icono «Run»), observando el mensaje que muestra «OutputView – Console».
**¿Qué cree que ha pasado, y por qué?**

17. Para comprobar lo que realmente está pasando, vuelva a cargar el programa. Ponga la vista de la memoria con 32 bits («Memory View», «Word Size» > 32 Bit) a partir de la dirección 00001000, de manera que verá, a partir de la dirección 0x1000 las instrucciones y después de ellas los datos. Ejecute paso a paso el programa y observe detenidamente cómo, a cada paso por el bucle, van cambiando esos contenidos. Deberá dar 29 vueltas por el bucle (el número de vuelta se ve en R3) antes de que se detenga el programa por el error.
**Explique lo más detalladamente posible el motivo del error.**

# Lab7: Llamadas en bajo nivel. Traducción y ejecución

### Objetivos

- Comprender la implementación en bajo nivel (ABI: Application Binary Interface) de las llamadas al sistema
- Comparar el uso de recursos (memoria y tiempo de CPU) de los programas compilados y ensamblados

### Programas

Se utilizarán cuatro programas cuyos listados se incluyen al final:

- `Lab7-1C.c`: programa en C casi idéntico al utilizado en el Lab4 (la llamada CREAT se ha sustituido por una OPEN equivalente porque el compilador cruzado que utilizaremos no entiende la CREAT)
- `Lab7-1S.s`: programa en ensamblador con la misma función que `Lab7-1C.c`
- `Lab7-2C.c`: programa en C que hace 300 millones de multiplicaciones
- `Lab7-2S.s`: programa en ensamblador con la misma función que `Lab7-2C.c`

### Herramientas

Además del compilador nativo de C para la arquitectura x86, en los ordenadores del laboratorio están instalados:

- Un ensamblador, un montador y un compilador cruzados para generar ejecutables para la arquitectura ARM (la cadena GNU «Sourcery CodeBench», ver apuntes, apartado B.3).
- Un emulador (`qemu`) junto con un módulo del kernel (`binfmt_misc`) que permiten ejecutar programas ejecutables para ARM en la arquitectura x86.

### Actividades previas

Lo mismo que en los anteriores, antes de la sesión del laboratorio deberá usted leer este documento e intentar realizar las actividades que en él se describen.
Además, **es necesario que:**

1. Escriba cuatro ficheros de texto con los cuatro programas y deje grabados estos ficheros en su cuenta, en un directorio `~/Lab7`, o bien en una memoria USB, para utilizarlo durante la sesión de laboratorio.
2. Analice el programa `Lab7-1S.s` tratando de comprender la correspondencia entre grupos de instrucciones y las sentencias del programa `Lab7-1C.c`.
3. Analice el programa `Lab7-2S.s` tratando de comprender la correspondencia entre grupos de instrucciones y las sentencias del programa `Lab7-2C.c`.

Por ejemplo, trate de responder a las preguntas sobre la relación entre los programas `Lab7-1C.c` y `Lab7-1S.s` que ya están señaladas como comentarios en el listado de `Lab7-1S.s`.

## Descripción de la práctica

La práctica tiene tres partes. Las dos primeras son las actividades a realizar en el laboratorio, entregando al final las respuestas a las preguntas planteadas. La tercera es opcional, y consiste en reproducir los experimentos sobre una Raspberry Pi o máquina equivalente (es decir, basada en ARM y con compilador, ensamblador y montador).

### Primera parte

Vamos a utilizar tanto el compilador para x86 (`gcc`) como el compilador cruzado que genera código para ARM. En los ordenadores del laboratorio este compilador está instalado en `/opt/arm/bin/arm-none-linux-gnueabi-gcc`[21]. También utilizaremos el ensamblador y el montador cruzados, que se encuentran en ese mismo directorio. (Las órdenes `gcc`, `as` y `ld`, sin especificar la ruta, remiten a `/usr/bin/gcc`, etc., que son los procesadores nativos para x86).

Para evitar tener que teclear toda la ruta puede definir estas variables:

```
export CROSS=/opt/arm/bin/arm-none-linux-gnueabi-
export AS=${CROSS}as
export LD=${CROSS}ld
export GCC=${CROSS}gcc
```

Si escribe estas líneas en un terminal, en lo sucesivo podrá escribir «`$AS`», «`$LD`» o «`$GCC`» en lugar de `/opt/arm/bin/arm-none-linux-gnueabi-as`, etc.[22]

1. Compile el programa `Lab7-1C.c` para las dos arquitecturas, guardando los resultados con nombres distintos:

   ```
   gcc -o Lab7-1Cx86 Lab7-1C.c
   $GCC -static -o Lab7-1Carm Lab7-1C.c
   ```

   (La opción «`-static`» le indica al montador incluido en el programa `$GCC` que incluya en el código ejecutable las bibliotecas necesarias).

   Compare (con la utilidad `file`) las diferencias principales entre los tipos de `Lab7-1Cx86` y `Lab7-1Carm` (arquitectura x86 o ARM y si están «statically» o «dynamically linked»), y sus tamaños (con `ls -l`).

| | **Diferencias en el tipo** | **Tamaño (bytes)** |
|---|---|---|
| **Lab7-1Cx86** | | |
| **Lab7-1Carm** | | |

[21] Si desea instalarlos en su equipo personal, lea el apartado B.3.

[22] Si quiere que esta definición de variables sea permanente para cada vez que abra un terminal puede editar el fichero `.bash_profile` e incluirlas en él. Pero tenga cuidado de no borrar ni modificar lo que ya tiene el fichero.

2. La disparidad en los tamaños de los ejecutables se explica porque `gcc` ha generado un programa objeto que carga bibliotecas del sistema en tiempo de ejecución, mientras que el generado por `$GCC` ya incluye todo lo necesario (observe las diferencias obtenidas con la utilidad `file` y con `ls -l`). Para obligar a `gcc` que genere un ejecutable «autónomo» escriba:

   ```
   gcc -static -o Lab7-1Cx86static Lab7-1C.c
   ```

| | **Diferencias en el tipo** | **Tamaño (bytes)** |
|---|---|---|
| **Lab7-1Cx86static** | | |

3. Ejecute ambos programas (sin y con nombre de fichero):

   ```
   ./Lab7-1Cx86
   ./Lab7-1Carm
   ```

   y

   ```
   ./Lab7-1Cx86 fich
   ./Lab7-1Carm fich
   ```

   (En el caso de «x86» el programa se ejecuta directamente en la CPU del ordenador, mientras que en el de «arm» intervienen el módulo del kernel y el emulador qemu).

   **¿Observa alguna diferencia en el comportamiento?**

4. Ahora vamos a trabajar con el programa fuente en ensamblador, `Lab7-1S.s`. Primero, pruebe esto:

   ```
   as -o Lab7-1Sarm.o Lab7-1S.s
   ```

   **Explique el motivo de los errores**

5. Ensamble `Lab7-1S.s`:

   ```
   $AS -o Lab7-1Sarm.o Lab7-1S.s
   ```

   El ensamblador genera un código objeto que tiene que pasarse por el montador para obtener un ejecutable (en los experimentos anteriores, `gcc` hace automáticamente el montaje, a menos que se indique lo contrario con la opción `-c`). Por tanto, monte `Lab7-1S.o`:

   ```
   $LD -o Lab7-1Sarm Lab7-1Sarm.o
   ```

   Ahora tenemos un ejecutable para ARM.

   **Compruebe las diferencias (tipo y tamaño) con los ficheros anteriores.**

6. Lo mismo que con el ejecutable para ARM que obteníamos con `$GCC`, podemos simular en el PC la ejecución de `Lab7-1Sarm` gracias a qemu y el módulo del kernel:

   ```
   ./Lab7-1Sarm
   ```

   **Observa alguna diferencia en el comportamiento?**

## Segunda parte

Hemos podido comprobar la diferencia en *tamaño* de los programas ejecutables obtenidos mediante compilación y mediante ensamblaje. Pero los programas fuente anteriores no son adecuados para comprobar la diferencia en *tiempos de ejecución*. Utilicemos ahora `Lab7-2S.s` y `Lab7-2C.c`, que no son interactivos y sí implican mucho uso de CPU.

Compile `Lab7-2C.c` para x86, para x86 sin carga dinámica de bibliotecas (`-static`) y para ARM:

```
gcc -o Lab7-2Cx86 Lab7-2C.c
gcc -static -o Lab7-2Cx86static Lab7-2C.c
$GCC -static -o Lab7-2Carm Lab7-2C.c
```

Ensamble `Lab7-2S.s`:

```
$AS -o Lab7-2Sarm.o Lab7-2S.s
$LD -o Lab7-2Sarm Lab7-2Sarm.o
```

El tiempo de ejecución de un programa se puede observar anteponiendo `time` a su nombre. Por ejemplo: `time ./Lab7-2Cx86`. Fíjese en la línea «user», que es la que indica el tiempo que la CPU ha estado ejecutando en modo usuario.

**Compruebe los tamaños y los tiempos de ejecución de los cuatro ejecutables generados.**

| | **Tamaño (bytes)** | **Tiempo (seg.)** |
|---|---|---|
| **Lab7-2Cx86** | | |
| **Lab7-2Cx86static** | | |
| **Lab7-2Carm** | | |
| **Lab7-2Sarm** | | |

**Escriba las conclusiones que cree que pueden extraerse de lo experimentado en esta práctica.**

## Tercera parte (opcional)

Reproduzca las actividades (las referentes a ARM, no las de x86) en una máquina basada en ARM, utilizando los procesadores `gcc`, `as` y `ld` nativos. También puede observar la ejecución paso a paso y las llamadas al sistema con el programa depurador `gdb`. Una breve introducción a su uso:
`http://www.akira.ruc.dk/~keld/teaching/CAN_e13/Readings/gdb.pdf`
Y una documentación completa: `http://www.gnu.org/software/gdb/documentation/`

Si realiza esta parte puede entregar por correo un informe sobre los resultados obtenidos: una tabla comparativa de los tamaños de los ficheros ejecutables y los tiempos de ejecución, y unas conclusiones.

## Listados

### Programa Lab7-1C.c

```
/* Directivas para incluir declaraciones de funciones de biblioteca
   necesarias para las llamadas al sistema: */
#include <sys/types.h> /* para open y creat */
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>     /* para printf */
#include <unistd.h>    /* para read y write */
#include <stdlib.h>    /* para exit */

void main(int argc, char *argv[]) {
  char palabra[10];
  int  descr, n1, n2;
  if (argc !=2) {
    printf("\nUso:  %s fichero\n\n", argv[0]);
    exit(1);
  }
  /* Si ha pasado la comprobación anterior... */
  if ((descr = open(argv[1], O_WRONLY)) == -1) /* Intenta abrir el fichero */
    descr = open(argv[1], O_WRONLY|O_CREAT, 0666);
      /* Si no existe, lo crea con permisos 666 */
  else   lseek(descr, 0, SEEK_END);    /* si existe, se va al final */
  printf("\nEscribe algo\n\n");
  n1 = read(1, palabra, 10);       /* lee la línea de stdin */
  n2 = write(descr, palabra, n1);  /* y la escribe en el fichero */
  exit(0);
}
```

### Programa Lab7-1S.s

```
@ Definimos símbolos para los números de las llamadas
@  y sus parámetros
.equ EXIT,1
.equ FORK,2
.equ READ,3
.equ WRITE,4
.equ OPEN,5
  .equ RDONLY,0  @ Son parámetros que se le pueden pasar a OPEN
  .equ WRONLY,1  @ (ver man 3 open)
  .equ RDWR,2
.equ CREAT,8
.equ CLOSE,6
.equ LSEEK,19     @ Parámetros que se le pueden pasar a LSEEK:
  .equ SEEK_SET,0 @  ir al principio del fichero
```

```
  .equ SEEK_CUR,1 @  ir a una posición
  .equ SEEK_END,2 @  ir al final
.equ STDOUT,1     @ Descriptores de la salida y
.equ STDIN,0      @ la entrada estándar

.text
.global _start
_start:
   ldr r0,[sp]   @ El número de argumentos al entrar (argc)  está
                 @ en cabeza de la pila, y los argumentos debajo
   cmp r0,#2
   beq sigue

/*** a) ¿A qué sentencia o sentencias del programa en C corresponden
las 16 instrucciones siguientes? */
   mov r0, #STDOUT
   ldr r1, =Uso1
   mov r2, #6      @ Número de caracteres de "Uso1"
   mov r7, #WRITE
   svc 0x0         @ Llamada1.
   mov r0, #STDOUT @ Hay que volver a hacerlo, porque el kernel modifica R0
   ldr r1, [sp,#4] @ El puntero al nombre del programa es el primer argumento
   mov r2, #6      @ Habría que contar los caracteres, pero es tedioso
   svc 0x0         @ Llamada2
   mov r0, #STDOUT
   ldr r1, =Uso2
   mov r2, #12
   svc 0x0         @ Llamada3. (PREG:En el programa en C printf genera las tres)
   mov r0, #-1      @ R0 != 0: retorno con error
   mov r7, #EXIT
   svc 0x0

/*** b) ¿A qué sentencia o sentencias del programa en C corresponden
las 6 instrucciones siguientes? ***/
 /* Si ha pasado la comprobación anterior... */
sigue:
   ldr  r0,[sp,#8] @ puntero al nombre del fichero
   mov  r1,#WRONLY
/* Intenta abrir el fichero */
   mov  r7,#OPEN
   svc  0x0
   cmp  r0,#0
   bpl  sigue1  @ Ya existe

/*** c) ¿A qué sentencia o sentencias del programa en C corresponden
las 4 instrucciones siguientes? ***/
```

```
   ldr  r0,[sp,#8] @ puntero al nombre del fichero
   ldr r1,=0666
   mov r7,#CREAT
   svc 0x0

/*** d) ¿A qué sentencia o sentencias del programa en C corresponden
las 6 instrucciones siguientes? ***/
sigue1:
/* Se va al final */
   ldr  r1, =descr
   strb r0, [r1]    @ Guarda en "descr" el descriptor del fichero abierto
   mov  r1, #0
   mov  r2, #SEEK_END
   mov  r7, #LSEEK
   svc  0x0

/*** e) ¿A qué sentencia o sentencias del programa en C corresponden
las 5 instrucciones siguientes? ***/
   mov  r0, #STDOUT
   ldr r1, =msg
   mov r2, #16
   mov r7, #WRITE
   svc 0x0

/*** f) ¿A qué sentencia o sentencias del programa en C corresponden
las 5 instrucciones siguientes? ***/
   mov r0, #STDIN
   ldr r1, =palabra
   mov r2, #10
   mov r7, #READ
   svc 0x0

/*** g) ¿A qué sentencia o sentencias del programa en C corresponden
las 9 instrucciones siguientes? ***/
   mov r2, r0  @ número de bytes leídos
   ldr r0,=descr
   ldrb r0, [r0]
   ldr r1, =palabra
   mov r7, #WRITE
   svc 0x0
   mov r0, #0      @ R0 = 0: código de retorno normal
   mov r7, #EXIT   @ R7 = 1: número de "exit"
   svc 0x0

.data
Uso1:    .ascii "\nUso: "       @ 6 bytes
```

```
Uso2:    .asciz " <fichero>\n"  @ 12 bytes
msg:     .asciz "\nEscribe algo\n\n" @ 16 bytes
descr:   .byte 1
palabra: .skip 11
.end
```

**Programa** `Lab7-2C.c`

```
int main ()
{
  int i, j, k, l;
  for (i = 0; i < 10000; i++)
   for (j = 0; j < 1000; j++)
    for (k = 0; k < 10; k++)
      l = 10*i - 100*j + 1000*k;
}
```

**Programa** `Lab7-2S.s`

```
.text
.global _start
_start:
  ldr r6, =10000    @ límite de i
  ldr r7, =1000     @ límite de j
  mov r8, #10       @ límite de k
  mov r0, #0  @ R0 = l
  mov r1, #0  @ R1 = i
buclei:
  mov r2, #0  @ R2 = j
buclej:
  mov r3, #0  @ R3 = k
buclek:
  mov   r4, r1
  add r4, r4, r4, lsl#2  @ 5*i
  add r4, r4, r4         @ 10*i
  add r0, r0, r4         @ (R0)+10*i-> R0
  mov   r4, r2
  add r4, r4, r4, lsl#2  @ 5*j
  add   r4, r4, r4, lsl#2  @ 5*j + 20*j = 25*j
  mov   r4, r4, lsl#2      @ 100*j
  sub r0, r0, r4         @ (R0) - 100*j -> R0
  mov r4, r3
  add r4, r4, r4, lsl#2  @ 5*k
  add r4, r4, r4, lsl#2         @ 25*k -> R4
  add r4, r4, r4, lsl#2  @ 125*k
  add r0, r0, r4, lsl#3  @ (R0) + 1000*k -> R0
  add r3, r3, #1
```

```
  cmp r3, r8
  blt buclek
  add r2, r2, #1
  cmp r2, r7
  blt buclej
  add r1, r1, #1
  cmp r1, r6
  blt buclei
  mov r7,#1
  svc 0x0
.end
```

# Apéndice D

# Ejercicios

---

## Sobre representación

1. Escribimos en un editor de texto la frase «En España se vive bien.» ¿Puede guardarse con codificación ASCII? ¿Y con codificación ISO-8859-1 (Latin-1)? ¿Y con codificación UTF-8? En cada uno de los casos, y si puede guardarse, ¿cuánto ocuparía?

2. Estime el número de «pinchos» de memoria *flash* de capacidad 4 GiB necesarios para almacenar una biblioteca de 2.000 libros que tienen por término medio 780 páginas, con texto codificado en ISO Latin-9 y compuesto a dos columnas de 40 líneas de 45 caracteres cada una.

3. Estime el número de libros como los del ejercicio anterior que pueden almacenarse en un CD-ROM (650 MiB) y en un DVD (4,7 GiB).

4. Decíamos al principio del capítulo 4 que actualmente la representación de datos numéricos en BCD sólo se utiliza para circuitos especializados. Hace años los procesadores dedicados a aplicaciones comerciales incluían instrucciones para operar con este tipo de representación, y para mantener la compatibilidad con programas antiguos *(legacy software)* algunos procesadores que han evolucionado hasta los modelos actuales las siguen teniendo. Así, en la arquitectura x86 de 32 bits los enteros en BCD empaquetado (es decir, con dos dígitos decimales empaquetados cada byte) se representan con diez bytes, con convenio de signo y magnitud. El bit más significativo es el bit de signo, y los otros siete bits de ese byte más significativo son todos ceros. El valor absoluto del número se representa en los nueve bytes restantes. El convenio de almacenamiento en memoria es extremista menor.

   *a*) Escriba los contenidos binarios de las direcciones de memoria *d* a *d+9* cuando en ellas está representado el número $-98.765 \times 10^{10}$ (decimal)

   *b*) ¿Cuáles son el mayor y el menor número representables?

5. En algunas aplicaciones la fecha se representa utilizando solamente dos dígitos decimales para el año, por lo que el año siguiente al «99» (1999) sería, según la representación interna en esos programas, el año «00» (1900). Este fue el «problema del año 2000». Se hicieron anticipadamente grandes inversiones para «parchear» los programas, y las perturbaciones fueron mínimas comparadas con algunas predicciones catastrofistas.

El «problema del año 2000» fue exclusivo de los sistemas derivados de MS-DOS. A Unix y sus derivados no les afectó, porque el tiempo se representa internamente en una palabra de 32 bits con signo (el signo es irrelevante en este caso, pero se sigue este convenio para que sea un tipo de datos más fácil de manejar en muchos lenguajes). El contenido de esta palabra se interpreta como el número de segundos transcurridos desde el 1 de enero de 1970. Podemos predecir que en estos sistemas aparecerá el «problema del año X». ¿Qué valor tiene X? (suponiendo que en ese año sigan en operación sistemas de 32 bits).

6. El pronosticado «problema del año 2000» no causó graves trastornos, pero el 1 de enero de 2010 sucedió algo imprevisto: en Alemania, unos 30 millones de tarjetas bancarias con chip dejaron de funcionar[1].

   Para entender la causa hay que saber que en estas tarjetas se utiliza un estándar llamado EMV (Europay-MasterCard-Visa) que define los convenios y los protocolos de comunicación entre las tarjetas y las máquinas procesadoras (por ejemplo, los cajeros, o los terminales de punto de venta). En este estándar la fecha se codifica en tres bytes, uno para el año, otro para el mes y otro para el día, *pero no se especifica un convenio para la representación*. ¿Cuál pudo ser la causa?

7. Un problema relacionado parece ser el que afectó a los usuarios de algunos modelos de la videoconsola PlayStation 3, que el 1 de marzo de 2010 no pudieron iniciar la sesión en el servicio *online*. Según el fabricante el problema se debió a un mal funcionamiento del reloj interno del equipo. Suponiendo que la causa fuese una falta de concordancia en los convenios entre el reloj hardware y el sistema operativo que lleva incorporado la consola, trate de responder a esta pregunta: ¿por qué precisamente el 1 de marzo?

8. Si los ordenadores en los que apareció el «problema NUXI» (apartado 3.5) hubiesen tenido una longitud de palabra de 32 bits ¿cómo se habría llamado el problema?

9. Con respecto al número entero $-6$ (menos seis):

   *a*) Escriba su representación binaria en un formato de coma fija de 32 bits con convenio de complemento a 2.

   *b*) Indique cómo se almacenaría en cuatro bytes consecutivos (de direcciones $d$, $d$+1, $d$+2 y $d$+3 de la memoria) en los casos de convenio «extremista mayor» y «extremista menor».

   *c*) Si los bytes del segundo caso los interpreta un programa como una cadena de caracteres codificados en ISO 8859-15 y los envía a una impresora, ¿cuál es el resultado?

10. Suponga que el contenido de un registro de 32 bits está formado por las codificaciones ASCII (en ocho bits, con 0 en el más significativo) de los caracteres «1234» («4» en el byte menos significativo, es decir, 0x31323334). Con dos operaciones sucesivas, una lógica y otra aritmética, se puede obtener la representación en coma fija del dígito menos significativo, «4». ¿Qué operaciones son ésas?

11. Indique una secuencia de operaciones (dos desplazamientos y una suma) para multiplicar por 10 un número entero $N$ representado en coma fija.

[1] `http://www.elpais.com/articulo/tecnologia/Masivo/bloqueo/tarjetas/bancarias/Alemania/causa/problema/chip/elpeputec/20100105elpeputec_10/Tes`

12. Apoyándose en los resultados de los ejercicios anteriores y en operaciones de rotación, esboce una secuencia de operaciones que permita obtener la representación en 16 bits de la cadena ASCII «1234» (o cualquier otra de cuatro dígitos) interpretada como número entero decimal. (Ésta es la esencia de la función «atoi», que mencionamos en la nota de la página 30).

13. Obtenga las representaciones en 32 bits del número decimal –21 (menos veintiuno), con los dos formatos siguientes:

    *a*) coma fija con convenio de complemento a 2;

    *b*) coma flotante con los mismos convenios del ejercicio anterior, pero con 32 bits, de los que 8 son para el exponente.

    Indique también, para cada uno de los casos, cómo se almacenarían los 32 bits en cuatro bytes consecutivos (de direcciones *d*, *d*+1, *d*+2 y *d*+3 de la memoria) en los casos de convenio «extremista mayor» y «extremista menor».

14. Averigüe qué número real, en decimal, se está representando en las posiciones *d* a *d+3* de la memoria de un ordenador que utiliza la norma IEEE 754 de precisión sencilla para representar los números reales y convenio de almacenamiento extremista menor, cuando el contenido de estas posiciones es:

```
Dir.  Contenido
[  d] 0x00
[d+1] 0x00
[d+2] 0x2D
[d+3] 0xC2
```

15. Obtenga la representación del número decimal –4.635,75 en el formato IEEE 754 con precisión sencilla.

16. Obtenga la representación del mismo número decimal, –4.635,75, en un formato de coma flotante de dieciséis bits con siete bits para el exponente, normalización fraccionaria y convenios de complemento a 2 para la mantisa y de exceso de $2^{e-1}$ para el exponente. Como hay que truncar la mantisa, compruebe cuál es el número realmente representado según que se trunque antes o después de hacer el complemento.

    **Nota:** Tenga en cuenta que en el formato IEEE 754 el bit más significativo de la mantisa se omite porque, al estar normalizada y con signo y magnitud, siempre es «1». Pero ahora el convenio es de complemento a 2, por lo que la fórmula es $N = \pm 0, M \times 2^E$, donde $E$ está con exceso de $2^{e-1}$ (no $2^{e-1}$-1).

## Sobre detección de errores y compresión

1. Para transmitir un mensaje se codifica cada uno de sus caracteres en ASCII con un bit de *paridad par* para la detección de errores. Una vez transmitido, el mensaje que llega a su destinatario, expresado en hexadecimal, es «`0x536FF9A0F96F`». ¿Qué es lo que dice el mensaje? ¿Se puede asegurar que no se ha producido ningún error en la transmisión?

2. Compruebe con algunos de los «bloques» (caracteres) del ejemplo del apartado 6.1 que el procedimiento del bit de paridad (con paridad par) resulta de utilizar como polinomio generador $G(x) = x + 1$ .

3. Determine el mensaje completo, en hexadecimal, que se enviará utilizando CRC si el polinomio generador es $x^4 + x + 1$ y el mensaje original es `0x4E6574` y se envía como un solo bloque.

4. A un extremo receptor le llega este mensaje: `0x21854A86896512CA89`. Se sabe que el polinomio generador es $G(x) = x^5 + x^2 + 1$, y que el emisor envía bloques de 24 bits, incluidos los de datos y los de redundancia.
   *a*) ¿Cuántos bloques se han recibido? En cada bloque, ¿cuántos bits son de datos, y cuántos de redundancia?
   *b*) Para cada uno de los bloques calcule si se ha producido error o no.

5. El estándar «Audio CD» especifica una codificación PCM con 16 bits por muestra en cada uno de los dos canales (estéreo) y 44,1 kHz.
   *a*) Calcule el ancho de banda (tasa de transferencia, en bps) de un reproductor de Audio CD.
   *b*) La más larga de las grabaciones de la Novena Sinfonía de Beethoven dura 74 minutos. Calcule la capacidad necesaria para su almacenamiento[2]
   *c*) La capacidad calculada es la que se especificaba en el estándar para Audio CD, de 1981. En 1984 se publicó el primer estándar de CD-ROM, para almacenamiento de datos (incluyendo algoritmos de detección y corrección de errores) en el mismo soporte físico utilizado por los Audio CD. En él se especifica una capacidad de almacenamiento de 650 MiB. Trate de encontrar una explicación de la discrepancia entre este dato y la capacidad calculada antes.

6. Digitalizamos una señal de audio de duración 10 seg. con una tasa de bits de 109,44 kbps.
   *a*) ¿Qué capacidad de memoria, en bytes, sería necesaria para guardarla?
   *b*) Le aplicamos un programa que genera un fichero MP3 y al ejecutar `ls -l` sobre él vemos que ocupa 15.200 bytes. ¿Cuál es el factor de compresión del algoritmo que implementa el programa?

7. Compra usted una tarjeta de memoria de 16 GiB para el móvil. La cámara fotográfica de su móvil captura imágenes con una resolución de 4.096 x 4.096 píxeles y una profundidad de color de 16 bits. Suponiendo que la tarjeta de memoria sólo se va a dedicar a almacenar fotos, ¿cuántas se podrán almacenar?

8. Siguiendo con el supuesto anterior, después de mucho uso comprueba usted que ha podido almacenar más de 12.000 fotos. ¿Con qué factor de compresión se están almacenando?

[2] Se dice que este requisito determinó, entre otras cosas, el tamaño (12 cm) de los CD: `http://www.snopes.com/music/media/cdlength.asp`

9. La imagen adjunta ha servido desde 1973 para ilustrar algoritmos de compresión[3].

El original, en formato TIFF, tiene una resolución de $512 \times 512$, por cada píxel hay tres muestras (RGB), cada muestra se codifica con 8 bits y no está comprimida. El fichero que la contiene ocupa exactamente 786.572 bytes, y si se convierte al formato BMP resulta un fichero de 786.486 bytes.

   *a*) ¿Cuál es la profundidad de color?

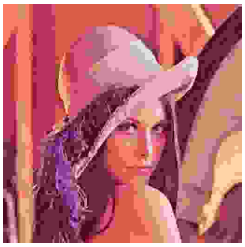
   *b*) Calcule el número de bytes necesarios para almacenar la imagen y explique las diferencias con los tamaños de los ficheros.

10. Las cuatro imágenes siguientes se han obtenido transformando la original mediante el programa `convert` de la herramienta ImageMagick a un formato JPEG con los factores de calidad (Q) que se indican en cada una junto con los tamaños de los ficheros.

Q = 100
398 KiB

Q = 50
23 KiB

Q = 10
7,4 KiB


Q = 1
2,9 KiB

Teniendo en cuenta los tamaños de los cuatro ficheros, calcule los factores de compresión aproximados que se han aplicado en cada uno de los cuatro casos.

11. Suponga que examina usted un fichero llamado `fich` con las órdenes `hd` y `ls` y obtiene estos resultados:

```
$ hd fich | head -1
00000000 47 49 46 38 39 61 80 02  e0 01 f3 00 00 ff ff ff  |GIF89as.........|
$ ls -lh fich
-rw-r--r-- 1 usuario grupo 20K abr 17  2015 fich
```

¿Qué puede deducir acerca del contenido de `fich`?

12. Imagine que trabaja usted para una empresa que comercializa productos multimedia. La empresa dispone en Andorra de 20.000 películas grabadas en DVD y desea trasladarlas urgentemente a Madrid. Son 100 paquetes de 2 Kg cuyo transporte por carretera en 8 horas, con todos los gastos incluidos, tiene un coste de 500 €. Se le pide un estudio sobre la viabilidad de hacer el transporte de manera telemática, utilizando una línea de fibra óptica que le garantiza un caudal de 100 Mbps con un coste de 40 €/mes. Cada película ocupa, aproximadamente, 5 GiB. Compare ambos medios de transmisión: coste, tiempo y ancho de banda[4]. (El carácter sospechoso de las actividades de su empresa y su postura ante él no forman parte de este ejercicio).

[3] En la entrada «Lenna» de la Wikipedia se dan detalles sobre su origen, polémicas sobre su uso y otras curiosidades.

[4] Este ejercicio es una versión actualizada del propuesto por A.S. Tanenbaum en su libro «Computer Networks» (4ª ed., 2002, p. 91), en el que se trataba de cintas magnéticas, y concluía con una moraleja: «nunca subestimes el ancho de banda de una furgoneta cargada de cintas y viajando a toda velocidad por una autopista».

## Sobre procesadores hardware

1. Para este ejercicio y los dos siguientes vamos a considerar un procesador imaginario de tipo von Neumann. Es decir, sólo tiene un registro visible, el acumulador. Este acumulador tiene dieciséis bits, y la memoria está organizada en palabras de dieciséis bits (no se puede acceder a bytes: cada dirección de memoria es la dirección de una palabra). Las instrucciones tienen también dieciséis bits con este formato: bits 0-12: CD; bits 13-15: CO.

   Las instrucciones son:

| **CO** | | **Significado** | **Ejemplo en ensamblador** (valores numéricos en decimal) |
|---|---|---|---|
| 000 | `LD` | (M[CD]) → AC | `LD 40`: carga el contenido de la dirección 40 en el acumulador |
| 001 | `ST` | (AC) → M[CD] | `ST 40`: almacena el acumulador en la palabra de dirección 40 |
| 010 | `MOV` | (CD) → AC | `MOV #-40`: carga *el número -40* en el acumulador |
| 011 | `ADD` | (AC) + (M[CD]) → AC | `ADD 40`: suma al acumulador el contenido de la dirección 40 |
| 100 | `ADDI` | (AC) + (CD) → AC | `ADDI #-40`: suma *el número -40* al contenido del acumulador |
| 101 | `BNE` | Si último resultado no ha sido cero, (PC) + (CD) → PC | `BNE -16`: bifurca a la dirección de esta instrucción + 1 – 16 si el último resultado no ha sido cero PC = contador de programa |
| 110 | | | |
| 111 | `HALT` | Detiene la ejecución | |

   *a*) ¿Qué modos de direccionamiento puede usted identificar en esas instrucciones? Para cada una, indique el modo que tiene.

   *b*) ¿Cuál es la capacidad máxima de memoria direccionable?

   *c*) ¿Qué rango de valores numéricos puede utilizarse con el direccionamiento inmediato?

   *d*) A la vista del ejemplo dado para `BNE` ¿se puede decir si el procesador tiene o no encadenamiento?

2. Siguiendo con el procesador definido en el ejercicio anterior, suponga que está acompañado de una memoria de la máxima capacidad cuyos contenidos iniciales expresados en hexadecimal son:

```
dir.  contenido
----  ---------
 0      4001
 1      6004
 2      2005
 3      E000
 4      000A
 5      0000
 6      A000
 7      0000
...     ...
```

   (a partir de la dirección 7 todos los contenidos son 0x0000)

   Comienza la ejecución a partir de dirección 0.

*a*) ¿Qué contenidos tendrá la memoria al final de la ejecución?

*b*) Si el contenido de la dirección 2 fuese 0x2003 ¿cuál sería el resultado?

*c*) ¿Y si, *además*, el contenido de la dirección 5 fuese 0x4000?

3. *a*) Para el mismo procesador, codifique en lenguaje ensamblador un programa que intercambie los contenidos de dos zonas de la memoria de acuerdo con estas especificaciones:
   - El programa se cargará a partir de la dirección 0.
   - Las zonas tienen 50 palabras y empiezan en las direcciones 100 y 200 (decimal).
   - Como sólo hay un registro aritmético, necesitará una palabra de la memoria para guardar resultados intermedios, y otra para un contador que puede ir de 50 a 1. Utilice, por ejemplo, direcciones a partir de la M[500].
   - Como tampoco hay registros de índice, el acceso a direcciones sucesivas puede conseguirse mediante el recurso de *modificar instrucciones en el curso de la ejecución*. Por ejemplo, si en la dirección *d*, en un momento dado, hay un contenido que se interpreta como la instrucción `LD 50` y en otro punto del programa se ejecutan instrucciones que le suman una unidad al contenido de *d*, la siguiente vez que se ejecute la primera instrucción se interpretará como `LD 51`.

   Hay varias maneras de hacerlo. Las instrucciones definidas son suficientes, pero si cree que le falta alguna instrucción puede inventársela utilizando el CO que está libre

   *b*) ¿Funcionaría igual el programa si se cargase en la memoria a partir de otra dirección?

4. Imagine un procesador hardware con una longitud de palabra de dieciséis bits, un repertorio de ocho instrucciones, cuatro registros de propósito general de dieciséis bits, y que puede direccionar una memoria de 4 KiB organizada por palabras (es decir, las direcciones son de palabra, no de byte). El repertorio incluye instrucciones que operan sobre un registro de propósito general y una palabra de la memoria. Por ejemplo: `LD r1,0x0AB`, o `ADDS r3,0x3FF`.

   *a*) ¿Cuál sería el tamaño mínimo en bits de estas instrucciones?

   *b*) En una instrucción como `ADDS r0,0x???`, ¿cuál es el valor máximo que puede tener «???»?

5. Suponga una memoria de 4 GiB organizada en palabras de 32 bits y un procesador que genera direcciones de palabra, no de byte.

   *a*) ¿Cuántas palabras se pueden almacenar en la memoria?

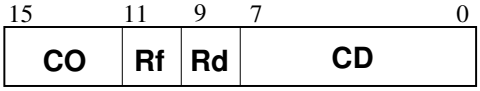
   *b*) ¿Cuál será el ancho del bus de direcciones?

   *c*) Si el procesador generase direcciones de byte, de modo que las direcciones de palabra fuesen 0, 4, 8..., ¿cuál sería el ancho del bus de direcciones?

   *d*) Suponiendo este último caso, ¿cuál será la dirección de palabra más alta?

   *e*) En el mismo caso, y si se reservan las 1.024 palabras de direcciones más altas para la pila, calcule el valor mínimo, en hexadecimal, que puede llegar a contener el registro puntero de pila. Suponga que, como es habitual, este registro apunta siempre a la cima de la pila, es decir a la dirección en la que se encuentra el último valor guardado.

6. Imagine un procesador con palabras y registros de dieciséis bits. Las direcciones de memoria son direcciones de palabra (no se puede acceder a un byte). El formato de instrucciones es:




Y algunas instrucciones son:

| **CO** (bin) | **CO** | **Significado** | **Ejemplo en ensamblador** (valores numéricos en decimal) |
|---|---|---|---|
| 0000 | `ADD` | (Rf)+(Rd) → Rd | `ADD R0,R1`: (R1)+(R0) → R0 |
| 0001 | `LD` | (M[CD]) → Rd | `LD R0,80`: (M[80]) → R0 (Rf indiferente) |
| 0010 | `ST` | (Rf) → M[CD] | `ST R0,80`: (R0) → M[80] (Rd indiferente) |
| 0011 | `LDi` | (CD) → Rd | `LDi R0,#80`: 80 → R0 (Rf indiferente) |

Rf y Rd direccionan a una sola memoria local.

*a*) ¿Cuántas instrucciones distintas puede tener?

*b*) ¿Cuántos registros en la memoria local?

*c*) ¿Cuántas direcciones de memoria *(espacio de direccionamiento)*?

*d*) La instrucción `LDi` carga un *operando inmediato* en Rd. Si este operando se interpreta como un número entero en complemento a 2 ¿qué rango de valores puede tener? Escriba en hexadecimal la codificación binaria de la instrucción `LDi R0,#-80` y el contenido de R0 después de ejecutarse esta instrucción.

*e*) ¿Clasificaría este procesador como CISC o como RISC?

7. Siguiendo con el procesador del ejercicio anterior, ahora nos dicen que hay dos instrucciones más, `LDR` y `STR`:

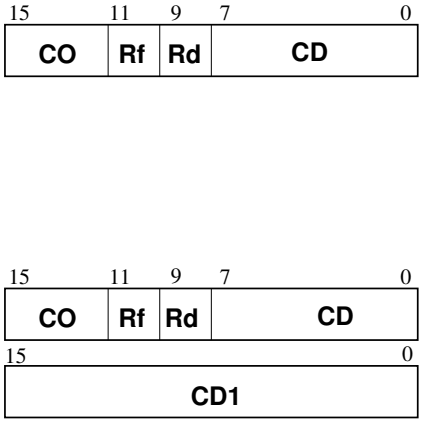
| **CO** (bin) | **CO** | **Significado** | **Ejemplo en ensamblador** |
|---|---|---|---|
| 0000 | `ADD` | (Rf)+(Rd) → Rd | |
| 0001 | `LD` | (M[CD]) → Rd | |
| 0010 | `ST` | (Rf) → M[CD] | |
| 0011 | `LDi` | (CD) → Rd | |
| 0100 | `LDR` | (M[(Rf)]) → Rd | `LDR R0,[R1]`: (M[(R1)]) → R0 |
| 0101 | `STR` | (Rf) → M[(Rd)] | `STR R0,[R1]`: (R0) → M[(R1)] |

Estas instrucciones tienen *direccionamiento indirecto* a través de registro.

¿Cambia alguna de las respuestas anteriores?

8. Y continuando con el mismo procesador, ahora nos enteramos de que también hay instrucciones que ocupan dos palabras, `LDL`, `STL` y `LDiL`. En ellas, el campo CD se extiende con los dieciséis bits de la segunda palabra:

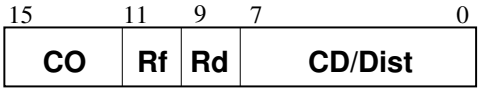
| **CO** (bin) | **CO** | Significado |
|---|---|---|
| 0000 | `ADD` | (Rf)+(Rd) → Rd |
| 0001 | `LD` | (M[CD]) → Rd |
| 0010 | `ST` | (Rf) → M[CD] |
| 0011 | `LDi` | (CD) → Rd |
| 0100 | `LDR` | (M[(Rf)]) → Rd |
| 0101 | `STR` | (Rf) → M[(Rd)] |
| 0110 | `LDL` | (M[CD,CD1]) → Rd |
| 0111 | `STL` | (Rf) → M[CD,CD1] |
| 1000 | `LDiL` | (CD,CD1) → Rd |




*a*) ¿Cambian las respuestas anteriores?

*b*) ¿Observa alguna incoherencia en este diseño?

9. Seguimos con el mismo procesador, pero con todas las instrucciones de una sola palabra (no existen las instrucciones `LDL`, etc.). Ahora el campo CD puede contener una distancia:




Y hay seis instrucciones nuevas, `ADDi`, `CMPi` `LDrel`, `STrel`, `BNE` y `HALT`:

| **CO** (bin) | **CO** | **Significado** | **Ejemplo en ensamblador** |
|---|---|---|---|
| 0000 | `ADD` | (Rf)+(Rd) → Rd | |
| 0001 | `LD` | (M[CD]) → Rd | |
| 0010 | `ST` | (Rf) → M[CD] | |
| 0011 | `LDi` | (CD) → Rd | |
| 0100 | `LDR` | (M[(Rf]) → Rd | |
| 0101 | `STR` | (Rf) → M[(Rd)] | |
| 0110 | `ADDi` | (Rf)+(CD) → Rd | `ADDi R0,R1,#5`: (R1)+5 → R0 |
| 0111 | `CMPi` | Compara (Rf) con (CD) y pone indicadores | `CMPi R0,#5`: Si (R0) = 5 pone Z=1, etc. |
| 1000 | `LDrel` | (M[DE]) → Rd | `LDrel R0,20`: si está en la dirección *d*, (M[*d*+ 2 + 20]) → R0 |
| 1001 | `STrel` | (Rf) → M[DE] | `STrel R0,-20`: si está en la dirección *d*, (R0) → M[*d* + 2 − 20] |
| 1010 | `BNE` | Si Z = 0, DE → PC | `BNE 100`: si está en la dirección *d* y si Z = 0, *d* + 2 + 100 → PC |
| ... | ... | ... | |
| 1111 | `HALT` | Detiene la ejecución | |

En las instrucciones `LD`, `ST`, `LDi`, `ADDi` y `CMPi` el campo CD/Dist es un número entero sin signo; en `LDrel`, `STrel` y `BNE` es un entero con signo y complemento a 2.

DE = (PC) + Dist = dir. instr.+ 2 + Dist

*a*) Escriba la codificación en hexadecimal de una instrucción que está en la dirección 5.000 y almacena (R3) en la dirección 5.020.

*b*) Escriba la codificación en hexadecimal de una instrucción que está en la dirección 5.000 y bifurca si Z = 0 a la dirección 4.980.

*c*) ¿El procesador tiene encadenamiento?

10. *a*) Codifique en un lenguaje nemónico (como un ensamblador, pero que no admite etiquetas), para el procesador definido en el ejercicio anterior, el programa del intercambio de zonas de 50 palabras que empiezan en las direcciones 100 y 200 (ejercicio 3), utilizando para el intercambio las instrucciones `LDR` y `STR`. Observe que en las instrucciones con direccionamiento inmediato el campo CD/Dist se interpreta como un número entero *sin signo*.

*b*) ¿Funcionaría igual el programa si se carga a partir de cualquier dirección de memoria (que no invada las zonas de datos: 100 a 149 y 200 a 249)? Compare con el resultado del ejercicio 3.

*c*) ¿Es posible adaptar el programa para que las zonas empiecen, por ejemplo, en 100 y 300?

11. Considere un procesador en el que todas las instrucciones ocupan dos bytes y están alineadas (es decir, las direcciones de las instrucciones son pares). Tiene una memoria local de registros de dieciséis bits, de los que R0 = PC es el contador de programa y R1 = LR es el registro de enlace. En el formato de instrucciones, los dos bits más significativos indican el tipo de instrucción, los dos siguientes, el código de operación y el resto depende del tipo. Los cuatro tipos de instrucciones son:

**T = 00:** instrucciones aritméticas y lógicas. En ellas los bits que siguen al código de operación se estructuran en tres campos que identifican, cada uno, a un registro: Rd, Rf1 y Rf2. Una instrucción de este tipo, escrita en ensamblador, sería, por ejemplo, «`add r3,r3,r2`» ((R2)+(R3) → R3)

**T = 01:** instrucciones de movimiento y de desplazamiento. Los cuatro bits que siguen al código de operación identifican a un registro, y el resto contienen un operando inmediato, entero con signo y complemento a dos. Por ejemplo: «`mov r2,#5`»

**T = 10:** instrucciones de transferencia de control. Los bits que siguen al código de operación representan un entero con signo y complemento a dos que se multiplica por dos para obtener la distancia. Por ejemplo, el resultado de ejecutar «`b 24`» sería: (PC) $+2\times24$ → PC, donde (PC) = dirección de esta instrucción + 4.

**T = 11:** instrucciones de acceso a la memoria y periféricos. Los cuatro bits que siguen al código de operación identifican a un registro, y el resto una distancia: un número con signo y complemento a dos. Por ejemplo: «`ldr r7,#-20`» ((PC) −20 → R7)

*a*) ¿Cuántas instrucciones en total puede tener el procesador?

*b*) ¿Cuantos registros en la memoria local? De ellos, ¿cuáles son de propósito general?

*c*) En una instrucción de movimiento como la del ejemplo, ¿qué rango de valores puede tener el operando inmediato?

12. Siguiendo con el mismo procesador,

*a*) Teniendo en cuenta la descripción dada para las instrucciones del Tipo 10, ¿qué rango de distancias, en bytes, se puede alcanzar desde una instrucción de bifurcación? Es decir, si la instrucción está en la dirección $d$, la dirección efectiva es $d + x$ o $d - y$, donde $x$ e $y$ son enteros positivos. ¿Qué valores máximos pueden tomar $x$ e $y$?

*b*) A la vista del ejemplo dado para las instrucciones del Tipo 10, ¿se puede deducir si el procesador tiene encadenamiento?

*c*) En las instrucciones del Tipo 11, ¿cuál es el rango de distancias en el que puede estar el operando con respecto a la dirección de la instrucción? (Como en la pregunta *a*).

## Sobre BRM y procesadores software

1. Suponga que en un procesador BRM los contenidos de los registros R1 y R2 son (R1) = 0xFFFFFF92 y (R2) = 0x0000006E, y que estos contenidos representan números enteros de 32 bits con convenio de complemento a 2 para los números negativos. Indique qué valores tomarán los indicadores Z, N, C, V después de ejecutar la instrucción `adds r3,r1,r2`.

2. Compruebe, con un ejemplo sencillo de sólo cuatro bits, que la condición «HI» (> sin signo) de BRM (tabla 9.1) se detecta con C = 1 y Z = 0.

3. Después de ejecutar estas tres instrucciones:

```
movs   r0,#cte1
cmp    r0,#cte2
addges r0,r0,#cte2 @ ge: mayor o igual con signo; s: pone indicadores
```

   ¿Con qué valores quedan (R0) y el indicador N, en función de los valores de `cte1` y `cte2`?

4. *a*) Escriba una sucesión de instrucciones, sin utilizar bifurcaciones, que realice la operación (R1)−(R2) → R0 y ponga en R3 el código ASCII de «+», el de «−» o el de «0» según que el contenido de R1 sea mayor, menor o igual que el de R2.
   (Sugerencia: use `SUBS`, `MOVHI`, `MOVMI` y `MOVEQ`)

   *b*) ¿Cómo sería el programa si, como ocurre en otros procesadores, ARM sólo tuviese condiciones para las instrucciones de bifurcación (es decir, no existiesen `MOVHI`, etc., sólo `MOV`)?

5. Dado este programa escrito en ensamblador para ARMSim#:

```
bucle:  add  r0,r0,r1  @  (R0)+(R1)-> R0
        mov  r3,r2     @  (R2)-> R3
        adds r2,r2,r0  @  (R0)+(R2)-> R2 y pone indicadores
        bvc  bucle     @   bifurca si V=0 (no desbordamiento)
        swi  0x11
```

   *a*) ¿Qué cálculo matemático se va realizando en las sucesivas veces que se ejecutan las instrucciones del bucle? Indique cómo se llama en matemáticas este tipo de cálculo, cuáles son sus parámetros, en qué registros se van guardando los parámetros y los resultados, y qué significado tienen los contenidos de R2 y R3 al terminar el programa.

   *b*) Teniendo en cuenta el formato de las instrucciones de bifurcación (figura 9.10), calcule la traducción a binario de la instrucción `bvc bucle`. Escriba el resultado en hexadecimal y explique cómo ha obtenido ese resultado.

*c*) ¿Qué modificación hay que realizar para convertir el programa anterior en un subprograma?

*d*) Escriba un programa principal que incluya las instrucciones necesarias para inicializar los registros con constantes definidas con la directiva `.equ` y una instrucción que llame al subprograma anterior. Ejecute el programa en el simulador ARMSim# y compruebe que realiza el cálculo esperado. ¿Qué contenidos tienen R0, R2 y R3 al finalizar la ejecución?

6. Si ha resuelto el ejercicio 11 de representación (operaciones para multiplicar por 10 un entero en coma fija) puede que haya llegado a una de estas dos soluciones:

   Solución 1:

   **1.** Desplazar $N$ dos veces a la izquierda $\rightsquigarrow 4 \times N$

   **2.** Sumar $N$ al resultado $\rightsquigarrow 5 \times N$

   **3.** Desplazar a la izquierda el resultado anterior $\rightsquigarrow 10 \times N$

   Solución 2:

   **1.** Desplazar $N$ una vez a la izquierda $\rightsquigarrow 2 \times N = N1$

   **2.** Desplazar $N1$ dos veces a la izquierda $\rightsquigarrow 4 \times N1 = 8 \times N = N3$

   **3.** Sumar $N1$ y $N3 \rightsquigarrow 2 \times N + 8 \times N = 10 \times N$

   *a*) Escriba subprogramas para cada una de estas soluciones. En ambos casos, el subprograma debe recibir el número $N$ en R0 y entregar el resultado también en R0.

   *b*) ¿Ve algún motivo por el que sea preferible una versión o la otra?

   *c*) Teniendo en cuenta que en ARM todas las operaciones de procesamiento y movimiento (`add`, etc.), y no sólo `mov`, permiten que al operando2 se le aplique una operación de desplazamiento, ¿se le ocurre alguna implementación de esta función que sea más eficiente?

7. Con este subprograma escrito para el lenguaje ensamblador de ARMSim# se suman las componentes de un vector almacenadas en sucesivas palabras de la pila:

```
sum_v_pila:
 ldr r4,[sp, #4]!
      ldr r0, [sp, #4]!
      sub r4, r4, #1
rep:  ldr r1, [sp, #4]!
      add r0, r0, r1
      subs  r4,r4,#1
      bne  rep
      mov pc, lr
```

   *a*) Explique su funcionamiento indicando, entre otras cosas:

   - cómo se sabe cuántas componentes del vector hay que sumar;
   - cómo se van sacando de la pila las distintas componentes;
   - qué modo de direccionamiento se usa;
   - qué funciones tienen los registros R0, R1 y R4.

*b*) Para poder probar el funcionamiento de este subprograma es preciso que el programa que lo llama introduzca previamente los valores correspondientes en la pila. Indique qué modo de direccionamiento se debe usar para incluir los valores en la pila.

*c*) Escriba un programa que introduzca en la pila tres componentes de un vector con valores 20, 7, 32 y luego llame al subprograma anterior.

*d*) Compruebe la ejecución del programa completo (la parte escrita en el punto *c* y el subprograma) en el simulador ARMSim#, seleccionando «View > Stack» para que el simulador muestre los contenidos de la pila. ¿Qué contenidos tienen las palabras de la pila de direcciones 0x5400 a 0x53F4?

8. *a*) Modifique el programa propuesto para la práctica del Laboratorio 6 de modo que en lugar de invertir la cadena "Nombre " la copie cambiando todos los caracteres a mayúsculas. (Observe en la tabla ASCII las diferencias en las codificaciones de las letras mayúsculas y las correspondientes minúsculas).

   *b*) ¿Qué ocurre si alguna de las letras de la cadena original es mayúscula? ¿Cómo puede evitarse?

   *c*) ¿Funcionará también si entre los caracteres hay eñes u otras letras con tilde?

9. El siguiente programa para ARMSim# calcula la letra (dígito de control) del DNI:

```
.text
.global _start
     .equ   DNI,14352384
_start:
      mov   r1, #DNI
rdiv23:
      sub r1, r1, #23
      cmp   r1, #23
      bgt   rdiv23
      ldr   r2,=letras
      ldrb  r0,[r2, r1]
      swi   0x11
.data
letras: .ascii "TRWAGMYFPDXBNJZSQVHLCKE"
.end
```

   *a*) Cargue el programa en el simulador y ejecútelo, empezando paso a paso, hasta que entienda bien cómo funciona el programa. Describa el método usado para obtener la letra que corresponde al DNI, en términos entendibles por cualquiera que no conozca ARM ni sepa programación. Indique qué letra corresponde al DNI 14352384 y dónde queda almacenada al terminar el programa.

   *b*) ¿Qué información se va guardando en los registros R1 y R2 durante el cálculo?

   *c*) Señale los diferentes modos de direccionamiento que se usan en el programa.

   *d*) Ponga como nuevo valor de DNI 16580608 y ejecute el programa. ¿Por qué no se obtiene la codificación ASCII de ninguna de las letras de la lista? ¿Qué hay que cambiar en el programa para que, con este valor de DNI, se obtenga la letra «T», que es la que le corresponde?

   *e*) Ponga ahora como nuevo valor de DNI 14352385, ¿Qué ocurre cuando vuelve a cargar el programa? ¿Por qué?

*f*) Indique qué otra modificación se debe hacer al programa para que genere siempre la letra de cualquier DNI válido sin que se produzca ningún error.

*g*) ¿Qué instrucción hay que añadir al programa para que el simulador escriba por la salida estándar la letra obtenida?

*h*) Escriba el programa resultante después de las tres modificaciones propuestas (puntos *d*, *f* y *g*).

10. Analice este programa fuente en el ensamblador GNU para ARM, con su traducción expresada en hexadecimal:

```
                .equ EXIT,1
                .equ WRITE,4
                .equ STDOUT,1
                .text
                .global _start
                _start:
0000 E3A00001       mov r0, #STDOUT
0004 E59F1014       ldr r1, =Saludo
0008 E3A02006       mov r2, #6
000C E3A07004       mov r7, #WRITE
0010 EF000000       svc 0x0
0014 E3A00000       mov r0, #0
0018 E3A07001       mov r7, #EXIT
001C EF000000       svc 0x0
                .data
0000 0A486F6C       Saludo:  .asciz "\nHola\n"
     610A00
                .end
```

*a*) ¿Cuántas llamadas al sistema hace el programa, y qué se pide en cada una?

*b*) En la instrucción `mov r2, #6`, ¿por qué el número «6»? ¿Qué ocurriría en la ejecución si en vez de «6» fuese «4»?

*c*) Fíjese en «`ldr r1, =Saludo`» ¿es una instrucción o una pseudoinstrucción? Si es una pseudoinstrucción, escriba la instrucción equivalente.

*d*) Teniendo en cuenta el formato de la instrucción `ldr` (figura 9.9), explique por qué el ensamblador la ha traducido como `E59F1014`.

- «E» (1110) porque...
- «5» (0101) porque...
- «9» (1001) porque...
- «F» (1111) porque...
- «1» (0001) porque...
- «014» (0...010100) porque...

11. En el apartado 10.8 proponíamos esta secuencia de instrucciones para leer caracteres del teclado (puertos 0x2000 y 0x2001) y hacer eco en un display (puertos 0x2002 y 2003) indefinidamente:

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

Escriba un programa completo que, además, haga que los caracteres que se van leyendo y escribiendo queden almacenados en la memoria en una zona de 100 KiB. El programa terminará cuando la zona esté llena o cuando se pulse `^D` (ASCII 0x04). A partir de este momento, el programa no debe hacer nada. Tenga en cuenta que este programa es el único en el sistema, de manera que no puede terminar con una llamada al sistema operativo.

12. Un controlador externo genera periódicamente interrupciones para BRM. Tiene dos puertos:

    - Al puerto de estado se le ha asignado (por hardware) la dirección 298, y al de datos la 299.
    - El puerto de datos sirve, fundamentalmente, para ajustar el período del reloj (frecuencia con la que el controlador genera las interrupciones). En este ejercicio no utilizaremos esta función (suponemos que un programa de inicialización ha escrito ya el valor de este período, que no se modifica: 1 seg.), pero sí tendremos que leerlo, de acuerdo con lo que se dice en el punto siguiente.
    - En el puerto de estado, el bit menos significativo («preparado») lo pone a «1» el controlador cada segundo, y se pone a «0» ejecutando una operación de lectura sobre el puerto de datos. El siguiente bit («permiso de interrupción») lo habrá puesto a «1» el programa de inicialización, y no lo tocaremos.

    Las instrucciones de la rutina deberán ejecutar el siguiente pseudocódigo:

    ```
    si SEG < 59 entonces SEG = SEG+1;
    si no {
       SEG = 0;
       si MIN < 59 entonces MIN = MIN + 1;
       si no {
           MIN = 0;
           HOR = HOR + 1
       }
    }
    ```

    «`SEG`», «`MIN`» y «`HOR`» son variables incluidas en la misma rutina a las que puede acceder también el programa de inicialización para poner sus valores a cero.

    *a*) Escriba en ensamblador la rutina de servicio para este controlador. Suponga que esta rutina es un subprograma llamado por la RIE (rutina de interrupciones externas), que ya se ocupa de salvar en la pila al entrar y restaurar al final todos los registros (de R0 a R12; R13=SP y R14=LR están duplicados para el modo supervisor y al volver al modo usuario permanecen con sus contenidos anteriores).

    *b*) ¿Cómo se modificaría la rutina en caso de utilizar un controlador de interrupciones hardware?

13. Tenemos conectados a BRM cuatro controladores de periféricos que funcionan por interrupciones sin controlador hardware, por lo que la consulta se hace mediante software con una rutina de interrupciones externas (RIE) cuyo listado fuente es el siguiente:

```
/* Rutina de interrupciones externas */
.global _start
.extern RS_ECG, RS_Reloj,
        RS_Tecl, RS_Displ
.text
_start:              @ 13 instrucciones
  str   r0,[sp],#-4  @ En ARM puede
  str   r1,[sp],#-4  @ hacerse con una
  ...                @ sola:
  str   r12,[sp],#-4 @ stmfd sp!,{r0,r12}
  ldr   r0,=296
  ldrb  r1,[r0]
  and   r1,r1,#3
  cmp   r1,#3
  bleq  RS_ECG
  ldr   r0,=298
  ldrb  r1,[r0]
  and   r1,r1,#3
  cmp   r1,#3
  bleq  RS_Reloj
  ldr   r0,=300
  ldrb  r1,[r0]
  and   r1,r1,#3
  cmp   r1,#3
  bleq  RS_Tecl
  ldr   r0,=302
  ldrb  r1,[r0]
  and   r1,r1,#3
  cmp   r1,#3
  bleq  RS_Displ
  ldr r12,[sp,#4]!@ En ARM se puede
  ...              @ hacer con una:
  ldr r0,[sp,#4]! @ ldmfd sp!,{r0,r12}
  subs pc,lr,#4
.end
```

Estos periféricos están conectados siguiendo un esquema similar al de la figura 10.2 en este orden:

1. El más cercano al procesador atiende a los datos que proceden de unos electrodos de ECG. Contiene un procesador de señal que cuando detecta cierta anomalía genera una interrupción. Su rutina de servicio pone en el registro de datos un valor que activa a unos circuitos que generan una alarma sonora y almacena en una zona de memoria las muestras de las señales durante los diez segundos siguientes. Se adjunta el pseudocódigo de esta rutina.

```
/* RS_ECG */
Pone un valor en el puerto
  de datos
Pone a 0 el bit PR del puerto
  de estado
Durante 10 segundos almacena
  en una zona de la memoria
  los datos que se van recibiendo
Vuelve a la RIE (mov pc,lr)
```

2. El siguiente es un generador de interrupciones de reloj como el descrito en la primera parte. Este controlador tiene, además, un pequeño visualizador y unos circuitos que hacen que se muestren continuamente las horas, minutos y segundos.
3. Un controlador de teclado.
4. Un controlador del display. Estos dos tienen también sus rutinas de servicio, como las escritas en pseudocódigo en la transparencia 64 del Tema 3-2.

Las direcciones fijadas en el hardware para los puertos son:

| | pto. estado | pto. datos |
|---|---|---|
| ECG | 296 | 297 |
| Reloj | 298 | 299 |
| Teclado | 300 | 301 |
| Display | 302 | 303 |

*a*) Para diseñar la RIE hay varias alternativas. Dos de ellas son:

- **Prioridad absoluta:** al terminar de ejecutarse una rutina de servicio se vuelve al modo usuario y si hay otra interrupción pendiente se vuelve a entrar en la RIE y se hace la consulta empezando por la más prioritaria hasta localizar la interrupción pendiente.
- **Prioridad circular, o «round robin»:** al terminar una rutina de servicio se vuelve a la RIE y se continúa consultando por si hay pendientes otras interrupciones.

¿Cuál de las dos alternativas es la que se sigue en la RIE dada?

*b*) Puesto el sistema en funcionamiento se observa que el reloj funciona mal. ¿En qué lo nota el usuario?

*c*) Llamado un técnico (un especialista electrónico), intercambia la posición de las tarjetas controladoras, de modo que la del reloj quede la primera (la más cercana al procesador). ¿Soluciona el problema? Razone la respuesta.

*d*) Llamado otro técnico (un programador) analiza el código de la RIE y observa que el ECG tiene prioridad sobre el reloj (pregunta primero por él). Edita el código fuente para que sea lo contrario: intercambia las instrucciones `ldr r0,=296` y `ldr r0,=298`, y también las instrucciones `bleq RS_ECG` y `bleq RS_Reloj`, ensambla el resultado y monta todos los módulos. ¿Soluciona el problema? Razone la respuesta.

*e*) Dé una solución razonada al problema introduciendo las modificaciones necesarias en el pseudocódigo dado anteriormente.

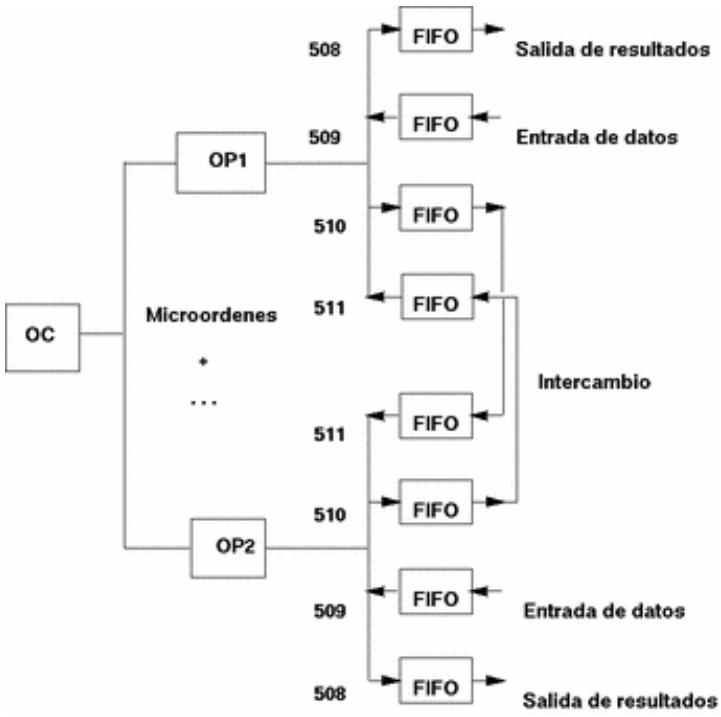
*f*) ¿Se le ocurre alguna manera más razonable de diseñar el controlador de los datos de ECG?

14. Este programa en C abre un fichero (se supone que es un fichero de texto), inserta en él una cadena de caracteres a partir del byte 80 y lo cierra:

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void main(int argc, char *argv[]) {
  int descr;
  char *cad = " ***insertado*** ";
  if ((descr = open(argv[1], O_RDWR)) < 0 ) {
    printf("\nError: no existe el fichero\n");
    exit(1)
  }
  else   lseek(descr, 80, SEEK_SET);
  write(descr, cad, 17);
  close(descr);
  exit(0);
}
```

Escriba un programa en lenguaje ensamblador de GNU para ARM que realice la misma función.

15. Se ha diseñado una arquitectura siguiendo el esquema que indica la figura, con tres procesadores: OC es un «órgano de control» y OP1 y OP2 son «órganos de procesamiento».




OC es un procesador como BRM, pero:

- carece de unidades aritmética lógica y de desplazamiento;
- genera señales de control (microórdenes) que se aplican en paralelo a OP1 y a OP2;
- genera otra información para OP1 y OP2

OP1 y OP2 son procesadores como BRM, pero:

- sólo tienen una unidad aritmética y lógica cada uno;
- pueden comunicarse con el exterior mediante los puertos 508 a 511:
  508: salida de resultados
  509: entrada de datos
  510: salida de datos hacia el otro OP
  511: entrada de datos desde el otro OP

- no es necesario hacer espera activa porque son entradas y salidas de memorias FIFO (es decir, colas: «el primero que entra es el primero en salir») de suficiente capacidad, y la disponibilidad de los datos se supone.

El OC *interpreta* (es decir, lee y decodifica las instrucciones; la ejecución la realizan OP1 y OP2 en paralelo) del siguiente programa (que está incompleto: deberá usted rellenar los espacios marcados con «?»).

```
.text
  ldr r0,=0x508
  ldr r1,=0x509
  ldr r2,=0x510
  ldr r3,=0x511
  ldr r4,#0   @ suma local
  ldr r5,#0   @ contador
bucle:
  add ???
  add r5,r5,#1
  ???
  bne bucle
  str r4,[r2]
  add r4,r4,[r3]
  ???
  mov r7,#1
  svc 0x0
.end
```

La función de este programa es:

- Cada OP toma 100 números de su entrada de datos y los suma
- Cada OP pasa el resultado de esa suma al otro OP
- Cada OP acumula a su propia suma la suma realizada por el otro OP y envía el resultado total a la salida de resultados

*a*) ¿En qué categoría de la clasificación de Flynn se puede incluir esta arquitectura?

*b*) Además de controlarlos con microórdenes, ¿qué información debe proporcionar OC a OP1 y OP2?

*c*) Complete el programa con las instrucciones que faltan, en los lugares señalados con «?»

16. Dados estos dos programas con los listados de los resultados de sus traducciones:

```
                /* Programa A */
                .global _start, s1, s2, res
                .extern suma3
                .text
                _start:
0000 E3A00001     mov r0,#1
0004 EBFFFFFE     bl suma3
0008 E3A00002     mov r0,#2
000C EBFFFFFE     bl suma3
0010 E3A07001     mov r7,#1
0014 EF000000     svc 0x0
                 .data
0000 03           s1: .byte 3
0001 05           s2: .byte 5
0002 00           res: .skip 1
                .end
```

```
                /* Programa B */
                .global suma3
                .extern s1, s2, res
                .text
                suma3:
0000 E59F101C     ldr  r1, =s1
0004 E59F201C     ldr  r2, =s2
0008 E5D11000     ldrb r1, [r1]
000C E5D22000     ldrb r2, [r2]
0010 E0803001     add  r3, r0, r1
0014 E0833002     add  r3, r3, r2
0018 E59F000C     ldr  r0, =res
001C E5C03000     strb r3, [r0]
0020 E1A0F00E     mov pc, lr
                 .end
```

*a*) Escriba las tablas de símbolos externos y de acceso y el diccionario de reubicación del Programa A y del Programa B.

*b*) Si el montador pone primero el Programa A, a continuación sus datos y finalmente el Programa B, ¿cuál será el diccionario de reubicación global? ¿Cuál será la dirección de comienzo (relativa) que generará el montador?

*c*) Si el montador pone primero el Programa B y a continuación el Programa A y sus datos, ¿cuál será el diccionario de reubicación global? ¿Cuál será la dirección de comienzo (relativa) que generará el montador?

# Apéndice E

# Soluciones de los ejercicios

---

## Sobre representación

1. En ASCII no (no hay código para la «ñ»). En Latin-1 22 bytes. En UTF-8 23 bytes (la ñ ocupa dos).

2. (40 lín/pág.) × (45 carac/lín.) × (2 columnas) = 3.600 carac/pág. = 3.600 bytes/pág.
   (3.600 bytes/pág.) × (780 pág.) = 2.808.000 bytes/libro
   2.808.000 × 2000 / (4 × 1.024 × 1024 × 1.024) = 1,31 (2 pinchos)

3. CD: 650 × 1.024 × 1.024 / 2.808.000 = 242,7
   DVD: 4,7 × 1.024 × 1.024 × 1.024 / 2.808.000 = 1.797,2

4. *a*)

| dir | cont.(dec) | cont.(bin) |
|---|---|---|
| d | 00 | 00000000 |
| d+1 | 00 | 00000000 |
| d+2 | 00 | 00000000 |
| d+3 | 00 | 00000000 |
| d+5 | 00 | 00000000 |
| d+5 | 65 | 01100101 |
| d+6 | 87 | 10000111 |
| d+7 | 09 | 00001001 |
| d+8 | 00 | 00000000 |
| d+9 | 80 | 10000000 |

   *b*) En nueve bytes caben 18 dígitos BCD. El número mayor es: 999.999... = 10 18 − 1, y el menor $-(10^{18} - 1)$

5. El número de segundos máximo que se puede representar en 31 bits es $2^{31} - 1$. En años, $(2^{31} - 1)/(60 \times 60 \times 24 \times 365) = 68{,}096$. Como la cuenta empieza el 1 de enero de 1970, sumando 68,096 años resulta algún día de 2038. Si tenemos en cuenta los años bisiestos, en 68 años hay 17, por lo que debemos restar a la cuenta 17/365 = 0,046 años, lo que no invalida el resultado anterior: X = 2038.

6. Una posible explicación es ésta:

Al insertar la tarjeta en el lector del terminal, éste envía al chip la fecha actual. Si el convenio del terminal es que los bits de cada byte son la representación en BCD, el año 2010 (o sea, «10») se representa así en un byte: 00010000. Si, por su parte, el chip de la tarjeta usa el convenio de que se trata de un entero en binario, interpretará que: 00010000 (binario) = 16 (decimal). Al comparar con la fecha de caducidad de la tarjeta, el programa de verificación incorporado en el chip rechaza la operación.

Este desajuste de convenios debería afectar también al mes y al día. Desde octubre de 2009 se rechazarían las tarjetas que caducasen entre octubre y diciembre de 2009: con el convenio BCD, el 1 de octubre de 2009 es 0x091001, mayor que la representación con el convenio binario de los últimos días de octubre (0x090A1F), noviembre (0x090B1E) y diciembre (0x090C1F). Y también las tarjetas que caducasen en septiembre, entre los días 20 a 30 de septiembre (con el convenio BCD el 20 de septiembre de 2009 es 0x090920, que es mayor que el 31 de septiembre de 2009 con el convenio binario, 0x09091F). No se sabe si tales problemas efectivamente aparecieron, atribuyéndose a fallos temporales, o si el programa trataba de forma diferente a los años de los días y meses.

Fuente:
`http://www.lightbluetouchpaper.org/2010/01/19/encoding-integers-in-the-emv-protocol/`

7. Si el reloj del sistema sigue el convenio BCD (por ejemplo, el 28 de febrero de 2010 estaría representado como 0x100228) y alguna parte del sistema operativo interpreta el año en binario creerá que se trata del 2016, que es un año bisiesto. Si esta parte está actualizando por su cuenta el mes y el día, a medianoche pasará al día 29 de febrero, en lugar del 1 de marzo. Y al enviar los datos a la red, el servidor los rechazará. Todo esto es especulativo, porque el fabricante sólo ofreció esta explicación: «Sabemos que la funcionalidad del reloj interno en las unidades PS3 que no son del modelo "slim" ha reconocido el año 2010 como un año bisiesto. Una vez que la fecha del reloj ha cambiado del 29 de febrero al 1 de marzo (ambos GMT), hemos comprobado que los síntomas se han resuelto y que los usuarios pueden ya usar normalmente su PS3».

   Fuente:
   `http://blog.us.playstation.com/2010/03/01/playstation-network-service-restored/`

8. XINU

9. *a*) +6: 00000000 00000000 00000000 00000110
   - -6 en c. a 1: 11111111 11111111 11111111 11111001
   - -6 en c. a 2: 11111111 11111111 11111111 11111010

   *b*)
   ```
   Extremista menor    Extremista mayor
   [d] 1111 1010       [d] 1111 1111
   [d+1] 1111 1111     [d+1] 1111 1111
   [d+2] 1111 1111     [d+2] 1111 1111
   [d+3] 1111 1111     [d+3] 1111 1010
   ```

   *c*) Las direcciones d a d+2 contienen 0xFF, que es la codificación de «ÿ», y la dirección d+3, 0xFA, que corresponde a «ú». Si el programa sigue también el convenio extremista mayor (primer carácter de la cadena por la izquierda en la dirección más baja) el resultado es que se imprime «ÿÿÿú».

10. Aplicando una máscara extraemos el byte menos significativo:

```
    0x31 32 33 34 = 00110001 00110010 00110011 00110100
AND 0x00 00 00 FF = 00000000 00000000 00000000 11111111
                    00000000 00000000 00000000 00110100)
```

    0b00110100 = 0x34 es la codificación de «4». Restándole 0x30 (codificación de «0») obtenemos el resultado:

```
    00110100
  + 11010000 (c. a 2 de 0x30)
 (1)00000100 (resultado: 4)
```

11. Una solución es:

    *a*) Desplazar $N$ dos veces a la izquierda $\rightsquigarrow 4 \times N$

    *b*) Sumar $N$ al resultado $\rightsquigarrow 5 \times N$

    *c*) Desplazar a la izquierda el resultado anterior $\rightsquigarrow 10 \times N$

12. «ROL8(operando)» significa «8 rotaciones a la izquierda del operando», y «MUL10(operando)» significa «multiplicar por 10 el operando»

    1. ROL8(0x31 32 33 34) $\rightsquigarrow$ 0x(32 33 34 31)
    2. (0x32 33 34 31) AND (0x00 00 00 FF) $\rightsquigarrow$ 0x31
    3. 0x31 - 0x30 = 0x1; MUL10(0x1) = 0xA
    4. ROL8(0x32 33 34 31) $\rightsquigarrow$ 0x(33 34 31 32)
    5. (0x33 34 31 32) AND (0x00 00 00 FF) $\rightsquigarrow$ 0x32
    6. 0x32 - 0x30 = 0x2; MUL10(0x2 + 0xA) = 0x78
    7. ROL8(0x33 34 31 32) $\rightsquigarrow$ 0x(34 31 32 33)
    8. (0x34 31 32 33) AND (0x00 00 00 FF) $\rightsquigarrow$ 0x33
    9. 0x33 - 0x30 = 0x3; MUL10(0x3 + 0x78) = 0x4CE
    10. ROL8(0x34 31 32 33) $\rightsquigarrow$ 0x(31 32 33 34)
    11. (0x31 32 33 34) AND (0x00 00 00 FF) $\rightsquigarrow$ 0x34
    12. 0x34 - 0x30 = 0x4; 0x4 + 0x4CE = 0x4D2

    Resultado (en 16 bits): 0x04D2 = 0000 0100 1101 0010

13. *a*) 0d21 = 0x15 = 0b0000...00010101;
    c. a 1: 1111...11101010;
    c. a 2: 1111...111101011
    Representación: 1111 1111 1111 1111 1111 1111 1110 1011
    Almacenamiento:

```
[d]   1110 1011   [d]   1111 1111
[d+1] 1111 1111   [d+1] 1111 1111
[d+2] 1111 1111   [d+2] 1111 1111
[d+3] 1111 1111   [d+3] 1110 1011
```

*b*) $21 = 10101 \times 2^0 = 0{,}10101 \times 2^5$;
M = 10101000...; c. a 2: 01010111 + 1 = 01011000...
$C = E + 2^{e-1} = 5 + 2^7 = 133 = 0x85 = 0b10000101$
Representación: 1 10000101 0101100...0
Almacenamiento:

```
[d]   0000 0000  [d]   1100 0010
[d+1] 0000 0000  [d+1] 1010 1100
[d+2] 1010 1100  [d+2] 0000 0000
[d+3] 1100 0010  [d+3] 0000 0000
```

14. En binario: 1100 0010 0010 1101 0000 0000 0000 0000

    Separando por campos: 1 10000100 01011010000000000000000,
    es decir: N = 1 (negativo); C = 0b10000100 = 0x84 = 0d132; M = 0b0101101

    Número representado = $-1{,}0b0101101) \times 2^{132-127}$ = -0b101011,01 = -0x2B,4 = -43,25

15. 0d-4.635,75 = 0x-121B; $C=(0x-0{,}121BC) \times 16^4 =$
    $(0b-0{,}00010010000110111100) \times 2^{16} = (0b-1{,}00100001101111) \times 2^{12}$

    C = 12 + 127 = 139 = 0x8B = 0b10001011

    Representación: 1 10001011 00100001101111000000000

    Puede usted comprobar, haciendo las operaciones inversas, que este resultado representa exactamente al número original, -4.635,75

16. $(0b-1{,}00100001101111) \times 2^{12} = (0b-0{,}100100001101111) \times 2^{13}$

    Mantisa en c. a 2: 011011110010001

    Característica: $C = 13 + 2^7 - 1 = 13 + 64 = 77 = 0x4D = 0b01001101$

    Truncando la mantisa, resulta la representación: 1 1001101 01101111, que corresponde al número decimal -4.640,0

    Si se trunca antes de hacer el complemento, resulta: 1 1001101 01110000, que corresponde al número decimal -4.608,0

## Sobre detección de errores y compresión

1. El mensaje dice «Soy yo».

   No se puede asegurar, porque, aunque no se ha detectado ningún error, podría haber un número par de errores en algún carácter y no se hubieran detectado

2. Por ejemplo, para «H» (0x48):

   Añadimos 0 a la derecha: 10010000. Al dividir por 11 resulta cociente 111000 y resto 0. Por tanto añadimos el bit de paridad 0 a la derecha: 10010000.

   Sin embargo, para «a» (0x61), al dividir 11000010 por 11 resulta cociente 1000001 y resto 1. El bloque es: 11000011.

3. `0x4E6574` = `0b01001110011100101011110100`; divisor: `10011` ($G(x) = x^4 + x + 1$)

```
0 1 0 0 1 1 1 0 0 1 1 0 0 1 0 1 0 1 1 1 0 1 0 0_0 0 0 0
  1 0 0 1 1
  0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1 0 1 0 0_0 0 0 0
            1 0 0 1 1
            0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 1 0 0_0 0 0 0
                        1 0 0 1 1
                        0 0 1 1 0 1 1 0 1 0 0_0 0 0 0
                            1 0 0 1 1
                            0 1 0 0 0 0 1 0 0_0 0 0 0
                              1 0 0 1 1
                              0 0 0 1 1 1 0 0_0 0 0 0
                                    1 0 0 1 1
                                    0 1 1 1 1_0 0 0 0
                                      1 0 0 1 1
                                      0 1 1 0_1 0 0 0
                                        1 0 0 1 1
                                        0 1 0_0 1 0 0
                                          1 0 0 1 1
                                          0 0_0 0 1 0
```

El resto resultante es 0010, que son los $k = 4$ bits que deben añadirse al mensaje. Por lo tanto el mensaje completo que se enviará es: 0x4E65742

4. *a*) Se han recibido $18 \times 4 = 72$ bits. El número de bloques es $72/24 = 3$.
   Como el polinomio generador es $x^5 + x^2 + 1$ (100101), se añaden 5 bits de redundancia, por lo que cada bloque contiene 5 bits de redundancia y $24 - 5 = 19$ bits de datos.

   *b*) Primer bloque: `0x21854A` = `0b001000011000100101001010` es divisible por `100101`:

```
0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 1 0 1 0_0 1 0 1 0
    1 0 0 1 0 1
    0 0 0 1 0 0 1 0 0 0 0 1 0 1 0 1 0_0 1 0 1 0
          1 0 0 1 0 1
          0 0 0 0 0 1 0 0 1 0 1 0 1 0_0 1 0 1 0
                    1 0 0 1 0 1
                    0 0 0 0 0 0 0 1 0_0 1 0 1 0
                                  1 0 0 1 0 1
                                  0 0 0 0 0 0 0 0
```

   por lo que no ha habido error. Lo mismo puede comprobarse para el segundo (`0x868965`). Pero no así en el tercero (`0x12CA89` = `0b000100101100101010001001`):

```
0 0 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 0 0_0 1 0 0 1
      1 0 0 1 0 1
      0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0_0 1 0 0 1
                1 0 0 1 0 1
                0 0 0 0 0 0 0 1 0 0_0 1 0 0 1
                              1 0 0 1 0 1
                              0 0 0_1 1 1 0 1
```

   en el que resulta un resto distinto de cero, por lo que ha habido algún error.

5. *a*) 2 canales× 16 bits/muestra× 44.100 muestras/seg. = 1.411.200 bps = 1,4112 Mbps.
   *b*) (1,4112×$10^6$ bps× 74×60 seg)/8 = 783,216×$10^6$ B = 783,216×$10^6/2^{20}$ MiB = 747 MiB.
   *c*) La explicación es que los requerimientos de detección y corrección de errores son mucho más fuertes cuando se trata de registrar información digital (un error en un bit no se aprecia en música, pero puede hacer inservible a un fichero). Los códigos detectores y correctores, al introducir redundancia, reducen la capacidad de almacenamiento. Concretamente, los sectores en Audio CD tienen 2.532 bytes aprovechables, mientras que los de CD-ROM tienen 2.048 bytes.

6. *a*) $109,44 \times 1000 \times 10/8 = 136.800$ B
   *b*) 136.800/15.200 = 9; factor de compresión: 9:1

7. num. fotos=capacidad/tamaño foto = $16 \times 2^{30}/16 \times 2^{20} = 1.024$

8. 12.000/1024 ≈ 12:1

9. *a*) La profundidad de color es 3×8 = 24 bpp
   *b*) $512 \times 512 \times 3 = 786.432$ bytes. Las diferencias con los tamaños de los ficheros se deben a las cabeceras de éstos.

10. 
    - Para Q=100, 786.572/(398×1.024) ≈ 2:1
    - Para Q=50, 786.572/(23×1.024) ≈ 33:1
    - Para Q=10, 786.572/(7,4×1.024) ≈ 100:1
    - Para Q=1, 786.572/(2,9×1.024) ≈ 265:1

11. Por el resultado de `hd` vemos que los seis primeros bytes son las codificaciones ASCII de «GIF89a». Los siguientes cuatro bytes son el tamaño de la imagen, expresada como número de píxeles horizontales×número de píxeles verticales. Estos dos números ocupan 16 bits cada uno y *siguen el convenio extremista menor:*

    «80 02» nos dice que tiene 0x0280 = $2 \times 16^2 + 8 \times 16 = 640$ píxeles horizontales.

    «E0 01» nos dice que tiene 0x01E0 = $16^2 + 14 \times 16 = 480$ píxeles verticales.

    Deducimos así que se trata de una imagen en formato GIF de dimensiones 640×480.

    Por otra parte, el resultado de `ls -l`, aparte de los datos sobre permisos y fecha de modificación, nos dice que el tamaño es «20K» (se entiende «20 KiB»). Como GIF utiliza para el color un byte, la imagen en formato pixmap debería ocupar 640×480 = 307.200 B = 300 KiB. El factor de compresión es, pues, 300:20 = 15:1.

12. Con una tasa de transferencia de 100 Mbps, el tiempo necesario para transferir 20.000×5 GiB es: $(20.000 \times 5 \times 2^{30} \times 8)/(100 \times 10^6 \times 3.600 \times 24) = 99,42$ días ≈ 3,3 meses.

    Coste: $3,3 \times 40 = 132$ €.

    Ancho de banda de la transmisión por carretera:

    $(20.000 \times 5 \times 2^{30} \times 8)/(8 \times 3.600) \approx 2,98 \times 10^{10}$ bps =29.800 Mbps

    Resumen:

| | **carretera** | **telemática** |
|---|---|---|
| Coste | 500 € | 132 € |
| Tiempo | 8 horas | 3,3 meses |
| Ancho de banda | 29.800 Mbps | 100 Mbps |

## Sobre procesadores hardware

1. *a*) `LD`, `ST` y `ADD` tienen direccionamiento directo (absoluto).

   `MOV` y `ADDI` tienen direccionamiento inmediato

   `BNE` tiene direccionamiento relativo

   *b*) Como CD tiene 13 bits, la capacidad de direccionamiento es $2^{13} = 8 \times 1.024$ palabras (16 KiB).

   *c*) Por los ejemplos, para `MOV` y `ADDI` el campo CD se interpreta como un número con signo. Suponiendo complemento a 2, el rango será $[-2^{12}, +2^{12} - 1]$, es decir, entre −4.096 y 4.095.

   *d*) No tiene encadenamiento porque, según el ejemplo, el contador de programa cuando se está ejecutando una instrucción está apuntando a la siguiente.

2. *a*) Para responder, primero tenemos que ver qué significado tienen esos contenidos cuando la CPU los interpreta como instrucciones. En la dirección 0: 0x4001 = 0b0100000000000001; vemos que los bits del código de operación son 010, que corresponden a `MOV`, y los siguientes indican el operando inmediato, 1. Haciendo lo mismo con las demás direcciones podemos escribir de forma nemónica los contenidos:

```
dir.  contenido contenido como instrucción
----  --------- --------------------------
 0      4001      MOV  #1
 1      6004      ADD  4
 2      2005      ST   5
 3      E000      HALT
 4      000A      LD   10
 5      0000      LD   0
 6      A000      BNE  0
 7      0000      LD   0
...     ....      ......
```

   Ahora ya es fácil seguir la ejecución: primero pone un 1 en el acumulador, luego le suma el contenido de la dirección 4, que es 10 (decimal) y el resultado, 0d11 = 0x000B, lo almacena en 5, tras lo cual se para (instrucción `HALT`). Los contenidos de 4, 5 y 6 no se han llegado a interpretar como instrucciones. La respuesta es que sólo se ha modificado el contenido de la dirección 5, que ha pasado de 0 a 0x000B = 0d11.

   *b*) En este caso, al ejecutarse la instrucción que está en 2, que ahora es `ST 3`, se «machaca» el contenido inicial de 3, que se convierte en 0x000B. La CPU pasa a ejecutar lo que hay en la dirección siguiente a la 2, la 3, cuyo contenido se acaba de modificar. Este nuevo contenido, interpretado como instrucción, es «`LD 11`»: carga en el acumulador el contenido de la dirección 11: 0x0000. Ahora la CPU ya no se para: pasa a ejecutar la siguiente (4), que pone en el acumulador el contenido de 10, también 0x0000 y la siguiente (5), que pone en el acumulador 0x4001, borrando el 0. Como este resultado es distinto de 0, la `BNE 0` bifurca a la dirección 0, donde vuelve a meter un 1 el acumulador y todo se repite indefinidamente: es un bucle sin fin. El único contenido que cambia es el de 3 (0x000B). En cada vuelta por el bucle se vuelve a cambiar, pero conservando el mismo valor.

   *c*) Hasta llegar a ejecutar el contenido de 5 todo es igual que antes, pero ahora ese contenido, 0x4000, interpretado como instrucción es `MOV #0`. Como se pone a 0 el acumulador, la

condición para la `BNE 0` no se cumple, y por tanto no se bifurca: la CPU pasa ejecutar lo que hay en la dirección siguiente, la 7: 0x0000, que, corresponde a `LD 0`. Por tanto, carga 0x4001 en el acumulador y pasa a la 8, que contiene lo mismo, y así sucesivamente. ¿Cuándo se para? Veamos: sabemos que cada vez que se lee una instrucción se incrementa el contador de programa para que apunte a la siguiente (en el caso de que ésta no sea una bifurcación). En este procesador, como la dirección máxima, $2^{13}-1$, se representa con trece unos, el contador de programa es un registro de trece bits. Al sumarle una unidad se ponen todos a cero (es un contador módulo $2^{13}$). Por tanto, después de ejecutar el contenido de la última dirección de memoria la CPU pasa a ejecutar la instrucción de dirección 0, y el resultado es el mismo anterior: un bucle sin fin (pero con muchas más instrucciones que repiten lo mismo: `LD 0`).

3. *a*) En esta solución se utiliza la dirección 500 como un contador de 50 a 1, y la dirección 501 para almacenamiento temporal de una palabra en el intercambio:

```
dir.
(dec).  Instrucc.  Comentarios
------  --------- ----------------------
     0  MOV  #50
     1  ST   500  ; pone el valor inicial 50 en el contador
     2  LD   100  ; las instrucciones 2 a 7 hacen el intercambio
     3  ST   501  ; almacenamiento temporal
     4  LD   200  ; las instrucciones 2 y 5 tienen que irse
     5  ST   100  ;  modificando para que se refieran a 100, 101..., 149
     6  LD   501  ;  y las instrucciones 4 y 7 a 200, 201..., 249
     7  ST   200
     8  LD     2  ; con las instrucciones 8 a 19 se modifican
     9  ADDI  #1  ;  las instrucciones 2, 4, 5 y 7
    10  ST     2  ;  antes de volver al bucle
    11  LD     4
    12  ADDI  #1
    13  ST     4
    14  LD     5
    15  ADDI  #1
    16  ST     5
    17  LD     7
    18  ADDI  #1
    19  ST     7
    20  LD   500  ; se actualiza
    21  ADDI #-1  ; el
    22  ST   500  ; contador
    23  BNE  -22  ; y se mira si ha llegado a 0; si no, 23+1-22 = 2
    24  HALT
```

   *b*) No funcionaría correctamente, porque las instrucciones que se tienen que ir modificando estarían en otras direcciones, y el programa siempre va a modificar las direcciones 2, 4, 5 y 7.

4. *a*) El formato de esas instrucciones debe tener tres campos:
   - Código de operación. Para 8 instrucciones, 3 bits ($2^3 = 8$)
   - Número de registro. Para 4 registros, 2 bits ($2^2 = 4$)

- Dirección de memoria. La memoria tiene 4 KiB = $2^{12}$ bytes. Las palabras son de dos bytes, por lo que será necesario direccionar $2^{11}$ palabras. Luego las direcciones son de 11 bits.

En total: 3 + 2 + 11 = 16 bits como mínimo.

*b*) Las direcciones de palabra están comprendidas entre 0 y $2^{11} - 1$. En binario, la dirección máxima es 111 1111 1111. En hexadecimal, 0x7FF.

5. *a*) Como cada palabra ocupa cuatro bytes, se podrán almacenar $4 \times 2^{30}/4 = 2^{30}$ palabras (más de mil millones)

*b*) La dirección más alta será $2^{30} - 1$, que en binario se escribe con treinta «1», por lo que el bus de direcciones tendrá 30 bits.

*c*) La dirección de byte más alta sería $2^{32} - 1$, que en binario se escribe con treinta y dos «1», por lo que el bus de direcciones tendría 32 bits.

*d*) Como la dirección de byte más alta es $2^{32} - 1$ = 0xFFFFFFFF y la palabra de dirección más alta ocupa cuatro bytes, esta dirección será 0xFFFFFFFF - 3 = 0xFFFFFFFC

*e*) 1.024 palabras son 4.096 bytes. En hexadecimal, 0x1000 bytes. 0xFFFFFFFF - 0x1000 + 0x1 = 0xFFFFEFFFF + 0x1 = 0xFFFFF000. Es decir, la zona reservada para la pila ocupa las direcciones 0xFFFFF000 a 0xFFFFFFFF. Por tanto, el puntero de pila no puede llegar a tener un contenido menor que 0xFFFFF000.

6. *a*) Como CO tiene 4 bits, $2^4$ = 16 instrucciones

*b*) $2^2$ = 4 registros

*c*) Como CD tiene 8 bits, $2^8$ = 256 direcciones de memoria

*d*) El rango de números enteros con 8 bits en c. a 2 es $[-2^7, +2^7 - 1] = [-128, +127]$

Codificación de `LDi R0,#-80`:

CO = 0011; Rf = 00 (o cualquier otro valor); Rd = 00;

CD = –80 en 8 bits y c. a 2 = 10110000

Resultado: 0b0011000010110000 = 0x30B0

Contenido de R0: –80 en 16 bits y c. a 2 = 0xFFFF30B0

*e*) Instrucciones de igual longitud, formato regular, «load/store» $\rightsquigarrow$ es un RISC

7. Al estar la dirección contenida en un registro de 16 bits, el *espacio de direccionamiento* en las instrucciones `LDR` y `STR` es $2^{16}$ direcciones de memoria (expresado en bytes serían $2 \times 2^{16}$ = 128 KiB).

8. *a*) Cambian las respuestas 3, 4 y 5:

- Con 8 + 16 = 24 bits para `LDL` y `STL` el espacio de direccionamiento con estas instrucciones es: $2^{24}$ direcciones (expresado en bytes, $2 \times 2^4 \times 2^{20}$ = 32 MiB).
- Para `LDiL`, el rango de operandos inmediatos es $[-2^{23}, +2^{23} - 1]$
- Instrucciones de distinta longitud $\rightsquigarrow$ es un CISC

*b*) La incoherencia está en la definición de `LDiL`: no puede cargarse un operando de 24 bits (CD,CD1) en un registro (Rd) de 16 bits

9. *a*) Instrucción en dirección 5.000: (R3) → MP[5.020]
CO = `STrel` = 1001
Rf = R3 = 11
Rd indiferente
Dist = 5.020 – 5.000 – 2 = 18 = 0b00010010 = 0x12

**Resultado:** `1001110000010010` **=** `0x9C12`

*b*) Instrucción en dirección 5.000 que bifurca a la dirección 4.980
CO = `BNE` = 1010
Rf y Rd indiferentes
Dist = 4.980 – 5.000 – 2 = –22 = -0x16 -0b00010110;
con 8 bits y complemento a 2: 11101001 + 1 = 11101010

**Resultado:** `1010000011101010` **=** `0xA0EA`

*c*) Según las expresiones dadas para DE, cuando se ejecuta la instrucción *i* el contador de programa está dos direcciones adelantado. Esto implica que sí hay encadenamiento: cuando se ejecuta *i* ya se ha leído *i* + 1 y se está leyendo *i* + 2.

10. *a*) Utilizando R0 y R1 como punteros que se van incrementando y R2 y R3 para el intercambio:

```
LDi R0,#100
LDi R1,#200 ; CD (8 bits) es sin signo, y 200 < 255
LDR R2,[R0]
LDR R3,[R1]
STR R3,[R0]
STR R2,[R1]
ADDi R0,R0,#1
ADDi R1,R1,#1
CMPi R0,#150
BNE  -9      ; si LDR R2,[R0] está en d, BNE está en d+7; d+7+2-9 = d
HALT
```

Si LDrel fuese relativo *e indexado*, podríamos incluir las zonas tras el código, en lugar de utilizar direcciones absolutas:

```
d   LDi R4,#0
d+1 LDRrel R1,R4,4  ; DE = d+1+4+ 4+(R4) = d+9+(R4)
d+2 LDRrel R2,R4,53 ; DE = d+2+4+53+(R4) = d+59+(R4)
d+3 STRrel R1,R4,52 ; DE = d+3+4+52+(R4) = d+59+(R4)
d+4 STRrel R2,R4,1  ; DE = d+4+4+1+(R4) = d+9+(R4)
d+5 ADDi R4,R4,#1
d+6 CMPi R4,#50
d+7 BNE -10         ; DE = d+7+4-10 = d+1
d+8 HALT
d+9 aquí empieza zona1
d+58 aquí termina
d+59 aquí empieza zona 2
```

Y con un ensamblador normal podríamos utilizar etiquetas:

```
        LDi R4,#0
bucle:  LDRrel R1,R4,zona1
        LDRrel R2,R4,zona2
        STRrel R1,R4,zona2
        STRrel R2,R4,zona1
        ADDi R4,R4,#1
        CMPi R4,#50
        BNE bucle
        HALT
zona1:  ...
zona2:  ...
```

*b*) Sí, funcionaría igual, porque, a diferencia del ejercicio 3, el programa no modifica sus propias instrucciones.

*c*) No hay una adaptación inmediata, porque la instrucción `LDi R1,#200` habría que cambiarla por `LDi R1,#300`, y esto no es posible con el campo CD de 8 bits. Una manera de hacerlo para este caso particular sería conservar `LDi R1,#200` y luego añadir `ADDi R1,R1,#100`.

11. *a*) Son cuatro tipos, y en cada uno el código de operación tiene dos bits, por lo que el número total es: $4 \times 2^2 = 16$ instrucciones.

*b*) Hay cuatro bits para codificar los registros, por lo que el número de registros es $2^4 = 16$. R0 y R1 tienen propósitos específicos; R2 a R15 son los de propósito general.

*c*) Hay $16 - 2 - 2 - 4 = 8$ bits para el operando inmediato;
el rango de valores es $[-2^7, +2^7 - 1] = [-128, +127]$.

12. *a*) Para la distancia hay $16 - 4 = 12$ bits. El rango de números representables es $[-2^{11}, +2^{11} - 1] = [-2048, +2047]$. Si la instrucción está en $d$, (PC) $= d + 4$, y la dirección efectiva mínima es $d + 4 - 2 \times 2048$ y la máxima $d + 4 + 2 \times 2047$. Por tanto, $y = 4092$ y $x = 4098$.

*b*) Cuando la instrucción se está ejecutando PC contiene la dirección de esta instrucción más cuatro, es decir, la dirección de la instrucción siguiente a la siguiente. Por tanto sí tiene encadenamiento.

*c*) En este caso para la distancia hay $16 - 8 = 8$ bits. El rango de números representables es $[-2^7, +2^7 - 1] = [-128, +127]$. Si la instrucción está en $d$, (PC) $= d + 4$, y la dirección efectiva mínima es $d + 4 - 128$ y la máxima $d + 4 + 127$. Por tanto, $y = 124$ y $x = 131$.

## Sobre BRM y procesadores software

1. Z = 1, N = 0, C = 1, V = 0

   Explicación: El resultado de sumar 0xFFFFFF92 y 0x0000006E es 0 (no negativo) por ser números complementarios. Y se produce acarreo pero en el bit de signo por lo que no hay desbordamiento.

2. Pongamos, por ejemplo, $x - y$, con $x$ = 0b1011 (0d11 sin signo) e $y$ = 0b1001 (0d9 sin signo). Una ALU de 4 bits, para restar, sumaría el complemento a 2 del sustraendo (independientemente de que la interpretación sea con o sin signo).
   ca2(1001) = 0111; 1011+0111 = 10010. Vemos que C = 1 y Z = 0.

   Ejemplos similares pueden comprobarse para las demás condiciones.

3. 
- Si cte1 ≥ cte2:
  - la primera instrucción pone 0 o 1 en N según el signo de cte1,
  - pero la segunda instrucción (compara cte1 con cte2) pone N = 0 (porque cte1 - cte2 ≥ 0)
  - y la condición para la tercera se cumple; al ejecutarse resulta (R0) = cte1 + cte2, y N depende del signo del resultado.
- Si cte1 < cte2:
  - la primera instrucción pone 0 o 1 en N según el signo de cte1;
  - la segunda instrucción pone N = 1 (porque cte1 - cte2 < 0)
  - la condición para la tercera no se cumple, por tanto no se ejecuta y queda (R0) = cte1 y N = 1

4. *a*)
```
subs r0, r1, r2
movhi r3, #0x2B  @ código ASCII de +
moveq r3, #0x30  @ código ASCII de 0
movmi r3, #0x2D  @ código ASCII de -
```

*b*) Si no existen `movhi`, etc. (pero sí `bhi`, etc.):
```
        subs r0, r1, r2
        beq igual
        bmi menos
        mov r3, #0x2B  @ código ASCII de +
        b seguir
igual:  mov r3, #0x30  @ código ASCII de 0
        b seguir
menos:  mov r3, #0x2D  @ código ASCII de -
seguir: ...
```

5. *a*) Suma los términos de una progresión aritmética cuya diferencia está contenida en el registro R3. Al iniciarse el programa R0 contiene el primer término y en cada bucle contiene el término siguiente. La suma de los sucesivos términos se va guardando en R2 pero antes de sumar el siguiente término su valor se copia en R3. Así, al finalizar el programa por desbordamiento, R2 contiene un valor no válido, mientras que R3 contiene el último valor válido de la suma antes de desbordarse.

*b*) Codificación de la instrucción: 7AFFFFFB

Explicación:

Código de condición VC: 0b0111 = 0x7

Tipo: 0b10 y L = 0; por tanto, segundo byte = 0b1010 = 0xA

4×Dist = DE – dir.instr – 8 = 0x100C – 0x1020 = – 20; por tanto, Dist = –5 = 0xFFFFFB

*c*) Basta cambiar `swi 0x11` por `mov pc, lr`. Si va a ser traducido independientemente del programa hay que añadir la directiva `.global bucle`

*d*)
```
.equ  a1,1    @ primer término
.equ  d,7     @ diferencia
 mov r0, #a1  @ Carga valores iniciales
 mov r1, #d
 mov r2, r0
 bl bucle
 swi 0x11
```

Los contenidos finales dependen de los valores puestos con la directiva `.equ`. Si son, por ejemplo, (R0) = 1, (R1) = 7, al finalizar serán: (R0) = 236289, (R2) = -2147445631 y (R3) = 2147285376

6. *a*) Solución 1:

```
mult10: mov r1, r0, lsl#2 @ 4×N → R1
        add r1, r1, r0    @ N + 4×N → R1
        mov r0, r1, lsl#1 @ 2×5×N → R0
        mov pc, lr
```

Solución 2:

```
mult10: mov r1, r0, lsl#1 @ 2×N → R1
        mov r2, r1, lsl#2 @ 4×2×N → R2
        add r0, r1, r2    @ 8×N + 2×N → R0
        mov pc, lr
```

*b*) Ambas versiones tienen tres instrucciones `mov` y una `add`. Desde el punto de vista de tiempo de ejecución no hay diferencia. Pero la segunda solución utiliza R1 y R2 como registros de trabajo, y la primera solamente R1. Es preferible utilizar el menor número posible de registros, porque sus valores quedan modificados, y en el programa invocante hay que tenerlo en cuenta y no usarlos si se quiere conservar sus valores (o, mejor, guardarlos en la pila al principio del subprograma y recuperarlos antes de volver).

*c*)
```
mult10: add r0, r0, r0, lsl#2 @ N + 4×N = 5×n → R0
        add r0, r0, r0        @ 5×N + 5×N → R0
        mov pc, lr
```

Ventajas: una instrucción menos (25 % menos de tiempo de ejecución) y solamente utiliza el registro R0.
Comprobación experimental: un bucle que realiza 1.000 millones de veces la operación 255×10 en una Raspberry Pi se ejecuta en 11,7 segundos con las dos primeras versiones y en 8,75 segundos con la tercera (hay que tener en cuenta que a las instrucciones anteriores hay que añadir las de control del bucle). Si se utiliza la instrucción `mul` de ARM el tiempo es 8,7 segundos (sólo es una instrucción en lugar de tres, pero implementa un algoritmo genérico de multiplicación, que requiere varios ciclos de reloj).

7. *a*)
   - El número de componentes es el contenido de la cima de la pila (el último introducido)
   - Las componentes se van sacando con la instrucción `ldr r1, [sp, #4]!`, que se repite (bucle `rep`) un número de veces igual al contenido de la primera palabra sacada de la pila.
   - Se usa direccionamiento preindexado inmediato, utilizando como registro de direccionamiento el puntero de pila e incrementando su contenido en 4 en cada ejecución
   - En R0 se van guardando las sucesivas sumas de componentes, en R1 se recogen los sucesivos valores de componentes extraídos de la pila, y R4 actúa de contador, inicializándose con el número de componentes (primer valor extraído de la pila).

*b*) Para incluir los valores en la pila se utilizará direccionamiento postindexado inmediato, usando como registro de dirección el puntero de la pila y con un valor de incremento igual a –4. Serán instrucciones de la forma `str r2, [sp], #-4`.

*c*)
```
_start:
    mov r2, #20
    str r2, [sp], #-4 @ guarda en la pila el valor
    mov r2, #7
    str r2, [sp], #-4
    mov r2, #32
    str r2, [sp], #-4
    mov r2, #3        @ número de componentes
    str r2, [sp], #-4
    bl  sum_v_pila
    swi   0x11
```

*d*)
```
000053F4   00000003
000053F8   00000020
000053FC   00000007
00005400   00000014
```

8. *a*) En la tabla ASCII ponemos ver que la codificación de «A» es 0d065 y la de «a» 0d097, y que en todas las letras sucesivas la diferencia entre la minúscula y la mayúscula es 32 (decimal). Por tanto, la modificación consiste en:
   - No recorrer la cadena destino de atrás adelante (`strb r2, [r1], #-1`), sino de adelante atrás (`strb r2, [r1], #1`).
   - A cada paso, restar 32 al byte en R2 antes de copiarlo.
   - Tener cuidado de conservar R2 porque se utiliza para comprobar el fin de la cadena (al final hay una nota sobre esto).

   Este programa incluye ya las modificaciones para las dos cuestiones siguientes:

```
        .text
        .global   _start
        .equ      NUL, 0
_start:   ldr   r0, =nombre
          ldr   r1, =NOMBRE
bucle:    ldrb  r2, [r0], #1
          mov   r4,r2     @ R2 hay que conservarlo, para la comparación
          cmp   r2,#91  @ ¿Es menor que el siguiente a "Z"?
          bcc   no      @ CC: < sin signo
          cmp   r2,#NUL @ Si es NUL, tampoco
          beq   no
            @ También hay que comprobar para 192 <= (R2) <= 221 (À - Ý)
          cmp   r2,#192    @ si es >= "Á" mirar si es <= "Ý"
          bcs   mirar      @ CS: >= sin signo
si:       sub   r2,r2,#32  @ Sólo se ejecuta si hay que alterar (R2)
no:       strb  r2, [r1], #1 @   pero ésta se ejecuta siempre
          cmp   r4, #NUL     @ Comprobamos con el byte original guardado
          bne   bucle
          swi   0x11
mirar:    cmp   r2,#222
          bcc   no        @ Es mayúscula con tilde
          b     si
       .data
nombre: .asciz "NÑAáÁeñ"
NOMBRE: .skip 8
      .end
```

*b*) Si alguna de las letras ya es mayúscula resultaría una codificación que no tiene nada que ver (por ejemplo, para «A» resultaría «!»). Se evita comprobando si la codificación es menor o igual que la de «Z» y en ese caso no restar 32. Otro detalle es el del byte de fin de la cadena (NUL), que tampoco hay que modificar.

*c*) En la tabla de ISO-Latin1 podemos ver que la diferencia de 32 se mantiene entre minúsculas y mayúsculas con tilde, por tanto funciona correctamente. Pero de nuevo hay que comprobar que la letra original no sea mayúscula, lo que es un poco más laborioso, como se ve en programa anterior. Este programa funciona siempre que la cadena original esté codificada en ISO-Latin1 (o ISO-Latin9). En efecto, en UTF-8 también hay una diferencia de 32 entre los puntos de código, pero las codificaciones ISO entre 0x80 y 0xFF se transforman según se indica en la tabla 3.4, y las comprobaciones son conceptualmente sencillas pero laboriosas: requieren máscaras y desplazamientos.

**Nota:** Si no se toma la precaución de conservar R2, el programa aparentemente también funciona, *siempre y cuando* el nombre no termine con (o contenga) un espacio en blanco. En efecto, el código ASCII del blanco es 0x20 = 0d32, y al restarle 32 resulta 0, que es el código de NUL.

9. *a*) El programa calcula el resto de dividir por 23 el valor del DNI y busca entre las 23 letras, almacenadas en la zona `letras`, la letra situada en la posición correspondiente al valor del resto. Al terminar el programa el registro R3 guarda la codificación ASCII de esta letra.

   *b*) Inicialmente R1 contiene el valor del DNI, y mientras el resultado sea mayor que 23 se le resta 23 en cada vuelta. R2 contiene siempre la dirección de la zona donde están guardadas las 23 letras.

   *c*) Modos de direccionamiento:

   - inmediato (`mov r1,#DNI`; `sub r1, r1,#23`; `cmp r1,#23`)
   - preindexado con registro (`ldrb [r0,r1]`)
   - relativo a programa (`bgt rdiv23` y `ldr r2,=letras`, que equivale a `mov r2,[pc, #dist]`)

   *d*) El valor del DNI dado es múltiplo de 23, por lo que, como la condición de salida del bucle es GT (mayor que), se sale del bucle con el valor (R1)=23 y la dirección efectiva de la instrucción `ldrb r0,[r2,r1]` es (R2)+23, apuntando al byte de valor 0x00 posterior a la última letra. Se evita ese error cambiando la condición GT por GE con lo cual se dará una vuelta más al bucle y, al restar de nuevo 23, dará como resultado (R1)=0 siendo ahora la dirección efectiva (R2), que apunta a la primera letra, «T».

   *e*) Al poner `.equ DNI,14352385` el ensamblador indica error porque no puede representar el número 14352385 en el operando 2 ya que no cumple los requisitos necesarios para ser representable.

   *f*) Sustituyendo la instrucción `mov r1,#DNI` por la pseudoinstrucción `ldr r1, =DNI` se evita el error (la pseudoinstrucción se traducirá por `mov` si el valor de DNI es un número representable como operando 2, y si no, se guardará la constante y se accederá a ella con una instrucción `ldr` y direccionamiento relativo a la dirección de esa constante).

   *g*) Con `swi 0x00` se saca por StdOut el carácter contenido en R0.

*h*)
```
/*********************************************************************
* Calcula la letra del DNI, la guarda en R3 y la muestra por StdOut *
*********************************************************************/
.text
    .global  _start
    .equ   DNI,17818559  @ puede ponerse cualquier valor de 8 cifras
_start:  ldr r1, =DNI    @ carga en r1 el valor del DNI
rdiv23:  sub r1,r1, #23  @ bucle para obtener el resto de dividir por 23
         cmp r1, #23       @ restándole sucesivamente el valor 23
         bge rdiv23        @ mientras sea mayor o igual que 23
         ldr   r2,=letras  @ Carga en r2 la dirección donde están las letras
         ldrb  r0,[r2, r1] @ Carga en r0 la letra correspondiente en ASCII
     swi   0x00          @ Escribe la letra por StdOut
     swi   0x11          @ Interrumpe el programa para finalizarlo
.data
letras: .ascii "TRWAGMYFPDXBNJZSQVHLCKE"   @ secuencia de letras para el DNI
.end
```

10. *a*) Hay dos llamadas. En la primera se pide escribir una cadena por la salida estándar, y en la segunda la terminación de la ejecución.

    *b*) Porque es el número de bytes que tiene la cadena a escribir. Si se pone `mov r2, #4` sólo se escriben los caracteres `\nHol`.

    *c*) Es una pseudoinstrucción. La instrucción equivalente es `ldr r1, [pc, #dist]`, donde `dist` es la distancia desde el PC hasta la palabra que contiene la dirección de `Saludo`.

    *d*)
    - «E» (1110) porque... no hay condición.
    - «5» (0101) porque... es tipo 01 con modo preindexado inmediato (I = 0, P = 1).
    - «9» (1001) porque... la distancia se suma (U = 1), es `LDR` (B = 0), Rn = PC no se modifica (W = 0), y es `LDR` (L = 1).
    - «F» (1111) porque... Rn = PC.
    - «1» (0001) porque... Rd = R1.
    - «014» (0...010100) porque... es la distancia a sumar a PC para obtener la dirección de la palabra en la que el montador pone la dirección de la cadena: (PC) = 0x4 + 0x8 = 0xC; dir(`Saludo`) = 0x1C + 4 = 0x20; 0x20 - 0xC = 0x14.

11.
```
/* Lee y hace eco indefinidamente */
.global _start
.text
.equ fin,0x4
.equ K100,100*1024
_start: ldr  r5,=buffer
        mov  r6,#0
        ldr  r7,=K100
        ldr  r1,=0x300
        ldr  r2,=0x301
        ldr  r3,=0x302
        ldr  r4,=0x303
sgte:   bl esptecl
        cmp  r0,#fin
        beq  final
        strb r0,[r5],#1
        adds r6,r6,#1
        cmp  r6,r7
        blt  sgte
final:  b final
.data
buffer: .skip K100
    .end
```

(Si este programa es el único en el sistema y no hay ningún otro que pueda interpretar la llamada `EXIT` la única manera de que no haga nada más es con una instrucción que bifurca a sí misma: `final: b final`).

12. *a*) Rutina de servicio:

```
.text
.global RS_Reloj, SEG, MIN, HOR
RS_Reloj:
   ldr  r2,=299 @ Sólo para leer
   ldrb r2,[r2] @ el puerto de datos
                @ y así 0 → PR
   ldr  r2,=SEG @ (R2) = dir. de SEG
   ldrb r3,[r2] @ (R3) = segundos
   cmp  r3,#59
   bcc  increm
   mov  r3,#0
   strb r3,[r2] @ 0 → SEG
   ldr  r2,=MIN @ (R2) = dir. de MIN
   ldrb r3,[r2] @ (R3) = minutos
   cmp  r3,#59
   bcc  increm
   mov  r3,#0
   strb r3,[r2] @ 0 → MIN
   ldr  r2,=HOR @ (R2) = dir. de HOR
   ldr  r3,[r2] @ (R3) = horas
   add  r3,r3,#1
   str  r3,[r2]
   mov  pc,lr   @ Vuelve
increm:
   add  r3,r3,#1
   strb r3,[r2]
   mov  pc,lr   @ Vuelve
.data
   SEG: .byte 1
   MIN: .byte 1
   .align  @ Necesario para alinear HOR
   HOR: .word 1    @ Por si HOR > 255
.end
```

*b*) Si hay un controlador hardware no se ejecuta la RIE, que se ocupaba de salvar los registros y de volver al programa interrumpido (como se ve en el listado de la siguiente pregunta). En la rutina de servicio hay que:

- añadir al principio instrucciones para salvar en la pila los registros R0 a R12 y al final otras tantas para recuperarlos, y
- sustituir la instrucción final de retorno a la RIE (`mov pc,lr`) por la de retorno de interrupción: `subs pc,lr,#4`.

13. *a*) Como podemos comprobar en el programa, al volver de la rutina de ECG (`bleq RS_ECG`) no se sale de la RIE, sino que se comprueba si está pendiente una interrupción de reloj, y lo mismo al volver de `RS_Reloj` y de `RS_Tecl`. Por tanto se sigue el algoritmo de prioridad circular.

*b*) Lo nota en que el reloj atrasa. Cada vez que entra la rutina de servicio del controlador del proceso, como tarda 10 segundos, se pierden 10 interrupciones de reloj, es decir, 10 segundos son contados como uno.

*c*) No soluciona nada: la proximidad física de los controladores no tiene nada que ver en este caso.

*d*) Tampoco lo resuelve: el problema no es por quién se pregunta primero (en el caso de que ambos soliciten la interrupción al mismo tiempo), sino que mientras se está ejecutando la rutina del ECG todas las interrupciones están inhibidas.

*e*) Hay que permitir el anidamiento de interrupciones. Para eso hay que modificar la rutina de servicio del controlador de ECG:

```
/* RS_ECG */
Inhibe las interrupciones de este controlador (0 → IT)
Da el permiso general de interrupciones (1 → I en el registro CPSR)
Pone un valor en el puerto de datos
Pone a 0 el bit PR del puerto de estado
```

```
Durante 10 segundos almacena en una zona de la memoria
     los datos que se van recibiendo
Inhibe las interrupciones en general (0 → I en el registro CPSR)
Permite las interrupciones de este controlador (1 → IT)
Vuelve a la RIE (mov pc,lr)
```

(Se supone que la RIE salva y luego restaura todos los registros).

*f*) El que la rutina de servicio se apodere durante 10 segundos del procesador no es razonable (no ya sólo por el reloj, sino por el teclado y el display). Podría hacerse mediante un controlador de acceso directo a memoria (robando un ciclo cada vez que se recibe una muestra de ECG), pero en este caso es más fácil el procedimiento de las interrupciones, ya que el periférico es lento: la frecuencia de muestreo típica para el ECG es 200 Hz. El controlador debe diseñarse de modo que se inicialice al detectar la anomalía y a partir de entonces interrumpa cada vez que recibe una muestra, y esto durante $10 \times 200$ muestras (con un contador en la rutina de servicio). Entre muestra y muestra el procesador, suponiendo 1.000 MIPS, puede ejecutar $10^9/200 = 5$ millones de instrucciones.

14.
```
.equ EXIT,1
.equ WRITE,4
.equ OPEN,5
  .equ RDWR,2
.equ CLOSE,6
.equ LSEEK,19
  .equ SEEK_SET,0
.equ STDOUT,1

.text
.global _start
_start:
/* if ((descr = open(...) */
   ldr  r0,[sp,#8]  @ argv[1]
   mov  r1,#RDWR
   mov  r7,#OPEN
   svc  0x0
   cmp  r0,#0
   blt  error
/* lseek(descr, 80, SEEK_SET) */
   ldr  r1, =descr
   strb r0, [r1]
   mov  r1, #80
   mov  r2, #SEEK_SET
   mov  r7, #LSEEK
   svc  0x0
/* write(descr, cad, 17) */
   ldr r0, =descr
   ldrb r0, [r0]
   ldr r1, =cad
   mov r2,#17
   mov r7, #WRITE
   svc 0x0
/* close(descr) */
   ldr r0, =descr
   ldrb r0, [r0]
   mov r7, #CLOSE
   svc 0x0
   mov r0, #0
   mov r7, #EXIT
   svc 0x0
error:
/* printf("\ nError: no existe el fichero\ n") */
   mov  r0, #STDOUT
   ldr  r1, =msg
   mov r2, #30
   mov r7, #WRITE
   svc 0x0
/* exit(0) */
   mov r0, #0
   mov r7, #EXIT
   svc 0x0
.data
   descr: .byte 1
   cad:   .ascii " ***insertado*** "
   msg:   .asciz "\ nError: no existe el fichero\ n"
   .end
```

**Nota:** Si se compila el programa en C y se ensambla la versión en ensamblador, y se ejecutan, se puede observar que funcionan igual: la cadena se inserta a partir del byte 80, pero borrando los que hay almacenados a partir de él hasta el final de la cadena. Para insertar sin borrar no hay ninguna llamada (ni tampoco función en la biblioteca estándar de C) que permita hacerlo. Una

forma sería desplazar previamente los bytes uno a uno para «hacer hueco». Otra, crear un nuevo fichero, copiar en él los bytes 0 a 79, después la cadena y después los bytes restantes del fichero original.

15. *a*) Es un SIMD: OC es un procesador de instrucciones, y OP1 y OP2 son procesadores de datos

*b*) Además de las microórdenes para ejecutar `add`, `bne`, `mov` y `svc`, OP1 y OP1 necesitan recibir la *dirección* para las instrucciones `ldr` y `str`

*c*) Complete el programa con las instrucciones que faltan, en los lugares señalados con «?»

```
.text
  ldr r0,=0x508
  ldr r1,=0x509
  ldr r2,=0x510
  ldr r3,=0x511
  ldr r4,#0    @ suma local
  ldr r5,#0    @ contador
bucle:
  add r4,r4,[r1]
  add r5,r5,#1
  cmp r5,#100
  bne bucle
  str r4,[r2]
  add r4,r4,[r3]
  str r4,[r0]
  mov r7,#1
  svc 0x0
.end
```

16. *a*)

| | Símbolos externos | Símbolos de acceso | Diccionario de reubicación |
|---|---|---|---|
| Prog. A | `suma3/0x0004;`<br>`0x000C` | En la sección .text:<br>`_start/0x0000`<br>En la sección .data:<br>`s1/0x0000`<br>`s2/0x0001`<br>`res/0x0002` | Vacío |
| Prog. B | `s1/0x0024`<br>`s2/0x0028`<br>`res/0x002C` | `suma3/0x0000` | `0x0024; 0x0028; 0x002C` |

*b*) Si el montador pone primero el Programa A, luego sus datos y a continuación el Programa B, la dirección de la primera instrucción de B sería, en principio, la siguiente a la última del último byte de los datos de A, es decir, 0x001B. Pero no es múltiplo de 4, lo que daría un error en la ejecución. El montador pondrá la instrucción a partir de la primera dirección múltiplo de 4: 0x001C. A todas las direcciones obtenidas por el ensamblador para el programa B se les suma 0x001C. El diccionario de reubicación, que eran las direcciones 0x0024 y siguientes resultará: 0x0040; 0x0044; 0x0048.

La dirección de comienzo de la ejecución (la de `_start`) relativa al comienzo del módulo será 0x0000.

*c*) Si el montador pone primero el Programa B y a continuación el A y sus datos hay que tener en cuenta que tras la última instrucción del B hay tres palabras en las que el montador habrá puesto las direcciones (relativas al comienzo del módulo) de `s1`, `s2` y `res`. Las direcciones de estas tres palabras, 0x0024, 0x0028 y 0x002C, son las que forman el diccionario de reubicación.

La primera instrucción del Programa A estará a continuación, en la dirección 0x0030, y como es la de `_start`, ésta será la dirección de comienzo que genera el montador-

Los contenidos de las direcciones 0x0024, 0x0028 y 0x002C serán las direcciones correspondientes a `s1`, `s2` y `res`, que a la salida del ensamblador eran 0x0000 y siguientes de la sección de datos. La última instrucción de A estará en la dirección 0x0030 + 0x0014 = 0x0044, por lo que esas direcciones serán 0x0045, 0x0046 y 0x0047. (Y cuando el programa se cargue a partir de una dirección habrán de reubicarse de nuevo sumándoles esa dirección).

**Nota:** Se puede comprobar (con `objdump`) que si se montan los programas objeto con `ld -o PrByA -PrB.o -PrA.o` resulta lo que se razona en el apartado *c)*. Sin embargo, si se montan con `ld -o PrAyB -PrA.o -PrB.o` los datos de A no se ponen antes sino después de B.

# Bibliografía

Para complementar o ampliar el contenido de estos apuntes hay numerosas fuentes que puede usted localizar buscando en la web. Añadimos a continuación algunas referencias a documentos impresos y a material adicional disponible en la red.

Libros recomendables para reforzar los conceptos básicos de sistemas operativos, representación y procesadores son:


- A. Prieto, A. Lloris y J.C. Torres: *Introducción a la informática,* 4ª *ed.* McGraw–Hill, Madrid, 2006.
- A. Prieto y B. Prieto: *Conceptos de Informática*. Serie Schaum, McGraw–Hill, Madrid, 2005.

  Es un texto complementario del anterior, que incluye gran cantidad de ejercicios (487), algunos resueltos (219), y 560 ejercicios de tipo test.
- A. Prieto: *Videoclases de Fundamentos de Informática*.
  `http://atc.ugr.es/APrieto_videoclases`, 2015. (Consultado el 23 de enero de 2015).

  Documentos audiovisuales que contienen 36 clases basadas en los textos anteriores.
- J. G. Brookshear: *Computer Science. An Overview, 11th. ed.* Addison-Wesley, 2011. Traducido al español con el título *Introducción a la computación*. Pearson Educación, Madrid, 2012.

  El capítulo 1 trata, esencialmente, los mismos conceptos que hemos visto aquí en los capítulos 2, 3 y 4. Los capítulos 3 (sistemas operativos) y 10 (gráficos por computadora) complementan muy bien lo que hemos visto en esta asignatura. Incluye preguntas y ejercicios resueltos y muchos problemas propuestos.
- N. Dale y J. Lewis: *Computer Science Illuminated, 4th. ed.* Jones and Barlett, 2011.

  En el capítulo 2 introduce los sistemas de numeración, y en el 3 la representación, incluyendo multimedia. En el 10 y el 11 los sistemas operativos y los sistemas de ficheros. También tiene muchos ejercicios propuestos.

Sobre sistemas operativos, un libro de texto de carácter general es:

- A. Silberstchatz, P. B. Galvin y G. Gagne: *Operating System Concepts, 9th ed.* John Wiley, 2012.

Sobre Unix son recomendables:

- A. S. Tanenbaum y A.S. Woodhull: *Operating Systems, Design and Implementation*, 3rd. ed. Prentice Hall, 2006.
  Incluye el código fuente de MINIX 3, un *microkernel* con sólo 9.000 líneas de código.

- M.J. Bach: *The Design of the Unix Operating System*. Prentice Hall, 1986.
  Es un libro clásico que sigue siendo la referencia más completa sobre el *kernel* de Unix System V.

Y específicamente sobre el funcionamiento interno de Linux:

- B. Ward: *How Linux Works*, 2nd. ed. O'Reilly, 2014.
- D.P. Bovet y M. Cesati: *Understanding the Linux Kernel*, 3rd Edition. O'Reilly, 2005.

En cuanto a los procesadores hardware, primero una referencia histórica. El documento de Burks, Goldstine y von Neumann utilizado en el capítulo 8 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 sólo por von Neumann, «First Draft of a Report on the EDVAC», que también se encuentra en la red (URL comprobados el 23 de enero de 2015):
`http://archive.org/details/firstdraftofrepo00vonn`

El libro de texto «clásico» sobre arquitectura de ordenadores, considerado la biblia de la disciplina, es:

- J.L. Hennessy y D.A. Patterson: *Computer Architecture: A Quantitative Approach, 5th ed.* Morgan Kaufmann, 2011.

En su edición impresa tiene 820 páginas y está acompañado de nueve apéndices disponibles *online* que en conjunto suman aún más páginas: `http://booksite.elsevier.com/9780123838728/`

De los mismos autores, una versión más básica es:

- D.A. Patterson y J.L. Hennessy: *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/`

Un libro específico sobre la programación de ARM (en el ensamblador de ARM, no en el de GNU) es:

- W. Hohl: *ARM Assembly Language: Fundamentals and Techniques*, CRC Press, 2009

La arquitectura Fermi, como otras de Nvidia, se describe en la web de la compañía:

```
http://www.nvidia.com/content/PDF/fermi_white_papers/
       NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
```

La especificación de la JVM está publicada en el libro:
Alex Buckley, Gilad Bracha, Frank Yellin, Tim Lindholm

- T. Lindholm, A. Buckley, G. Bracha y F. Yellin: *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley, 2014.
  También se encuentra en `http://docs.oracle.com/javase/specs/jvms/se8/html/`.