FUNDAMENTOS DE LOS SISTEMAS TELEMÁTICOS

Apuntes de clase + transparencias

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Escuela Técnica Superior de Ingenieros de Telecomunicación

Universidad Politécnica de Madrid

Fundamentos de los Sistemas Telemáticos
Tema 1: Introducción a los Sistemas Operativos

Objetivos del tema

- Identificar los elementos principales de un sistema informático
- Comprender cómo funciona un sistema informático
- Conocer las características principales de los Sistemas Operativos
- Identificar los componentes principales de los Sistemas Operativos
- Conocer los servicios que proporcionan los Sistemas Operativos

Contenidos

Material de estudio y trabajo:

- Estas transparencias
- http://es.wikipedia.org/wiki/Sistema_operativo http://en.wikipedia.org/wiki/Operating_system
- En el principio……. fue la línea de comandos:

http://biblioweb.sindominio.net/telematica/command_es/ http://garote.bdmonkeys.net/commandline/index.html

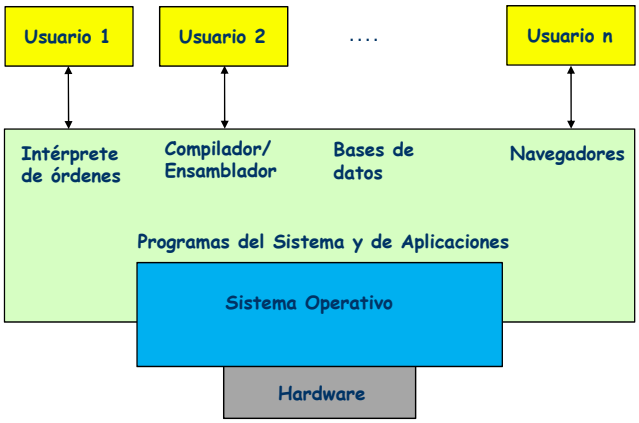
- Actividades y prácticas de laboratorio propuestas

1. Introducción a los sistemas informáticos
1.1 Estructura de un Sistema Informático

Se puede dividir en elementos:

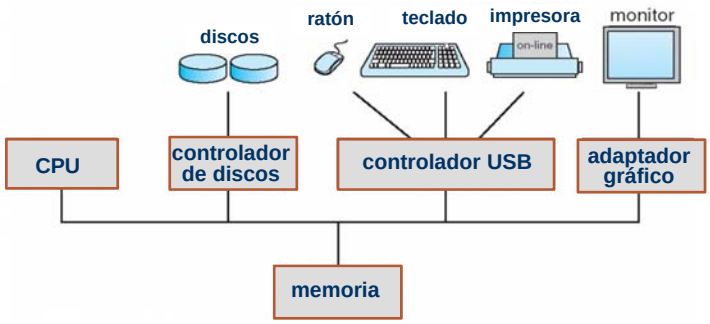
- **Hardware** (Hw) – Recursos de cómputo básicos
  - CPU (UCP), memoria, dispositivos de E/S
- **Software** (Sw)
  - **Sistema Operativo (SO) y otros programas del sistema**
    - El SO controla y coordina el uso de los recursos Hw repartiéndolos entre los programas
  - **Programas de Aplicación** – utilizan los recursos Hw y los servicios del SO para resolver problemas o proporcionar servicios a los usuarios
    - Procesadores de texto, compiladores, navegadores, juegos,…
- **Usuarios**
  - Personas, máquinas, otros sistemas informáticos

1.2 Elementos de un sistema informático



Hardware de los sistemas informáticos

- Una o más CPUs (procesadores), memoria y controladores de dispositivos, conectados por buses
- En la memoria se almacenan los programas
- Funcionamiento concurrente de las CPUs y de los dispositivos, que compiten por el uso de la memoria (aunque algunos tienen su propia memoria)



Almacenamiento

- Memoria en la CPU: pequeña, rápida
- Memoria principal (MP) RAM (Random Access Memory):
  - Conjunto de “palabras” que se identifican por una dirección única
  - Acceso aleatorio
  - Volátil
  - Los programas deben estar en MP para poderse ejecutar
  - Demasiado pequeña para contener todos los programas
- Almacenamiento secundario (MS):
  - Gran capacidad de almacenamiento no volátil: discos, memorias flash,…
  - Discos magnéticos: superficies de metal cubiertas con material magnético donde se puede grabar información
  - La mayoría de los programas se almacenan en MS hasta que se cargan en MP

Programas / Procesos

- Programa: código y datos estáticos
  - Resuelven un problema concreto o con un fin determinado
- Proceso: programa en ejecución
  - Un proceso necesita recursos: tiempo de CPU, memoria, ficheros y dispositivos de E/S
  - Los recursos se le asignan en el momento de crear el proceso o mientras se está ejecutando
- La mayoría de los SO son multiproceso
  - Con un solo proceso se desaprovecha la CPU y los dispositivos
  - Se ejecutan varios procesos “a la vez” (“concurrentemente”)
  - La multiprogramación organiza los procesos para intentar que la CPU tenga algo que ejecutar en todo momento

Programas de Sistema

Proporcionan un entorno adecuado para desarrollar y ejecutar programas

- gestión de ficheros (archivos) y directorios (carpetas)
  - crear, borrar, copiar, cambiar nombre, etc.
- información de estado
  - fecha, hora, espacio en disco, usuarios, rendimiento, historial, etc.
- modificación de ficheros
  - editores de texto, búsqueda, comparación, etc.
- procesadores de lenguajes
  - compiladores, intérpretes, montadores, depuradores, etc.
- ejecución de programas
  - carga en memoria y arranque de la ejecución
- comunicaciones
  - entre procesos, usuarios y otras máquinas

Programas de aplicación

Hacen uso de las facilidades proporcionadas por los programas del sistema para resolver problemas concretos

- Ejemplos
  - navegadores de web
  - procesadores de textos
  - hojas de cálculo
  - servidores de web
  - sistemas de gestión de bases de datos
- Algunos vienen con el sistema operativo
  - pero no necesariamente ligados a un SO
  - se suelen distribuir por separado

1.3 Arranque de un sistema informático

- Un programa inicial (*bootstrap loader*) se ejecuta cuando se arranca el sistema informático
- Se suele almacenar en ROM (*firmware)*
- Inicia adecuadamente todos los elementos del sistema
- Carga en la MP el SO, éste comienza su ejecución y espera a que se produzca algún suceso:
  - Interrupción hardware: teclado, impresora, disco,..
  - Interrupción software: programa que quiere sacar información por pantalla, programa que quiere leer un fichero de disco,…

Contenidos

1. Introducción a los sistemas informáticos

1. Estructura de un sistema informático
2. Elementos de un sistema informático
3. Arranque de un sistema informático

2. Introducción a los sistemas operativos

1. Funciones y tipos de sistemas operativos
2. Componentes del sistema operativo. Núcleo
3. Gestión de almacenamiento secundario. Ficheros y directorios
4. Protección y seguridad
5. Interfaz al sistema operativo

2 Introducción a los Sistemas Operativos

- ¿Qué es un sistema operativo (SO)?
  - "programa/s" (software) que:
    - Administra el hardware de un sistema informático
    - Proporciona un entorno para ejecutar los programas de aplicación
    - Actúa como intermediario entre los usuarios y el hardware de un sistema informático
- Objetivos del sistema operativo
  - Facilitar el uso del sistema informático: abstracciones adecuadas
    - Para usuarios y desarrolladores de aplicaciones
    - Se adapta al tipo de aplicaciones a ejecutar y al uso
    - Oculta complejidad del hardware (dispositivos muy variados y complejos)
  - Utilizar el sistema informático eficientemente
    - Gestión de recursos: procesos, memoria, E/S, ficheros, ..
    - Control de intentos de acceso simultáneos

El SO desde el punto de vista del usuario

- PC independiente:
  - Fácil de usar
  - Rendimiento para un solo usuario
- Mainframe con terminales:
  - Maximizar la utilización de recursos (UCP, memoria, E/S,..)
  - Reparto equitativo
- Estación de trabajo y servidores conectados mediante redes (recursos dedicados y compartidos)
  - Compromiso entre los dos tipos anteriores
- Sistemas informáticos de mano (móviles)
  - Fáciles de usar
  - Rendimiento en base a sus limitaciones

El SO desde el punto de vista del sistema informático

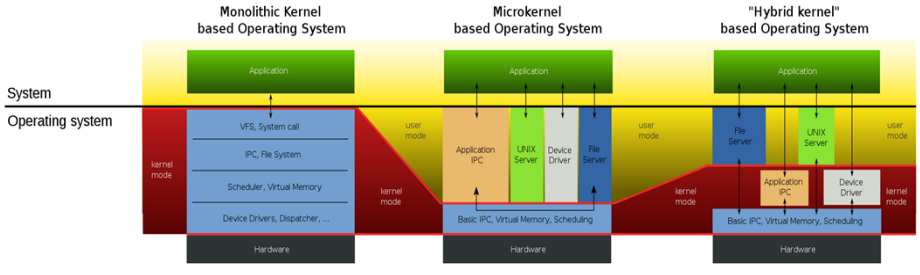
- Asigna y controla los recursos (CPU, memoria, E/S,..)
  - Cuando hay peticiones en conflicto:
    - decide quién usa el recurso
    - aplica criterios de equidad y eficiencia
  - Controla la ejecución de los programas para prevenir errores o uso inadecuado del computador
- Mecanismos de gestión de recursos dependientes del tipo de sistema:
  - Interactivos: optimizar el tiempo de respuesta, da prioridad a procesos con mucha E/S
  - Lotes: maximizar el uso del procesador, sin importar el tiempo de respuesta
  - Tiempo real: prioridad a procesos más urgentes

2.2. Componentes del Sistema Operativo

- El SO se puede dividir desde un punto de vista funcional:
  - Gestor de Procesos
  - Gestor de Memoria
  - Gestor de Almacenamiento Secundario
    - Sistemas de ficheros
  - Gestor de Entrada Salida
  - Protección y Seguridad

Núcleo (kernel) del sistema operativo

- Procesador con dos modos: usuario / supervisor
- Nucleo (kernel):
  - Sw que implementa servicios básicos del SO
  - Se ejecuta en modo supervisor (“modo kernel”)



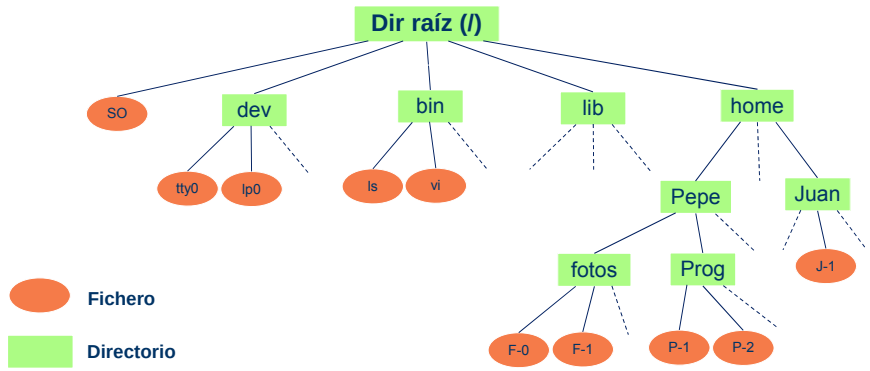
Linux /Android Mach / Hurd Windows / Mac OS X

2.3 Gestión de Almacenamiento secundario

- Proporciona una visión uniforme y virtual del almacenamiento de información
  - Cada dispositivo tiene características diferentes
  - Fichero: colección de información relacionada. Es una abstracción de almacenamiento de información que oculta los detalles de los dispositivos físicos
- Sistema de ficheros
  - Los ficheros se suelen organizar en directorios
  - Los directorios son a su vez ficheros con información de los ficheros que cuelgan de ellos
  - Control de acceso a la información (quién y en qué forma)
  - Actividades del gestor de ficheros del SO:
    - Creación y borrado de ficheros y directorios
    - Operaciones para manipular ficheros y directorios
    - Almacenar los ficheros en los dispositivos
    - Copias de seguridad en dispositivos no volátiles

Estructura de directorios

- El sistema de ficheros es jerárquico
  - Se parte de un directorio raíz (/)
  - El resto se monta en alguna de las ramas



2.4 Protección y Seguridad

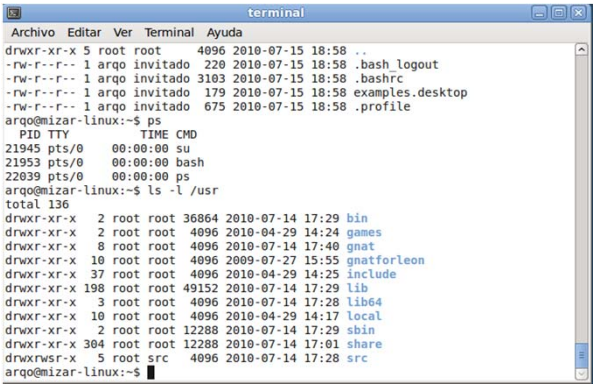
- Protección: mecanismos para controlar el acceso de procesos o usuarios a recursos del sistema
- Seguridad: defensa del sistema contra ataques internos y externos
  - Denegación de servicio, virus, gusanos, robo de identidad, robo de servicios, etc.
- Se suele usar la identidad del usuario para determinar qué puede hacer
  - Cada usuario tiene un identificador (ID)
  - Los usuarios pertenecen a un grupo o más grupos. Cada grupo tiene un identificador
  - Se asigna el ID a cada fichero y proceso del usuario para determinar las operaciones permitidas

2.5 Interfaz al sistema operativo

- Llamadas al sistema:
  - Interfaz de programación: invoca al SO para realizar operaciones
  - Ejemplo:
    - Procesos: crear, matar, esperar por un proceso, comunicación entre procesos.
    - Ficheros: abrir, cerrar, leer, posicionarse.
- Interfaces de usuario:
  - Permiten a los usuarios interactuar con el sistema:
    - Acceder, reproducir o modificar ficheros; ejecutar programas
  - Ordenes: Intérprete de órdenes (“comandos”)
  - Gráfica
  - No son propiamente parte del SO
  - Emplean llamadas al sistema para realizar sus funciones

Interfaz de usuario de órdenes

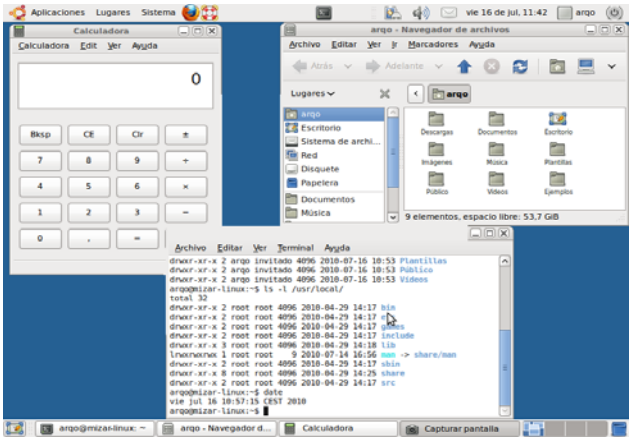
- Proporciona acceso al sistema operativo mediante órdenes (*commands*) que se introducen en forma de texto
- Para ello se ejecuta un intérprete de órdenes
  - shell (unix y derivados): sh, bash, ksh, csh, ..
  - command prompt (Windows): cmd.exe
- El intérprete carga programas que ejecutan las órdenes



Ejemplo de *bash* en GNU/Linux

Interfaz de usuario gráfica (GUI)

- Paradigma de escritorio
  - ventanas, menús, ratón
- Más fácil de usar para usuarios normales
- Pero las interfaces textuales tienen ventajas para administrar el sistema
  - acceso a detalles de configuración
  - *scripts* de *shell*



http://biblioweb.sindominio.net/telematica/command_es/ http://garote.bdmonkeys.net/commandline/index.html

Fundamentos de los Sistemas Telemáticos

Curso 2014-2015

Laboratorio 1

Comandos básicos para el manejo de ficheros

Entrega obligatoria al final de la sesión de laboratorio

Objetivos:

- Familiarizarse con la línea de comandos (órdenes) de Unix.
- Conocer los comandos básicos para trabajar con ficheros.
- Saber utilizar un editor de texto.
- Aprender a consultar las páginas del manual.

Entorno:

Uno de los ordenadores del laboratorio. Pero las actividades que se describen pueden realizarse en cualquier equipo con un sistema de tipo Unix.

Actividades previas:

Antes de la sesión de laboratorio que corresponda a su grupo deberá usted leer este documento e intentar realizar las actividades que en él se describen (excepto el apartado 10), bien en un ordenador del mismo laboratorio (en horario libre) o en cualquier otro ordenador. De no hacerlo, es posible que no le dé tiempo a realizar la entrega. De este modo, la sesión de laboratorio tiene la función de una clase de tutoría para resolver las dudas que se le hayan podido presentar.

Resultados: Al final de la sesión de laboratorio, y desde su puesto de trabajo, deberá subir al Moodle un fichero de texto conteniendo todas las órdenes que ha utilizado, cómo se indica al final de este documento (apartado 10). No se podrá subir dicho fichero después de la hora de terminación del laboratorio de su grupo.

1. Inicio

Como ya ha comprobado al realizar la actividad A0, 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 estas actividades desde un terminal de texto "puro". Para ello:

1. pulse *simultáneamente* tres teclas: `Ctrl, Alt y F1`[1]. La pantalla se

[1] 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 este caso, pulse Alt y F2 para cambiar a la siguiente consola.

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:

   ```
   sulogin$
   ```

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" (también conocida como "Ret" o "Retorno") tras la `c` (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.

Para que tenga clasificados todos los documentos que va a generar en cada práctica, debe crear un directorio para cada práctica. En el segundo laboratorio aprenderá a crear y gestionar directorios. Para la realización de esta primera práctica teclee:


```
mkdir lab1
```

```
cd lab1
```

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`").

2. 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 "–", y pulsar la tecla "`Intro`" (también conocida como "`Ret`" o "`retorno`") tras terminar de escribir la última letra de la orden (todas las órdenes se terminan con esta tecla)):

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`
o bien:
`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).

3. 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 "retorno") y salga del editor(^X).

 `ls -l`

Verá que el fichero `prueba` tiene cinco bytes[2]. Esto es porque al guardarlo se ha añadido un carácter invisible: el "retorno" pero si usted ha pulsado la tecla "retorno" 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:

 `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`

4. 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`

[2] 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

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 retrocecer) 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 Laboratorio 3 veremos cómo esa salida puede redirigirse a otro fichero en lugar de la pantalla.

5. 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 sobreescribe.

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 anterior para cambiar el nombre de `prueba` (`mv prueba pp`) se podría haber conseguido 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`.

6. 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 (`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
  ```

7. 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*`

8. 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[3].

 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 "↓"): con pulsaciones sucesivas de "↑" se le irán mostrando la última orden realizada, la anterior, etc. (y con "↓"se moverá en sentido inverso). Puede confirmar la repetición de una orden con "retorno", y si es necesario, editar antes sus opciones o parámetros (con las teclas "←","→" y "Supr") .

[3] Si quiere hacerlo ya, escriba `history | less`. El símbolo "|" se obtiene pulsando `AltGr` y `1`.

9. Las páginas del 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 del manual, a veces muy extensas: escriba "`/size`" y "`retorno`" y verá que le lleva a la primera aparición de `size`[4]. 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.

10. Entrega de resultados

Escriba esto:

```
history –w Grupo-Apell1-Apell2-L1.txt
```

Esto hará que la historia se grabe en el fichero `Grupo-Apell1-Apell2-L1.txt`. compruébelo con:

```
less Grupo-Apell1-Apell2-L1.txt
```

(Naturalmente, `Grupo`, `Apell1` y `Apell2` deben ser los que le correspondan, por ejemplo, `G12-Perez-Gomez-L1.txt`. Recuerde que, por una limitación del Moodle, no se pueden utilizar caracteres con tilde en los nombres de los ficheros a subir.)

Un truco: cuando haya escrito los dos o tres primeros caracteres del nombre del fichero (por ejemplo, `G12`) 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., como se indica en la nota de la primera página).
Compruebe en el Moodle que se ha grabado el fichero y el contenido del mismo (por ejemplo con `less`).

[4] Esto, suponiendo que su sistema tiene las páginas de manual en inglés. Si las tiene en español, escriba `tamaño`.

11. Resumen

Orden	**Uso**
ls	Lista nombre de ficheros y directorios: Algunas opciones: - l: listado largo - a: incluye ficheros ocultos
nano [<fichero>]	Editor de texto
wc <fichero>	Cuenta de líneas, de palabras y de bytes Algunas opciones: - w: Cuenta palabras; - l: Cuenta líneas - 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 Algunas opciones: - i: si <f2> existe, pregunta antes de sobreescribirlo - 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 - i: pide confirmación antes de borrar
grep <cad> <f1>	Escribe las líneas de <f1> que contienen la cadena de caracteres <cad> Algunas opciones: - i: Ignora la caja (mayúsculas = minúsculas) - v: Escribe las líneas que no contienen la cadena - n: Escribe también los números de las líneas p
man <orden>	Página de manual de la orden Algunas opciones: - a: Si hay varias páginas para la orden las presenta secuencialmente - 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.

ANEXO: Sobre la línea de comandos

El texto que sigue está tomado de http://translate.flossmanuals.net/CommandLineIntro_es y es traducción de parte de un documento publicado en 2009 por la Free Software Foundation con el título "Command Line" que contiene ilustraciones y está disponible en páginas web y en PDF en http://en.flossmanuals.net/command-line.

Otros documentos de lectura recomendada son los referenciados en la última transparencia del Tema 1.

Ventajas de utilizar comandos

Muchas personas que le dan una oportunidad a la línea de comandos quedan tan sorprendidas de sus posibilidades que no desean regresar a la interfaz gráfica de usuario. ¿Porqué? Bien, en resumen, las principales ventajas que ofrece la línea de comandos sobre los programas gráficos comunes son:

- Flexibilidad. Con los programas gráficos algunas veces se alcanza un límite, al encontrar que no puede hacer lo que desea, o tendrá que encontrar formas incómodas de trabajar considerando los límites del programa. No obstante, trabajando en la línea de comandos, usted puede combinar instrucciones para producir un intervalo virtualmente infinito de funciones nuevas e interesantes. Al combinar instrucciones de manera creativa, usted puede lograr que la línea de comandos realice exactamente lo que usted desea, lo pone al control de su computadora.

- Confiabilidad. Los programas gráficos a menudo son inmaduros o incluso inestables. En contraste, la mayoría de las herramientas que ofrece la línea de comandos son altamente confiables. Una de las razones de esto es su madurez: los programas de línea de comandos más antiguos han estado ahí desde finales de los años 1970, lo que significa que han sido evaluados durante tres décadas. También tienden a trabajar de la misma manera a través de diferentes sistemas operativos, a diferencia de la mayoría de las interfaces gráficas. Si usted desea una navaja suiza en la que pueda confiar, la línea de comandos es para usted.

- Rapidez. Las imágenes de las interfaces gráficas consumen gran cantidad de los recursos de hardware, a menudo resultando en lentitud o inestabilidad. La línea de comandos, por otra parte, utiliza los recursos de cómputo mucho más escasamente, liberando a la memoria y potencia de procesamiento para las tareas que usted desea llevar a cabo. La línea de comandos es también intrínsecamente más rápida: en lugar de dar clicks a través de largas cadenas de menús gráficos, usted puede escribir una docena o algo similar de golpes de teclado, y a menudo aplicarlos a múltiples archivos o otros objetos. Si usted es un mecanógrafo rápido, esto le habilitará a incrementar su productividad drásticamente.

- Experiencia. El utilizar la línea de comandos es una grandiosa experiencia de aprendizaje. Cuando usted utiliza la línea de comandos, se comunica con su computadora más directamente que con los programas gráficos, aprende así mucho sobre la forma como trabaja internamente la computadora: el utilizar la línea de comandos de manera natural es el camino para convertirse en un gurú de GNU/Linux.

- Diversión. ¿Alguna vez ha deseado ser como aquellos hackers de la computadora que pueden lograr que las máquinas GNU/Linux hagan cosas que usted ni siquiera ha soñado? La mayoría de ellos lo hacen utilizando la línea de comandos. Una vez que haya aprendido cómo utilizar esta poderosa herramienta, usted se verá a sí mismo haciendo todas esas cosas tan divertidas e interesantes que ni siquiera ha pensado que sean posibles.

El valor de las instrucciones de comandos

Pero espere, ¡aún hay más!. Usted puede incluso almacenar instrucciones en archivos de texto. Estos archivos de texto son llamados scripts y pueden utilizarse en lugar de teclear una larga serie de instrucciones cada vez. Por ejemplo, si usted almacena instrucciones en un archivo llamado `mycommand.sh`, no tiene que volverá a escribir las instrucciones, sino simplemente teclear:

```
mycommand.sh
```

Adicionalmente se pueden combinar instrucciones de manera simple o sofisticada. Más aún, usted puede programar scripts para que ocurran en un tiempo o fecha específicos o con un evento específico en su computadora.

También puede escribir scripts para que acepten información adicional del usuario. Por ejemplo, un script de redimensionamiento de imagen puede preguntarle al inicio el tamaño al que hay que redimensionar la imagen.

¿Alguna vez intentó hacer algo remotamente parecido a eso utilizando una interfaz gráfica de usuario?

Tal vez ahora pueda ver cómo el trabajo con la interfaz de línea de comandos comienza a abrir todo un mundo nuevo de utilidad a su computadora.

¿Está enferma mi computadora?

Otro uso de la línea de comandos consiste en revisar el buen estado de su computadora. Existen muchas instrucciones que usted puede usar para revisar cada aspecto de la salud de su computadora, desde la cantidad de espacio disponible en el disco duro hasta la temperatura de la unidad central de proceso (CPU). Si el desempeño de su computadora es pobre y usted no sabe qué es lo que está pasando, unas cuantas instrucciones lo ayudarán a determinar rápidamente si se trata de un asunto de Hardware o Software, y le ayudará a efectuar una reparación rápida.

Atravesando la red

Existe otra característica importante de la línea de comandos que las interfaces gráficas no pueden igualar: la interacción sobre la red. Imagine que tiene una computadora en otra habitación y desea apagarla. ¿Cómo puede hacer eso? Fácil ¿No? Levántese, camine hacia la computadora y haga click sobre el botón apagar.

Bien, aquellos que tienen conocimiento de la materia pueden conectarse a la computadora de la habitación contigua por medio de la línea de comandos e introducir `halt`.

Esto puede parecer trivial. Tal vez sea mejor para usted levantarse de esa silla confortable y gastar 5 calorías caminando a la siguiente habitación. Sin embargo, ¿qué tal que la computadora que desea apagar está en otro suburbio? ¿En otra ciudad? ¿Otro país? El control remoto de esa computadora podrá ser entonces muy útil.

Apagar una computadora remota es sólo un comienzo. Cualquier cosa que usted pueda hacer en la línea de comandos la puede hacer en la computadora remota. Eso significa que puede correr scripts, ejecutar instrucciones, editar archivos de texto, revisar los diagnósticos, y llevar a cabo muchas otras tareas. El mundo de la línea de comandos ha crecido bastante.

Incluso los programas gráficos son comandos

Cuando usted hace click sobre un icono o menú de opciones para abrir un programa, usted está ejecutando una instrucción. Encontrará ocasiones en las que es útil saber cuál es la instrucción o comando que se está ejecutando. Por ejemplo, si sospecha que un programa está ejecutándose de manera invisible en segundo plano y está disminuyendo la velocidad de su computadora, usted puede encontrar la instrucción de dicho programa y terminar el mismo. Los programas de interfaz de usuario gráfica conocidos como GUI, a menudo, envían más mensajes de error a la interfaz de línea de comandos (CLI), los cuales son mostrados en los cuadros de diálogo. Siendo esto de gran utilidad para diagnosticar problemas.

Fundamentos de los Sistemas Telemáticos

Curso 2014-2015

Laboratorio 2

Comandós básicos para la gestión de ficheros y directorios

Entrega obligatoria al final de la sesión de laboratorio

Objetivos:

- Conocer los comandos básicos de Unix para la gestión de ficheros y directorios
- Conocer los comandos que permiten visualizar y gestionar la estructura de directorios de Unix

Entorno:

Uno de los ordenadores del laboratorio. Pero las actividades que se describen pueden realizarse en cualquier equipo con un sistema de tipo Unix.

Actividades previas:

Antes de la sesión de laboratorio que corresponda a su grupo deberá usted leer este documento e intentar realizar las actividades que en él se describen (excepto subir al Moodle el fichero generado), bien en un ordenador del mismo laboratorio (en horario libre) o en cualquier otro ordenador. De no hacerlo, es posible que no le dé tiempo a realizar la entrega. De este modo, la sesión de laboratorio tiene la función de una clase de tutoría para resolver las dudas que se le hayan podido presentar.

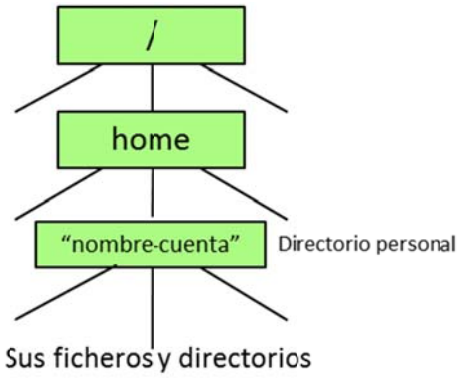
Resultados: Al final de la sesión de laboratorio, y desde su puesto de trabajo, deberá subir al Moodle un fichero de texto conteniendo todas las órdenes que ha utilizado para realizar el ejercicio que se propone en el séptimo apartado. No se podrá subir dicho fichero después de la hora de terminación del laboratorio de su grupo.

Abra un terminal de texto «puro» tal como hizo en el primer laboratorio. Para ello pulse simultáneamente tres teclas: `Ctrl, Alt y F1`. La pantalla se convertirá en una consola de texto, de 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`.

En lo sucesivo, el icono  significa que debe usted escribir la orden indicada.

1. Directorio home

El árbol del sistema de ficheros parte del directorio raíz (/). De dicho directorio cuelgan a su vez varios subdirectorios. Uno de ellos es el directorio `home`, de donde cuelga a su vez los directorios personales de los usuarios. El nombre de su directorio personal es `nombre-cuenta` y de ese segundo directorio es de donde cuelgan todos sus ficheros y directorios.



El nombre del directorio personal es diferente para cada usuario, se corresponde con el nombre de usuario asignado a cada uno, `nombre-cuenta`, por ejemplo `juan.garcía`.

Denominaremos "directorio de trabajo actual" al directorio en el árbol del sistema de ficheros, donde se encuentra situado en un instante determinado. Todas las órdenes que ejecute lo harán respecto al directorio de trabajo actual.

Cuando se conecta a su cuenta en el ordenador, el sistema sitúa su directorio de trabajo actual en su directorio personal.

Para descubrir qué hay en su directorio personal teclee:

```
ls
```

Observará que aparece el directorio lab1 de la práctica anterior.

2. Creación de directorios

Para crear subdirectorios se utiliza la orden `mkdir`:

```
mkdir laboratorio2
```

Crea el subdirectorio `laboratorio2` que cuelga del directorio de trabajo actual. El directorio de trabajo actual, que en este momento, si no ha ejecutado ninguna orden más, es su directorio personal. Por tanto se crea el directorio `laboratorio2` que cuelga de su directorio personal.

Compruébelo tecleando:

```
ls -l
```

3. Cambiarse 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 `laboratorio2`, cambiando por tanto su posición actual en el árbol del sistema de ficheros:

 `cd laboratorio2`

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 donde 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.

Tecleando "`cd /`" le lleva al directorio raíz.

4. Rutas

Para identificar un fichero o directorio debe dar la ruta o camino de dicho 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, para identificar el directorio `laboratorio2` que ha creado previamente, se puede hacer mediante su ruta absoluta: `/home/nombre-cuenta/laboratorio2` o mediante su ruta relativa, si por ejemplo su directorio de trabajo actual es su directorio personal sería `laboratorio2` o si su directorio de trabajo actual es `home` sería `nombre-cuenta/laboratorio2`.

Mediante la orden `pwd` podrá 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.

5. Entendiendo las rutas

Se puede referir a su directorio personal con el carácter "~"[1]. Para cambiarse al directorio `laboratorio2`, que ha creado previamente, independientemente desde donde se encuentre actualmente en el árbol del sistema de ficheros, teclee:

```
cd   ~/laboratorio2
```

Cree un fichero que se llame `prueba`:

```
nano prueba
```

Escriba una línea con un texto cualquiera en dicho fichero `prueba` y salga del editor.

Teclee:

```
cd ..
ls -l
ls -l prueba
```

Mediante estás órdenes se ha cambiado al directorio padre, que es su directorio personal, se ha listado los nombres de ficheros y directorios que se encuentran en dicho directorio y por último se ha ordenado generar un listado largo del fichero `prueba`.

Como resultado de ésa última orden ha 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, si no que cuelga del directorio `laboratorio2.`

Para identificar el fichero o directorio que va a utilizar una orden, por ejemplo la orden `ls –l` sobre el fichero `prueba`, deberá dar la ruta absoluta o relativa de dicho fichero o directorio. La ruta relativa dependerá del directorio actual de trabajo. Por ejemplo:

- Que el fichero o directorio se encuentre en el directorio actual de trabajo. Para ello copie primero el fichero `prueba` a su directorio actual de trabajo y cámbiele de nombre a `prueba1`

  ```
  cp laboratorio2/prueba  prueba1
  ls -l prueba1
  ```

- Se puede previamente cambiar el directorio actual de trabajo al directorio donde se encuentre el fichero `prueba`

  ```
  cd  laboratorio2
  ls -l  prueba
  ```

- Se puede utilizar una ruta relativa para identificar el fichero o directorio desde el

[1] cd es equivalente a cd ~ y a cd ~/

directorio de trabajo actual. Teclee `cd` para el directorio de trabajo actual vuelva a ser su directorio personal

```
ls -l laboratorio2/prueba
```

- Se puede utilizar la ruta absoluta para identificar al fichero o directorio desde el directorio raiz, independientemente de donde esté situado su directorio actual de trabajo. Suponiendo que su directorio de trabajo actual es cualquiera, por ejemplo teclee: `cd ~/laboratorio2`

```
ls -l /home/nombre-cuenta/laboratorio2/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.

6. Borrar directorios

Con la orden `rmdir` se borran directorios. En concreto si se teclea `rmdir nombre-directorio`, se borrara dicho directorio pero únicamente si está vacío.

Si `rm` se utiliza con la opción `–r` borra además del directorio todos los ficheros y directorios que cuelgan de él, de forma recursiva. Esta opción es muy peligrosa ya que puede borrar mucha información no deseada 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 a borrar la información.

Sitúese en su directorio personal y borre el directorio `laboratorio2` y los ficheros que cuelgan de él.

```
cd
rm –ri laboratorio2
```

7. Ejercicio a realizar

a) Si se encuentra en la sesión correspondiente del laboratorio, donde va a generar el fichero para subir al moodle, borre antes todos los ficheros y directorios que ha podido crear previamente, con las ordenes `rmdir` y `rm`

b) Borre la historia de las órdenes que ha tecleado previamente:

```
history –c
```

De esta forma a partir de este momento va generando un historial de las órdenes que va a teclear y que luego deberá subir al Moodle, tal como se le indica al final de este apartado.

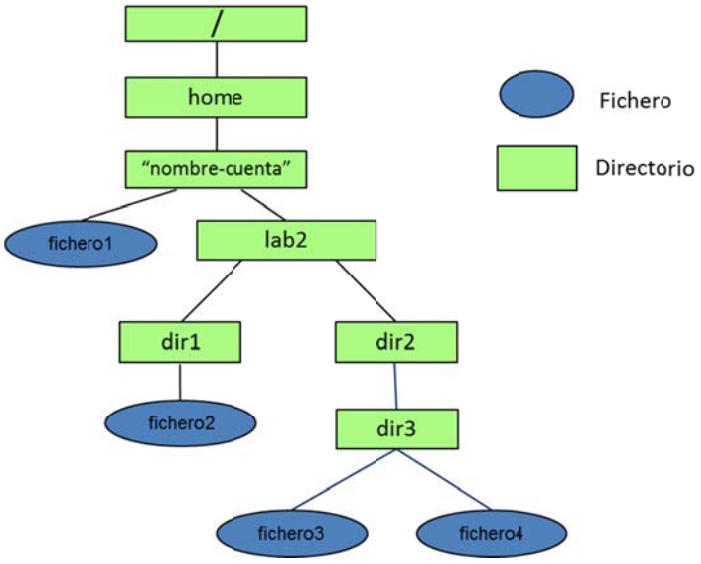
Si en algún momento se equivoca al teclear o en las órdenes que debe ejecutar, no se preocupe y repita las órdenes. No es necesario que borre todo el historial y repita de nuevo el ejercicio, se consideraran válidas las últimas órdenes ejecutadas de cada apartado.

c) Escriba su nombre y apellidos, para que quede registrada en la historia, aunque la Shell le contestará que no existe ninguna orden con ese nombre:

```
Nombre Apell1 Apell2
```

d) Cree la siguiente estructura de directorios y ficheros. Cree los directorios con la orden `mkdir`. Con el editor `nano`, que utilizó en el laboratorio 1, cree los ficheros teniendo en cuenta que cada uno de los ficheros deberá tener el siguiente contenido:

- Su nombre y dos apellidos
- Su grupo de FTEL
- Nombre del fichero



Cambie su directorio de trabajo actual a su directorio personal y compruebe que ha creado la estructura anterior con la orden:

```
tree
```

Observe que además de los ficheros y directorios creados previamente verá un directorio `escritorio` con los enlaces de su entorno gráfico.

e) **Sitúese en su directorio personal y utilizando rutas absolutas** para identificar a los directorios, ejecute las órdenes necesarias que le permitan:

1. Moverse al directorio `dir3`
2. Ir al directorio `dir1`
3. Volver a su directorio personal

f) **Sitúese en su directorio personal y utilizando rutas relativas** para identificar a los directorios, ejecute las órdenes necesarias que le permitan:

1. Moverse al directorio dir2
2. Ir al directorio dir1
3. Ir al directorio dir3
4. Volver a su directorio personal

g) **Sitúese en su directorio personal, sin moverse de dicho directorio y utilizando rutas absolutas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias (utilice parte de las órdenes que se le enseñaron en el laboratorio1) que le permitan:

1. Sacar por pantalla el contenido del `fichero1(`no utilice un editor)
2. Editar el `fichero4` y añadir el nombre de esta asignatura
3. Cambiar el nombre del `fichero3` a `fichero5`

h) **Sitúese en su directorio personal, sin moverse de dicho directorio y utilizando rutas relativas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias (utilice parte de las órdenes que se le enseñaron en el laboratorio1) que le permitan:

1. Ver un listado largo con las características (nombre, permisos, tamaño,…) de los ficheros que hay en el directorio `dir3`
2. Editar el `fichero4` y añadir la fecha actual
3. Sacar por pantalla el contenido del `fichero2(`no utilice un editor)
4. Cambiar el nombre del `fichero4` a `fichero6`

i) Cambie su directorio de trabajo actual a su directorio personal y compruebe con la orden `tree` que los cambios de nombre de los ficheros se han realizado correctamente.

j) **Sitúese en el directorio `dir1`, sin moverse de dicho directorio y utilizando rutas absolutas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias que le permitan:

1. Crear un nuevo fichero llamado `fichero7` que cuelgue del directorio `dir2` y cuyo contenido sea el mismo que el del fichero `fichero1(`no utilice un editor)
2. Borrar todos los ficheros cuyo nombre comience por `fich` y se encuentren en el directorio `dir3`

k) **Sitúese en el directorio `dir1`, sin moverse de dicho directorio y utilizando rutas relativas** para identificar a los ficheros y directorios, ejecute las órdenes necesarias que le permitan:

1. Crear un nuevo fichero llamado `fichero8` que cuelgue del directorio `dir3` y cuyo contenido sea el mismo que el del fichero `fichero1(`no utilice un editor)
2. Borrar todos los ficheros cuyo nombre comience por `fich` y se encuentren en el directorio `dir2`

l) 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 el fichero `Grupo-Apell1-Apell2_L2.txt`, donde Grupo, Apell1, Apell2 deben ser los que le corresponden (recuerde no utilizar en el nombre del fichero tildes) para ello teclee:

```
history  –w  Grupo-Apell1-Apell2-L2.txt
```

Compruebe con `less` el contenido del fichero generado

No borre los directorios que ha creado ya que los utilizará en el siguiente laboratorio.

m) Suba al Moodle el fichero generado, para ello lo más sencillo es que vuelva al entorno gráfico (`Alt+F7`) y lo suba desde el navegador `Firefox`. Pero antes de volver al entorno gráfico cierre su sesión abierta en este terminal de texto con:

```
logout
```

n) Compruebe en el Moodle que se ha grabado el fichero y el contenido del mismo

o) Para que le quede claro 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.

8. 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	Saca la ruta absoluta del directorio de trabajo actual
rmdir <dir>	Borra un directorio vacío
rm <dir>	Opción: -r: borrar de forma recursiva todos los ficheros y directorios que cuelgan del directorio a borrar Opción: -i: pide confirmación antes de borrar
tree	Lista los ficheros, directorios y contenidos de los directorios recursivamente en forma de árbol

Fundamentos de los Sistemas Telemáticos

Curso 2014-2015

Laboratorio 3

Comandos básicos para redirigir la E/S, gestionar la seguridad de los ficheros y directorios y gestionar los procesos

Entrega obligatoria al final de la sesión de laboratorio

Objetivos:

- Conocer los comandos básicos de Unix para redirigir la entrada y salida y encadenar órdenes
- Conocer los comandos que permiten gestionar los permisos de los ficheros y directorios
- Conocer los comandos que permiten gestionar los procesos

Entorno:

Uno de los ordenadores del laboratorio. Pero las actividades que se describen pueden realizarse en cualquier equipo con un sistema de tipo Unix.

Actividades previas:

Antes de la sesión de laboratorio que corresponda a su grupo deberá usted leer este documento e intentar realizar las actividades que en él se describen (excepto subir al moodle el fichero generado), bien en un ordenador del mismo laboratorio (en horario libre) o en cualquier otro ordenador. De no hacerlo, es posible que no le dé tiempo a realizar la entrega. De este modo, la sesión de laboratorio tiene la función de una clase de tutoría para resolver las dudas que se le hayan podido presentar.

Resultados: Al final de la sesión de laboratorio, y desde su puesto de trabajo, deberá subir al Moodle un fichero, utilizando la plantilla `FTEL-Tema1-actividades-Lab3-2014`, con las respuestas a las actividades que se le indican a partir del punto f del sexto apartado, incluida la pregunta que el profesor le indique al comienzo de la sesión de laboratorio. No se podrá subir dicho fichero después de la hora de terminación del laboratorio de su grupo.

Abra un terminal de texto desde el entorno gráfico (`gnome`) con los menús de la barra superior: `"Aplicaciones/Accesorios/Terminal"`

En lo sucesivo, el icono  significa que debe usted escribir la orden indicada.

Utilizando los conocimientos que adquirió en la práctica anterior, cree un directorio nuevo denominado `lab3` que cuelgue de su directorio personal y cámbiese a dicho directorio `lab3`.

1. Redirección

Muchos de los procesos iniciados al ejecutar una orden UNIX 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 la pantalla es tanto la salida estándar cómo la salida de errores.

En el primer laboratorio usó la orden `cat prueba` para ver el contenido del fichero `prueba` en la pantalla. `cat` por tanto leía los datos del fichero y los sacaba por la salida estándar, es decir la pantalla. Si únicamente pulsa `cat` sin indicar un nombre de fichero, la orden `cat` lee los datos de la entrada estándar que es el teclado, y los sigue sacando por la pantalla. Pruebe a teclear:

 `cat`

A continuación teclee palabras en el teclado, por ejemplo nombres de frutas, todas ellas separadas por `Intro`, es decir cada una en una línea. Finalmente teclee `ctrl D` (`^D`) (fin de fichero) para indicar que se termina la entrada de texto por teclado. Observe que según introduzca un palabra seguida de `Intro`, dicha palabra se sacará inmediatamente por la pantalla, viéndolas por tanto duplicadas.

1.1 Redirección de la salida

En Unix se puede redirigir tanto la entrada como la salida de las órdenes.

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 va introduciendo por el teclado, teclee:


```
cat  > frutas
Pera
Manzana
Plátano
Melocotón
^D
```

Dicha 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, previamente crea dicho fichero. Si `frutas` existe, antes de redirigir la salida a dicho fichero, borra su contenido.

Para comprobar el contenido del fichero `frutas` teclee:

 `cat  frutas`

Podemos ordenar las líneas de un fichero de texto alfabéticamente mediante la orden `sort` (la orden `sort` ordena alfabéticamente o numéricamente una lista). Por

ejemplo, `sort frutas` muestra por pantalla todas las líneas de **f**rutas ordenadas alfabéticamente.

 `sort frutas`

Si queremos que el resultado se guarde en otro fichero utilizaremos la opción `–o` seguida del nombre del fichero:

 `sort frutas –o frutas2`

Para ver otro ejemplo de redirección de la salida, teclee:

 `sort > verduras`
`Puerro`
`Coliflor`
`Lechuga`
`^D`

Dicha orden lee los datos de la entrada estándar, el teclado, una vez ordenados los nombres de las verduras como la salida está redirigida (“>”) al fichero `verduras`, los guarda ya ordenados en el fichero `verduras`.

Si ahora queremos añadir más frutas al fichero `frutas` ejecutaremos la orden `cat > frutas`, las nuevas frutas que escribiera en el teclado se escribirán en el fichero `frutas`, pero perderíamos las que habíamos introducido con la primera orden. Para poder añadir información a un fichero sin perder los datos que ya tiene dicho fichero, utilizamos “>>”.

 `cat >> frutas`
`Cereza`
`Melón`
`Sandía`
`^D`

Con esta orden, hemos añadido los nuevos nombres de frutas al fichero `frutas`. Es decir, se han seguido leyendo los datos de la entrada estándar, el teclado, pero la salida estándar se ha redirigido al fichero sin borrar su contenido previo.

Compruébelo:

 `cat frutas`

Ahora ya tenemos dos ficheros, `frutas` con 7 frutas y `verduras` con 3 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 estamos haciendo es introducir a la orden `sort` el contenido de dos ficheros, `frutas` y `verduras`. Dicha orden ordena los datos y la salida es redirecionada con "`>`" al fichero `comida`. Compruébelo:

```
cat  comida
```

1.2 Redirección de la entrada

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, usando redirección de la salida, 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 (con el símbolo "`<`" se hace redirección de la entrada estándar) se cuenta el número de palabras del fichero `nombres.`

El resultado de ejecutar ambas órdenes permite obtener por tanto el número de ficheros y directorios que hay en su directorio de trabajo actual. Tenga en cuenta que la primera orden creará primero el fichero `nombres` y luego ejecuta la orden `ls`, con lo cuál luego cuando ejecute la orden `wc`, contará también dicho fichero `nombres.`

2. Tuberías

Pero ejecutar estás dos órdenes seguidas hace que el proceso sea más lento y que cuando hayamos terminado nos tengamos que acordar de borrar el fichero de trabajo, `nombres`, que hemos creado y ya no lo necesitamos.

Para evitar estos problemas se pueden utilizar las tuberías (`|`). Las tuberías conectan directamente la salida de una orden a la entrada de otra orden.

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.

3. Permisos de acceso en el sistema de ficheros

Cuando ha tecleado:

 `ls -l`

habrá observado que para cada fichero o directorio que se lista, la primera información que se obtiene es algo como:

```
-rwxrw-r--
```

Es una cadena de 10 símbolos que pueden tomar los siguientes valores: `d`, `r`, `w`, `x`, `-`
El primer símbolo indica si es un fichero o directorio:

- `d`: indica que es un directorio
- `-`: indica que es un fichero

Los nueve siguientes símbolos indican los permisos de acceso y se agrupan en tres grupos:

- El primer grupo de 3 símbolos indican los permisos para el usuario propietario del fichero o directorio. En el ejemplo `rwx`
- El segundo grupo de 3 símbolos indican los permisos para el grupo al que pertenece el propietario del fichero o directorio. En el ejemplo `rw-`
- El último grupo de 3 símbolos indican los permisos para el resto de usuarios. En el ejemplo `r--`

Los símbolos `r`, `w`, `x` tienen diferentes valores en función de que sea un fichero o directorio.

Si es un fichero los permisos de acceso indican:

- `r`: permiso de lectura, permiso para leer o copiar el fichero. Si en su lugar hay un `-` indica que no tiene permiso de lectura
- `w`: permiso de escritura, permiso para cambiar el fichero. Si en su lugar hay un `-` indica que no tiene permiso de escritura
- `x`: permiso de ejecución, permiso para ejecutar el fichero. Si en su lugar hay un `-` indica que no tiene permiso de ejecución

Si es un directorio los permisos de acceso indican:

- `r`: permiso a los usuarios para visualizar los ficheros en el directorio. Si en su lugar hay un `-` indica que no tiene permiso para visualizar los ficheros
- `w`: permiso para borrar, crear o mover ficheros en el directorio. Si en su lugar hay un `-` indica que 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 `-` indica que no tiene permiso de acceso

En el ejemplo anterior `-rwxrw-r--` significa:

- El primer `-` indica que es un fichero
- `rwx` indica que el propietario del fichero puede leerlo, escribirlo y ejecutarlo
- `rw-` indica que el grupo del propietario del fichero, puede leerlo y escribir sobre él, pero no ejecutarlo
- `r--` indica que el resto de usuarios sólo pueden leerlo

4. Cambiar los permisos de acceso en el sistema de ficheros

Con la orden `chmod` se pueden cambiar los permisos de un fichero. Los permisos de un fichero únicamente pueden ser cambiados por el propietario de dicho fichero. La orden `chmod` tiene dos parámetros:

- Tres dígitos en octal que indican los nuevos permisos que se le da 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):

- 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

Por lo tanto los 3 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) indica que el grupo puede leerlo, escribirlo pero no ejecutarlo. El `4` (100) indica 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 dichos permisos y con la tercera orden comprobará cómo dicho fichero no puede ser modificado

Si mira las características de `chmod` en el manual con la orden `man`, observará que hay una alternativa a la hora de indicar los permisos que se cambian a un fichero. Esta alternativa consiste en utilizar símbolos como `+` (añadir), `-` (quitar), `r`, `w`, `x`,.. en lugar de los tres dígitos en octal.

5. Procesos

Un proceso es un programa en ejecución. Cada proceso tiene un identificador único, `PID` (IDentificador de Proceso).

El comando `ps` nos da información sobre los procesos de usuario que se están ejecutando en el terminal donde se ejecuta el comando `ps`. En concreto proporciona la siguiente información por cada proceso: su `PID`, el terminal asociado al proceso (TTY), el tiempo de CPU acumulado y el nombre ejecutable del proceso. Teclee:

```
ps
```

Y observe los procesos de usuario que se están ejecutando en el terminal desde el que ha lanzado la orden `ps`. Observe que hay dos, el propio `ps` y el terminal de texto (`bash`) donde estamos trabajando.

Para obtener todos los procesos que se están ejecutando en el sistema teclee:

```
ps  -e
```

Al comando `ps` se le puede añadir la opción `–f` para que nos muestre más información de los procesos, en concreto nos proporcionará de forma adicional: el identificador del propietario del proceso (UID), el PID del proceso padre (PPID), el porcentaje de uso del procesador (C) y 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 dicho proceso se ha metido en un bucle infinito.

Para matar un proceso se utiliza la orden `kill` acompañada del `PID` del proceso que se quiere matar. Si el proceso se resiste a ser matado utilice la opción `-9`. Tenga en cuenta que usted únicamente puede matar procesos de los cuales es usted propietario.

En el entorno gráfico abra un nuevo terminal de texto, con los menús de la barra superior: `“Aplicaciones/Accesorios/Terminal”`. En dicho terminal 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 la orden `ps -ef`. Obtendrá los procesos que se están ejecutando en el sistema. Entre dichos procesos verá el `sleep`, que ha lanzado en el otro terminal y el nuevo terminal de texto (`bash`). Por lo tanto verá dos terminales, en el que está trabajando en este momento y el nuevo terminal de texto que ha abierto y donde se está ejecutando el proceso `sleep.`

Para saber que PID tiene asignado el terminal en el que está situado en un momento dado ejecute:

```
ps | grep bash
```

Para ver los PID de todos los terminales que tiene abiertos teclee:

```
ps –e | grep bash
```

Apunte el PID del proceso `sleep` y del terminal donde se está ejecutando dicho proceso `sleep` y mátelos (primero el proceso `sleep` y luego el `bash`) con la orden:

```
kill  -9  [PID]
```

6. Ejercicio a realizar

a) Si se encuentra en la sesión correspondiente del laboratorio, donde va a generar el fichero para subir al moodle, borre antes todos los ficheros que ha podido crear previamente en el subdirectorio `lab3`, con la orden `rm`

b) Cree un directorio llamado `procesos` que cuelgue del directorio `lab3`

c) 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 estás tres órdenes para ejecutarla en dicho terminal de texto (no repita órdenes):

```
sleep  100000
cat
sort
```

Con lo cual tendrá tres procesos: un `sleep`, un `cat` y un `sort`, todos ellos esperando y cuatro terminales de texto

d) Vuelva al terminal de texto primero, donde no tiene ningún proceso corriendo.

e) Cambie su directorio de trabajo actual al directorio `procesos` que acaba de crear.

f) A partir de este momento, responda a todas las preguntas que se le plantean en la plantilla `FTEL-Tema1-actividades-Lab3-2014` que se podrá descargar del moodle.

g) Sin cambiarse de directorio de trabajo (`procesos`), **utilizando rutas relativas** y **en una sola línea de comandos** (utilizando tuberías, redirección y las órdenes `ps` y `wc`), cree un fichero llamado `numero-procesos`, en el directorio `lab3`, con el número de procesos que se están ejecutando en este momento **en el sistema.**

Si analiza el número de procesos contados, observará que aparecen dos más de los esperados. 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.

h) 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 leer, ni escribir, ni ejecutar dicho fichero.

i) Sin cambiarse de directorio de trabajo (`procesos`), **utilizando rutas relativas** y **en una sola línea de comandos** (utilizando tuberías), **añada** al fichero `numero-procesos` la lista de procesos (en formato extendido) que se están ejecutando en este momento **en el sistema**, ordenados alfabéticamente.

j) Sin cambiarse de directorio de trabajo (`procesos`), **utilizando rutas absolutas** y **en una sola línea de comandos** (utilizando tuberías), **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)

k) Compruebe que el contenido del fichero `numero-procesos` es correcto y que sus permisos son los deseados.

l) Responda la pregunta que le plantee el profesor en el laboratorio

m) Mate primero 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 por ejemplo con `ps | grep bash` y asegúrese de que no es uno de los terminales que mata).

Guarde el fichero de esta plantilla con el nombre `Grupo-Apell1-Apell2-L3.odt`, donde Apell1, Apell2 deben ser los que le corresponden (recuerde no utilizar en el nombre del fichero tildes).

Suba al Moodle el fichero generado y **compruebe en el Moodle que se ha grabado el fichero y el contenido del mismo**.

7. 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 al fich0
sort sort –o	Ordena alfabética o numéricamente los datos Guarda el resultado en otro fichero
chmod [3díg] fichero	Cambia los permisos de acceso del fichero
ps ps –e ps –f	Lista los procesos de usuario en el terminal Lista los procesos en el sistema Muestra más información de los procesos (UID, PPID, C, STIME)
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

Fundamentos de los Sistemas Telemáticos

Tema 2: Representación de la información

Gregorio Fernández Fernández

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

Este documento está diseñado para servir de material de estudio a los alumnos de la asignatura «Fundamentos de los Sistemas Telemáticos» del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Madrid.

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

Índice general

Capítulo 1

Terminología y objetivos

En este capítulo introductorio se presentan algunos conceptos generales y se concretan los objetivos del Tema.

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]:

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

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

objetivos, y utiliza los dos primeros para, en función de los datos que percibe, generar las acciones que conducen a los objetivos.

Debe usted interpretar «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, entender las diferencias entre «datos», «información» y «conocimiento» es fácil:

- 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 oido 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 que estudiará usted en una asignatura de tercer curso llamada «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 situaciones igualmente probables. En la asignatura «Teoría de la información» verá usted cómo se define con rigor en términos matemáticos.
- «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 en este Tema.

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...

Lo que vamos a estudiar en este Tema son 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.

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»*, que podemos traducir por **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 de este documento 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. No vamos a decir nada más de ellos aquí, porque se estudian en la asignatura «Introducción a la ingeniería». 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 al concepto de **jerarquía de memorias**. Lo único que a los efectos de este Tema nos interesa es conocer algunas características funcionales básicas de esos soportes para comprender los convenios de almacenamiento de los datos.

Biestables y registros

Un **biestable**, o **flip-flop**, es un circuito que estudiará usted en la asignatura «Electrónica digital». 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 3 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.

En el Tema 3 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

La memoria principal y las memorias ocultas (más conocidas como *«caches»*) se implementan mediante otras tecnologías electrónicas con las que se consiguen costes por bit y densidades de integración tales que actualmente es factible llegar a capacidades de almacenamiento del orden de KiB y MiB (para las memorias ocultas) y de GiB (para la memoria principal).

La memoria principal y las ocultas se diferencian en los tiempos de acceso (mayores los de la memoria principal), en las capacidades de almacenamiento (también mayores en la memoria principal, debido a la mayor densidad de integración de la tecnología empleada) y en los costes por bit (menores en la memoria principal). Por lo demás, ambas son volátiles, ambas son de lectura y escritura y en ambas se puede acceder a un byte o a varios bytes (normalmente a una palabra) en una sola operación.

Para poder acceder a un byte determinado es necesario poder identificarlo. 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».

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** (también llamados «**archivos**», ver apartado 5.4) como los que hemos manejado en el Tema 1. En el capítulo 6 veremos algunos convenios concretos para el almacenamiento en ficheros.

1.4. Extremistas mayores y menores

Antes de entrar a estudiar los convenios de representación de los distintos tipos de datos, hay un convenio general sobre el almacenamiendo en las memorias de acceso aleatorio que debemos conocer.

Hemos dicho que cada dirección identifica a un byte dentro de la memoria principal (o la oculta). 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 sus bytes, pero ¿en qué direcciones se almacenan éstos? Parece obvio que en direcciones consecutivas, pero ¿en qué orden?

[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).

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»).

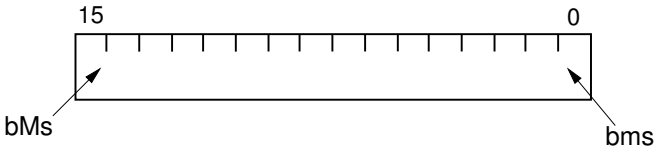


Figura 1.1: Representación gráfica de una palabra de 16 bits.

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: $16^6 = 2^4 \times 2^{20}$.

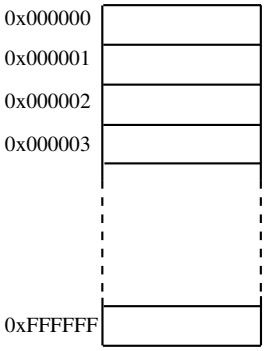


Figura 1.2: Representación gráfica de una RAM.

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)*.

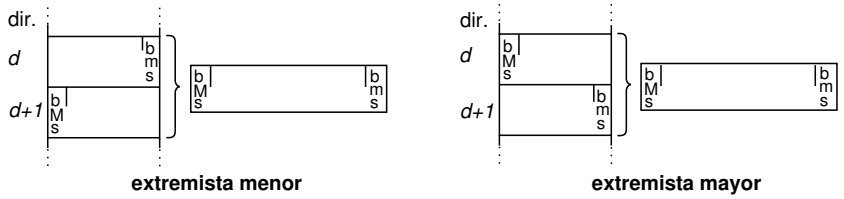


Figura 1.3: Convenios de almacenamiento para una palabra de dos bytes.

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. Objetivos y contenido del Tema

La Guía de aprendizaje de la asignatura establece que los resultados del aprendizaje relacionados con este Tema son:

- Conocer los convenios de representación binaria, transmisión y almacenamiento de la información.
- Conocer los principios de los algoritmos de detección de errores y compresión

Estudiaremos los convenios de representación en bajo nivel (en binario) de datos de tipo textual (capítulo 2), numérico (capítulo 3) y multimedia (capítulo 4, pero aquí introduciremos también algunas formas de representación simbólica 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 explicaremos en el capítulo 4 (aunque algunos también están incluidos el programa de la asignatura «Introducción a la ingeniería de telecomunicación»).

Veremos los principios de la detección de errores y la compresión en el capítulo 5. Y en el capítulo 6 resumiremos algunos convenios para el almacenamiento en ficheros de todos estos tipos de datos.

Capítulo 2

Representación de datos textuales

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 2.6), 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.

2.1. 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 2.4) es de longitud variable. También veremos un ejemplo de codificación con longitud variable relacionado con la compresión en el apartado 5.4.

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 2.1: 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 2.1 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».

2.2. 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.

La tabla 2.2 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>.

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 2.2: Código ASCII.

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.

2.3. 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 2.3 reproduce 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 (un lenguaje que estudiaremos en el Tema 3), 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	μ	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 2.3: 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».

2.4. 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 2.4 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 10xxxxxx
U+0800 a U+FFFF	zzzzyyyy yyxxxxxx	1110zzzz 10yyyyyy 10xxxxxx

Tabla 2.4: 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 2.5. 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
₧	PESETA SIGN	U+20A7	0xE282A7
€	EUROSIGN	U+20AC	0xE282AC
⇒	RIGHTWARDS DOUBLE ARROW	U+21D2	0xE28792
∞	INFINITY	U+2210	0xE2889E
∫	INTEGRAL	U+222B	0xE288AB

Tabla 2.5: 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 2.3), y, como puede usted comprobar, su codificación en UTF-8 se obtiene siguiendo la regla de la tabla 2.4.

- Sin embargo, «€», que también está en la tabla 2.3, 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, «¤».

2.5. 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 estas palabras en memoria: el programa seguía el convenio del PDP-11 (extremista menor), mientras que el ordenador que lo ejecutaba era extremista mayor.

2.6. 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 XML, un metalenguaje que mencionaremos más adelante. 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.

2.7. Hipertexto y documentos XML

Se llama «hipertexto» al texto que contiene referencias a otros textos (o a otros documentos de tipo audiovisual, en cuyo caso es más propio llamarlo «hipermedia») de modo que el acceso a estas referencias sea inmediato. Se trata de un concepto que actualmente es bien conocido por ser el fundamento de la web. Para su implementación se utiliza un lenguaje, HTML, que estudiaremos en el Tema 3. Baste decir aquí que, 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.

Lo mismo puede decirse de XML, un metalenguaje que también estudiaremos en el Tema 3: los documentos XML están formados por texto plano.

Capítulo 3

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 2.1), 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.). En la asignatura «Electrónica digital» estudiará usted algunos casos.

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 3.1 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).

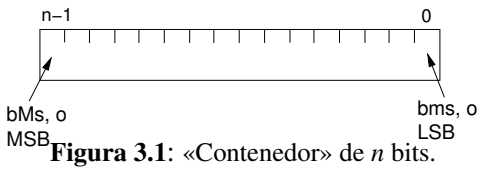


Figura 3.1: «Contenedor» de n bits.

3.1. 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.

3.2. 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.

3.3. 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 3.1 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 3.1: Comparación de convenios para formatos de coma fija.

3.4. Operaciones básicas de procesamiento

En este Tema 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 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, obviamante 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:

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

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:

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

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:

$$\begin{array}{rcl} 0x433 & \rightsquigarrow & 010000110011 \\ 0x3A5 & \rightsquigarrow & \underline{001110100101} \\ & & 011111011000 \rightsquigarrow 0x7D8 \end{array}$$

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 siguiente Tema 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 o resta 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 de Álgebra, 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 3.2, 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 3.2: 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 3.2), 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):

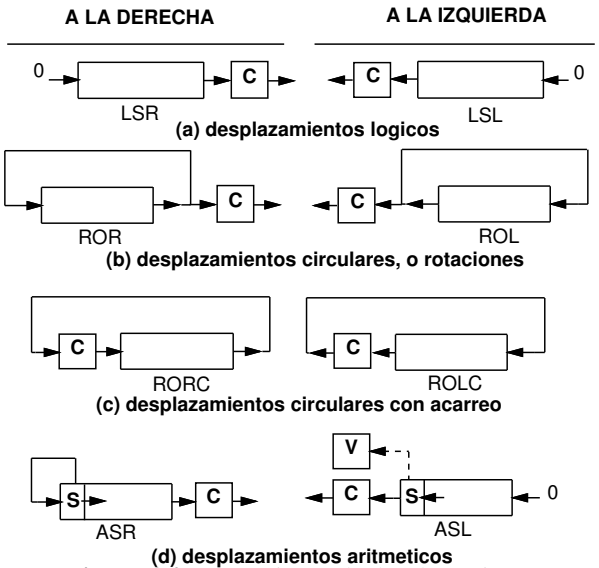


Figura 3.2: Operaciones de desplazamiento.

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 3.2(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 3.2(b). Los nemónicos son `ROR` («Rotate Right») y `ROL` («Rotate Left»).

Las rotaciones a través del indicador C funcionan como muestra la figura 3.2(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 3.2(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).

3.5. Números racionales

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

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.

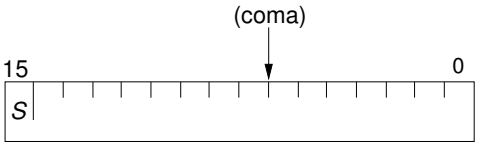


Figura 3.3: Formato de 16 bits para representación en coma fija de números racionales.

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 3.3).

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 3.4):

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

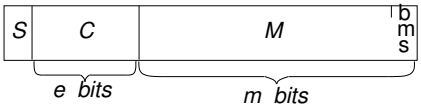


Figura 3.4: Formato de coma flotante.

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 π :

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

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 $C = 2^{e} - 1$).

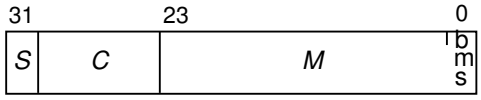


Figura 3.5: Un formato de coma flotante con 32 bits.

Como ejemplo, veamos la representación del número π en el formato de la figura 3.5, 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 5×10^{-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 3.6.[3]

Los convenios son:

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

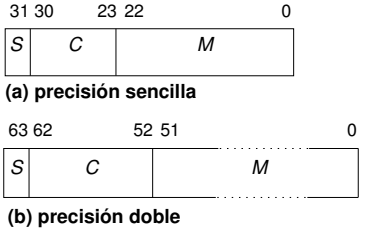


Figura 3.6: Formatos de la norma IEEE 754.

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 3.6(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 3.6 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 $M \neq 0, X = \text{NaN}$
   («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 3.6(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 $M \neq 0, X = \text{NaN}$

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 4

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»[1] 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 «Introducción a la ingeniería de telecomunicación» se introducen los principios de las conversiones de señales analógicas a digitales y viceversa (principios que estudiará usted con más profundidad en «Sistemas y señales», de segundo curso). 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.

[1] Free On-Line Dictionary Of Computing, `http://foldoc.org`

4.1. Digitalización

La cadena analógico-digital-analógico

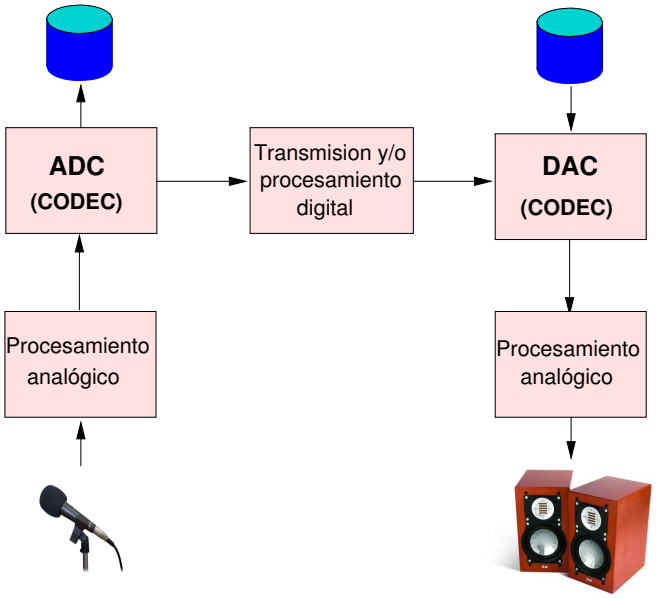


Figura 4.1: Procesamiento del sonido

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 4.1 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.

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 4.2):

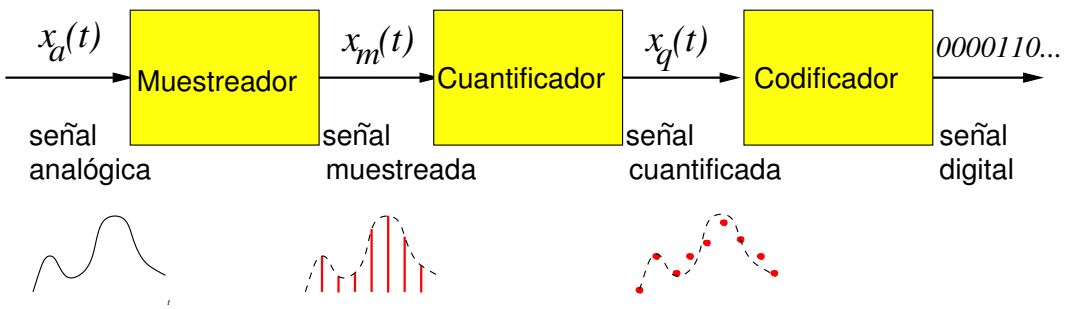
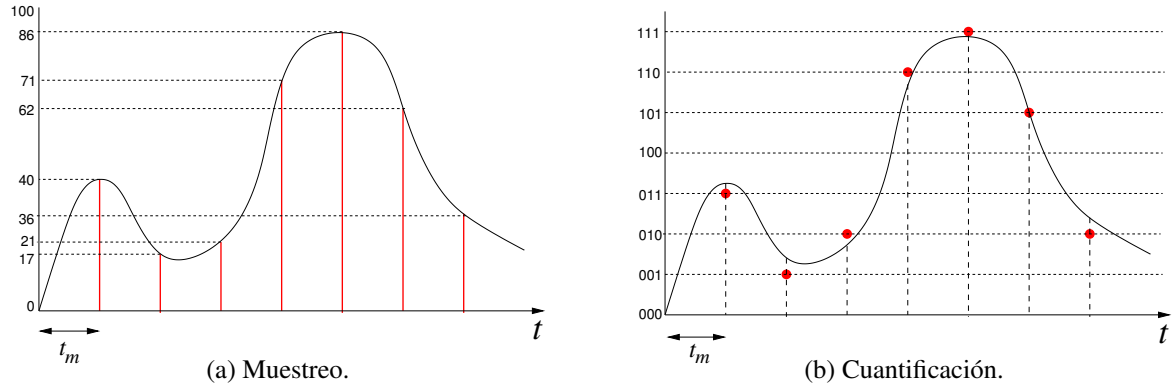


Figura 4.2: Conversión de analógico a digital.

1. El **muestreo** es una *discretización en el tiempo*[2]. 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 4.3a, 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 4.3b indica el resultado de cuantificar las muestras del fragmento de señal del ejemplo con 8 niveles (resolución $n = 3$).



(a) Muestreo. (b) Cuantificación.

Figura 4.3: 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 4.4, 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.

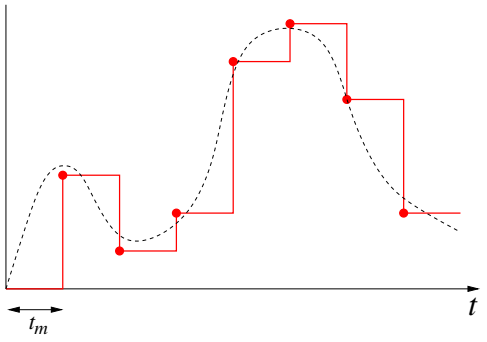


Figura 4.4: Reconstrucción con *zero-holder hold*.

[2] «Discretizar» una magnitud contínua 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 $f_m = 1/t_m$. Un resultado fundamental en este campo es el **teorema del muestreo de Nyquist-Shannon**[3], que podemos resumir así: 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 4.3b 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

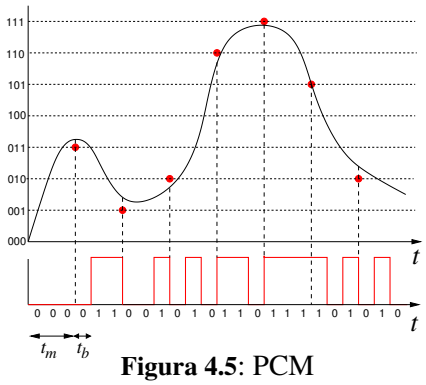


Figura 4.5: PCM

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 4.5 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 4.4) 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[4].

[3] «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).

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

4.2. 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 4.1 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 4.1: 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[5]. 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 refererimos 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 encontar 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.

[5]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`.

4.3. 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 4.6 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 4.6: 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 totalmente inadecuada.

4.4. 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 contínuo 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**»[6].

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

[6]Es importante la matización «en este contexto». En transmisión de datos «frame» se traduce por «trama» (y así lo haremos en el Tema 4), 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 5

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.

Hay muchos algoritmos para detectar si se ha producido un error en la transmisión de un mensaje codificado en binario. En general, se basan en *añadir redundancia* al mensaje. Como ilustra la figura 5.1, al mensaje original se le añaden bits de comprobación, que no aportan información para el receptor, pero sirven para verificar si se ha producido o no algún error.

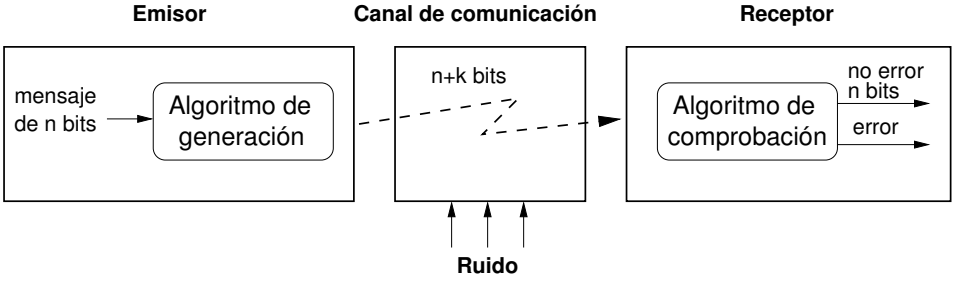


Figura 5.1: Detección de errores.

Hay algoritmos que permiten no sólo detectar errores, también pueden corregirlos. Este asunto se estudia en otras asignaturas de Telemática. Veremos aquí el principio de los dos algoritmos de detección más sencillos, y de las acciones posibles cuando se detecta un error.

Después entraremos en 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 $C = 100/k\,\%$.

5.1. Bit de paridad

El algoritmo 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 se conviene en 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 3.2) 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»):

- 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 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 es un caso degenerado de otros más robustos y muy utilizados en redes (como Ethernet) y en las transferencias con discos: los CRC (Cyclic Redundancy Check). Los CRC tienen una fundamentación matemática: la teoría algebraica de anillos. Si se ve usted obligado estudiar fundamentos matemáticos como éstos u otros piense que no es por martirizarle: en otras asignaturas irá viendo que son imprescindibles para la ingeniería.

5.2. 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.

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 diferentes modos de definir la función, y, por tanto, varios algoritmos de detección de errores. Uno de los más sencillos es 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*

5.3. 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 contínua**: 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 dúplex»), 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 . Los algoritmos y protocolos concretos para la implementación de estos métodos se estudian en la asignatura «Teoría de la información».

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 y en los dispositivos de CD y DVD.

Los códigos correctores tienen 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.

Finalmente, señalar que 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.

5.4. 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 de interés para 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).

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 dato

`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, en la asignatura «Introducción a la ingeniería» le explican que 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 5.2: 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

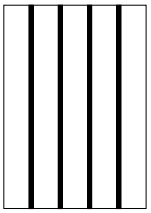


Figura 5.2

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.

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»).

	N	Cod.
B	11	1
D	8	01
G	5	001
E	4	0001
F	3	00001
A	3	000001
C	2	0000001

Tabla 5.1: Ejemplo de código Huffman.

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 5.1.

Con este código, el dato original se codificaría como 000001000001.... El número total de bits es $11 \times 1 + 8 \times 2 + 5 \times 3 + 4 \times 4 + 3 \times 5 + 3 \times 6 + 2 \times 7 = 105$ 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 5.2: 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 5.2. 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 10:1.

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»[1].

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», apartado 6.1) 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 que no hemos tratado en este Tema: el cifrado (a veces llamado «encriptación»).

No es causalidad 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

[1] Como tampoco debería traducirse «directory» por «carpeta»: ésta sería la traducción de «folder».

deben hacer una sola cosa (pero hacerla bien) y otro, que deben 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 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.
- Como ya sabe usted por las prácticas del Tema 1, «`>`» hace una *redirección*: dirije 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`).

5.5. 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 4.1), 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 5.4, 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, y las estudiará usted en la asignatura «Señales y sistemas». La más utilizada en los algoritmos de compresión con pérdidas es la **DCT** (transformada discreta del coseno), que es una variante de la más clásica transformada de Fourier.

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 4.1). Como hemos visto en el apartado 4.1, 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 4 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 5.3, tomada de la Wikipedia[2], ilustra muy claramente la idea de este método de compresión.

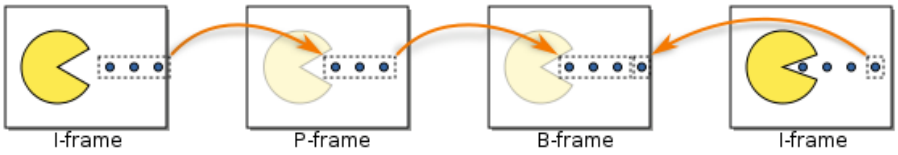


Figura 5.3: Fotogramas I, P y B.

[2] `http://en.wikipedia.org/wiki/File:I_P_and_B_frames.svg`

Capítulo 6

Almacenamiento en ficheros

En el Tema 1 se introdujo el concepto de fichero y en éste hemos vuelto sobre él para matizar la diferencia entre «fichero» y «archivo», que es un tipo de fichero (apartado 5.4). También se explicó que el sistema de gestión de ficheros (SGF), que forma parte del sistema operativo, contiene programas para acceder a los datos de los ficheros, almacenados en los dispositivos de memoria secundaria (disco magnético, CD, DVD, *flash*, etc.).

La transferencia de datos entre las memorias secundarias y la memoria principal (de acceso aleatorio) se realiza mediante la técnica de acceso directo a memoria (DMA, apartado 1.3): en cada operación de lectura o escritura en la memoria secundaria se transfiere un «bloque». Un **bloque** es un múltiplo de 512 bytes formado por uno o varios **sectores**, que es como normalmente se organizan los datos en el soporte físico. El SGF se ocupa de la correspondencia entre los sectores y los ficheros, asignando sectores a ficheros cuando los necesitan, o liberándolos cuando ya no son necesarios. Por ejemplo, cuando se borra un fichero, realmente no se borran físicamente los datos de los sectores, simplemente se liberan esos sectores (por eso hay herramientas que permiten recuperar ficheros borrados).

Una cosa es el SGF, conjunto de programas que hacen todo eso, y otra el **sistema de ficheros**, que es la forma de organización de los ficheros, directorios y dispositivos periféricos. Todos los sistemas operativos tienen un SGF que trata con un sistema de ficheros, pero es frecuente que pueda tratar varios. Por ejemplo, en los sistemas Windows actuales el sistema de ficheros se llama «NTFS», pero el SGF puede tratar también el más antiguo «FAT32», que aún se usa, por ejemplo, en memorias *flash* extraíbles. En Linux, los sistemas de ficheros principales son «ext3», o el más reciente «ext4» (que es el que hemos utilizado en las prácticas del Tema 1), pero el SGF puede tratar otros, como FAT32 y NTFS. Además, hay sistemas de ficheros especializados para ciertos soportes, como el «ISO 9660» para CD y DVD.

En este capítulo presentaremos algunas características de los ficheros que son comunes e independientes del sistema de ficheros. Pero veamos antes los distintos tipos de ficheros.

6.1. Tipos de ficheros

Ficheros especiales

Todos los sistemas operativos modernos han adoptado una idea procedente de Unix: integrar en el SGF los dispositivos de entrada y salida. En Unix se agrupan en el directorio `/dev`. El objetivo es hacer uniforme el acceso mediante llamadas estándar al sistema. Hay *ficheros especiales de caracteres* y *ficheros especiales de bloques*. Los primeros corresponden a periféricos lentos, como el teclado (`/dev/tty`) o el ratón (`/dev/input/mouse`), y los segundos a los rápidos, que transfieren mediante DMA, como los discos magnéticos (`/dev/hda`, o `/dev/sda`) o los ópticos (`/dev/cdrw` o `/dev/dvdrw`) (independientemente de que dentro de éstos haya ficheros «normales», o «regulares»).

También hay ficheros especiales con funciones específicas llamados «pseudodispositivos». Un ejemplo es `/dev/null`, que sirve de «sumidero»: todo lo que se envía a él se pierde. Otros ficheros especiales son los directorios. Y hay algunos más, como los enlaces simbólicos y los *sockets*.

Ficheros regulares

En el resto de este capítulo trataremos únicamente de los ficheros que se llaman «regulares»: los que contienen datos almacenados en algún soporte de memoria secundaria. La abstracción que ofrece el sistema de gestión de ficheros permite que los programas que manipulan esos ficheros los «vean» simplemente como una secuencia de bytes, sin ocuparse de cómo se almacena esa secuencia de bytes en el soporte. Por ejemplo, en un disco magnético un fichero se almacena en «sectores» (normalmente, de 512 bytes) que están distribuidos por las distintas pistas del disco, pero para los programas que hacen llamadas al SGF se abstraen esos detalles de localización.

Por otra parte, en la mayoría de los sistemas operativos el sistema de ficheros no se ocupa del contenido de los ficheros: para él, un fichero no es más que una secuencia de bytes. Es decir, solamente ofrece un tipo de fichero regular. La estructura y la interpretación del contenido de un fichero queda a cargo de los programas de aplicación que lo usan.

Metadatos

Para realizar las operaciones con los ficheros (crear, borrar, modificar, etc.), el SGF tiene que poder identificar a cada fichero por su nombre y conocer sus propiedades: tipo (especial, directorio, regular, etc.), fechas de creación y de última modificación, propietario, permisos de lectura, escritura y ejecución, y si es regular, bloques que tiene asignados, tamaño (en bloques o en bytes), etc. Algunos de estos datos son los que ha visto usted en las prácticas del Tema 1 con la orden «`ls -l`». Como son datos sobre los datos, se llaman **metadatos**. Pero observe que estos metadatos se refieren a propiedades generales del fichero, no a su contenido. También hay metadatos sobre el contenido, como veremos en el siguiente apartado.

La forma de mantener los metadatos es distinta para cada sistema de ficheros. El sistema «FAT» debe su nombre a que guarda los de todos los ficheros en una estructura de datos llamada «tabla de asignación de ficheros» (File Allocation Table). En NTFS es la «MFT» (Master File Table). El enfoque de Unix es diferente: cada fichero tiene asociada una pequeña estructura (normalmente, 256 bytes), llamada «**inode**» con sus metadatos. (Se pueden ver los números de los inodes asociados a los ficheros con «`ls -i`»).

Pero veamos los convenios para los tipos de ficheros regulares, independientemente de los sistemas de ficheros.

6.2. Ficheros regulares

Identificación del tipo de contenido

Para el sistema de ficheros todos los ficheros regulares son iguales, 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 2.5), 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 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 6.5).
- 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 3 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.

6.3. Ficheros de texto, documentos, archivos y datos estructurados

Hay que distinguir entre texto plano y texto con formato (apartado 2.6). En el primer caso, los datos están formados por la cadena de bytes correspondiente a las codificaciones de los caracteres (apartado 2.5), 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.

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 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 2.6, 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 5.4, formando lo que se llama un «**tarball**». Un fichero comprimido tiene también su cabecera antes de los datos comprimidos. En `gzip` son diez bytes con el número mágico, versión y fecha, y, tras los datos, ocho bytes para comprobación de errores con CRC-32 (apartado 5.1).

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.

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** *(«record»)*[1]. 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**, cuyos fundamentos estudiaremos en el Tema 5.

[1] Aunque en español los llamemos igual, estos registros son estructuras de datos que no tienen nada que ver con los registros hardware *(«registers»)*, componentes físicos de almacenamiento (apartado 1.3).

6.4. 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 4.1) y los formatos de los ficheros generados están íntimamente relacionados, y dependen de la aplicación. Los podemos clasificar según el grado de compresión (apartados 5.4 y 5.5) que aplican:

- Sin compresión. La mayoría están basados en PCM (apartado 4.1). 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 5.4) 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 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, libre 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 6.1 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 dos píxeles, y las dimensiones de la imagen (32×10) son muy pequeñas.

```
P1    # Número mágico
32 10 # 32x10 píxeles
#   F   -    T    -   E    -   L
11111110 111111110 11111110 1100000
11111110 111111110 11111110 1100000
11000000 000110000 11000000 1100000
11000000 000110000 11000000 1100000
11111100 000110000 11111100 1100000
11111100 000110000 11111100 1100000
11000000 000110000 11000000 1100000
11000000 000110000 11000000 1100000
11000000 000110000 11111110 1111111
11000000 000110000 11111110 1111111
```

Figura 6.1: FTEL.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 6.2 se ha utilizado M = 5.

```
P3    # Número mágico
32 10 # 32x10 pixels
5     # Máxima intensidad
#                   F                   -                   T                       -\
                    E                   -                   L
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 0 5 5 0 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
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 0 5 5 0 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 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 5 0 5 5 0 5 5 5 5 5 5\
0 5 0 0 5 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 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 5 0 5 5 0 5 5 5 5 5\
0 5 0 0 5 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 0 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 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 0 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 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 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 5 0 5 5 0 5 5 5 5 5\
0 5 0 0 5 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 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 5 0 5 5 0 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 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 0 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 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 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 5 0 5 5 0 5 5 5 5 5\
0 5 0 0 5 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 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 5 0 5 5 0 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 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0
5 3 0 5 3 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 5 0 5 5 0 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 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0
```

Figura 6.2: FTEL.ppm.

Si analiza usted el contenido del fichero verá que los píxeles de la F están codificados con R = 5, G = 3, B = 0, lo que da un resultado naranja, los de la T con R = 5, G = 0, B = 5 (magenta), los de la E

con R = 0, G = 5, B = 0 (verde) y los de la L con R = 5, G = 0, B = 0 (rojo). Los píxeles blancos, con R = 5, G = 5, B = 5. 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 6.1.

Número mágico	Tipo	Extensión	bpp
P1 (0x5031)	bitmap en ASCII	`.pbm`	8×1
P2 (0x5032)	graymap en ASCII	`.pgm`	entre 8×1 y 8×3
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 6.1: 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 6.1 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 tres últimos ya hemos comentado algo sobre los algoritmos de compresión que utilizan (apartados 5.4 y 5.5) y sus números mágicos (apartado 6.2). 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 5.4) 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 5.5). 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 4.3 vimos el principio de los formatos gráficos vectoriales, ilustrándolo con un ejemplo sencillo en el formato SVG (figura 4.6). Hay otros formatos (algunos muy conocidos, 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 4.6 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
```

Figura 6.3: Beastie (Logo de freeBSD).

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 6.3 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.

El fichero SVG completo es demasiado grande para incluirlo aquí; la figura 6.4 muestra una pequeña parte: la que dibuja 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.

```
 ...
 <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 6.4: El cuerno derecho del demonio.

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 4.6. Para dibujos no triviales sería demasiado laborioso, y es imprescindible utilizar un programa (herramienta). La imagen de la figura 6.3 (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 6.1): al explorar el contenido encuentra en la primera línea «`<?xml...>`», que indica que utiliza el metalenguaje XML, 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 5.5), 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 5.5 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 sofware 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 6.2 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. MP 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 el códec de audio Vorbis 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 6.2: 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, aún en desarrollo, es HTML5, que 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 (2014) 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.

6.5. 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 Tema 3, 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. Un programa de ese tipo es el intérprete de órdenes (o «shell») con el que hemos practicado en el Tema 1. 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**. Cuatro 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[2]. 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 (Tema 1) 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:

- **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, apartado 6.1). 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.

[2]Como veremos en el Tema 3, lo que genera el traductor no es directamente ejecutable, tiene que pasar por otro proceso, llamado «montaje».

«**Blob**» es un acrónimo de *binary large object* y en principio es un término ligado a las bases de datos. El *Free On-Line Dictionary Of Computing* (`http://foldoc.org`) 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*, Tema 1), 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».

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 solo 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 solo 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:

$$0b\underbrace{10}_{2}\underbrace{1001}_{9}\underbrace{1110}_{E}\underbrace{1111}_{F} = 0x29EF = 2\times16^3 + 9\times16^2 + 14\times16 + 15 = 10735$$

$$0b\underbrace{111}_{7}\underbrace{1010}_{A}\underbrace{0000}_{0}\underbrace{1100}_{C}\underbrace{1101}_{D} = 0x7A0CD = 7\times16^4 + 10\times16^3 + 12\times16^1 + 13 = 499917$$

Bibliografía

Para complementar o ampliar el contenido de este Tema hay numerosas fuentes que puede usted localizar buscando en la web. Para reforzar los conceptos con ejercicios, algunos libros recomendables son:

- A. Prieto, A. Lloris y J.C. Torres: *Introducción a la informática,* 4ª *ed.* McGraw–Hill, Madrid, 2006.

  La representación de datos se trata en el Capítulo 4, con numerosos ejercicios propuestos.

- 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.

- J. Glenn 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 estudiado en este Tema. 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, pero a un nivel más superficial. También tiene muchos ejercicios propuestos.

Laboratorio 4. Representación de la información

Objetivos:

- Experimentar con distintos formatos de codificación de textos y aprender a averiguar el tipo de contenido de un fichero examinando las cabeceras internas.
- Conocer el formato y practicar el uso de herramientas que permiten cambiar metadatos, y manipular ficheros de audio mp3.

Recursos:

- El ordenador del laboratorio o cualquier equipo con un sistema de tipo Unix.
- Portal Moodle de la asignatura

Resultados:

Como resultado de esta práctica deben subirse a Moodle dos entregas en dos tareas:

- Un formulario (`resultados-lab4-2014.odt` disponible en Moodle) que debe ir rellenando mientras realiza la práctica, siguiendo las indicaciones **en negrita** incluidas en diversos apartados de este enunciado.
- Un fichero mp3 procesado según se indica al final de este documento.

Actividades previas:

Antes de la sesión de laboratorio, debe leer previamente este documento e intentar realizar las actividades indicadas para asegurar que puede completarlo antes de finalizar la sesión.

1. Codificación de textos

Empiece desde el entorno gráfico, abra un terminal de texto, cree un nuevo directorio `lab4`, vaya a este directorio `lab4` y baje del Moodle el fichero `resultados-lab4-2014.odt`.

Sin moverse del directorio `lab4` realice paso a paso las siguientes actividades:

A. Cree un fichero con el nombre `apell` que contenga los caracteres de su primer apellido (no ponga tildes y, si tiene alguna ñ, cámbiela por n), seguidos de un espacio y el símbolo €. Puede hacerlo con un editor de texto o, de forma más sencilla, usando la orden `echo`:

```
$ echo su-apellido € > apell
```
[1]

Para comprobar la creación del fichero `apell` visualice su contenido en la pantalla.

B. Utilice la orden `wc -c` para contar el número de bytes del fichero `apell`.

Tenga en cuenta para los siguientes apartados que los editores de texto y la orden `echo` añaden el carácter `0x0A` (salto de línea) al final de cada línea.

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? **Anote en el formulario el número de bytes del fichero y responda a la pregunta formulada, justificando su respuesta.**

[1] En lo sucesivo el símbolo $ significa que debe escribir la orden indicada y, si en la descripción de la orden aparece algún campo en cursiva, como *su-apellido*, debe sustituirlo por un valor concreto (su primer apellido).

C. Compruebe la codificación del fichero `apell` con la orden `file`:

```
$ file apell
```

D. El programa `iconv` convierte un fichero de una codificación a otra. La orden

```
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 (equivalente a ISO 8859-15) y UTF-8[2].

Ahora, si su fichero `apell` está en UTF-8, conviértalo a LATIN9 y guárdelo en otro fichero de nombre `apell-cod` con la órden:

```
$ 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.

Con la orden `file` compruebe que ha cambiado la codificación del fichero `apell-cod`

E. Mire con `wc -c` el número de bytes del nuevo fichero `apell-cod`. **Copie al formulario la salida de las órdenes `file apell` y `file apell-cod` y explique las diferencias que se aprecian en ambos ficheros.**

F. 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 columna se muestra su posible interpretación como ASCII (si no la tiene, muestra un punto).

Compruebe con la orden `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. **Copie al formulario la salida que genera `hd` para ambos ficheros `apell` y `apell-cod` y explique sus diferencias.**

G. 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 a ficheros conteniendo a su vez un conjunto de directorios y ficheros de texto (en xml) y que están comprimidos con ZIP.

Para ver qué ficheros y directorios contiene el *archivo* `respuestas-Lab4-2014.odt` (el formulario de entrega que está rellenando) ejecute la orden:

```
$ unzip –l resultados-lab4-2014.odt
```

Copie al formulario el resultado que aparece en pantalla.

[2] Puede comprobar cuál es la codificación por defecto en su sistema con echo $LANG

2. Identificación del tipo de contenidos: Metadatos

Para el sistema de ficheros todos los ficheros regulares son iguales, pero los programas que trabajan con ellos necesitan conocer qué es lo que contienen: texto, imagen, o código ejecutable. Hay varias maneras de hacerlo y una de ellas es incluir datos sobre el tipo de contenido (es decir, *metadatos*) dentro del propio fichero, normalmente al principio delante de los datos propiamente dichos. Para explorar cómo identificar el tipo de contenidos de un fichero, realice los siguientes pasos:

A. Copie en su directorio `lab4` un fichero que contenga una imagen codificada con el formato GIF, poniéndole como nombre `imag.gif`. Esta imagen GIF puede descargarla de la web, pero en su sistema hay cientos de imágenes que puede localizar con la orden `locate .gif`

Primero mire qué tipo de contenido tiene el fichero `imagen.gif` con la orden:

```
$ file imag.gif
```

Luego copie el fichero `imag.gif` a un fichero `imag.exe`

```
$ cp imag.gif imag.exe
```

y compruebe, con la orden `file` si el fichero `imag.exe` es de tipo ejecutable. **Escriba en el formulario el resultado de la comprobación y añada una explicación sobre la relación que observa entre el tipo de un fichero y su extensión.**

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. Debe mantener el permiso de lectura para que `file` pueda leer la cabecera.

Compruebe, con la orden `file` si el fichero `imag.exe` es de tipo ejecutable. **Escriba en el formulario el resultado de la comprobación y añada una explicación sobre la relación entre el tipo de un fichero y sus permisos.**

B. La cabecera de un fichero de tipo GIF empieza con la "firma" (3 bytes) que representan las codificaciones ASCII de los caracteres "GIF", y sigue con la "versión" (3 bytes), que puede ser la codificación de los caracteres "87a" o la de los caracteres "89a"[3].

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 la orden

```
$ hd imag.gif | less
```

Copie al formulario los primeros 20 dígitos hexadecimales de la primera línea. De ellos, empiece por observar los 12 primeros y busque en la tabla de representación de caracteres ASCII a qué 6 caracteres corresponden (si no tiene la tabla, haga `man ascii`). **Copie al formulario los 6 caracteres resultantes.** Observe luego los 4 dígitos hexadecimales siguientes e interprételos como un número entero codificado en formato de coma fija de 16 bits, almacenado con convenio extremista menor. **Copie al formulario el número entero resultante**. Repita lo mismo con los 4 dígitos hexadecimales siguientes. **Copie este segundo número al formulario y explique qué relación guardan los números obtenidos con el resultado que muestra la orden `file imag.gif`.**

C. Genere un fichero de texto plano de nombre `raro.gif` que contenga solamente "GIF89a2014" (respetando mayúsculas y minúsculas) y compruebe el tipo con:

[3] La especificación completa está en http://www.w3.org/Graphics/GIF/spec-gif89a.txt

```
$ file raro.gif
```

Explique en el formulario por qué el resultado de la orden `file raro.gif` indica que el fichero `raro.gif` contiene una imagen de tamaño 12338 x 13361 cuando, en realidad, no contiene ninguna imagen. Explique de dónde sale que el tamaño de la imagen sea 12338 x 13361.

D. Baje de moodle a su directorio `lab4` los ficheros `glow.tiff` y `gmarbles.tiff`. Con la orden `file` *`nom-fich`* obtenga sus metadatos y deduzca su significado. **Cópielo al formulario.**

Use la orden `hd` *`nom-fich`* `| less` sobre ambos ficheros para averiguar cuál es el número mágico de cada uno de ellos (8 primeros dígitos hexadecimales). **Copie al formulario los 8 dígitos hexadecimales y su correspondiente valor ASCII**. **Explique la relación entre el número mágico y los metadatos** (busque en Internet el significado de los 4 primeros caracteres del número mágico para los ficheros tiff).

E. Use la orden `hd` para averiguar cuál es el número mágico de un fichero ejecutable en Unix, contenido en los cuatro primeros caracteres. Por ejemplo aplíquelo al fichero que contiene la orden `cat`:

```
$ hd /bin/cat | less
```

Copie al formulario los 8 primeros dígitos hexadecimales y su correspondiente valor ASCII. Explique la relación entre el número mágico y los metadatos (busque en Internet el significado de los 3 últimos caracteres**).**

3. Manipulación de contenidos de ficheros mp3

El objetivo de esta parte de la práctica es aprender el uso de dos herramientas muy utilizadas para manipular los contenidos de ficheros de audio:

- `avconv`, conversor de audio y video, que permite manipular ficheros de audio y de video[4]
- `id3`, herramienta de edición de las etiquetas informativas (metadatos) que se incluyen en los ficheros mp3 para facilitar su catalogación, siguiendo el estándar ID3[5]

Para realizar esta parte de la práctica debe seguir los siguientes pasos:

A. Empiece por descargase un fichero mp3 disponible bajo licencia "Creative Commons". Para ello, abra el navegador y vaya a la página: http://creativecommons.org/wired/ y descargue, al directorio `lab4`, alguna de las canciones en formato mp3 (en el enlace subrayado de la canción, seleccione con el botón derecho del ratón "*guardar enlace como*"). Cambie el nombre original por otro nombre más corto que no contenga espacios en blanco, manteniendo la extensión mp3 y utilizando una orden similar a la del siguiente ejemplo (¡comillas necesarias!):

```
$ mv "Beastie Boys - Now Get Busy.mp3" nom-fich
```

En el terminal de texto, averigüe el número de bytes que ocupa su fichero mp3 (por ejemplo con la orden `ls -l`). **Escriba en el formulario el nombre de su fichero mp3 y su número de bytes.**

[4] Más información en http://www.libav.org/
[5] Información adicional en http://www.id3.org

B. La herramienta `avconv` admite muchas opciones y formatos. Para "cortar" el fichero mp3 que se ha bajado y generar otro fichero mp3 de menor duración, utilícela en la forma:

```
$ avconv -i nom-fich -ss inicio -t duración -acodec copy nom-fragm
```

en donde:

`-i` *nom-fich*	indica el fichero audio origen
`-ss` *inicio*	indica el instante en segundos en el que se quiere empezar a extraer
`-t` *duración*	la duración en segundos del fragmento a extraer (a partir del inicio)
`-acodec copy`	indica que el audio debe copiarse sin ninguna transformación
nom-fragm	es el nombre dado al fichero de audio mp3 resultante de la extracción

Ejecute la orden `avconv` poniendo los valores adecuados de las opciones para que "corte" el fichero *nom-fich* a partir de algún momento distinto del de inicio y obtenga un fragmento de duración 10 segundos, al que debe dar un nombre *nom-fragm* distinto del anterior pero con igual extensión mp3. **Copie al formulario de resultados la orden que ha dado para cortar el fichero.**

C. Ejecute ahora

```
$ avconv -i  fich
```

sobre ambos ficheros, el original *nom-fich* y el cortado *nom-fragm*, y podrá ver que en las dos líneas anteriores a la última donde se indica que debe proporcionar un fichero de salida, se muestran los metadatos, la duración, la tasa de bits (bitrate) y la frecuencia de muestreo del fichero de audio. **Copie al formulario de resultados estas dos líneas, para ambos ficheros.**

D. A continuación va a practicar con las utilidades `id3` e `id3v2` que facilitan la edición[6] de etiquetas en ficheros de audio siguiendo respectivamente las versiones 1 y 2 del estándar ID3 definido para incluir metadatos (etiquetas) en ficheros de audio. La versión 1 de ID3 establece las siguientes etiquetas, que se guardan en un bloque de 128 bytes al final del fichero mp3:

- Título (30 caracteres)
- Artista (30 caracteres)
- Álbum (30 caracteres)
- Año (4 caracteres)
- Comentario (30 caracteres)
- Género musical (1 carácter)
- Pista (valor entre 0 y 255)

La versión 2 de ID3 incluye las mismas etiquetas al final del fichero pero, además, añade otras etiquetas al comienzo.

Tecleando `man id3`, o simplemente `id3`, puede ver las opciones que se pueden usar para cambiar las diferentes etiquetas. (Lo mismo para `id3v2`)

Por ejemplo `id3 -a "Pepe Perez"` *nom-fich* cambiaría 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).

[6]Existen también editores gráficos, como EasyTAG, que facilitan la edición de etiquetas tipo ID3 en los ficheros de audio. No se va a utilizar EasyTag en la práctica, sin embargo es recomendable que experimente con él.

Ejecute `id3` con opciones `-l` o `-lR` para obtener los valores de las etiquetas ID3:

`$ id3 -l nom-fich` o `$ id3 -lR nom-fich`

Copie los valores de las etiquetas al formulario de resultados de la práctica.

Con la orden `hd` puede ver los contenidos del principio y del final del fichero y averiguar con qué versión de ID3 se incluyeron las etiquetas al crearse el fichero original. Dando la orden:

`$ hd nom-fich| head`

verá si el fichero comienza con 0xFFFB, en cuyo caso se tratará de la versión 1 de ID3v1, o comienza con "ID3" y, en tal caso, las etiquetas corresponderán al estándar ID3v2.

Ejecutando la orden:

`$ hd nom-fich| tail`

si el fichero tiene etiquetas ID3v1, verá que al final aparece "TAG", luego el título,…

E. Borre las etiquetas que hayan podido quedar en el fichero cortado, utilizando la herramienta `id3v2`, que elimina tanto las etiquetas ID3v1 como las ID3v2, con la orden:

`$ id3v2 -D nom-fragm`

Añada nuevas etiquetas ID3 personalizadas al fichero que contiene el fragmento. En concreto añada las etiquetas: *Artist* (poniendo las iniciales de su nombre), *Year* (el año actual) y *Comment* ("modificado por" seguido de sus iniciales). Una vez hechas las modificaciones, liste con la orden `id3` los valores resultantes y **cópielos al formulario de resultados de la práctica.**

F. Realice, desde el navegador Firefox, las dos entregas de esta práctica:

- Cambie el nombre al fichero que contiene el fragmento de audio con las nuevas etiquetas, asignándole un nombre según el formato establecido para las prácticas pero manteniendo, en este caso, la extensión mp3 (Grupo-Apellido1-Apellido2-L4.mp3) y súbalo en la tarea "LAB4-Gxx. Entrega del fichero mp3" del Moodle
- Suba al Moodle, en la tarea "LAB4-Gxx. Entrega del fichero de resultados" el fichero que ha ido rellenando durante la realización de la práctica, asignándole el nombre según el formato establecido: Grupo-Apellido1-Apellido2-L4.odt.

Fundamentos de los Sistemas Telemáticos

Tema 3: Estructura y funcionamiento de procesadores

Gregorio Fernández Fernández

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

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

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

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

Índice general

Capítulo 1

Terminología y objetivos

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

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

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

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

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

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

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

Dos comentarios:

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

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

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

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

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

Podemos, pues, hablar de dos tipos de procesadores:

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

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

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

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

[3]De momento, la R.A.E. no ha enmendado la más que obsoleta definición de microprocesador: «Circuito constituido por millares de transistores integrados en un chip, que realiza alguna determinada función de los computadores electrónicos digitales». La barrera de los millares se pasó ya en 1989 (Intel 80486: 1.180.000 transistores); actualmente, el Intel Xeon Westmere-EX de 10 núcleos tiene más de dos millardos: 2.500.000.000, y el procesador gráfico Nvidia GK110 más de 7 millardos.

1.2. Niveles y modelos

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

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

Procesadores y niveles de lenguajes

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

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

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

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

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

Si los componentes básicos de un lenguaje de bajo nivel son las **instrucciones**, los de un lenguaje de alto nivel son las **sentencias**. Una sentencia típica del lenguaje C (y también de muchos otros) es la de asignación: «`x = x + y`» significa «calcula la suma de los valores de las variables `x` e `x` y el resultado

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

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. Este es el nivel estudiado (aunque muy superficialmente) en el Tema 1.

Modelos funcionales, estructurales y procesales

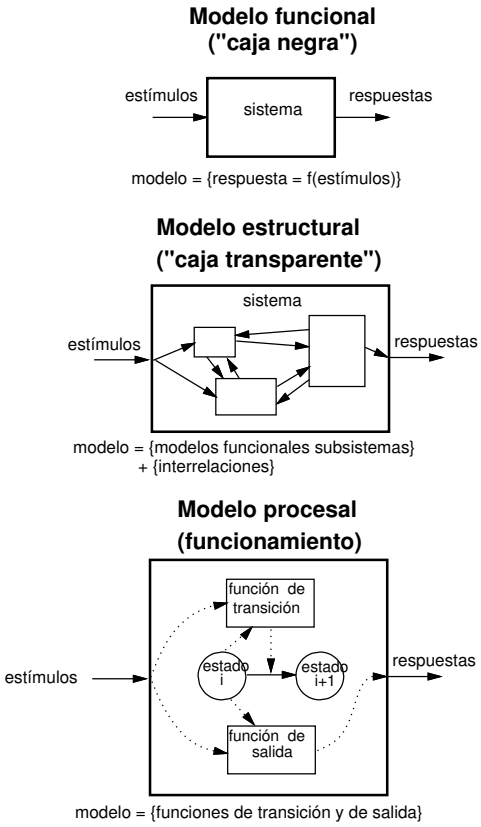


Figura 1.1 Tipos de modelos.

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

Un ejemplo:

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

Regresa usted a la tierra acompañado de su mejor amigo, que, fascinado ante la majestuosidad de un Opel Corsa, quiere aprender a manejarlo. Le describirá mejor el *modelo estructural* (volante, pedales...), y luego le proporcionará, mediante la experiencia, un *modelo funcional* mucho más completo: cómo hay que actuar sobre los distintos elementos para conseguir el comportamiento deseado. Pero su amigo alienígena, ya con el permiso de conducir, resulta ser un individuo inquieto y ávido de conocimientos

y quiere convertirse en un mecánico. Entre otras cosas, necesitará *modelos estructurales y procesales* mucho más detallados que, seguramente, exceden ya sus competencias, y tendrá que matricularlo en un módulo de FP.

Pero volvamos a la cruda realidad de los procesadores hardware.

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

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

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

1.3. Arquitectura de ordenadores

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

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

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

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

Es lo que en el apartado anterior hemos llamado «**ISA**» (Instruction Set Architecture).

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

Arquitectura, implementación y realización

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

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

1.4. Objetivos y contenido del Tema

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

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

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

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

Capítulo 2

La máquina de von Neumann y su evolución

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

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

2.1. Modelo estructural

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

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

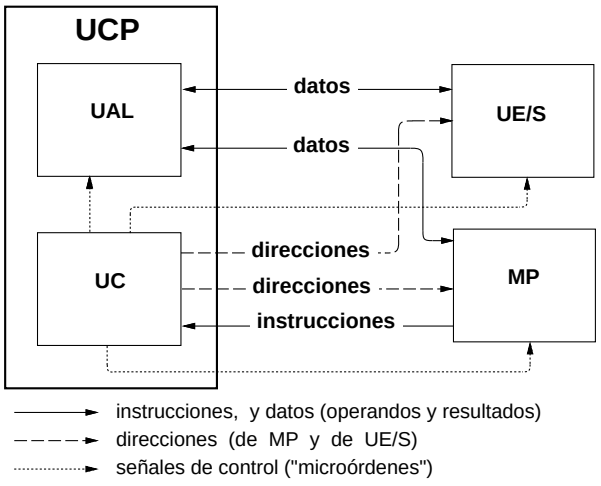


Figura 2.1 La «máquina de von Neumann».

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

Programa almacenado y propósito general

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

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

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

Memoria

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

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

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

Y en otro lugar dice:

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

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

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

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

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

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

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

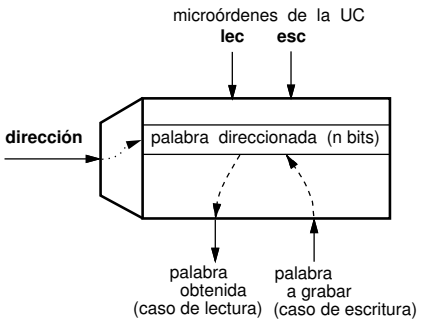


Figura 2.2. Lectura y escritura en una RAM.

Dos propiedades importantes de la MP son:

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

Unidad aritmética y lógica

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

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

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

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

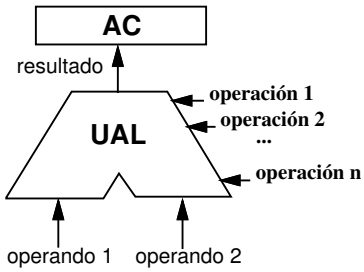


Figura 2.3. Diagrama de la UAL.

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

Unidades de entrada y salida

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

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

Unidad de control

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

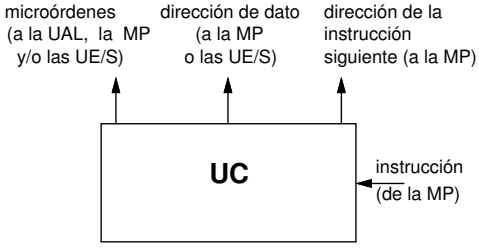


Figura 2.4 Unidad de control.

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

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

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

2.2. Modelo procesal

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

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

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

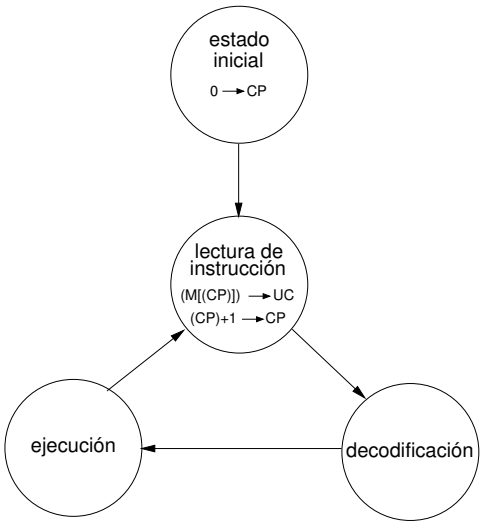


Figura 2.5 Modelo procesal.

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

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

2.3. Modelo funcional

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

Representación de números

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

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

> *«... La principal virtud del sistema binario frente al decimal radica en la mayor sencillez y velocidad con que pueden realizarse las operaciones elementales...»*
>
> *«... Un punto adicional que merece resaltarse es éste: una parte importante de la máquina no es de naturaleza aritmética, sino lógica. Ahora bien, la lógica, al ser un sistema del sí y del no, es fundamentalmente binaria. Por tanto, una disposición binaria de los órganos aritméticos contribuye de manera importante a conseguir una máquina más homogénea, que puede integrarse mejor y ser más eficiente...»*
>
> *«... El único inconveniente del sistema binario desde el punto de vista humano es el problema de la conversión. Sin embargo, como se conoce perfectamente la manera de convertir números de una base a otra, y puesto que esta conversión puede efectuarse totalmente mediante la utilización de los procesos aritméticos habituales, no hay ninguna razón para que el mismo computador no pueda llevar a cabo tal conversión.»*

Formato de instrucciones

> *«... Como la memoria va a tener* $2^{12} = 4.096$ *palabras de cuarenta dígitos* [...] *un número binario de doce dígitos es suficiente para identificar a una posición de palabra...»*
>
> *«... Dado que la mayoría de las operaciones del computador hacen referencia al menos a un número en una posición de la memoria, es razonable adoptar un código en el que doce dígitos binarios de cada orden se asignan a la especificación de una posición de memoria. En aquellas órdenes que no precisan extraer o introducir un número en la memoria, esas posiciones de dígitos no se tendrán en cuenta...»*
>
> *«... Aunque aún no está definitivamente decidido cuántas operaciones se incorporarán (es decir, cuántas órdenes diferentes debe ser capaz de comprender el control), consideraremos por ahora que probablemente serán más de* 2^5*, pero menos de* 2^6*. Por esta razón, es factible asignar seis dígitos binarios para el código de orden. Resulta así que cada orden debe contener dieciocho dígitos binarios, los doce primeros para identificar una posición de memoria y los seis restantes para especificar una operación. Ahora podemos explicar por qué las órdenes se almacenan en la memoria por parejas. Como en este computador se va a utilizar el mismo órgano de memoria para órdenes y para números, es conveniente que ambos tengan la misma longitud. Pero números de dieciocho dígitos binarios no serían suficientemente precisos para los problemas que esta máquina ha de resolver [...] De aquí que sea preferible hacer las palabras suficientemente largas para acomodar dos órdenes...»*
>
> *«... Nuestros números van a tener cuarenta dígitos binarios cada uno. Esto permite que cada orden tenga veinte dígitos binarios: los doce que especifican una posición de memoria y ocho más que especifican una operación (en lugar del mínimo de seis a que nos referíamos más arriba)...»*

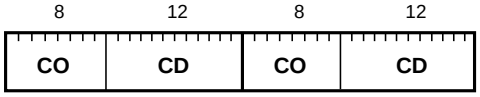


Figura 2.6 Formato de instrucciones.

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

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

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

Programas

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

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

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

2.4. Evolución

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

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

2.5. Modelos estructurales

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

[1] Inicialmente, Gordon Moore estimó el período en un año, pero luego, en 1975, lo amplió a dos. Si se tiene en cuenta que el primer microprocesador, el 4004 de Intel, tenía 2.300 transistores, la ley predice que en 2011, cuarenta años después, se integrarían $2.300 \times 2^{20} \approx 2.400$ millones. El Intel Xeon Westmere-EX de 10 núcleos tiene más de 2.500 millones.

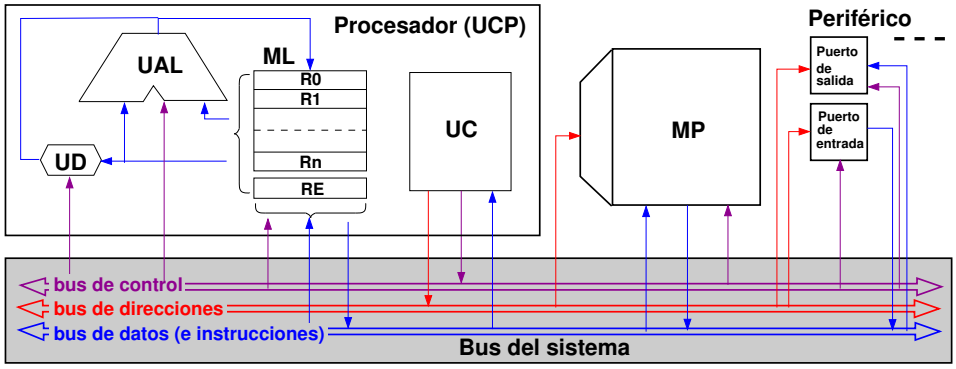


Figura 2.7 Modelo estructural de un sistema basado en procesador

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

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

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

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

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

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

En sistemas muy sencillos, como algunos microcontroladores, éste es un modelo estructural que refleja bastante bien la realidad. Pero incluso en ordenadores personales, el tener un solo bus es muy ineficiente. En efecto, las transferencias entre la UCP y la MP son muy rápidas, pero hay una gran variedad de periféricos, más o menos lentos. Esto conduce a separar los buses y a establecer una jerarquía de buses, como puede apreciarse en la figura 2.8, que es el modelo estructural de un *pecé* del año 2011.

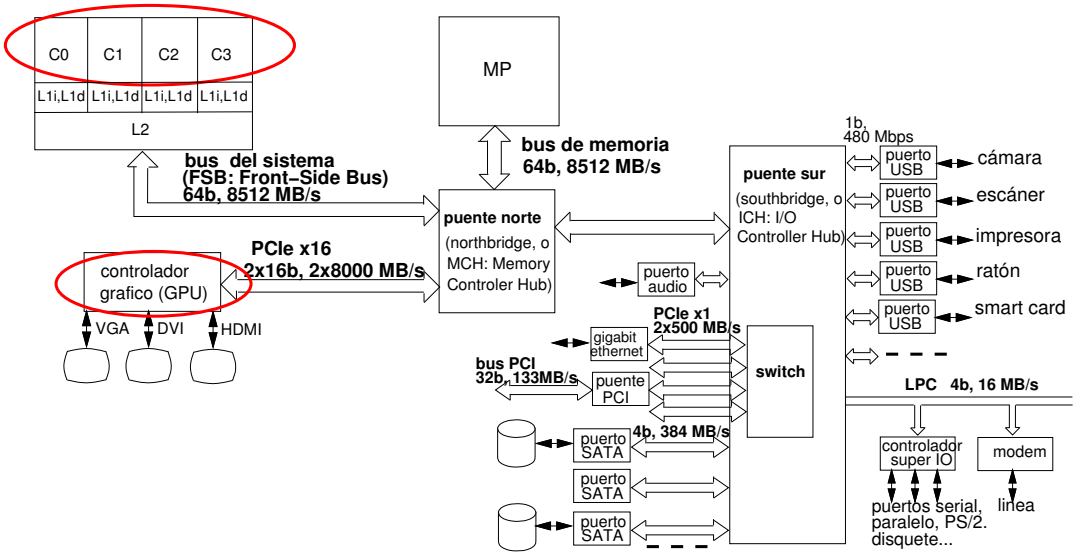


Figura 2.8 Modelo estructural de un ordenador personal.

Sin entrar en detalles, cabe destacar que:

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

2.6. Modelos procesales

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

Encadenamiento *(pipelining)*

Uno de los inventos que más ha contribuido a aumentar la velocidad de los procesadores es el **encadenamiento** (en inglés, *pipelining*). Es, conceptualmente, la misma idea de las cadenas de producción:

si la fabricación de un producto se puede descomponer en etapas secuenciales e independientes y si se dispone de un operador para cada etapa, finalizada la etapa 1 del producto 1 se puede pasar a la etapa 2 de ese producto y, *simultáneamente* con ésta, llevar a cabo la etapa 1 del producto 2, y así sucesivamente.

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

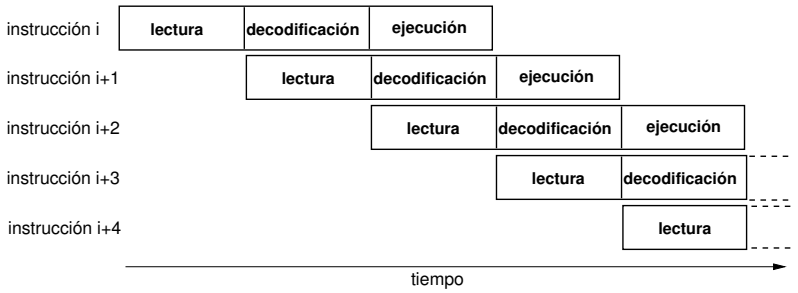


Figura 2.9 Una cadena de tres etapas.

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

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

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

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

Memorias ocultas *(caches)*

Una expresión ya clásica en arquitectura de ordenadores es la de «cuello de botella de von Neumann». Se refiere a que, desde los primeros tiempos, los procesadores se implementan con tecnologías mucho más rápidas que las de la MP. Como toda instrucción tiene que ser leída de la MP, el tiempo se acceso se convierte en el factor que más limita el rendimiento. En este caso, de poco serviría el encadenamiento. Por ejemplo, si modificamos la figura 2.9 para el caso de que la etapa de lectura tenga una duración cien veces superior a las de descodificación y ejecución, la velocidad del procesador, medida en instrucciones por segundo, no se multiplicaría por tres, sino por un número ligeramente superior a uno[3].

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

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

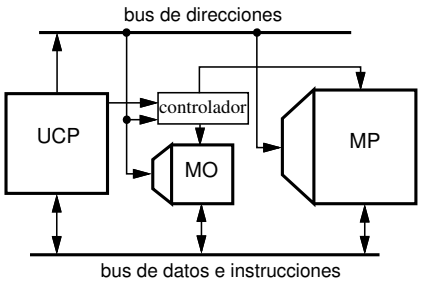


Figura 2.10. Memoria oculta *(cache)*.

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

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

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

En las UCP que contienen varios procesadores («cores») la L2 (y, en su caso, la L3) es común para todos, pero cada procesador tiene su propia L1 y, además está formada por dos (es una arquitectura Harvard, apartado 2.1): L1d es la MO de datos y L1i es la MO de instrucciones (figura 2.8), lo que, como hemos dicho antes, reduce los conflictos en el encadenamiento.

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

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

Esto significa (aunque la salida del programa `lscpu` no es muy explícita) que el microprocesador contiene cuatro procesadores[4], cada uno de ellos acompañado de una MO de datos y una MO de instrucciones, de 32 KiB cada una, y que hay una MO de nivel 2 de 3 MiB común a los cuatro (es el ordenador de la figura 2.8).

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

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

2.7. Modelos funcionales

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

Direcciones y contenidos de la MP

En la máquina de von Neumann las palabras tenían cuarenta bits, lo que se justificaba, como hemos leído, porque en cada una podía representarse un número entero binario con suficiente precisión. Y en cada acceso a la MP se leía o se escribía una palabra completa, que podía contener o bien un número o bien dos instrucciones. Pero pronto se vio la necesidad de almacenar caracteres, se elaboraron varios códigos para su representación (Tema 2) y se consideró conveniente poder acceder no ya a una palabra (que podía contener varios caracteres codificados), sino a un carácter. Algunos códigos primitivos eran de seis bits y se llamó **byte** a cada conjunto de seis bits. Más tarde se impuso el código ASCII de siete bits y se redefinió el byte como un conjunto de ocho bits (el octavo, en principio, era un bit de paridad), por lo que también se suele llamar **octeto**.

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

En una arquitectura de dieciséis bits, la palabra de dirección d está formada por los bytes de direcciones d y $d + 1$. Si es de treinta y dos bits, por los cuatro bytes de d a $d + 3$, etc. Y como hemos visto en el Tema 2 (apartado 1.4), el convenio puede ser *extremista mayor* (el byte de dirección más baja contiene los bits *más* significativos de la palabra) o el contrario, *extremista menor*.

Junto con el convenio de almacenamiento, hay otra variante: el alineamiento de las palabras. En

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

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

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

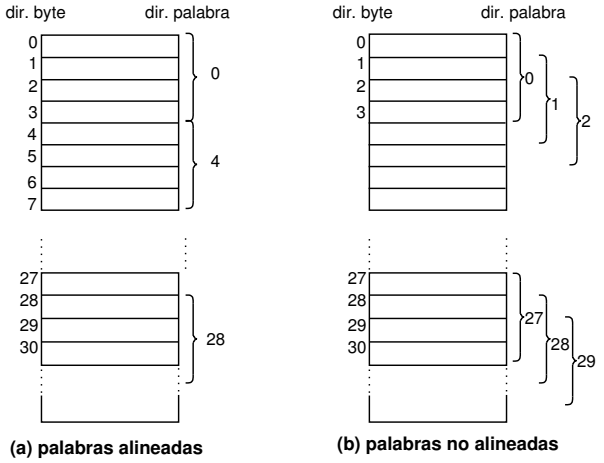


Figura 2.11 Direcciones alineadas y no alineadas en una arquitectura de 32 bits.

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

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

Formatos y repertorios de instrucciones

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

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

complicada, y se inventó una técnica llamada **microprogramación**: la unidad de control se convierte en una especie de «procesador dentro del procesador», con su propia memoria que alberga microprogramas, donde cada microprograma es un algoritmo para interpretar y ejecutar una instrucción. (De ahí el «nivel de micromáquina»).

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

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

CISC y RISC

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

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

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

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

Modos de direccionamiento

En la máquina de von Neumann, las instrucciones que hacen referencia a memoria (almacenar el acumulador en una dirección de la MP, sumar al acumulador el contenido de una dirección, bifurcar a una dirección...) indican la dirección de memoria *directamente* en el campo CD. Hay diversos mecanismos mediante los cuales la dirección real, o **dirección efectiva** no se obtiene directamente de CD, sino combinándolo con otras cosas, o, incluso, prescindiendo de CD, lo que facilita la programación y hace más eficiente al procesador. 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. Esto es muy útil para recorrer zonas de la MP. Por ejemplo, para sumar los componentes de un vector que están en direcciones consecutivas.

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

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

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

Observe que:

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

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

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

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

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

Subprogramas

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

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

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

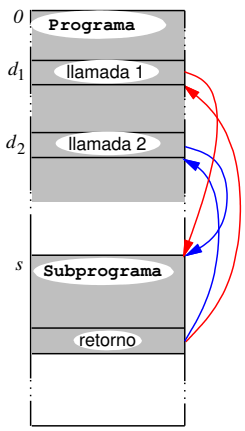


Figura 2.12. Llamadas y retornos.

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

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

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

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

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

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

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

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

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

Pilas

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

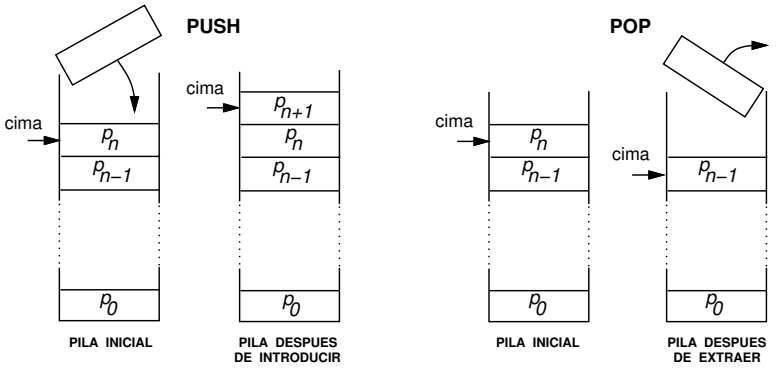


Figura 2.13 Memoria de pila.

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

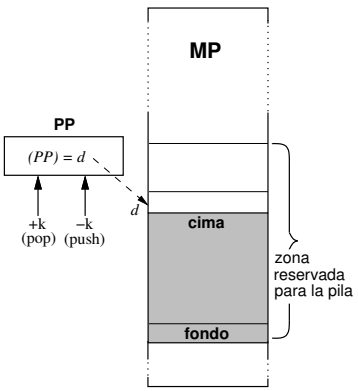


Figura 2.14 Una pila en una RAM.

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

La ejecución de la instrucción de retorno se hace con un *pop* para extraer de la pila la dirección de retorno y ponerla en el contador de programa. La operación *pop* consiste en incrementar el contenido de PP y luego leer de la dirección apuntada por PP (si el convenio es que PP apunta a la cima, se invierten los pasos).

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

Repare en tres detalles de esta solución:

- Las operaciones *push* y *pop* mencionadas son transparentes al programador. Es decir, éste solamente tiene que saber que existe una instrucción (en ensamblador se suele llamar `CALL`) que le permite transferir el control a una dirección donde comienza un subprograma, y que la última instrucción de este subprograma debe ser la de retorno (en ensamblador, `RET`).

- El «**anidamiento**» de subprogramas (que un subprograma llame a otro, y éste a otro... o, incluso, que un subprograma se llame a sí mismo de manera recursiva) no plantea ningún problema: las direcciones de retorno se irán apilando y recuperando en el orden adecuado. (El único problema práctico radica en que la pila no es infinita).

- La pila puede utilizarse también para transmitir parámetros entre el programa y el subprograma: antes de `CALL`, el subprograma utiliza instrucciones «`PUSH`» para introducirlos en la pila, y el subprograma puede acceder a ellos con un modo de direccionamiento que sume una distancia al puntero de pila.

2.8. Comunicaciones con los periféricos e interrupciones

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

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

- **Espacios de direccionamiento independientes**: las direcciones de entrada/salida no tienen relación alguna con las direcciones de la memoria. La dirección *d* puede referirse a una posición de memoria o a un puerto. En estos procesadores son necesarias instrucciones específicas para la entrada/salida.

- **Espacio de direccionamiento compartido**: ciertas direcciones generadas por el procesador no corresponden a la memoria, sino a puertos. Por tanto, el acceso a estos puertos no requiere instrucciones especiales, son las mismas que permiten el acceso a la memoria. Se dice que la entrada/salida está *«memory mapped»*.

La forma de programar estas operaciones depende de 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», Tema 2, apartado 5.1):

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

- **Periféricos de bloques**: son aquellos cuya tasa de transferencia es comparable a la velocidad del procesador (disco, pantalla gráfica, USB, controlador ethernet, etc.), y se hace necesario que el

controlador del periférico se comunique directamente con la memoria, leyendo o escribiendo en cada operación no ya un byte, sino un bloque de bytes. Es la técnica de **DMA** (acceso directo a memoria, *Direct Memory Access*).

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

Interrupciones

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

El tratamiento de las interrupciones es uno de los asuntos más fascinantes y complejos de la arquitectura de procesadores y de la ingeniería en general. Fascinante, porque gracias a él dos subsistemas de naturaleza completamente distinta, el hardware (material) y el software (intangible), trabajan conjuntamente, y de esa simbiosis 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 MP.
- Atender a una interrupción implica abandonar temporalmente el programa que se está ejecutando para pasar a la rutina de servicio. Y cuando finaliza la ejecución de la rutina de servicio, el procesador debe volver al programa interrumpido para continuar con su ejecución.
- Esto recuerda a los subprogramas, pero hay una diferencia fundamental: la interrupción puede no tener ninguna relación con el programa, y puede aparecer en cualquier momento. Por tanto, no basta con guardar automáticamente la dirección de retorno (dirección de la instrucción siguiente a aquella en la que se produjo la atención a la interrupción), también hay que guardar los contenidos del registro de estado y de los registros que use la rutina (para que el programa continúe como si nada hubiese pasado). Naturalmente, al volver al programa interrumpido deben recuperarse.
- Como puede haber muchas causas que provocan la interrupción, el mecanismo debe incluir la identificación para poder ejecutar la rutina de servicio que corresponda. En algunos diseños esta identificación se hace por software, en otros por hardware, o, lo que es más frecuente, mediante una combinación de ambos.

2.9. Conclusión

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

Capítulo 3

El procesador BRM

ARM (Advanced RISC Machines) es el nombre de una familia de procesadores, y también el de una compañía, ARM Holdings plc. A diferencia de otras, la principal actividad de esta empresa no consiste en fabricar microprocesadores, sino en «licenciar» la propiedad intelectual de sus productos para que otras empresas puedan incorporarlos en sus diseños. Por este motivo, aunque los procesadores ARM sean los más extendidos, la marca es menos conocida que otras. En 2010 se estimaba en 25 millardos el número total acumulado de procesadores fabricados con licencia de ARM, a los que se sumaban 7,9 millardos en 2011 y 8,7 millardos en 2012[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 digitales, 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 este Tema, describirlas por completo. Es por eso que nos hemos quedado con los elementos esenciales, esquematizando de tal modo la arquitectura que podemos hablar de un «ARM simplificado», al que hemos bautizado como «BRM» (Basic RISC Machine).

BRM es totalmente compatible con los procesadores ARM. De hecho, todos los programas que se presentan en el capítulo siguiente se han probado con un simulador de ARM 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 1.2). Hay varios lenguajes para la arquitectura ARM, y en el capítulo siguiente seguiremos la sintaxis del *ensamblador GNU*, que es también el que se utiliza en el simulador *ARMSim#* que nos servirá para practicar. De todas formas, en este capítulo no hay programas, solo instrucciones sueltas, y los convenios (nemónicos para códigos de operación y registros) son los mismos en todos los ensambladores.

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

3.1. Modelo estructural

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

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

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

Registros de comunicación con el exterior

En la parte superior izquierda, el *registro de direcciones*, RA, 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 MP y los puertos de entrada/salida. Es decir, algunas direcciones corresponden a puertos. Pero estas direcciones no están fijadas: será el diseñador de los circuitos externos que conectan la MP y los periféricos a los buses el que, mediante circuitos descodificadores, determine qué direcciones no son de la MP sino de puertos.

Abajo a la izquierda hay dos **registros de datos**. En RDL se introducen los datos que se leen del exterior, y en RDE los que se van a escribir.

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

Cuando la UC 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 3.5), pone la dirección en el registro RA, si es una salida de datos (escritura) pone también el dato en RDE, y activa las señales **req**, **r/w** y **size**. Si ha sido una petición de lectura (entrada de dato o de instrucción)

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

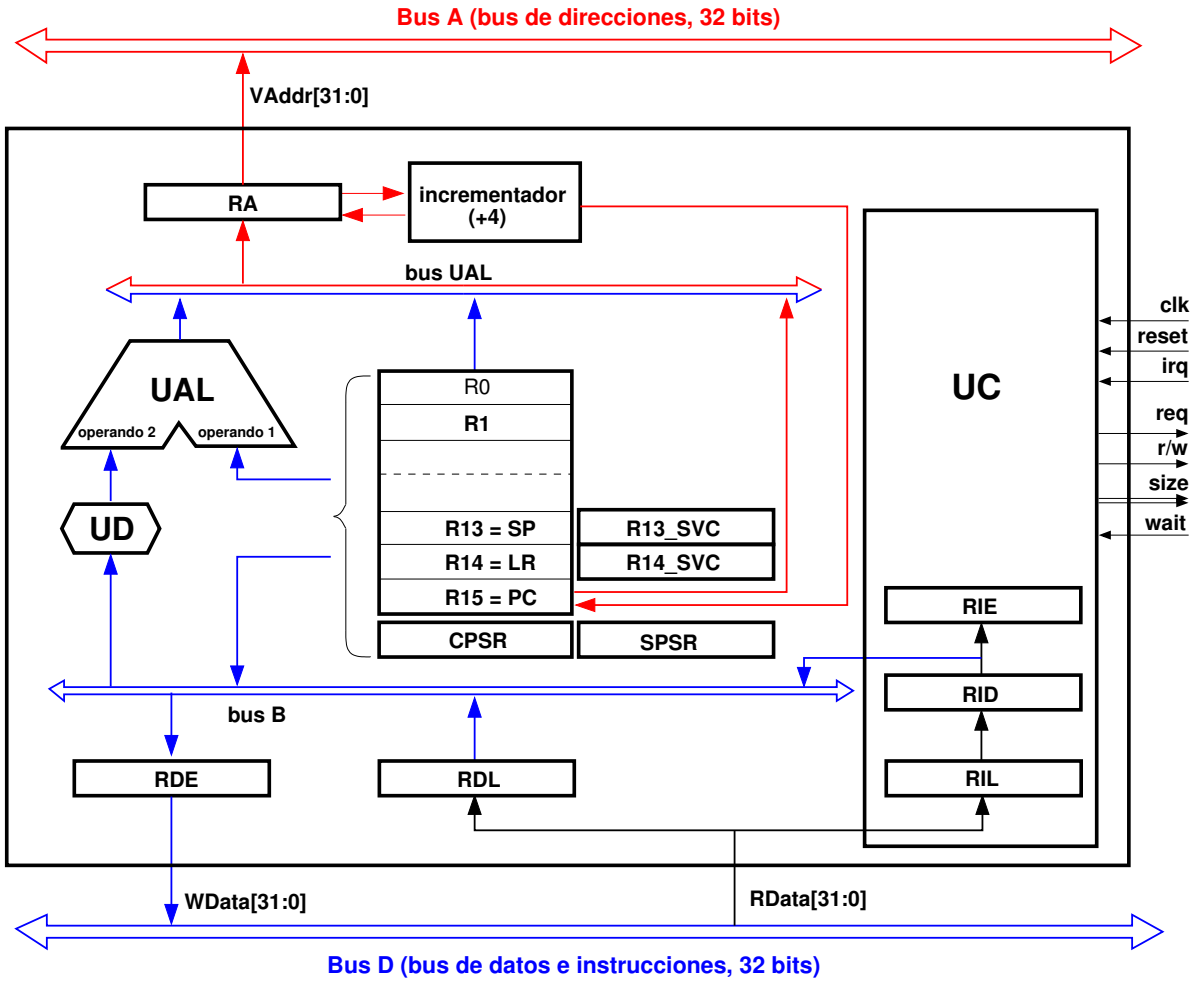


Figura 3.1 Modelo estructural de BRM.

después de un ciclo de reloj la UC introduce el contenido del bus D en RDL o RIL, según proceda.

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

Memoria local y registros especiales

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

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

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

Modos usuario y supervisor

Los **modos de funcionamiento** de un procesador *no tienen nada que ver* con los **modos de direccionamiento** de la memoria, que hemos explicado de manera general en el apartado 2.7 y que veremos para el caso particular de BRM en el apartado 3.5.

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 MP distinta para la pila. El procesador, según el modo en que se encuentre, elije uno u otro registro. Por ejemplo, ante una instrucción que diga «sacar de la pila *(pop)* y llevar a R5», el procesador, si está en modo usuario, copiará en R5 el contenido en la dirección apuntada por R13 (e incrementará R13), mientras que si está en modo supervisor lo hará con la dirección apuntada por R13_SVC (e incrementará R13_SVC). Con el registro de enlace, R14 pasa algo parecido: la instrucción «bifurca y enlaza» (`BL`) guarda la dirección de retorno en R14 si está en modo usuario, o en R14_SVC si está en modo supervisor.

Registro de estado

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

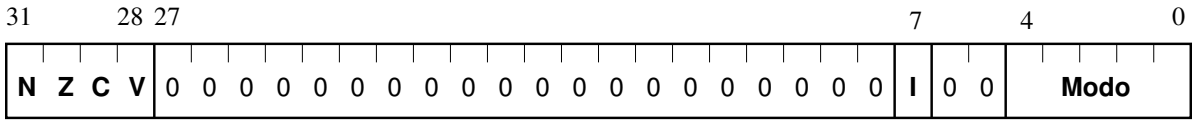


Figura 3.2 Registro de estado, CPSR.

Las informaciones que contiene CPSR son (figura 3.2):

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

Los demás bits de CPSR tienen siempre el valor 0.

[4]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.

3.2. Modelo procesal

La UC sigue el modelo procesal de tres fases de la figura 2.5 con encadenamiento (figura 2.9). Pero hay que matizar un detalle con respecto a esas figuras: como las instrucciones ocupan cuatro bytes, si una instrucción está en la dirección i, la siguiente no está en $i + 1$, sino en $i + 4$. Consecuentemente, el registro PC no se incrementa de 1 en 1, sino de 4 en 4.

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

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

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

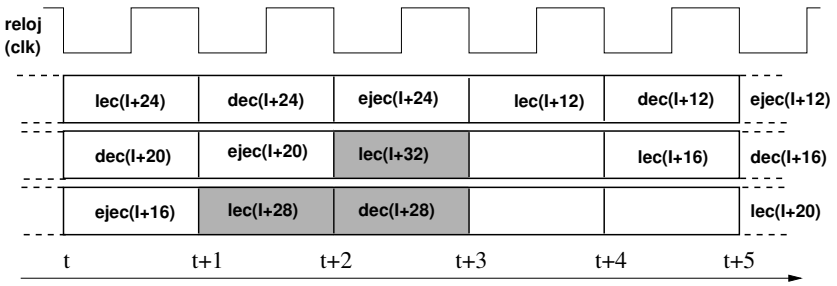


Figura 3.3 Bifurcación en la cadena.

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

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

3.3. Modelo funcional

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

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

Representación de instrucciones

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

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

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

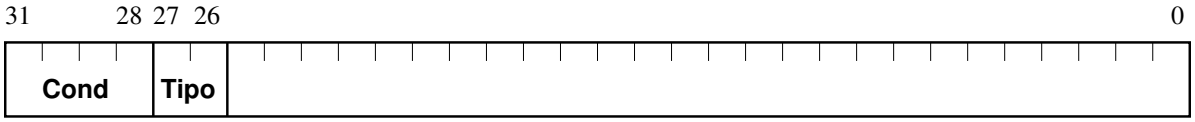


Figura 3.4 Formato de instrucciones: campos «condición» y «tipo».

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

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

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

  Tipo 10: Instrucciones de bifurcación (dos instrucciones).

  Tipo 11: Instrucción de interrupción de programa.

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

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

Tabla 3.1 Códigos de condición.

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

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

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

3.4. Instrucciones de procesamiento y de movimiento

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

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

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

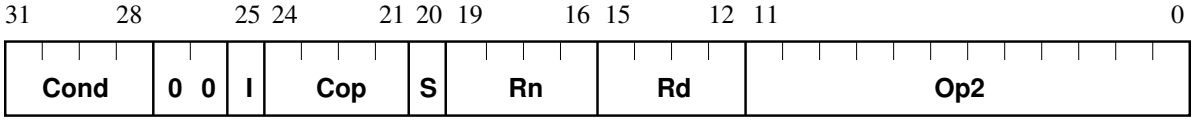


Figura 3.5 Formato de las instrucciones de procesamiento y movimiento.

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

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

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

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

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

Operando inmediato

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

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

Para empezar, la mayoría de las constantes que se usan en un programa son números pequeños. Según el formato de la figura 3.5, disponemos de doce bits 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.

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

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

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

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

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

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

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

¿Y si el número fuese mayor que 4.095? El convenio de BRM es que solo ocho de los doce bits del campo Op2 son para el número (lo que, en principio, parece empeorar las cosas), y los cuatro restantes dan la magnitud de una *rotación a la derecha* de ese número multiplicada por 2 (figura 3.6). Conviene volver a mirar la figura 3.1: la entrada 2 a la UAL (en este caso, procedente del registro RID a través del bus B) pasa por una unidad de desplazamiento, y una de las posibles operaciones en esta unidad es la de rotación a la derecha (ROR, figura 3.2 del Tema 2). Los ocho bits del campo «inmed_8» se extienden a 32 bits con ceros a la izquierda y se les aplica una rotación a la derecha de tantos bits como indique el campo «rot» multiplicado por dos. Si «rot» = 0 el resultado está comprendido entre 0 y el valor codificado en los ocho bits de «inmed_8» (máximo: $2^8 = 255$). Con otros valores de «rot» se pueden obtener valores de hasta 32 bits.

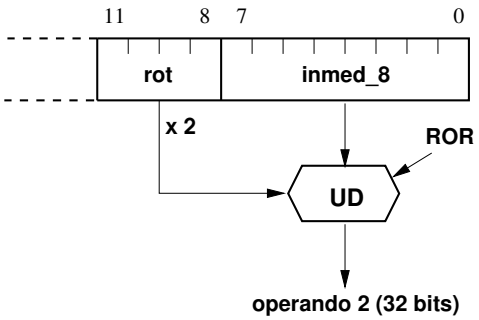


Figura 3.6 Operando inmediato.

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

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

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

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

Probemos con un número muy grande:

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

En efecto, como «rot» = 4, el número de rotaciones a la izquierda es $32-4\times2 = 24$. 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 4.3.

En los ejemplos hemos utilizado `MOV` y R0. Pero todo es aplicable al segundo operando de cualquiera de las instrucciones de este tipo y a cualquier registro. (Teniendo en cuenta que R13, R14 y R15 tienen funciones especiales; 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`[7]. Pero ¿para meter −5 en un registro? Probemos:

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

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

Operando en registro

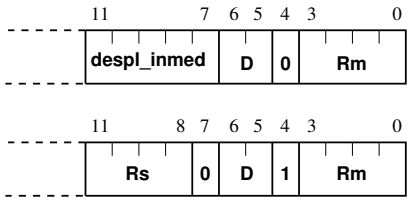


Figura 3.7 Operando en registro.

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

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

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

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

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

Ejemplos con operando inmediato:

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

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

Ejemplos con operando en registro:

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

Movimientos con los registros de estado

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

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

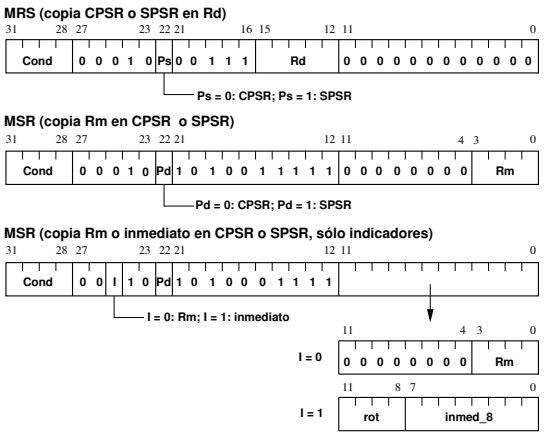


Figura 3.8 Instrucciones `MRS` y `MSR`

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

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

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

Ejemplos:

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

3.5. Instrucciones de transferencia de datos

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

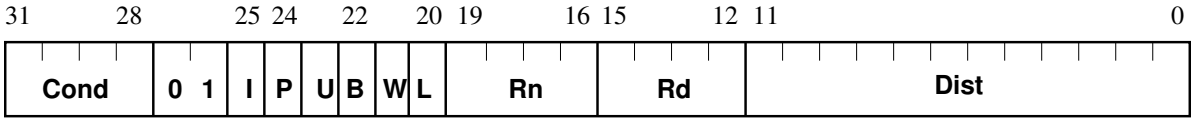


Figura 3.9 Formato de las instrucciones de transferencia de datos.

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

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

Tabla 3.3 Instrucciones de transferencia de datos.

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

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

Resultan así **cuatro modos de direccionamiento:**

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

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

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

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

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

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

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

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

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

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

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

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

Ejemplos:

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

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

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

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

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

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

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

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

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

Ejemplos:

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

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

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

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

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

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

Ejemplos:

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

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

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

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

3.6. Instrucciones de transferencia de control

Puesto que el contador de programa es uno más 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 solo se aplica cuando la dirección de destino está muy alejada de la instrucción, y también para retornar de una interrupción, como veremos en el siguiente apartado. Normalmente se utilizan instrucciones de bifurcación con direccionamiento relativo.

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

Instrucciones de bifurcación

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

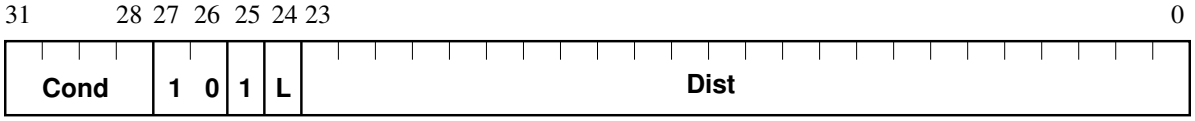


Figura 3.10 Formato de las instrucciones de bifurcación.

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

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

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

Instrucción de interrupción de programa

Esta instrucción, de código ensamblador `SWI` («software interrupt») o `SVC` («supervisor call»), tiene un formato muy sencillo, el de la figura 3.11.

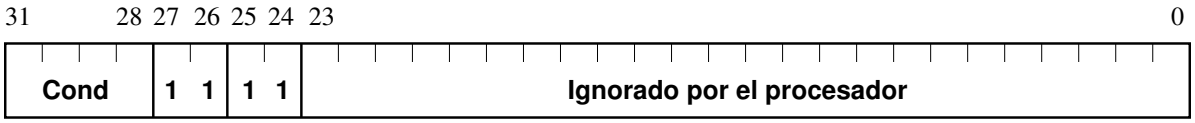


Figura 3.11 Formato de la instrucción `SWI`.

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

3.7. Comunicaciones con los periféricos e interrupciones

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

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

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

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

Todas estas operaciones las hace automáticamente el hardware.

Causas y vectores de interrupción

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

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

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

Tabla 3.4 Tabla de vectores de interrupción.

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

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

Retorno de interrupción

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

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

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

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

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

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

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

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

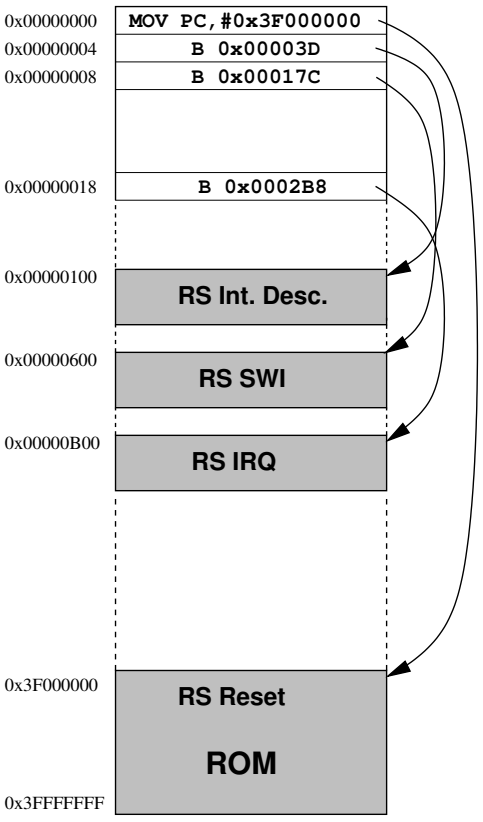


Figura 3.12. Mapa de memoria de vectores y rutinas de servicio.

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

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

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

Capítulo 4

Programacion de BRM

Ya hemos dicho en el apartado 1.2 que un lenguaje ensamblador, a diferencia de uno de alto nivel, está ligado a la arquitectura (ISA) de un procesador. Pero no solo es que cada procesador tenga su lenguaje, es que para un mismo procesador suele haber distintos convenios sintácticos para codificar las instrucciones de máquina en ensamblador.

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

Los listados se han obtenido con un **ensamblador cruzado** (apartado A.3): un programa ensamblador que se ejecuta bajo un sistema operativo (Windows, Linux, MacOS...) en una arquitectura (x86 en este caso) y genera código binario para otra (ARM en este caso). Los mismos programas se pueden introducir en ARMSim#, que se encargará de ensamblarlos y de simular la ejecución. También se pueden ensamblar directamente y ejecutar en un sistema con arquitectura ARM, como se explica en el apartado 4.10. 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.

4.1. Primer ejemplo

Construimos un programa en lenguaje ensamblador escribiendo secuencialmente instrucciones con los convenios sintácticos que hemos ido avanzando en el capítulo anterior. Por ejemplo, para sumar 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 (Tema 2, apartado 5.5).

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

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

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

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

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

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

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

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

[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 «`;`».

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 A.3 se explica cómo hacerlo), obtenemos lo que muestra el programa 4.1.

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

Programa 4.1 Comentarios y directivas `equ` y `end`.

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

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

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

4.2. Etiquetas y bucles

En el apartado 2.7, al hablar de modos de direccionamiento, pusimos, en pseudocódigo, un ejemplo de bucle con una instrucción de bifurcación condicionada, e, incluso, adelantamos algo de su codificación en ensamblador. Como ese ejemplo requiere tener unos datos guardados en memoria, y aún

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

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:

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

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

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

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

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

Programa 4.2 Ejemplo de bucle.

Fíjese en cuatro detalles importantes:

- Aparte del símbolo «`Num`», que se iguala con el valor 233, en el programa se definen tres símbolos, que se llaman **etiquetas**. Para no confundirla con otro tipo de símbolo, la etiqueta, en su definición, debe terminar con «`:`», y toma el valor de la dirección de la instrucción a la que

acompaña: «bucle» tiene el valor 0x10, «si» 0x2C, y «no» 0x30. De este modo, al escribir en ensamblador nos podemos despreocupar de valores numéricos de direcciones de memoria.

- Debería usted comprobar cómo ha traducido el ensamblador las instrucciones de bifurcación:
  - En la que está en la dirección 0x18 (bifurca a «si» si son iguales) ha puesto «3» en el campo «Dist» (figura 3.10). Recuerde que la dirección efectiva se calcula (en tiempo de ejecución) como la suma de la dirección de la instrucción más 8 más 4×Dist. En este caso, como 0x18 = 24, resulta $24 + 8 + 4 \times 3 = 34 =$ 0x2C, que es el valor del símbolo «si».
  - En la que está en 0x1C (bifurca a «no» si mayor) ha puesto la misma distancia, 3. En efecto, la «distancia» a «no» es igual que la anterior.
  - En la que está en 0x28 (40 decimal) (bifurca a «bucle») ha puesto Dist = FFFFF8, que, al interpretarse como número negativo en complemento a dos, representa «-8». DE = $40 + 8 + 4 \times (-8) = 16 =$ 0x10, dirección de «bucle».
- Las direcciones efectivas se calculan en tiempo de ejecución, es decir, con el programa ya cargado en memoria. Venimos suponiendo que la primera dirección del programa es la 0 (el ensamblador no puede saber en dónde se cargará finalmente). Pero si se carga, por ejemplo, a partir de la dirección 0x1000 (como hace ARMSim#) todo funciona igual: `b bucle` (en binario, por supuesto) no estará en 0x28, sino en 0x1028, y en la ejecución bifurcará a 0x1020, donde estará `add r0,r1,r2`.
- Una desilusión provisional: la utilidad de este programa es muy limitada, debido a la forma de introducir el número a comprobar en un registro: `mov r3, #Num`. En efecto, ya vimos en el apartado 3.4 que el operando inmediato tiene que ser o bien menor que 256 o bien poderse obtener mediante rotación de un máximo de ocho bits.

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

4.3. El *pool* de literales

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

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

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

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

y cargándolo en el simulador. Obtendrá este resultado:

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

Veamos lo que ha hecho al ensamblar:

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

En resumen, utilizando la *pseudoinstrucción* `LDR` podemos poner cualquier constante (entre -2^{-31} y $2^{31} - 1$) y el ensamblador se ocupa, si puede, de generar una `MOV` (o una `MVN`), y si no, de ver si ya

tiene la constante creada, de crearla si es necesario y de generar la *instrucción* `LDR` adecuada. Al final, tras el código objeto, inserta el «pool de literales», que contiene todas las constantes generadas.

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

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

4.4. Más directivas

Informaciones para el montador

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

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

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

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

Introducción de datos en la memoria

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

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

- El listado se ha obtenido con el ensamblador GNU, que pone tanto la sección de código (`.text`) como la de datos a partir de la dirección 0. Será luego el montador el que las coloque una tras otra. Sin embargo, el simulador ARMSim# hace también el montaje y carga el resultado en la memoria a partir de la dirección 0x1000, de modo que el contenido de «palabra1» queda en la dirección 0x1024.

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

Programa 4.3 Intercambio de palabras.

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`	

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

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

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

Dos comentarios:

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

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

Programa 4.4 Suma de las componentes de un vector.

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

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

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

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

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

Programa 4.5 Copia una cadena de caracteres

4.5. Menos bifurcaciones

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

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

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

Pero condicionando la `RSB` podemos prescindir de la bifurcación:

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

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

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

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

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

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

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

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

4.6. Subprogramas

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

Con subprogramas se pueden implementar operaciones comunes que no están previstas en el repertorio de instrucciones. ¿Ha pensado usted en cómo resolver el problema de la suma de bytes cuando alguno de ellos es negativo (programa 4.4)? El problema se produce porque, por ejemplo, la representación de −10 en un byte con complemento a 2 es 0xF6. Y al cargarlo con la instrucción LDRB en un registro, el contenido de ese registro resulta ser 0x000000F6, que al interpretarlo como entero con signo, es +246.[3]

[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.

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

Programa 4.6 Subprograma MCD y llamadas.

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

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

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

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

Paso de valor y paso de referencia

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

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

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

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

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

Paso de parámetros por la pila

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

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

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

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

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

Operación	Instrucción
(MP[(sp)+4]) → R0	`ldr r0,[sp,#4]`
(MP[(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 4.7 es otra versión del subprograma MCD y de las llamadas en la que los parámetros y el resultado se pasan por la pila. Observe que ahora el subprograma puede utilizar cualquier pareja de registros, pero los salva y luego los recupera por si el programa invocador los está usando para otra cosa (aquí no es el caso, pero así el subprograma es válido para cualquier invocador, que solo tiene que tener en cuenta que debe introducir en la pila los parámetros y recuperar el resultado de la misma).

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

Programa 4.7 Subprograma MCD y llamadas con paso por pila

Anidamiento

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

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

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

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

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

Programa 4.8 Programa principal para probar MCD3

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

Programa 4.9 Subprogramas MCD3 y MCD

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

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

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

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

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

4.7. Módulos

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

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

Para que el ensamblador entienda que ciertos símbolos están definidos en otros módulos se 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. Solo hay que escribir, en tres ficheros diferentes, los tres módulos «`pruebaMCD3`» (programa 4.8), «`MCD3`» y «`MCD`» (programa 4.9), añadiendo las oportunas `global`. Le invitamos a hacerlo como ejercicio.

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

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

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

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

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

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

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

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

Programa 4.12 Subprograma para multiplicar

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

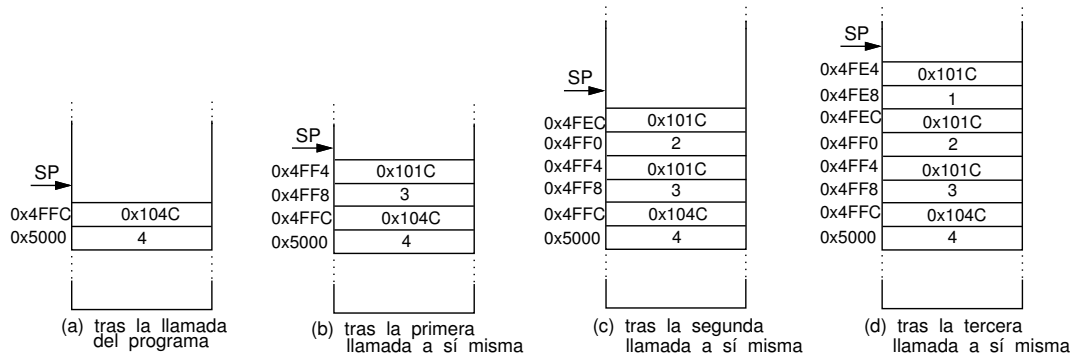


Figura 4.1 Estados de la pila en el cálculo del factorial.

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

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

4.8. Comunicaciones con los periféricos e interrupciones

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

Espera activa

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

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

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

Cuando se pulsa una tecla, el controlador 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 3.2). En total, cada paso por el bucle se ejecuta en un tiempo $t_B = 5/10^9$ segundos. Por otra parte, imaginemos al campeón o campeona mundial de mecanografía tecleando con una velocidad de 750 pulsaciones por minuto[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

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

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

Interrupciones de periféricos de caracteres

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

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

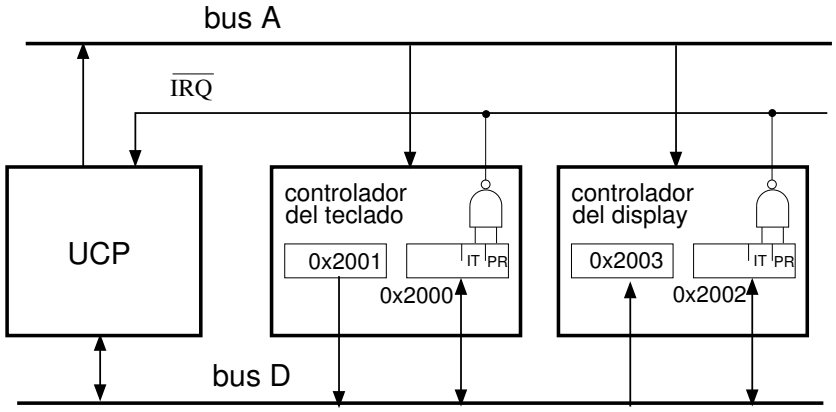


Figura 4.2 Conexión de controladores.

El procesador y los controladores están conectados como indica la figura 4.2. El símbolo sobre los bits PR e IT de cada controlador representa un circuito que realiza la operación lógica NAND. Es decir, cuando ambos bits son «1» (y solo en ese caso) la salida de ese circuito es «0», lo que activa a la línea $\overline{\text{IRQ}}$, que es la entrada de interrupciones externas al procesador.[7] Inicialmente, el hardware, antes de ejecutarse el programa de arranque en ROM (figura 3.12), inhibe las interrupciones de los controladores (IT = 0) y el permiso general de interrupciones (I = 0 en el CPSR, figura 3.2)

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

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

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

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

Rutinas de servicio de periféricos de caracteres

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

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

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

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

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

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

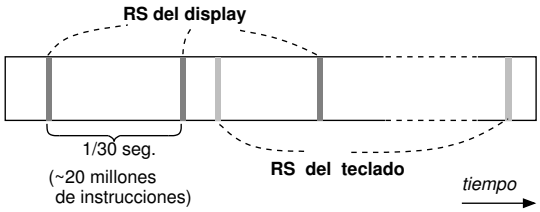


Figura 4.3 Tiempos de ejecución de las rutinas.

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

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

Consulta de interrupciones

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

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

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

Periféricos de bloques

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

Por esta razón, para periféricos rápidos, o periféricos de bloques, se sigue la técnica de acceso directo a memoria, o **DMA** (apartado 2.8). Para realizar una transferencia, el programa inicializa al controlador de DMA con los datos necesarios para que 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 3.1) y se encarga de transferir los bytes *directamente* con la memoria, sin pasar por la CPU.

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

4.9. Software del sistema y software de aplicaciones

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

El software de aplicaciones, pues, hace uso de esos servicios, cuya implementación es «software del sistema». Esto ya lo sabíamos del Tema 1. Lo que quizás pueda usted entender mejor en este momento, en el que hemos descendido al nivel de máquina convencional, es el mecanismo básico que permite esa implementación. Y este mecanismo se apoya en una sola instrucción, la de *interrupción de programa*, `SWI`, también llamada `SVC`.[8] Con esta instrucción los programas de aplicación hacen **llamadas al sistema**.

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

[8] En ensambladores para otros procesadores se llama `TRAP`, o `INT`, pero esencialmente tiene la misma función.

4.10. 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 4.13, escrito en el ensamblador GNU para ARM, hace eso mismo.

```
                .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 4.13 Programa «Hola mundo».

En un lenguaje de alto nivel, la llamada para escribir en un fichero es una función de nombre «`write`» que tiene como parámetros un número entero (el «descriptor del fichero», que en el caso de la salida estándar es 1), un puntero a la zona de memoria que contiene los datos a escribir y el número de bytes que ocupan éstos. En ensamblador, estos parámetros se pasan por registros: R0, R1 y R2, respectivamente. El número que identifica a `write` es 4, y se pasa por R7. Esto explica las cuatro primeras instrucciones que, *junto con `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 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 llamada (que en alto nivel se llama «`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 hace activo a otro proceso.

Saludo

Si el descriptor de la salida estándar es 1, el de la entrada estándar (el teclado) es 0. El programa 4.14 pregunta al usuario por su nombre, lo lee, y le saluda con ese nombre. (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 A.1) se pueden adaptar para que realicen los cálculos con datos del teclado.

Ahora bien, esas llamadas sirven para leer o escribir *cadenas de caracteres*. Si, por ejemplo, adaptamos el programa 4.2 para que diga «Dame un número y te digo si es de Fibonacci» y luego lea, y si escribimos «233», lo que se almacena en memoria es la *cadena de caracteres «233»*, 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 sencilla (recuerde el ejercicio planteado en clase, transparencia 36 del Tema 2), pero laboriosa (especialmente si, como debe ser, se incluye la detección de errores: caracteres no numéricos, etc.). 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»[9] o «BeagleBone Black»[10]) 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 4.4 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).

[9] `http://www.raspberrypi.org`
[10] `http://beagleboard.org/`

```
                .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 4.14 Programa «Saludo».

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 (apartado A.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`[11]

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, como el que estudiará en la asignatura «Análisis de diseño de software».
- 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 harramientas (`as`, `ld`, etc.).
- 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 (apartado A.3) sobre su uso;
  - el dispositivo con Android modificado para tener permisos de administrador y con una aplicación de terminal (por ejemplo, «Terminal emulator»);
  - una conexión por cable USB (o por alguna aplicación wifi) para transferir el programa ejecutable del ordenador al dispositivo.

[11] 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 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 A.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 4.4 muestra esas operaciones realizadas en el terminal (se ha ocultado el teclado virtual para que se vean todas las líneas), y en la figura 4.5 puede verse el resultado de la ejecución. La figura 4.6 es el resultado del programa que veremos a continuación

Usando la biblioteca estándar de C (libc)

Un programa tan sencillo como el de sumar dos números enteros (programa 4.1) 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`.

El software del sistema tiene bibliotecas que incluyen esos subprogramas y muchas otras cosas de utilidad. 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 4.15 ilustra 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`).

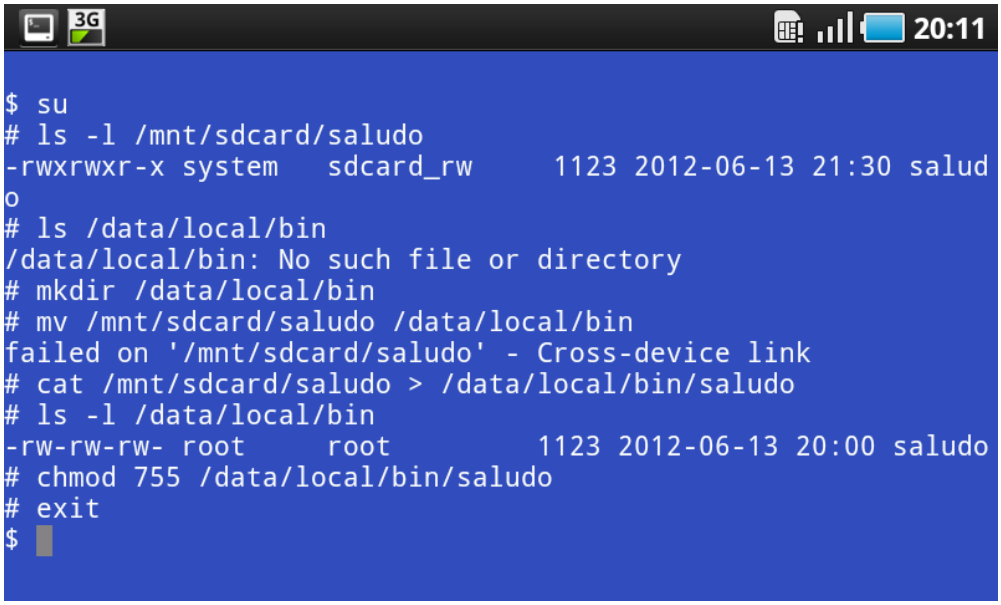


Figura 4.4 Órdenes para copiar el programa ejecutable en un directorio con permisos de ejecución.

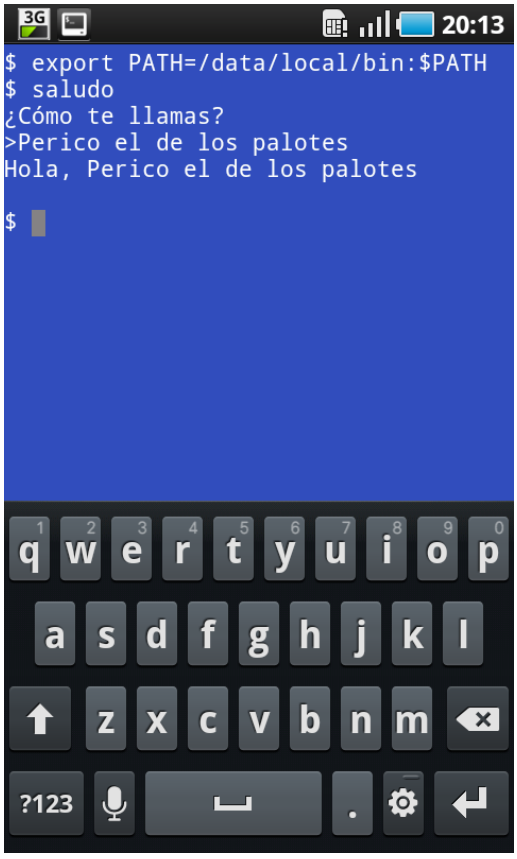


Figura 4.5 Ejecución de `saludo`.

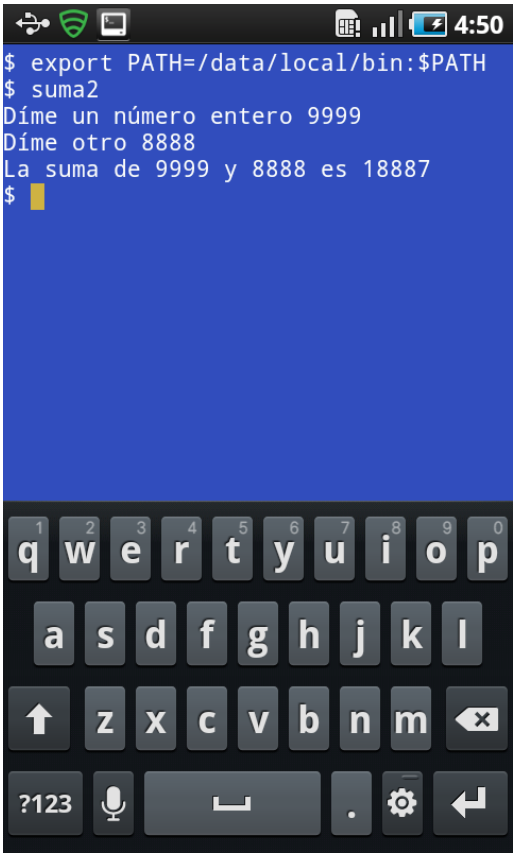


Figura 4.6 Ejecución de `suma2`.

```
             .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 4.15 Programa «Suma2».

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 anadir `-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 4.16, 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.

```
#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 4.16 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 (y de casi todo el Tema) 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 5

Procesadores software y lenguajes interpretados

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

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

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

5.1. Ensambladores

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

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

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

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

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

Tabla 5.1. Tabla de símbolos para el programa 4.2.

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

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

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

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

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

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

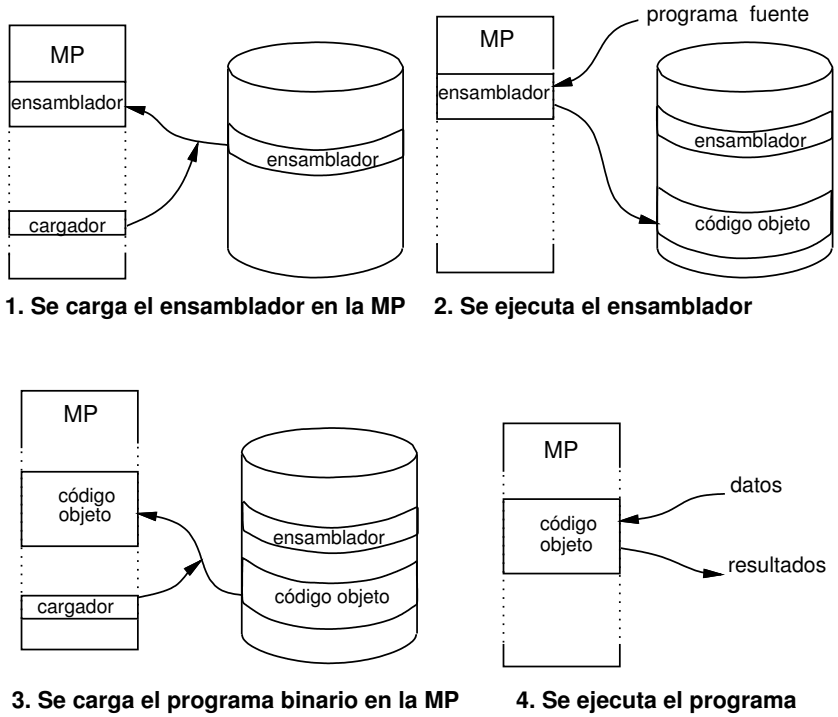


Figura 5.1 Procesos para el ensamblaje y la ejecución (incompleto).

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

5.2. El proceso de montaje

El ensamblador genera, para cada módulo, un código objeto *incompleto:* además del código, genera una **tabla de símbolos de externos** que contiene, para cada símbolo a importar (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 4.7. Probablemente no habrá usted reparado en la curiosa forma que ha tenido el ensamblador de traducir la instrucción `bl fact` del programa 4.10 (felicitaciones si lo ha hecho): `0xEBFFFFFE`. Los ocho bits más significativos, 0xEB = 0b11101011, corresponden efectivamente, según el formato de la figura 3.10, a la instrucción `BL`, pero para la distancia ha puesto 0xFFFFFE, que es la representación de -2. La dirección efectiva resultaría así: DE = 0xC $+8+(-4\times 2)$ = 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 4.11 con la instrucción `bl mult`. En este otro programa, `fact` está declarado como símbolo a exportar («`.global`»), y el ensamblador lo pone en la tabla de símbolos de acceso con su valor (0x0). Será el montador el que finalmente ajuste las distancias en las instrucciones de cada módulo teniendo en cuenta las tablas de símbolos de acceso de los demás módulos y el orden en que se han montado todos. Si, por ejemplo, ha puesto primero el programa 4.11 (a partir de la dirección 0) y a continuación el 4.10 (a partir de 0x38 + 4 = 60), la instrucción `bl fact` quedará en la dirección 0x0C + 60 = 72, y su distancia deberá ser Dist = $-72 \div 4 + 8 = -26$ =-0x1A, o, expresada en 24 bits con signo, 0xFFFFEC. La codificación de la instrucción después del montaje será: 0xEBFFFFEC, como puede usted comprobar si carga en ARMSim# los tres módulos en el orden indicado.

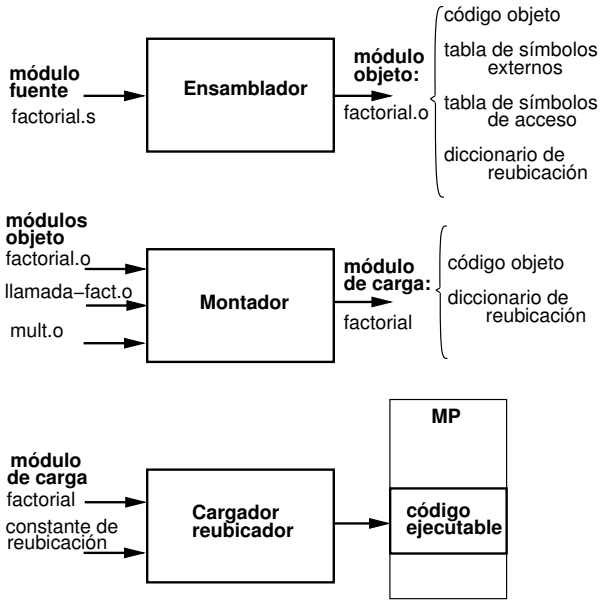


Figura 5.2. Procesos para el ensamblaje, el montaje y la ejecución.

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

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

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

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

5.3. Compiladores e intérpretes

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

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

Compiladores

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

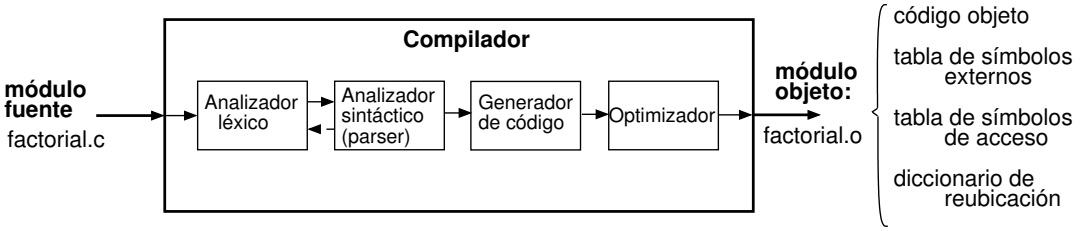


Figura 5.3 Componentes de un compilador.

Análisis léxico

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

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

Análisis sintáctico *(parsing)*

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

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

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

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

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

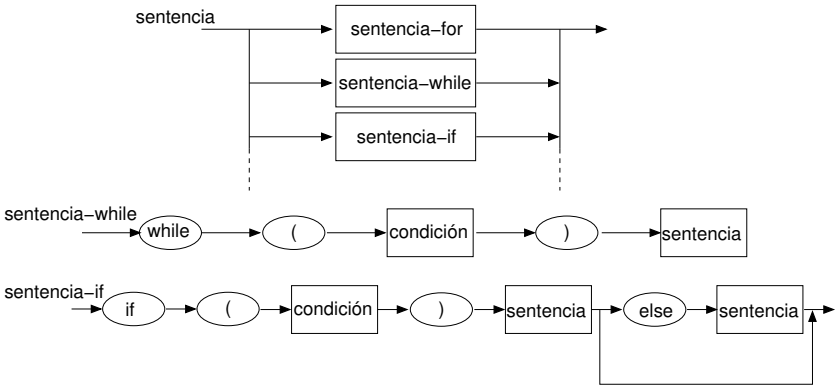


Figura 5.4 Diagramas sintácticos.

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

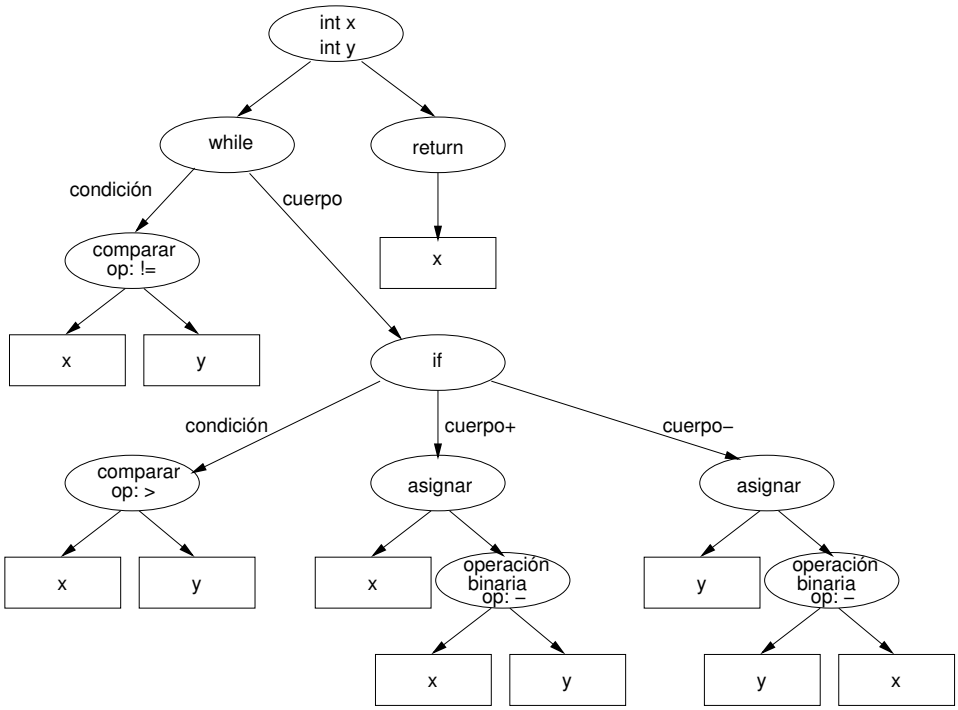


Figura 5.5 Árbol sintáctico de la función MCD.

Generación de código

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

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

Optimización

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

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

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

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

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

Intérpretes

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

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

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

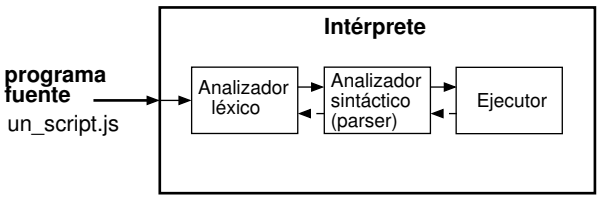


Figura 5.6 Componentes de un intérprete.

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

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

5.4. Lenguajes de marcas

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

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

HTML

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

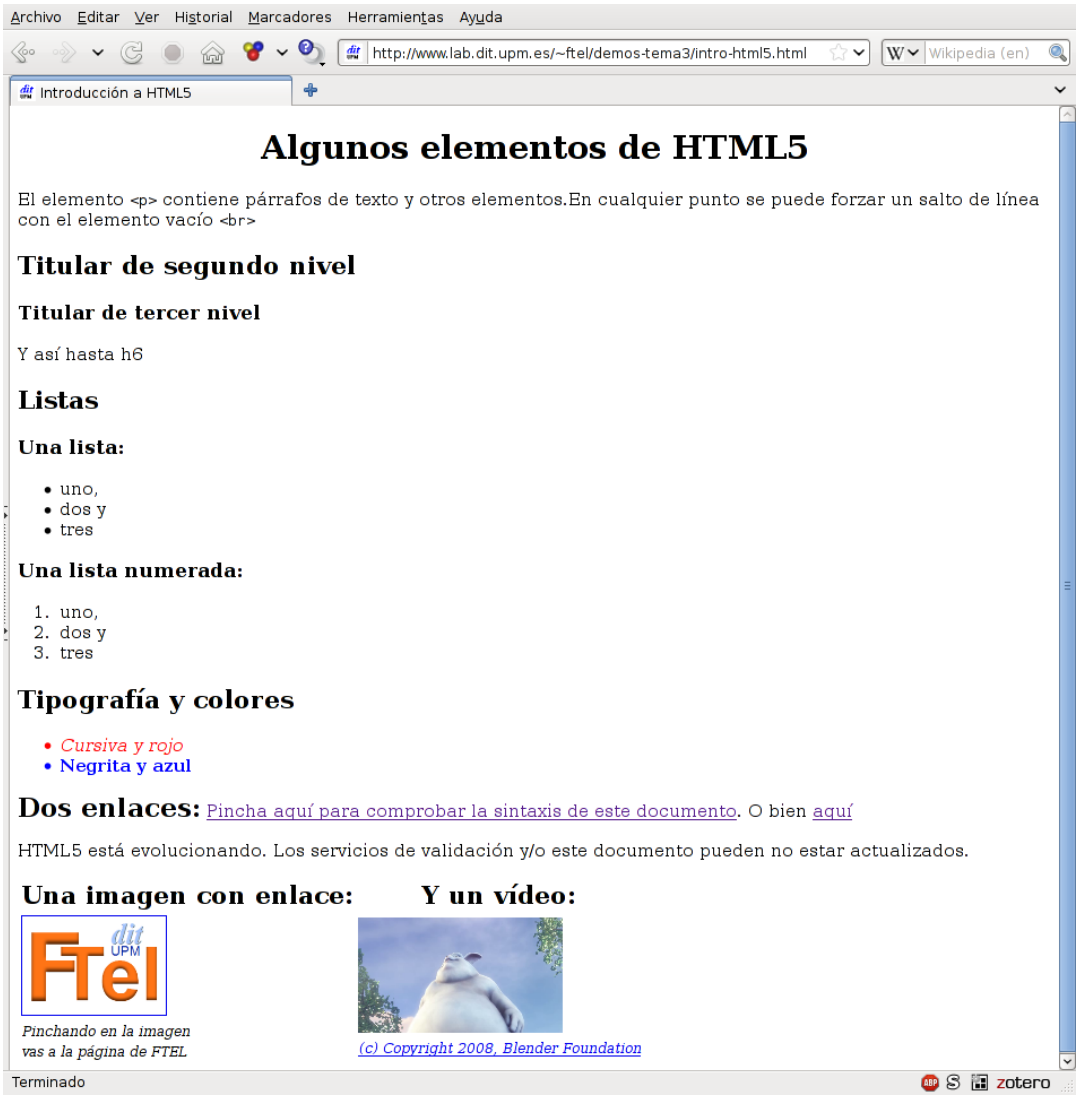


Figura 5.7 La página de muestra en el navegador

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

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

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

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

XML

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

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

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

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

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

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

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

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

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

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

Después están las especificaciones de valores de atributos:

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

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

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

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

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

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

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

5.5. Lenguajes de *script*

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

JavaScript

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

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

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

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

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

Una breve explicación del código:

- «`script`» es el nombre del elemento cuyo contenido son las sentencias del lenguaje. El valor del atributo «`type`» indica el lenguaje que el navegador debe interpretar. En HTML5 este atributo es innecesario: si no se pone, se entiende que es JavaScript.

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

- Las marcas de comentario «<!--...-->» son para los navegadores que no aceptan JavaScript. Al final, «//» es la indicación de comentario de JavaScript, para que el intérprete omita «-->».
- «write», «location» y «lastModified» son procedimientos incorporados en el lenguaje que se aplican al objeto «document» (el documento).
- «Date()» es una función de JavaScript que devuelve los datos que pueden verse en el resultado. Se pueden obtener los datos en español, pero escribiendo una función adecuada; Date() está ya incorporada «de serie».

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

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

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

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

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

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

- Contrariamente al ejemplo anterior, no utiliza document.write, porque esto sustituiría toda la página por una nueva, y lo que queremos es escribir en ciertos lugares sin sustituir la página.

- La función `getElementById("nombre")`, aplicada al objeto `document`, devuelve el elemento identificado por `"nombre"`, y en este elemento el procedimiento `innerHTML` sustituye su contenido HTML por el que le digamos. Así es como cambiamos los contenidos de texto de los tres `<p>`.

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

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

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

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

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

- `<form>` se utiliza normalmente para rellenar y enviar datos al servidor; aquí solo para recoger dos datos y llamar a un función JavaScript. En sí mismo, no es un elemento «visible»: sirve para contener otros elementos.
- `<input>` es para recoger datos del usuario. Puede adoptar distintas formas dependiendo del valor del atributo `type`. En los dos primeros casos del ejemplo, `"text"` presenta un cuadro para que se introduzca un texto con un máximo de 10 caracteres.
- El valor del atributo `onChange` se ejecuta cuando se cambia el contenido del texto: se asigna a la variable `x` el resultado de la función `numero(this)`. «`this`» es el propio elemento `<input>`.
- En la siguiente línea, el tipo es `button`. Al pulsarlo se ejecuta `Calcular(x,y)`.

Las funciones de la cabecera son:

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

AJAX

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

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

La figura 5.8 resume el principio. Ante un evento en el cliente, es decir, el navegador (como pulsar una tecla) se crea, mediante JavaScript, un objeto que se llama HttpRequest. En la asignatura «Programación» entenderá usted qué es eso de un «objeto». Digamos, simplificadamente, que es un conjunto estructurado de datos junto con unas funciones asociadas (abrir, enviar, etc.). El objeto se envía al servidor, que, tras analizarlo, elabora la respuesta adecuada y la remite al cliente. A diferencia de una interacción web «clásica», al recibir la respuesta el cliente no abre una página nueva: ejecuta funcio-

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

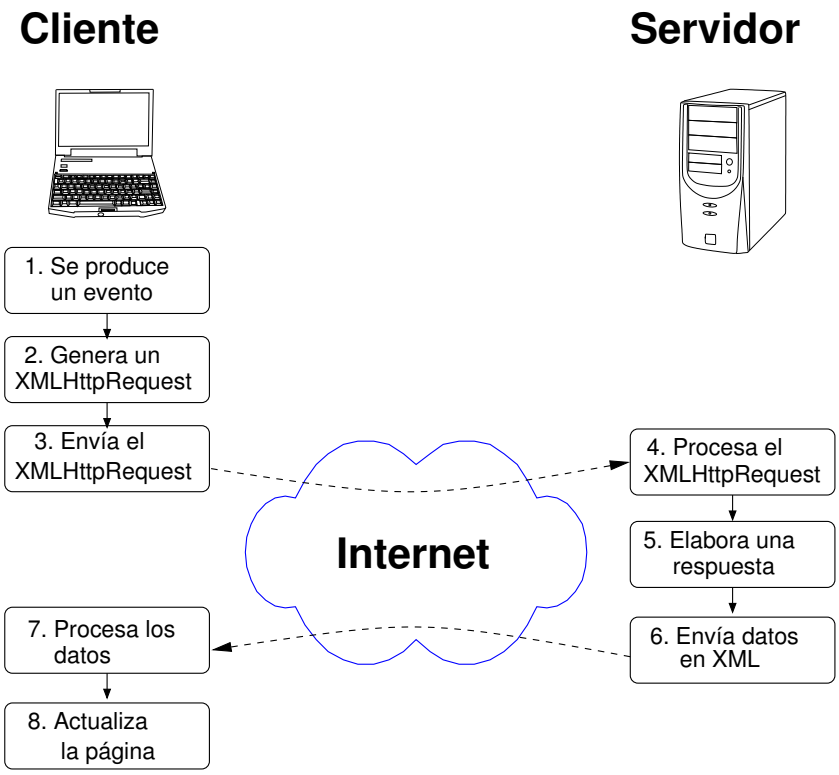


Figura 5.8 Procesos en AJAX

nes de JavaScript para modificar lo necesario en la página actual. Los datos se transmiten codificados en XML o en una alternativa más ligera como JSON (JavaScript Object Notation). Y en el caso del buscador de Google esta secuencia se produce cada vez que se pulsa una tecla.

Apéndice A

El simulador ARMSim# y otras herramientas

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

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

A.1. Instalación

Instalación en Windows

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

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

Instalación en UN*X

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

Estas son las instrucciones para instalar el simulador:

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

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

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

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

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

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

   pase a ser:

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

5. Con 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).

A.2. Uso

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

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

Vistas

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

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

Botones de la barra de herramientas

Debajo del menú principal hay una barra con seis botones:

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

Puntos de parada («breakpoints»)

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

Códigos para la interrupción de programa

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

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

Tabla A.1 Códigos de SWI

A.3. Otras herramientas

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

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

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

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

Herramientas GNU

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

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

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

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

Algunas otras opciones:

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

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

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

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

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

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

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

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

  `-D` desensambla también las secciones de datos.

  `-r` muestra los diccionarios de reubicación.

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

Android SDK

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

Herramientas ARM

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

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

Bibliografía

El documento de Burks, Goldstine y von Neumann utilizado en el capítulo 1 se publicó en 1946 como un informe de la Universidad de Princeton con el título «Preliminary discussion of the logical design of an electronic computing instrument». Se ha reproducido en muchos libros y revistas, y se puede descargar de `http://www.cs.princeton.edu/courses/archive/fall10/cos375/Burks.pdf`
Es más claro y explícito que un documento previo de 1945 firmado solo por von Neumann, «First Draft of a Report on the EDVAC», que también se encuentra en la Red:
`http://archive.org/details/firstdraftofrepo00vonn`
(Estos URL se han comprobado el 12/10/2013).

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

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

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

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

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

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

Fundamentos de los Sistemas Telemáticos

Curso 2014-2015

Laboratorio 5. HTML, CSS y JavaScript

Objetivos:

- Estudio básico del lenguaje HTML y de las hojas de estilo CSS.
- Inclusión de código JavaScript en páginas web.

Realización:

- La práctica se realizará en Linux. La duración estimada es de una hora.

Recursos:

- PC+Linux con configuración de red operativa.
- Editor de texto.
- Navegador Firefox.

Entrega de resultados.

Se deben subir al portal moodle tres ficheros (lab5.html, lab5.css, lab5.js) que se crearán a lo largo de la práctica, así como un fichero de imagen con la captura de pantalla que muestre el aspecto de la página creada. Se deben subir las versiones existentes al finalizar la tarea 10, más las modificaciones que se describen en la tarea de entrega. En la descripción de la tarea se indica cómo se deben subir los ficheros pedidos. En el enunciado se hacen algunas cuestiones para que considere el efecto que tienen determinados cambios sobre la página presentada, pero no se pide entrega de las respuestas.

Creación de una página básica HTML.

- Tarea 1. Creación de la página. Arranque un editor de texto (nano, gedit, etc.) y escriba el siguiente contenido:

```
<html>
<!-- Esto es un comentario: texto del laboratorio 5 -->
<head>
<title>
Laboratorio 5.
</title>
</head>
<body>
<h1>Ejemplo de p&aacute;gina web b&aacute;sica.</h1>
<p>Esto es un p&aacute;rrafo creado por XXXXX.</p>
</body>
</html>
```

  En el texto anterior cambie la cadena XXXXX por su nombre y apellidos. En el menú de "Archivo"/"File" seleccione la opción de "Guardar como"/"Save as" y guarde el fichero con el nombre lab5.html en una carpeta de su ordenador.

- Tarea 2. Visualización de la página en el navegador. Arranque el navegador firefox. Una vez arrancado, pulse Ctrl-O. Aparecerá un explorador en el que puede seleccionar el fichero lab5.html para abrirlo. ¿Dónde le muestra el navegador el texto "Laboratorio 5." que aparece en la página? ¿Hay alguna diferencia de presentación entre el encabezado y el contenido del párrafo? NOTA: en esta tarea el navegador presenta el contenido de un fichero HTML que se encuentra en el sistema de ficheros del ordenador, sin que haya comunicación con un servidor web usando el protocolo HTTP.

- Tarea 3. Modificación de la presentación del texto. Usando el editor de textos, cambie el contenido del fichero lab5.html para que su nombre y apellidos vayan entre etiquetas como sigue:

XXXXX . Una vez guardado el fichero con dicho cambio, en el navegador vuelva a cargar el fichero que está visualizando. ¿Qué ha cambiado? Modifique nuevamente el fichero para que ahora su nombre y apellidos vaya entre etiquetas en lugar de : XXXXX ¿Qué ha cambiado? Pruebe también a anidar las etiquetas y : XXXXX.

- Tarea 4. Encabezados de diferentes niveles. Usando el editor de textos, cambie el cuerpo (contenido entre las etiquetas <body>) del documento a lo siguiente (XXXXX se debe sustituir por su nombre y apellidos):

```
<h1>Curso 2014/2015</h1>
<h2>Listado de asignaturas que est&aacute; cursando XXXXX durante el primer
cuatrimestre</h2>
<p>Aqu&iacute; aparecer&aacute; la lista de asignaturas que est&aacute; cursando este
cuatrimestre.</p>
<h2>Listado de asignaturas que cursar&aacute; XXXXX durante el segundo cuatrimestre</h2>
<p>Aqu&iacute; aparecer&aacute; la lista de asignaturas de las que se matricular&aacute;
el segundo cuatrimestre.</p>
```

  Cargue el fichero actualizado en el navegador. Indique qué ha cambiado. Experimente también con encabezados de nivel inferior (<h3>, etc.).

- Tarea 5. Listas. Sustituya el contenido del párrafo bajo el encabezado "`Listado de asignaturas que est&aacute; cursando XXXXX durante el primer cuatrimestre`" por la lista de asignaturas con la estructura que sigue. Se da como muestra el elemento de lista correspondiente a FTEL, añada otros elementos (cada uno "encerrado" entre las etiquetas <li>...) con el resto de asignaturas de las que está matriculado este cuatrimestre.

```
<p>
<ul>
<li>Fundamentos de los Sistemas Telem&aacute;ticos (FTEL).</li>
<!-- A continuación añada más elementos de lista con el resto de asignaturas -->
</ul>
</p>
```

  Cargue el fichero actualizado en el navegador. Pruebe también a sustituir etiqueta de lista <ul> por la de lista ordenada <ol> (y la correspondiente etiqueta de cierre) y observe la diferencia en el tipo de lista que se muestra. Modifique el fichero para incluir también, bajo el párrafo correspondiente, la lista de asignaturas que tiene intención de cursar en el segundo cuatrimestre, y muéstrelo en el navegador.

- Tarea 6. Enlaces a otras páginas. Edite el fichero lab5.html y añada, justo antes de la etiqueta </body>, el siguiente contenido:

```
<p>Pr&aacute;ctica realizada en la
<a href="http://www.etsit.upm.es"  target="_blank">ETSI de Telecomunicaci&oacute;n</a>
</p>
```

  Cargue el fichero actualizado en el navegador. ¿Qué ocurre si pica en el enlace? ¿Dónde se abre la nueva página? ¿Qué ocurre si borra el texto `target="_blank"` y mantiene el resto?

- Tarea 7. Enlace de imágenes en una página HTML. Con su navegador, vaya al enlace http://www.upm.es/canalUPM/archivo/imagenes/logos/color/EtsiTeleco_new.jpg y descargue el fichero con el logotipo de la Escuela (que se llamará *EtsiTeleco_new.jpg*) en el mismo directorio en el que está el fichero lab5.html. Con el editor de texto edite ahora el fichero lab5.html y cambie el párrafo añadido en la tarea 6 para que tenga el siguiente contenido:

```
<p>Pr&aacute;ctica realizada en
<a href="http://www.etsit.upm.es"  target="_blank"><img src="EtsiTeleco_new.jpg"></a>
</p>
```

  Cargue el fichero actualizado en el navegador. ¿Dónde debe picar con el ratón para acceder a la página de la Escuela?

Cambie ahora la etiqueta <img> anterior por esta otra:

```
<img src="EtsiTeleco_new.jpg" height="50" width="120">
```

Repita la carga del fichero en su navegador y describa los cambios que observa. ¿En qué unidades cree que están expresados los valores de los atributos “height” y “width”?

Hojas de estilo CSS.

- Tarea 8. Con el editor de texto, cree un fichero de nombre lab5.css y con el siguiente contenido:

```
h1 {/* Estilo de encabezado 1 */    color:blue; text-align:center;}
h2 {/* Estilo de encabezado 2 */    color:green; text-align:left;}
p  {/*Estilo de párrafo */          color:red; text-align:left;}
```

Guárdelo en el mismo directorio en que se encuentra el fichero lab5.html. Edite ahora el fichero lab5.html y añada la siguiente línea en la sección de cabecera (definida por la etiqueta <head>):

```
<link rel="stylesheet" type="text/css" href="lab5.css">
```

Con el navegador cargue el fichero lab5.html y compare lo que observa con lo observado en las tareas anteriores.

- Tarea 9. Modificaciones en la hoja de estilo. Edite el fichero lab5.css y haga los siguientes cambios:
  - Añada un estilo para que el color de fondo de los encabezados h1 y h2 aparezcan en color marfil ("*ivory*"). Puede consultar este enlace como ayuda.
  - Añada un estilo para que el fondo del cuerpo de la página (<body>) sea el contenido del fichero http://www.ics.upm.es/imagenes/upm.gif.
  - Añada un estilo para que los elementos de la lista aparezcan en color naranja.
  - Modifique el estilo del encabezado de segundo nivel para que aparezca centrado.

Inclusión de código JavaScript en una página web.

- Tarea 10. Abra un editor de texto y copie el código JavaScript de la función saludo() que aparece en el ejemplo 2 de las transparencias de JavaScript del tema 3 (lo comprendido entre las etiquetas <script> y </script>, sin incluir dichas etiquetas). Guarde el fichero editado en el mismo directorio en que están los ficheros lab5.html e lab5.css, y dele el nombre lab5.js.

  Edite ahora el fichero lab5.html y haga los siguientes cambios:

  - Entre las etiquetas </title> y </head> inserte la siguiente línea:

    ```
    <script type="text/javascript" src="lab5.js"></script>
    ```

  - En el cuerpo (entre las etiquetas <body> y </body>) de la página HTML, donde crea oportuno para la presentación, añada las siguientes líneas:

    ```
    <p>Pinche en el bot&oacute;n para saber la hora.</p>
    <button onClick="saludo()">Bot&oacute;n</button>
    <p id="result"></p>
    ```

  - Guarde el fichero con los cambios anteriores y ábralo con su navegador. Compruebe que el comportamiento es el esperado según lo explicado en el ejemplo de JavaScript.
  - ¿Qué relación tiene el atributo id=”result” con el código JavaScript? En el fichero lab5.html cambie el valor del atributo id a “resultado” y cargue el fichero cambiado en el navegador. ¿Qué ocurre y por qué? ¿Qué habría que cambiar en el código JavaScript para que el comportamiento volviese a ser el mismo de antes? Haga el cambio y compruebe que funciona.

  Al concluir esto, debe realizar las modificaciones que se describen en la tarea de entrega. Una vez hechas, muestre la página creada con la presentación del saludo y la hora tras haber pulsado el botón, maximice la ventana del navegador para que ocupe toda la pantalla del PC y no haya otras ventanas sobre el navegador. Haga una captura de pantalla (pulse la tecla "Impr Pant" del teclado). La imagen está en la carpeta "Imágenes" de su cuenta. Puede acceder a ella en el menú Lugares→Imágenes. Tendrá un nombre com “Screenshot from …”. png, en el que los puntos suspensivos representan la fecha y hora de la captura. Suba este fichero a moodle, junto con los ficheros lab5.html, lab5.css y lab5.js, siguiendo las instrucciones que se describen en la tarea de entrega..

Laboratorio 6. Procesadores hardware. Simulación con ARMSim#

Objetivos:

- Experimentar la ejecución simulada de programas escritos en ensamblador de ARM.
- Comprobar los resultados de la ejecución de instrucciones de movimiento y procesamiento (`mov`, `add`, `sub`, `cmp`), de instrucciones de bifurcación (`b`, `bl`, `blt`), de instrucciones de acceso a la memoria (`ldr`, `srtb`) y de interrupción (`swi`).

Recursos:

- El ordenador del laboratorio o cualquier equipo que tenga instalado el simulador ARMSim#.
- El Moodle de la asignatura

Resultados:

Como resultado de esta práctica debe subirse a Moodle en la tarea "LAB6-Gxx. Entrega del fichero de resultados" la plantilla rellena con las respuestas a las cuestiones planteadas en este enunciado de la práctica (escritas en negrita) asignándole el nombre según el formato establecido: Grupo-Apellido1-Apellido2-L6.odt.

Actividades previas:

Antes de la sesión de laboratorio, debe leer previamente este documento e intentar realizar las actividades indicadas para asegurar que puede completarlo antes de finalizar la sesión.

Programa objeto de la práctica

Empiece desde el entorno gráfico, abra un terminal de texto, cree un nuevo directorio `lab6`, vaya a este directorio `lab6` y baje del Moodle el fichero `saludo.s` que contiene un programa de saludo, similar al de la función en javascript (`saludo.js`) con la que trabajó en el laboratorio anterior pero escrito ahora en ensamblador para el simulador ARMSim#.

Baje también al directorio lab6 la plantilla para ir escribiendo los resultados: `resultados-lab6-2014.odt`.

Realice paso a paso las siguientes actividades:

1. Utilizando un editor de texto plano, abra el fichero `saludo.s` y analice el programa contenido en él[1]. Con este mismo editor, modifique en el programa los valores numéricos que, con la directiva `.equ`, se asignan a `H` (hora), `M` (minutos) y `S` (segundos) sustituyéndolos por los valores que se indican en la definición de la tarea para su grupo. Al guardarlo mantenga la codificación UTF-8.

Arranque el simulador ARMSim# y compruebe que tiene habilitado `SWIInstructions` en `File >Preferences>Plugins`. Cargue (`File >Load`) el fichero `saludo.s` en el simulador. Observe que, al decirle «Load» al simulador, éste automáticamente ensambla el programa antes de cargarlo. Si le diera algún error al cargarlo es porque, al modificar el programa, ha cometido algún error. Abra el editor de texto, corrija lo necesario, sálvelo y vuelva a cargarlo en el simulador (`File >Reload`, o icono en la barra de herramientas).

Seleccione `Hexadecimal` en la tabla de registros (a la izquierda). Observará que todos los registros están a cero salvo el R13 (sp) que contiene el valor `0x5400` y el R15 (pc), que tiene el valor `0x1000` (dirección de comienzo de la ejecución del programa cargado) y verá que en el registro CPSR los indicadores N, Z, C y V están a 0.

[1] Para facilitar su comprensión, al final de este enunciado tiene una tabla explicativa de las instrucciones y directivas utilizadas en este programa

2. Ejecute, una a una las cuatro primeras instrucciones del programa (con el icono «Step Into»  de la barra de herramientas **y anote en el formulario de entrega de resultados los contenidos de los registros R0, R1 y R15, explicando brevemente qué significan los nuevos valores que han tomado estos registros.**

3. Continúe ejecutando paso a paso y pare cuando R15 (cp) tome el mismo valor que R14 (lr). (Si se pasa, pulse el icono «Restart»  y reinicie la ejecución paso a paso, hasta que los valores de ambos coincidan). **Anote en el formulario el contenido de R15 y explique brevemente por qué ambos registros tienen el mismo valor. Anote también los contenidos de los registros R1 y R2 y explique a qué corresponden sus valores.**

4. Si no tiene abierta la ventana MemoryView, actívela desde el menú View. (Si tiene activado Stack, puede desactivarlo porque no se usa). En MemoryView seleccione WordSize>8bits y ponga 000010e6 como valor de dirección. Continúe ejecutando el programa paso a paso para ir observando lo que hace. Una vez que lo haya entendido, puede seleccionar Stdin/Stdout/Stderr en OutputView, pulsar el icono «Run»  y ejecutar todo el programa de una vez. Cuando el programa se pare (instrucción `swi 0x11`), **anote en el formulario de entrega el resultado que aparece en Stdin/Stdout/Stderr y todos los valores en rojo (contenidos de registros, indicadores, y contenidos de bytes de memoria).**

5. Modifique el programa según indica la definición de la tarea para este grupo, cárguelo en el simulador, ejecútelo y **escriba en el formulario las diferencias que ve en los resultados**.

6. El programa saludo.s no comprueba si los valores de hora que ponemos con las directivas .equ son correctos. Edite el fichero saludo.s y cambie el valor de M (minutos) a 100. Ejecute el programa y **anote en el formulario el texto que aparece en Stdin/Stdout/Stder explicando por qué no** se **escribe**n **correctamente los minutos.**

7. Para evitar que acepte datos no válidos, debe modificar el código de modo que se compruebe que los valores asignados son correctos y, si no lo son, escribir un mensaje adecuado.

   Para ello, edite de nuevo el fichero saludo.s y realice los siguientes cambios:

   a) detrás de la instrucción `crea_cad:  mov r1, #H` que carga en r1 el valor decimal de H, añada dos nuevas instrucciones:

```
cmp r1, #24  @ comparar (r1), que es igual a H, con el valor 24
bge error    @ si (r1) es mayor o igual que 24 bifurcar a otra parte del
             @ programa en la que se escribe un mensaje de error
```

   b) añada otras dos instrucciones como las anteriores, detrás de la instrucción que carga en r1 el valor decimal de M, y otras dos detrás de la instrucción que carga en r1 el valor decimal de S pero, en ambos casos, poniendo 60 como valor límite máximo válido.

   c) justo al final del código (antes de .data) añada la siguiente rutina:

```
error: ldr r0, =m_error @ escribe por pantalla el mensaje de error
       swi 0x02         @ guardado en m_error como .asciz
       swi 0x11         @ y finaliza la ejecución
```

   d) y justo antes de .end añada:

```
m_error: .asciz "Valor no válido: hora debe ser < 24, minutos y segundos
deben ser < 60"
```

8. Ejecute de nuevo el programa (con el valor de M igual a 100) y **anote en el formulario el resultado que aparece en la ventana Stdin/Stdout/Stderr. Anote también los valores que aparecen en rojo (contenidos de registros, bytes de memoria e indicadores).**

9. Para terminar, **copie al formulario de entrega el programa en ensamblador saludo.s tal como le ha quedado después de la realizar todos los cambios**.

Tablas con las instrucciones y directivas usadas en el programa saludo.s

(Aparecen en orden alfabético.)

Instrucciones	Significado
`add rd, rf1, #val`	Suma el valor *val*[2] al contenido del registro *rf1*[3] y lo guarda en el registro *rd*
`add rd, rf1, rf2`	Suma los contenidos de los registros *rf1* y *rf2* y guarda el resultado en *rd*.
`b etiq`	Bifurca a la instrucción que tiene la etiqueta *etiq*[4].
`bl etiq`	Bifurca a la instrucción con etiqueta *etiq* pero antes guarda el valor del contador de programa (r15) en el registro de enlace (r14).
`blt etiq`	Si como consecuencia de una operación anterior se cumple la condición menor-o-igual (lt), bifurca a la instrucción de etiqueta *etiq*.
`cmp rf, #val`	Compara el contenido del registro *rf* con el valor *val*. Si son iguales, pone el indicador Z = 1 y, si no, pone Z = 0. Si el contenido del registro *rf* es menor que *val*, pone el indicador N = 1y, si no, pone N=0.
`ldr rd, =etiq`	Carga en el registro *rd* la dirección de la memoria etiquetada con *etiq*.
`mov rd, rf`	Copia el contenido del registro *rf* en el registro *rd*.
`mov rd, #val`	Copia el valor *val* al registro *rd*.
`strb rf1, [rf2], #val`	Almacena los 8 bits menos significativos del registro *rf1* en el byte cuya dirección es el contenido de *rf2*, y luego suma *val* al contenido de *rf2*
`sub rd, rf1, rf2`	Guarda en el registro *rd* el resultado de restar los contenidos de los registros *rf1* y *rf2*.
`swi 0x11`	Simula una llamada al sistema operativo que para la ejecución del programa
`swi 0x02`	Simula una llamada al sistema operativo para escribir en la pantalla una cadena de caracteres almacenada en memoria, a partir de la dirección que coincide con el contenido del registro R0 y continúa con los caracteres almacenados en las siguientes direcciones hasta que encuentra un carácter nulo (0)

Directivas	Significado
`.ascii "… "`	Indica al ensamblador que cargue en memoria, en la zona de datos, la cadena de caracteres `"… "`[5], justo detrás de los datos que se hayan definido previamente con directivas similares (o al principio de la zona de datos si no hay directivas previas).
`.asciz "… "`	Hace lo mismo que `.ascii` y además añade al final el carácter nulo (0)
`.data`	Indica que empieza la parte de definición de datos
`.byte valores`	Indica al ensamblador que cargue en memoria, en la zona de datos, la lista de *valores* especificados (tipo byte), a continuación de los definidos previamente
`.equ cte, val`	Establece la equivalencia entre el nombre de una constante *cte*[6] y su valor *val*
`.end`	Indica al ensamblador el fin del programa (incluido código y datos)
`.skip val`	Indica que debe reservarse memoria para almacenar un número *val* de bytes
`.text`	Indica al ensamblador que empieza la zona de código ejecutable

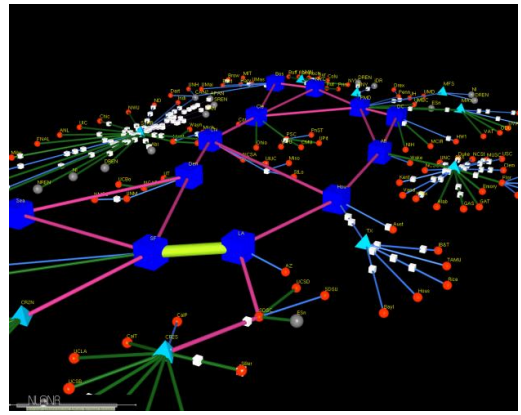
[2] *val* representa un valor en decimal (por defecto), o en hexadecimal si empieza por 0x.

[3] *rf* (registro fuente 1 o 2) y *rd* (registro destino) representan a alguno de los 16 registros (r0-r15).

[4] *etiq* debe interpretarse como una etiqueta del programa

[5] `"… "` representa una cadena de caracteres ASCII de cualquier longitud

[6] *cte* representa el nombre de una constante



Fundamentos de los Sistemas Telemáticos
Tema 4: Internet

Material de estudio

- Estas transparencias
  - Disponibles en moodle
- J.F. Kurose, K.W. Ross. Redes de Computadoras. 5ª ed. Pearson, 2010.
  - Cap. 1, apartados 1.1, 1.2, 1.3 (excepto conmutación de circuitos), 1.4 y 1.5.
  - Cap. 2, apartados 2.1, 2.2, 2.5
  - Disponible el Cap.1 como material descargable en el sitio web de la editorial
    - http://www.pearsoneducacion.com
    - buscar por Autor = Kurose

Tema 4. Internet

- Internet
- Núcleo de la red
- Conceptos de prestaciones
- Arquitectura de red
- Principios de las aplicaciones de red
- La web y HTTP
- DNS

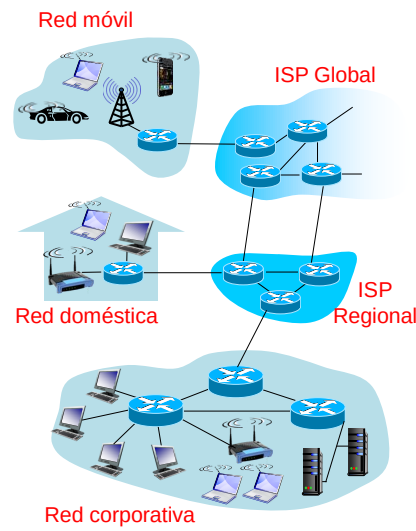
Tema 4. Internet

- Internet
  - Componentes, conceptos, servicios, estructura
- Núcleo de la red
- Conceptos de prestaciones
- Arquitectura de red
- Principios de las aplicaciones de red
- La web y HTTP
- DNS

Internet: componentes

¿Qué es Internet?

- Red de comunicaciones formada por cientos de millones de dispositivos denominados **sistemas finales** o también **sistemas terminales** (“*hosts*”)
  - Ej.: ordenadores, teléfonos móviles, televisores, consolas, sensores, etc.
- conectados entre sí mediante **enlaces** de comunicaciones y **conmutadores de paquetes** (routers).
- Los enlaces de comunicaciones utilizan diferentes medios físicos a través de los cuales los datos se transmiten en la red.
  - Ej. de medios físicos: hilo de cobre, fibra óptica, ondas de radio

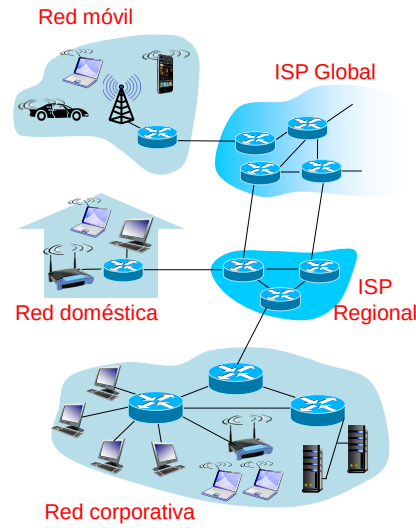


Internet: conceptos (1)

- Los datos se transmiten a través de los distintos enlaces a distinta velocidad
  - Depende del medio físico que utilicen
- Esa velocidad se denomina capacidad del enlace o caudal
  - Se mide en bits/segundo (bps)
- Los datos viajan en unidades que se denominan paquetes.
  - El emisor segmenta los datos y añade bits de cabecera a cada segmento, formando los paquetes que se envían a través de la red
  - Cuando los paquetes llegan al receptor, vuelven a ser ensamblados para obtener los datos originales
  - Su tamaño depende de la aplicación (si es una petición, o una respuesta, una descarga, etc.)
  - Tienen un tamaño máximo establecido, típicamente 1.500 octetos.

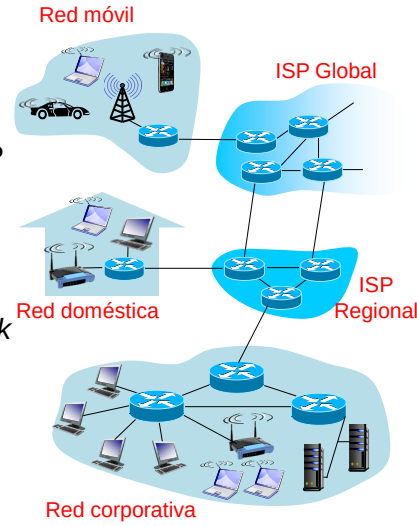
Internet: conceptos (2)

- Un "*router*":
  - toma los paquetes que llegan por sus enlaces de entrada y los reenvía por sus enlaces de salida
    - Encamina los paquetes hacia su destino final
- **Ruta** que sigue un paquete:
  - Secuencia de enlaces de comunicaciones y routers desde un SF emisor hasta un SF receptor

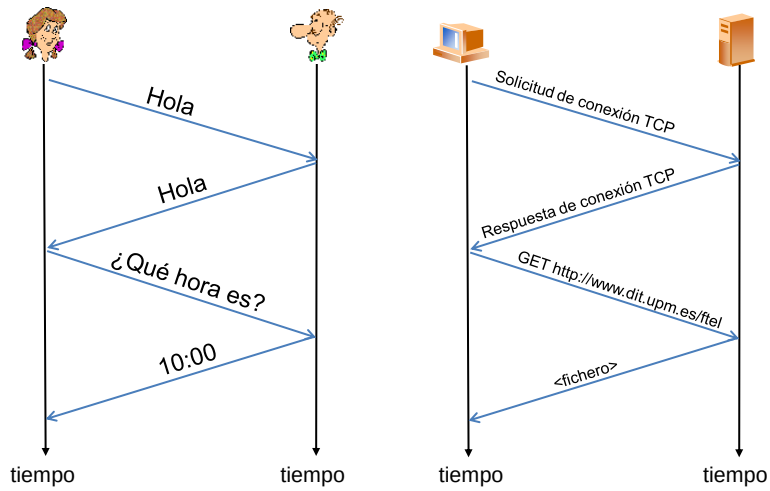


Protocolos y estándares

- Protocolo: conjunto de reglas que gobierna el envío y recepción de mensajes
  - Ej.: HTTP, SMTP, IP, TCP, UDP
  - "Familia" TCP/IP
- Estándares de Internet:
  - RFC: *Request for Comments*
  - IETF: *Internet Engineering Task Force*

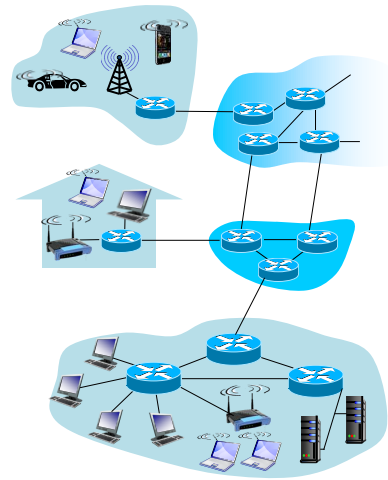


Ejemplo de protocolo humano y protocolo de red



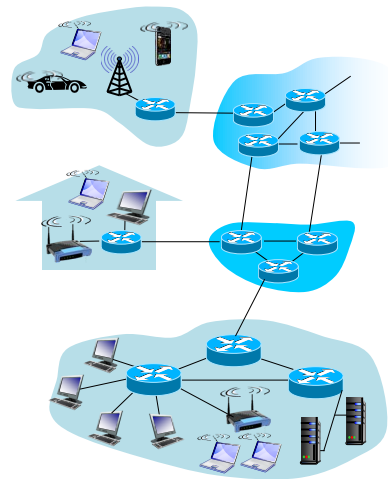
Internet de los servicios

- Internet es una infraestructura que ofrece un servicio básico de transferencia de paquetes a las aplicaciones
  - Permite la creación de servicios avanzados
- Aplicaciones distribuidas = programas que utilizan la infraestructura
  - Aplicaciones de red
  - Se ejecutan en los sistemas finales
- Ejemplos de aplicaciones
  - correo-e, navegación web, VoIP, IPTV, juegos en red, P2P, videoconferencia, etc.



Estructura de Internet

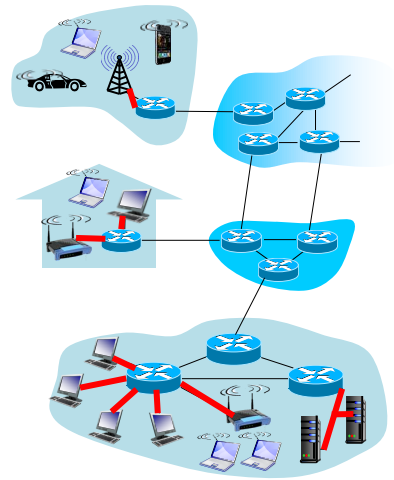
- **Periferia (o frontera) de la red**:
  - aplicaciones y sistemas finales (SF)
- **Redes de acceso**:
  - conectan a los usuarios con los proveedores de servicios
- **Núcleo de la red**:
  - routers
  - red de redes



Redes de acceso

¿Cómo conectar los sistemas finales a los routers de los proveedores de servicios?

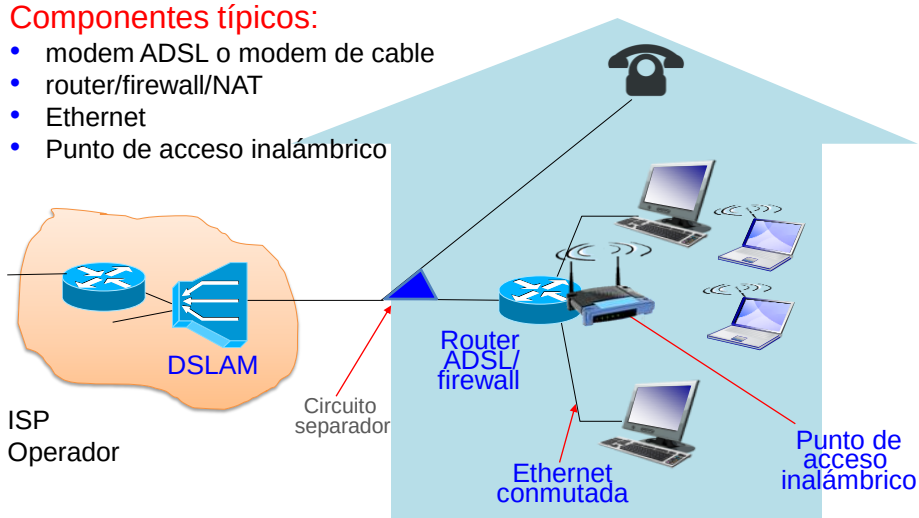
- Redes residenciales
- Redes corporativas (empresa, universidad, ...)
- Redes inalámbricas



Acceso residencial: redes en casa

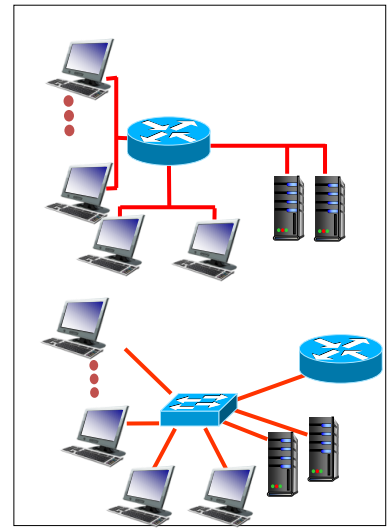
Componentes típicos:

- modem ADSL o modem de cable
- router/firewall/NAT
- Ethernet
- Punto de acceso inalámbrico



Acceso en organizaciones: redes de área local

- Las redes de área local (LAN) permiten la interconexión entre sí de sistemas finales en una organización y la conexión al primer router en el camino
- Ethernet:
  - medio compartido y dedicado
  - 10 Mbps, 100Mbps, 1Gbps, 10 Gbps Ethernet



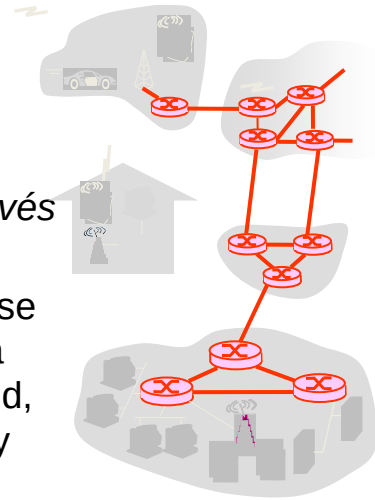
Tema 4. Internet

- Internet
- Núcleo de la red
  - Conmutación de paquetes
  - Encaminamiento
  - Organización de la red
- Conceptos de prestaciones
- Arquitectura de red
- Principios de las aplicaciones de red
- La web y HTTP
- DNS

El núcleo de la Red

- Malla de conmutadores de paquetes (routers) interconectados
- *¿Cómo se realiza la transferencia de datos a través de la red?*

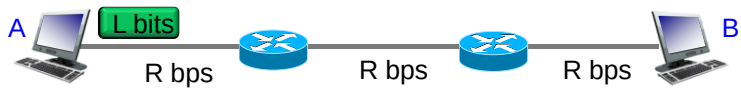
➡ conmutación de paquetes: se envían paquetes de datos a través de los nodos de la red, mediante almacenamiento y reenvío.



Conmutación de paquetes

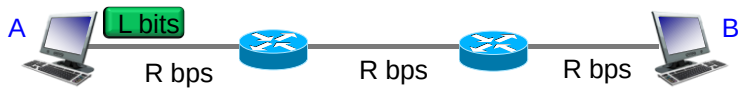
- El emisor divide los mensajes en paquetes
- Los paquetes viajan a través de los enlaces de comunicaciones y de los conmutadores (nodos de la red)
  - Comparten los recursos existentes
  - Cada paquete utiliza toda la capacidad del enlace
- Los conmutadores emplean el método de almacenamiento y reenvío
  - El conmutador recibe el paquete completo antes de pasarlo al enlace de salida

Conmutación de paquetes: almacenamiento y reenvío

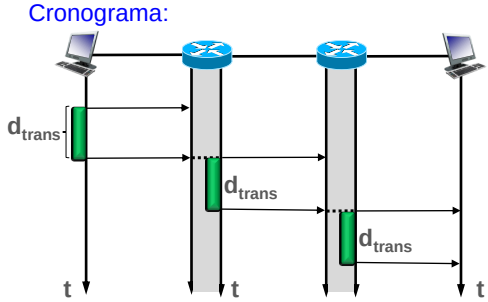


- Lleva $d_{trans} = L/R$ segundos entregar un paquete de L bits a un enlace de capacidad R bps
  - **d_{trans} es el retardo de transmisión**
- Almacenamiento y reenvío: todos los bits del paquete deben llegar al router antes de que pueda ser reenviado por el siguiente enlace
- Si hay Q enlaces iguales, se acumula un retardo mínimo = Q*L/R

Conmutación de paquetes: almacenamiento y reenvío



Cronograma:



Ejemplo:

L = 3.000 bits
R = 1,5 Mbps

Tiempo que tarda en llegar el paquete de A a B = 6 ms.

Encaminamiento

¿Cómo se determina la ruta que debe seguir un paquete hasta alcanzar su destino?

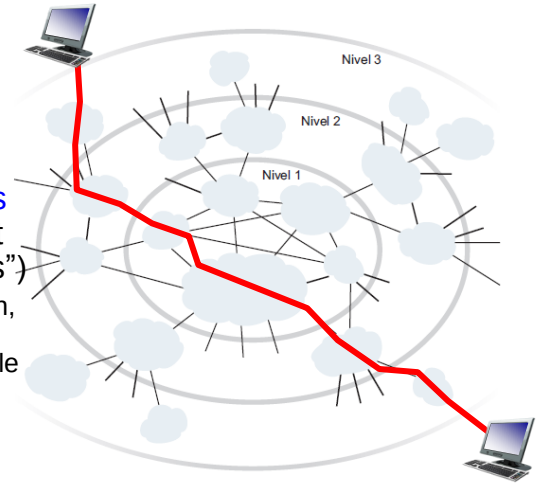
- Cada paquete lleva una cabecera con la dirección destino
- Cuando un router recibe un paquete, realiza el proceso de reenvío ("forwarding"):
  - Examina la dirección destino
  - Busca en una "tabla de encaminamiento" el enlace de salida adecuado
  - Reenvía el paquete por ese enlace de salida al router vecino
- Los protocolos de encaminamiento determinan las rutas más cortas y configuran las tablas de encaminamiento de los routers.

Proveedores de servicios de Internet

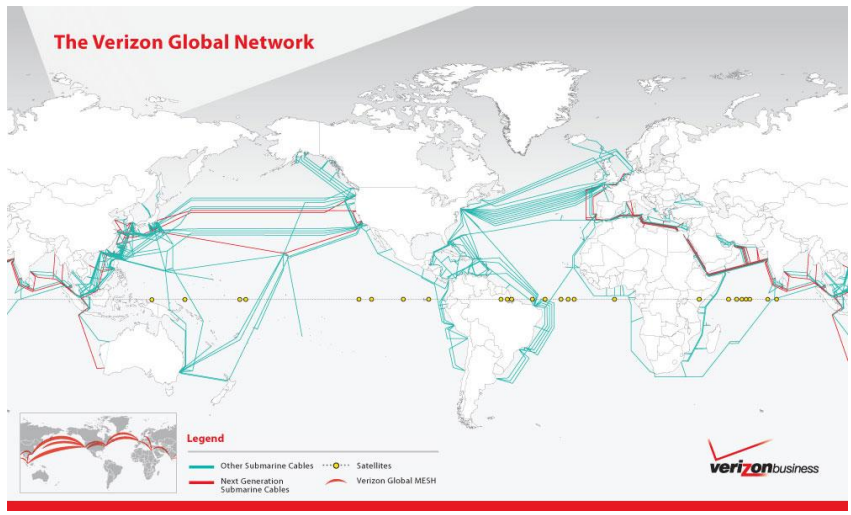
- Los sistemas finales acceden a Internet a través de los ISP
  - *Internet Service Providers*
- Cada ISP es una red de routers y enlaces
  - Proporcionan conectividad
  - Ofrecen diferentes tipos de accesos y servicios (ej.: ADSL, cable, inalámbrico, contenidos, …)
- Pueden ser de ámbito local, regional, nacional o internacional

Organización de Internet: red de redes

- Jerarquía de proveedores ISP
- En el centro: ISPs de nivel 1
  - Se denominan redes troncales de Internet ("Internet backbones")
    - Ejem.: Sprint, Verizon, MCI, AT&T, NTT, Level3, Qwest y Cable & Wireless
  - Conectados entre sí



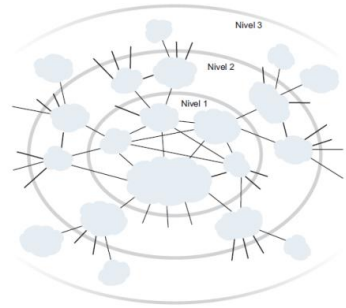
Ejemplo de ISP de nivel 1



http://www.verizonbusiness.com/us/about/network/

Organización de Internet

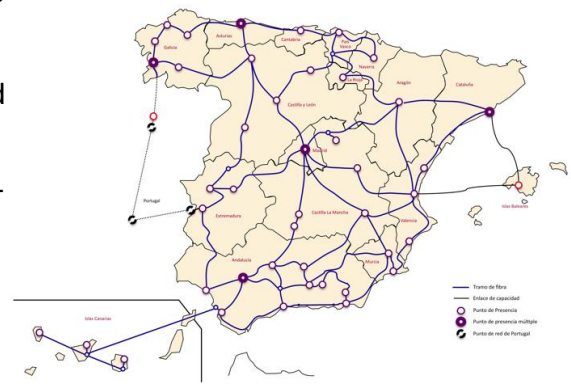
- Los ISPs de nivel 2 tienen cobertura regional o nacional
  - "Clientes" de ISPs de nivel 1
  - Encaminan el tráfico a través de ISPs de nivel 1



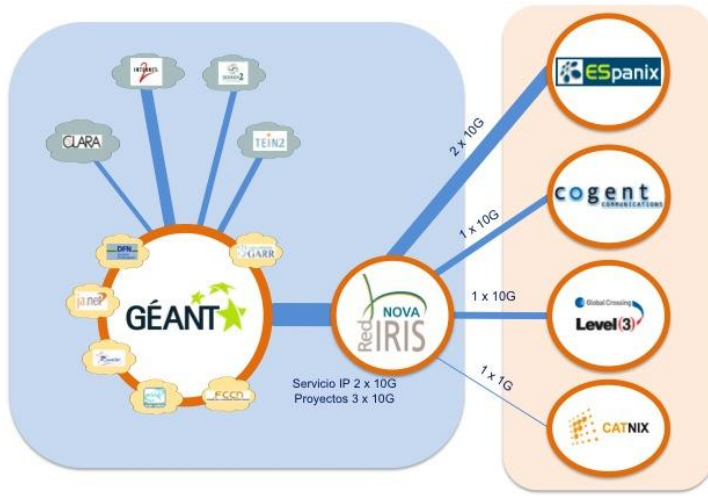
- Por debajo del nivel 2 están los ISPs locales
  - Red del "último salto" (*last hop network*)

RedIRIS

- Proveedor de Internet (ISP) para Universidades y Centros de Investigación nacionales
- Proporciona conectividad de red (IP) y servicios de comunicaciones (operación de red, proxy-cachés, correo web, etc)
- Un nodo por comunidad autónoma
- Red IP sobre una red óptica de alta capacidad



RedIris: conexiones externas



Tema 4. Internet

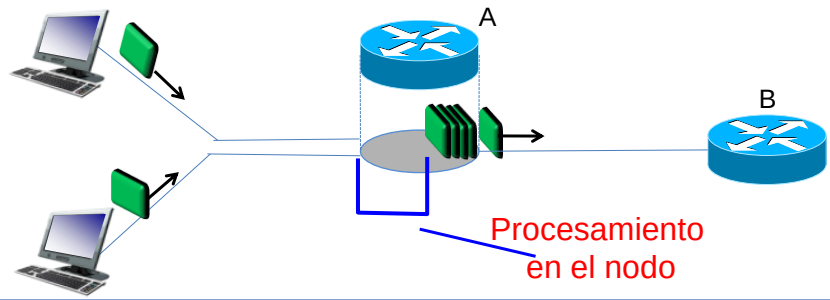
- Internet
- Núcleo de la red
- Conceptos de prestaciones
  - Retardos, pérdidas y caudales en redes de conmutación de paquetes
- Arquitectura de red
- Principios de las aplicaciones de red
- La web y HTTP
- DNS

Retardos

- En la ruta desde un sistema final origen a un SF destino, un paquete sufre varios tipos de retardos en cada uno de los routers y enlaces que atraviesa
- Retardo total (nodal) en un router, suma de:
  - Retardo de procesamiento en el router, d_{proc}
  - Retardo de espera en cola, d_{cola}
  - Retardo de transmisión, d_{trans}
  - Retardo de propagación, d_{prop}

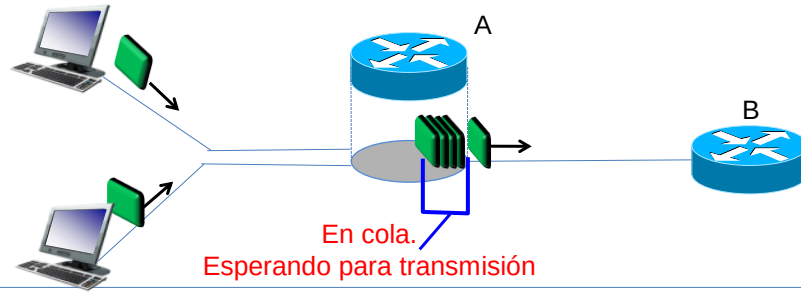
Retardo de procesamiento (d_{proc})

- Tiempo requerido por el router para examinar la cabecera del paquete y determinar el enlace de salida por donde hay que enviarlo
  - Del orden de microsegundos



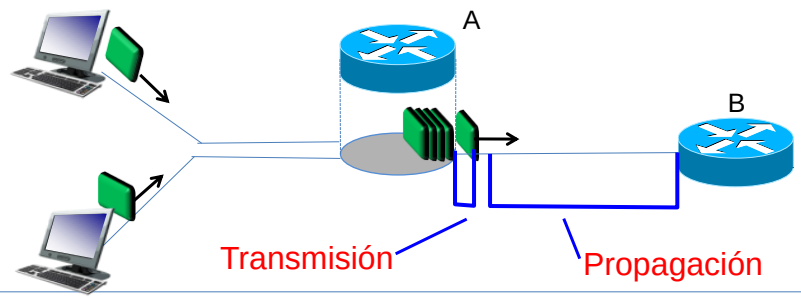
Retardo de espera en cola (d_{cola})

- Tiempo de espera del paquete en la cola de transmisión (de salida) del router antes de ser transmitido
  - Depende del número de paquetes que hayan llegado antes a la cola
  - Si la cola está vacía el paquete se transmite inmediatamente y el retardo es cero



Retardo de transmisión (d_{trans})

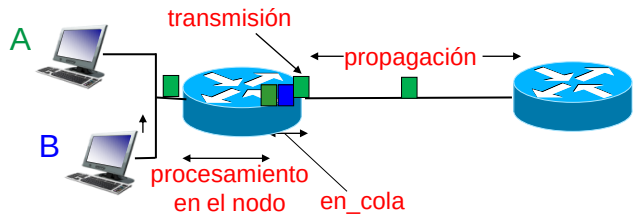
- Tiempo necesario para transmitir todos los bits del paquete desde el nodo al enlace
  - **d_{trans} = L / R segundos**
  - **L** es la longitud en bits del paquete y **R** la capacidad del enlace en bps (bits/segundo)



Retardo de propagación (d_{prop})

- Tiempo necesario para que un bit se propague a través de un mismo enlace, desde el principio del enlace hasta el final.
  - El bit se propaga a la velocidad de propagación del enlace, velocidad que depende del medio físico
    - Entre $2*10^8$ metros/segundo y $3*10^8$ metros/segundo
- **d_{prop} = long / s segundos**
  - **long** es la longitud del enlace (distancia entre el router A y B) en metros
  - **s** es la velocidad de propagación en el medio, en metros/segundo

Total: cuatro causas del retardo de los paquetes



$$d_{nodal} = d_{proc} + d_{cola} + d_{trans} + d_{prop}$$

d_{proc}: de procesamiento (~ μseg)
- comprobación cabecera del paquete
- determinación del enlace de salida

d_{trans}: de transmisión
= L / R
L: longitud del paquete (bits)
R: capacidad del enlace (bits/seg)

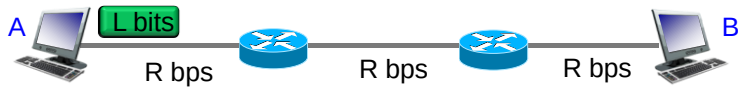
d_{cola}: de espera en cola (variable)
- espera a la salida para trasmisión
- depende del nivel de congestión

d_{prop}: de propagación
= long / s,
long: longitud del enlace físico (m);
s: velocidad de propagación en el medio ($2..3x10^8$ m/seg)

Ejemplo de cálculo del retardo total

Versión más detallada del ejemplo anterior

Envío de un paquete de L bits de A a B a través de 2 routers (3 enlaces)



Datos:

L = 3.000 bits

R = 1,5 Mbps (en los 3 enlaces)

$d_{proc} \approx 0$, $d_{cola} = 0$

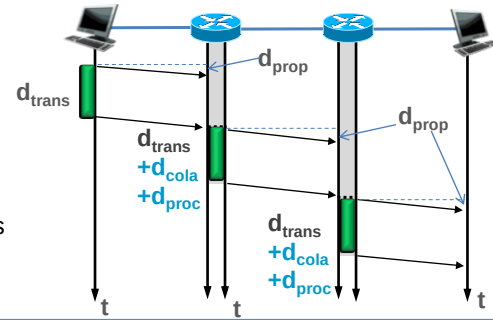
Cálculo del retardo total:

$d_{A-B} = 3 \times (d_{trans} + d_{prop})$

$d_{trans} = L/R = 3 \times 10^3 / 1{,}5 \times 10^6 = 2$ ms

$d_{A-B} = 6$ ms $+ 3 \times d_{prop}$

Cronograma:

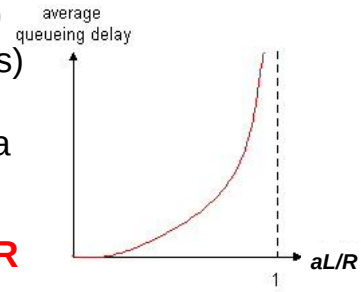


Espera en colas

R=capacidad del enlace (bps)

L=longitud de un paquete (bits)

a=velocidad media a la que llegan los paquetes a la cola (paquetes/segundo)

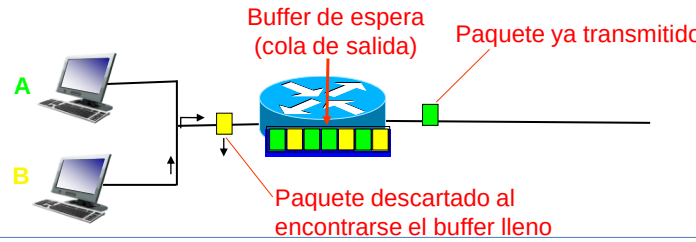


Intensidad de tráfico= a*L / R

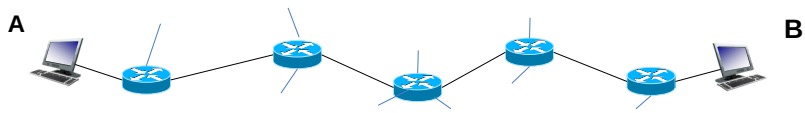
- **a *L / R ~ 0**: tiempo de espera 0 o muy pequeño (carga ligera)
- **a *L / R –>1**: tiempo de espera significativo , incrementándose cada vez más conforme se acerca a 1 (bastante carga)
- **a *L / R > 1**: la velocidad media de llegada excede la velocidad a la que los bits pueden ser transmitidos (congestión)

Pérdida de paquetes

- La capacidad de la cola es limitada
- Si llega un paquete y la cola está llena, el router elimina el paquete
  - El paquete se pierde
    - En ciertos casos, el sistema final origen retransmite
- El número de paquetes perdidos aumenta cuando la intensidad de tráfico aumenta



Retardo extremo a extremo

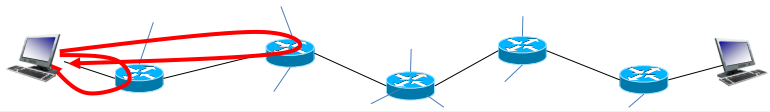


- Retardo total entre origen y destino
- Si hay N-1 routers entre A y B
  - y suponiendo que se envía un paquete de tamaño L, que la red no está congestionada ($d_{cola} \approx 0$), el retardo de proceso d_{proc} es igual para todos los routers y los enlaces tienen todos la misma capacidad R:

$$\mathbf{d_{A\text{-}B} = N * (d_{proc} + d_{trans} + d_{prop})}$$

Rutas y retardos en Internet

- **TRACEROUTE**: es un programa que puede ejecutarse en cualquier ordenador y que permite reconstruir la ruta seguida por los paquetes en Internet desde un origen hasta el destino y determinar el retardo de ida y vuelta de todos los routers en el camino. Para todo i:
  - El sistema origen envía tres paquetes que, en la ruta hacia el destino, alcanzarán al router i
  - El router i devolverá tres paquetes de contestación al origen
  - El origen registra el tiempo transcurrido desde que envió el paquete hasta que recibe el correspondiente mensaje de vuelta.



Rutas y retardos en Internet (ejemplo)

[ftel@jungla ~]$ **traceroute gaia.cs.umass.edu** **(desde jungla.dit.upm.es)**
traceroute to gaia.cs.umass.edu (128.119.245.12), 30 hops max, 38 byte packets

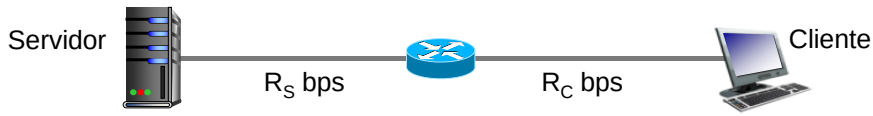
retardos de ida y vuelta 3 pruebas

1 fw.dit.upm.es (138.4.5.1) 0.163 ms 0.093 ms 0.138 ms
2 r7000-ext.dit.upm.es (138.4.0.1) 0.441 ms 0.399 ms 0.381 ms
3 138.100.254.13 (138.100.254.13) 3.486 ms 6.762 ms 9.985 ms
4 192.168.200.5 (192.168.200.5) 11.501 ms 0.352 ms 0.339 ms
5 XE1-0-0-108.EB-Madrid0.red.rediris.es (130.206.215.209) 1.658 ms 1.443 ms 1.429 ms
6 MAD.XE7-0-0.EB-IRIS4.red.rediris.es (130.206.250.21) 1.841 ms 1.465 ms 1.505 ms
7 rediris.rt1.mad.es.geant2.net (62.40.124.53) 1.717 ms 1.553 ms 1.482 ms
8 so-7-2-0.rt1.gen.ch.geant2.net (62.40.112.25) 23.705 ms 23.606 ms 23.565 ms
9 as0.rt1.fra.de.geant2.net (62.40.112.22) 31.778 ms 31.713 ms 31.706 ms
10 as1.rt1.ams.nl.geant2.net (62.40.112.58) 39.241 ms 39.015 ms 39.006 ms
11 102.rtr.newy32aoa.net.internet2.edu (198.32.11.50) 126.354 ms 126.071 ms 146.828 ms
12 nox300gw1-vl-110-nox-internet2.nox.org (192.5.89.221) 131.110 ms 130.911 ms 131.067 ms
13 noxgw1-peer-nox-umass-102.nox.org (192.5.89.102) 135.055 ms 134.685 ms 134.727 ms
14 lgrc-rt-106-8-gi1-6.gw.umass.edu (128.119.2.193) 134.975 ms 134.669 ms 135.038 ms
15 128.119.3.153 (128.119.3.153) 136.937 ms 136.931 ms 137.413 ms
16 nscs1bbs1.cs.umass.edu (128.119.240.253) 136.964 ms 136.414 ms 136.323 ms
17 gaia.cs.umass.edu (128.119.245.12) 135.682 ms 136.236 ms 135.489 ms

enlace transoceánico

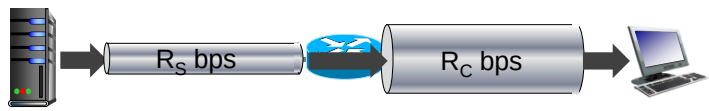
Caudal eficaz (1)

- Tasa a la que se transmiten los bits entre un emisor y un receptor (*throughput*)
  - C_{ef} = bits transmitidos / tiempo total (bits/segundo)
  - Extremo a extremo, medida de dos formas:
    - Instantáneo: en un cierto instante de tiempo
    - Tasa media: durante un periodo largo

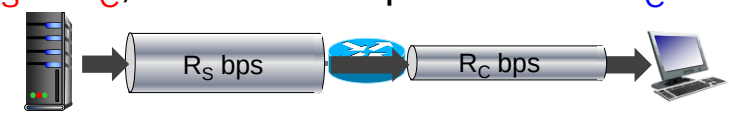


Caudal eficaz (2)

$R_S < R_C$, el caudal no podrá ser > R_S



$R_S > R_C$, el caudal no podrá ser > R_C



- El caudal eficaz se aproxima a mín{R_S, R_C}
  - ✓ … es decir, a la capacidad del enlace que hace de cuello de botella

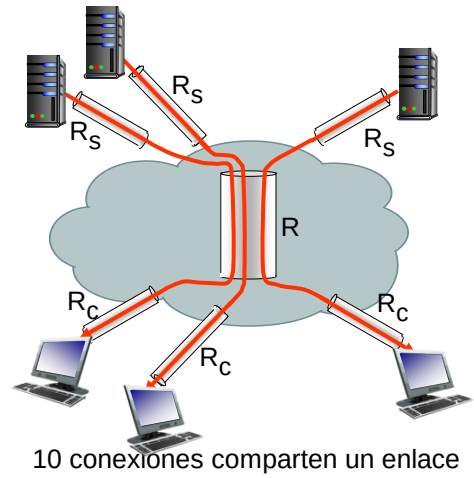
Ejemplo

Suponga que en la red de la figura, el cliente está descargando un archivo MP3 de L = 32 millones de bits, el enlace entre el servidor y el router tiene una capacidad R_s=2 Mbps y entre el cliente y el router R_c = 1 Mbps. Suponiendo que no hay más tráfico en los enlaces y sin tener en cuenta otros posibles retardos en la red, ¿cuál será, como mínimo, el tiempo T necesario para transferir el archivo?

$T = L / mín\{R_S, R_C\}$ = 32 segundos

Caudal eficaz (3)

- El caudal eficaz depende de:
  - las capacidades de los enlaces
    - Se aproxima a la capacidad mínima existente a lo largo de la ruta entre origen y destino
  - El tráfico existente
  - Los protocolos



10 conexiones comparten un enlace cuello de botella de R bps

Tema 4. Internet

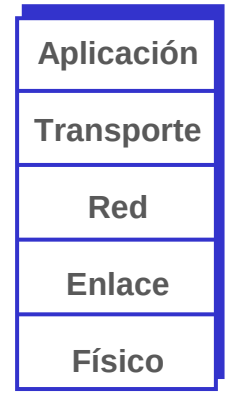
- Internet
- Núcleo de la red
- Conceptos de prestaciones
- Arquitectura de red
  - Modelos de capas, Internet, OSI
- Principios de las aplicaciones de red
- La web y HTTP
- DNS

Arquitectura de red

- Estructura jerarquizada de módulos que realizan todas las tareas involucradas en el intercambio de información
  - Proporciona una forma estructurada de estudiar los componentes del sistema
- Modelo de protocolos por niveles o capas
  - Cada protocolo pertenece a un nivel
  - Cada nivel ofrece unos servicios al nivel superior
- Pila de protocolos
  - Protocolos de las distintas capas tomados en conjunto

Protocolos de Internet (TCP/IP)

- Aplicación: donde residen los programas de aplicación
  - FTP, SMTP, HTTP, …
- Transporte: se ocupa de la transferencia entre procesos de aplicación
  - TCP, UDP
- Red: transferencia de datagramas entre SF origen y SF destino en la red o en la red de redes
- Enlace: transferencia de datos entre nodos vecinos sobre un medio físico
  - Ethernet, WiFi, PPP
- Físico: transferencia de bits individuales de un nodo al siguiente usando un medio físico



Existen diversas interpretaciones del modelo de capas de TCP/IP:
http://en.wikipedia.org/wiki/TCP/IP_model#Layer_names_and_number_of_layers_in_the_literature

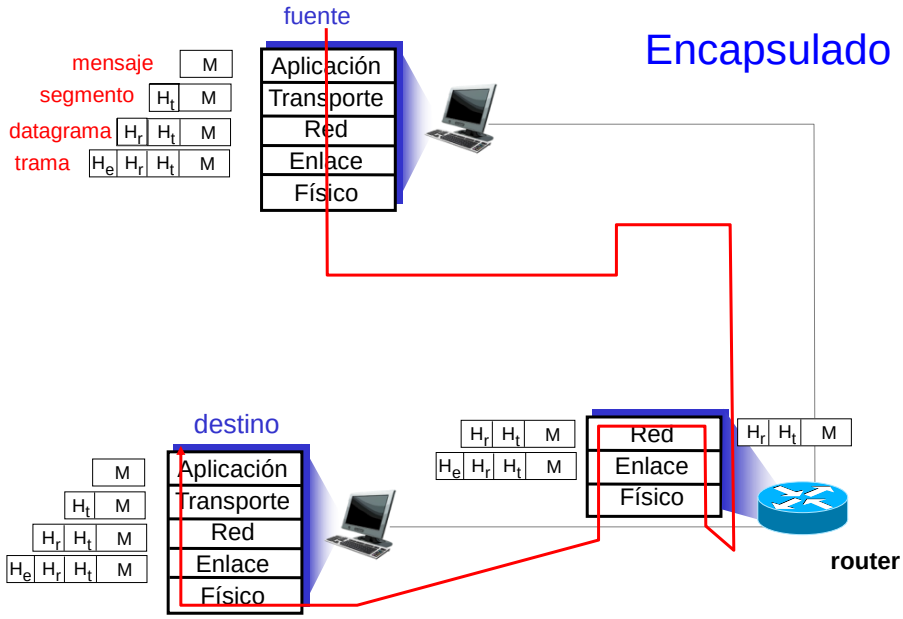
Modelo de referencia OSI/ISO

- Modelo OSI (Open Systems Interconnection)
  - Estandarizado por ISO (International Organization for Standardization) en 1984
- Existen diferencias con el modelo TCP/IP

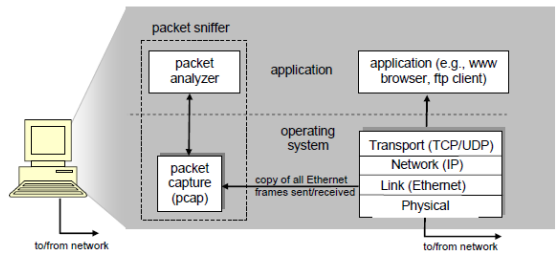
Capas del modelo

http://en.wikipedia.org/wiki/TCP/IP_model#OSI_and_TCP.2FIP_layering_differences

Encapsulado



Wireshark



- Wireshark es un analizador de protocolos
  - Captura los paquetes que se envían y reciben a través de alguna de las interfaces de red de un ordenador
  - Interpreta los datos de las cabeceras y muestra los valores de todos los campos de los protocolos que forman el paquete

Resumen de notación

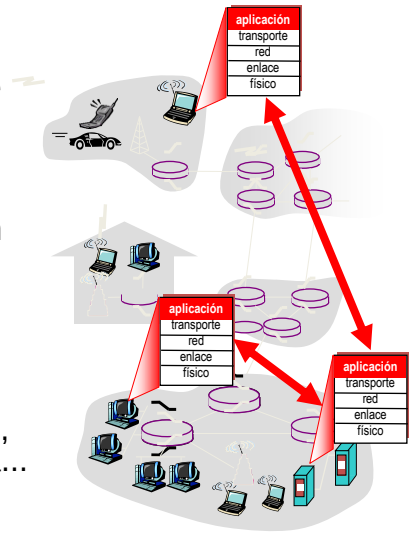
		Notación Kurose
Retardo de Proceso	segundos (s)	d_{proc}
Retardo de Espera en Cola	segundos (s)	d_{cola}
Retardo de Transmisión	segundos (s)	d_{trans}
Retardo de Propagación	segundos (s)	d_{prop}
Capacidad de un enlace	bits/segundo (bps)	R
Velocidad de propagación	metros/segundo(m/s)	s
Longitud de un enlace	metros (m)	long
Longitud de un paquete	bits	L

Tema 4. Internet

- Internet
- Núcleo de la red
- Conceptos de prestaciones
- Arquitectura de red
- Principios de las aplicaciones de red
  - Aplicaciones y protocolos
  - Procesos de aplicación
  - Arquitectura de aplicaciones
    - Cliente/servidor
    - Peer-to-peer (P2P)
    - Híbridas
  - Servicios de transporte
- La web y HTTP
- DNS

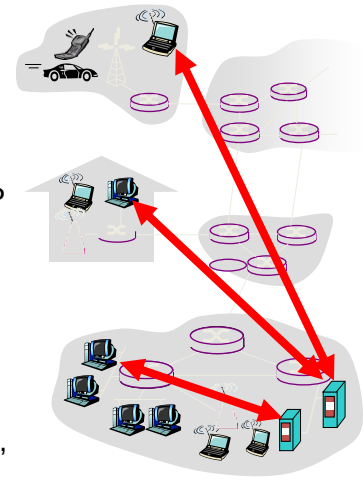
Aplicaciones y protocolos

- Aplicación de red
  - **procesos distribuidos** que se comunican entre sí e intercambian mensajes a través de la red
  - los procesos se ejecutan en los SFs
    - no en los routers
  - ej.: web, correo, VoD
- Protocolo de aplicación
  - define formato de mensajes, reglas, acciones, semántica...
  - ej.: http, smtp, snmp



Arquitectura cliente/servidor

- Aplicación distribuida típica: dos piezas, cliente y servidor.
- Cliente
  - inicia la conexión
  - realiza una petición y espera respuesta
  - en muchos casos, tiene dirección IP dinámica y conexión intermitente a Internet
- Servidor
  - siempre conectado y activo
  - espera conexiones y responde a peticiones de servicio
  - dirección IP permanente
  - los sitios populares instalan granjas, *clusters* o centros de datos



Centro de datos de Google, Hamina (Finlandia)

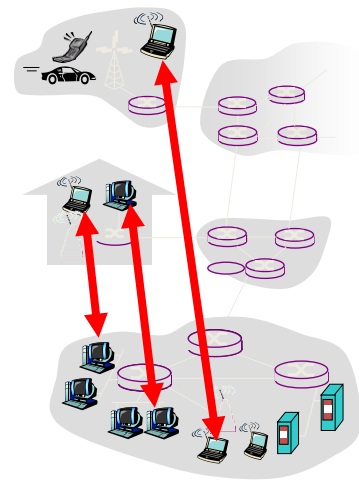
Servidores en un centro de datos

Centro de datos de Google, Oregon

Arquitectura *peer-to-peer* (p2p)

Arquitectura P2P pura

- Comunicación directa entre sistemas finales (*peers*)
  - Sin pasar por un servidor dedicado
- Muy escalable pero difícil de gestionar
- Peers generalmente con dirección IP dinámica y conexión intermitente a Internet
- Ejemplo: BitTorrent, PPLive



Arquitecturas híbridas cliente/servidor y P2P

Ejemplo: Skype

- Aplicación P2P de VoIP (*Voice over IP*)
- Servidor centralizado: encuentra las direcciones de participantes remotos
- Conexión cliente-cliente directa (sin intervención del servidor)

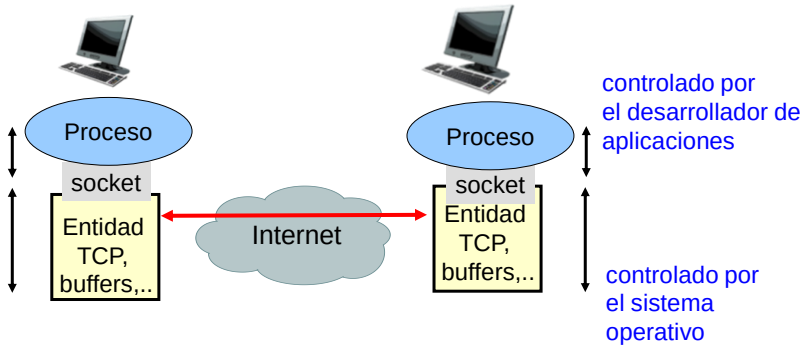
Ejemplo: mensajería instantánea

- Conversaciones entre clientes
- Servicio centralizado: localización y detección de presencia
  - Cuando un usuario se conecta, registra su dirección IP en un servidor central
  - Los usuarios encuentran direcciones IP de otros participantes a través del servidor central

Aplicaciones y procesos:terminología

- Proceso*:* es un programa en ejecución
- En diferentes SF, los procesos se comunican usando los servicios del nivel de transporte
  - intercambiando mensajes
  - a través de *sockets*
    - Interfaz entre la capa de aplicación y la capa de transporte (es decir, entre el proceso y la red)
    - Interfaz estándar de programación para desarrollo de aplicaciones (API)
    - Recurso del S.O.
  - y siguiendo las reglas de un protocolo de aplicación

Procesos de aplicación



Identificación de procesos

¿Cómo se identifica el proceso destino?

- **Dirección IP** (IPv4) del sistema final
  - 32 bits
  - Notación "punto": 138.4.27.10
- **Número de puerto**, identificador que permite discriminar entre los distintos procesos de aplicación locales en un mismo SF
  - 16 bits
  - 0 - 1023 son estándar: Well-known ports
    - ej.: HTTP=puerto 80, DNS = puerto 53, SMTP=puerto 25
- Por ejemplo, para enviar un mensaje HTTP al servidor web del DIT hay que indicar:
  - dirección IP: 138.4.2.61
  - número de puerto: 80

¿Qué servicios de transporte necesita una aplicación distribuida?

- Fiabilidad
  - algunas aplicaciones (ej. audio) pueden tolerar algunas pérdidas
  - otras aplicaciones (ej. trasferencia de ficheros) requieren una fiabilidad del 100% (no toleran pérdidas)
- Disponibilidad de caudal
  - algunas aplicaciones (ej. multimedia) requieren una mínima cantidad de caudal disponible para poder funcionar
  - otras aplicaciones ("aplicaciones elásticas") utilizan todo el caudal que pueden conseguir
- Límite estricto en retardos
  - algunas aplicaciones (ej. juegos interactivos) requieren que el retardo sea suficientemente bajo para poder funcionar
  - otras tienen requisitos estrictos (respuestas en *tiempo real*)
  - en otras los retardos no son importantes
- Seguridad
  - cifrado, integridad de los datos, …

Requisitos de las aplicaciones

Aplicación	Fiabilidad	Caudal	Sensible a retardo
transferencia de ficheros	si	elástica	no
e-mail	si	elástica	no
Web	si	elástica	no
audio/vídeo en tiempo real	tolerante a pérdidas	audio: 5Kbps-1Mbps vídeo:10Kbps-5Mbps	si
audio/vídeo bajo demanda	tolerante a pérdidas	audio: 5Kbps-1Mbps vídeo:10Kbps-5Mbps	si
juegos interactivos	tolerante a pérdidas	hasta 10Kbps	si
mensajería instantánea	si	elástica	si/no

Servicios de los protocolos de transporte

Servicio TCP

- *orientado a conexión*: requiere acuerdo entre los procesos emisor y receptor
- *transporte fiable* entre los procesos emisor y receptor
- *control de flujo*: el emisor no "agobia" al receptor
- *control de con*gestión: el emisor reduce la tasa de envío cuando detecta que la red se sobrecarga
- *no proporciona*: límite de retardo, garantía de un mínimo caudal , seguridad, …

Servicio UDP

- *transferencia de datos no fiable entre procesos*
- *no proporciona*: establecimiento de conexión, fiabilidad, control de flujo, control de congestión, límite de retardo, mínimo ancho de banda garantizado, seguridad, …

Aplicaciones de Internet: aplicaciones, protocolos de transporte

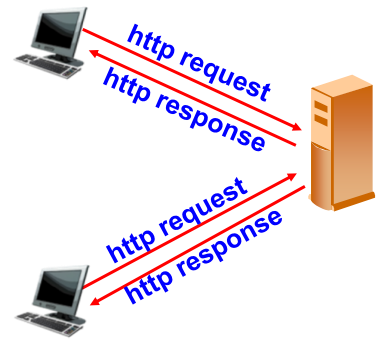
Aplicación	Protocolo de nivel de aplicación	Protocolo de transporte subyacente
correo electrónico	SMTP [RFC 5321]	TCP
acceso remoto a terminal	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
transferencia de ficheros	FTP [RFC 959]	TCP
flujos multimedia	HTTP (ej. YouTube) RTP [RFC 1889]	TCP o UDP
telefonía por internet	SIP, RTP, o propietario (ej. Skype)	normalmente UDP

Tema 4. Internet

- Internet
- Núcleo de la red
- Conceptos de prestaciones
- Arquitectura de red
- Principios de las aplicaciones de red
- La web y HTTP
  - Protocolo HTTP
  - Conexiones persistentes y no persistentes
  - Formato de los mensajes HTTP
  - Cookies, GET condicional y servidor proxy
- DNS

Protocolo HTTP

- Conviven dos versiones
  - HTTP/1.0: RFC 1945
  - HTTP/1.1: RFC 2616
- Protocolo **petición/respuesta**
  - sobre conexiones TCP
- En una "sesión":
  - el cliente inicia una conexión TCP, puerto 80
  - envía un mensaje de petición http de un objeto al servidor
  - el servidor devuelve un mensaje de respuesta http con el objeto
  - se cierra la conexión TCP
- Sin estados ("stateless")
  - el servidor no mantiene información sobre las peticiones del cliente



HTTP: persistencia

Conexión no persistente

- Una nueva conexión TCP por cada objeto contenido en el documento (html, jpg, gif, ...)
  - el servidor analiza la petición, responde y cierra la conexión
  - tiene influencia en las prestaciones
  - 2 RTT (Round Trip Time) por cada objeto transferido

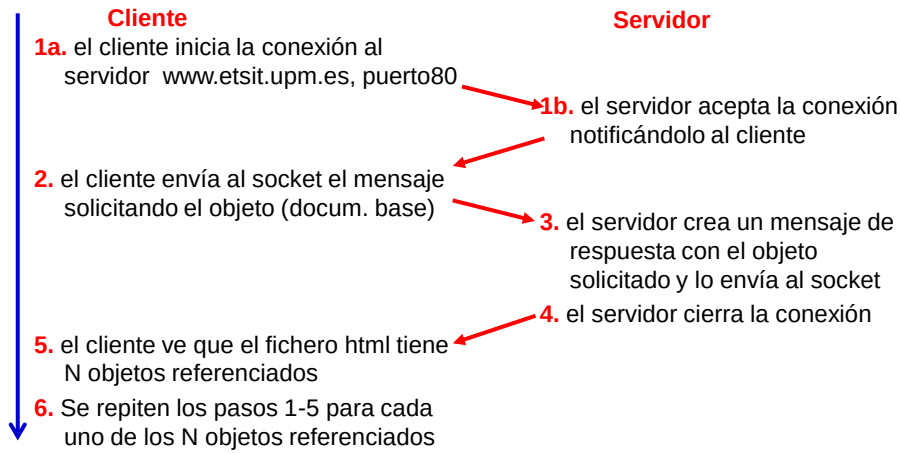
Conexión persistente

- Sobre la misma conexión TCP se transfieren varios objetos
- El cliente envía peticiones de los objetos referenciados
  - el servidor va respondiendo a cada una de las peticiones de forma secuencial
  - el cliente conoce los objetos cuando recibe el documento base HTML
  - menos RTTs

HTTP no persistente

Sobre cada conexión TCP sólo se envía un objeto, por tanto, hay que abrir una nueva conexión para cada objeto

Ejemplo: acceso a la página inicial de la Escuela

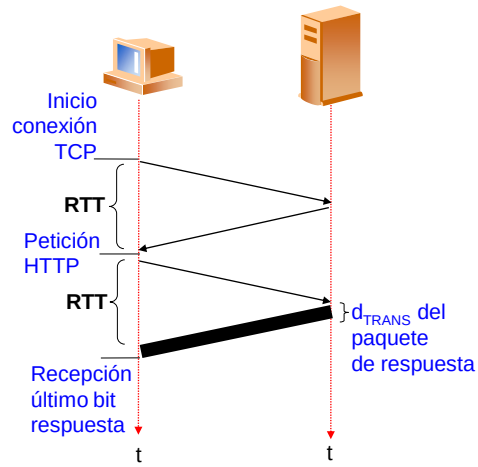


Modelado del tiempo de respuesta

Definición de RTT (*Round Trip Time*): tiempo que tarda un paquete (con d_{TRANS} despreciable) en hacer el recorrido de ida y vuelta entre cliente y servidor

Tiempo de respuesta:

$T = 2 \times RTT + d_{TRANS}$



Conexiones persistentes

- Si la conexión es persistente, existen dos formas de realizar peticiones de objetos:
  - secuencial
    - el cliente espera primero a recibir un mensaje de respuesta desde el servidor antes de enviar un nuevo mensaje de petición
  - *pipelining*
    - el cliente envía mensajes de petición tan pronto encuentra las referencias a los objetos en la página base

Formato de los mensajes HTTP

- Mensaje HTTP *request*
  - ASCII (caracteres legibles)
  - Método + URL + versión + cabecera + CR LF

Línea en blanco

- Métodos definidos:
  - GET: recupera el objeto que identifique URL
  - POST: el servidor acepta datos destinados a URL
  - HEAD: recupera sólo líneas de la cabecera
- URL:

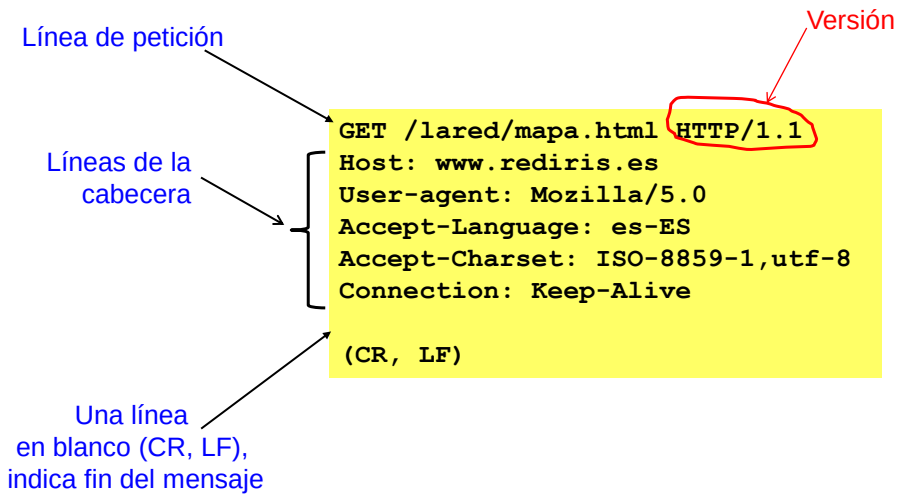
Sólo la ruta en el mensaje de request

`www.rediris.es/lared/mapa.html`

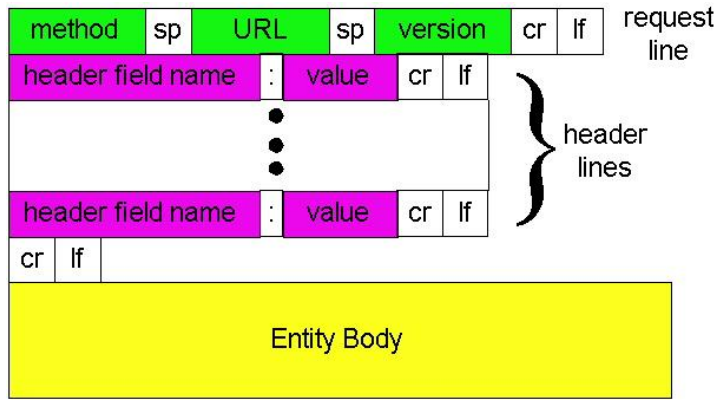
Nombre de servidor

Ruta (*path*) del objeto

HTTP *request*: ejemplo



HTTP *request*: formato general



HTTP *response*: ejemplo

- Mensaje **HTTP *response:***
  - cabecera + CR LF + cuerpo

Código de estado

Líneas de la cabecera

cuerpo, objeto pedido

```
HTTP/1.1 200 OK
Date: Tue, 15 Nov 2011 15:17:09 GMT
Server: Apache/2.2.3 (RedHat)
Last-Modified:  Fri, 23 Oct 2009 ...
Content-Length: 5479
Connection: close
Content-Type: text/html;charset=UTF-8

data data data data data ...
```

Códigos de estado

- Se devuelve un ***código de estado*** en la primera línea del mensaje HTTP *response*.

En general:

1xx	información
2xx	ok
3xx	redirección
4xx	error en cliente
5xx	error en servidor

Ejemplos de códigos:

`200 OK`
- éxito en la petición

`301 Moved Permanently`
- el objeto pedido ha cambiado de ubicación, la nueva ubicación se incluye en el propio mensaje de respuesta (**`Location`**:)

`400 Bad Request`
- el servidor no reconoce el mensaje de petición

`404 Not Found`
- el objeto pedido no se encuentra en el servidor

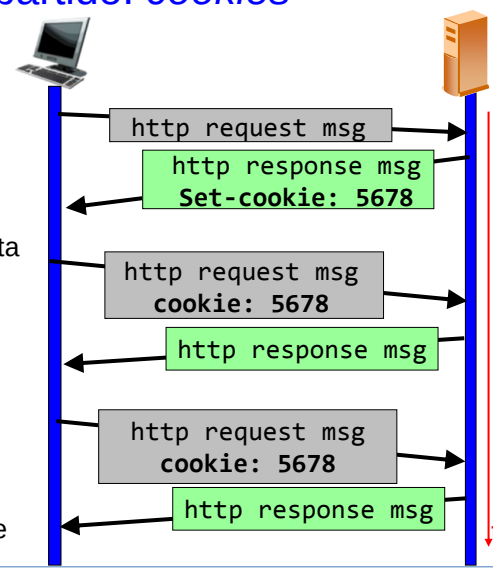
`505 HTTP Version Not Supported`

Estado compartido: *cookies*

- ***Cookies,*** RFC 2109
  - "Huellas" del cliente, autenticación, preferencias, ...
  - gestión del estado
- el servidor envía una *"cookie"* al cliente en un mensaje de respuesta

  `Set-cookie: 5678`
- el cliente devuelve la "cookie" en posteriores peticiones al mismo servidor

  `cookie: 5678`
- el servidor **establece correspondencia** con valores almacenados y reconoce al cliente



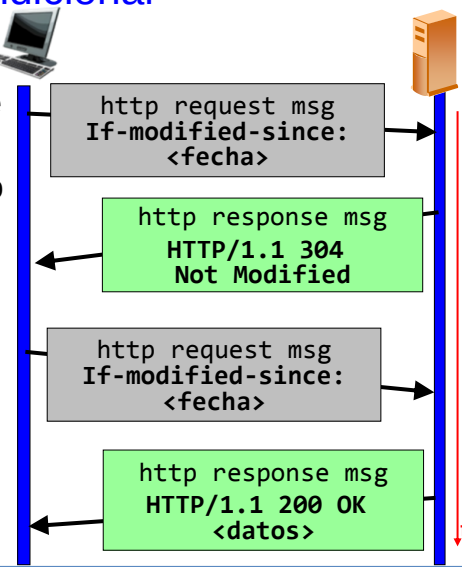
GET condicional

El **almacenamiento en cachés**, mejora los retardos y disminuye tráfico en la red

- GET condicional: enviar objeto sólo si la copia caché está obsoleta
- el cliente indica la fecha de la copia caché en petición

  `If-modified-since: <fecha>`
- la respuesta del servidor no incluye el objeto si la copia caché está actualizada

  `HTTP/1.1 304 Not Modified`

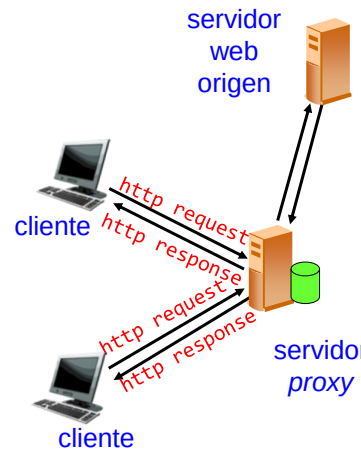


Caché web: servidor *proxy*

Objetivo: disminuir tráfico y retardos, enviando objetos al cliente sin involucrar al servidor origen.

Servidor Proxy:

- Guarda copias de páginas recientemente solicitadas
  - se debe configurar el navegador
- el cliente envía las peticiones al servidor proxy
  - si el objeto está en su caché, devuelve el mensaje de respuesta
  - en caso contrario, el proxy hace la petición al servidor origen y devuelve la respuesta al cliente



Tema 4. Internet

- Internet
- Núcleo de la red
- Conceptos de prestaciones
- Arquitectura de red
- Principios de las aplicaciones de red
- La web y HTTP
- DNS
  - Servicios y servidores DNS
  - Proceso de resolución de consultas
  - Registros de recursos

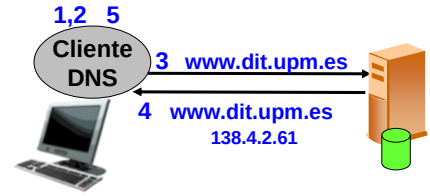
Domain Name System (DNS)

- Los sistemas finales y routers tienen asignada una **dirección IP**
  - cuatro octetos: 138. 4.7.189
- Para facilitar el manejo de las direcciones IP, a cada dirección se le asocia uno o más **nombres de dominio**
  - 138.4.7.189 jungla.dit.upm.es
- Se necesita un mecanismo de conversión: **DNS**
- **DNS**: servicio de directorio de Internet que traduce los nombres de dominio en direcciones IP
- Es una **base de datos distribuida**
  - se gestiona de forma descentralizada
    - jerarquía de numerosos *servidores de nombres*
  - orientada al uso por parte de aplicaciones
    - protocolo de aplicación para *resolución* de nombres
      - *resolución* = traducción de nombres en direcciones

Servicios proporcionados por el DNS

- traducción entre nombres de sistemas finales y direcciones IP
- alias de sistemas finales (host),
  - nombres más reconocibles que los *nombres canónicos*
- alias del servidor de correo
  - nombres simplificados (google.com, …)
- distribución de carga
  - servidores replicados con un solo nombre canónico y un conjunto de direcciones IP asociadas

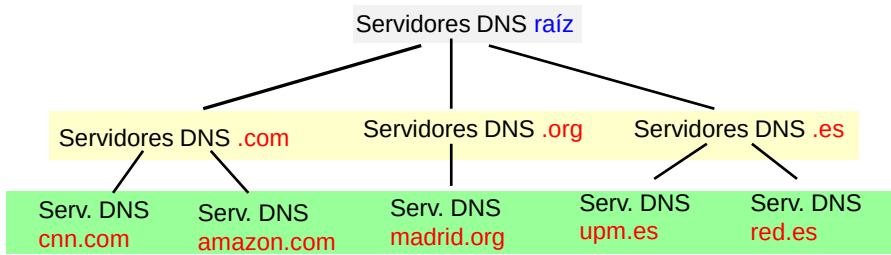
(RFC 1034, RFC 1035)



Ejemplo: *un cliente HTTP quiere acceder a la web www.dit.upm.es*

1. *Ejecuta el lado cliente del DNS*
2. *El navegador pasa al cliente DNS el nombre del host*
3. *El cliente DNS envía la consulta con el nombre del host a un servidor DNS*
4. *El cliente DNS recibe la respuesta que incluye la dirección IP*
5. *El navegador ya puede iniciar la conexión TCP con el servidor HTTP*

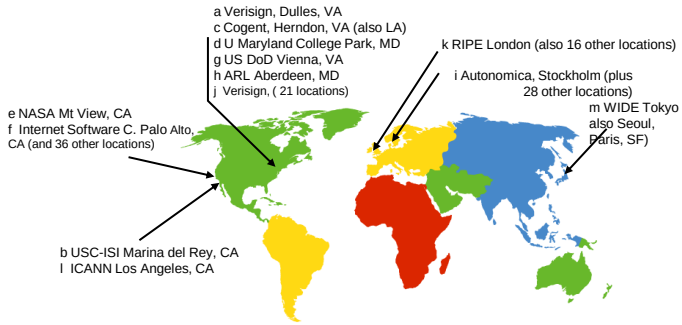
Base de datos distribuida y jerárquica



- Tres clases de servidores en la jerarquía:
  - Servidores DNS raíz
  - Servidores DNS de dominios de primer nivel (top level domains, TLD)
    - org, edu, net, com, int, info, pro, museum, aero, …
    - es, uk, de, mx, cl, cn, ru, …
  - Servidores DNS autoritativos
    - Mantenidos por las organizaciones e ISPs

Servidores raíz

- Conocen a todos los servidores de dominios de primer nivel
- Reciben consultas de servidores locales que no saben resolver un nombre
  - Devuelven referencias a servidores de dominios de primer nivel
- Hay 13 servidores raíz ubicados en distintos continentes
  - desplegadas réplicas en numerosos lugares

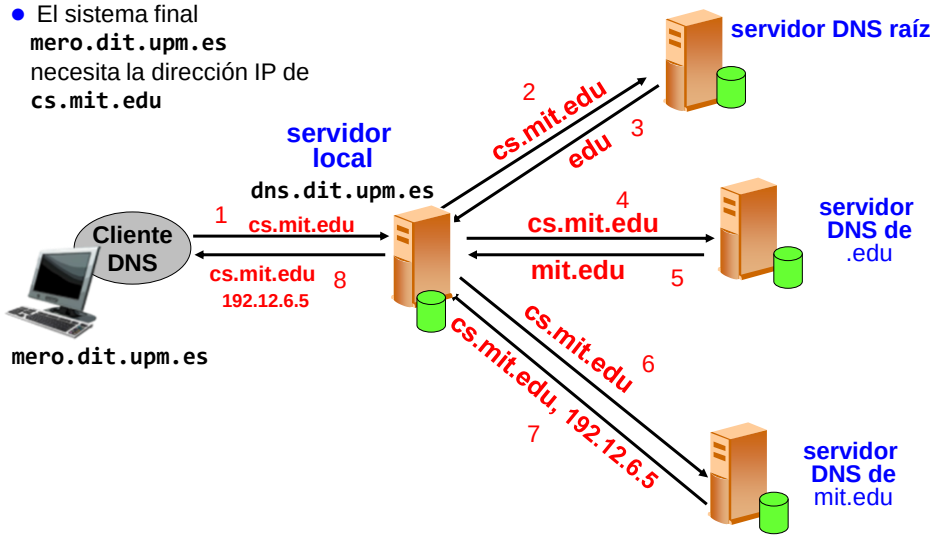


Servidor DNS local

- Cada organización o ISP tiene un servidor DNS local
  - también se llama predeterminado
- Las consultas para encontrar direcciones y nombres, se envían al DNS local
  - Actúa como proxy
  - Si no conoce la respuesta, reenvía la consulta a la jerarquía de servidores, comenzando por la raíz

Ejemplo de resolución

- El sistema final **mero.dit.upm.es** necesita la dirección IP de **cs.mit.edu**



Registros de recursos

- La BD distribuida del DNS está formada por registros de recursos (RR)

Formato RR: **(name, value, type, ttl)**

- Si `type=A` (Address)
  - **name** es un nombre de dominio
  - **value** es la dir IP
  - **(www.red.es 194.69.254.50 A)**
- Si `type=NS` (Name Server)
  - **name** es un nombre de dominio
  - **value** es el nombre de dominio de un servidor DNS autoritativo para ese dominio
  - **(red.es ns1.red.es NS)**
- Si `type=CNAME` (Canonical Name)
  - **name** es un alias
  - **value** es el nombre "real", el nombre canónico
  - **(www.rtve.es rtve.es.edgesuite.net CNAME)**
- Si `type=MX` (Mail eXchanger)
  - **value** es el nombre del servidor de correo asociado con **name**
  - **(red.es correo.red.es MX)**
- **ttl** *(Time To Live)*, cuánto tiempo un RR puede estar en la copia caché antes de ser descartado

```
[ftel@jungla ~]$ dig +trace cs.mit.edu
; <<>> DiG 9.3.1 <<>> +trace cs.mit.edu
.                   479214  IN      NS      b.root-servers.net.
.                   479214  IN      NS      a.root-servers.net.
.                   479214  IN      NS      i.root-servers.net.
.                   479214  IN      NS      j.root-servers.net.
.                   479214  IN      NS      k.root-servers.net.
;; Received 504 bytes from 138.4.2.10#53(138.4.2.10) in 1 ms

edu.                172800  IN      NS      c.edu-servers.net.
edu.                172800  IN      NS      l.edu-servers.net.
edu.                172800  IN      NS      f.edu-servers.net.
edu.                172800  IN      NS      a.edu-servers.net.
edu.                172800  IN      NS      d.edu-servers.net.
edu.                172800  IN      NS      g.edu-servers.net.
;; Received 263 bytes from 192.228.79.201#53(b.root-servers.net) in 202 ms

mit.edu.            172800  IN      NS      bitsy.mit.edu.
mit.edu.            172800  IN      NS      strawb.mit.edu.
mit.edu.            172800  IN      NS      w20ns.mit.edu.
;; Received 137 bytes from 192.26.92.36#53(c.edu-servers.net) in 115 ms

cs.mit.edu.         3600    IN      CNAME   EECS.mit.edu.
EECS.mit.edu.       3600    IN      A       18.62.1.6
mit.edu.            3600    IN      NS      STRAWB.mit.edu.
mit.edu.            3600    IN      NS      BITSY.mit.edu.
;; Received 172 bytes from 18.72.0.3#53(bitsy.mit.edu) in 132 ms
```



```
[ftel@jungla ~]$ dig +trace www.lab.dit.upm.es

; <<>> DiG 9.3.1 <<>> +trace www.lab.dit.upm.es
;; global options:  printcmd
.                   478890  IN      NS      l.root-servers.net.
.                   478890  IN      NS      m.root-servers.net.
.                   478890  IN      NS      c.root-servers.net.
.                   478890  IN      NS      h.root-servers.net.
.                   478890  IN      NS      b.root-servers.net.
.                   478890  IN      NS      i.root-servers.net.
.                   478890  IN      NS      d.root-servers.net.
.                   478890  IN      NS      g.root-servers.net.
.                   478890  IN      NS      j.root-servers.net.
.                   478890  IN      NS      a.root-servers.net.
.                   478890  IN      NS      k.root-servers.net.
.                   478890  IN      NS      f.root-servers.net.
.                   478890  IN      NS      e.root-servers.net.
;; Received 488 bytes from 138.4.2.10#53(138.4.2.10) in 1 ms

es.                 172800  IN      NS      a.nic.es.
es.                 172800  IN      NS      b.nic.es.
es.                 172800  IN      NS      f.nic.es.
es.                 172800  IN      NS      ns1.cesca.es.
es.                 172800  IN      NS      ns3.nic.fr.
es.                 172800  IN      NS      ns15.communitydns.net.
es.                 172800  IN      NS      ns-ext.nic.cl.
es.                 172800  IN      NS      sns-pb.isc.org.
;; Received 468 bytes from 2001:500:3::42#53(l.root-servers.net) in
80 ms

upm.es.             7200    IN      NS      galileo.ccupm.upm.es.
upm.es.             7200    IN      NS      sun.rediris.es.
upm.es.             7200    IN      NS      chico.rediris.es.
upm.es.             7200    IN      NS      einstein.ccupm.upm.es.
;; Received 197 bytes from 194.69.254.1#53(a.nic.es) in 3 ms

lab.dit.upm.es.     7200    IN      NS      dns.lab.dit.upm.es.
lab.dit.upm.es.     7200    IN      NS      dns2.dit.upm.es.
lab.dit.upm.es.     7200    IN      NS      dir.etsit.upm.es.
lab.dit.upm.es.     7200    IN      NS      dns.dit.upm.es.
lab.dit.upm.es.     7200    IN      NS      galileo.ccupm.upm.es.
;; Received 263 bytes from 138.100.4.4#53(galileo.ccupm.upm.es) in 3
    ms

www.lab.dit.upm.es. 7200    IN      A       138.4.26.58
lab.dit.upm.es.     18000   IN      NS      dir.etsit.upm.es.
lab.dit.upm.es.     18000   IN      NS      galileo.ccupm.upm.es.
lab.dit.upm.es.     18000   IN      NS      dns.lab.dit.upm.es.
lab.dit.upm.es.     18000   IN      NS      dns.dit.upm.es.
lab.dit.upm.es.     18000   IN      NS      dns2.dit.upm.es.
;; Received 323 bytes from 2001:720:1500:42::32#53(dns2.dit.upm.es)
    in 0 ms
```

Fundamentos de los Sistemas Telemáticos

Curso 2014-2015

Laboratorio 7. Analizador de protocolos Wireshark. Pila de protocolos TCP/IP. La orden `traceroute`.

Objetivos:

- Familiarización con la herramienta de análisis de protocolos Wireshark.
- Comprensión del apilamiento de protocolos y el encapsulado de PDUs en la pila TCP/IP.
- Identificación de información relevante en las PDUs y sobre el comportamiento de los protocolos: direcciones IP origen y destino, longitudes de cabeceras IP y TCP.
- Uso de la orden traceroute para obtener la ruta que siguen los paquetes IP. Uso de la orden ping para obtener la misma información.

Realización:

- La práctica se realizará en Linux y en el laboratorio del DIT A-127 (los resultados que se piden dependen de en qué ordenadores se realiza la práctica, por lo que **debe** hacerla en el laboratorio del DIT).
- La duración estimada es de una hora.
- Antes de realizar la práctica, el alumno habrá leído la sección del capítulo 1 del libro "Computer Networking: a Top Down Approach" dedicada a la herramienta traceroute. También debe haber leído la introducción y la segunda sección de este artículo dedicado a la orden ping: http://en.wikipedia.org/wiki/Ping_%28networking_utility%29

Resultados:

- Debe subir a moodle un fichero ZIP que incluya el formulario con los resultados pedidos en este enunciado, así como dos ficheros con captura de pantalla que se piden en las tareas 4 y 5. En la tarea de entrega se indican los detalles de la subida de resultados.

Tareas a realizar.

- Tarea 1. Obtención, desde un terminal, de la dirección IP de su ordenador y del enrutador que está usando.
  - Abra una ventana de terminal y dé la orden **`ifconfig eth0`**
  - La salida de dicha orden muestra varias informaciones. El texto que hay a continuación de "inet addr" es la dirección IPv4 de su ordenador. Anote la dirección IP de su ordenador.
  - En el terminal, dé la orden **`netstat -rn`**
  - Se le mustra una tabla con varias filas. Busque la fila con destino ("Destination") 0.0.0.0 y máscara de red ("Genmask") 0.0.0.0. Anote la dirección IP de la pasarela ("Gateway"). Es la dirección IP del enrutador por defecto de su ordenador.

- Tarea 2. Arranque del analizador e inicio de la captura de tráfico.
  - Vaya al menú de Aplicaciones→Laboratorios docentes del DIT y encontrará una entrada para el programa "Wireshark". Inícielo.
  - Si le aparece una ventana pequeña que le avisa de que está ejecutando el programa como "root" (administrador), indique que no desea volver a recibirlo y pulse el botón de OK

(mientras no lo haga, el analizador no responderá a ningún otro tipo de orden). Si le aparece otra ventana que avisa con el mensaje "Lua: Error during loading...", pulse OK.
    - Arranque el navegador (se pide que se haga antes de iniciar la captura con Wireshark para evitar capturar el tráfico de la página de inicio).
    - En la ventana del analizador, a la izquierda, aparece una lista de "conexiones de red" (interfaces) de las que puede capturar. Arranque una captura de tráfico marcando con el ratón la interfaz `eth0` y pulsando a continuación "Start".

- Tarea 3. Generación de tráfico y su captura.
    - En el navegador, vaya a la página http://ftel-prac.lab.dit.upm.es/lab7.php
    - Cuando la página se haya mostrado, detenga la captura de tráfico en el analizador pulsando el botón rojo cuadrado, que está en la parte superior.
    - Aplique un filtro de presentación para mostrar sólo el tráfico con el servidor web. Para ello, escriba **`http and ip.addr == 138.4.17.254`** en el espacio junto a "*Filter*" y pulse el botón "*Apply*".

- Tarea 4. Estudio del protocolo IP.
    - En el panel superior del analizador, donde se muestran los paquetes capturados, busque los paquetes cuya dirección IP origen sea la de su ordenador.
    - Examine el análisis la cabecera del protocolo IP del primero de dichos paquetes. Tendrá que usar la información que proporciona el panel intermedio; el detalle de cada protocolo se consigue picando con el ratón sobre los signos "▷" que apuntan a cada protocolo; para "colapsar" de nuevo la información, se vuelve a picar sobre el "▷" que se expandió).
    - Para ese paquete anote la longitud de la cabecera IP y la longitud total del paquete IP.
    - En el panel intermedio, pique con el ratón sobre la línea que empieza con "▽ Internet Protocol version 4". Observe que en el panel inferior se marcan en azul varios octetos. Son la cabecera IP. Haga una captura de pantalla en la que el analizador muestre dichos octetos pulsando la tecla ImprPant. La imagen capturada la puede encontrar en la carpeta Imágenes de su directorio personal. Renómbrela a tarea4.png.

- Tarea 5. Estudio del protocolo de transporte.
    - Identifique el protocolo de transporte que se está usando en el paquete examinado.
    - En el panel intermedio, pulse sobre la línea que empieza por "▷" correspondiente a la capa de transporte y observe la cabecera de transporte en el panel inferior. Haga una captura de pantalla en la que se muestren los octetos de la cabecera de transporte. Renombre el fichero de la imagen capturada a tarea5.png. ¿En dónde se encuentra la cabecera de transporte en relación con la cabecera del protocolo IP?
    - Examine ahora la cabecera de transporte del paquete estudiado en el apartado anterior. Dé la longitud de su cabecera.
    - Anote los puertos origen y destino del paquete examinado. ¿Qué protocolo es el que está asociado al puerto destino?

- Tarea 6. Uso de la orden traceroute. Esta orden nos va a permitir conocer la ruta que sigue el tráfico IP entre su ordenador y la máquina que se especifica como destino.
    - Abra una ventana de terminal y en ella dé la orden siguiente:
      `traceroute -N 1 ftel-prac.lab.dit.upm.es`
    - Incluya la salida de dicha orden en el formulario de respuesta ¿A cuántos saltos se encuentra la máquina ftel-prac.lab.dit.upm.es de su ordenador?
    - ¿Cuál es la dirección IP del equipo que se encuentra a un salto de distancia de su ordenador? ¿En qué tarea anterior le ha aparecido esta dirección? ¿Por qué aparece aquí también?
    - Pregunte a un compañero la dirección IP de su puesto de laboratorio y haga un traceroute a dicha dirección (traceroute *pongaAquíLaDirIPDelPCDeSuCompañero*). ¿A cuántos saltos está dicha máquina de su ordenador? ¿La máquina que está a un salto de distancia es la misma que en el apartado anterior? ¿Por qué?

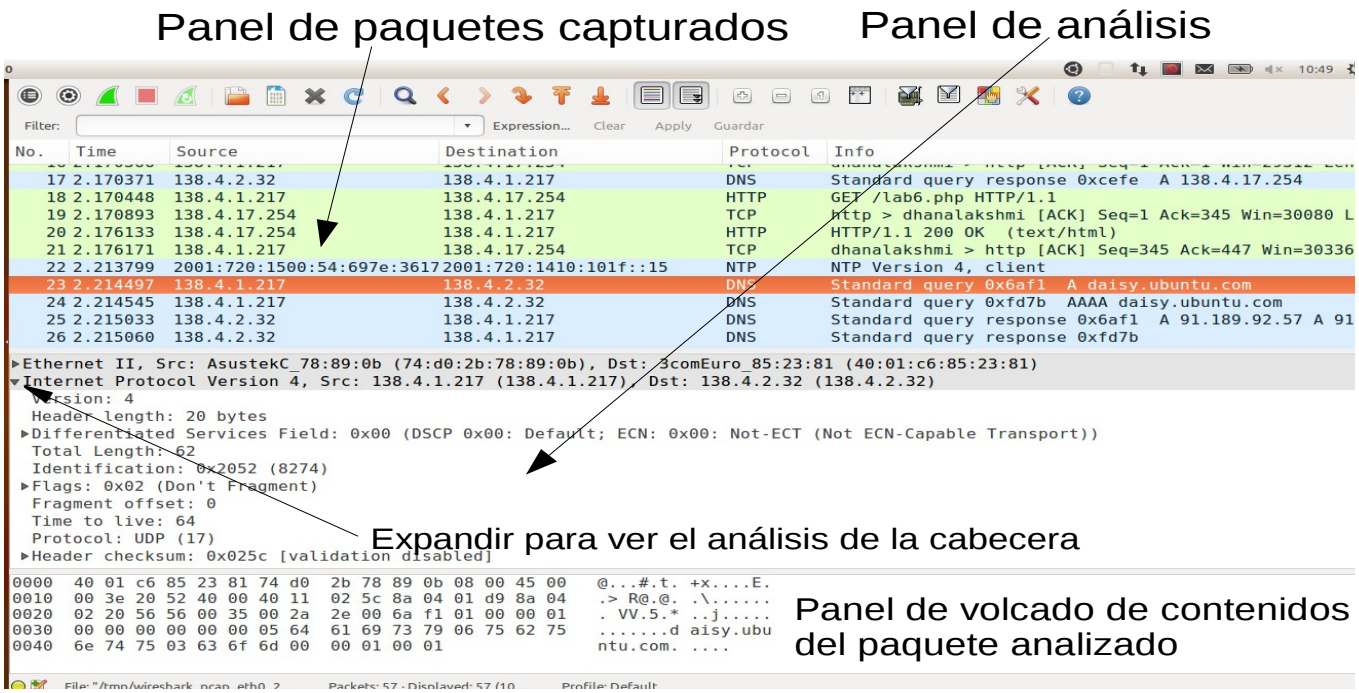
- Durante la sesión del laboratorio el profesor indicará el nombre de un servidor web al que se le realizará un traceroute. En la ventana de terminal dé ahora la orden siguiente:
  `traceroute -N 1 XXXXXX`
  Siendo XXXXXX el nombre que ha indicado el profesor. Incluya la salida de dicha orden en el formulario de respuesta.
- ¿A cuántos saltos se encuentra de su ordenador el servidor indicado?
- ¿Entre qué máquinas (dé sus nombres y direcciones IP) se produce el mayor incremento del RTT? Indique una posible explicación de este incremento.

NOTA: la opción -N de traceroute permite controlar el número de mensajes de sonda que se envían simultáneamente por el programa traceroute. Por defecto, traceroute envía un número grande, que en algunos cortafuegos se puede descartar. Al limitar ese número, los cortafuegos no ven un tráfico tan agresivo y baja la probabilidad de descarte.

- Tarea 7. Uso de la orden ping. La orden ping admite varias opciones, como cambiar el tamaño de los mensajes que se envían, indicar cuántos mensajes de sonda se envían, o controlar cuántos "saltos" como máximo (enlaces por los que transita) puede experimentar el paquete en su ruta hacia el destino especificado. En esta tarea se usarán las dos últimas opciones. Indicaremos a la orden ping que envíe tres mensajes de sonda con la opción -c 3. Si *N* es el número máximo de saltos que permitimos, en Linux se especifica con la opción -t *N*. De este modo, la orden a dar es (las cadenas en cursiva se deben sustituir por los valores adecuados):
  `ping -c 3 -t *N nombre.del.sitio*`
  - Haga un ping al sitio ftel-prac.lab.dit.upm.es, especificando la opción de número máximo de saltos, con un valor inicial de 1 y tres paquetes de sonda.
  - ¿Qué mensajes le presenta ping? ¿Ve alguna información relacionada con la respuesta de alguna tarea anterior?
  - Incremente uno a uno el valor de *N* hasta obtener contestación de ftel-prac.lab.dit.upm.es. NOTA: la contestación con éxito debe ser de la forma "64 bytes from ftel-prac.lab.dit.upm.es …".
  - ¿Con qué valor del número de saltos obtiene respuesta del servidor?
  - ¿Qué función realizan las máquinas que contestan con el mensaje de número de saltos insuficiente para llegar al destino final?
  - ¿Puede deducir cómo opera el programa **`traceroute`** a la vista de lo estudiado aquí?

Apéndice. Pantalla principal de Wireshark.

Como orientación, la imagen siguiente muestra el aspecto del analizador tras haber terminado una captura, viendo el detalle del protocolo IP de paquete nº 23 que aparece seleccionado en el panel superior.



Laboratorio 8. HTTP y DNS.

Objetivos:

- Estudio del protocolo HTTP. Examen de algunos de los campos de la cabecera HTTP.
- Estudio de las consultas al DNS.

Realización:

- De la práctica anterior se conoce el manejo básico de la herramienta Wireshark.
- El alumno habrá leído el enunciado antes de realizar la práctica y habrá repasado los conceptos expuestos en la teoría sobre HTTP y DNS.
- La práctica se realizará en el laboratorio del DIT y en Linux. La duración estimada es de una hora.

Recursos:

- PC+Linux con configuración de red operativa y el software Wireshark instalado.
- Navegador Chrome.

Resultados:

- Durante el turno de laboratorio se debe subir a moodle un fichero ZIP que incluya el formulario de resultados de la práctica en moodle, y varios ficheros de captura de pantalla que se piden en el enunciado.

Estudio básico del protocolo HTTP.

- Tarea 1. Arranque del analizador e inicio de la captura de tráfico.
  - Arranque el analizador Wireshark (en el menú de Aplicaciones→Laboratorios docentes del DIT→Utilidades). Si le aparece una ventana pequeña que le avisa de que está ejecutando el programa como "root" (administrador), indique que no desea volver a recibirlo y pulse el botón de OK (mientras no lo haga, el analizador no responderá a ningún otro tipo de orden).
  - Arranque el navegador Chrome (se pide que se haga antes de iniciar la captura con Wireshark para evitar capturar el tráfico de la página de inicio).
  - En la ventana del analizador, a la izquierda, aparece una lista de "conexiones de red" (interfaces) de las que puede capturar. Para arrancar la captura, pique con el ratón sobre la interfaz `eth0` o `eth1` (según la que se use en su ordenador para salir a Internet, como ya hizo en el laboratorio 7).
- Tarea 2. Generación de tráfico y su captura.
  - Vacíe la caché de su navegador, para eliminar posibles copias de los ficheros descargados si ha accedido antes a la página. En el navegador Chrome, pulse con el ratón sobre el símbolo de tres rayas horizontales a la derecha de la barra de dirección, vaya al menú de Configuración, muestre las opciones avanzadas, vaya a Privacidad→Borrar datos de navegación, marque “Archivos e imágenes almacenados en caché” y seleccione “Eliminar elementos almacenados desde el origen de los tiempos”. Pulse el botón “Borrar datos de navegación”.
  - A continuación, vaya a la página de dirección http://ftel-prac.lab.dit.upm.es/lab8.php
  - Cuando la página se haya mostrado, detenga la captura de tráfico en el analizador pulsando el icono con un botón rojo con un aspa blanca, en la parte superior. No cierre el navegador.
  - Aplique un filtro de presentación para mostrar sólo el tráfico HTTP de interés para la práctica. Para ello, escriba lo siguiente en el espacio junto a "*Filter*" y pulse "*Apply*":
    `http and (ip.addr == 138.4.17.254 or ip.addr == 138.100.17.2)`
- Tarea 3. Estudio de las peticiones HTTP. En la captura realizada se van a estudiar las peticiones

HTTP que se realizan.

- ¿Cuántas peticiones HTTP GET realiza su navegador web? Para cada una de ellas, escriba cuál es el recurso que se está solicitando y la dirección IP del servidor.
- En el panel superior de Wireshark seleccione el paquete que realiza la petición GET correspondiente al recurso lab8.php. Vaya ahora al panel intermedio de Wireshark y expanda el análisis del protocolo de aplicación. Observe que en la cabecera HTTP aparecen diversos campos que describen aspectos relacionados con la petición. Identifique:
  - Versión del protocolo que usa el navegador (HTTP1.0 o HTTP1.1).
  - ¿Cuál es la identificación del navegador que realiza la petición? ¿Cuál es el nombre de la línea de cabecera en la que va?
  - ¿Cuál es el nombre de la línea de cabecera que identifica la máquina a la que se hace la petición?
  - ¿Cuáles son los idiomas en los que se prefiere recibir la información que envíe el servidor? ¿Cuál es el nombre de la línea de cabecera donde se indican?
  - ¿Qué lenguajes de contenidos acepta el navegador? ¿Cuál es el nombre de la línea de cabecera donde se indican?
  - Haga una captura de pantalla en la que se muestre el panel intermedio del analizador y se observe la información anterior. Guarde la captura en un fichero de nombre tarea3.png.

- Tarea 4. Estudio de las respuestas a la petición HTTP. Examine ahora el detalle de los paquetes que corresponden a las respuestas del servidor. Concretamente, identifique, para la respuesta que corresponde al logotipo de la ETSIT:
  - Dirección IP del servidor que responde.
  - Código de la respuesta.
  - Información acerca del tipo de servidor web que responde (Netscape, Apache, etc.), y nombre de la línea de la cabecera que la proporciona.
  - Formato del contenido descargado. Nombre de la línea de la cabecera donde se indica dicho formato.
  - Tamaño del fichero del logotipo y dónde ha encontrado dicha información (que debe haberse obtenido usando Wireshark).
  - Haga una captura de pantalla en la que se muestre el panel intermedio del analizador y se observe la información anterior. Guarde la captura en un fichero de nombre tarea4.png.

Estudio básico del DNS.

- Tarea 5. Estudio de las peticiones y respuestas DNS.
  - Antes de hacer la captura pedida en este apartado, en un termnal debe dar la siguiente orden:

    ```
    sudo /usr/sbin/nscd -i hosts
    ```

    La orden anterior borra la caché DNS del ordenador (es decir, “olvida” las correspondencias entre nombres y direcciones que pueda haber aprendido). Si por cualquier motivo debe repetir el acceso a la página web y la captura de tráfico, en este apartado debe haber hecho esa orden inmediatamente antes.
  - Arranque una captura de tráfico con Wireshark.
  - En su navegador, vuelva a cargar la página http://ftel-prac.lab.dit.upm.es/lab8.php (debe forzar que se vuelve a pedir la página al servidor pulsando con el ratón sobre el símbolo de “recarga” que hay al final del cuadro de texto de la dirección web visitada).
  - Detenga la captura de Wireshark. En el cuadro de texto del filtro de presentación escriba el siguiente filtro:

    ```
    (dns and ip.addr == 138.4.30.1) or (http and (ip.addr == 138.4.17.254 or ip.addr == 138.100.17.2))
    ```

    y pulse el botón “Apply”.
  - Estudie los paquetes DNS que envía su ordenador (la dirección IP origen debe ser la de su ordenador). ¿Por qué tipo de registro DNS se está preguntando? ¿Qué información se quiere

obtener con dichas peticiones?

- Estudie ahora las respuestas que le llegan. ¿Qué máquina las genera? ¿Qué información proporcionan? ¿Qué relación tiene esa información con las comunicaciones HTTP que ha examinado en los apartados anteriores?
- Haga una captura de pantalla en la que se muestre el panel intermedio del analizador y se observe la información de alguna de las respuestas. Guarde la captura en un fichero de nombre tarea5.png.

- Tarea 6. Realización de consultas iterativas al DNS. Se va a usar la herramienta "dig" para estudiar el tipo de consultas que realiza un servidor de DNS al que le llega una petición de un cliente para resolver un nombre que no es de su dominio. En las distribuciones de Linux suele estar disponible. En todo caso, existen también sitios web que ofrecen acceso a dicha herramienta, para no depender de tenerla instalada en el ordenador, y en la práctica se usará este procedimiento, ya que el cortafuegos del laboratorio prohíbe las consultas a servidores externos.
  - En el navegador, vaya a la dirección http://www.digwebinterface.com/
  - En el cuadro de texto "Hostnames or IP addresses", escriba el nombre DNS del servidor web que el profesor indicará durante la sesión de laboratorio.
  - En la columna "Options", marque la opción "trace".
  - Pulse el botón "Dig".
  - Incluya en el formulario de la práctica la salida de la orden anterior.
  - Indique cuál es la dirección IP del servidor web buscado.
  - Observando los resultados de la orden, describa cómo se obtiene la dirección de un servidor DNS autoritativo del dominio del servidor web.
  - ¿El nombre del servidor web que ha indicado el profesor es un “alias”? ¿Por qué?

Fundamentos de los Sistemas Telemáticos
Tema 5: Bases de Datos

Bases de datos

- **Bases de datos relacionales**
  - Sistemas de información. Definiciones. Modelo de datos
  - Modelo Relacional.
    - Conceptos básicos: relación, tupla, atributos, claves
  - Lenguaje SQL.
    - Operadores para crear/modificar relaciones y atributos con SQL
    - Operador select para realizar consultas con SQL
- **Diseño de una base de datos: del modelo ER al modelo relacional**
  - Modelo Entidad-Relación. Conceptos. Diferencias con el modelo relacional
  - Diseño conceptual con el modelo ER
  - Proceso de transformación de un diagrama ER a un esquema relacional

Material de estudio y trabajo:

- Estas transparencias
- Ejercicios y prácticas de laboratorio propuestos

Sistemas de Información

Sistema de información = Software que ayuda a organizar los datos

Todos los programas de aplicación tienen que manipular datos,

- si los datos no son numerosos y sólo van a ser usados en una aplicación, se podrían tratar usando el **sistema de gestión de ficheros** incluido en el S.O.
- pero en general es mucho más útil usar **sistemas de información** predefinidos, que proporcionan estructuras adecuadas para guardar los datos y operaciones útiles para procesarlos y gestionarlos. Especialmente útiles son:
  - **Bases de datos**
    - ✓ Cada vez más utilizadas, especialmente en Internet (librerías digitales, sitios de recomendación, video interactivo, proyecto Genoma Humano, …)
  - **Hojas de cálculo**
    - ✓ solución más simple, usada fundamentalmente para datos numéricos

Definiciones

Base (D.R.A.E)

~ de datos.

1. f. *Inform.* Conjunto de datos organizado de tal modo que permita obtener con rapidez diversos tipos de información.

Base de Datos

Colección integrada de datos estructurados, interrelacionados, almacenados en un soporte informático, independientes de los usuarios y aplicaciones que los utilicen.

Base de Datos relacional

Una Base de Datos en la que todos los datos están organizados siguiendo el modelo relacional, es decir, como una colección de tablas que representan las relaciones que existen entre los datos, sobre las cuales se realizan todas las operaciones.

Sistema de Gestión de Base de Datos (SGBD o DBMS)

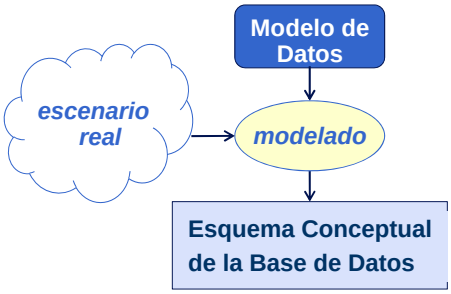
Una aplicación software que incluye procedimientos para actualizar y recuperar datos de una Base de Datos, manteniendo su integridad, seguridad y confidencialidad.

Modelo de Datos

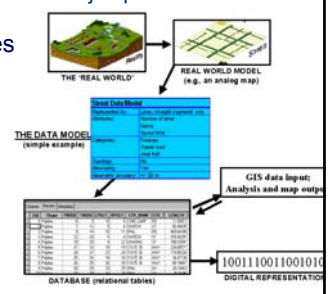
Un **modelo de datos** es una colección de notaciones (gráficos, tablas, …) que facilitan la representación de un escenario real, identificando las cosas (objetos o entidades) que son relevantes para el caso y cómo se relacionan estas cosas.

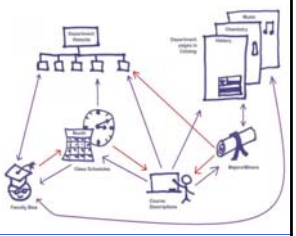
Modelando la realidad con un modelo de datos se consigue:

- dar una visión comprensible para los usuarios
- independizar los datos de cualquier aplicación que los use y de cualquier método o forma de almacenarlos.



Ejemplos de modelado





Modelos de datos usados en Bases de Datos

Para definir el esquema conceptual de una base de datos se pueden usar diferentes modelos de datos

- Una gran mayoría de sistemas de gestión de bases de datos utilizan el denominado **Modelo Relacional → Bases de Datos relacionales**
  - un modelo formal (basado en lógica de predicados y teoría de conjuntos) que facilita el modelado de las organizaciones del mundo real (*p.ej. una universidad*) describiendo las entidades que la forman (*p.ej. alumnos, asignaturas, aulas…*) y sus relaciones (*p.ej. Ana está matriculada en Ftel en el grupo G16*).
- No es fácil pasar directamente del escenario real a un esquema relacional (un esquema conceptual expresado en términos del modelo relacional)
  - por eso se suelen usar modelos de datos intermedios, siendo el más conocido el **Modelo Entidad-Relación** (Modelo ER)
    - primero se define un esquema gráfico siguiendo el modelo Entidad-Relación
    - luego se va transformando este esquema gráfico en un esquema relacional

Modelo relacional

Modelo de datos lógico más utilizado actualmente, definido por Codd en 1970.

Concepto básico: **Relación** R(A1, …,Ai,...An), un conjunto de datos organizados en una tabla compuesta de *filas* (*tuplas*, o *registros*) y *columnas* (*atributos*, Ai, o *campos*), en la que cada intersección de fila y columna contiene un valor (ej. *asignatura*).

Atributo: representa cada una de las características que definen un objeto del mundo real; tiene asignados un nombre (ej. *curso*) y un dominio (ej. *enteros*)

Tupla: conjunto de valores que toman todos los atributos de un objeto concreto
ej. la *asignatura* con *codigo=95000002, siglas=CALC, nombre=Calculo, curso =1*

asignatura

codigo	siglas	nombre	curso
95000001	ALGE	Algebra	1
95000002	CALC	Calculo	1
95000003	FIS1	Fisica General 1	1
95000004	IING	Introd. a la Ingenieria de Teleco	1
95000005	FTEL	Fund. de los Sist. Telematicos	1

atributos

tuplas

dominios: enteros caracteres texto enteros

Características asociadas al modelo relacional

Esquema (o intensión), es la parte constante de una relación: su nombre y su estructura en forma de lista de atributos con sus correspondientes dominios.

Instancia (o extensión), conjunto actual de tuplas que contiene una relación (todas las tuplas de una relación comparten los mismos atributos)

Cardinalidad: número de tuplas que contiene una relación, valor cambiante con relativa frecuencia (cada vez que se añade o elimina una tupla).

Grado: número de atributos de una tupla, valor constante ya que un cambio de grado por añadir o quitar atributos implica definir una nueva relación

- **En el ejemplo de la relación** asignatura

Esquema: **asignatura (codigo:int, siglas:char, nombre:char, curso:int)**

Instancia:

codigo	siglas	nombre	curso
95000001	ALGE	Algebra	1
95000002	CALC	Calculo	1
95000003	FIS1	Fisica General 1	1
95000004	IING	Introd. a la Ingenieria de Teleco	1
95000005	FTEL	Fund. de los Sistemas Telematicos	1

Cardinalidad = 5
Grado = 4

Claves de una relación

Clave candidata de una relación es un conjunto de atributos mínimo que identifica unívocamente cada tupla de la relación.

Clave primaria es la *clave candidata* que el usuario elige para identificar las tuplas de la relación
- si sólo hay una *clave candidata*, esta será la *clave primaria*
- si hay más *claves candidatas*, las no elegidas serán *claves alternativas*

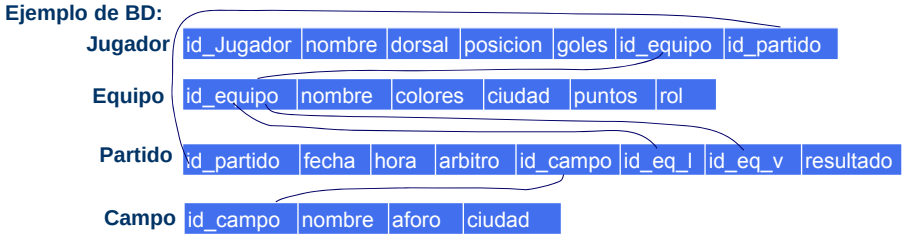
Clave ajena (o foránea) de una relación R2 es un atributo (o un conjunto de atributos) cuyos valores coinciden con los valores de la clave primaria de una relación R1

En el ejemplo, el atributo **grupo** de la relación **alumno** no es *clave candidata* porque no identifica unívocamente (hay valores repetidos) pero es *clave ajena* porque sus valores coinciden con los de la *clave primaria* de la relación **clase**

Vinculación entre esquemas (o relaciones)

Las claves ajenas establecen vínculos o interrelaciones entre esquemas de una BD

Ejemplo de BD:



Existe una interrelación entre dos esquemas si uno posee alguna clave ajena del otro
Dependiendo de los números de tuplas relacionados, las interrelaciones pueden ser:

1:1, a cada tupla de una relación le corresponde 1 y sólo 1 tupla de la otra
p.e: un partido sólo puede jugarse en un campo

1:N, a cada tupla de una relación le corresponden varias en otra
p.e.: un equipo juega muchos partidos

N :M, N tuplas de una relación se pueden corresponder con M tuplas de la otra
p.e.: muchos jugadores (N) juegan muchos partidos (M), o
2 equipos juegan dos partidos (ida y vuelta) interrelación 2:2

Condiciones de diseño

El modelo de datos relacional es un modelo basado en *relaciones* que deben cumplir determinadas condiciones mínimas de diseño:

- No deben existir tuplas duplicadas, es decir, todas las filas deben ser distintas
  eso garantiza que siempre existirá, al menos, una clave candidata formada por el conjunto de todos sus atributos
- Es irrelevante el orden de las tuplas dentro de una relación. También es irrelevante el orden en el que se definen los atributos de una relación.
- Cada atributo sólo puede tomar un único valor del dominio, es decir, no pueden contener listas de valores.

❖ Se puede asignar a un atributo el valor *NULL*, salvo si es una clave primaria, para indicar *ningún valor del dominio.* Con ello se permite insertar nuevas tuplas aún desconociendo el valor de algún atributo.

Lenguaje de consulta en bases de datos relacionales
SQL (Structured Query Language)

SQL es el lenguaje más extendido en los SGBD relacionales (MySQL, Oracle, ...)

- es un lenguaje algebraico (con algunas extensiones), normalizado por ANSI
- proporciona **operadores** que se aplican a las relaciones para formular consultas y, previamente, para crear/modificar las propias relaciones o sus atributos. Estos operadores se agrupan en tres sub-lenguajes:

DDL (Data Definition Language)

- operadores para crear/borrar/... relaciones

DML (Data Manipulation Language)

- operadores para realizar consultas y para insertar/borrar/modificar datos de tuplas

DCL (Data Control Language),

- operadores auxiliares para otras gestiones (prioridades, transacciones)

DDL	DML	DCL
CREATE	SELECT	GRANT
DROP	INSERT	REVOKE
ALTER	DELETE	COMMIT
TRUNCATE	UPDATE	ROLLBACK

Información completa sobre SQL y MySQL en: http://dev.mysql.com/doc/refman/5.0/es/
o en: http://mysql.conclase.net/curso/

Ejemplo al que aplicaremos los operadores SQL

Para explicar la aplicación de los operadores SQL, usaremos como ejemplo una base de datos de gestión de matrículas, definida por los esquemas:

alumno (dni:char, nom_apell:char, login: char, edad:integer, grupo:char)
asignatura (siglas:char, nombre:char, creditos:real, semestre:integer)
matriculado (dni:char, siglas:char, nota:real, nc:integer)

cuyas instancias actuales serán las reflejadas en las siguientes tablas:

alumno

dni	nom_apell	login	edad	grupo
52123123	Ana Sanz	asanz	20	G16
36781234	Ivan Ruiz	iruiz	19	G11
76543218	Juan Saez	jsaez	21	G17
12345678	Sol Diaz	sdiaz	18	G12

asignatura

siglas	nombre	creditos	semestre
ALGE	Algebra	6	1
FTEL	Fund de S. Telematicos	4.5	1
IINT	Intro. Ingenieria Teleco	3	1
PRG	Programacion	6	2
INEL	Intro. Electronica	4.5	2

matriculado

dni	siglas	nota	nc
52123123	ALGE	5.7	3
76543218	FTEL	6.6	1
12345678	FTEL	8.3	1
36781234	IINT	9.2	1
12345678	PRG	7.5	2
52123123	INEL	5.0	1

Crear/Borrar Relaciones con SQL

Operación: crear tabla

```
CREATE TABLE nombre_de_tabla (
   campo_1 tipo [(tamaño)] [NOT NULL] [DEFAULT def]
   [, campo_2 tipo[(tamaño)] [NOT NULL] [DEFAULT def], ...],
   PRIMARY KEY (campo_i [, campo_j , ...]) );
```

Ejemplo: crear las tablas alumno y matriculado de la transparencia anterior

```
CREATE TABLE alumno (
  dni        CHAR (9) NOT NULL,
  nom_apell  CHAR(50) NOT NULL,
  login      CHAR(10),
  edad       INT (2),
  grupo      CHAR(3) DEFAULT "G17",
  PRIMARY KEY (dni) );
```

```
CREATE TABLE matriculado (
   dni char(9),
   siglas char(4),
   nota real,
   nc int,
   PRIMARY KEY (dni, siglas));
```

Operación: borrar una tabla

```
DROP TABLE nombre_de_tabla_1 [, nombre_de_tabla_2];
```

Ejemplo: borrar la tabla alumno

```
DROP TABLE alumno;
```

Nota1: Es indiferente poner mayúsculas o minúsculas. Se usan aquí mayúsculas sólo para resaltar.

Nota2: Los tipos pueden variar dependiendo del SGBD que se use. En el enunciado del laboratorio se indican los tipos básicos de datos admitidos en el SGBD MySQL

Modificar una relacion con SQL

Operaciones	***Ejemplos***
Insertar filas (registros) `INSERT INTO nombre_de_tabla VALUES ([campo1],[campo2]);`	*insertar nuevo alumno* `INSERT INTO alumno VALUES ('52123123','Juan Saez', 'jsaez', 20,'G17');`
Reemplazar (o insertar) filas `REPLACE INTO nombre_de_tabla [(campo1, campo2, ...)] VALUES (,...),(,...);`	*sustituir el alumno antes insertado por otro* `REPLACE INTO alumno VALUES ('52123123', 'Ana Sanz', 'asanz', 20, 'G16');`
Modificar o actualizar filas `UPDATE nombre_de_tabla SET campo1 = nuevo_valor [,campo2 = nuevo_valor] WHERE condición;`	*cambiar alumnos del grupo 16 al 11* `UPDATE alumno SET grupo = "G11" WHERE grupo = "G16";`
Borrar filas `DELETE FROM nombre_de_tabla [WHERE condición];`	*borrar alumno de nombre "Eva Alba"* `DELETE FROM alumno WHERE nombre = "Eva Alba";`
Borrar todas las filas de una tabla `TRUNCATE nombre_de_tabla;`	*borrar todos los matriculados* `TRUNCATE matriculado;`

Nota: Es indiferente usar comillas dobles o simples para delimitar los valores de tipo texto

Select: operador para consultar datos en SQL

Operación:

```
SELECT selección
       FROM nombre_de_tabla_1 [, nombre_de_tabla_2]
            WHERE condición
            ORDER BY campo1 [, campo2];
```

Consultas simples (incondicionales). Ejemplos:

Mostrar todas las tuplas de la relación **alumno** (el carácter "*" indica todos los campos)

```
SELECT *  FROM alumno;
```

Mostrar todos los valores de los campos **nombre** y **creditos,** de la relación **asignatura**

```
SELECT nombre, creditos  FROM asignatura;
```

Mostrar el resultado de aplicar una función a los valores de un campo de una relación:

```
SELECT MIN(creditos) FROM asignatura;
```

Mostrar valores combinando nombres de campo y funciones:

```
SELECT MAX (nota), siglas FROM matriculado;
```

Mostrar valores de campos seleccionados, previamente ordenados por alguno de los campos

```
SELECT nombre, creditos FROM asignatura ORDER BY siglas;
```

Selección de campos con valores condicionados por expresiones lógicas. Ejemplos

```
SELECT siglas FROM asignatura WHERE creditos < 6 ;
SELECT dni FROM alumno WHERE grupo = "G11" OR grupo = "G12";
SELECT nom_apell, dni, grupo FROM alumno WHERE edad >= 18
       ORDER BY grupo, nom_apell;
```


Operador *select* para consultas complejas

En las bases de datos relacionales se pueden realizar consultas complejas combinando tuplas las tablas que están interrelacionadas mediante claves ajenas

Un caso ejemplo: *obtener una lista ordenada de los nombres y apellidos de alumnos matriculados en "Fund de S. Telematicos", de la base datos formada por las tablas de **alumno**, **matriculado** y **asignatura,** cuando los nombres de alumnos están en la tabla **alumno**, el nombre de la asignatura en **asignatura** y la relación entre estas dos tablas no es directa sino a través de las claves ajenas de la tabla **matriculado**:*

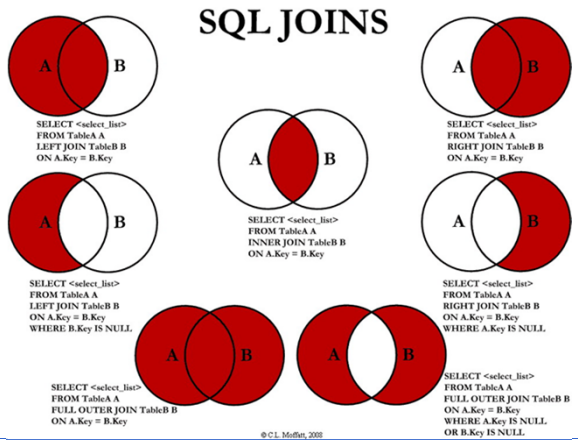
Una forma de *cruzar* los datos de varias tablas es añadir como condición la igualdad entre las claves ajenas y sus correspondientes claves primarias. En el ejemplo:

```
SELECT alumno.nom_apell FROM alumno, matriculado, asignatura
       WHERE asignatura.nombre = "Fund de S. Telematicos"
             AND matriculado.dni = alumno.dni
             AND matriculado.siglas = asignatura.siglas
       ORDER BY alumno.nom_apell;
```

JOIN en SQL

Para el tratamiento de datos cruzados de bases de datos relacionales formadas por muchas tablas interrelacionadas, SQL facilita una sentencia especial**, join**, que permite realizar la operación de composición relacional (del álgebra de Boole).

La figura representa todas las forma posibles de JOIN (que no vamos a usar en este curso)



Otros modelos de bases de datos

- Bases de datos orientadas a objetos
- Bases de datos distribuidas
- …

En 2009 se creó el movimiento NO-SQL (Not only SQL) que refleja un importante surgimiento de bases de datos *no relacionales*, *distribuidas*, *de código abierto*, y *horizontalmente escalables*, que podrían llegar a sustituir a las bases de datos relacionales clásicas (englobadas bajo el término de bases SQL) que no resultan suficientemente eficaces para almacenar la información de grandes sitios web como Google, Amazon... (http://nosql-databases.org/ , http://www.nosql.es/)

Ejercicio de clase (1)

artist	album	released	genre	copies
Michael Jackson	Thriller	1982	Pop, Rock, R&B	42.4
Eagles	Their Greatest Hits	1976	Country rock, folk rock	32.2
Whitney Houston	The Bodyguard	1992	R&B, soul, pop	27.4
Fleetwood Mac	Rumours	1977	Soft rock	26.8
AC/DC	Back in Black	1980	Hard rock, heavy metal	25.9
Pink Floyd	The Dark Side of the Moon	1973	Progressive rock	22.7
Bee Gees	Saturday Nigth Fever	1977	Disco	18.9

Esta tabla copiada de la wikipedia muestra los álbumes de discos más vendidos a lo largo de los tiempos. Escribir en el lenguaje SQL, sentencias para:

a) crear una tabla con los campos adecuados para incluir este tipo de información
b) insertar, en la tabla creada, los valores correspondientes al primer registro
c) consultar autores y títulos de los álbumes lanzados con posterioridad a los años 70, y de los cuales se han vendido más de 25 millones de copias.

Ejercicio de clase sobre consultas en SQL (2)

Suponiendo que nuestra base de datos actual es la reflejada en las siguientes tablas de alumno, asignatura y matriculado (vistas anteriormente), escribir las sentencias SQL que den respuesta a las siguientes cuestiones:

a) ¿Cómo se llama el alumno con DNI 12345678?
b) ¿Cuál es la nota media de los alumnos matriculados en FTEL?
c) ¿Algún estudiante ha agotado el número de convocatorias (nc) en FTEL?
d) ¿Cuántos alumnos están matriculados en Fund de S. Telematicos?

alumno

dni	nom_apell	login	edad	grupo
52123123	Ana Sanz	asanz	20	G16
36781234	Ivan Ruiz	iruiz	19	G11
76543218	Juan Saez	jsaez	21	G17
12345678	Sol Diaz	sdiaz	18	G12

asignatura

siglas	nombre	creditos	semestre
ALGE	Algebra	6	1
FTEL	Fund de S. Telematicos	4.5	1
IINT	Intro. Ingenieria Teleco	3	1
PRG	Programacion	6	2
INEL	Intro. Electronica	4.5	2

matriculado

dni	siglas	nota	nc
52123123	ALGE	5.7	3
76543218	FTEL	6.6	1
12345678	FTEL	8.3	1
36781234	IINT	9.2	1
12345678	PRG	7.5	2
52123123	INEL	5.0	1

Bases de datos

- **Bases de datos relacionales**
  - Contexto. Definiciones. Arquitectura
  - Modelo Relacional.
    - Conceptos básicos: relación, tupla, atributos, claves
  - Lenguaje SQL.
    - Operadores para crear/modificar relaciones y atributos con SQL
    - Operadores para realizar consultas con SQL
- **Diseño de una base de datos: del modelo ER al relacional**
  - Modelo Entidad-Relación. Conceptos. Diferencias con el modelo relacional
  - Diseño conceptual con el modelo ER
  - Proceso de transformación de un diagrama ER a un esquema relacional

Material de estudio y trabajo:

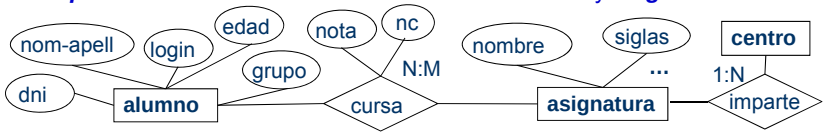
- Estas transparencias
- Ejercicios y prácticas de laboratorio propuestos

Modelo Entidad-relación

El **Modelo entidad-relación (E-R)** es el más utilizado para iniciar el diseño de una base de datos, representándola de forma gráfica mediante ***diagramas ER***, que permiten visualizarla en términos de:

- **Entidades**, representan los objetos o conceptos distinguibles en el mundo real, p.ej. alumno, asignatura
- **Atributos,** propiedades que describen el objeto representado (similares a los del modelo relacional), p.ej. dni, nom-apell, login, edad, grupo, nota, nc, ...
- **"Relaciones"** (*relationships*), un concepto muy distinto del de **Relación** (*relation*) en el modelo relacional, indicando aquí **asociaciones** o dependencias entre entidades

  p.ej. ***cursa*** es una relación entre las entidades ***alumno*** y ***asignatura***,
  imparte es una relación entre las entidades ***centro*** y ***asignatura***

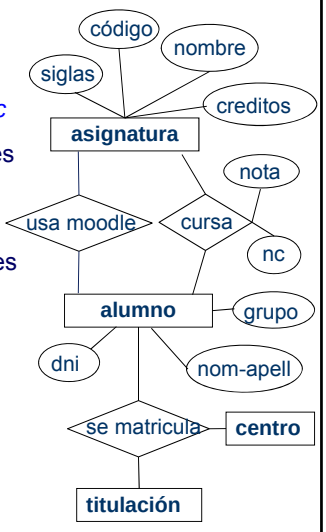


Modelo ER - Relaciones (relationships)

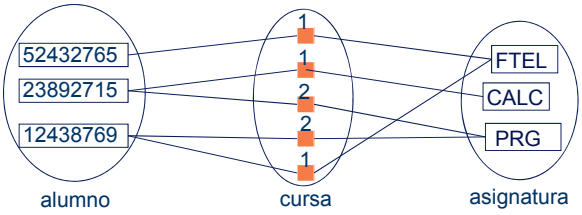
Relación: **asociación** entre dos o más entidades

- puede tener atributos propios
  - *ej. Ana-Sanz cursa la asignatura FTEL en primera convoc*
- una relación *n-aria* (*binaria*, *ternaria*…) relaciona *n* entidades
  - *"cursa" es una relación binaria (alumno –asignatura); "se_matricula" es ternaria (alumno – titulación – centro)*
- una misma entidad puede participar en diferentes relaciones y entre dos entidades puede haber más de una relación
  - *ej. un alumno se matricula en una titulación; cursa asignatura y usa el moodle de esa asignatura*

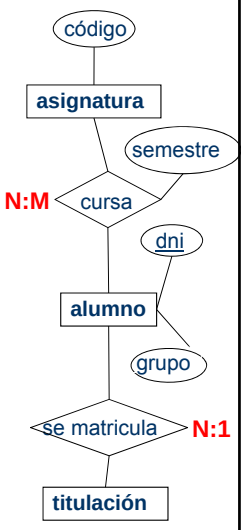
*!! El concepto **relación** no tiene el mismo significado en el modelo entidad-relación que en el modelo relacional !!*



Modelo ER – Restricciones en las relaciones

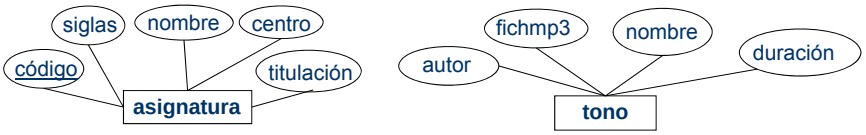


Ejemplo: ***un*** *alumno cursa* ***varias*** *(N) asignaturas y* ***una*** *asignatura es cursada por* ***varios*** *(M) alumnos, en cada semestre (1 ó 2):* **relación binaria N:M**



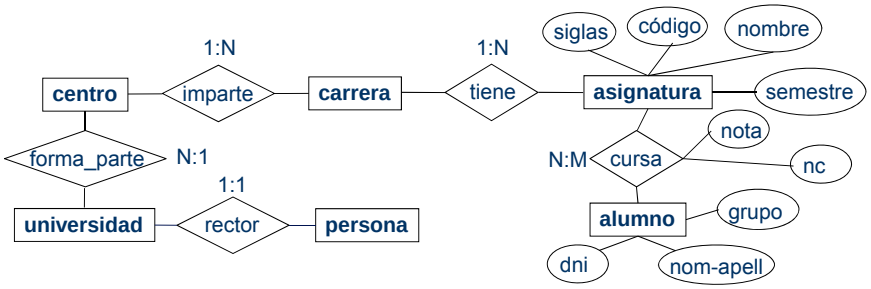
- Otras formas de relación binaria :
  - uno a uno (1:1) si en la relación se establece la restricción de que, cómo mucho, puede haber 1 instancia de cada entidad
  - uno a muchos (1:N o N:1) si la restricción establece que en una de las entidades de la relación puede haber, cómo máximo 1 instancia y en la otra varias (debe especificarse de algún modo el sentido de la relación, p.e. con flechas)
    - *ej.* ***muchos*** *alumnos se matriculan en* ***una*** *titulación*

Modelo ER. Dominios, claves



- los atributos representados reflejan el nivel de detalle con el que se quiere describir la entidad : *sólo se incluyen los que se consideran de interés*
- aunque no se especifique en el gráfico, se debe especificar para cada atributo su ***dominio*** de valores posibles
  - *ej. la duración del tono es un valor entre 3 y 10 segundos*
- para cada entidad debe especificarse una ***clave***, definida igual que en el modelo relacional como el conjunto mínimo de atributos que la identifican unívocamente
  - *ej. el código de una **asignatura** (ya que es un valor único para la universidad)*
  - *en el diagrama ER de **tono** no hay clave, por lo que habría que añadir alguna*

Ejemplo de diagrama ER con restricciones



Entidades: *universidad, centro, carrera, asignatura, alumno, persona*

Atributos: *siglas*, *nombre, código; semestre, dni, nom-apell, grupo, nota*, *nc*

Tipos de relación: *forma_parte, tiene, imparte, cursa, rector* (binarias)

Cardinalidad: número máximo de entidades que se asocian en una relación (*interpretación diferente a la del modelo relacional*)

1:1, indica que la cardinalidad máxima en ambas direcciones es 1

1:N, indica que la cardinalidad máxima en una dirección es 1 y en la otra varios (N)

N:M, indica que la cardinalidad máxima en ambas direcciones es varios

La notación de cardinalidad varía según el lenguaje usado para describir los diagramas

Diseño de una Base de Datos

Para realizar el diseño completo de una base de datos relacional, pasando por un modelo entidad-relación intermedio, es necesario seguir las siguientes fases:

A. Realizar el diseño conceptual de la base de datos, analizando los objetos del dominio, sus propiedades y sus asociaciones para determinar las **entidades**, los **atributos** de cada entidad, los dominios de los atributos y las **asociaciones** que existen entre las entidades.

   Mayor problema: en algunos casos, es difícil saber si un objeto debe modelarse como una entidad o como un atributo de otra entidad

B. Expresarlo en forma de **diagrama Entidad-Relación**

C. Seguir, paso a paso, las reglas del proceso de transformación desde el diagrama Entidad-Relación obtenido previamente hasta la definición de las **relaciones** que constituyen la base de datos: **esquemas** y/ **o tablas**.

Proceso de transformación (Fase C) del diagrama ER al esquema relacional

1. *Cada **entidad** se transforma en una tabla con sus propios atributos*
2. *Se agregan **claves** donde sea necesario*
   Cada tabla de entidad tiene que tener, al menos, una clave primaria
3. *Cada relación N:M se transforma en una tabla cuyos atributos son las claves de cada entidad*.

El número de tablas y las claves primarias dependen de la cardinalidad:

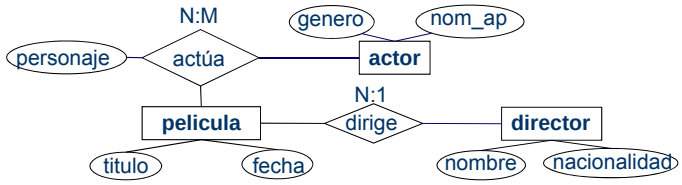
1:1, sólo se necesita una tabla con los atributos de las entidades que participan; la clave primaria será la de una de sus entidades

1:N, se necesita una tabla para cada entidad; la clave primaria será la de la entidad con cardinalidad N cuya tabla debe incluir también la clave de la otra entidad

N:M, se necesitan tres tablas: una por entidad y otra que contiene los atributos propios de la relación más las claves primarias de las entidades que participan en la relación (si es necesario, se añade una clave primaria de la relación)

Del diagrama ER al esquema relacional (1)

1. Cada entidad se transforma en una tabla con sus propios atributos
2. Se agregan claves donde sea necesario (si no se habían definido)
3. Cada relación N:M se transforma en una tabla cuyos atributos son las claves de cada entidad



pelicula

título	fecha
Ocho apellidos vascos	2014
El lobo de Wall Street	2013
Argo	2012
El gran Gatby	2013
Lo imposible	2012

actor

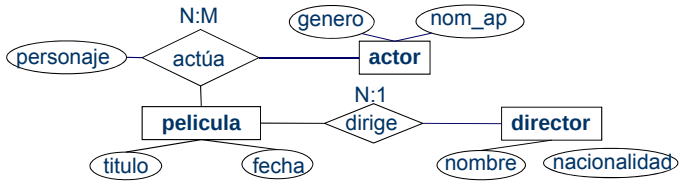
nom_ap	gen
Dani Rovira	H
Leonardo diCaprio	H
Naomi Watts	M
Clara Lago	M
Ben Affleck	H

director

nombre	nac
E. Martínez-Lázaro	ESP
M. Scorsese	USA
Baz Luhrman	AUS
B. Affleck	USA
J.A. Bayona	ESP

Del diagrama ER al esquema relacional(2)

1. Cada entidad se transforma en una tabla con sus propios atributos
2. Se agregan claves donde sea necesario (si no se habían definido)
3. Cada relación N:M se transforma en una tabla cuyos atributos son las claves de cada entidad



pelicula

id_p	título	fecha
1	Ocho apellidos vascos	2014
2	El lobo de Wall Street	2013
3	Argo	2012
4	El gran Gatby	2013
5	Lo imposible	2012

actor

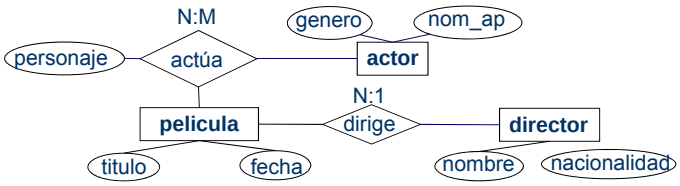
id_a	nom_ap	gen
1	Dani Rovira	H
2	Leonardo diCaprio	H
3	Naomi Watts	M
4	Clara Lago	M
5	Ben Affleck	H

director

id_d	nombre	nac
1	E. Martínez-Lázaro	ESP
2	M. Scorsese	USA
3	Baz Luhrman	AUS
4	B. Affleck	USA
5	J.A. Bayona	ESP

Del diagrama ER al esquema relacional(3)

1. Cada *entidad* se transforma en una tabla con sus propios atributos
2. Se agregan *claves* donde sea necesario (si no se habían definido)
3. Las relaciones N:1,1:M se reflejan en claves ajenas añadidas a la tabla. Cada relación N:M se transforma en una tabla cuyos atributos son las claves de cada entidad más los atributos propios de la relación.



actua

id_a	id_p	personaje
1	1	Rafa
2	4	Jay Gatsby
2	2	Jordan Belfort
3	5	María
4	1	Amaia
5	3	Tony Méndez

pelicula

id_p	título	fecha	id_d
1	Ocho apellidos vascos	2014	5
2	El lobo de Wall Street	2013	2
3	Argo	2012	4
4	El gran Gatsby	2013	3
5	Lo imposible	2012	1

actor

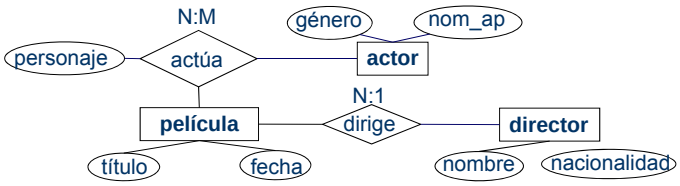
id_a	nom_ap	gen
1	Dani Rovira	H
2	Leonardo diCaprio	H
3	Naomi Watts	M
4	Clara Lago	M
5	Ben Affleck	H

director

id_d	nombre	nac
1	J.A. Bayona	ESP
2	M. Scorsese	USA
3	Baz Luhrman	AUS
4	B. Affleck	USA
5	E. Martínez-Lázaro	ESP

Del diagrama ER al esquema relacional. Conclusión

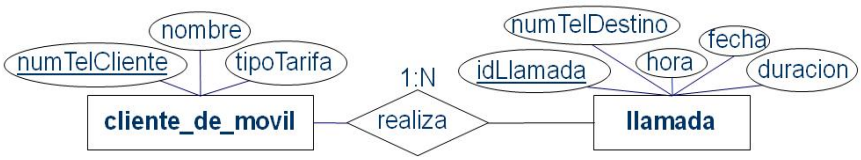
- Como resultado del diseño conceptual del caso (Fases A y B), se obtiene el diagrama Entidad-Relación:



- Tras el proceso de transformación y con la especificación de los dominios se obtienen los esquemas del modelo relacional:

pelicula (id_p:int, titulo:varchar, fecha:year, id_d:int)
actor (id_a: int nom_ap:varchar, gen: char)
director (id_d:int, nombre:char, nac:char)
actua (id_a:int, id_p: int, personaje:varchar)

Ejercicio de clase (3)



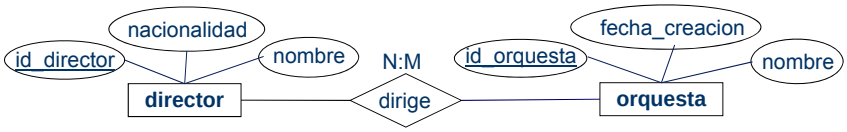
a) Escriba las sentencias SQL que permiten crear las tablas que se generarían al pasarlo al modelo relacional.

b) Escriba la sentencia SQL que permita obtener los nombres de los clientes que han efectuado llamadas al 555555555 en la fecha 2012-11-29.

Ejercicio de clase (4)

Supongamos que en una base de datos de música guardamos información de nombre y nacionalidad de cada director y de nombre y fecha de creación de cada orquesta, considerando que un director puede haber dirigido varias orquestas y una orquesta puede haber sido dirigida por varios directores.

El diagrama Entidad-Relación de esta base de datos sería de la forma:



1. Escriba las sentencias SQL que permiten crear las tablas del modelo relacional que se corresponde con este diagrama entidad-relación.
2. Escriba la sentencia SQL que permita obtener el nombre de los directores que han dirigido la orquesta: 'Academy of St Martin in the Fields'

Laboratorio 9. Bases de datos

Objetivos:

- Experimentar con la creación de tablas en bases de datos relacionales y la realización de consultas SQL.
- Familiarizarse con el uso del sistema de gestión de bases de datos relacionales MySQL, disponible en los ordenadores del laboratorio y en el servidor personal en la nube.

Realización:

- Las tareas de esta práctica se realizarán operando sobre el gestor de la base de datos del servidor personal en la nube, al que se accede desde una consola remota (abierta con ssh...)

Recursos:

- PC+Linux (o Windows) con configuración de red operativa y navegador Firefox.
- Portal Moodle de la asignatura
- Servicios en la nube de Amazon-AWS para la asignatura FTEL

Resultados:

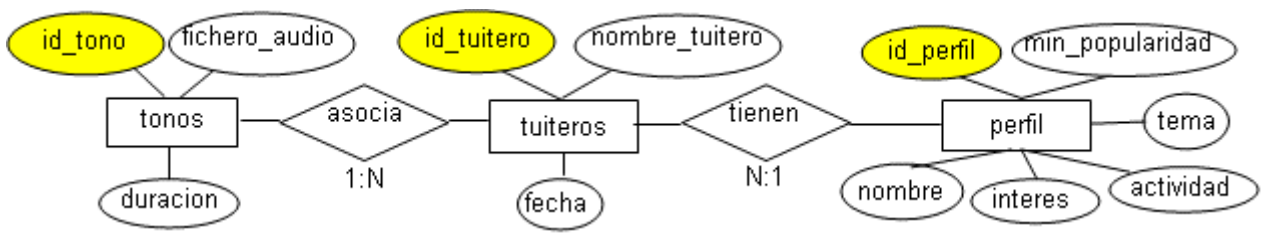
- Entregar el formulario de resultados.
- En el servidor personal deberá haber quedado la base de datos *mttpdb* que habrá creado en esta sesión de laboratorio.

Diseño de la base de datos para miPitatuiter

El servicio telemático que realizará como práctica final debe seleccionar unos *tuiteros* a seguir y asignar a cada uno de ellos un *tono* de aviso específico que suene cada vez que llegue uno de sus mensajes para identificarlo, de forma similar a cómo se hace en telefonía móvil.

Para organizar los datos de este servicio, debe definir una base de datos, *mpttdb*, con tres tablas. Una tabla *tonos* contendrá información de cada uno de los tonos que se oirán cada vez que llegue un mensaje del correspondiente tuitero. Una tabla *tuiteros* en la que se guardará información básica de los tuiteros seleccionados. Y una tabla adicional *perfiles* en la que se almacenarán parámetros que permiten caracterizar a los tuiteros.

En la figura se muestra el **diagrama Entidad-Relación** de esta base de datos. Se resaltan en el esquema las claves de cada tabla.



Tarea 1. Acceso al servidor personal en la nube y al SGBD MySQL

Puesto que el Sistema de Gestión de Bases de Datos (SGBD) MySQL con el que tiene que crear su base de datos está disponible en su servidor personal en la nube, lo primero que debe hacer es entrar en el portal Moodle de la asignatura y acceder a su servidor personal en la nube (enlace "**FTEL: Servidor personal en la nube Amazon AWS: Arranque, estado y acceso**").

Una vez arrancado el servidor personal en la nube y después de comprobar que está operativo, desde una consola remota (abierta con ssh...), acceda al SGBD escribiendo:

```
mysql -uroot –pftel
```

Aparecerá la invitación (promt) de mysql:

```
mysql>
```

a la espera de que se introduzcan sentencias SQL que deben terminar con el signo de puntuación ";" antes del fin de línea. En caso de no ponerlo, MySQL no dará ninguna respuesta, esperando ese ";" final. Es indiferente usar mayúsculas o minúsculas en las sentencias ya que SQL no las distingue.

Tarea 2. Creación de la base de datos mpttdb

Se indican a continuación las sentencias que hay que escribir para crear la base de datos *mpttdb*. Para crear una base de datos se usa una sentencia CREATE DATABASE y, para que sea compatible con el resto de componentes del proyecto final, es **muy importante** que el nombre que demos a la base de datos sea exactamente **`mpttdb`**. La sentencia que deberá escribir será, por tanto

```
mysql> create database mpttdb;
Query OK, 1 row affected (0.01 sec)
```

(*En color más suave se ve el resultado que dará MySQL*)

Para comprobar que la base de datos se ha creado, escribimos la sentencia SHOW DATABASES que nos da el resultado que se muestra después de la sentencia (en color suave):

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mpttdb             |
| mysql              |
| phpmyadmin         |
+--------------------+
4 rows in set (0.00 sec)
```

Como aparecen otras bases de datos que ya estaban creadas antes de definir la nuestra, es necesario indicar al SGBD MySQL qué base de datos se va a usar. Para ello, escribimos la sentencia USE:

```
mysql> use mpttdb;
Database changed
```

Tarea 3. Definición y creación de las tablas

Según el diagrama E_R, las tres tablas que forman la base de datos son ***tonos***, ***tuiteros*** y ***perfil*** con los campos que se indican a continuación:

*tonos (**id_tono**, fichero_audio, duracion)*
*tuiteros (**id_tuitero**, nombre_tuitero, fecha, id_tono, id-perfil)*
perfil (id_perfil, nombre, tema, interes, actividad, min_popularidad)

Para crearlas, debe usar la sentencia CREATE TABLE en su forma más simple, indicando nombre de la tabla, y nombre y tipo de cada atributo (Ver Apéndice final). Puesto que esta base de datos debe formar parte de la práctica final y debe haber una compatibilidad total con el resto de componentes, **ES MUY IMPORTANTE** que se usen nombres y tipos determinados para los siguientes campos:

```
id_tono  char(3)
fichero_audio  char(30)
id_tuitero char(15)
```

Ejecute las siguientes sentencias para crear las tablas de **tonos** y de **tuiteros**:

```
mysql> create table tonos (id_tono char(3), fichero_audio char(30), duracion
int, primary key (id_tono) );
Query OK, 0 rows affected (0.xx sec)

mysql> create table tuiteros (id_tuitero char(15), nombre_tuitero varchar(50),
fecha date, id_tono char(3), id_perfil int, primary key (id_tuitero) );
Query OK, 0 rows affected (0.xx sec)
```

Escriba una sentencia SQL para crear la tabla **perfil** con los siguientes campos:

id_perfil: un entero que identifique unívocamente cada perfil
nombre: nombre con el que quiera identificar el perfil
tema: nombre de su tema predominante: deporte, música, tecnología, noticias, ...
interes: indicador del tipo de usuario: empresa, experto en algo, famoso, político, ...
actividad: baja, media o alta, según el número medio de tuits que lanza cada día
min_popularidad: número de seguidores que como mínimo tienen los tuiteros de cada perfil

Para cada campo deberá seleccionar el tipo de dato más adecuado: CHAR, VARCHAR, INT, ... Al final del enunciado hay una tabla con información de tipos de datos básicos en MySQL.

3.1. Copie al formulario de resultados la sentencia con la que ha creado la tabla **perfil**.

Una vez creadas las tres tablas, puede comprobarlo con la sentencia `SHOW TABLES`:

```
mysql> show tables;
+------------------+
| Tables_in_mpttdb |
+------------------+
| tonos            |
| tuiteros         |
| perfil           |
+------------------+
3 rows in set (0.00 sec)
```

Puede ver la estructura de cada tabla, escribiendo sentencias DESCRIBE:

```
mysql> describe tonos;
+---------------+----------+------+-----+---------+-------+
| Field         | Type     | Null | Key | Default | Extra |
+---------------+----------+------+-----+---------+-------+
| id_tono       | char(3)  | NO   | PRI | NULL    |       |
| fichero_audio | char(30) | YES  |     | NULL    |       |
| duracion      | int      | YES  |     | NULL    |       |
+---------------+----------+------+-----+---------+-------+
3 rows in set (0.02 sec)
```

```
mysql> describe tuiteros;
+----------------+-------------+------+-----+---------+-------+
| Field          | Type        | Null | Key | Default | Extra |
+----------------+-------------+------+-----+---------+-------+
| id_tuitero     | char(15)    | NO   | PRI |         |       |
| nombre_tuitero | varchar(50) | YES  |     | NULL    |       |
| fecha          | date        | YES  |     | NULL    |       |
| id_tono        | char(3)     | YES  |     | NULL    |       |
| id_perfil      | int(11)     | YES  |     | NULL    |       |
+----------------+-------------+------+-----+---------+-------+
5 rows in set (0.02 sec)

mysql> describe perfil;
```

3.2. Copie al formulario de resultados lo que muestra MySQL con la sentencia anterior.

Tarea 4. Introducción de valores

Utilice sentencias INSERT INTO para introducir valores concretos en las tres tablas de forma que:

- en la tabla **tuiteros** esté incluida la información de todos los tuiteros (4 como mínimo) que ha seleccionado para realizar su seguimiento,
- en la tabla **tonos** esté incluida la información correspondiente a los tonos (3 como mínimo) que se asignan a los tuiteros, teniendo en cuenta que cada tuitero debe tener asignado sólo un tono pero el mismo tono puede estar asignado a varios tuiteros;
- en la tabla **perfil** esté incluida la información de los perfiles (3 como mínimo) que tienen los **tuiteros,** teniendo en cuenta que cada tuitero tiene un perfil pero puede haber varios tuiteros con el mismo perfil.

Para definir estas sentencias, es **MUY IMPORTANTE** cumplir las siguientes reglas, para que la base de datos sea compatible con el resto de componentes del servicio *mipitatuiter*:

a) Cada valor del campo **id_tono** en la tabla **tuiteros** debe coincidir con algún valor de **id_tono** en la tabla **tonos**. También debe haber correspondencia entre los valores del campo **id_perfil** de las tablas **tuiteros** y **perfil**.

b) El nombre del tuitero debe corresponder al nombre de algún *twitter* realmente existente y su identificador debe insertarse en el campo **id_tuitero** sin el carácter @.

c) El nombre de fichero a incluir en el campo "**fichero_audio**" de la tabla "**tonos**" debe:

- corresponder al nombre de un fichero mp3 que luego vaya a utilizar en la práctica final.
- incluir la extensión del fichero, no incluir ningún nombre de directorio ni ningún carácter "/" y, por supuesto, respetar las mayúsculas, minúsculas y caracteres de separación que tenga el nombre del fichero real (por ejemplo: **Kill_Bill-Whistle.mp3** o bien **001.mp3**).

d) Todos los campos deben tener algún valor distinto de NULL

- la duración de un tono no debe superar los 10 segundos.
- en la tabla **perfil,** puede *inventarse valores razonables* para los campos que desconozca, aunque estos valores deberán ser distintos para cada perfil incluido y las fechas deberán corresponder a años distintos (anteriores a 2015).

4.1. Copie al formulario de resultados todas las sentencias INSERT INTO que haya utilizado para definir los tonos, los tuiteros y sus perfiles.

Tarea 5. Consultas de comprobación

Utilice la sentencia SELECT para realizar consultas. Empiece por obtener una instancia completa de cada una de las tablas, para que compruebe si los datos introducidos cumplen todas las normas:

```
mysql> select * from tuiteros;
mysql> select * from tonos;
mysql> select * from perfil;
```

Si observa algún error puede utilizar las sentencias REPLACE, UPDATE, DELETE, TRUNCATE para hacer correcciones. Si se ha equivocado al definir el tamaño de algún campo puede actualizarlo usando la sentencia ALTER. En el Anexo final se explica el uso de estas sentencias.

5.1. Copie en el formulario el resultado de las tres sentencias **select *** anteriores.

5.2. Escriba la sentencia SELECT para realizar la consulta que se indique en la tarea correspondiente a su grupo y copie al formulario dicha sentencia.

5.3. Copie en el formulario el resultado que da MySQL al ejecutar la sentencia SELECT anterior.

Cierre y copias de seguridad de la base de datos creada

Una vez rellenado todo el formulario con las cuestiones planteadas en la práctica, debe cerrar la base de datos escribiendo EXIT o QUIT.

```
mysql> exit
```

Para evitar que por cualquier circunstancia pierda todo el trabajo que ha realizado para crear la base de datos en la nube, **ES MUY IMPORTANTE** que haga una copia de seguridad (respaldo) y, para ello, se debe ejecutar en el servidor personal en la nube, la orden:

```
ftel@...:~$ mysqldump -uroot -pftel --opt mpttdb > mpttdb-backup.sql
```

y, a continuación debe bajar a su cuenta local en el laboratorio este fichero de respaldo que ha creado (**mpttdb-backup.sql**).

Cuando necesite recuperar la base de datos, deberá subir el fichero de respaldo **mpttdb-backup.sql** que se creó con `mysqldump` a su servidor personal en la nube y allí ejecutar la orden:

```
ftel@...:~$ mysql -uroot -pftel  mpttdb < mpttdb-backup.sql
```

Apéndice. Sentencias SQL y tipos de datos utilizables en la práctica Lab9

Explicaciones sobre la información contenida en la tabla:

- aunque en las sentencias se puede usar indistintamente mayúsculas o minúsculas, se ponen en mayúsculas los términos que deben escribirse tal como aparecen en la tabla.
- `tab` debe sustituirse por el nombre concreto de la tabla; `cx` por el del campo o columna `x`
- las sentencias aparecen en orden alfabético

Sentencias	**Utilidad**
`CREATE DATABASE nom-db;`	Crea un nueva base de datos de nombre `nom-db`
`ALTER TABLE tab DROP COLUMN cx;`	Elimina en la tabla `tab` el campo `cx`
`ALTER TABLE tab` `CHANGE c-ant c-nuevo tipo-c;`	Cambia un campo `c-ant` de la tabla `tab` y le da otro nombre `c-nuevo` y otro tipo `tipo-c`.
`CREATE TABLE tab (c1 tipo1, c2 tipo2, …,` `PRIMARY KEY (cx, …) )`	Crea una tabla de nombre `tab` con campos `c1`, `c2`… y sus tipos, e indica los que forman la clave primaria
`DELETE FROM tab` `WHERE condición;`	Borra de la tabla `tab` las filas que cumplan la condición indicada en WHERE
`DESCRIBE tab;`	Muestra la estructura (nombre y tipo de los campos) de la tabla de nombre `tab`
`DROP TABLE tab1, tab2, …;`	Borra las tablas indicadas (`tab1`, …)
`INSERT INTO tab` `VALUES (val1, val2, val3,...)`	Inserta nuevas filas en la tabla `tab` poniendo en los respectivos campos los valores `val1`, `val2`, …
`REPLACE INTO tab` `VALUES (val1, val2, val3,...)`	Si ya existía esa fila la sustituye por la nueva, si no hace lo mismo que INSERT
`SELECT * FROM tab`	Muestra el contenido completo de la tabla `tab`
`SELECT c1, c2, ... FROM tab;`	Muestra los valores de `c1`, `c2`, … en la tabla `tab`
`SELECT tab1.cx, tab2.cy...` `FROM tab1, tab2, ...  WHERE condicion ;`	Muestra los valores indicados de los campos de las tablas que cumplan la `condición` especificada

`SELECT tab1.cx, ... FROM tab1, tab2, ...`
`WHERE condicion`
`ORDER BY nomc1, c2,...;`
	Selecciona los valores de campos de varias tablas, como en la sentencia anterior y los muestra ordenados según los campos especificados

`SHOW DATABASES;`	Muestra todas las bases de datos ya definidas
`SHOW TABLES;`	Muestra las tablas de la BD que se está usando
`TRUNCATE tab;`	Vacía totalmente la tabla `tab`
`UPDATE tab SET c1=val1, c2=val2, ...` `WHERE condición;`	Modifica valores de campos (`c1`, `c2`…) en las filas de la tabla `tab` que cumplan la `condicion`
`USE nom-db;`	Selecciona la base de datos de nombre `nom-db` como la que se va a usar a partir de ese momento

Tipos de datos	**Valor que toma**
`INT`	número decimal entero
`FLOAT`	número decimal en coma flotante
`CHAR(`*nc*`)`	cadena de un número fijo de caracteres especificado por *nc*
`VARCHAR (`*nc*`)`	cadena de número variable de caracteres, entre 0 y *nc*
`DATE`	fecha en formato "aaaa-mm-dd" (deben incluirse las comillas dobles o simples)
`TIME`	la hora en formato "hh:mm:ss"