68000
68010 / 68020
Arquitectura y programación en ensamblador

Stan Kelly-Bootle y Bob Fowler
GRUPO WAITE

ANAYA MULTIMEDIA

INFORMATICA PERSONAL-PROFESIONAL

Título de la obra original:
68000, 68010, 68020 PRIMER

Traducción: Francisco J. López Aligué

Diseño de colección: Narcís Fernández

Primera edición, agosto 1987
Primera reimpresión, septiembre 1989

Anaya Multimedia agradece a Apple Computer España la cesión de la diapositiva de la cubierta, que representa la placa del Apple Macintosh II.

Publicado por acuerdo con
Howard W. Sams & Co., Inc.

Josefa Valcárcel, 27. 28027 Madrid
Depósito legal: M. 24.617-1989
ISBN: 84-7614-136-X
Printed in Spain
Imprime: Anzos, S. A. - Fuenlabrada (Madrid)

Indice

Agradecimientos

En un principio, estuvimos tentados de romper con la tradición, proclamando que este libro se debe a nuestro trabajo exclusivamente, habiendo sido concebido, escrito y producido sin ninguna clase de ayuda externa. Sin embargo, cierta honestidad y caballerosidad, junto con presiones procedentes de ciertas áreas, nos convencieron de que debíamos presentar nuestro agradecimiento en la forma usual.

En primer lugar, debemos estar agradecidos al MOS Integrated Circuit Group, de Motorola, sin el cual (como ellos dicen) este libro estaría dedicado a una familia de circuitos mucho menos interesante. En particular, agradecemos a James J. Farrell III, manager de Comunicaciones Técnicas, y a Margaret Dickie, de Motorola Inc. (Austin, Texas), por su ayuda con permisos, figuras y diagramas.

Nuestra deuda con lo distintos autores de trabajos sobre el M68000 y, en general, los micros de 16/32 bits es enorme y queremos confesarla aquí en una relación breve y no exhaustiva: Stritter, Treddenick, Scanlon, Starnes, Kane, Hawkins, Leventhal, Alexandridis, Waite y Morgan.

También recibimos ayuda editorial de James Rounds (de Howard W. Sams & Co.), útiles advertencias del doctor Roger C. Gledhill, de International Micro Technologies Inc., y unos imprescindibles halagos de Mitch Waite y Jerry Wolpe, del Grupo WAITE.

Del lado de la producción, queremos agradecer a Lynella Cordell y su equipo en el Grupo WAITE, junto con Marla Rabinowitz y Walter Lynam,

la solución de los problemas tipográficos y estilísticos creados por nuestro golpes sobre las teclas.

Los ejemplos de programas han sido realizados con el equipo Alpha Microsistems Am-100/L mediante su ensamblador M68. Los autores desean agradecer a Alpha Microsistems y a la división de San Francisco del AMUG (Sociedad de Usuarios de Alpha Micro) su ayuda técnica. También hemos recibido ayuda a través de muchas discusiones informales con el doctor Michael Godfrey, del ICL (Londres), y con Bob Toxen, de Stratus Computer Inc. (Boston). A pesar de todo, aceptamos la total responsabilidad sobre cualquier pequeño defecto que pueda quedar y agradeceremos las correcciones que nos hagan.

Stan Kelly-Bootle proclama desde aquí su eterna devoción a su esposa Iwonka y a su hijastra Natasha Leof, por su amor y apoyo durante el proyecto.

Introducción

Este libro ha sido realizado pensando en el creciente número de programadores y usuarios, tanto noveles como experimentados, que desean poder utilizar el potente **set de instrucciones** de la familia Motorola 68000 de microprocesadores de 16/32 bits. El set de instrucciones constituye el lenguaje implementado dentro del circuito y es, en definitiva, al que deben de traducirse todos los programas escritos para el 68000, sea desde ADA, BASIC, C, o cualquiera otro.

Con los micros de 8 bits era difícil, pero posible, escribir y manejar directamente en código máquina sin necesidad de ensambladores. Con las palabras de 16 bits del código máquina, de las que puede haber hasta siete en una instrucción, el usar directamente el código máquina es para masoquistas reacios a adquirir un ensamblador. Por ello, el set de instrucciones del 68000 será visto desde la perspectiva de un ensamblador.

Los intentos llevados a cabo por los autores para analizar los ensambladores del 68000 pusieron de manifiesto un espeluznante vacío en la literatura al uso. No existía ninguna introducción suficientemente detallada y elemental, ni siquiera para las instrucciones y modos de direccionamiento comunes a todos los modelos de la serie 68000. Además, no había ninguna extensión inteligible a la máquina virtual 68010 ni a la estructura real de 32 bits del 68020. Este libro constituye nuestro intento de cubrir estas faltas.

Puede usarse el libro como un primer paso fácil para aquellos que deseen conseguir fluidez con los abundantes ensambladores simples y cruzados que existen (véase el apéndice E).

La documentación sobre el lenguaje ensamblador puede asustar, salvo que se conozca de antemano cómo funcionan los códigos, además de la poca claridad de muchos manuales sobre los modos de direccionamiento permitidos para cada instrucción. Y eso además de la maraña de directivas, macros, enlazamientos, saltos condicionales, librerías, llamadas a monitor, etc.

Las recompensas al trabajo son gratificantes. Aparte de las ventajas obvias, tales como velocidad de ejecución y reducción del tamaño de memoria ocupado (obsérvese cómo los anuncios de *software* claman por unos ensambladores rápidos y de poco espacio), hay pocas alegrías en los ordenadores como la que da el ver que un pequeño programa ensamblado por nosotros funciona correctamente. Suena a algo así como: "Vaya, lo hemos conseguido entre el 68000 y yo".

Este libro será también de utilidad para aquellos que deseen experimentar con alguna de las varias placas de entrenamiento existentes.

Adrede hemos evitado una descripción detallada de la microelectrónica y tecnología de circuitos integrados del MC68000. Nuestro sencillo análisis del 68000 lo enfoca como una "caja negra", pero es suficiente para, esperamos, poner de manifiesto la sutil interacción entre *hardware* y *software* desarrollada por Motorola.

Si usted desea profundizar más en sus conocimientos (y en el campo de los ordenadores no hay límite de profundidad), este libro le ayudará a elegir en la vasta literatura existente sobre la arquitectura del 68000, circuitos adicionales, Entrada/Salida, coprocesadores y diseño de sistemas.

El éxito engendra más éxito. Esta es una máxima originaria de la industria de microordenadores. Los precios de los semiconductores caen drásticamente a medida que la producción y fabricación aumentan, de forma que circuitos de tanto éxito como el M68000, especialmente la versión económica M68008, acaban por introducirse en el campo de los ordenadores personales y domésticos. El circuito básico, el MC68000, que puede encontrarse por cerca de 15.000 pesetas la unidad, forma ya parte de diversos elementos en estaciones gráficas y sistemas multiusuario en empresas, y no sólo como unidad central de proceso (CPU), sino también como elemento de circuito en dispositivos "inteligentes" de Entrada/Salida. Incluso uno más caro, el 68010, puede encontrarse en ciertos periféricos, tal como la impresora láser de Apple.

En cuanto al 68020, con un precio muy superior al de los anteriores, se acercará al del 68000 en los próximos años. El impacto que esto producirá hará tambalearse el mundo de los ordenadores personales y de empresa.

El ordenador Macintosh de Apple nos permite vislumbrar lo que puede conseguirse con la potencia del básico 68000. La facilidad de manejo derivada de las pantallas sobrepuestas, las imágenes de control (iconos), ventanas y menús abatibles controlados por el ratón impulsan demandas crecientes de memoria y capacidad de CPU.

La alta velocidad del MC68020 (16,67 MHz con memoria RAM rápida), menor consumo (1,5 vatios), adaptación a coprocesador incoporado para multiproceso y cálculo numérico de alta velocidad, así como la incrementada capacidad de direccionamiento de memoria (más de cuatro millones de

bytes), permitirán sistemas operativos mejores y más accesibles, tratamiento de gráficos en color más complejos (incluyendo animación) y lenguajes de alto nivel más naturales que faciliten el acceso a grandes bases de datos.

Una ventaja extra será el disponer de varios sistemas operativos en el mismo ordenador, obviando los actuales problemas de incompatibilidad (¿qué funciona y en cuál?) y el acabar de una vez por todas con las medievales discusiones, tales como: "UNIX frente AMOS" o "CP/M frente MS-DOS". Y este libro le preparará para esta revolución.

Todos los componentes de la familia del 68000 comparten el mismo set de instrucciones básico, pero ampliado en cada nuevo modelo, de forma que se conserva la compatibilidad **hacia arriba** de los códigos objeto, garantizando el trabajo de los programadores, tanto del simple aficionado como de la mayor empresa de *software*. Con la misma velocidad que el *hardware* cae, los costes de *software* aumentan. Por ello, el 68000 fue diseñado pensando en la necesaria facilidad de programación y comprobación, de forma que, por mucho que la tecnología de los circuitos integrados avance, los "viejos" programas sigan corriendo en las nuevas máquinas. Evidentemente, a medida que esta revolución avance, habrá que estudiar nuevas cosas, pero los lectores encontrarán gratificante saber que tan sólo unas pocas cosas de este libro tendrán que ser olvidadas.

Prerrequisitos

Uno de los primeros problemas que todo autor debe resolver es el relacionado con el nivel de los lectores a los que éste va destinado. Hemos tomado la decisión de hacer una exposición esquemática de las bases de los ordenadores, por lo que confiamos en el criterio del lector para saltar aquellas partes conocidas.

Nuestro método se ha guiado por la presencia de los programadores del año 86, que no han sufrido contacto previo con los micros de 8 bits, de forma que si usted conoce los lenguajes de programación del Intel 8080, del Zilog Z80 o del Motorola M6800, los actuales códigos le sonarán y podrá penetrar en las sutilidades que la riqueza del set de instrucciones del 68000 le ofrece. Pero para aquellos que se enfrentan de nuevas con los códigos, hemos tratado de exponer tanto el funcionamiento como la motivación para la existencia de cada uno de ellos, haciendo uso de numerosos ejemplos sencillos. Además, se ha cuidado del orden de aparición de los códigos, agrupando aquellos que comparten alguna característica en común.

Al final se ofrecen cuatro apéndices (A, B, C, D), en los que se presentan los códigos y sus modos de direccionamiento de diversas maneras, amén de una carta de referencia extraíble.

El libro

Tras los conceptos básicos y opcionales del capítulo 1 (hemos rechazado el gastado tópico de llamarlo capítulo 0), el capítulo 2 expone la perspectiva histórica y cita las características que distinguen los cinco modelos del 68000 actualmente en uso. El capítulo 3 analiza el circuito desde una perspectiva *software* (organización de la memoria y disposición de los registros). Las instrucciones y los modos de direccionamiento se describen de forma progresiva con ayuda de ejemplos, a partir de los más comunes y útiles en el capítulo 4, acelerando hasta los más avanzados en el capítulo 5. Las restantes instrucciones se estudian en el capítulo 6. El capítulo 7 está dedicado a analizar el concepto de "máquina virtual" de la familia 68010 y, finalmente, el capítulo 8 describe las numerosas mejoras que presentan los 32 bits reales del MC68020.

Equipos y sistemas M68000

Si, tal como esperamos, tras la lectura del libro, usted se siente animado a saber más del 68000, al final del libro, en el apéndice E, encontrará una relación de los productos *hardware* y *software* realizados con él. Esta lista, sin embargo, tiene una vigencia muy limitada y sólo depende del momento en que se realice su consulta, sobre todo en lo referente a precios y direcciones.

Como podrá comprobar, hay equipos que van desde los domésticos de menos de 75.000 pesetas, los personales más sofisticados, con un precio comprendido entre las 225.000 y las 450.000 pesetas (en cuya cumbre se halla el omnipresente Apple Macintosh), hasta los sofisticados sistemas UNIX (de 750.000 pesetas), el sistema de laboratorio 9000 de IBM, los equipos multiusuario de empresa, los Alpha Micro, Stride, Cromenco y un largo etcétera creciente día a día.

Con precios desde 30.000 hasta 30 millones de pesetas, todos usan la instrucción

MOVE.z Dm,Dn

Cuando llegue al capítulo 4, usted sabrá por qué.

1

Conceptos básicos de microprocesadores

"Dado que el sistema total será un ordenador de propósito general, deberá contener elementos específicos dedicados a realizar operaciones aritméticas, memoria de datos, control y comunicación con el operador humano."

(A. W. BURKES, H. H. GOLSTINE y J. VON NEUMANN, *Discusiones preliminares al diseño lógico de un instrumento electrónico de cómputo*, 1946.)

Este capítulo presenta un cierto número de ideas básicas de gran utilidad que se necesitarán para poder comprender mejor la familia del 68000. Una "Introducción" no puede, por definición, exigir demasiados conocimientos previos; por ello, queremos advertir desde ahora que estudiaremos ciertos conceptos fundamentales para los próximos capítulos, tales como bits, bytes, aritmética binaria y *buses* [1].

Microprocesadores

La primera impresión que se tiene al oír la palabra "microprocesador" es que se trata de algo pequeño y, efectivamente, es un elemento de cóm-

[1] Las palabras bits, bytes y *buses* se respetan en su forma inglesa original debido a su extendido uso en español.

puto físicamente pequeño construido sobre una oblea de silicio (el prefijo "micro" indica, en el campo de la ciencia, la millonésima parte, tal como en "microsegundo").

La MPU (unidad microprocesadora)[2] es, únicamente, un elemento más dentro de todo el **sistema** de ordenador. Las figuras 1.1 y 1.2 muestran unos sistemas típicos. La "pastilla"[3] de la MPU, cuando se monta en una placa de circuito junto con otras de soporte, pasa a denominarse MPS (sistema microprocesador) o CPU (unidad central de proceso)[4]. A menudo suele llamarse el "cerebro" del sistema a la MPU, pues es donde reside la inteligencia programable que coordina todos los otros elementos conectados a él. La MPU puede realizar operaciones lógicas y aritméticas, tomar decisiones y ejercer control de muchas formas distintas. Sin embargo, como

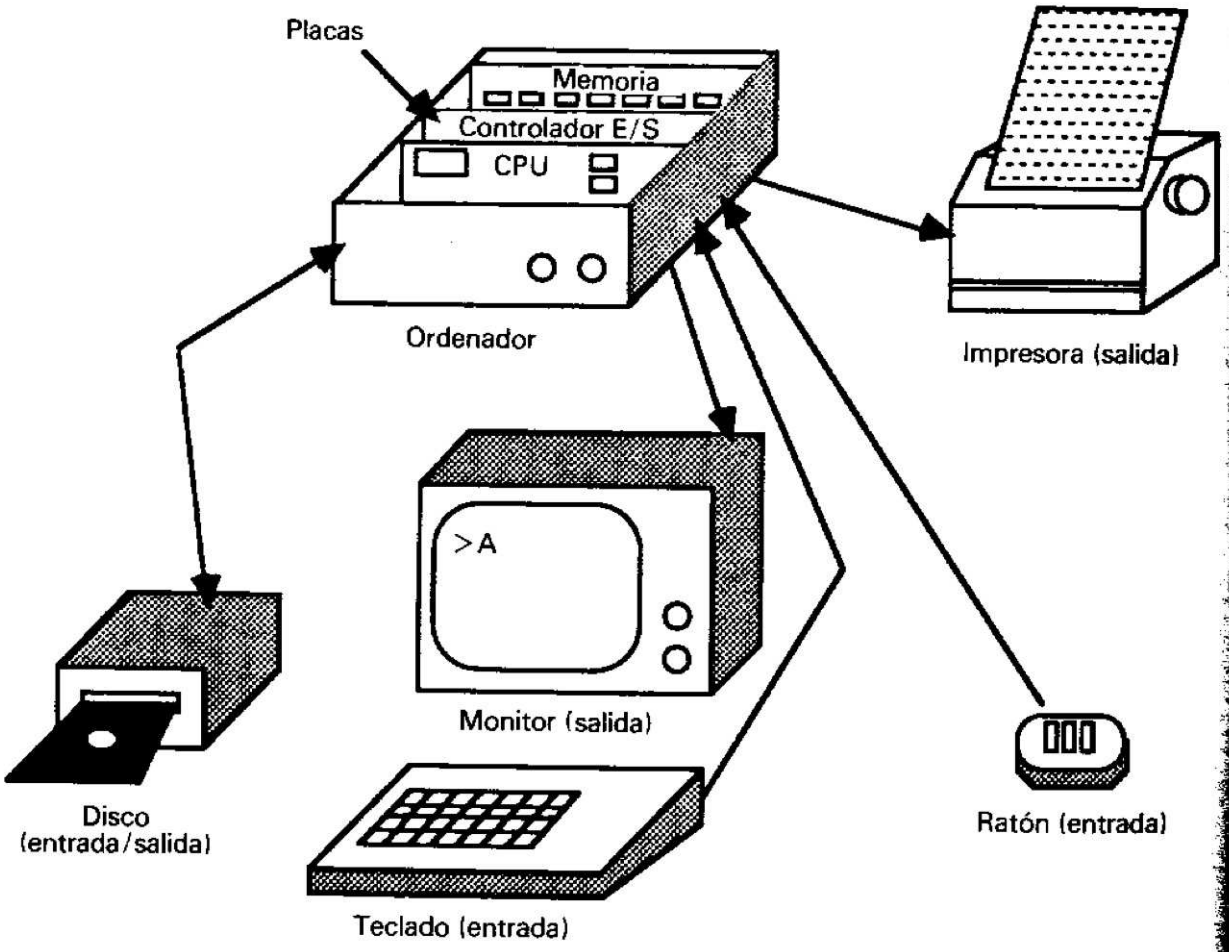


Figura 1.1
Configuración típica de un pequeño ordenador

[2] También es norma en español el empleo de las siglas MPU, e incluso las de CPU *(Central Process Unit),* tomadas directamente del inglés.

[3] En español es usual emplear la palabra pastilla en lugar de la más correcta de "circuito integrado", razón por la que aquí la conservaremos.

[4] Asimismo, no suelen distinguirse tan claramente, en nuestro idioma, las acepciones dadas a MPU y a CPU, siendo raro encontrar referencias a MPS, por lo que aquí emplearemos MPU y CPU siguiendo el original inglés, concordante por reducción con el español

veremos, todo esto sucede de forma predeterminada, impuesta por los **programas** (secuencias ordenadas de instrucciones muy precisas).

Las MPU no son, en absoluto, las piezas más caras de un sistema, hasta el punto de que pueden encontrarse varias juntas en cada uno de ellos (lo que recibe el nombre de sistema multiprocesador). En estos casos, muy a menudo una de ellas es la MPU **maestra** y las restantes las **esclavas** a las que se asignan determinadas tareas. A veces, cada MPU funciona de forma independiente de las demás realizando su tarea cuando es requerida para ello.

¿Qué se procesa?

La pregunta es: ¿Qué y cómo se procesa? Los cocineros "procesan" alimentos, las plantas depuradoras "procesan" residuos, y los microprocesa-

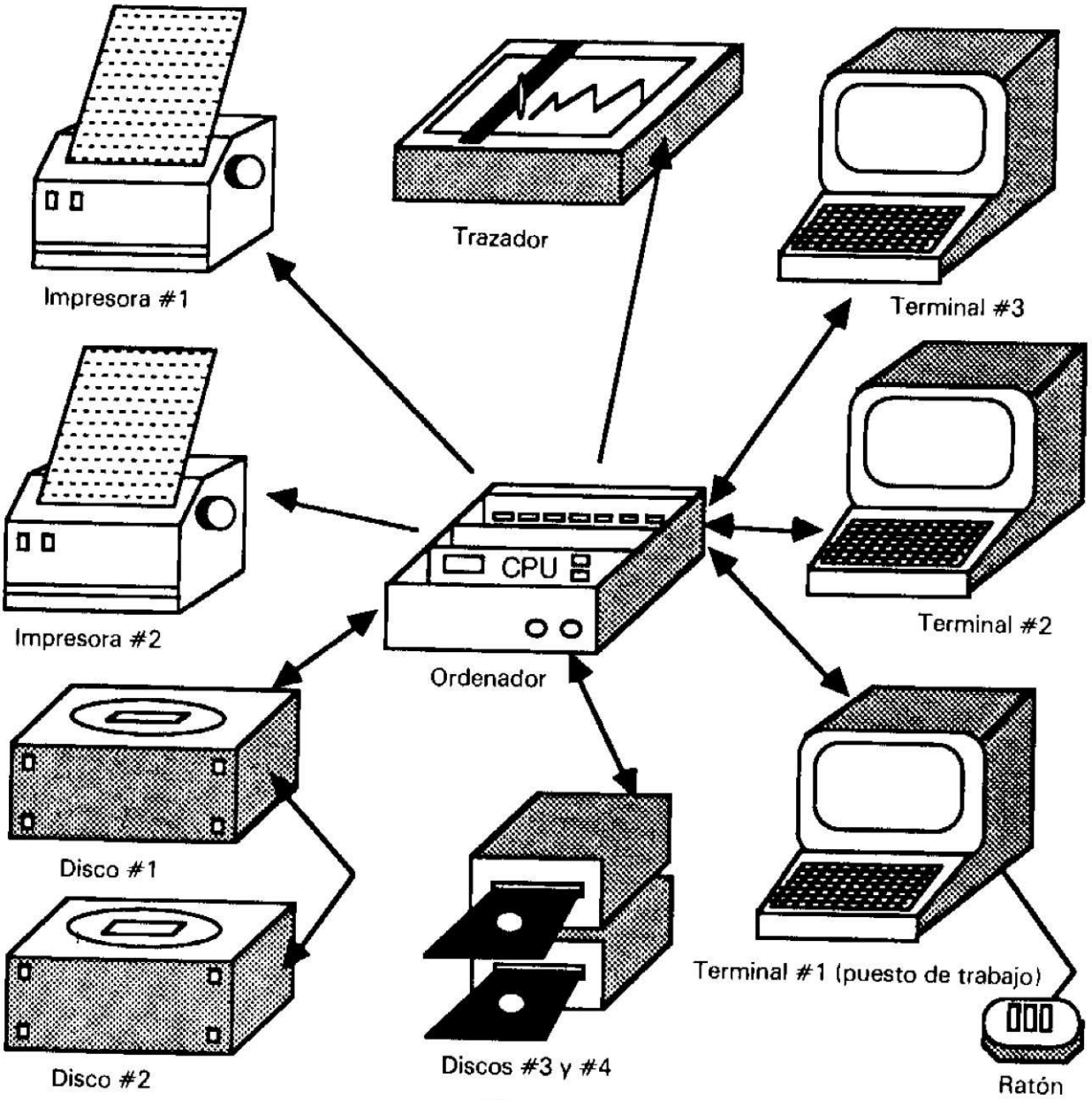


Figura 1.2
Sistema multiusuario típico

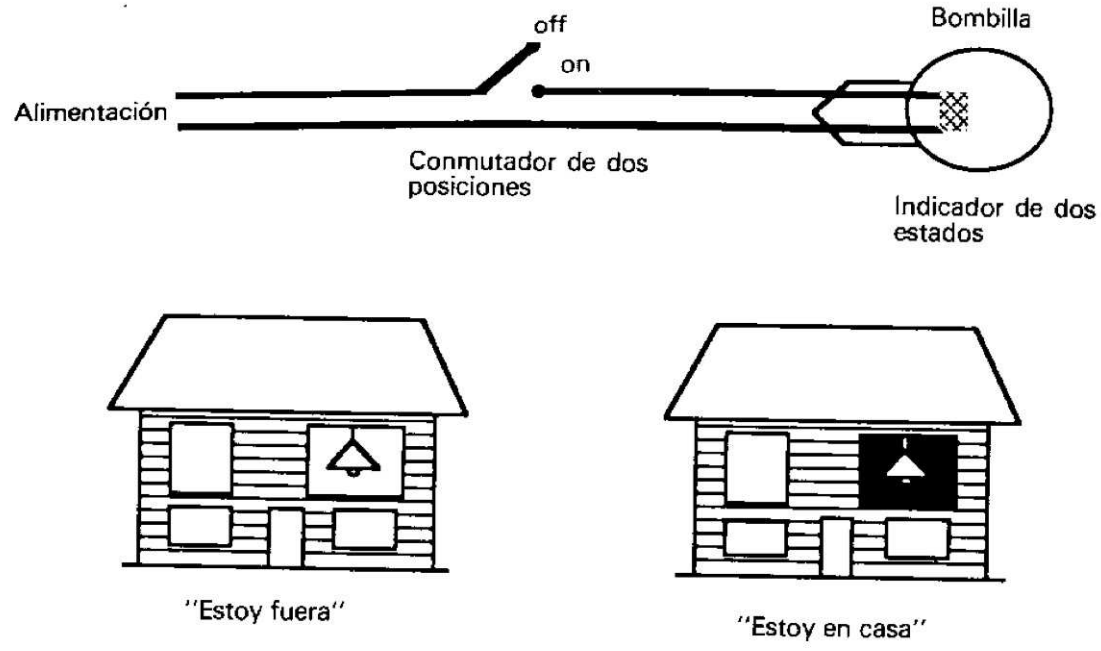


Figura 1.3
El interruptor de la luz como un bit

dores "procesan" DATOS y (con la ayuda de varios artilugios y programa bien detallados) producen **información.**

Si todo esto parece abstracto e intangible, es porque en cierto sentido as es. El ordenador es una máquina de propósito general que manipula cie gamente los símbolos, y es usted, el usuario, quien da sentido y finalida a este proceso.

En las descripciones siguientes emplearemos la palabra "datos" en s adscripción más genérica posible.

Información y datos

Información, por regla general, es todo aquello que reduce la incert dumbre. Claude E. Shannon, de los Laboratorios de la Bell, refinó es idea, en los años cuarenta, dando lugar a una nueva rama de las matem ticas, llamada **teoría de la información.** Mostró que, en muchos casos, cantidad de información que contiene un mensaje puede expresarse y m dirse por **dígitos binarios** o **bits.** Así como la leche se manipula en recipie tes adecuados, la información se transvasa en bits.

Bits en acción

El milagro del cálculo automático reside, en definitiva, en este concep uno de los más simples de todas las matemáticas. El bit es la unidad bás de información, capaz de destruir la incertidumbre de un "Sí" o un "No Por tanto, un bit puede tener solamente uno de entre dos valores, a

cuales se denomina, usualmente, por los símbolos "0" y "1", pero que puede ser interpretado de muy diversas formas: conectado-desconectado, negro-blanco, cierto-falso, sí-no, pero incapaz de indicar términos intermedios, tales como grises, o quizá, o tal vez.

Resulta evidente que muchos elementos usuales, tales como los conmutadores domésticos de la luz, poseen esta misma característica Sí/No, y pueden, por tanto, ser empleados para "almacenar" un bit de información.

A tales dispositivos que tienen un limitado número de posibles situaciones (estados) se les llama dispositivos **discretos**, en contraposición con aquellos que, como los mandos del volumen de un aparato de radio, pueden variar de forma continua sobre un número "infinito" de tales estados.

En la figura 1.3, el estado del conmutador viene reflejado por el de la lámpara: encendida o apagada. La información que se ofrece a través de la ventana puede ser: o bien "Sí, estoy en casa", o bien "No, he salido". Sin embargo, ¿cuál es cada una? La lámpara encendida podría significar, perfectamente, su ausencia. Ello significa que la capacidad de enviar mensajes depende de la existencia de un código de significados establecido previamente entre usted y el posible receptor. Para este caso, sólo hay dos posibilidades:

Código A	*Código B*
Encendida = En casa	Encendida = Fuera de casa
Apagada = Fuera de casa	Apagada = En casa

El punto importante radica en el hecho de que el bit carece de significado por sí mismo, precisando de un compromiso previo entre el enviador (codificador) y el receptor (decodificador).

Shannon definía el bit como la cantidad de información precisa para destruir la ambigüedad de algo con dos únicas posibilidades. Dos eventos igualmente posibles comparten una misma probabilidad de 1/2 (una posibilidad del 50 por 100 cada uno), de forma que un bit puede resolver completamente esta incertidumbre, pero no más. Para poder manejar más información, se precisan más bits y, consecuentemente, disponer de mecanismos para realizar un eficaz almacenamiento, acceso, cambio, envío y decodificación de los mismos.

Cada bit que se añada duplica la cantidad de información. Por ejemplo, con dos lámparas en la ventana, puede generarse un código de cuatro mensajes diferentes:

Lámpara 1	*Lámpara 2*	*Mensaje*
Apagada	Apagada	No hace falta leche
Apagada	Encendida	Hace falta un litro de leche
Encendida	Apagada	Hacen falta dos litros
Encendida	Encendida	Hacen falta tres litros

Por supuesto, usted debe estar seguro de que el destinatario (el lechero en este ejemplo) conoce el código y, sobre todo, qué lámpara es cada una de las citadas. Si, por el contrario, las lámparas no pueden ser marcadas de forma que resulten distinguibles, puede comprobarse que sólo pueden generarse tres mensajes. Así, para obtener el máximo rendimiento de nuestros dos bits, deberemos ordenarlos previamente. Con esta premisa es fácil establecer una relación directa entre el número de bits y el de posibles mensajes:

Con 1 bit pueden codificarse 2	(2^1) mensajes
Con 2 bits pueden codificarse $2 \times 2 = 4$	(2^2) mensajes
Con 3 bits pueden codificarse $2 \times 2 \times 2 = 8$	(2^3) mensajes
Con N bits pueden codificarse $2 \times 2 \ldots \times 2$	(2^N) mensajes

Por esta razón, las potencias de 2 juegan un papel fundamental en la Teoría de la Información y en las Ciencias del cálculo automático.

También es importante el caso en que N = 10, puesto que $2^{10} = 1.024$, que se suele conocer como "kilo", e indicar por una K. Así, al leer 32K de memoria, estamos leyendo $32 \times 1.024 = 32.768$, en lugar de 32.000 justos.

Los mensajes codificados pueden significar cualquier cosa, instrucciones, símbolos, números, nombres, o quizá incluso nada en absoluto (un mensaje totalmente correcto es el de "ignorar este mensaje"). Incluso algunas combinaciones de bits pueden ser empleadas para denotar errores: Por regla general, hay siempre más combinaciones de bits (los llamados patrones) que mensajes a decodificar. Esta redundancia puede utilizarse para detectar errores de transmisión o de almacenamiento.

Aritmética binaria

Existe una forma natural de relacionar los bits con los números decimales. A la forma anterior se la denomina **aritmética binaria,** porque, a través de los símbolos 0 y 1, emplea las potencias de 2, frente a la aritmética convencional, que emplea las potencias de 10 mediante los símbolos 0 al 9. Prácticamente, todos los cálculos realizados interiormente por un ordenador lo son en aritmética binaria, incluso si el resultado debe ofrecerse en términos decimales. Siguiendo con el ejemplo del código de las dos lámparas, podemos establecer la siguiente correspondencia:

Lámpara encendida = 1	Lámpara apagada = 0
Lámpara 2 = 2	Lámpara 1 = 1

que nos lleva a los resultados:

Lámpara 2	*Lámpara 1*	*Litros*
0	0	0
0	1	1
1	0	2
1	1	3

de acuerdo con la ley de decodificación:

$$\text{Litros} = (\text{Lámpara } 2 \times 2) + (\text{Lámpara } 1 \times 1)$$

Con tres lámparas podemos alcanzar hasta siete litros, de la forma:

Lámpara 3	*Lámpara 2*	*Lámpara 1*	*Litros*
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

A la lámpara 3 se le ha asignado el valor $4 = (2 \times 2)$, de forma que la decodificación es ahora:

$$\text{Litros} = (\text{Lámpara } 3 \times 4) + (\text{Lámpara } 2 \times 2) + (\text{Lámpara } 1 \times 1)$$

En esta descripción pueden reconocerse las similitudes con el método decimal usual. Entendiendo las lámparas como posiciones de columnas, cada una de ellas representa una potencia de 2. Así, al escribir, en la notación decimal normal, el número 2379, se está empleando una notación abreviada cuyo significado correcto es:

2 millares 3 centenas 7 decenas 9 unidades
$2 \times (10 \times 10 \times 10)$ más $3 \times (10 \times 10)$ más 7×10 más $9 \times 1 = 2379$

donde cada columna representa una potencia de 10. A pesar de que la primera columna no parezca una tal potencia de 10, en realidad sí lo es, puesto que se trata de 10^0, que es igual a 1.

De la misma forma, el número binario 1101 se evalúa como:

1 ocho 1 cuatro 0 dos 1 unidad
$1 \times (2 \times 2 \times 2)$ más $1 \times (2 \times 2)$ más 0×2 más $1 \times 1 = 13$

Podemos, por tanto, establecer una correspondencia directa entre las configuraciones de bits (patrones) y los números binarios y decimales. Se trata, únicamente, de uno de los muchos esquemas posibles de codificación,

y debe ser admitido previamente como cualquier otro si se desea que sea capaz de "transportar" información.

Para enfatizar el papel que juegan los bits, volvamos a nuestro ejemplo de las tres lámparas. Inicialmente, el número de litros necesarios es desconocido, salvo en el hecho de que estará entre 0 y 7 (ocho valores posibles). La lámpara 1 reduce la incertidumbre de la forma:

Lámpara 1 encendida = el núm. de litros es 1, 3, 5 ó 7
Lámpara 1 apagada = el núm. de litros es 0, 2, 4 ó 6

con lo cual la indeterminación ha sido reducida a cuatro posibilidades, es decir, ha sido reducida a la mitad.

También las lámparas 2 y 3 dividen la incertidumbre por la mitad:

Lámpara 2 encendida = el núm. de litros es 2, 3, 6 ó 7
Lámpara 2 apagada = el núm. de litros es 0, 1, 4 ó 5
Lámpara 3 encendida = el núm. de litros es 4, 5, 6 ó 7
Lámpara 3 apagada = el núm. de litros es 0, 1, 2 ó 3

Al considerar las tres lámparas en conjunto, desaparece la indeterminación, puesto que tan sólo es posible un mensaje concordante con la situación planteada (patrón). Queda determinado uno de entre los ocho posibles.

Resumen de bits y mensajes

Con un conjunto ordenado de N bits, pueden codificarse hasta 2^N mensajes diferentes. Un sistema muy frecuente de codificación es el que relaciona estos 2^N números binarios con los decimales que van desde el 0 hasta el 2^n-1. Dado que es muy fácil almacenar electrónicamente los números binarios, los ordenadores realizan todas sus operaciones aritméticas y lógicas en binario.

Agrupaciones especiales de bits

Muy a menudo se encuentran los bits agrupados en bloques de 4, 8, 16 ó 32. A tales grupos se les da un nombre característico [5]:

- Un *nibble* = 4 bits, que pueden almacenar 16 mensajes.
- Un byte = 8 bits, que pueden almacenar 256 mensajes.

[5] Nuevamente hemos tratado de respetar la nomenclatura al uso en nuestro idioma. Sin embargo, las denominaciones aquí dadas corresponden a los sistemas Motorola y afines, no siendo totalmente válidas en otros casos. Es notorio el hecho de que, a menudo, se emplea la expresión de "palabra de X bits", lo que no debe, sin embargo, hacernos creer que el aquí presentado es de poca validez, pues se corresponde con el estándar propuesto por el IEEE (Institute of Electrical and Electronic Engineers) y es el más frecuente.

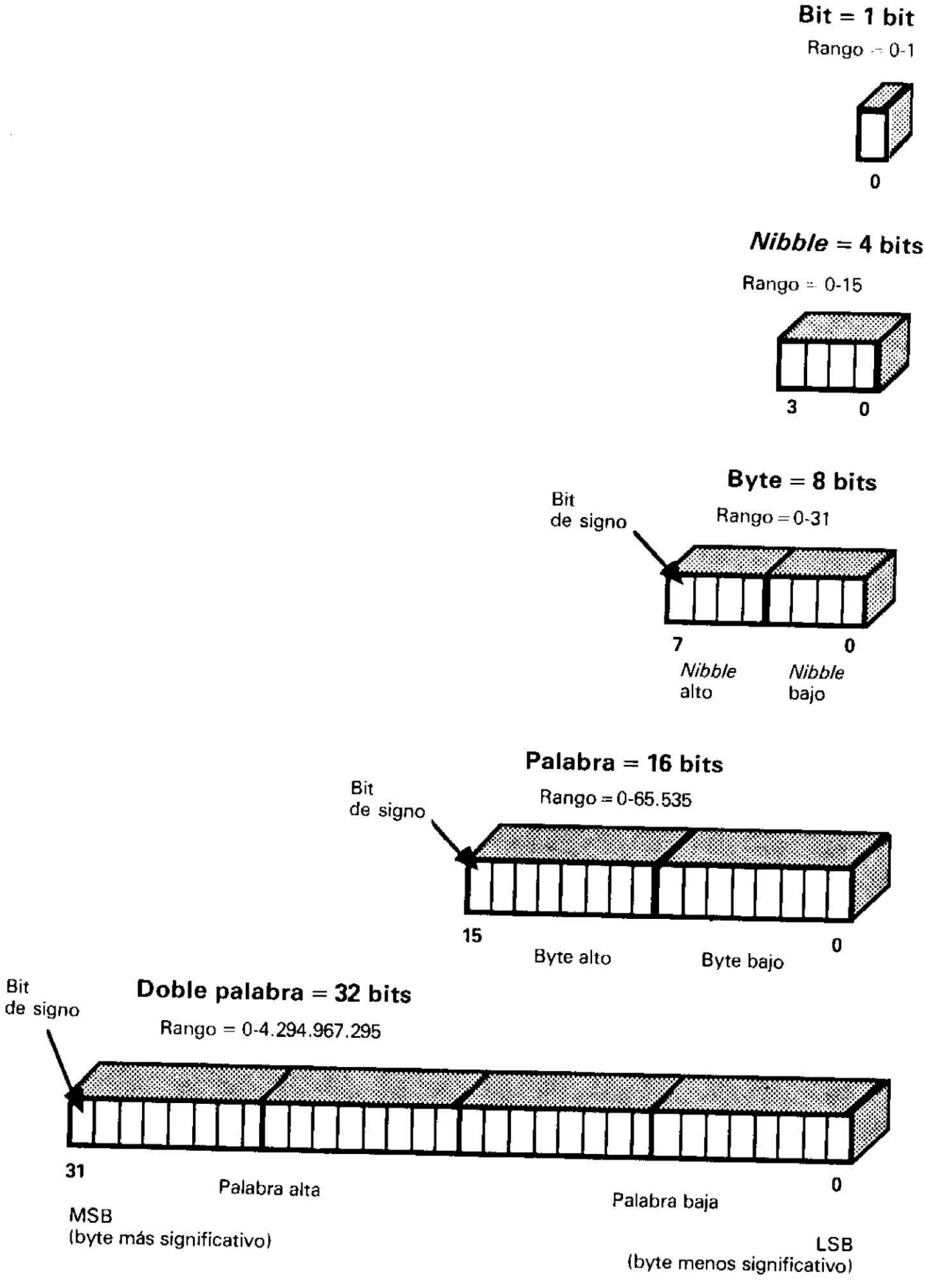


Figura 1.4
Grupos de bits: nibble, *byte, palabra, doble palabra*

- Una palabra = 16 bits, que pueden almacenar 65.536 mensajes.
- Una doble palabra = 32 bits, que pueden almacenar 4.294.967.296 mensajes.

Estos nombres y sus "rangos" asociados surgirán repetidamente, porque, debido a excelentes razones que más adelante veremos, el 68000 ha sido diseñado para trabajar con estas agrupaciones de bits.

En la figura 1.4 puede verse la numeración ordinal de los bits, empezando por el 0, llamado LSB (del inglés *Least Significant Bit,* bit menos significativo), situado a la derecha, hasta el extremo de la izquierda, denominado MSB (del inglés *Most Significant Bit,* bit más significativo). Dado su uso en los números negativos y positivos, el bit MSB también suele llamarse **bit de signo.**

BCD: Números decimales codificados en binario

Una de las más importantes aplicaciones de los *nibbles* (de 4 bits) consiste en la codificación de los dígitos decimales comprendidos entre el 0 y el 9. Con tres bits, como hemos visto, se puede codificar desde 0 hasta 7, luego se precisa un mínimo de 4 bits para alcanzar los 10. El código resultante, conocido por BCD (del inglés *Binary Coded Decimal*), tiene seis combinaciones carentes de asignación. En la siguiente tabla se muestran estas seis, más las diez restantes posibles:

BCD	*Decimal*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	No usado
1011	No usado
1100	No usado
1101	No usado
1111	No usado
(0001)(0000)	10
(0001)(0001)	11

Los dos últimos casos de la tabla muestran la forma de escribir un número que supere al 9. En este caso, se trata del 10 y del 11. Como se ve, este es-

quema es ineficiente desde el punto de vista del "consumo" de bits. Comparémoslo, por ejemplo, con la forma binaria convencional para el caso del decimal 2379:

2379 decimal = 100101001011 – 12 bits en código binario
2379 decimal = (0010)(0011)(0111)(1001) – 16 bits en BCD

Sin embargo, su utilidad reside en cálculos financieros, donde la exactitud exige su uso, no permitiéndose conversiones decimal-a-binario.

Código ASCII para caracteres

Una de las mayores razones para la existencia de los bytes es que las 256 combinaciones que pueden codificar resultan ser un número excelente para emplearlos en conjuntos de caracteres, cuales son los de un teclado de máquina de escribir. Las mayúsculas y las minúsculas, amén de los símbolos gramaticales y los controles (vuelta de carro, espacio en blanco, etc.), ocupan tan sólo 128 combinaciones, codificables con 7 bits, pero, dado que los 8 bits ofrecen 256 posibilidades, se puede emplear el resto en señales gráficas. La norma de asignación de símbolos a cada configuración de los 8 bits se llama ASCII (del inglés *American Standard Code for Information Interchange*). Esta norma es universal y posee ciertas modificaciones particulares para los diferentes idiomas (como la "ñ" del español).

El rango

Un byte abarca números comprendidos entre 0 y 255, luego para la mayoría de las operaciones matemáticas harán falta más bits. Esta situación se agrava cuando se manejan números negativos, de forma que debe emplearse un bit para el signo (que será 0 si el número es positivo y 1 si es negativo).

Bajo el convenio de la notación conocida por "complemento a 2" (véase la tabla 1.1), un *nibble* puede codificar números en el rango desde –8 hasta + 7 (lo que sigue siendo un total de 16 números) y un byte un número comprendido entre –127 y + 127 (lo que supone un total de 256).

Los bits de un byte se numeran de 0 a 7, empezando por la derecha hasta la izquierda (recuérdese que el primer bit por la derecha es el 0). En el caso del complemento a 2 el bit 7 (el más a la izquierda) es el **bit de signo** [6].

En el caso de la palabra de 16 bits, podemos alcanzar el rango comprendido entre 0 y 65536 sin signo (es decir, números positivos), pero si empleamos el bit de signo, podemos utilizar el complemento a 2 para movernos en el rango entre –32768 y + 32767.

---

[6] El complemento a 1 se obtiene cambiando simplemente los "1" por "0" y viceversa. El complemento a 2 se obtiene haciendo primero el complemento a 1 y sumándole después 1.

TABLA 1.1

Interpretación decimal de 4 bits en binario

Binario	*Complemento a 1*	*Complemento a 2*	*Sin signo*
0111	7	7	7
0110	6	6	6
0101	5	5	5
0100	4	4	4
0011	3	3	3
0010	2	2	2
0001	1	1	1
0000	0	0	0
1111	−0	−1	15
1110	−1	−2	14
1101	−2	−3	13
1100	−3	−4	12
1011	−4	−5	11
1010	−5	−6	10
1001	−6	−7	9
1000	−7	−8	8

Antes de la llegada de los micros de 32 bits, el rango de las palabras de 16 bits era una restricción que exigía recursos de programación para manejar números mayores. Asimismo, dado que se usaban los 16 bits para direccionar los registros de las memorias (véase más adelante), el resultado era una restricción a 65536 posiciones máximo (lo que exigía el empleo de recursos de *software* y *hardware* para superarla). Ahora ya puede comprenderse el porqué del revuelo provocado por la aparición de los micros de 32 bits: una doble palabra puede almacenar cualquier número sin signo comprendido entre 0 y 4.294.967.295 o, en números con signo, desde −2.147.483.648 hasta +2.147.483.647. Cuando no se realizan sumas tan largas, estos nuevos micros pueden también trabajar con dos palabras (16 bits), cuatro caracteres ASCII u ocho números BCD.

Sumas binarias

Trabajar con números binarios tiene su lado bueno y su lado malo. Del lado bueno está la mayor sencillez de las leyes de composición:

$$1+0=1 \qquad 1\times 0=0 \qquad 1-1=0$$
$$1+1=10 \qquad 1\times 1=1 \qquad 10-1=1$$

La parte mala reside en la falta de compacidad: es mucho más difícil para la vista y el cerebro recordar el número 100101001011 que el decimal equivalente 2379.

Si realizamos un ejemplo de suma binaria podremos fijar las reglas anteriores:

$$\begin{array}{rl} 10111 = \text{decimal } 16 + 0 + 4 + 2 + 1 = 23 \\ +\ 11101 = \text{decimal } 16 + 8 + 4 + 0 + 1 = 29 \\ \hline 110100 = \text{decimal } 32 + 16 + 0 + 4 + 0 + 0 = 52 \end{array}$$

donde al hacer la suma 1 + 1 = 10, hemos tomado el resultado 0 y el arrastre 1.

Octal y hexadecimal

En este apartado vamos a describir dos notaciones numéricas que se encuentran con cierta frecuencia y que pueden derivarse con facilidad de la binaria, pero con mayor compacidad y sencillez de uso.

El sistema **octal** utiliza la base de numeración 8, por lo que solamente se emplean los números del 0 al 7, y cada columna representará, por lógica, una potencia de 8. He aquí a continuación algunos ejemplos:

Binario	*Octal*	*Decimal*
111	7	7
1000	10	8
100000	40	32
111111	77	63

La conversión de binario a octal es muy simple: Todo consiste en subdividir la expresión binaria en grupos de tres bits empezando por la derecha y, después, calcular el equivalente decimal de cada uno de estos grupos. En efecto,

$$111111 = (111)(111) = (7)(7) = 77 \text{ octal}$$
$$100101001011 = (100)(101)(001)(011) = (4)(5)(1)(3) = 4513 \text{ octal}$$

El sistema **hexadecimal** emplea la base 16. Por ello, se precisan 16 símbolos diferentes para cada uno de los números. Los números convencionales del 0 al 9 son correctos para los diez primeros símbolos, y para los seis restantes se emplean las letras del abecedario: A = 10, B = 11, C = 12, D = = 13, E = 14, F = 15. He aquí algunos ejemplos:

Binario	*Octal*	*Decimal*	*Hexadecimal*
111	7	7	7
1000	10	8	8
1010	12	10	A
1111	17	15	F
100000	40	32	20
111111	77	63	3F

Una vez acostumbrado a su uso, resulta la notación más útil para el trabajo con ordenadores de 16/32 bits. Cada símbolo hexadecimal supone un *nibble,* dos son un byte, etc. La conversión binario a hexadecimal puede realizarse "in situ" por un método similar al del caso octal. En efecto, basta dividir el binario en grupos de cuatro bits, empezando por la derecha, y buscar su equivalente [7]. Un ejemplo:

111111 = (0011)(1111) = (3)(F) = 3F hexadecimal

Evidentemente, sobre todo cuando se interna alguien en la programación en código máquina, la solución más fácil y segura consiste en recurrir a una calculadora de bolsillo que tenga implementadas todas estas conversiones de forma directa.

También se han investigado bases de más de 16, pero la mejora en compacidad ha supuesto una pérdida excesiva en legibilidad. Un pionero de las matemáticas aplicadas al cálculo automático, Alan M. Turing (1912-1954), trabajó sobre base 32. Eso requiere, por tanto, 32 símbolos distintos, desde 0 hasta 9 y desde la A hasta la V. Así,

1111111111 = 1777 (octal) = 1023 (decimal) = 3FF (hexa) = VV (base 32)

La notación de Turing era aún más engañosa por cuanto estaba condicionada por los 32 caracteres arbitrarios en su teleimpresora de cinco canales (una de las primeras impresoras, funcionando con cinta de papel de cinco pistas).

Resumen de los sistemas de numeración

Los números pueden expresarse empleando bases diferentes de la familia de 10. En matemáticas de cálculo automático aparecen de forma natural las bases binaria (base 2), octal (base 8) y hexadecimal (base 16).

Lo positivo de la numeración binaria radica en la posibilidad de diseñar circuitos electrónicos rápidos y sencillos que ejecuten las operaciones aritméticas fundamentales (suma, resta, multiplicación y división) automáticamente. Como muestra, por ejemplo, valgan las calculadoras de bolsillo, como la mencionada anteriormente. El salto desde los conmutadores Sí/No a la aritmética requiere un pequeño rodeo a través de los operadores lógicos y del álgebra de Boole.

[7] Para este método, se rellenará el *nibble* más a la izquierda con tantos ceros como sean precisos hasta completar el total de cuatro.

Algebra de Boole

El álgebra de Boole se basa en las reglas matemáticas de operaciones lógicas, desarrolladas primeramente por George Boole (1815-1864).

La **lógica** está relacionada con las situaciones binarias por el hecho de estar basada en relaciones del tipo CIERTO/FALSO. Boole asignó el valor 1 a la condición de cierto y el 0 a la de falso, definiendo a continuación las operaciones NOT, AND y OR para los números binarios en lugar de las operaciones aritméticas convencionales [8].

En lenguaje corriente, si las proposiciones A y B son *ambas* ciertas, se dice que la proposición única (A AND B) también lo es. Si alguna de ellas, o ambas, es falsa, (A AND B) también es falsa. De forma análoga, si A es cierta, (NOT A) es falsa. Por regla general, se emplean los siguientes símbolos [9]:

$$\& = \text{AND}$$
$$* = \text{OR}$$
$$\sim = \text{NOT}$$

Sustituyendo cierto y falso por 1 y 0, las leyes de composición resultan totalmente similares a las de la aritmética binaria sin más que hacer "& = multiplicar" y "* = sumar". Existen, sin embargo, algunas sutiles diferencias:

Lógica	*Booleana*	*Binaria*
Falso OR Falso = Falso	0 * 0 = 0	0 + 0 = 0
Cierto OR Falso = Cierto	1 * 0 = 1	1 + 0 = 1
Falso OR Cierto = Cierto	0 * 1 = 1	0 + 1 = 1
Falso AND Falso = Falso	0&0 = 0	0 × 0 = 0
Cierto AND Cierto = Cierto	1&1 = 1	1 × 1 = 1
Cierto AND Falso = Falso	1&0 = 0	1 × 0 = 0
Falso AND Cierto = Falso	0&1 = 0	0 × 1 = 0

Hasta ahora todo va bien, pero las reglas difieren en los siguientes casos:

Lógica	*Booleana*	*Binaria*
Cierto OR Cierto = Cierto	1 * 1 = 1	1 + 1 = 10
NOT(Cierto) = Falso	−1 = 0	Compl. de 1 = 0
NOT(Falso) = Cierto	−0 = 1	Compl. de 0 = 1

[8] Nuevamente nos encontramos ante expresiones inglesas. Como siempre, en función de su amplia difusión (a nivel internacional), las respetaremos como originalmente aparecen en los textos ingleses. En cualquier caso, la traducción correcta al español es:

NOT = NO (negación)
AND = Y
OR = O

[9] El convenio seguido en esta obra es totalmente libre, aunque es fácil encontrarlo en muchas obras de distintos autores, particularmente de habla castellana.

Operadores lógicos compuestos

A partir de los operadores antes definidos, pueden generarse otros derivados, tal como, por ejemplo, NOT y AND pueden combinarse para dar lugar a NAND. A continuación, se muestran algunos de los más importantes en esquemas de cálculo automático y programación. Para ello, se emplean las llamadas Tablas de Verdad (porque nunca mienten) [10]:

AND

A	*B*	*A&B*
0	0	0
0	1	0
1	0	0
1	1	1

NAND (Not AND)

A	*B*	*∼(A&B)*
0	0	1
0	1	1
1	0	1
1	1	0

OR

A	*B*	*A*+*B*
0	0	0
0	1	1
1	0	1
1	1	1

NOR (Not OR)

A	*B*	*∼(A*+*B)*
0	0	1
0	1	0
1	0	0
1	1	0

EOR (Exclusive OR)

A	*B*	*(A*+*B)&∼(A&B)*
0	0	0
0	1	1
1	0	1
1	1	0

La función exclusive OR (O exclusiva) da respuestas de Cierto cuando lo es una u otra de sus variables, pero no ambas a la vez. En lenguaje común no se hacen distinciones entre ambas funciones OR, lo que indica que éste no es adecuado para "razonar" con un ordenador.

Lógica para programación

La precisión del álgebra booleana es de vital importancia para el programador, puesto que a menudo está interesado en poder evaluar la respuesta de una rutina (cierto o falso) para ejecutar diferentes programas según aquélla. Un ejemplo que resultará familiar es el relacionado con una (supuesta) declaración de la renta:

> Si usted es hombre, casado, de más de cuarenta años y con ingresos inferiores a 1.500.000 pesetas, o mujer, soltera, de menos de cincuenta años y con ingresos superiores al millón de pesetas, conteste la línea 3. En otro caso, salte al punto 12.

[10] En realidad, con una única función del tipo NAND pueden generarse todas las demás, incluidas las iniciales. Lo mismo ocurre con NOR, etc. No así con AND o con OR solamente, tal y como se demuestra en el álgebra de Boole. Sin embargo, para los propósitos del libro pueden tomarse como ciertas (con esta salvedad) las afirmaciones del texto, puesto que aquí no se persiguen intereses teóricos estrictos.

En términos de álgebra de Boole, esto se resume en evaluar una expresión de la forma:

$$(\sim M \times C \times (E>40) \times Ingr.<1.500.000) +$$
$$+ (M \times \sim C \times (E<50) \times Ingr.>1.000.000)$$

donde cada término individual, llamado variable (o expresión) booleana, se iguala a 1 si es cierto y a 0 si es falso. Si la respuesta obtenida es un 1, se pasa al punto 3, mientras que si es un 0 se salta al 12, según las instrucciones. Evidentemente, si usted es hombre, entonces M (abreviatura de Mujer) es falso e igual, por tanto, a 0, mientras que ~M (es decir, No Mujer) es cierto e igual a 1, y así con el resto.

Puertas lógicas

Electrónicamente es posible construir circuitos, llamados **puertas,** que combinan las señales a su entrada de acuerdo con las reglas del álgebra de Boole. La figura 1.5 indica la representación de dichas puertas en los circuitos. En los micros modernos, estas puertas están formadas por transistores "embutidos" en silicio. El M68000 tiene, parece que por pura coincidencia, 68000 de tales puertas. El anterior micro, el 6800, tenía 6800 transistores, lo que indica la lógica de la firma Motorola para sus numeraciones. En cuanto al 1 y al 0, pueden representar dos estados eléctricos diferentes: el 0 puede asociarse a + 5 voltios y el 1 a 0 voltios. La función NOT se denomina **inversor,** puesto que realmente invierte el 0 por el 1, y viceversa. Las puertas son los bloques constitutivos básicos de los ordenadores: pueden interconectarse de infinitas formas para realizar una gran variedad de funciones, tales como la decodificación y la aritmética binaria. Las pastillas IC (circuitos integrados) están disponibles con cientos de variedades, permitiendo cualquier combinación concebible al diseñador. Y, si usted consume una cantidad suficientemente alta, puede conseguir una pastilla especialmente diseñada para sus propósitos que le evitará tener que interconectar muchos circuitos integrados convencionales (Custom).

Desde las puertas hasta las sumas

Veamos ahora de qué forma el álgebra de Boole permite diseñar circuitos tales que dan el salto desde la lógica hasta la aritmética. En la figura 1.6 puede verse una tabla de un sencillo sumador binario A + B, donde A y B pueden ser 0 ó 1. Las salidas S y C: S es la suma y C el acarreo, que, a su vez, pueden ser 0 ó 1 de acuerdo con nuestras reglas de suma binaria. Puede comprobarse que S coincide con la función EOR (OR exclusiva) de A, B, mientras que C es A&B. El circuito de la figura 1.6 se obtiene reemplazando los símbolos de silicio equivalentes: dos puertas AND, una puerta OR y un inversor. Lo más importante es que, gracias a la tecnología de los

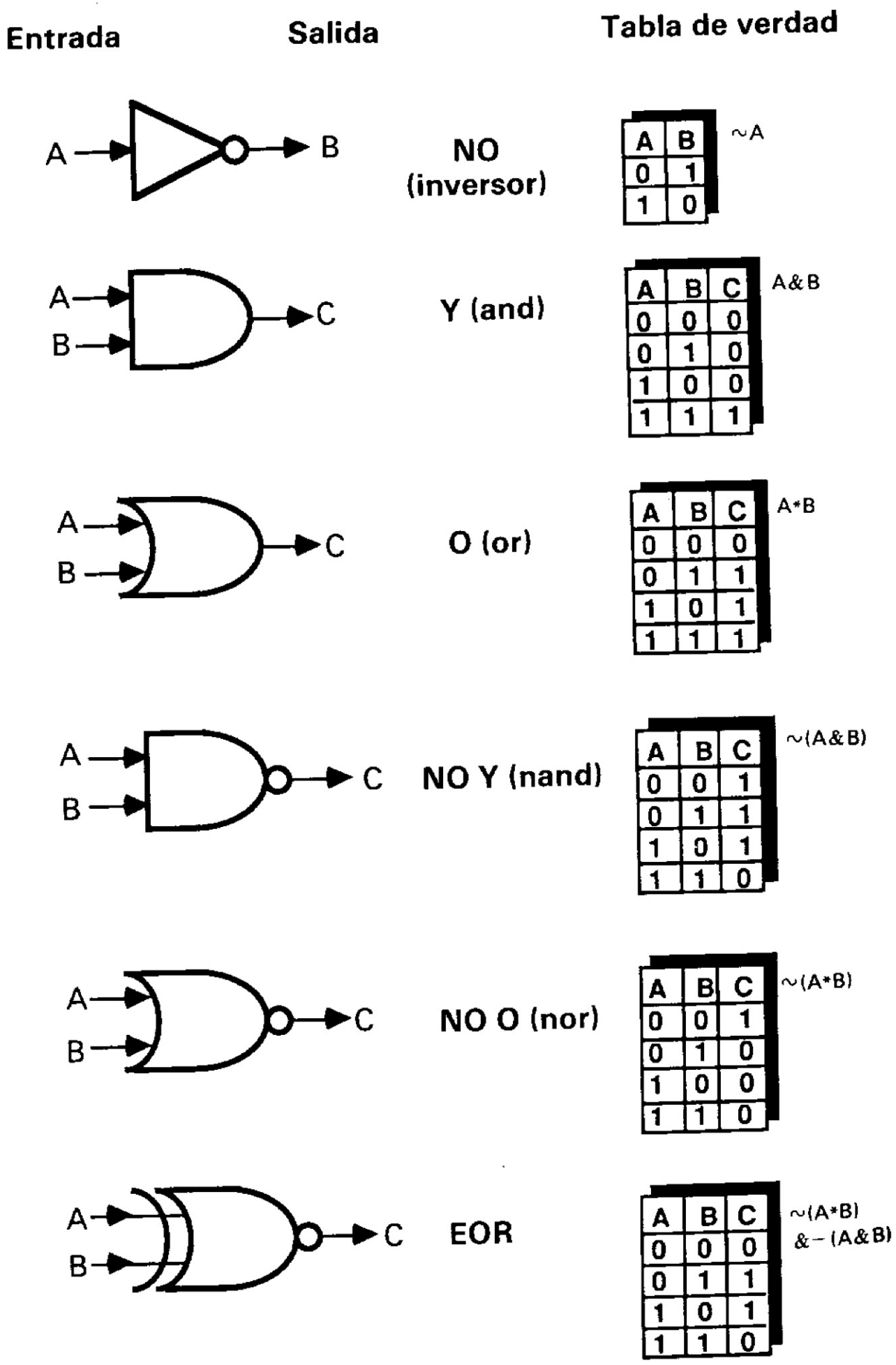


Figura 1.5
Puertas lógicas

ENTRADA		SALIDA	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

S = Suma = (A * B) & ~(A & B)
C = Acarreo = A & B

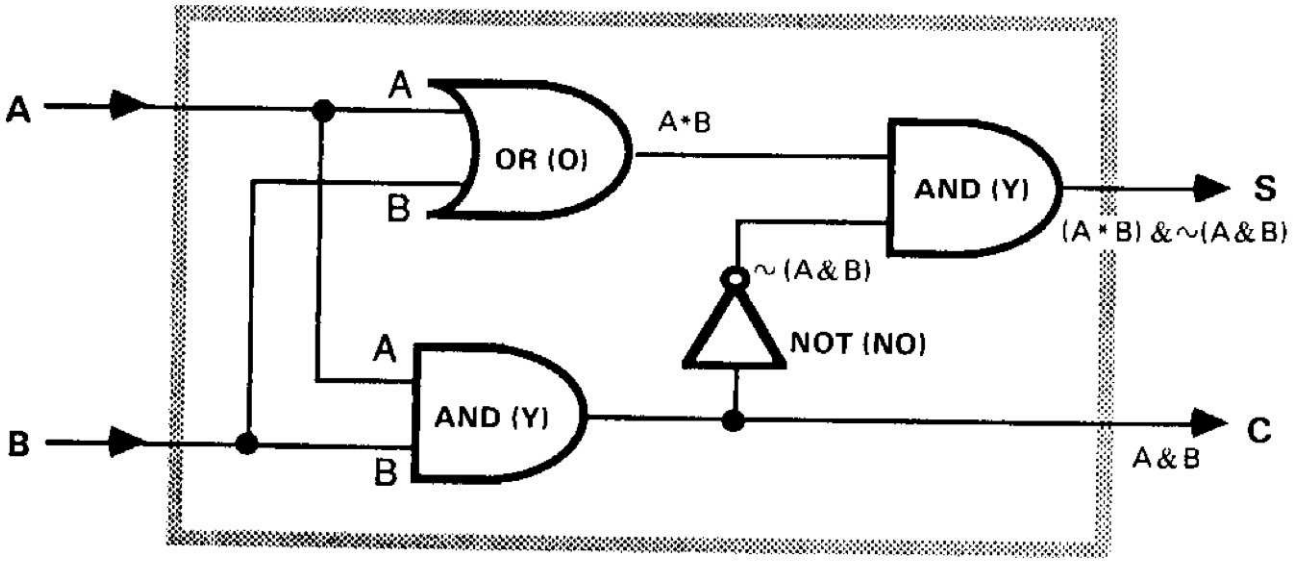


Figura 1.6
Sumador binario

circuitos integrados, pueden incorporarse (integrarse) muchos miles de tales circuitos en una pastilla producida masivamente.

El semisumador de la figura 1.6 realiza la suma de dos bits exclusivamente, pero no es difícil extenderlo a sumas de 8 y 16 bits con otros tantos circuitos. Un sumador precisa de tres entradas, puesto que, además de los dos bits, hay que tener en cuenta el acarreo precedente, aunque su principio de funcionamiento es idéntico al mostrado en la figura 1.6.

Biestables

Antes de dejar el estudio de los circuitos básicos de cómputo, debemos analizar una ingeniosa combinación de puertas que nos indica de qué forma

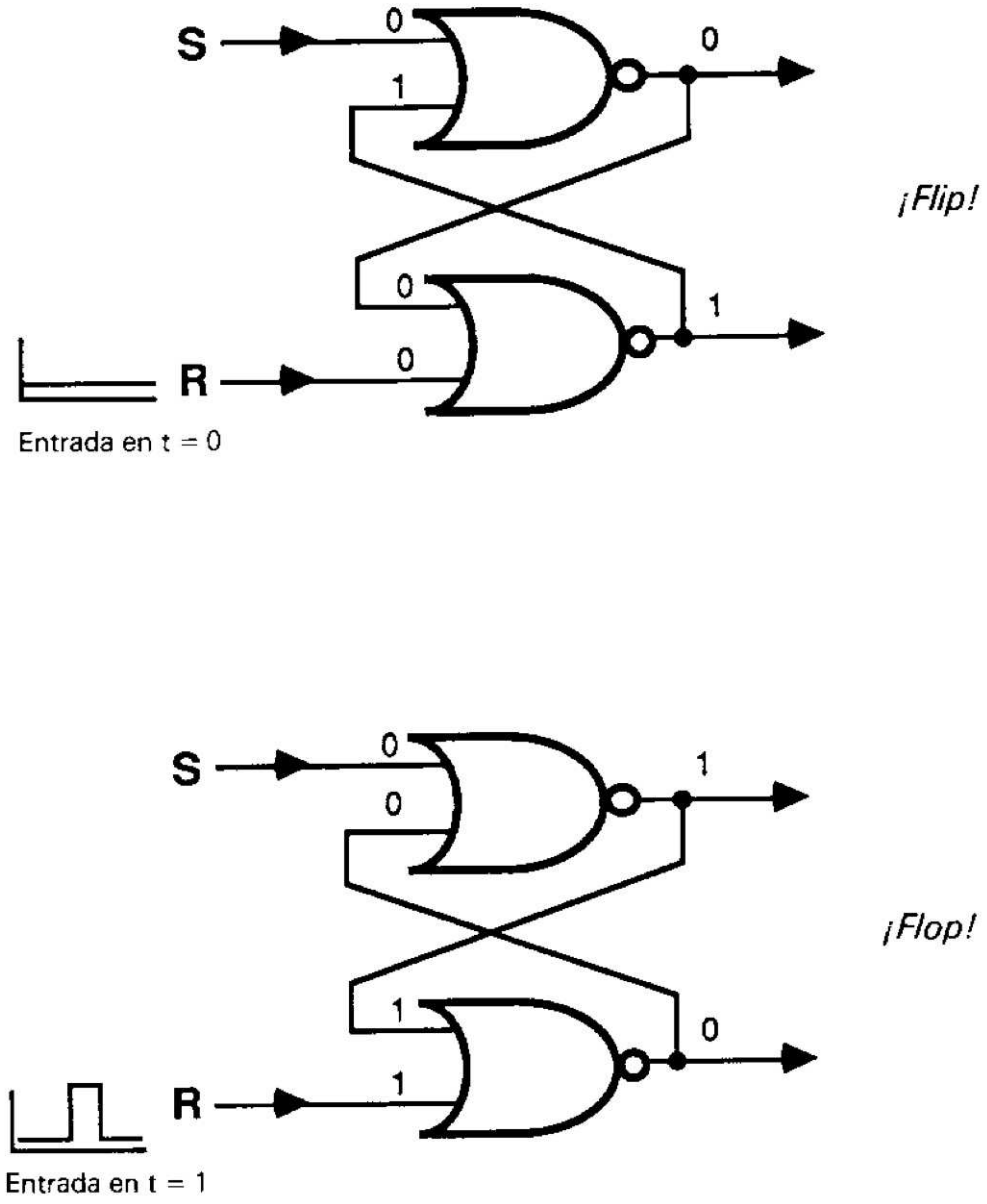


Figura 1.7
Biestable (Flip-Flop) *básico*

pueden almacenarse electrónicamente los datos. El *biestable* consiste en dos puertas NOR, conectadas de la forma que indica la figura 1.7. Conectando pulsos en las entradas S (set) y R (reset) de distintas maneras, se consigue que las dos salidas basculen entre los valores 0/1 y 1/0[11]. Estas salidas se conservan —"son recordadas"— mucho tiempo después de que las entradas que las provocaron hayan desaparecido, de forma que ya disponemos de la forma básica de almacenar 1 bit y de la forma de leer o escribir este

[11] En este punto, recomendamos que el lector consulte obras más extensas dedicadas a la electrónica digital, donde podrá ver cómo funcionan correctamente, sus distintos tipos y la mejor forma de interconectarse. Por otra parte, queremos recordar que a los biestables también se les conoce como *flip-flops* (uso directo desde la onomatopeya inglesa) o como "báscula" (por las razones dadas en el texto), aunque cada vez es más común el de biestable. Asimismo, las entradas set y reset son una adaptación directa del inglés, que usualmente se conservan, aunque pueden encontrarse obras que los hayan traducido por su equivalente español de "Puesta a Uno" y "Puesta a Cero".

valor almacenado. Con una idea semejante a la de los sumadores, pueden conectarse varios de estos biestables en serie, de manera que se obtienen unidades de almacenamiento de varios bits, conocidas como **registros.**

Hemos visto, de forma muy general, cómo existe un equivalente en silicio para cada una de las puertas lógicas básicas AND, OR y NOT, y sus derivadas NAND, NOR y EOR. Estas puertas lógicas pueden interconectarse para dar lugar a circuitos que realicen funciones mucho más complejas, incluyendo la suma binaria y el almacenamiento.

Para coordinar el funcionamiento de estas puertas, se precisa otro ingrediente: el tiempo.

Relojes

En la realidad, la configuración de unos y ceros de las entradas A, B, S y R de las figuras 1.6 y 1.7 llegan como una ristra de pulsos eléctricos de alta velocidad sincronizados por un reloj del sistema. Este reloj controlado por un cristal de cuarzo (que puede estar incorporado al *chip* de la CPU, o formar parte de un circuito aparte) marca la pauta de ejecución de las actividades de una MPU. En términos generales, cuanto más rápido sea el reloj, más rápido será el ordenador. Un reloj típico del MC68000 funciona con una frecuencia de 8 MHz (8 millones de ciclos por segundo), lo que supone un tiempo de ejecución de ciclo de 125 nanosegundos (un nanosegundo es una milésima de millonésima de segundo, 1/1.000.000.000, el tiempo que necesita la luz para recorrer 30 cm aproximadamente).

Microordenadores: Los tres elementos

Ahora vamos a hacer un breve análisis de la organización global de un sistema típico de microordenador, con vistas a identificar los elementos constitutivos con prioridad de un posterior análisis más detallado. Veremos cómo un microordenador puede ser visto como "tres cajas negras alrededor de un *bus*" [12].

Bus del sistema

La figura 1.8 muestra el "*bus* del sistema", una especie de caminos de cables, alrededor del cual se disponen los componentes principales, **MPU, memoria** y **E/S,** para comunicarse entre sí.

---

[12] Nuevamente hemos de recurrir a emplear un término en su forma inglesa original. Sin embargo, el caso de la expresión "BUS" es uno de los más justificados, por cuanto no existe traducción precisa al español, siendo usado comúnmente por nuestros especialistas.

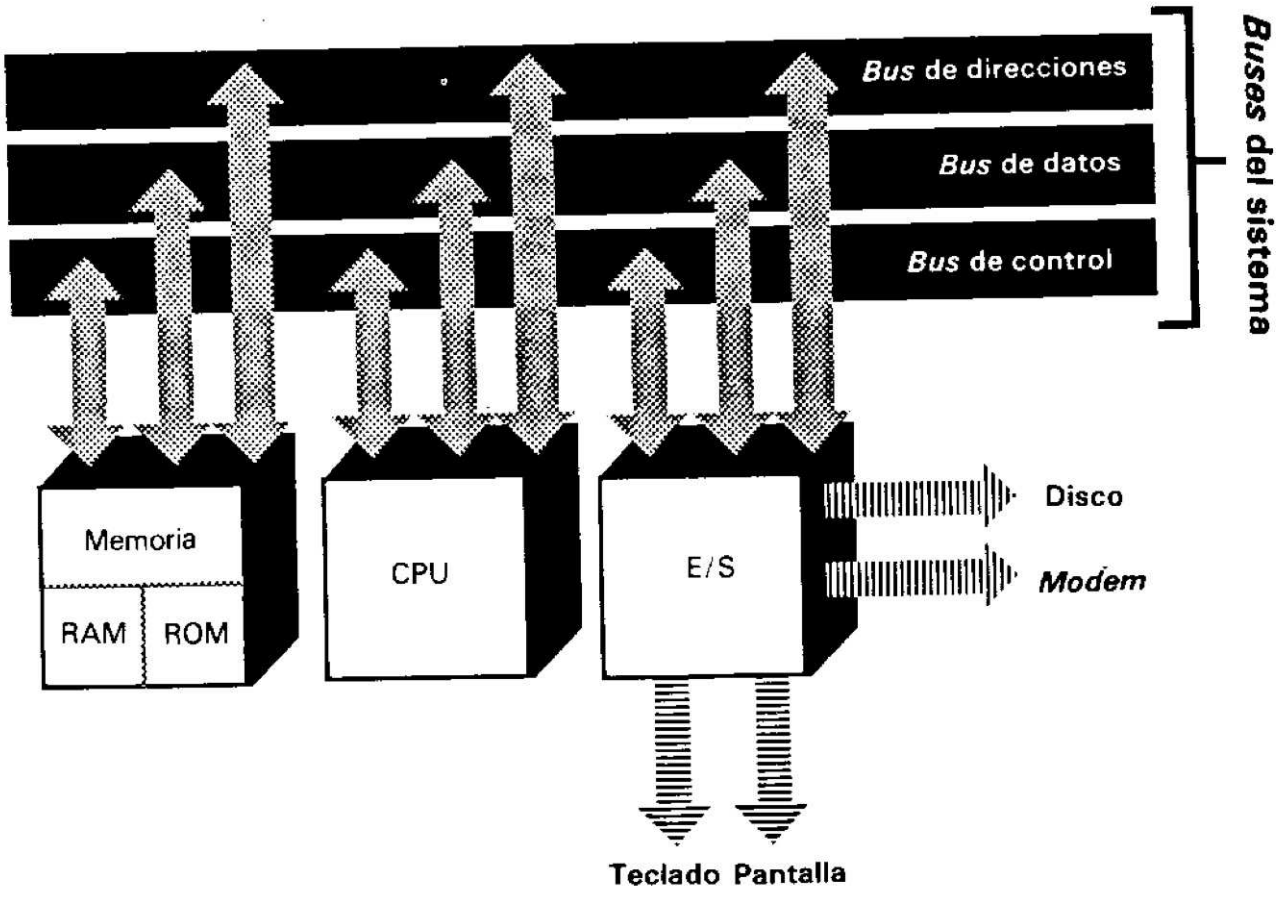


Figura 1.8
Buses *de los sistemas microordenadores*

La memoria

La caja denominada *memoria,* subdividida en RAM y ROM, representa el medio principal de almacenamiento *inmediato* de datos y programas. Cada "pieza" de la memoria posee una única dirección propia que permite un acceso rápido y directo desde la MPU. ROM es la memoria de "sólo lectura", mientras que la RAM (del inglés *Random Access Memory*) puede ser leída y escrita.

E/S (entrada/salida)

La caja única marcada con E/S (entrada/salida) comprende una multitud de dispositivos —unidades de disco, terminales de usuario, impresoras, *modems,* etc.—, cada uno con su circuito de adaptación particular (interfaz), conocidos como controladores de E/S, soportados por los programas especiales para ello, tales como los circuitos de terminales. La E/S es la parte más visible y con mayor influencia sobre el usuario medio. En el centro de nuestros profundos diseños microelectrónicos, debemos hacer sitio para el usuario medio, tecleando datos y sacando gráficas.

El *bus* del sistema lleva tres tipos de señales: **datos, direcciones y control.** En algunos ordenadores, estos tres conjuntos de señales están eléctri-

camente aislados (cables independientes). A tales tipos se les conoce como sistemas *multibus,* puesto que pueden identificarse por separado un *bus* de datos, de direcciones y de control. En sistemas más económicos existe un único *bus* compartido temporalmente por los tres mediante el esquema conocido por multiplexación.

Funcionamiento del microordenador

En un ciclo de lectura (Fig. 1.9), la MPU toma datos (números o instrucciones) en la memoria o en la E/S, enviando las señales apropiadas por el *bus* de control. Si el *bus* estuviera ocupado, la MPU deberá esperar durante unos pocos ciclos (a lo que llamamos un estado de *espera*).

Cuando el *bus* está libre, la MPU coloca una dirección en el *bus* de direcciones. Esta es decodificada por circuitos especiales de la memoria o de la E/S y, si todo va bien, el dato hallado en la dirección solicitada es llevado desde la memoria o E/S al *bus* de datos. En este momento, el *bus* de control indica a la MPU que el dato está disponible en el *bus* de datos. Una vez que el dato es transferido al receptor *(buffer)* [13] de la MPU, el *bus* de control indica su disponibilidad nuevamente. En seguida veremos de qué forma la MPU maneja los datos recibidos.

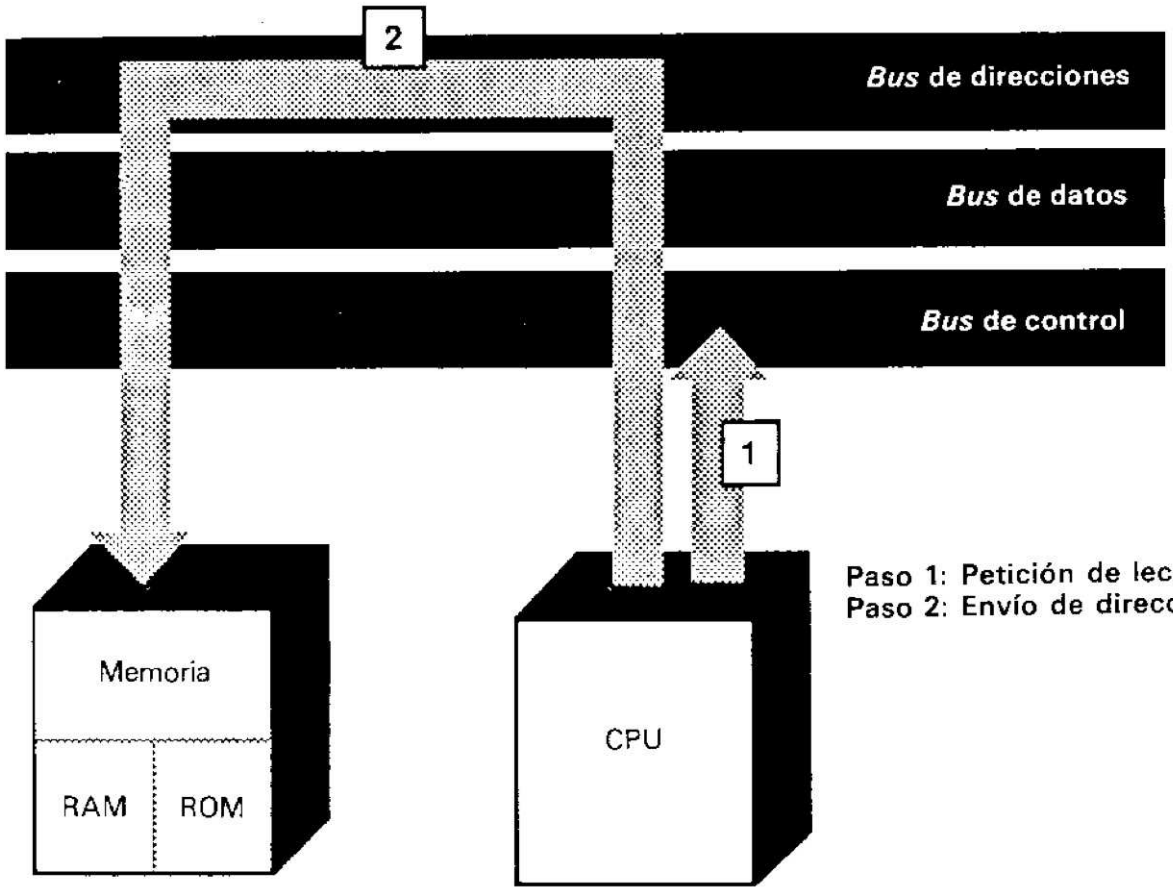
La anchura del *bus* de datos (medida en bits) marca el número de datos que pueden ser adquiridos durante cada ciclo de lectura, por lo que cuanto más ancho es mejor. Como veremos, el tamaño del *bus* de datos juega un papel fundamental en la determinación de la flexibilidad, el rendimiento y el *coste* de una MPU. El aspecto del coste se debe a que cada bit del *bus* de datos precisa del correspondiente *pin* [13] en la pastilla de la MPU y de las pistas respectivas en su interior.

Cuando se habla de ordenadores de 8, 16 ó 32 bits, sin más especificaciones, *debemos* entender que se refiere al tamaño del *bus* de datos de la MPU. Pero tenga en cuenta que algunos vendedores tienden a mentir. Recuerde la ley de Gerswhin: "¡No necesariamente!".

En realidad, se precisan cuatro tamaños para caracterizar debidamente una MPU: el del *bus* de datos, el de la ALU (unidad aritmética y lógica), el de los registros y el del *bus* de direcciones. Así, ¿quién desea comprar un micro de 8/16/32/20 bits? (por ejemplo, el 68008).

El tamaño del *bus* de direcciones determina el número total de distintas direcciones de memoria (incluyendo E/S) que pueden ser directamente accedidos desde la MPU y que, por tanto, implican la máxima memoria utilizable. Como en el caso del *bus* de datos, cuanto mayor, mejor será, pero también más caro resultará. Aquí podemos usar nuestra teoría anterior sobre los mensajes-por-bit. La mayor parte de los micros de 8 bits tiene un *bus* de direcciones de 16 bits que, como veíamos, puede alojar un máximo

[13] Como en las anteriores ocasiones, nos encontramos con expresiones inglesas que son del dominio común en nuestro idioma. Así, *pin* en lugar de terminal, *buffer* en lugar de amplificador digital, pero que respetamos con la misma norma de siempre.



Buses del sistema

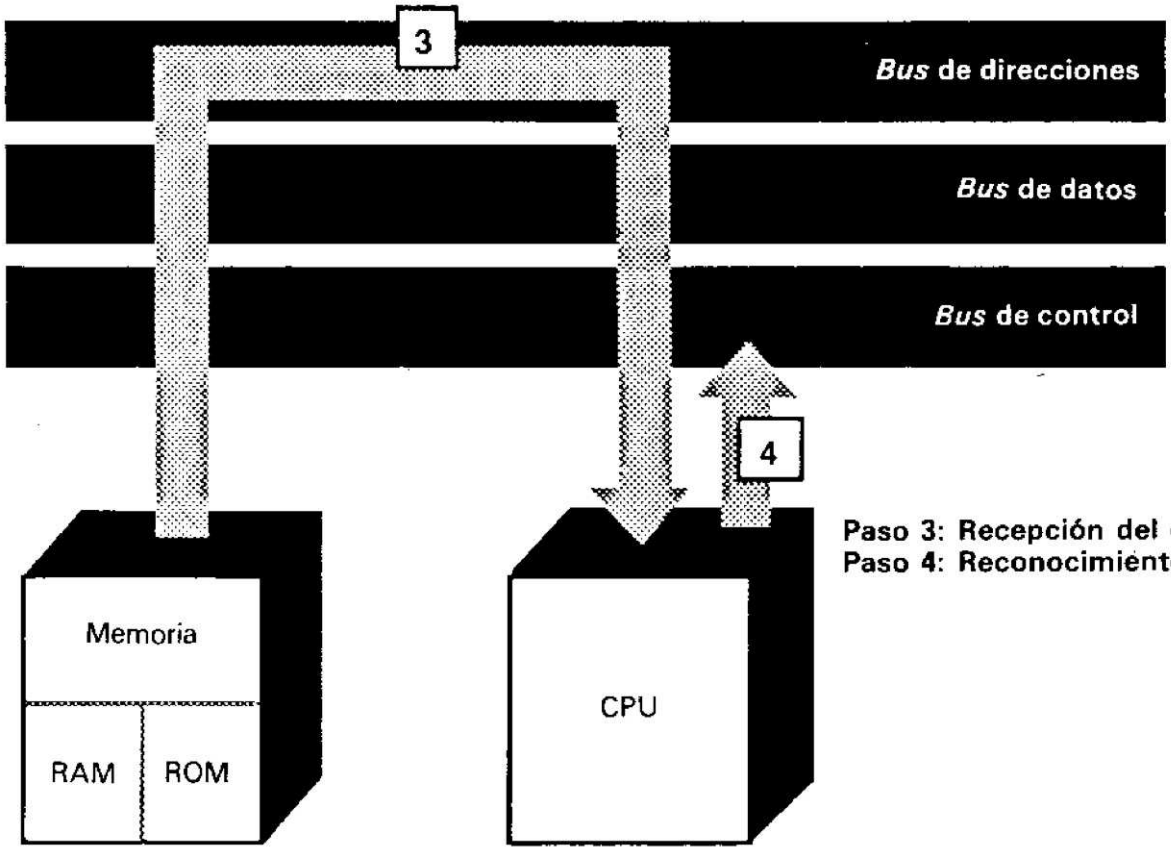


Figura 1.9
Ciclo de lectura

de 64K direcciones de memoria (cada una de las cuales corresponde a un byte), puesto que $2^{16} = 65.536 = 64K$.

Aunque existen muchas formas inteligentes de acceder a más de 64K con un *bus* de 16 bits, todas ellas exigen un aumento del coste y del tiempo. La familia 68000 dispone de tamaños de *bus* de direcciones comprendidos entre 20 y 32 bits, lo que significa espacios de direccionamiento comprendidos entre un Megabyte y cuatro Gigabytes.

El ciclo de escritura (Fig. 1.10) permite que la MPU envíe datos a memoria, o E/S. La MPU indica su intención *(solicitud de escritura)* al *bus* de control, y cuando el camino está despejado, coloca el dato en el *bus* de datos y la dirección de destino en el *bus* de direcciones. Entonces la MPU puede pasar a ejecutar la siguiente instrucción.

Recuérdese que el trasiego de datos aparece como rápidas ristras de pulsos (ceros y unos) sincronizadas por el reloj del sistema. Cuando hablamos de ciclos de lectura o escritura, estamos hablando de períodos de tiempo definidos por la velocidad del reloj del sistema.

Resumen del *bus* del sistema

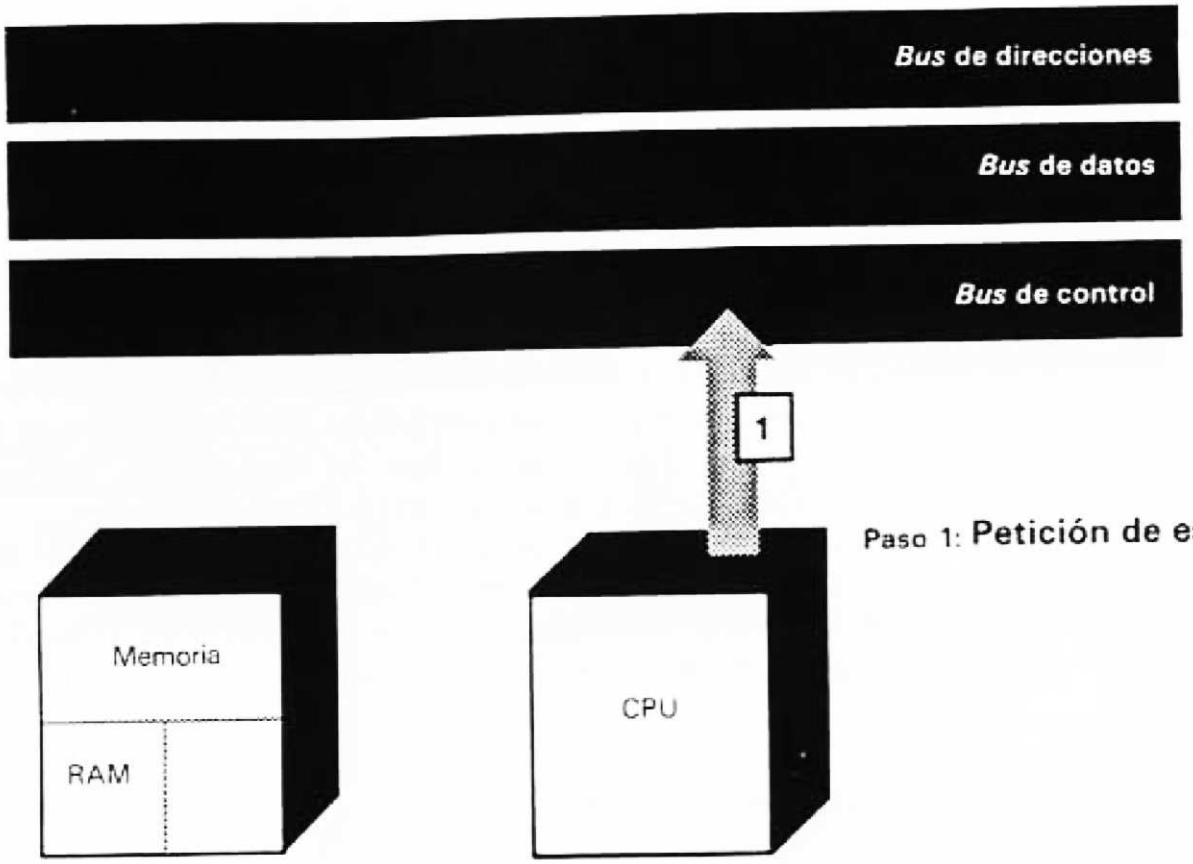
El *bus* del sistema enlaza entre sí las principales unidades del ordenador, ofreciendo un camino eléctrico entre la MPU, la memoria y la E/S. Por él circulan tres tipos de señales: datos, direcciones y control. Ahora que usted conoce de forma general la interacción entre la MPU y el *bus* del sistema, puede mirar al interior de la MPU y comprender cómo funciona. Aunque existen millones de MPU diferentes, hay una arquitectura "genérica" que sirve para todas.

La unidad central de proceso (CPU)

Lo primero que se deduce de la observación de la figura 1.11 es que el microprocesador dispone de tres vías de comunicación con el "mundo exterior", que son las citadas de los *buses* de datos, direcciones y control. Además, dispone también de un *bus* interno (puede haber incluso varios) que conecta las distintas unidades funcionales de la CPU. De hecho, en el interior de la CPU, las señales de datos y control circulan por el *bus* interno, sincronizadas por el reloj del sistema —no distinto del que dibujábamos como regulador del tráfico en el *bus* del sistema.

En correspondencia con los programas de usuario del sistema de ordenador, el funcionamiento interno de la CPU es gobernado por microprogramas almacenados en una ROM interior especial (esto es cierto para el 68000 y para muchas otras CPU, pero en otros casos se emplea lógica de control cableada).

Sigamos el camino de una secuencia típica de eventos:



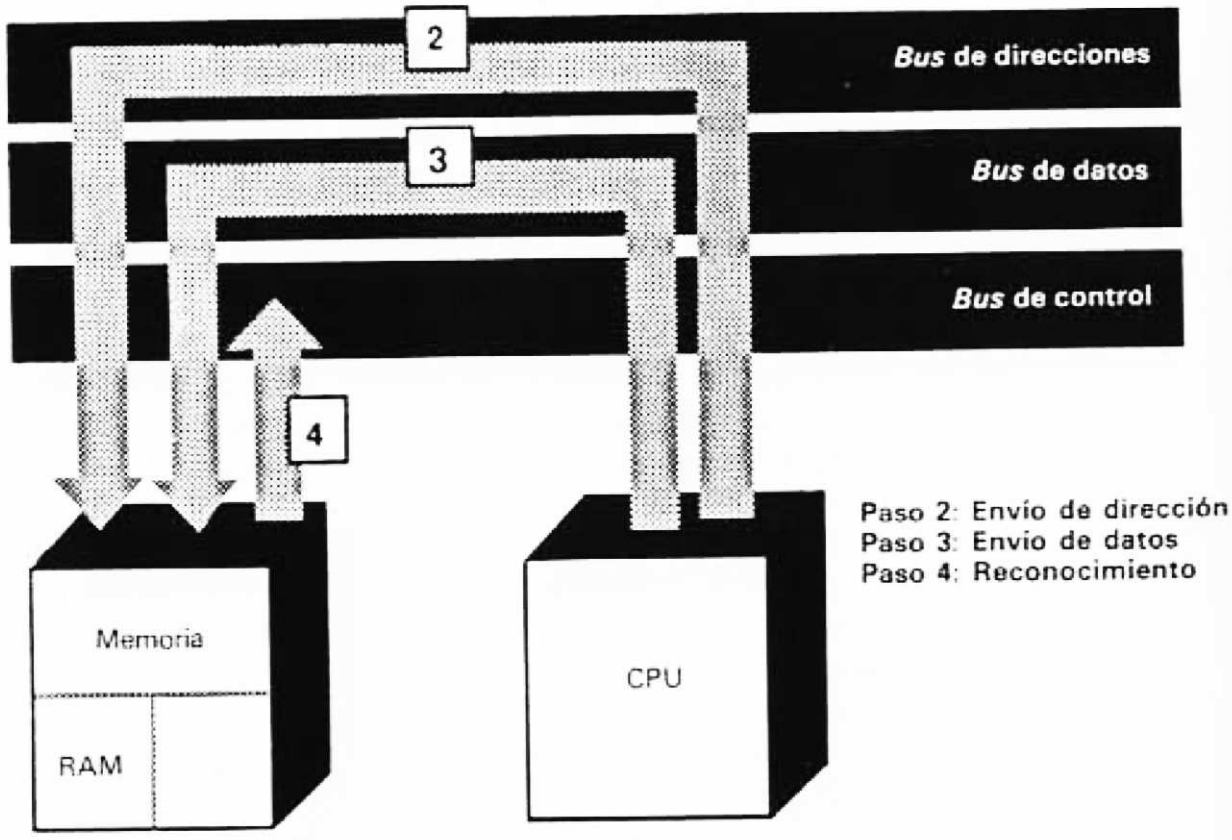


Figura 1.10
Ciclo de escritura

relativas a las operaciones (resultado negativo, resultado cero, rebose, etc.).

Una vez que se han completado los pasos del 1 al 4, la CPU está dispuesta para empezar la siguiente instrucción, puesto que, como usted recordará, el PC (contador de programa) ya se había incrementado y, por tanto, contiene la dirección de la siguiente instrucción. Debemos, pues, recordar que, por regla general, las instrucciones están dispuestas secuencialmente en la memoria.

Entre las muchas posibles desviaciones de la sencilla ordenación secuencial antes citada, una muy común es la denominada de **ramificación.** En este tipo de ordenamiento, una determinada circunstancia emanada del propio programa (o de su ejecución) requiere la ejecución de una instrucción que se halla fuera de la secuencia normal. La ramificación se consigue mediante la generación del nuevo valor del PC necesario.

Otras desviaciones posibles pueden venir provocadas por errores, excepciones o interrupciones externas. De estos tipos nos ocuparemos en los próximos capítulos, puesto que el 68000 dispone de unos mecanismos únicos para el tratamiento de errores y excepciones.

Otro interesante hecho es el concerniente a la velocidad *real* con que se ejecutan las cuatro etapas básicas. La velocidad de una CPU se mide en MIPS (millones de instrucciones por segundo). Desde luego, no todas las instrucciones exigen el mismo tiempo, de forma que la medida en MIPS puede resultar engañosa. Una determinada instrucción puede ocupar entre 4 y más de 200 ciclos básicos. Para darnos una idea, el MC68020 funciona a una velocidad mantenida de 2 a 3 MIPS, con puntas de velocidad ocasionales de hasta 8 MIPS.

Prebúsqueda y segmentación

Al observar nuestras cuatro etapas —búsqueda de instrucción, decodificación, captura de datos y ejecución—, puede uno imaginarse la enorme actividad del *bus*, tanto en el sistema total como en el interno de la CPU. En la lucha sin fin para conseguir mayores rendimientos de los sistemas, los diseñadores intentan, obviamente, aumentar la velocidad de ejecución de cada ciclo, especialmente reduciendo la posibilidad de retardos (estados de espera) consecuencia de contenciones del *bus* o de tiempos de acceso a memorias lentas. Pero cuando se han alcanzado los límites prácticamente posibles, aún queda la posibilidad de aumentar la velocidad **solapando** operaciones, lo que se conoce por **concurrencias.** Se ve fácilmente que se pueden solapar (ejecutar simultáneamente) varios de los pasos del 1 al 4. Una estrategia típica de **prebúsqueda** supone tener tres instrucciones en la CPU a un mismo tiempo: en ejecución, en decodificación y una tercera siendo buscada. Con el método de la **segmentación,** puede simultanearse un mayor número de instrucciones. Estos métodos exigen, a menudo, disponer de distintos recursos en el sistema, suponiendo que la pastilla de la CPU tenga los

Una fuente importante de datos, tanto de entrada como de salida, para la ALU es un área especial de memoria rápida, organizada como **registros** de longitud fija. A diferencia de los registros temporales y los *buffers*, éstos sí son accesibles para el programador.

Registros

Los registros juegan un importante papel, porque son los que dan a la CPU su personalidad y le confieren su programabilidad, y los hay de todos los tamaños, formas y colores. El 68000 básico tiene 17 registros de propósito general, cada uno de 32 bits. En el capítulo 3 nos ocuparemos con detalle de este asunto, puesto que la organización de los registros está intimamente ligada con el set de instrucciones.

Cómo son las memorias

La memoria rápida más normal en los modernos ordenadores es la memoria RAM *(Random Acces Memory)*[1]. Hay pastillas de RAM de varios tamaños, y se montan en placas para conseguir un total por placa comprendido entre 16 Kbytes y 2 Megabytes, y un sistema puede tener más de una de tales placas. La estructura física de la memoria carece de importancia para el programador, el cual está interesado solamente en el esquem

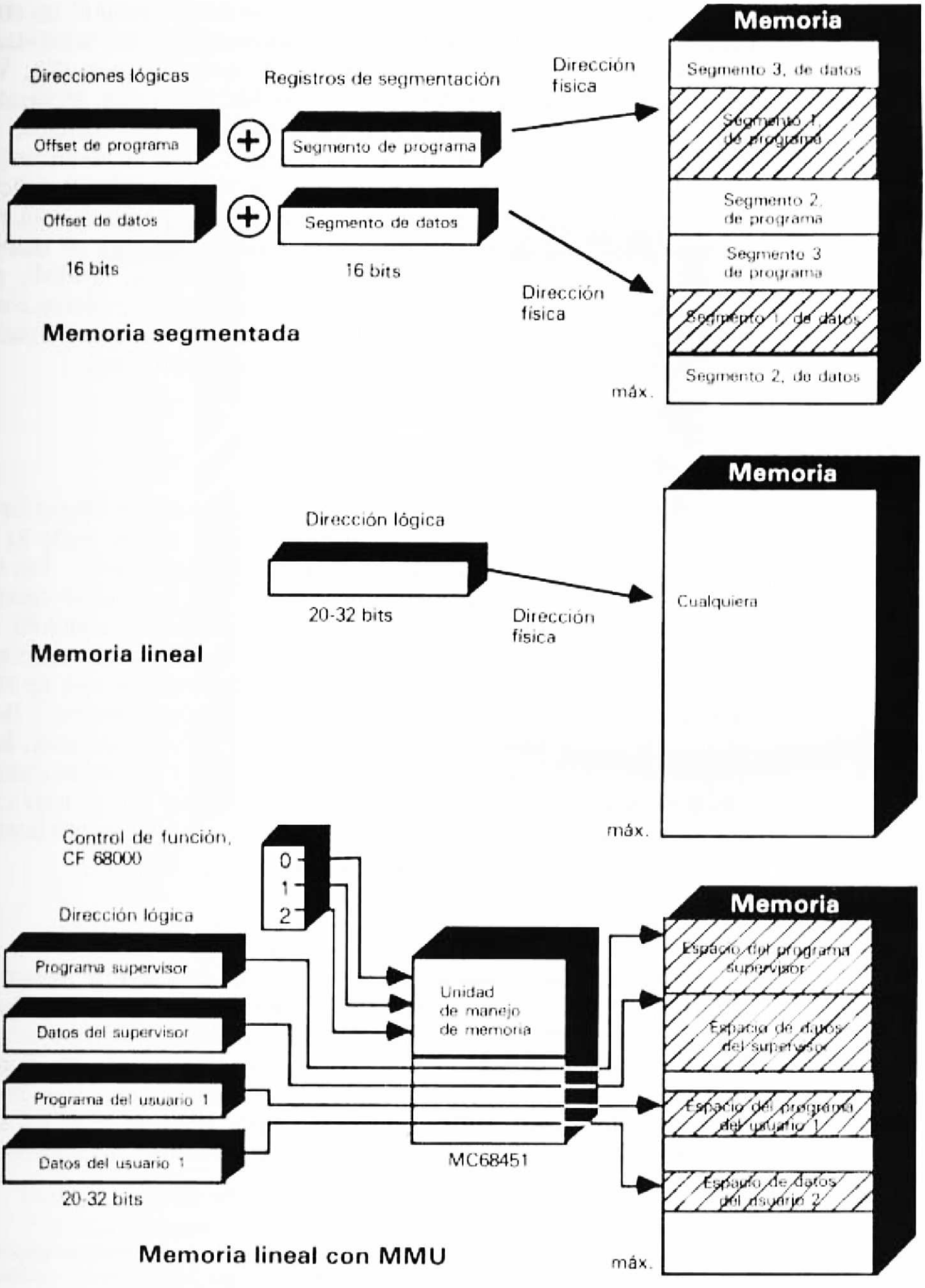


Figura 1.13
Memoria segmentada y memoria lineal

él (de 1 Megabyte hasta 4 Gigabytes). Con dicho tamaño, las tareas pueden distribuirse según el óptimo para sus necesidades. La seguridad debe conseguirse desde "fuera", y el 68000 está diseñado para ello. Varios *chips* (pastillas) especializadas, tales como el MC68451 (de Motorola), realizan algo parecido a la segmentación. Estas unidades son llamadas unidades de manejo de memoria o MMU. En la figura 1.13 se ve cómo realizan su función. Los terminales FC (control de función) del 68000 indican automáticamente el **estado del procesador**, que significa cuándo está en modo **supervisor**, cuándo en modo usuario y cuándo se trata de datos o de programa. Con estos valores y la dirección lógica real, la MMU construye la dirección física. Además, la MMU puede ofrecer protección contra escrituras indebidas en ciertas áreas de la memoria y disponer un espacio separado para operaciones de DMA (acceso directo a memoria).

Memoria de masas

Una de las más importantes características de las memorias RAM en el diseño de ordenadores es su **volatilidad**. En la mayor parte de las RAM se pierde toda la información al desconectar la alimentación. Por tanto, es necesario disponer de una memoria permanente, lo que se consigue con diversos tipos de almacenamientos, conocidos como **memoria de masas** [18], tales como discos duros o flexibles, cintas de papel y magnéticas.

Los programas se almacenan, por lo general, en uno de estos medios, tales como discos flexibles *(diskettes)*, y deben ser vertidos a la RAM antes de utilizarse. Durante una ejecución típica de un programa, habrá mucho tráfico de E/S: datos llevados al disco, y datos y programas leídos en el disco. El rendimiento del sistema total vendrá, por tanto, muy condicionado por la memoria RAM disponible: cuanto más RAM, menos accesos al disco.

Memoria virtual

A menudo, la memoria RAM disponible es bastante menor que la capacidad de direccionamiento de la CPU. De hecho, muchos tendrán que bajar los *chips* de memoria hasta que veamos muchos 68020 con toda la memoria RAM de que es capaz: ¡4 Gigabytes! La técnica llamada de **memoria virtual** (MV), inventada por Ferranti Ltd. (Manchester, Gran Bretaña), en la década de los cincuenta (y reinventada por IBM en la de los setenta), permite acceder a los datos residentes en disco como si de memoria RAM se tratase.

---

[18] La expresión correcta en castellano debería ser la de **memoria permanente**, puesto que éste es su verdadero significado. Sin embargo, este tipo de memoria es siempre de gran capacidad (por obvias razones de rentabilidad), lo que ha motivado su actual nombre. No se citan, por su obsolescencia, las otrora inevitables memorias de ferritas, que seguramente recordarán muchos lectores.

lo dicho para el *software*, el *firmware* ya viene totalmente listo para funcionar a la primera.

El *firmware*, desde luego, se escribe y comprueba con mucho más cuidado antes de ser permanentemente introducido en el sitema. Un buen ejemplo lo constituye el intérprete de BASIC del IBM PC™.

Los costes

Los avances espectaculares en la producción masiva de circuitos integrados *(chips)* continúan provocando la caida de los precios. Haciendo un simil, si hubiera ocurrido lo mismo en la industria del automóvil desde hace treinta años, un Rolls-Royce costaria unas 15.000 pesetas. Desafortunadamente, hoy el *software* se ha convertido en una difícil ocupación, resistente a todos los esfuerzos tendentes a la producción masiva. La programación es una intensa labor, que exige mentes creadoras e impone salarios crecientes.

Los costes del *software* han resultado ser el factor dominante en todos los proyectos, grandes y pequeños, de ordenadores. En nuestro simil del coche, el mismo Rolls necesitaria un chófer con un sueldo anual de 15 millones de pesetas (¡!) y funcionaría con una gasolina de 1.500 pesetas el litro (¡!).

A menudo se emplea la expresión de **ingeniería *software*** para describir los esfuerzos realizados en orden a conseguir los métodos de producción, tales como la dirección de proyectos y el control de calidad, en la parte más intelectual de la programación.

El mayor desafío con que se enfrenta la industria actual de los ordenadores es la corrección del desequilibrio entre los costes del *hardware* y los del *software*.

La familia del 68000 fue diseñada "por programadores para programadores" y constituye un intento serio en este sentido.

Conclusión

Aquí se termina nuestro galopar a través de varios de los conceptos más básicos. Muchos de ellos serán ampliados y, esperamos, clarificados en los próximos capítulos.

2

La familia 68000

En este capítulo se describe someramente la evolución histórica de la familia de microprocesadores Motorola M68000, de forma que podamos situarla en el sitio que le corresponde en el inquieto mundo actual de los micros. Asimismo, describiremos algunos de sus componentes y le mostraremos su aspecto (Fig. 2.1). Por otra parte, le sugerimos que vuelva a leer este capítulo después de haber terminado el 3 y el 4. Varias de las características discutidas aquí adquieren su importancia a la luz de las instrucciones en funcionamiento.

Introducción

El primer micro 68000 de 16/32 bits (véase figura 2.2), presentado por Motorola en 1979, representó un salto cuántico hacia adelante, en cuanto a potencia y flexibilidad se refiere. Desde entonces, tal como prometió, Motorola ha desarrollado una familia completa de MPU (unidades de microprocesadores) y circuitos integrados *(chips)* de soporte que van, en precio y rendimiento, desde el "económico" MC68008, pasando por el 68000, el 68010 y el 68012, hasta el último de 32/32 bits, el MC68020. El 68000 mereció el calificativo de **supermicro,** dejándonos sin los adecuados superlativos para calificar el 68020. Para añadir devoración de números (recuérdese lo dicho en el anterior capítulo) a la familia, ha aparecido el FPFC (coprocesador de coma flotante) MC68881.

Cortesía de Motorola, Inc.

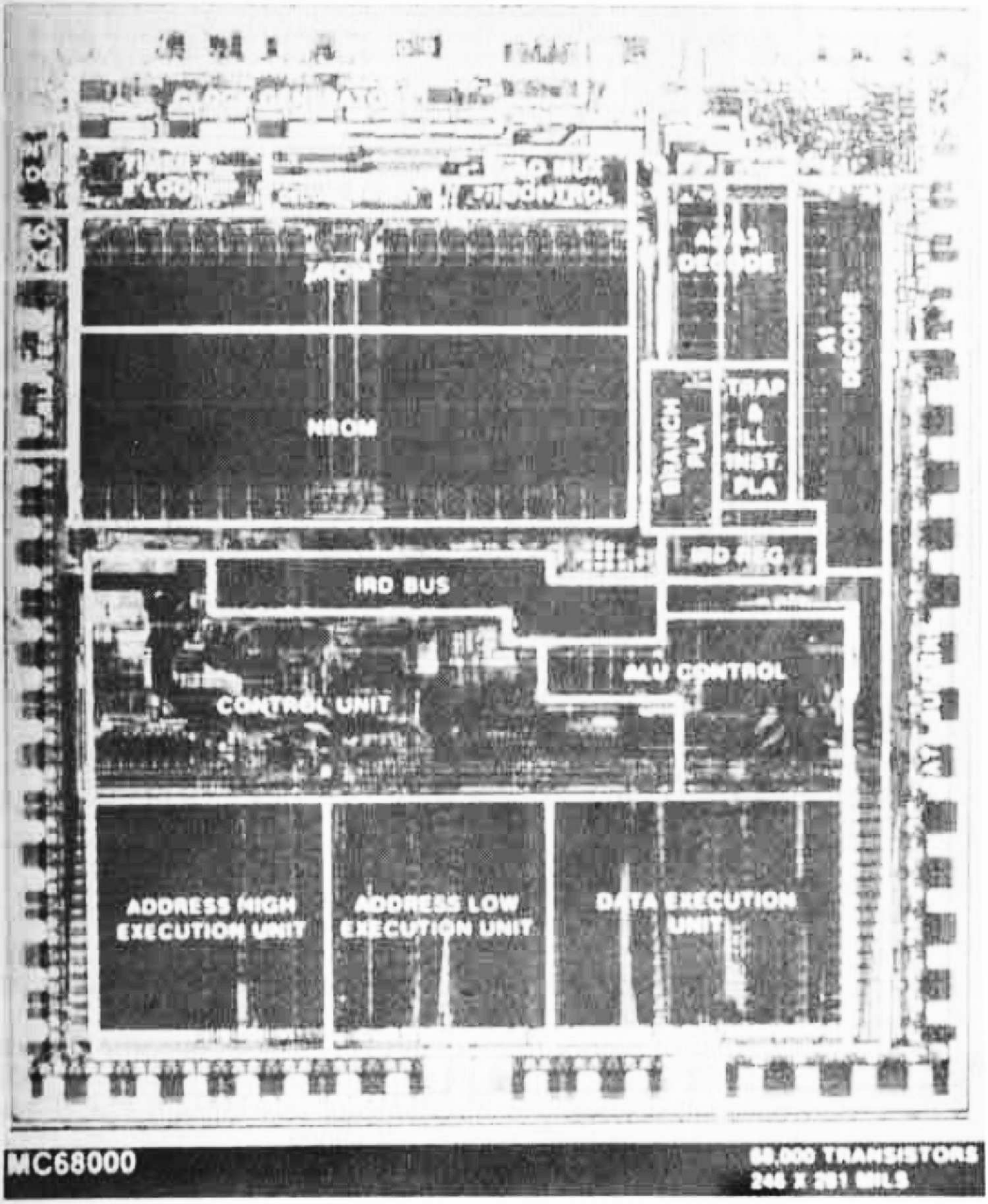
Figura 2.1
La familia 68000

Esperamos desvelar algunas de las características de diseño de la familia Motorola 68000 y señalar los puntos principales en que se tomaron las decisiones que llevaron a construir esta auténtica ruptura en cuanto a rendimiento, programabilidad y expectativas de crecimiento se refiere.

El *chip* del microprocesador es una de las más intrincadas realizaciones del hombre, con densidades VLSI (integración en muy alta escala) cercanas a los límites teóricos máximos. Hay muchas tecnologías VLSI distintas según las necesidades de densidad y potencia requeridas. Sin embargo, todas poseen el acrónimo común de MOS (dispositivos de estructura metal óxido semiconductor). Por ejemplo, el MC68020 ha sido construido mediante el proceso HCMOS (MOS, complemento de alta densidad) [1], alcanzando densidades de 200.000 transistores en un sustrato cuadrado de 3/8 de pulgada (véase la figura 2.3).

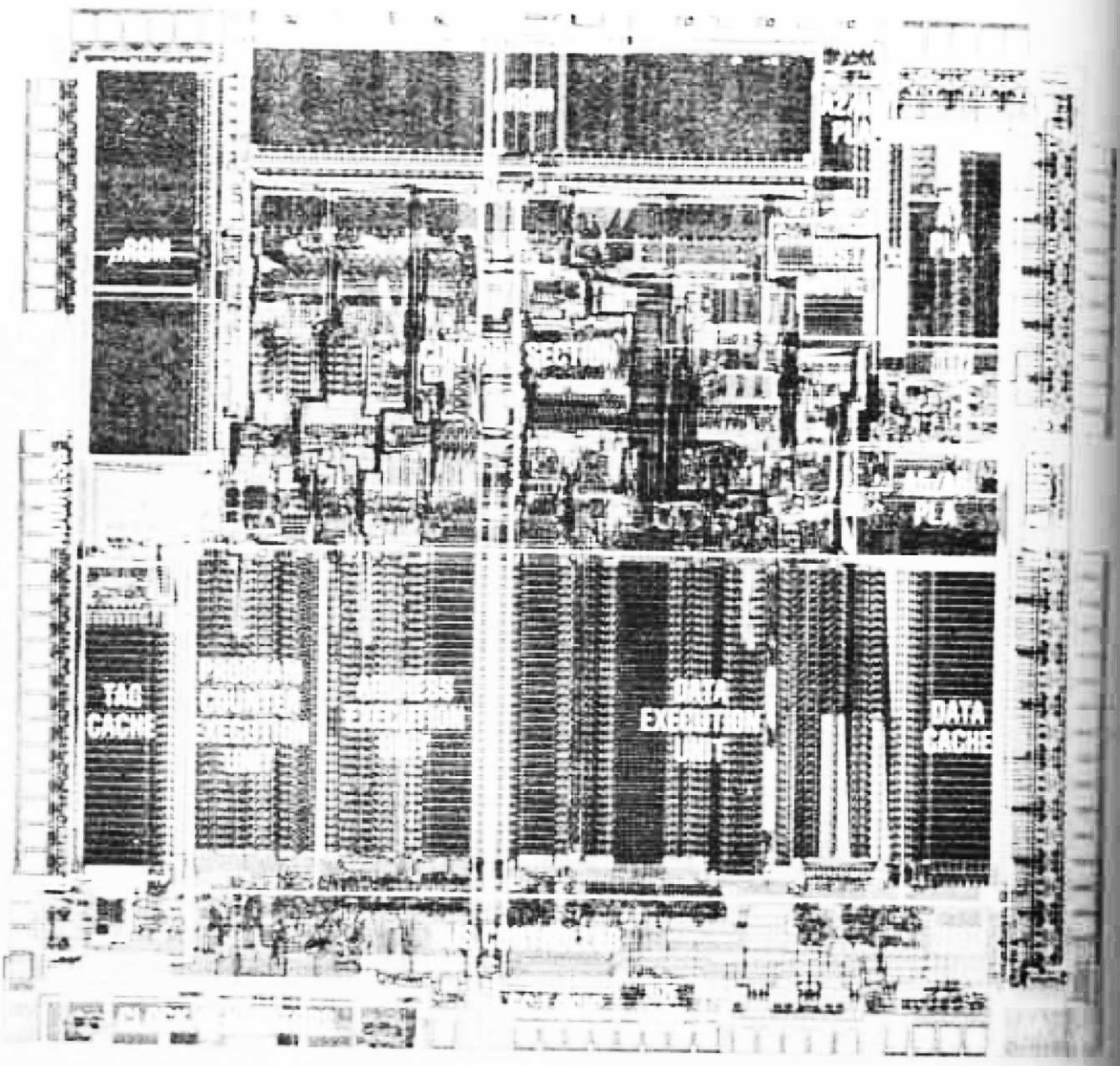
Desde luego, se halla fuera de los objetivos de este libro el hacer un es-

[1] Los conceptos que aquí se emplean para las distintas tecnologías de fabricación no son, en absoluto, ni sencillas ni fáciles de comprender. Para ello, recomendamos que el lector no demasiado experto en el tema consulte alguna de las obras dedicadas a su estudio, que pueden encontrarse en la mayoría de los catálogos de nuestras editoriales especializadas, con excelente nivel y calidad.



Cortesía de Motorola, Inc.

Figura 2.2
La unidad microprocesadora MC68000



Cortesía de Motorola, Inc.

Figura 2.3
MC68020

tudio detallado del vasto campo del diseño de los microcircuitos y de la microelectrónica en general.

Nuestro propósito, por tanto, consiste en ofrecerle un buen conocimiento de los aspectos más sobresalientes de esta notable familia, centrándonos en aquellos elementos directamente relacionados con el tema central del libro: el set de instrucciones del 68000.

Siguiendo la nomenclatura de Motorola, usaremos el convenio "68000" para designar oficialmente la familia en general, mientras que reservamos el prefijo "MC" para particularizar algún *chip* en concreto dentro de la familia total.

En la figura 2.4 se muestra un aspecto general de los miembros que constituyen la familia y un esquema de cómo deben interconectarse. En la práctica, raras veces se encuentra una configuración tan lógica. De hecho,

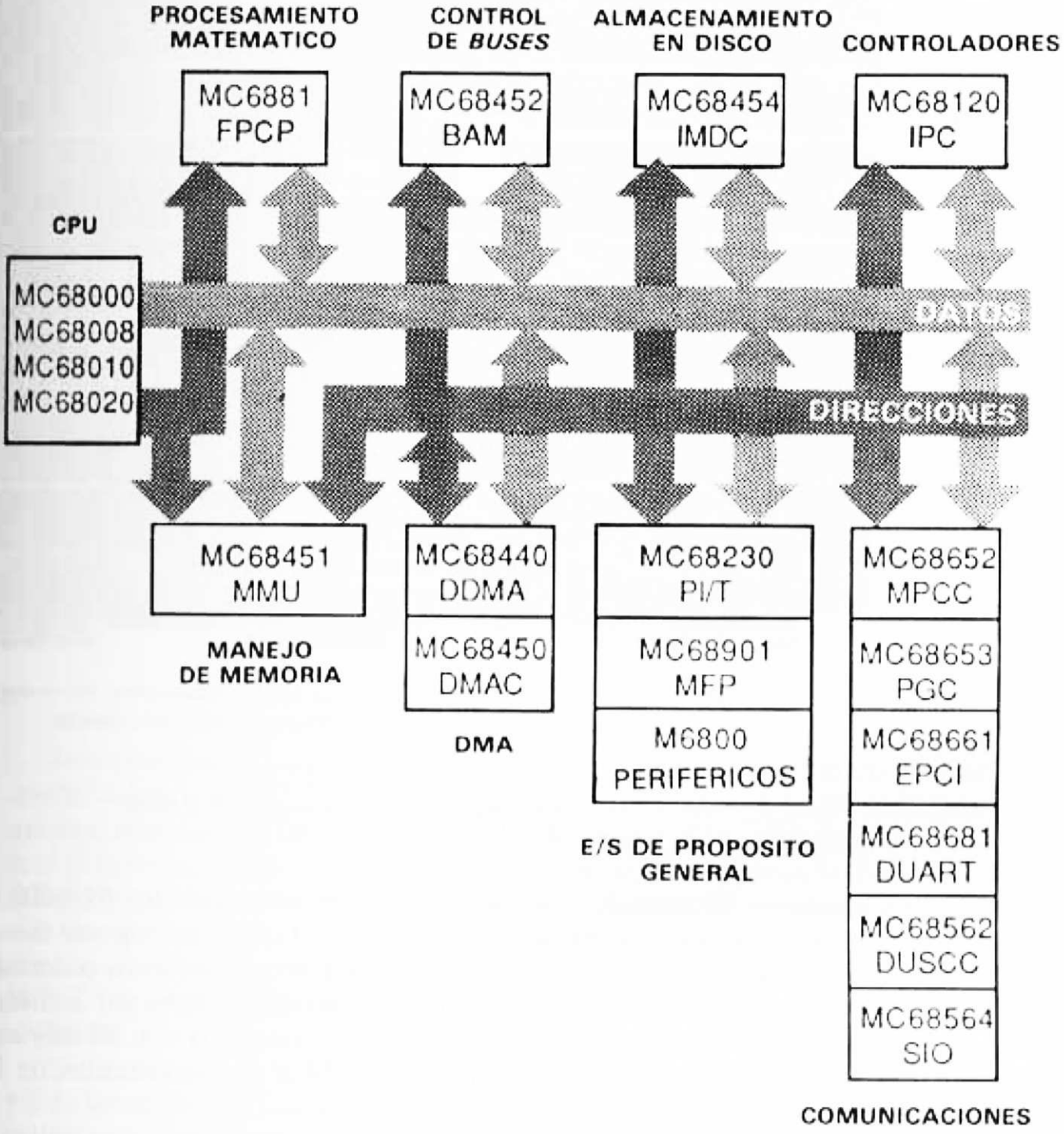
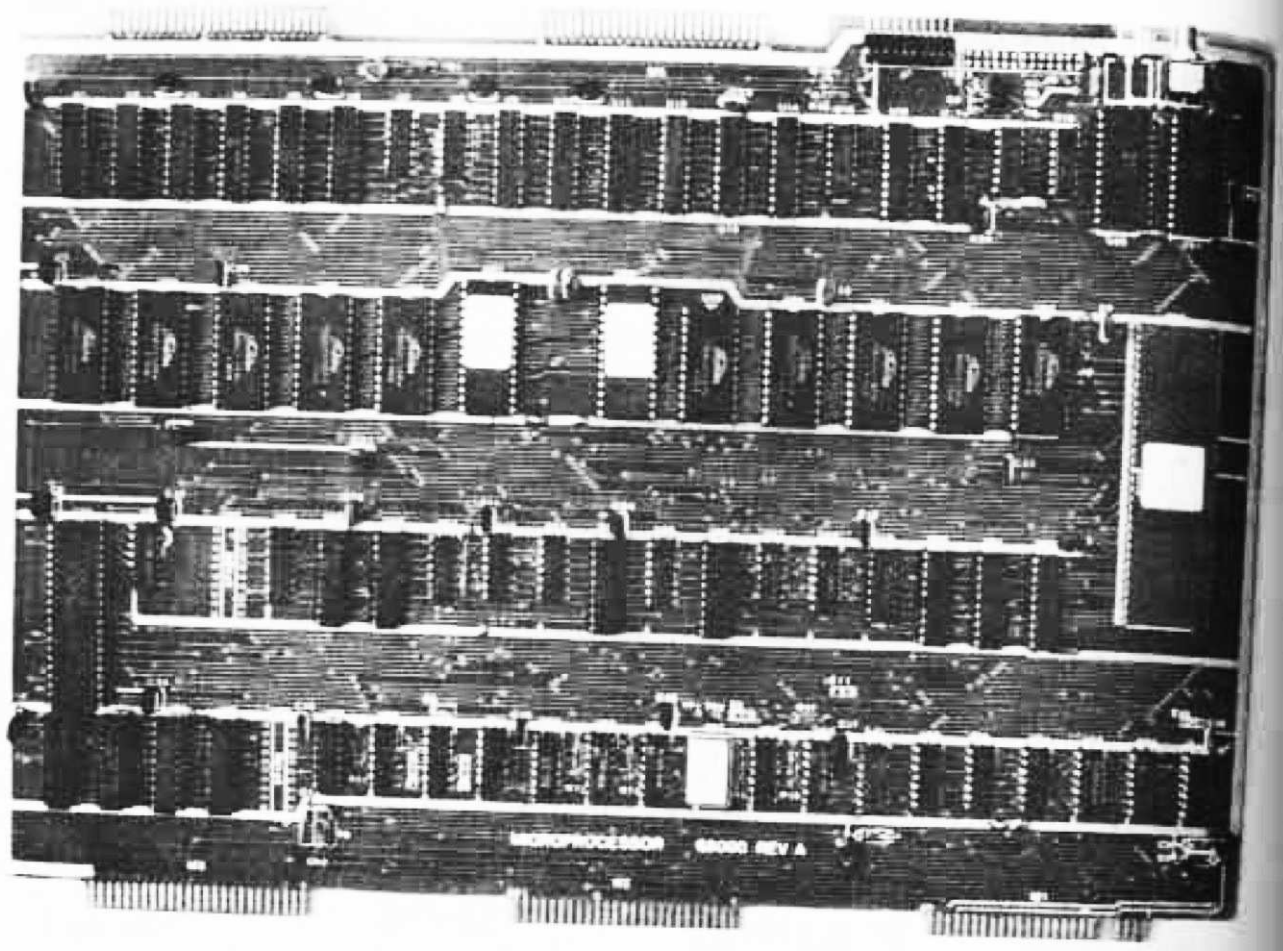


Figura 2.4
Familia M68000 y chips

principalmente gracias a los esfuerzos de la propia Motorola, un *bus* estándar conocido por "*bus* VME" ha merecido la aprobación internacional del ISO y del IEEE [2]. Su creciente importancia significa que muchos equipos de

[2] ISO son las siglas de la International Standard Organization, e IEEE lo son del Institute of Electrical and Electronics Engineers. Ambas, de la máxima categoría y prestigio mundiales.



Cortesía de EMS (Sistemas Educativos de Microordenadores —Educational Microcomputer Systems—), Irvine, California. Véase el apéndice E para más detalles.

Figura 2.5
Ordenador monoplaca basado en el 68000

distintos fabricantes pueden interconectarse con los M68000 y, recíprocamente, podremos encontrar muchos circuitos de soporte del *bus* VME en sistemas no Motorola. En la figura 2.5 puede verse un ordenador monoplaca educativo simple de la familia 68000 (fabricado por Educational Microcomputer Systems, Irvine, California) completo con 20 Kbytes de memoria RAM, 16 Kbytes de memoria EPROM y varios circuitos de E/S.

Historia del éxito del 68000

Hoy en día se precisa de bastante más que de una buena relación precio/rendimiento en el *hardware* para hacerse un sitio en el mercado. Los fabricantes de ordenadores y los diseñadores independientes de *software* necesitan, también, soporte básico de *software* con el que construir sistemas orientados al usuario y su *software* de aplicaciones. Con esta finalidad, Motorola fabricó herramientas de desarrollo *software*, tales como el EXORmacs™, EXORset™ y EXORciser™, así como componentes modulares para

la construcción de sistemas tales como los VMEmódulos™ y los VERSAmódulos™.

Todos ellos están soportados por un ordenador central con sistemas operativos desarrollados por Motorola, como el System V/68 derivado del UNIX, ayudas para traducción y comprobación, ensambladores y compiladores.

Además, Motorola estableció prontamente acuerdos de intercambio de patentes con los principales fabricantes de Estados Unidos, Japón y Europa (véase el apéndice E). Ahora se pueden comprar sistemas 68000 de varios fabricantes. Si no se dispone de estas segundas fuentes, los vendedores carecen de garantías y se convierte en poco menos que imposible la penetración en los mercados militar e industrial a ambos lados del océano.

El fruto de estos esfuerzos está en las innumerables implementaciones disponibles del 68000. En el apéndice E se relacionan muchos de los fabricantes que ya han adoptado estos circuitos. Van desde los gigantes de la industria, tales como IBM, Apple, Honeywell, NCR, ICL y Hewlett-Packard, hasta los menores, muy especializados, tales como AlphaMicro, Charles Rives, Alcyon y SBE, pasando por los fabricantes de equipos para consumo de masas, como Sinclair y Commodore. Probablemente no sea descabellado asegurar que el 68000 es el mejor, sino el más conocido en todo el mundo, sistema de microordenadores [3].

El diseño

Los circuitos VLSI (integración en muy alta escala) tienen un ciclo de diseño-a-fabricación tan largo y costoso que, como el avión del chiste, se quedan fácilmente obsoletos antes de despegar. Hay dos formas de evitar este peligro potencial: La primera consiste en procurarse la bola de cristal más puro posible y tratar de adivinar la evolución del mercado en los próximos dos o tres años. La segunda se basa en intentar reducir los periodos de diseño, automatizando los procesos de diseño y prueba. Como dijo Oscar Wilde, las predicciones son muy arriesgadas, sobre todo cuando se refieren al futuro.

Sin embargo, existe una subindustria dedicada al estudio del mercado y a la intención de futuro del mismo, que hace lo que puede en las situaciones más difíciles, asesorando no solamente sobre los tipos de productos que se necesitarán, sino también sobre los muchos impoderables asociados a los mismos, tales como el valor óptimo de la relación precio/rendimiento, reparto de mercado, patrones de consumo y cosas por el estilo. Las constantes fluctuaciones de exceso de stocks, escasez de oferta, paro y contratación en la industria del semiconductor subrayan el hecho de que la planificación en este campo está aún lejos de ser una ciencia exacta.

---

[3] En nuestro país, paradójicamente, parece que son otras "familias" las más populares. Sin embargo, esto es sólo el reflejo de una propaganda poco evolucionada, que no coincide con la opinión de numerosos especialistas.

A tiempo

Una de las principales causas del éxito del 68000 fue, sin lugar a dudas, el aparecer a tiempo. A mediados de los setenta, la bola de cristal predijo que los micros de 8 bits estarían pronto fuera de juego. De hecho, su gran éxito atrajo la atención hacia aplicaciones más complejas. Los usuarios, tanto comerciales como científicos, pedían procesamiento cada vez más rápido sobre bases de datos cada vez mayores, reservadas, anteriormente, a los macros y minis. Los avances en las memorias de masas de bajo coste (unidades de discos rígidos y flexibles) y de RAM apoyaron estos requerimientos de varias maneras. Se precisaba de sistemas operativos y lenguajes más elaborados para poder explotar las crecientes bases de datos, los cuales, a su vez, exigieron sets de instrucciones y de modos de direccionamiento más potentes.

Un nuevo mercado

El creciente mercado de los ordenadores personales empezó a exigir interfaces cada vez más "amigables" (ergódicas). Aunque el aficionado manitas era feliz con los diseños de los interfaces primitivos, el usuario doméstico y de empresa se preguntaba: "¿Qué pueden hacer por mí?" y "¿por qué son tan difíciles de usar estos aparatos?"

Un innegable hecho real es que el aislamiento del usuario de los bits y los bytes supone una enorme sobrecarga del *software*. Para salvar la falta de eficacia que supone la ergodicidad, se necesita más potencia pura de procesamiento y un espacio de direccionamiento de memoria más amplio, de la misma forma que una caja de cambio automática de coche precisa de un motor más potente para conseguir un rendimiento comparable al cambio manual.

Una de las razones de exigir sets de instrucciones más potentes con capacidades de direccionamiento más posibles fue el poder satisfacer las exigencias que planteaban los ingenieros del *software* de los sistemas profesionales en sus investigaciones.

El proyecto MACSS

Motorola se encontró con la situación descrita cuando arrancó el proyecto MACSS (sistema de ordenador avanzado en silicio de Motorola) a mediados de los setenta. Se trataba de un desafío habitual en la industria de los ordenadores, cómo reconciliar los dos objetivos fundamentales autoexcluyentes: crear una nueva generación a la vez que se mantiene la compatibilidad con el *software* y los periféricos actuales.

Ya no estaba bien visto el invocar al dios romano Júpiter, al que, según

se dice, adoraban propios y extraños. En el tumultuoso mundo real, el grupo del MACSS adquirió difíciles compromisos entre los aspectos técnicos y los de mercado, tomando realmente miles de decisiones, unas forzadas por hechos reales y las otras apoyadas en la falible intuición.

La existencia de numerosos usuarios supone una sustancial inversión, tanto en *hardware* como en *software*, que no puede ser ignorada a la ligera, aunque ello podría significar al mismo tiempo la imposibilidad de conseguir un total aprovechamiento de los avances en la tecnología del silicio, las arquitecturas de ordenador y los métodos de programación.

Los adelantos en estado sólido han continuado deslumbrándonos desde la aparición del primer transistor a finales de los cuarenta en los laboratorios de la Bell. Siempre hay cientos de adelantos en el proceso de fabricación de los IC (circuitos integrados) en alguno de los siguientes aspectos: Predicciones teóricas de los físicos de estado sólido, modelos matemáticos y simulaciones, pruebas en pequeña escala en I + D (investigación y desarrollo), esquemas piloto para el establecimiento de los métodos de control de calidad y estudios de determinación de la factibilidad económica de la producción en gran escala. Un aspecto crucial para completar el ciclo es la predicción de la demanda del mercado y de la vida útil del producto. Ninguna otra industria se enfrenta, como ésta, a tan delicadas ecuaciones que relacionan costes de creación, volúmenes de producción y coste unitario.

La herencia

La diversidad de CPU aparecidas entre la pionera Intel 4004 (1969 a 1971 y el proyecto MACSS (1976 a 1979) pone de manifiesto la variedad de posibles respuestas. La evolución histórica ha sido "un salto de mata" de los principales involucrados: Intel, Motorola y Zilog.

En 1979, los microprocesadores dominantes eran la Intel 8080/8085, el Zilog Z80 y el Motorola 6800 [4], todos ellos de 8 bits y no demasiado distintos en cuanto a arquitectura global y rendimiento. Aprovechando las facilidades de las técnicas de integración en mayores densidades, tales como el proceso HMOS (metal óxido de gran densidad), los tres evolucionaron hacia los diseños de 16 bits. Intel y Zilog mantuvieron la compatibilidad del código máquina con la gran cantidad de 8 bits en funcionamiento, repitiendo la filosofía inicial, tal como muestra, por ejemplo, la secuencia Intel de 8008 a 8080, de 8080A a 8085; cada uno de ellos, un hermano del anterior pero más rápido. Al aparecer los Intel de 16 bits 8088/8086, se mantuvieron los signos de esta tradición.

---

[4] En honor a la verdad, el micro de mayor éxito ha sido, sin lugar a dudas, el Zilog Z80, a salvo de la popularidad inicial del 8080, debida, principalmente, a su pronta aparición en el mercado. El Motorola 6800, con ser el más prometedor, tuvo su principal virtud en sus "sucesores", el 6809 y, principalmente, el 68000.

Romper con el pasado

El equipo del MACSS tomó la gran decisión de romper con el pasado y crear, como si no hubiera existido, el mejor diseño posible de 16 bits. La única concesión a los usuarios del 6800 de 8 bits fue la dotación de circuitos de reloj para poder manejar los lentos periféricos síncronos del MC6800. Así, aunque el *software* tuvo que ser desarrollado partiendo de cero, había por lo menos una amplia gama de dispositivos de E/S y de soporte ya en funcionamiento.

Esta rotura con el pasado fue una jugada muy arriesgada, pero puso de manifiesto la realidad en la que los micros de 8 bits habían evolucionado a su aire, durante una época en que las necesidades de los programadores colocaban en un segundo plano los requerimientos de los diseñadores de *hardware*. Cuando el primer Intel 4004 de 4 bits engendró el 8088 a principios de los setenta, a este último se le añadieron muchas de las características de una calculadora o de un circuito de control de pantalla de vídeo. Esto no es, en absoluto, una crítica hacia aquellos nobles pioneros. El mismísimo Júpiter no hubiera sido capaz de predecir la explosión de las aplicaciones de los microprocesadores que iba a ocurrir en la siguiente década.

Durante esta época, una vez que el set de instrucciones quebaba establecido, se desarrollaba un importante cuerpo de *software* a su alrededor, de forma que las subsecuentes mejoras en las CPU estaban dominadas por la idea de la compatibilidad de los programas —el comprensible deseo de que la nueva máquina pudiera ponerse inmediatamente a trabajar.

La lección era muy clara: El nuevo diseño del Motorola de 16 bits debería ser de tal forma que las futuras mejoras conservaran la compatibilidad del *software* sin sacrificar el *hardware*. Al final, el equipo MACSS ha demostrado tener razón.

TABLA 2.1

La familia de los microprocesadores M68000

Modelo	*MC68008*	*MC68000*	*MC68010*	*MC68012*	*MC68020*
Tecnología	HMOS	HMOS	HMOS	HMOS	HCMOS
Terminales	64 DIP	64 DIP	64 DIP/68 QP	84 GA	114 SPG
Relojes (MHz)	4-12,5	4-12,5	4-12,5	4-12,5	16,67
Núm. registros	17	17	20	20	23
Longitud de instrucciones (palabras)	1 a 5	1 a 5	1 a 5	1 a 5	1 a 7
Tamaño de los registros (bits)	32	32	32	32	32
Tamaño de ALU	16	16	16	16	32
Bus de datos	8	16	16	16	8/16/32
Bus direcciones	20	24	24	31	32
Capacidad de direccionamiento	1Mb	16Mb	16Mb	2Gb	4Gb

¿Por qué 16 bits?

Tal como vimos en el capítulo 1, los números empleados para identificar un micro pueden conducir a errores. Como muestra la tabla 2.1, incluso entre los miembros de una familia, la estadística vital varia mucho.

Todos los parámetros expuestos están astutamente interrelacionados, y todos tienen su particular significado en el resultado final de la relación precio/rendimiento del circuito. Pero, si hubiera que decidirse por un solo parámetro, el programador elegiría, probablemente, la longitud de la palabra de instrucción. Esta impone la riqueza y potencia del set de instrucciones, y esto es de la máxima importancia, cara a las consecuencias, para el programador.

El ancho del *bus* de datos, por ejemplo, determina el número de ciclos de lectura/escritura necesarios para acceder a los datos. Pero si la velocidad es la suficiente, ¿qué le importa a usted que el *bus* sea de 1 o de 1.000 bits? Desde luego, eso importa al ingeniero y al responsable de ventas (¿cómo puede venderse una máquina de 1 bit?), pero el programador estará contento con tal de que los registros sean razonablemente capaces y grandes.

El tamaño del *bus* de direcciones, como hemos visto ya, determina el máximo espacio direccionable, y desde luego que todos deseábamos liberarnos de la tiranía de los 64 Kbytes impuestos por los 16 bits antiguos. Existen varios métodos de manejo de memoria que pueden resolver este problema.

La importancia del tamaño de la palabra de instrucción es clara si observamos un código típico de 8 bits, como el mostrado en la figura 2.6. Aunque, a primera vista, es capaz de generar hasta 256 códigos diferentes, una vez que se asignan unos bits para determinar el modo de direccionamiento (para incluir en código de operación la memoria o registro sobre los que se

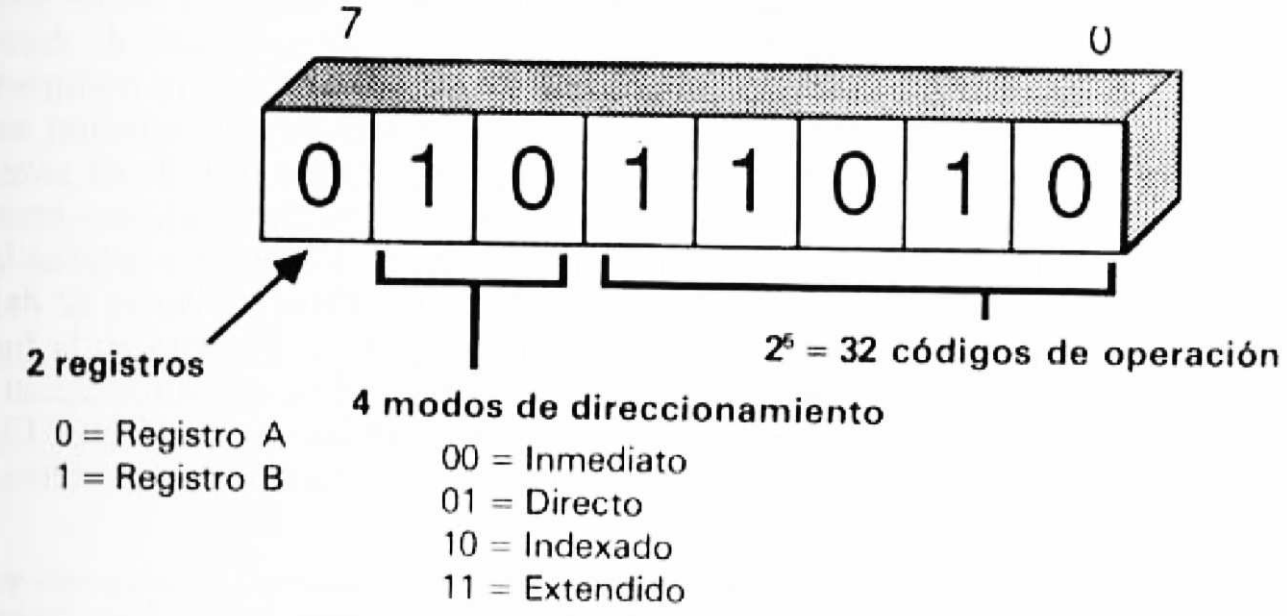


Figura 2.6
Instrucciones de 8 bits

debe actuar), el total se reduce a 32 códigos de operación diferentes. Ahora, el conflicto está entre el número y tipo de registros, el número de modos de direccionamiento y el número de códigos de operación. Ocho bits es demasiado restrictivo.

El salto dado por Motorola a los 16 bits de la palabra de instrucción cambió el panorama por completo, aunque a costa de una mayor complejidad de los circuitos.

Capacidad de los *chips*

Los diseñadores hablan de la capacidad de los *chips*, queriendo significar cuantas decisiones sobre su capacidad de funcionamiento pueden hacerse. Para cada tecnología en concreto (sea NMOS, HMOS, etc.), con la que se consigue una determinada densidad de componentes y superficie del *chip*, queda establecido el máximo número de puertas lógicas implementables. Esto, a su vez, establece el límite a las características y funciones que pueden implementarse en él, y define aquellas que deberán ser realizadas con circuitos externos de soporte.

Número de terminales

Las funciones escogidas deberán, obviamente, revertir al exterior, por lo que el diseño físico (el "empaquetamiento") debe hacerse con todo esmero. Al mismo tiempo que la existencia de los diseños de 8 bits provocaba dudas sobre el posible progreso, el número de terminales de los *microchips* existentes entonces sumaba otro obstáculo. La disposición estándar de 40 *pines* tenía la ventaja del bajo coste de fabricación y prueba —y había grandes cantidades de zócalos de 40 terminales dispuestos a recibir estos circuitos—, pero ello limitaba el número de líneas de los *buses* de datos, dirección y control. A menudo, los diseñadores optaban por la multiplexación temporal o el reparto de los caminos, pero esto es una solución autodestructiva. La solución MACSS consistió en una disposición de 64 *pines* en DIP (disposición en doble fila, una a cada lado del circuito), que permitió mayor libertad a la hora de escoger el tamaño de los *buses* y eliminó la necesidad de la multiplexación o el reparto de líneas entre los *buses* de datos y de direcciones [5]. La disposición de los terminales se muestra en la figura 2.7.

Volviendo al tamaño de las instrucciones, se establecieron vías de comunicación entre los *buffers* y los decodificadores de 16 bits. En ellos, se pueden colocar hasta 65.536 instrucciones diferentes —una tarea nada trivial si

[5] Esto es, desde luego, el punto de vista de Motorola. En este punto queremos animar al lector a que consulte obras dedicadas a los sistemas Intel o Zilog. Su trascendencia en la práctica ha demostrado que no son tan deficientes como aquí se sugiere. La multiplexación de las líneas supone un ahorro de pistas de comunicación y evitación de ruidos muy respetables.

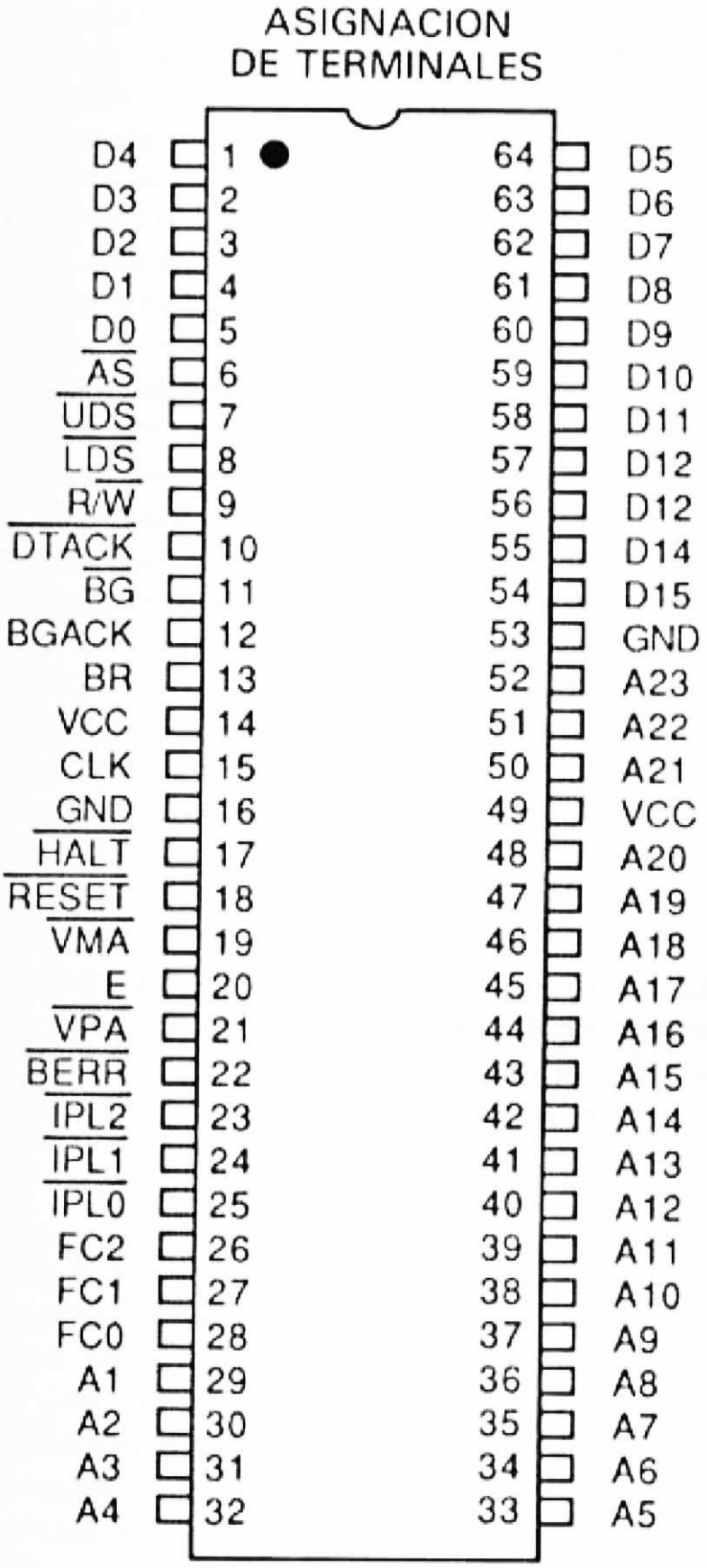


Figura 2.7
Terminales del MC68000

la comparamos con la situación de los 8 bits, en que lo códigos y los decodificadores se definían casi por sí solos.

Los registros: Tipo y número

Ahora, ya disponemos de suficiente espacio para implementar un conjunto simétrico de órdenes, esto es, para disponer de códigos de operación capaces de trabajar de la misma forma sobre un buen número de registros de propósito general. Con 8 bits, la única forma de incrementar el número de registros es haciéndolos de propósito especial, **dedicados.** Por ejemplo, si la instrucción de suma, ADD, trabaja sólo con el registro A, no es necesario incluir la identificación de A en la codificación de la operación. Como veremos, los registros simétricos son mucho más fáciles de utilizar siguiendo la filosofía de programación del 68000.

El paso fundamental estuvo en decidir que todos los registros tendrían el mimo tamaño de 32 bits, incluso en aquellos casos, como el contador de programa o el puntero de pila, en que con 24 hubiera sido suficiente para los fines inmediatos que se perseguían.

Motorola estableció un compromiso razonable entre el número y el tipo de registros, decidiendo un total de 16 registros básicos programables, 8 para operaciones generales de datos y 8 para operaciones generales de direccionamiento. En algunas instrucciones, se precisaron 3 bits para la codificación del registro (en concreto, para las instrucciones de tipo implícito), mientras que se necesitaron 4 para el caso más general en que deberán poder trabajar con todos los registros. Si ahora lo comparamos con el 6800, que sólo tiene dos registros generales y un índice, podemos apreciar la buena noticia que supuso el nuevo sistema.

Datos

La palabra de instrucción de 16 bits permite también asignar algunos bits a la selección del tamaño del operando. A pesar de que los registros tienen 32 bits y el *bus* 16 (en el caso del 68000), Motorola quiso ofrecer al programador una forma sencilla y uniforme de manipular datos en forma de bytes de 8 bits, palabras de 16 y dobles palabras de 32. Para tal misión, se destinan dos bits en todas las instrucciones, con los que se determina qué tipo de dato va a manejarse. A nivel de lenguaje ensamblador, basta con un único código (B, W o L) para efectuar esta misma especificación.

Modos de direccionamiento

En la misma línea que las instrucciones, el número y capacidad de los modos de direccionamiento han pasado de 4 en el MC6800 a 14, o más, en el 68000. Su importancia se pondrá de manifiesto en los capítulos del 4 al 8.

Implementación

La última decisión de un diseño, después de haber tratado todos los datos importantes —número y tamaño de los registros, capacidad de direccionamiento, códigos de instrucción y modos de direccionamiento—, es la forma en que va a ser implementado en silicio. Hay dos caminos para disponer las complejas conexiones y los pasos lógicos necesarios para poner el *chip* en marcha.

Lógica *random*

El método tradicional, llamado **lógica *random*** [6], requiere la especificación completa y con todo detalle de las tareas a realizar, para plasmarse en la red a construir con elementos lógicos discretos que hará lo indicado. Este método lleva a un diseño de *chip* económico, en el que no se malgasta su capacidad. Sin embargo, a medida que los circuitos VLSI aumentan su complejidad, va siendo más y más difícil esta solución. La lógica *random* es, sencillamente, demasiado rígida. La alternativa, inventada de hecho por Maurice Wilkes en los primeros días del EDSAC (Cambridge, 1949-57), se denomina **microprogramación**, nombre que en aquellos días no se prestaba a confusión tal como hoy ocurre [7].

Microprogramación

Motorola adoptó la microprogramación para la realización del núcleo del 68000. Con microprogramación, se tiene algo parecido a una CPU dentro de nuestra CPU. Cada instrucción es subdividida en subinstrucciones, o si se prefiere microinstrucciones, de la misma forma que un programa exterior (un "macro") se reduce a instrucciones convencionales y subrutinas.

Por ejemplo, si se describen todos los detalles de la ejecución de una instrucción de movimiento de datos entre dos registros y, después, se hace la misma operación con la suma de esos registros, se podrá observar que hay muchas "microrrutinas" comunes. Estas subrutinas, y todas las demás necesarias, constituyen el microprograma, que está almacenado en una microROM interior.

Hay un microsecuenciador que encamina el flujo de datos y señales de

---

[6] En realidad, el adjetivo *random* es una acepción inglesa que significa "aleatorio". Aquí, tanto en castellano como en inglés, lo que quiere dar a entender es que depende de los deseos del diseñador que, por tanto, son imprevisibles (de ahí lo de aleatorio). De todas maneras, se trata de una acepción absolutamente general.

[7] Evidentemente, microprogramar no es programar un microprocesador en el sentido habitual de escribir un programa (BASIC, Ensamblador, etc.) para un microprocesador, sino que se trata de realizar el decodificador de instrucciones que la CPU debe llevar en su interior para poder interpretar el significado de las instrucciones del programa a medida que las va leyendo en la memoria.

control para cada instrucción de acuerdo con el microprograma. Ello supone una enorme ventaja en cuanto a flexibilidad de diseño, prueba y ajuste a las necesidades reales, aunque sea a costa de la capacidad del *chip*. La microprogramación puede ser implementada y comprobada antes de embarcarse en la costosa fabricación del *chip*.

Conclusión

Detrás del microcódigo, el 68000 emplea la nanoprogramación —y así hasta infinito—. Su estudio nos llevaría más allá de nuestros objetivos, que han sido describir algunos datos necesarios para su apetito de programador, relativos a las decisiones que condujeron a la realización del 68000, para poder llegar al producto final. En el capítulo 3 nos ocuparemos de las características de funcionamiento accesibles al programador.

3

Modelos de programación del 68000

"Tus dones, tus características, están en mi cerebro totalmente grabados con memoria permanente..."

(SHAKESPEARE, Soneto CXII)

En el capítulo 1 nos ocupamos de varios conceptos básicos comunes a todos los microprocesadores, y en el capítulo 2 analizamos algunas de las decisiones relativas al diseño y fabricación enfrentadas por Motorola al abandonar la gama 6800 de 8 bits para saltar al nuevo rango de los 16 y 32 bits de la familia 68000. Tal como vimos, el principal objetivo de Motorola fue la facilidad de programación. Por ello, veremos el 68000 desde el punto de vista del programador.

En primer lugar, deberemos identificar con precisión las formas de clasificación de los lenguajes de programación, los distintos papeles que cada uno juega y de qué manera pueden ser empleados por el programador según sus conocimientos del microprocesador.

Niveles de programación

El término *programación* cubre un amplio rango de actividades que precisan de muy diferentes conocimientos y apreciaciones. Como se vio en el

capítulo 1, todos los programas acaban, antes o después, en un listado de **instrucciones** extraídas de la memoria del procesador (RAM o ROM) y traducidas en acciones concretas, tal como el acceso y manipulación de datos de otras zonas de memoria.

Hay programadores, y programas escritos por ellos, de todos los colores y tamaños. Se puede programar para ganarse el pan de cada día, para salvar al mundo libre o por el simple placer de hacerlo. El nivel de detalle necesario sobre la forma interna concreta de trabajar el procesador, tal como el mecanismo para acceder a los datos y programas de la memoria, depende muy fuertemente de los **niveles** de los lenguajes de programación utilizados. La figura 3.1 ilustra algunas de las principales categorías de *software* con el fin de darle una idea sobre lo que significa, para nosotros, el *nivel* de programación.

Lenguajes de alto nivel

El programador con lenguajes de alto nivel, tales como BASIC, FORTRAN o Pascal, escribe un programa "fuente" [1] que la máquina es incapaz de reconocer sin algunas transformaciones previas. El programador está virtualmente aislado, tanto del procesador como de su arquitectura y su soporte de memoria.

El programador de alto nivel puede centrarse en resolver sus problemas de aplicación en el lenguaje elegido, dejando al **compilador** o al **intérprete** la tarea de convertir el lenguaje fuente en instrucciones de **lenguaje máquina** que el procesador es capaz de comprender. Estas instrucciones en lenguaje máquina son series inescrutables de unos y ceros que sustituyen a aquellas de "lenguaje inglés" [2], que son una forma mucho más natural y genérica de expresar nuestro problema.

La salida de un compilador es una versión **compilada** del programa fuente comprensible ya por el programador, lo que se conoce con los nombres de programas **objeto, ejecutable** o listo para **correr**, con lo que se quiere dar a entender que no precisa de más traducciones para que pueda ser ejecutado. Como puede verse en la figura 3.1, los módulos de código máquina [3] creados por el compilador se almacenan normalmente en discos, o cualquier otro mecanismo de almacenamiento masivo, del que podrán ser llevados a

---

[1] Aunque a lo largo del texto se deduce su sentido, queremos anticipar que por programa "fuente" se entiende aquel que está expresado de acuerdo con ciertas reglas, pero siempre mediante frases y/o órdenes de un lenguaje comprensible para las personas, y que es el que, precisamente, utilizan los programadores (por regla general) para hacer sus programas.

[2] Evidentemente, nuestro lenguaje más natural es el castellano, pero como la totalidad de los programadores debemos, por imprescindible, saber utilizar el inglés (incluso como medio de construcción de los programas). Hemos querido respetar la versión inglesa original.

[3] Por regla general, se sobrentiende que el código máquina es aquel que la CPU entiende directamente. Así mismo, se denomina código objeto a la versión en código máquina de un programa escrito, inicialmente, en lenguaje de nivel alto o medio.

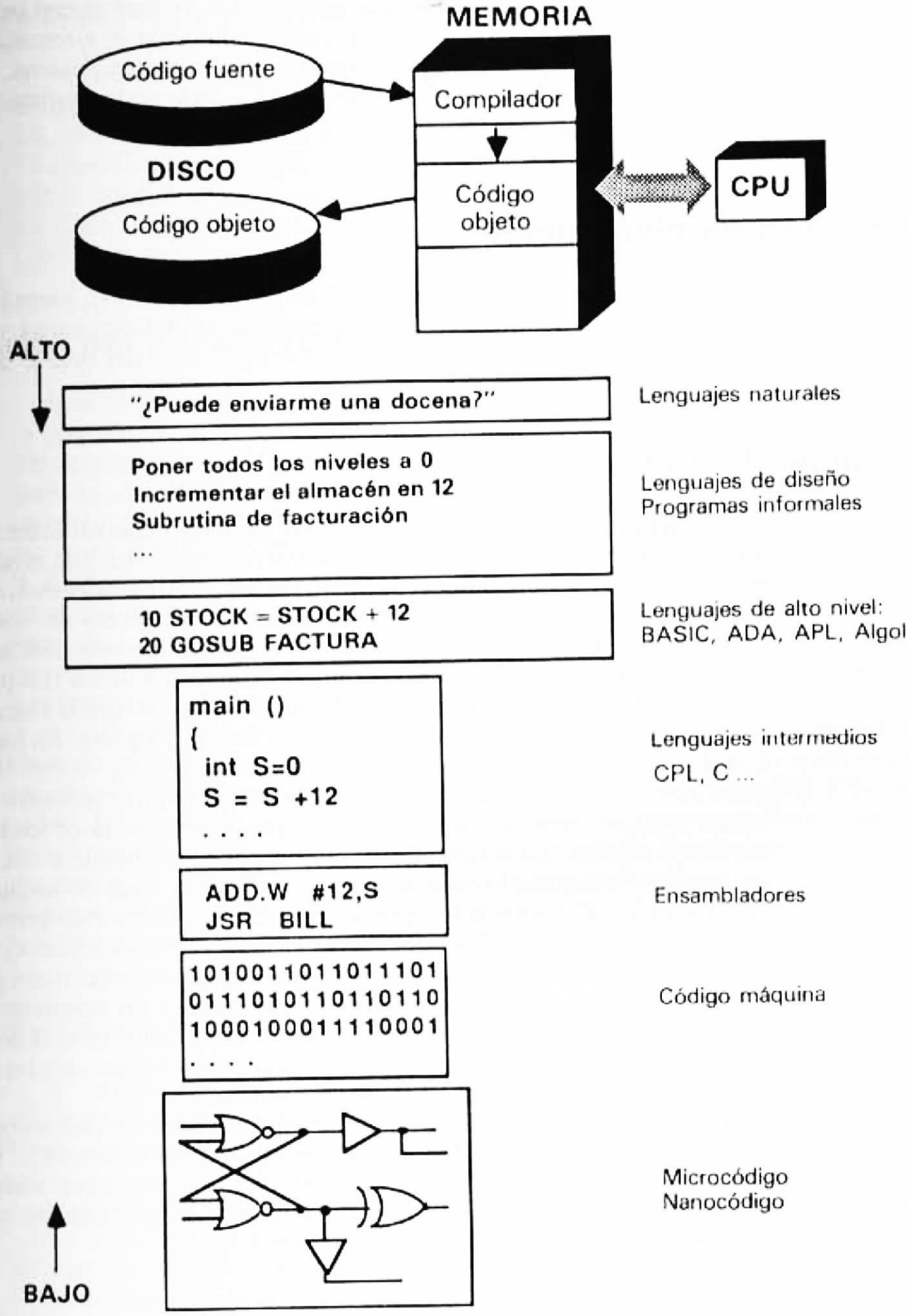


Figura 3.1
Niveles de lenguajes de programación

memoria de trabajo cuando sea preciso. La diferencia con los intérpretes consiste en que en éstos cada instrucción traducida es ejecutada inmediatamente. Para nosotros, estas diferencias no son significativas, por lo que a partir de ahora los englobaremos a todos bajo el denominador común de compiladores.

Lenguajes de nivel medio

Un lenguaje de nivel medio, tal como FORTH o C, necesita, como los anteriores, de un compilador, pero permite un mayor control sobre el procesador por parte del programador que en el caso del BASIC o del Pascal.

Lenguaje de bajo nivel

Un **lenguaje ensamblador** ofrece un detallado control sobre la forma en que el procesador ejecutará nuestras instrucciones. En este nivel de programación se escriben realmente las instrucciones especificas de la máquina en una versión sencilla, simbólica, de **mnemónicos** fáciles de recordar, tales como ADD, MOVE y SUB [4]. El apéndice E contiene una relación alfabética de los mismos, cuya forma de actuar es el objetivo de los tres próximos capítulos. Por ahora, lo único que debemos saber es que el lenguaje ensamblador ofrece un camino fácil de leer el código máquina. En lugar de tener que escribir 0010101100110000, o algo parecido, en código binario para cada instrucción, los ensambladores permiten emplear símbolos más practicables y significativos. Hay que notar que, a diferencia de los lenguajes de los otros niveles, cada linea de ensamblador corresponde a una instrucción en lenguaje máquina. Cuando usted escribe una linea en lenguaje BASIC, por ejemplo, el compilador puede generar 50, 100 o más instrucciones en código máquina. Para los programas en lenguaje ensamblador, en lugar de emplear un complicado compilador, nexo de unión entre usted y el *chip*, se usa un sencillo y rápido **ensamblador**, que traduce los mnemónicos en códigos máquina. Así, los ensambladores son compiladores sencillos —dan un único salto de nivel en su conversión. Son más sencillos que los compiladores, porque el salto de nivel es mucho menor.

Para trabajar a nivel de lenguaje ensamblador es preciso conocer bien en qué forma el procesador busca y maneja cada instrucción. Pero, como pronto veremos, las ventajas son grandes: los programas escritos en lenguaje ensamblador son compactos, eficaces y super rápidos —¡y no tan difíciles como pudiera parecer!

---

[4] Aquí queremos recordar lo dicho anteriormente sobre esta terminología: merece la pena hacer el pequeño esfuerzo de recordarlos en el inglés original en que vienen **todos** los ensambladores.

Nivel de microcódigo y de nanocódigo

Dentro del propio nivel de máquina, hay instrucciones escritas en **microcódigo** de forma permanente para interpretación del anterior y, aún más allá, el 68000 dispone de un **nanocódigo** para interpretar el microcódigo. Ya apuntábamos en el capítulo 2 que estos esotéricos niveles están fuera del interés normal de un programador, pero nosotros nos beneficiamos de la tremenda flexibilidad que suponen en las etapas de diseño y prueba del *chip*. Todo nivel tiene ventajas e inconvenientes para el programador.

Comparación entre los niveles de lenguaje

Una de las razones del éxito de los lenguajes de nivel alto y medio radica en el hecho de que los programas pueden escribirse de forma que corran con configuraciones distintas del mismo procesador o, incluso, en procesadores totalmente distintos. Para este último caso, todo lo que se necesita es un compilador adecuado al *chip* de que se trate. Por ejemplo, un programa escrito en el MBASIC™ de Microsoft® funcionará en un Apple, un Radio Shack™, un IBM y cientos de otros sistemas basados en muchos procesadores distintos, con posibles diferencias sin límite en cuanto a tamaño de su memoria y estructura. Por supuesto, este feliz estado de cosas se debe al esfuerzo de Microsoft para crear las diferentes versiones de compiladores e intérpretes de MBASIC para todas esas máquinas.

Al *software* que puede funcionar con pocos cambios, o sin ninguno en absoluto, en distintos sistemas, se le llama **portátil** y, con vista a los constantemente crecientes costes de la programación, es la propiedad más deseable. Sin embargo, como suele decirse, nadie da nada por nada, hay un precio que pagar por la "portabilidad". De la misma forma, hay que pagar un precio por la mayor productividad de programación que suponen los lenguajes de alto nivel: El precio es la menor velocidad y eficacia, así como el mayor tamaño de los programas resultantes [5].

Velocidad, tamaño y eficacia

A medida que uno desciende en la jerarquía de los lenguajes hacia los ensambladores y la máquina, los programas van siendo menos portátiles y más dependientes del *chip*. Por otra parte, los niveles más bajos le permiten un control más directo de las posibilidades del *chip*, pudiéndose aprovechar las características del mismo de formas no accesibles desde niveles superiores. En resumidas cuentas, los lenguajes de bajo nivel son más difíciles de utilizar y menos portátiles, pero son mucho más compactos y corren mucho más deprisa. Con programas típicos de pruebas, los llama-

[5] El autor se refiere aquí al programa máquina resultante de la compilación, frente al directo, obtenido por programación en ensamblador.

dos **bancos de pruebas** *(benchmarks)*, un programa escrito en ensamblador resulta cerca de unas 10.000 veces más rápido que su versión en BASIC, ocupando solamente el 1 por 100 de la memoria. Desde luego, hay muchos factores que determinan el rendimiento de un ordenador, y no es significativo acelerar una etapa de la secuencia de funcionamiento de un ordenador, a menos que el resto siga los mismos derroteros.

Los propios compiladores son también programas, de forma que quienes los escriben deben conocer muy profundamente los procesadores específicos a nivel de lenguaje máquina. Su principal objetivo es la **optimización**, para conseguir reducir la sobrecarga que supone la traducción desde alto hasta bajo nivel, y producir el código más eficaz. Por ello, muchos compiladores están escritos en lenguaje ensamblador. Otra forma consiste en escribirlos en lenguaje C, traducirlos a lenguaje ensamblador y tratarlos para conseguir la mejor versión ejecutable posible. En otras áreas de desarrollo *software* se emplean técnicas similares, especialmente en el caso de los sistemas operativos, utilidades y en todo el campo de lo que se conoce por sistemas *software*. Tales programas, empleados en dar soporte a todos los usuarios y grabados en la ROM muy a menudo, deben ser, evidentemente, los más compactos y eficaces posible.

Una regla general, a menudo comprobada, es que merece la pena escribir en el nivel más bajo posible los programas de uso frecuente, y ese nivel suele ser el ensamblador. La mayor parte de los actuales lenguajes de alto nivel permite saltar o **encadenar** a rutinas frecuentemente usadas, aunque estén escritas en lenguaje ensamblador. A esta posibilidad se la podría llamar como la de "lo mejor de ambos mundos". Por esta razón, hoy día se suele exigir que los programadores conozcan varios lenguajes de alto nivel y, por lo menos, una versión del ensamblador. Como en los automóviles, uno puede pasar de una marca a otra, de forma que los mandos pueden estar dispuestos de distinta manera, pero los principios de conducción son los mismos. Continuando con el símil, el lenguaje ensamblador sería algo así como deslizarse por detrás del volante de un Ferrari. Hay muchas técnicas nuevas que conocer antes de que usted pueda alcanzar su máxima potencia, pero caerán poco a poco, y entonces ¡zas!, estará conduciendo con auténtico estilo.

Seguridad

Merece la pena mencionar brevemente (porque puede aclarar ciertos aspectos de los niveles de los lenguajes) el hecho de que los editores de *software* tratan de proteger sus programas fuente contra la piratería y los accesos ilegales. En muchos casos, el público puede adquirir solamente la versión en máquina de un programa, y aun ésta está protegida contra el uso no autorizado y la copia de muy distintas formas, que van desde las severas advertencias por escrito hasta el empleo de sofisticados medios criptográficos. Hay varios paquetes *software* bien conocidos que pueden adquirirse por 500 dólares, pero cuyo programa fuente vale cinco millones. La clave del

asunto está en que el programa fuente desvela la estrategia de la programación y que su conocimiento permite a las personas sin escrúpulos aprovecharse indebidamente de los esfuerzos del programador.

El programa fuente puede ser impreso, leído, modificado y recompilado (o reensamblado) y vendido como nuevo con bastante impunidad, mientras que la versión objeto sólo es válida para hacerla correr. Sin embargo, el Homo Sapiens, siendo la astuta especie que es, le dará vueltas constantemente a la situación. Algunos, pacientemente y de forma manual, seguirán los unos y ceros binarios del módulo máquina para descubrir sus secretos ocultos; otros harán mal uso de los programas especiales llamados **descompiladores** o **desensambladores,** para automatizar esta conversión de máquina a fuente. Queremos poner énfasis en la expresión hacer mal uso, porque los descompiladores y los desensambladores son herramientas legítimas en manos apropiadas. Incluso los programadores más brillantes pierden a veces sus propios códigos fuente (de lo que le echamos siempre la culpa al ordenador), por lo que el uso del descompilador no es, en estos casos, ningún delito. El punto fundamental es que la mágica estructura del programa fuente puede recuperarse raras veces por medio de la descompilación, y el descifrado de un programa desensamblado (recuérdese lo cerca que estamos del lenguaje máquina) es una forma demasiado dura de ganarse la vida deshonestamente.

Resumen de los niveles de los lenguajes

Nuestra discusión sobre los niveles puede resumirse de la siguiente forma:

Nivel Alto = Fácil de programar y adaptar, portátil, mayor tamaño y más lento.

Nivel Bajo = Difícil de programar y adaptar, poco portátil, menor tamaño y más rápido.

Para aprovechar la velocidad y potencia del lenguaje máquina, debe usted aprender más acerca de las instrucciones del procesador y de cómo las busca en la memoria. Entonces podrá ocuparse de unos elementos esenciales (llamados **registros**) implementados en el interior del *chip,* que son los que ofrecen al programador control directo sobre el funcionamiento del procesador.

El set de instrucciones del 68000: Breve introducción

Ya vimos que el microprocesador, cuando funciona ejecutando un programa, toma la memoria, y entonces obedece una secuencia de instrucciones representadas por secuencias binarias de unos y ceros, en lenguaje máqui-

na. Estas instrucciones contienen mucha información empaquetada, que el procesador debe **decodificar** en el **decodificador de instrucciones** antes de que pueda saber dónde encontrar los datos, qué hacer con ellos y dónde poner los resultados. Después necesita saber dónde conseguir la *próxima* instrucción.

Como en la transmisión y recepción del código Morse, deben observarse algunos convenios en los extremos enviador y receptor para que pueda establecerse la comunicación de forma correcta. Uno de los convenios que, obviamente, debe conocer el procesador, es el relativo al número de unos y ceros que deben recibirse antes de empezar a decodificar. En el 68000, las reglas son sencillas:

Las instrucciones están en grupos de palabras de 16 bits.

La instrucción más sencilla ocupará un mínimo de una palabra.

Las instrucciones más complejas requerirán de palabras adicionales hasta un máximo de siete en total.

La primera palabra de cada instrucción indicará al procesador si debe buscar más palabras o no.

Asi, el procesador coge de la memoria y decodifica 16 bits de una vez, hasta que "sabe" qué debe hacer. Dependiendo de cada instrucción en concreto, el procesador realizará ciertas tareas. Si la instrucción indica "MOVE datos desde un sitio de la memoria a otro", por ejemplo, deberá contener información sobre las direcciones de origen y de destino de los datos.

Para entender la manera en que estas instrucciones y los datos a los que hacen referencia están dispuestos en la memoria, debemos fijarnos en la organización de ésta en el 68000. Aquí no nos importa nada relativo a las placas de memoria, los *chips* empleados o si se trata de memoria RAM o ROM. Lo que necesitamos entender es el modelo conceptual de organización de la memoria, tal y como lo ve el programador. La memoria del 68000 es un modelo de sencillez: exactamente una enorme pila de cajas numeradas, en cada una de las cuales hay un byte de 8 bits. El número de la caja es lo que se conoce como **dirección del byte.**

Modelo de memoria

La memoria del 68000, desde el punto de vista del programador, es una simple sucesión de direcciones de bytes, que van desde 0, 1, 2, 3,... hasta la máxima dirección permitida. En cada dirección encontrará un solo dato de 8 bits esperando a ser utilizado. Esta simple secuencia de direcciones forma lo que llamamos un **espacio lineal de direcciones,** en contraste con otras distribuciones más complicadas en las que la memoria está segmentada, tal como la estructura del Intel 8086/8088 (véase el capítulo 1).

La máxima dirección permitida viene fijada por el tamaño del *bus* de direcciones, el cual, a su vez, depende del miembro de la familia 68000 en concreto que se esté empleando. Los distintos miembros de esta familia del 68000 varían en cuanto al número de líneas físicas que conforman el *bus* de dirección (véase la tabla 3.1).

TABLA 3.1

Familia 68000: Tamaño de los buses

Modelo	*Tamaño de registros*	*Tamaño del* bus *de datos*	*Tamaño del* bus *de direcciones*	*Tamaño de la ALU*
MC68000	32	16	24	16
MC68008	32	8	20	16
MC68010	32	16	24	16
MC68012	32	16	30	16
MC68020	32	8/16/32	32	32

La determinación de la máxima dirección alcanzable es un simple ejercicio de aritmética binaria. Cada bit que se añade al *bus*, *dobla* el número de bytes direccionables de memoria. Un *bus* de dirección con una única línea física sólo podrá apuntar dos direcciones, llamémoslas dirección 0 y 1. Un *bus* de direcciones de dos líneas doblará esta capacidad hasta cuatro direcciones de bytes, digamos 0, 1, 2 y 3.

La capacidad de todos los 32 bits del *bus* de 32 direcciones del 68020 permite un **espacio lineal de memoria** de 2^{32} = 4 Gigabytes = 4.294.967.296 bytes. Así, las direcciones de los bytes del 68020 pueden ir desde 0 hasta 4.294.967.295. Es conveniente recordar este límite superior. Aparecerá muchas veces como el mayor número sin signo que se puede almacenar con un dispositivo de 32 bits. Así, le resultará más fácil recordar el binario:

11111111111111111111111111111111 (cuéntelos)

o el hexadecimal:

$FFFFFFFF (cuidado, el "$" significa hexadecimal, no dinero)

La importancia de los espacios de direccionamiento extralargos

Durante años, el precio de las memorias ha ido cayendo drásticamente, pero aún tardará un poco antes de que el usuario medio pueda adquirir los 4 Gigabytes de memoria física RAM. Sin embargo, las técnicas de VM (me-

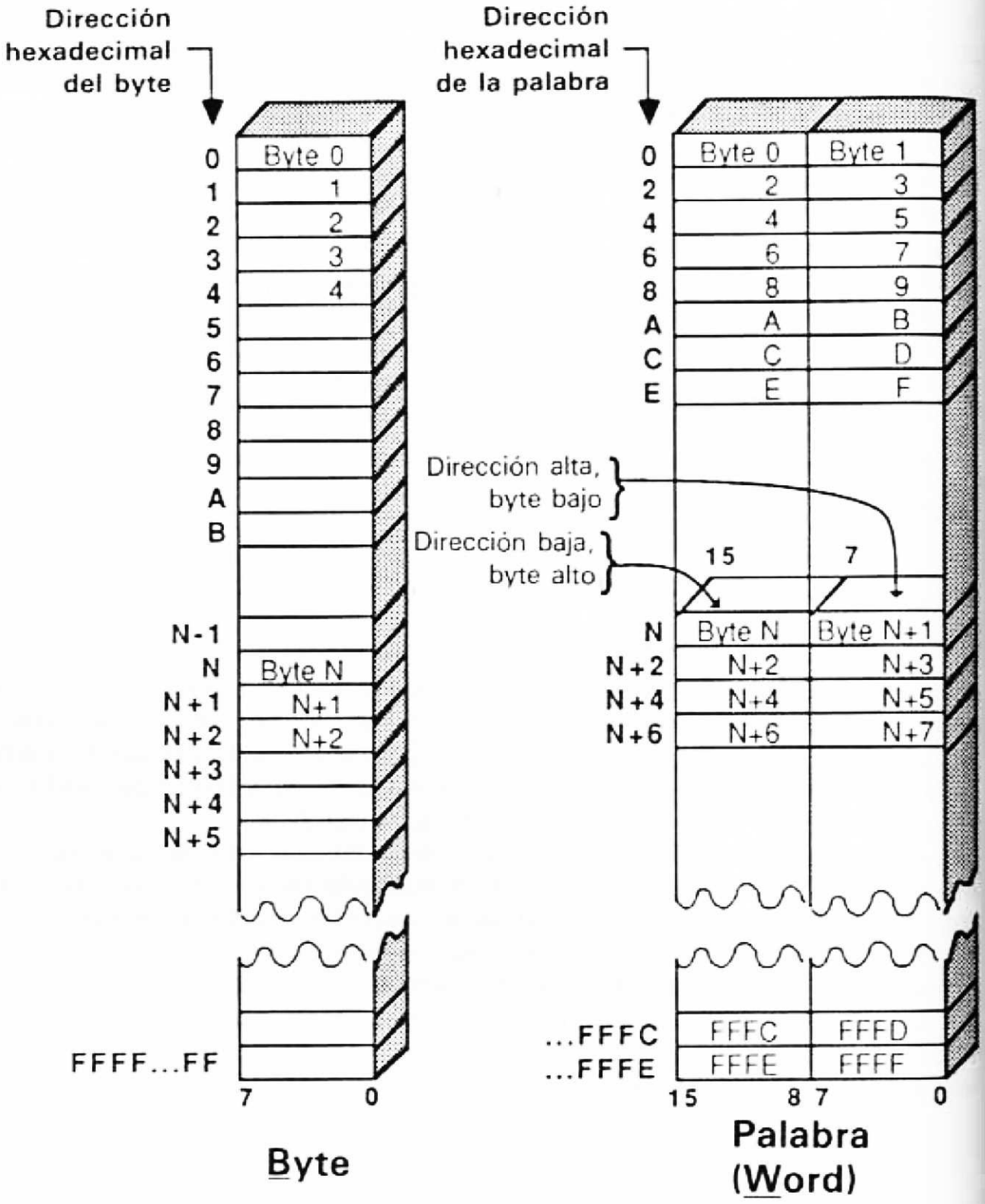


Figura 3.2a
Modelo de la memoria del 68000 (primera parte)

moria virtual) desarrolladas originalmente por Ferranti Atlas, a finales de los cincuenta, están disponibles en los últimos super micros. La VM le permite almacenar datos en direcciones de memoria situadas fuera del rango cubierto por la RAM realmente instalada, con lo que se evita la necesidad de instalar toda la RAM necesaria para cubrir el espacio total. Los analistas de sistemas están continuamente pensando formas de conseguir que una

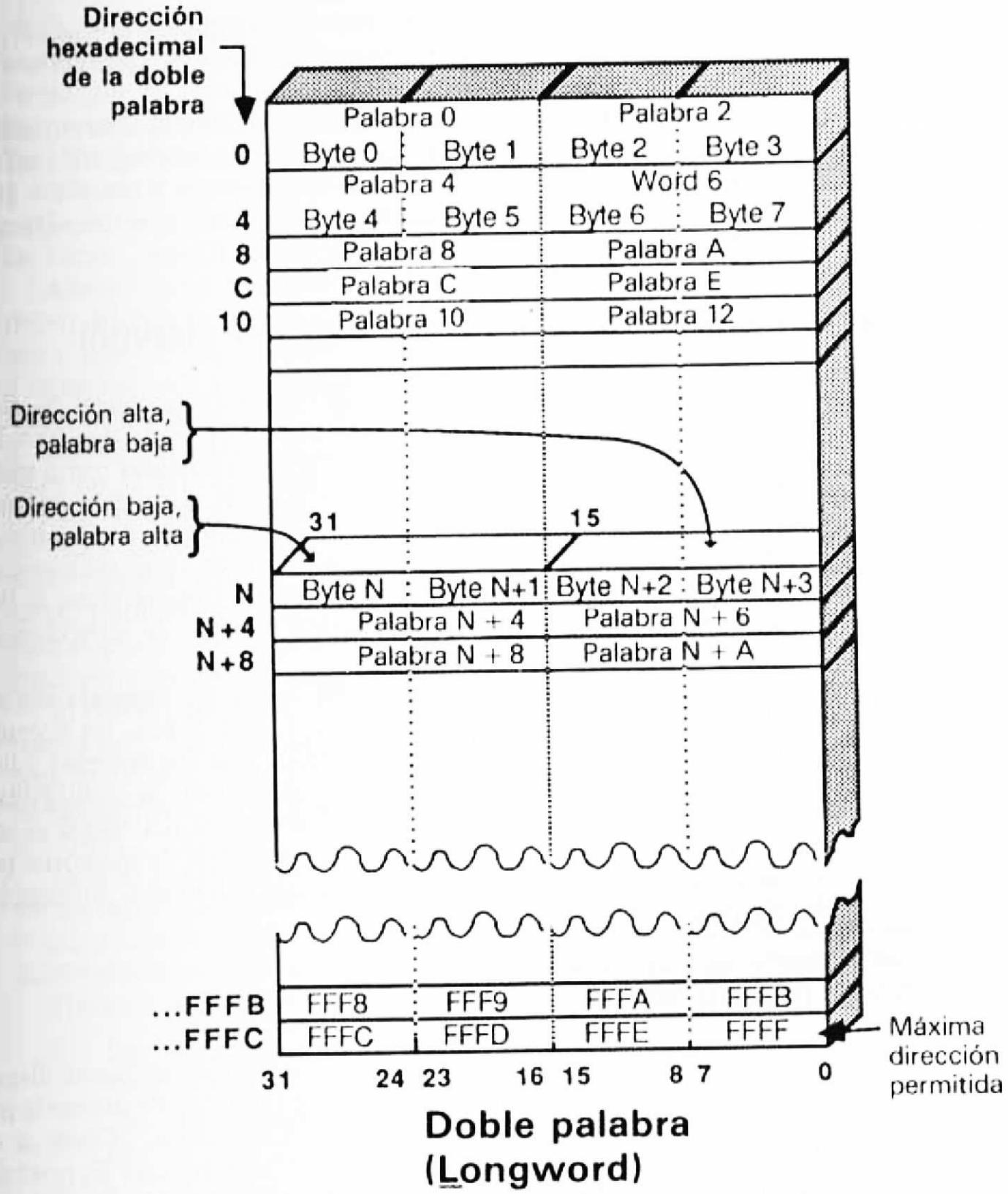


Figura 3.2b
Modelo de la memoria del 68000 (segunda parte)

cosa aparezca como otra, un arte conocido como **emulación.** La RAM-disco, por ejemplo, le permite acceder a la RAM como si del disco se tratara; pues bien, ¡la VM le permite acceder al disco como si fuera RAM! Lo último en emulación es la **máquina virtual,** una técnica que permite que un microprocesador funcione como si de otro se tratara (aunque aún no haya sido fabricado). Como veremos en los capítulos 7 y 8, el MC68010 y el MC68020 tienen ciertas características que animan a emplear esta técnica.

El 68000 tiene 24 de las 32 líneas **conectadas** al exterior, lo que significa un espacio de direccionamiento de 2^{24} = 16 Megabytes = 16.777.216 = = $1000000 bytes.

El procesador a "escala reducida" MC68008 tiene solamente 20 líneas de las 32 interiores conectadas al exterior, lo que significa un espacio de direccionamiento de "sólo" 2^{20} = 1 Megabyte = 1.046.576 = $100000 bytes. ¡Cómo cambian los tiempos! Parece que era ayer cuando nos moríamos por microprocesadores que soportaban 64 Kbytes.

El espacio reservado: Sólo para uso del sistema

Habiendo establecido nuestro amplio espacio lineal de direcciones, reservemos inmediatamente las direcciones comprendidas entre la 0 y la 1024 para usos específicos esenciales del 68000, conocidas como **memoria del sistema** y **datos del sistema.** Estas direcciones contienen importantes tablas para el tratamiento de las interrupciones, cargas de operativos, etc., y no deben ser alteradas por el usuario. Algunas de estas sacrosantas direcciones pueden estar en la ROM, de forma que no desaparezcan al desconectar la alimentación, pero seguirán utilizando espacio de direcciones como cualquiera otra posición de memoria.

Habrá, con toda seguridad, otras áreas de memoria asignadas permanentemente a otras funciones vitales y, por tanto, no accesibles al mero usuario de programas y datos. A medida que los sistemas y los programas de usuario van siendo más ergódicos, aumentan en complejidad y tamaño. Por ello, el gran espacio de direccionamiento del 68000 es una auténtica bendición. Hay muchos micros que llevan 128K de memoria, pero al cargar el sistema operativo y demás programas auxiliares, al usuario le quedan 50 Kbytes, o algo por el estilo.

Acceso a memoria

Aún nos queda gran cantidad de espacio de memoria disponible, y suponemos que tenemos algunas placas de circuitos de memoria para conectar parte o la totalidad de las direcciones disponibles. ¿Cómo se obtienen los datos y las instrucciones almacenadas en la memoria? Es preciso observar el ***bus* de datos** del 68000, un camino eléctrico de doble dirección, que conecta la memoria con el procesador.

El *bus* de datos

La tabla 3.1 nos trae a colación otro dato importante y significativo de la familia 68000: el **tamaño del *bus* de datos.** Este indica cuánta información puede transferirse en cada ciclo de lectura o escritura de memoria. Un *bus* estrecho precisará de más ciclos de lectura o escritura para una misma cantidad de datos.

En primer lugar, observemos que las líneas del *bus* de datos son totalmente independientes de las de los otros *buses*. Varias CPU están hechas de forma que se ahorran silicio y terminales multiplexando o compartiendo temporalmente líneas de datos y direcciones, e incluso otro tipo de líneas. Por *bus* **dedicado** entendemos aquel por el que, en cada ciclo de lectura o escritura de memoria, los bits del *bus* de datos pueden transferirse en paralelo, sin tener que esperar a que termine alguna otra actividad anterior de las líneas.

Ahora, como los distintos miembros de nuestra feliz familia del 68000 tienen diferentes tamaños de *bus* de datos (que van desde 8 hasta 22), deseará saber cómo pueden emplearse las mismas instrucciones de acceso a memoria para todos ellos. He aquí la forma: Cada instrucción, independientemente del tamaño del *bus*, incluye una letra L, W o B, que identifica el **tamaño del dato.** La CPU lo interpreta, y así transferirá una **doble palabra** de 32 bits (L), una **palabra** de 16 bits (W) o un **byte** de 8 bits (B) a o desde memoria. Cada 68000 realiza la acción apropiada. El número real de ciclos de lectura/escritura es **transparente** para el programador: una doble palabra precisará 4 ciclos (accesos de 4 × 8 bytes) en el 68008, 2 en el 68000 y solamente 1 en el 68020. Evidentemente, el 68020 es más rápido, pero el punto que ahora tocamos (otra vez) es el alto grado de compatibilidad entre todos los miembros de la familia 68000.

El 68000 básico lee 16 bits cada vez, lo que resulta banstante apropiado, puesto que, como ya sabemos, las instrucciones están codificadas en palabras de 16 bits, o múltiplos suyos. Pero si lo que se desea es un único byte de la memoria, el proceso empleará también un ciclo, igual que para leer dos bytes.

Debido al tamaño de los datos de las instrucciones del 68000, podemos ver su memoria como si estuviera dividida en bytes en las direcciones de bytes, palabras en las direcciones de palabras o dobles palabras en las de dobles palabras. En las figuras 3.2a y 3.2b se muestra cómo se hace esto.

He aquí las reglas:

1. Las direcciones de los bytes pueden ser pares o impares. Sus incrementos son de 1 en 1.
2. Las direcciones de las palabras son siempre pares. Sus incrementos son de 2 en 2.
3. Las direcciones de las dobles palabras son también pares. Sus incrementos son de 4 en 4.
4. Cuando se accede a una palabra de la dirección N (donde N es un número par cualquiera), el byte de la dirección N es el **byte alto** y el de la dirección N + 1 es el **byte bajo.** Vuelva a leerlo bien. Esto significa que, cuando se mira una palabra de memoria, su byte más significativo está en la dirección más baja, y su byte menos significativo está en la dirección más alta. Ahora, borde la frase siguiente en un cuadro y cuélguela de la cabecera de su cama:

Dirección baja/byte alto, dirección alta/byte bajo

5. Al acceder a una palabra doble en la dirección N (donde N es un número par cualquiera), la palabra de la dirección N es la más significativa, mientras que la de la dirección N + 2 es la menos significativa. Aparece una inversión, como en la regla número 4, a tener en cuenta. De las dos palabras de una doble palabra, la más significativa está en la dirección de palabra más alta. Recite la siguiente consigna:

   Dirección baja/palabra alta, dirección alta/palabra baja

6. Si se intenta acceder a una palabra o una doble palabra en una dirección impar, se conseguirá un mensaje de error. El 68020 es más misericordioso, pero por ahora ésta es una sabia regla que debemos seguir.

Si el 68000 es el primer microprocesador que usted estudia con este detalle, no le resultarán chocantes las reglas Motorola del direccionamiento de bytes, palabras y dobles palabras. Si no, se dará inmediatamente cuenta de que los puntos 4 y 5 son exactamente los contrarios de muchos otros *chips*. Es como conducir por el lado contrario en Inglaterra —no es una cuestión profunda de ética sobre el bien y el mal, pero hay que conocer y seguir las reglas—. Cuando, en los próximos apartados, lleguemos a los registros, veremos que las reglas de direccionamiento del 68000 son consistentes y sensatas[6].

Resumamos ahora lo que hemos aprendido de la organización y direccionamiento de la memoria en el 68000.

Resumen del modelo de memoria

- Un gran espacio de direccionamiento lineal. Sin restricciones en el tamaño del programa (hasta el máximo valor permitido).
- El MC68008 puede direccionar hasta 1 Megabyte. Los MC68000/68010/68012 hasta 16 Megabytes, y el MC68020 hasta 4 Gigabytes.
- El direccionamiento a bytes (8 bits), palabras (16 bits) o dobles palabras (32 bits) se controla mediante códigos especiales en la propia instrucción.
- Las direcciones de palabra y dobles palabras son pares.
- Las instrucciones de los programas se almacenan como palabras; los datos pueden tratarse como bytes, palabras, o dobles palabras.

---

[6] Evidentemente, esta forma de direccionar la memoria resultará familiar a todos los que ya hayan utilizado microprocesadores de Motorola, aun aquellos de 8 bits, donde los datos se almacenan en el orden en que estamos acostumbrados a escribirlos sobre el papel. Los que hayan trabajado con Intel o Zilog, recordarán que las magnitudes de dos bytes se almacenaban en el orden inverso.

- La palabra de la dirección N tiene en N el byte más significativo, y en N + 1 el menos significativo.
- La doble palabra de la dirección N tiene en N la palabra más significativa, y en N + 2 la menos significativa.
- La doble palabra de la dirección N tiene 4 bytes en las direcciones N, N + 1, N + 2 y N + 3.

Modelos de registros

Ahora que ya tenemos una idea general de la forma en que el 68000 direcciona la memoria RAM y ROM exterior, podemos aventurarnos a conocer los dispositivos internos del propio *chip*, llamados **registros**, que permiten al programador realizar un control directo o indirecto sobre las distintas facetas del funcionamiento del 68000.

El corazón de la programación del 68000 está en su potente, aunque sencillo, set de instrucciones. Cada una de las más o menos 60 instrucciones básicas, tales como ADD o MOVE, realiza un paso de programa específico manipulando o interactuando con el contenido de los registros internos y con la memoria externa, de forma que nuestro modelo de programación deberá incluir los registros y los modos de direccionamiento de la memoria como accesibles al programador, antes de dar sentido al set de instrucciones.

Hasta cierto punto, estamos enfrentándonos a una situación similar a un "plato combinado", puesto que los registros, los modos de direccionamiento y las instrucciones están íntimamente relacionadas y son mutuamente dependientes. Algunas propiedades de los registros del 68000 pueden parecer algo arbitrarias, hasta que, en los capítulos del 4 al 6, se describan algunas instrucciones que revelen súbitamente la belleza y la simetría del 68000. Le rogamos que espere hasta que describamos cómo son los registros y las muchas funciones que realizan.

¿Qué es un registro?

En primera aproximación, un registro puede ser visto como una pequeña parte de una RAM muy rápida montada en el interior del *chip*, rápida, debido a que los datos contenidos en él pueden ser accedidos y actualizados por el procesador sin necesidad de perder tiempo en realizar ciclos de búsqueda en memoria. Los registros también tienen direcciones numéricas como la RAM, pero sólo a efectos de referencia interna en las instrucciones en lenguaje máquina; el programador en lenguaje ensamblador siempre utiliza direcciones simbólicas, tales como D1, PC, o A6. La gran diferencia entre los registros y la RAM es que aquéllos disponen de varias funciones específicas implementadas y están conectados directamente con las unidades de control del *chip* para poder llevar a cabo estas funciones. Desde el punto de vista del programador, los registros son unidades "inteligentes" y ultrarrápidas de RAM.

Tipos de registros y sus funciones

Como vimos en el capítulo 2, una de las principales cuestiones en el diseño de los *chips* es la que se refiere al número de registros y a cuántas funciones diferentes deben tener. Las siguientes funciones son esenciales y, de una u otra forma, deben siempre existir:

Areas pasivas de trabajo para retención de resultados intermedios

La más inmediata, pero no la menos importante, de las funciones es la de mantener los resultados intermedios de la ejecución de un programa. El hecho fundamental ahora es que los datos almacenados en registros internos pueden ser accedidos y procesados muy rápidamente, con instrucciones cortas y económicas. Desde luego, cuantos más registros se tengan y mayores sean éstos, más datos (tanto números como direcciones) pueden almacenarse listos para ser usados, sin necesidad de malgastar tiempo en intercambios con la memoria externa.

Por más rápida que sea la RAM, habrá que emplear ciclos en el cálculo de direcciones, acceder a los datos, recuperarlos y almacenar los resultados en la memoria. Para alcanzar el máximo rendimiento, siempre intentamos alimentar el sistema con datos, de combustible, de los registros. Es un caso muy semejante al de las máquinas de calcular de sobremesa: si sólo se dispone de un registro, deberemos escribir a menudo los subtotales y volver a introducirlos, mientras que el uso de una máquina con múltiples registros nos evitará este engorro.

Operaciones lógicas y aritméticas

Los registros, tradicionalmente conocidos por **acumuladores**, están conectados con la ALU (unidad aritmética y lógica) para recibir los resultados de sus operaciones. En los primeros microprocesadores había, por regla general, un único acumulador, obligando al programador a "barajar" datos antes y después de realizar cada operación aritmética.

Con los **registros de datos** de propósito general, tal como los existentes en el 68000, se pueden efectuar las operaciones aritméticas sin necesidad de codificar este trasiego de datos.

Operaciones de direccionamiento

Los registros de dirección ofrecen al procesador las direcciones de memoria donde encontrar o salvar los datos. Antes de que una instrucción pueda acceder a la memoria deberá, obviamente, determinar la dirección sobre la que se quiere operar; la forma más frecuente consiste en obtener

esta información en un registro de direcciones, pero hay otras formas más complejas en las que las direcciones se calculan a partir de dos o más registros.

Los distintos *microchips* emplean distintos esquemas de direccionamiento, y su consecuencia está en los distintos tipos de registros de propósito especial empleados en las distintas tareas fundamentales para el cálculo de la dirección **efectiva**, donde están los datos precisados por una instrucción. En los Intel 8086/8088, por ejemplo, tal y como dijimos en el capítulo 1, hay registros especiales destinados exclusivamente a contener el valor de los segmentos, de las bases y de índice con los que se genera la dirección efectiva. Veremos que la estructura lineal no segmentada del 68000 simplifica considerablemente la situación: Hay un único tipo de registros de dirección (del que hay cinco diferentes) y cualquier registro de datos puede emplearse como registro índice.

Secuencia de programa

El PC (contador de programa) es un registro de dirección especial, dedicado a mantener la dirección de memoria donde está la instrucción presente; de esta forma, el procesador sabe dónde está la próxima dentro del programa.

Punteros de pila

Teóricamente, un puntero de pila es, simplemente, un registro de dirección empleado para apuntar a una zona específica de memoria, conocida como "pila" [7]. Las pilas, como se verá en el capítulo 5, juegan un importante papel como zonas de memoria donde salvar los datos mientras la CPU interrumpe su trabajo para realizar algún otro. La mayor parte de las CPU tiene uno o más registros dedicados a conservar tales direcciones de memoria.

Estado del procesador

Bajo diferentes nombres, tales como SR (registro de estado o de estatus), PSW (palabra de estado del procesador) o CCR (registro de códigos de condición), se engloban ciertos registros necesarios para contener los

[7] El nombre de "pila" se debe al concepto de "apilamiento" sobre la forma en que los datos se almacenan en una zona de memoria. A diferencia del direccionamiento convencional de la memoria RAM, en este caso, por razones de la urgencia con que se suele utilizar, basta con conocer la dirección del primer dato; el resto viene a continuación, además en un determinado orden preestablecido, uno tras otro, lo que, visualizado como se ha hecho con las direcciones de memoria, supone un verdadero apilamiento.

datos relativos a los distintos estados, o condiciones, acaecidos en los sucesivos pasos del programa.

Típicamente, el procesador **marcará** los eventos, tales como si el resultado de una suma es cero o negativo, poniendo a 0 o a 1 determinados bits del SR. Ciertas instrucciones permiten que el programador compruebe el valor de estos bits y, de acuerdo con ello, se altere o no la ejecución del programa.

Otros bits del SR marcan los niveles de **prioridad de interrupción.** Son los empleados para controlar ciertas vicisitudes que pueden suspender temporalmente la ejecución del programa en funcionamiento. A tales suspensiones se les llama, naturalmente, interrupciones. Cuando se solicita una interrupción por parte de, pongamos por caso, una unidad de E/S, la CPU debe decidir la urgencia relativa de lo que esté haciendo en este momento, frente a lo que la unidad de E/S desea hacer. Tal decisión se tomará en función de los valores numéricos de los niveles de prioridad almacenados en el SR.

Tras esta breve revista a los tipos y funciones de los registros, estamos ya en condiciones de abordar el modelo básico de los registros del 68000.

Modelo básico de registros del 68000

La figura 3.3 muestra el modelo básico de registros aplicable a todo el rango de CPU 68000. A medida que nos movemos desde el 68008 a través del 68000, hasta el 68010 y el 68020, vamos encontrando mejoras del modelo, pero sin abandonar nada en absoluto de los anteriores. Para el programador se trata de **mejoras compatibles** —lo que es posiblemente lo mejor que pueda pedirse de un rango completo de microprocesadores.

A pesar de que la tecnología del IC, los terminales, los tamaños y formas, las velocidades y las capacidades de direccionamiento de los distintos elementos de la familia sean diferentes, la gama 68000 es de mejora compatible, en cuanto al código máquina se refiere. Los programas escritos para el 68000, cosecha de 1979, funcionarán sin problemas en el 68020, de 1986.

En seguida nos ocuparemos de los cinco tipos de registros de nuestro modelo, analizando cada uno con detalle:

1. Registros de 32 bits de *datos,* de los cuales hay ocho, denotados por D0-D7.
2. Registros de 32 bits de *direcciones,* de los cuales hay siete, denotados por A0-A7.
3. *Punteros de pila* de 32 bits. Hay dos de éstos: el USP (puntero de pila de usuario) y el SSP (puntero de pila supervisor). Puesto que en cada momento sólo uno puede estar activo, a ambos se les denota por A7; pero debemos recordar que hay dos punteros de pila diferentes manteniendo dos pilas distintas (llamadas, naturalmente,

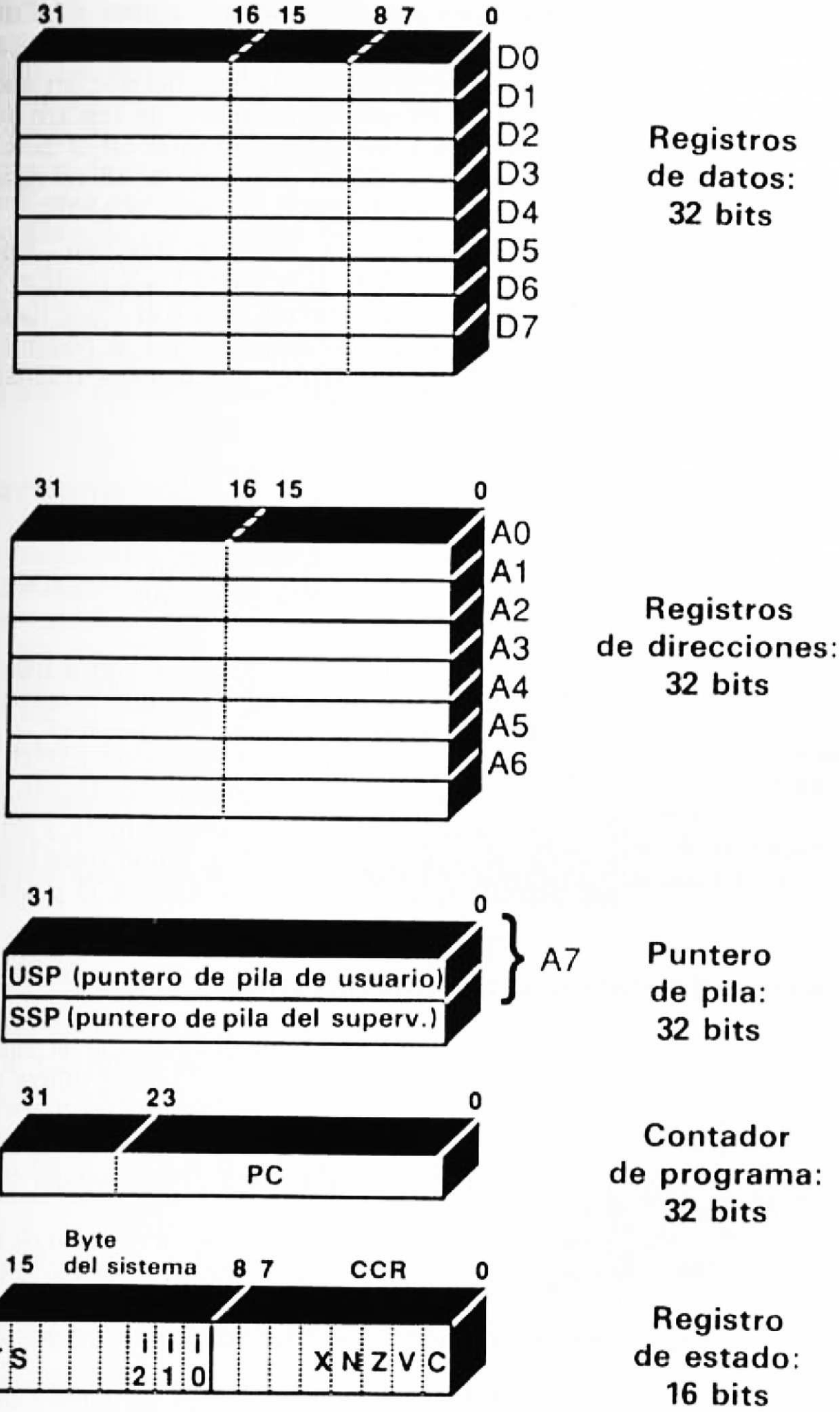


Figura 3.3
Modelo básico de los registros

pila de usuario y de supervisor), y ambos mantienen sus propios valores, a pesar de compartir una misma denominación (dirección).

4. *Contador de programa* de 32 bits, del que hay uno solamente, denotado por PC. Es muy semejante a un registro de direcciones, pero está especializado en retener el valor de la dirección de la instrucción en ejecución. Según el modelo de 68000 varía el número de bits efectivos: 24 para el 68000, 32 para el 68020.
5. *Registro de estado*, del que sólo hay uno, denotado por SR. Su byte inferior (bits 0 al 7) se llama CCR (registro de códigos de condición), mientras el superior (bits 8 al 15) se llama **byte del sistema**. El CCR tiene cinco indicadores, que se ponen a 1 o se borran a 0 para indicar las diferentes condiciones resultantes de cada operación:

C = Acarreo
V = Rebose
Z = Cero
N = Negativo
X = Extensión

El byte del sistema dispone de uno o varios indicadores *(flags)* para saber en cuál de los dos estados posibles se encuentra (bien en el **privilegiado** o en el **no privilegiado** de usuario), y es este indicador el que determina qué numero de pila, SSP o USP, es activo. Además, el byte del sistema tiene una máscara de interrupciones de 3 bits (i0-i2) para marcar el nivel de prioridad (0-7), y un indicador para determinar si el procesador está en **modo traza** (una forma de trabajo que permite que el procesador ejecute el programa paso a paso).

Registros de datos y de direcciones

Los registros de datos y de direcciones son el caballo de batalla del 68000, y la mayor parte de las instrucciones los utiliza de un modo u otro. Como su nombre indica, los registros D0-D7 de datos se emplean para la manipulación genérica de *datos*, mientras que en los de direcciones A0-A7 estarán las direcciones necesarias para poder acceder o actualizar informaciones de la memoria.

Tipicamente, una instrucción emplea un registro de dirección que indica dónde conseguir un número de la memoria y colocarlo en un registro de datos; entonces, hacemos las sumas referidas a un registro de datos y, finalmente, usamos un registro de direcciones para poder devolver la respuesta a la memoria.

Las distintas funciones de los registros de datos y de direcciones son un reflejo de la interconexión que hay entre ellos, y de las reglas que gobiernan las instrucciones a utilizar.

La única operación aritmética que puede pretenderse hacer con un registro de dirección es la de sumarle o restarle alguna cantidad para indicar **(indexar)** una determinada posición: es muy poco frecuente necesitar multiplicar o dividir direcciones. Además, las direcciones son siempre magnitudes de 16 ó 32 bits, por lo que careceria de sentido diseñar registros de direcciones capaces de manejar bits o bytes.

Todo esto se refleja en el set de instrucciones, al haber algunas de ellas que actúan sólo con registros de datos o de direcciones, o que, quizá, actúan de distinta forma con cada uno de ellos. Estas excepciones aparentes pronto aparecerán bastante lógicas y naturales en el contexto del 68000.

Los registros de datos han sido diseñados para trabajar con todo tipo de operaciones aritméticas y lógicas y, dado que su tamaño es de 32 bits, también pueden utilizarse como **registros índice.**

Simetría de los registros

Todos los registros de datos se comportan de la misma manera, así como todos los registros de direcciones, que lo hacen igual, de forma que, comparado con muchos otros microprocesadores, hay muchas menos personalizaciones que recordar.

Si se compara el 68000 con sus rivales más directos, veremos que Motorola ofrece un conjunto de registros muy claro y simétrico y una uniforme estructura de 32 bits para datos y direcciones. Un set de instrucciones que goza de esta uniformidad se suele llamar **ortogonal** y, aunque incrementa la complejidad de la CPU, facilita enormemente la programación.

Ahora, veamos los registros de datos con más detalle, deteniéndonos en las subdivisiones básicas de sus 32 bits y la forma en que la CPU las utiliza.

Registros de datos

Todos los registros de datos, D0 a D7, son como el mostrado en la figura 3.4. Los bits se enumeran desde el 0, en el extremo de la derecha correspondiente al LSB (bit menos significativo), hasta el 31, el MSB (bit más significativo). Hay que acordarse de que la numeración empieza en el 0, a la derecha, hasta el 31, a la izquierda. Si no se respeta este convenio, se conseguirán espectaculares "resultados". El bit 1 es el *segundo* por la derecha.

Los 32 bits pueden utilizarse en casi todas sus combinaciones (si usted se ve inclinado a ello, puede utilizar los bits 5, 19 y 28, por ejemplo); sin embargo, el set de instrucciones ha sido concebido para la más rápida actuación sobre las tres subdivisiones más comunes, que son:

Doble palabra de 32 bits: una por registro de datos
Palabra de 16 bits: dos por registro de datos
Byte de 8 bits: cuatro por registro de datos

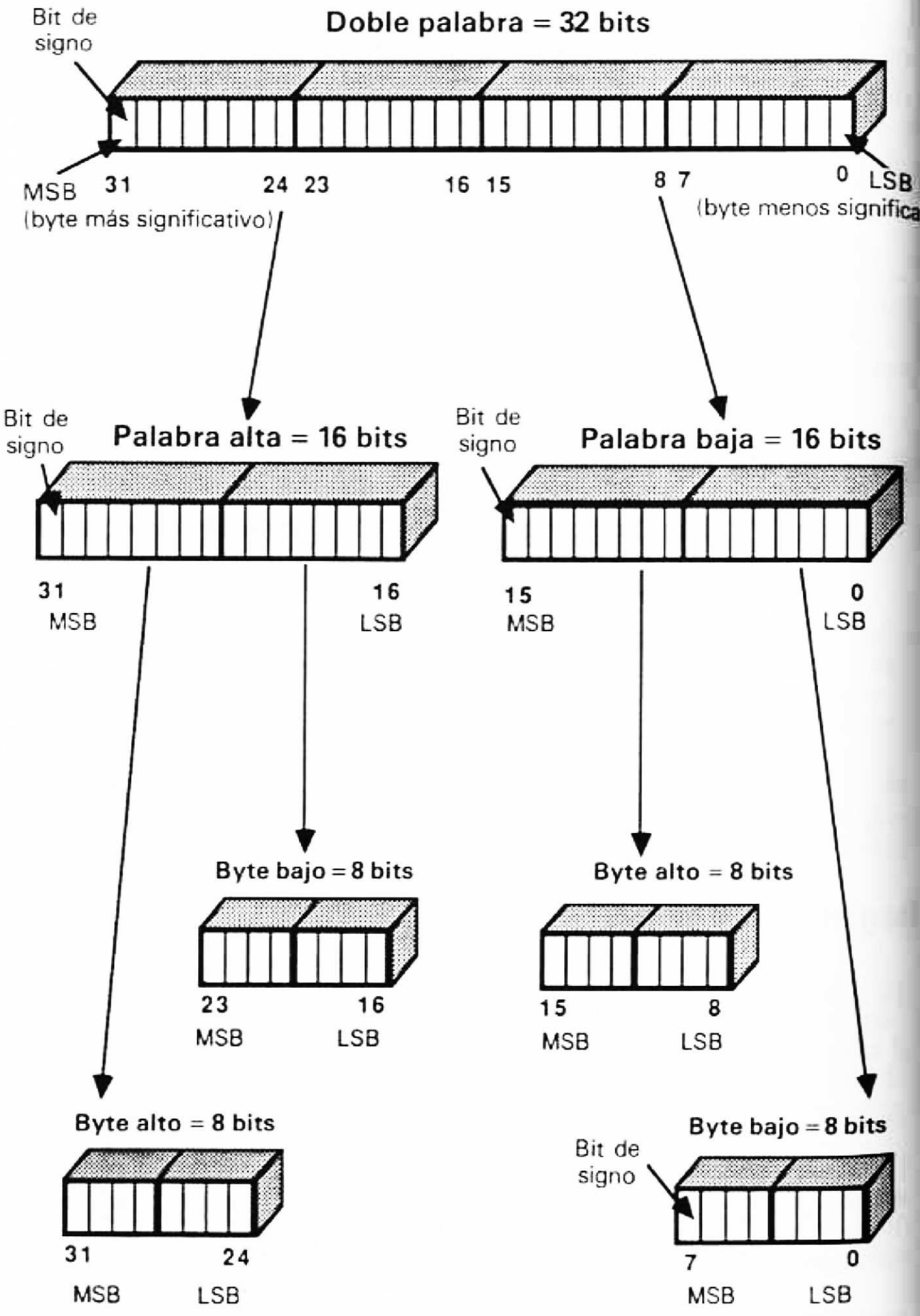


Figura 3.4
El registro de datos

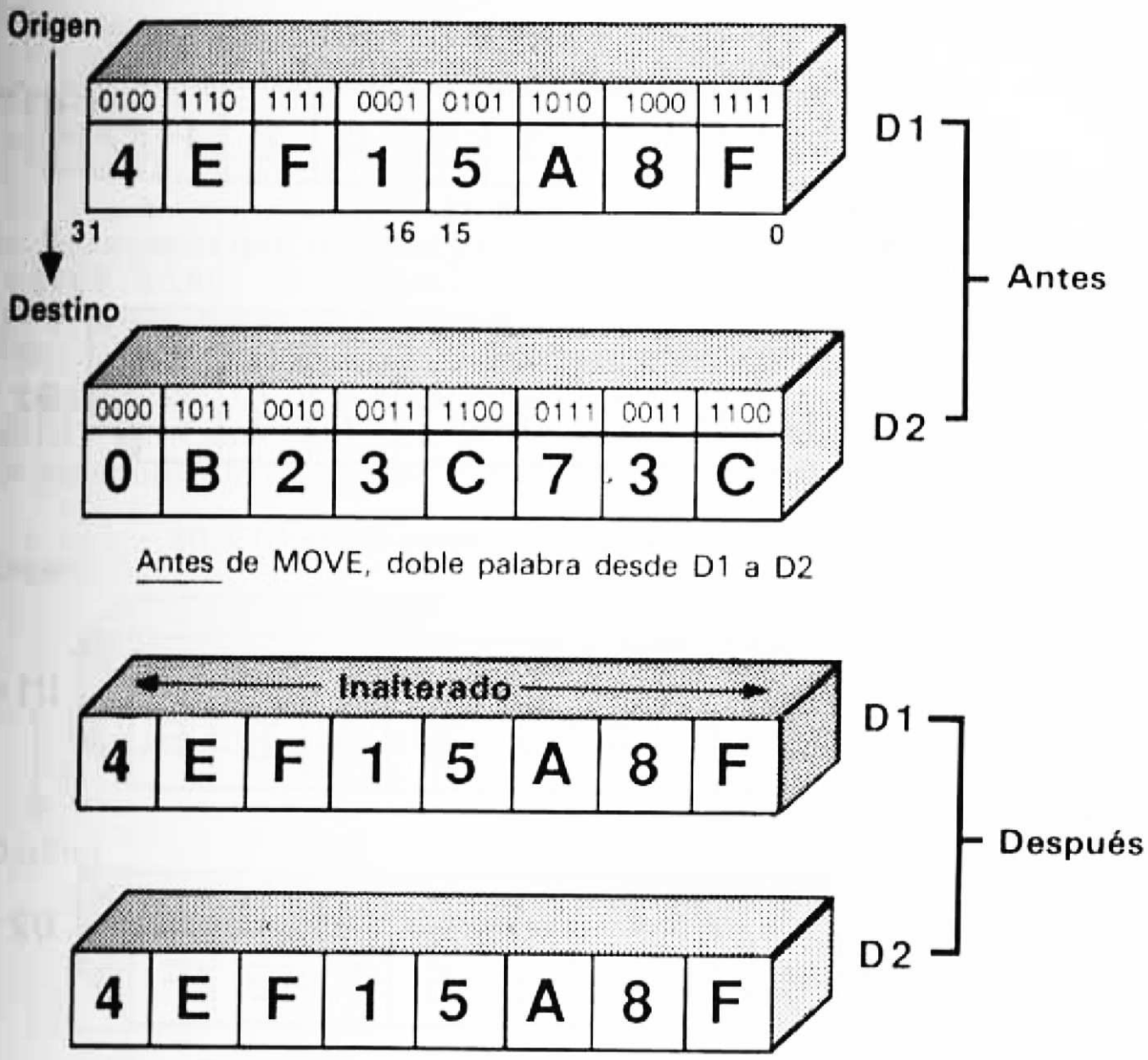
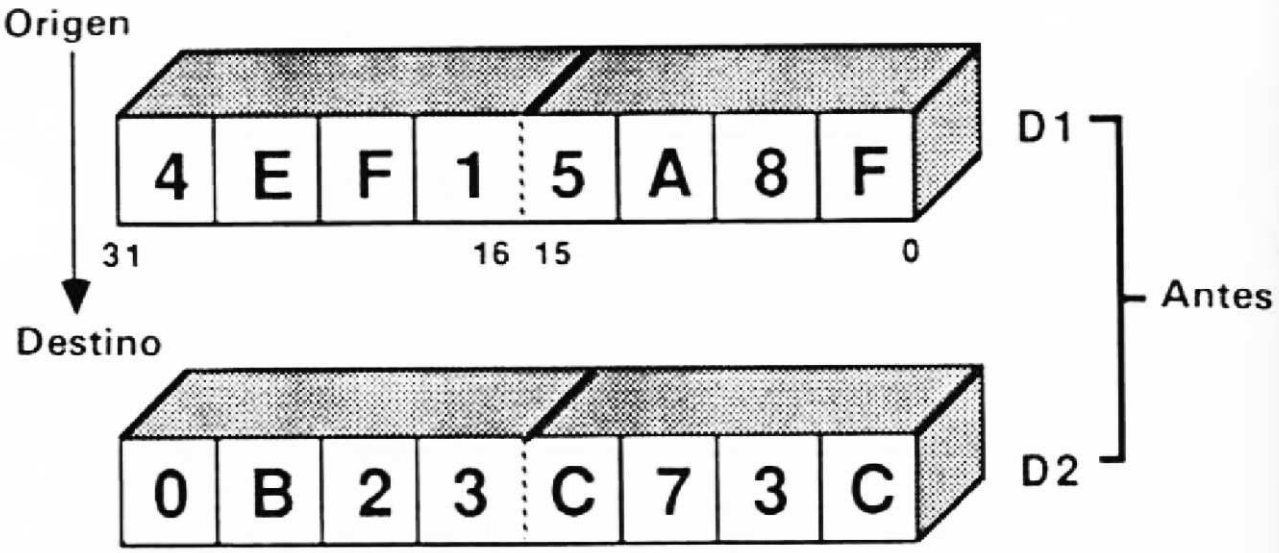


Figura 3.5
Operación con doble palabra: MOVE.L

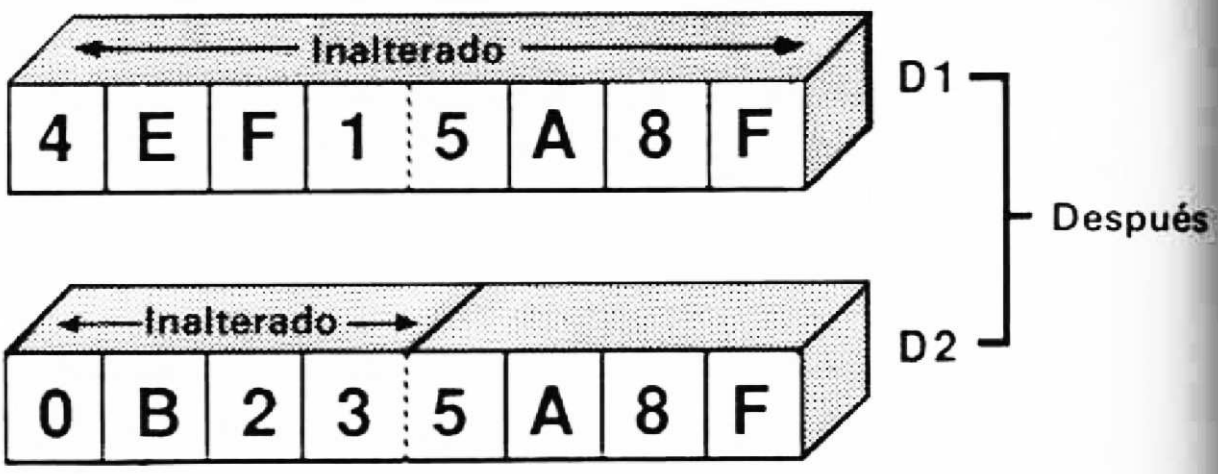
Con menor frecuencia, podemos precisar trabajar con la subdivisión en cuatro-ocho *nibbles* de 4 bits cada uno, pero este aspecto lo dejamos hasta que tratemos de las operaciones en BCD (binarios codificadas en decimal) en el capítulo 6.

La figura 3.4 muestra los nombres aplicados en estas subdivisiones. Se habla de palabra alta y palabra baja en una doble palabra, y de bytes alto y bajo en una palabra.

En el apartado de memoria aparecían los direccionamientos de bytes, palabras y dobles palabras. Por tanto, no es inesperado que en las operaciones con registros de datos tengamos las mismas posibilidades.



Antes de MOVE, palabra desde D1 a D2



Después de MOVE, palabra desde D1 a D2

Figura 3.6
Operación de palabra. MOVE.W

Códigos del tamaño de los registros

La mayor parte de las instrucciones trabaja en los tres modos, es decir, dobles palabras, palabras y bytes, lo cual viene indicado por las letras L, W y B en la instrucción. Así, se indica qué parte del registro es la afectada: el total para la doble palabra, la palabra inferior para la palabra, o el byte inferior para el byte.

Veamos ahora cómo funciona esto, centrándonos en la instrucción MOVE para llevar los datos de un registro a otro. Siempre se llevan los datos *desde* el registro fuente *hasta* el de destino.

El formato de la instrucción MOVE es:

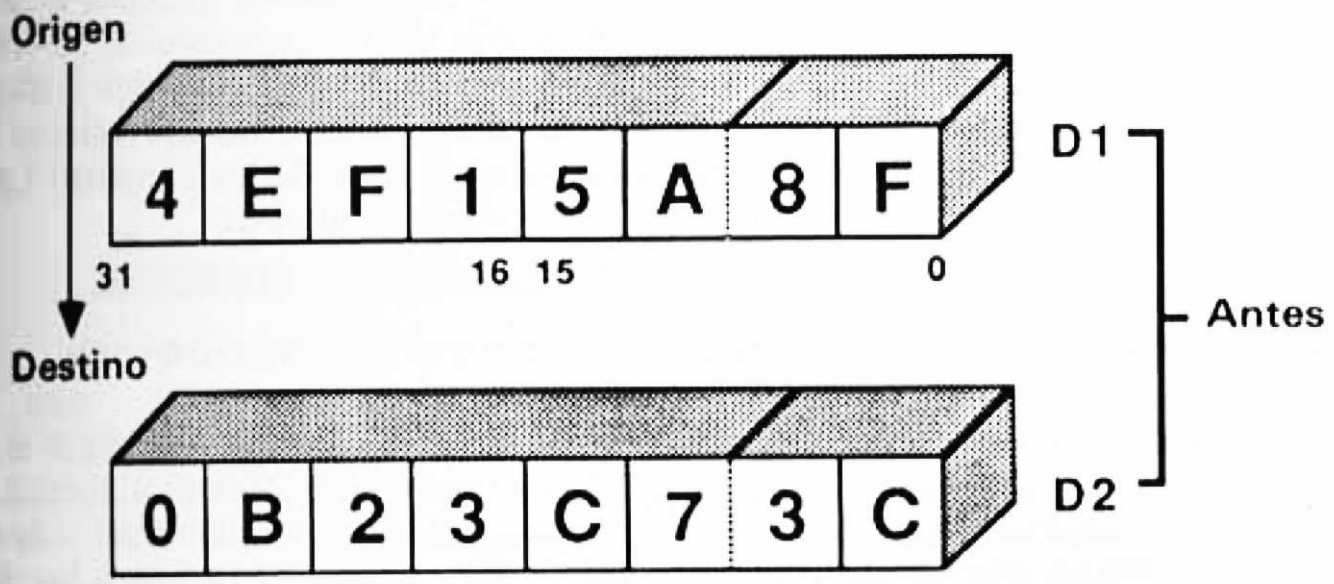
MOVE.<código de tamaño> <fuente>,<destino>

Nuestros ejemplos son:

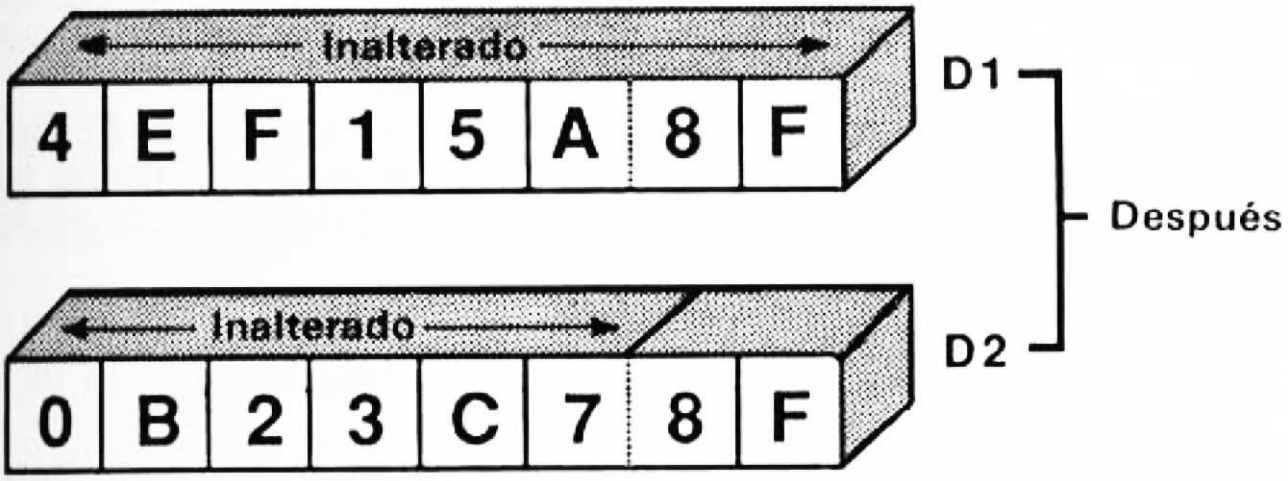
```
MOVE.L D1,D2
MOVE.W D1,D2
MOVE.B D1,D2
```

Supongamos que, inicialmente, D1 contiene el decimal 1.324.440.200 = = $4EF15A8F ($ = hexadecimal) y que D2 contiene el 186.894.140 = $0B23C 73C.

En la figura 3.5, la instrucción MOVE.L D1,D2 lleva la doble palabra de D1 a D2. Como esperábamos, D2 acaba teniendo el valor $4EF15A8F. Se han transferido los 32 bits. El registro fuente queda con su valor inicial.



Antes de MOVE, byte desde D1 a D2



Después de MOVE, byte desde D1 a D2

Figura 3.7
Operación de byte: MOVE.B

En la figura 3.6 se ha cambiado el tamaño a MOVE.W. Esta vez, sólo la palabra inferior de D2 es la afectada: toma el valor de la inferior de D1. La superior, en ambos, queda con su valor inicial.

Finalmente, en la figura 3.7 se ha hecho un transvase en **modo byte.** Solamente se transfiere el byte inferior (bits 0-7) de D1 al byte inferior de D2. Los tres bytes superiores de D2 permanecen inalterados.

Resulta de gran utilidad en muchos casos tener la posibilidad de operar sobre unas partes de los registros, dejando las demás inalteradas. En algunos ordenadores, el movimiento de un byte, como el de la figura 3.7, puede precisar de tres o más pasos, o quizá incluso de un registro más.

Aritmética de registros

Dado que la principal misión de los registros de datos es la suma, debemos fijarnos ahora en la aritmética de los registros de 32 bits. Como veremos, de la misma forma que con los MOVE, pueden realizarse cálculos en 8, 16 y 32 bits. También, como entonces, las operaciones de sumas en bytes y palabras no afectan las partes altas de los registros. La elección del tamaño determinará el rango numérico cubierto.

Código de tamaño de datos y rango numérico

Tal y como vimos en el capítulo 1, a cada bit que añade el diseñador se dobla el rango del registro y, por tanto, su exactitud numérica. Así, cuanto mayores sean los registros, mayor exactitud se alcanzará —de ahí la excitación que produjo el paso desde los registros de 4 bits a los de 8, y de ahí a los de 16 bits. La búsqueda interminable de mayor exactitud está hoy en los 80 bits de los registros disponibles en el coprocesador 68881 de Motorola y similares devoradores de números—. Como veremos, hay muchos trucos de programación útiles para combinar los registros con vistas a conseguir la que se llama resultados en multiprecisión, pero lo importante es disponer del *hardware* que lo haga.

Como vimos en el capítulo 1, los registros de 32 bits cubren un rango sin signo que va desde 0 hasta + 4.294.967.295, mientras que con complemento a 2 se cubren los números con signo dentro del rango desde −2.147.483.648 hasta + 2.147.483.647.

El signo

Con la notación del complemento a 2 (véase la tabla 1.1 del capítulo 1), veíamos que los números negativos tienen su bit 31 (el situado a la izquierda del todo) a 1, mientras que los positivos, incluyendo el cero, tienen el bit 31 a 0. Por tanto, a ese bit se le denomina como el **bit de signo.**

De forma similar, en las operaciones con números de 16 y 8 bits en no-

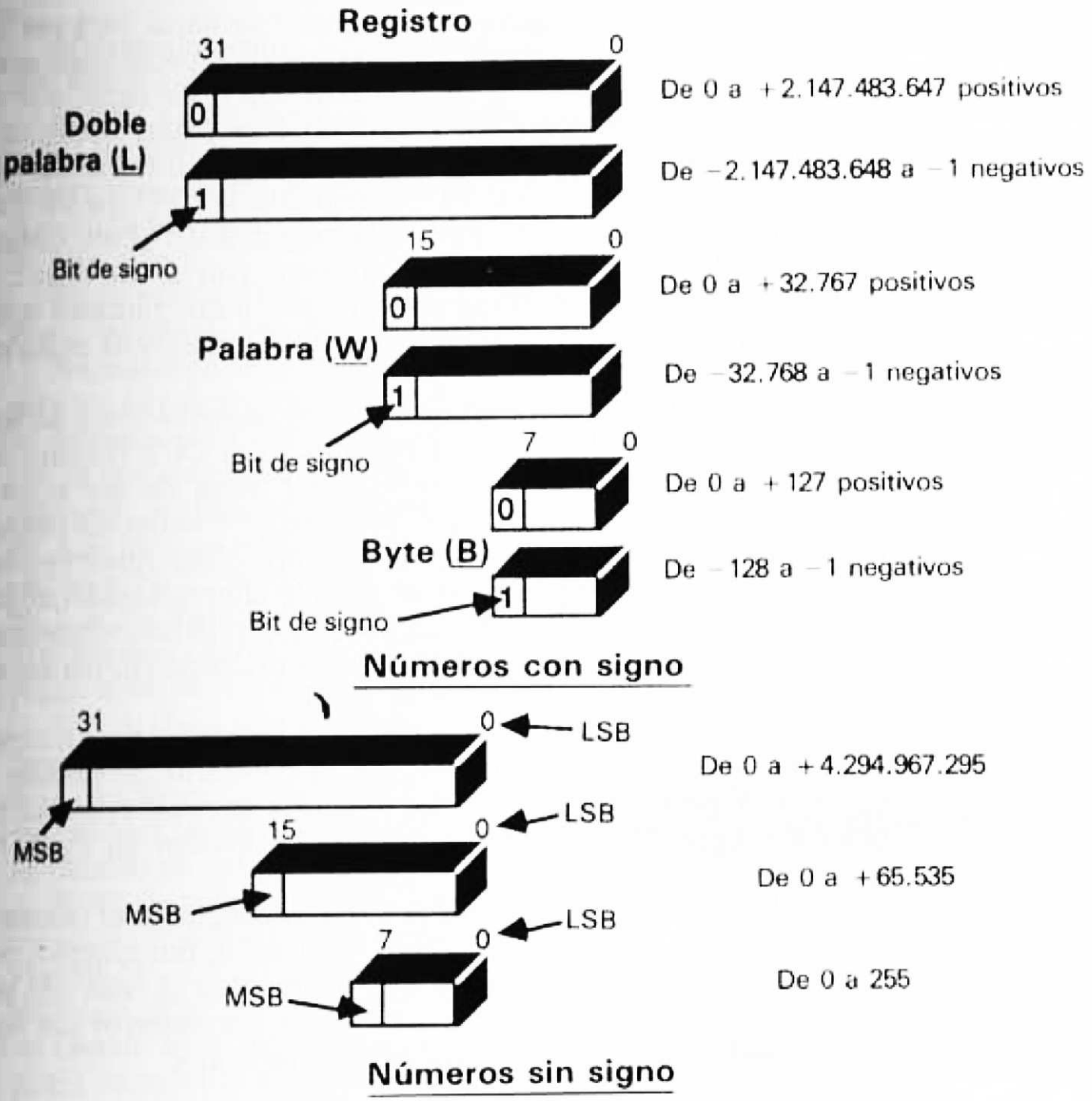


Figura 3.8
Rango de los números con y sin signo

tación de complemento a 2, el bit de signo está, respectivamente, en las posiciones 15 y 7, indicando asi el signo de la palabra, o del byte.

La regla importante para los números con signo es:

Bit de signo = 0 para los números positivos (incluyendo el cero)
Bit de signo = 1 para los números negativos

Para los números sin signo, desde luego, este bit representa su valor binario normal (2^{31}, 2^{15} ó 2^{7}). Los números sin signo no disponen, evidentemente, de bit de signo: emplean todos los bits disponibles para alcanzar el rango positivo máximo. En la figura 3.8 pueden verse los rangos alcanzados con cada tamaño de datos: L, W y B.

Ahora, si usted se asoma a un registro de datos, verá una fila de 32 bits: ceros y unos en abundancia. El 68000 no da especial importancia intrínseca a estos bits; puede tratarse de números con o sin signo, o de caracteres no numéricos. Se obedecerá sin preguntas a cualquier operación legal que se programe sobre este registro. Si, por ejemplo, usted decide una operación tal como ADD para sumar "1" al registro, la CPU tratará el contenido del registro como un número. Pero a la pregunta de: ¿Debe conocer la CPU si el contenido del registro es un número con o sin signo?; la respuesta es:¡No! Resulta que ADD es una operación que funciona igual con números de uno y otro tipo, con tal de que el resultado esté en los rangos descritos en la figura 3.8.

Dado que los rangos varían entre los números con y sin signo, el 68000 proporciona dos indicadores diferentes en el CCR (registro de códigos de condición). El indicador V (rebose) = 1 avisa de que se ha superado el rango con signo. El indicador C (acarreo) = 1 indica del mismo hecho para los números sin signo. En seguida veremos cómo funciona esto. Lo importante es que la CPU no debe sacer cuándo el programador utiliza uno u otro tipo de numeración. Los indicadores V y C reflejan, simplemente, una condición resultante de sus operaciones. Es discrecional, por parte del programador, decidir hacer uso de ellos o no.

Si se emplean números con signo, el indicador C es irrelevante, pero el V es de importancia crucial. Si, por el contrario, se emplean números sin signo, el V no tiene importancia, mientras que el C es crucial (sólo hay una posible excepción a esta regla durante la división sin signo. Nos ocuparemos de esto en el capítulo 5).

Tanto el indicador V como el C están adaptados al tamaño de los datos que se empleen en las instrucciones. Sumando, por ejemplo, en modo byte, el indicador referirá resultados que sobrepasan el valor 255 para los casos sin signo, o el rango + 127 a −128 para los números con signo. Veamos esta situación. Primero detalladamente para el C.

Acarreo

Los registros sin signo son algo así como los cuentakilómetros de los coches: tras un cierto kilometraje vuelven al 0, perdiéndose el 1 vital (que significaría 100.000 kilómetros), con lo que se vuelve a ser el orgulloso propietario de un coche (o registro) sin casi kilómetros. El mismo resultado se obtiene sumando 1 a un registro de 32 bits que contenga el valor 4.294.967.295. El resultado correcto de la suma es:

$$4.294.967.296 = 2^{32} = 10000000000000000000000000000000$$

Sin embargo, ¡ay!, hacen falta 33 bits para este resultado, por lo que nuestro registro, haciéndolo lo mejor posible, nos da solamente los 32 bits más bajos a cero. El registro nos da un resultado de 0 en lugar del correcto de ¡4.294.967.296!

Afortunadamente, a diferencia del cuentakilómetros del coche, el bit más significativo no está perdido. Está en el bit C del registro CCR.

Este bit puede chequearse. Si está a 0, significa que no hubo acarreo; si está a 1, significa que lo hubo y podemos obrar en consecuencia. En nuestro caso, debemos comprobar que el resultado es 2^{32} en lugar de 0. (En el capítulo 6 veremos que hay instrucciones especiales de aritmética extendida para esto.)

En definitiva, el bit C es una extensión de nuestro registro de datos, con lo que las sumas son de, en realidad, 33 bits.

Sin embargo, hay un solo CCR, con lo que el único bit C debe servir para los 8 registros de datos. Si no se comprueba inmediatamente después de realizada cada operación, se corre el riesgo de que la próxima lo altere, con lo que reaparece el error de los 4.294.967.296. Para manejar estas situaciones, se dispone del misterioso bit X (extensión). Por regla general, el bit X se pone al mismo valor que el C, pero hay muchas instrucciones que alteran el bit C, pero no el X. Por el momento, podemos considerar que el bit X es una especie de memoria de C. Al detectar un acarreo, debemos realizar algunas operaciones antes de corregir la situación. Si durante ellas se pierde el bit de C, aún conservaremos el X.

Si se trabaja con palabras, se producirá un acarreo cuando la palabra menos significativa sobrepase el límite 65.535; de la misma forma, si se trabaja con bytes, se producirá el acarreo al sobrepasar el valor 255. Esto, nuevamente, es muy uniforme y favorable para el programador. El código de tamaño de datos hace muchas cosas por nosotros.

El bit de acarreo indica, también, otro tipo de peligro de la aritmética sin signo. Si se restan dos números sin signo y el resultado es negativo (por ejemplo, $1-2=-1$), el resultado será almacenado erróneamente en un registro sin signo. En este caso, el bit de acarreo indica que se ha producido un **cambio** en el límite alto de la operación de resta. Así, el programador podrá comprobar el estado del indicador de acarreo tras las restas y tomar las medidas de ajuste necesarias.

La multiplicación estándar sin signo del 68000 no precisa comprobar el estado del bit de acarreo. En este caso, la multiplicación de dos números de 16 bits sin signos nos da un perfecto resultado de 32 bits y siempre pondrá a cero (borrará) este indicador. No es posible exceder los límites.

Acarreo y extensión: Resumen

En la suma hay un indicador o bit C (acarreo) del CCR que nos previene ante las sumas sin signo que puedan exceder el límite impuesto para las operaciones con 8, 16 ó 32 bits. Tanto el C como el X (extensión) se ponen a 1 cuando se produce un acarreo, o un cambio de signo. El bit de X se conserva para usos posteriores, aunque se borre el C. El programador puede comprobar el indicador de acarreo y tomar las medidas correctivas necesarias.

Rebose

El indicador o bit V (rebose) sirve para evitar errores en la aritmética con signo. Para entenderlo, veamos un ejemplo de tal aritmética. Usaremos solamente 4 bits, pero el caso es inmediatamente extendible a 8, 16 y 32. Si sumamos 1 y −1, tendremos:

	Decimal		*Binario*
	−1	=	1111 (complemento a 2)
	+ 1	=	0001
SUMA	0	=	1000

Tenemos un resultado correcto en los bits del 0 al 3, con lo que podemos ignorar el acarreo en el bit 4. Es decir, esto explica que el C carece de importancia en la aritmética con signo. En la suma en complemento a 2, se descartar simplemente.

Ahora sumemos 6 + 7:

	+ 6	=	0110
	+ 7	=	0111
SUMA	+ 13	=	1110 ??? = −2 (complemento a 2)

Aquí obtenemos un resultado falso, a pesar de no haber acarreo. ¿Por qué está equivocada la suma en complemento a 2? La razón estriba en que + 13 está fuera del rango con signo de 4 bits (−8 a + 7).

De la misma forma, cuando sumamos dos enteros con signo en 32 bits, el indicador de acarreo no nos previene del rebose de los límites. Para prevenir contra la violación del rango comprendido entre −2.147.483.648 y + 2.147.483.647, el 68000 debe ser mucho más sinuoso. Debe comprobar los bits de signo de los dos enteros, así como los *acarreos* desde el bit 30 al 31. Los detalles carecen de importancia, con tal de comprender el resultado final: Si el indicador V es puesto a 1 en nuestra operación con signo, el resultado es incorrecto —habremos excedido los límites legales del modo en complemento a 2.

Nuestra discusión sobre los rangos de los registros, los bits de signo, acarreo y rebose, ha preparado la escena para nuestro próximo tipo de registro: los registros de direcciones.

Registros de direcciones

Los siete registros de direcciones de 32 bits, denotados por A0-A6, pueden almacenar un valor del mismo rango que los registros de datos. En-

tonces, ¿cuál es la diferencia? La diferencia reside en las subdivisiones permitidas de esos 32 bits.

Nunca se permite el modo de byte en las operaciones con registros de direcciones

Los registros de dirección están hechos para trabajar con direcciones largas de 32 bits o cortas de 16, por lo que estamos limitados a dobles palabras o palabras al manejar de A0 a A6 (y también con A7).

A pesar de que el 68000 realiza un direccionamiento basado en los 32 bits, se emplea el formato de 16, que permite circunscribir la memoria física entre límites de 64 Kbytes, cuando se puedan evitar ciclos del *bus*. El 68000 dispone de una característica propia en el manejo de las direcciones de 16 bits, denominada **extensión del bit de signo.** Veamos cómo funciona.

El modo de doble palabra de los registros de dirección es exactamente igual que para los registros de datos: están involucrados los 32 bits en la forma vista en la figura 3.4. Veamos el primer movimiento de la figura 3.9. D1 está como antes, pero ahora llevaremos la doble palabra a A0 en lugar de D2. El resultado final es el mismo: los 32 bits de D1 se copian en los de A0.

El movimiento de palabras a los registros de dirección es distinto, y la diferencia es importante. La segunda parte de la figura 3.9 muestra el efecto de llevar una palabra desde D1 hasta A0. Esta vez queda afectado todo el efecto A0 de destino, no solamente la palabra inferior. La palabra inferior de A0 se copia de la inferior de D1 de la forma usual, pero la palabra superior de A0 se carga con el valor del bit de extensión de D1.

El bit de signo al que nos referimos es el bit de signo (el 15) de la palabra inferior de D1, que en la figura es 0. Este 0 se copia en todos los bits desde el 16 hasta el 31 de A0. El resultado final es que A0, como un todo, refleja el valor y el signo de la palabra fuente, es decir, de la palabra inferior de D1. Como resultó que la palabra inferior de D1 era positiva (bit de signo = 0), se forzó a que A0 fuera también positivo, haciendo que su bit 31 fuera igual a 0.

En la figura 3.10 puede verse qué ocurre si la palabra menos significativa de D1 es negativa, con el bit de signo (el 15) igual a 1. Un movimiento de doble palabra de D1 a A0 funciona de manera normal, pero el movimiento de palabra fuerza a todos los bits de la palabra alta de A0 al valor 1. La extensión del bit de signo ha obligado a que A0 conserve su signo.

Resumen de los registros de dirección

Los registros de dirección contienen valores de direcciones completas de 32 bits, pero el trabajar con 16 bits ahorra tiempo y memoria en muchos casos. La extensión del bit de signo preserva la integridad aritmética sin necesidad de preocupar al programador con ello.

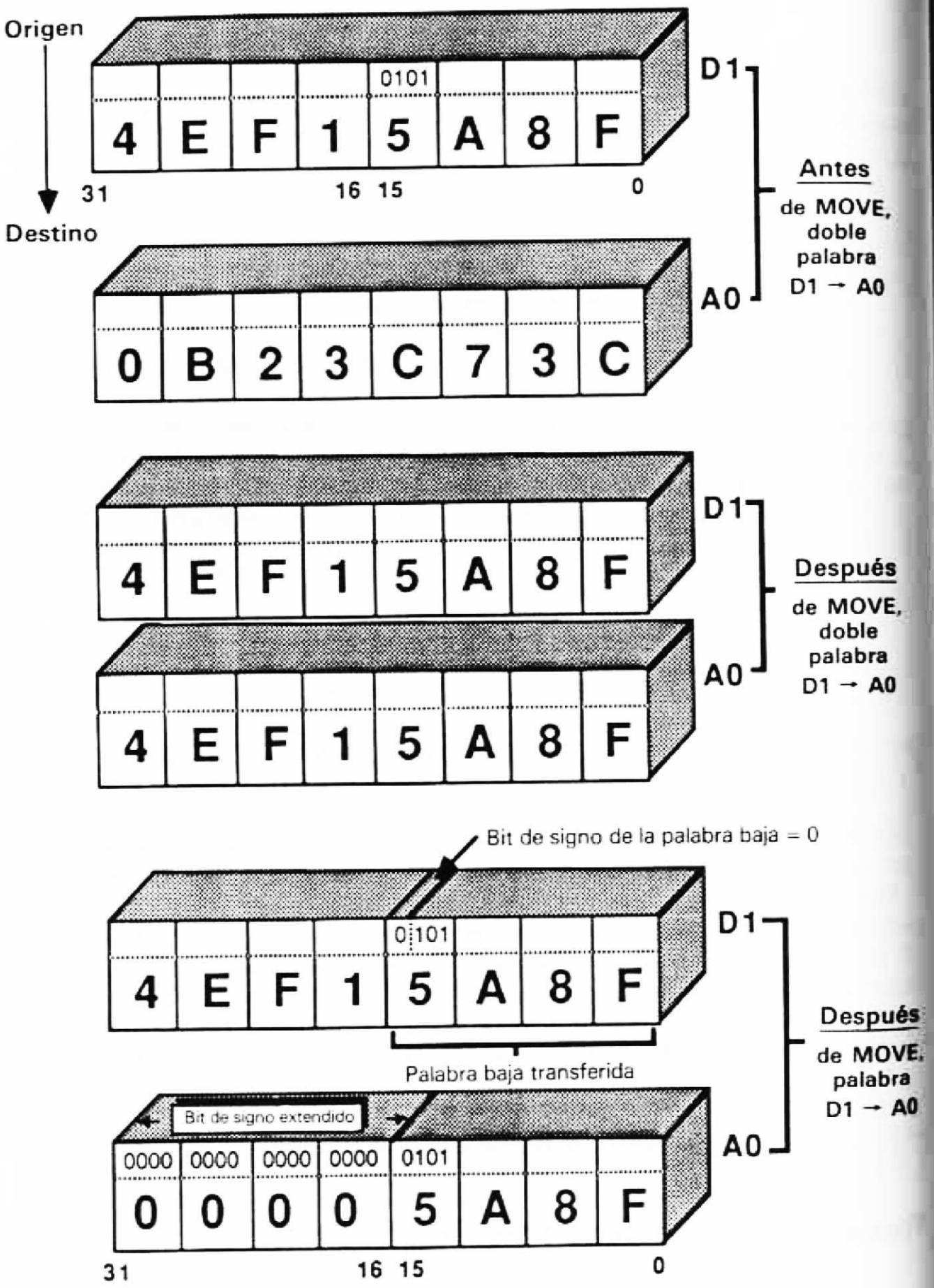


Figura 3.9
Extensión del bit de signo en operaciones de palabra con registros de dirección: bit de signo = 0

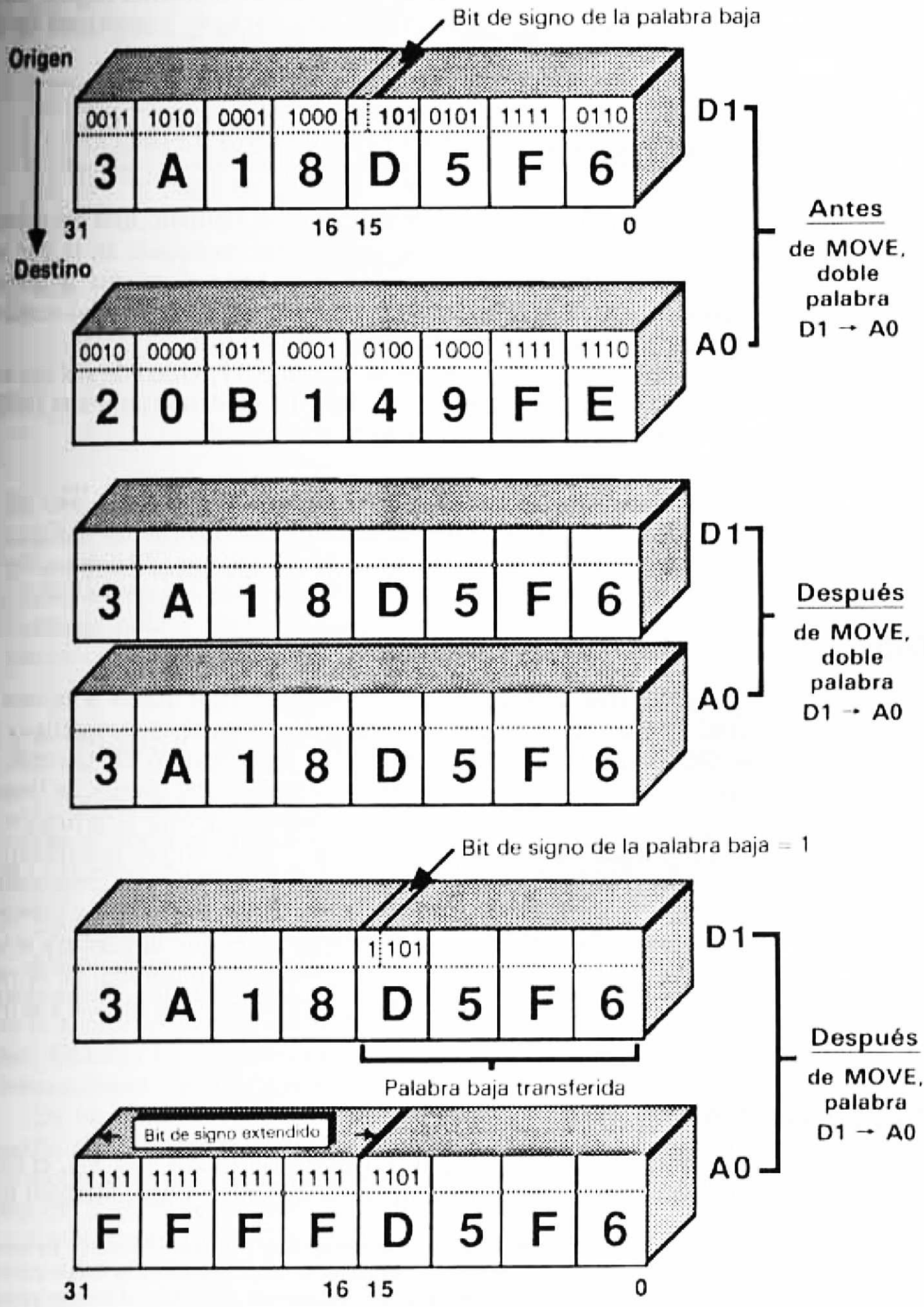


Figura 3.10
Extensión del bit de signo en operaciones de palabra con registros de dirección: bit de signo = 1

A continuación, vamos a ver el importantísimo registro SR (registro de estado), que, como vimos, tiene dos bytes de importantes datos, uno para el sistema y el otro para el usuario.

Byte del sistema

El byte más signiticativo del SR es el llamado **byte del sistema.** Su nombre se debe al hecho de que es un área protegida en la que se almacenan datos referentes al funcionamiento de todo el sistema. Puede ser leído por todos los usuarios, pero sólo puede ser escrito (alterado) cuando la máquina está en el estado privilegiado de "supervisor".

Tal como puede verse en la figura 3.11, cinco de sus bits están dispuestos de la siguiente manera (los bits 12 y 14 se usan en el 68020 solamente, y se analizan en el capítulo 8):

Bits 8-10 = máscara de interrupciones (I0, I1, I2)
Bit 15 = T (indicador de modo traza)
Bit 13 = SS (indicador de estado supervisor)

Máscara de interrupciones

Estos tres bits permiten al sistema fijar hasta 8 niveles de prioridad (0 al 7) para determinar qué interrupciones serán aceptadas y atendidas por el 68000. El nombre de máscara es significativo del concepto involucrado, puesto que así se enmascaran o prohiben los niveles de interrupción.

Los dispositivos externos pueden pedir una interrupción con un nivel comprendido entre 0 y 7 (siendo el 7 el de mayor prioridad), enviando una señal en tres bits a la CPU. Cuando la CPU haya terminado su operación presente, compara el nivel de la petición con el de la máscara. Sólo en el caso de que la petición sea de mayor nivel que la máscara se atenderá la petición de interrupción. En otro caso será ignorada. Si la máscara está al valor 0, se atenderán todas las solicitudes. Una máscara al nivel 7 significa ¡no interrumpan! [8]

Modo de traza

Cuando el bit indicador T (modo de traza) está a 1, el 68000 cae en un estado especial de funcionamiento de único paso, llamado **modo de traza.**

[8] La "máscara" de niveles de interrupción son tres elementos de memoria (biestables) cuyo valor, a 0 ó 1, codificado en binario, indica el valor de la codificación de prioridad que debe recibirse del exterior para poder retener la ejecución del programa en curso y saltar a un programa especial de tratamiento de interrupciones. La razón del nombre en español procede del concepto "pasa-no pasa" asociado a una "máscara" real, aunque su verdadero significado sería el de "nivel". Si el nivel de la petición es mayor que el nivel interno, ésta es atendida, no siéndolo en caso contrario.

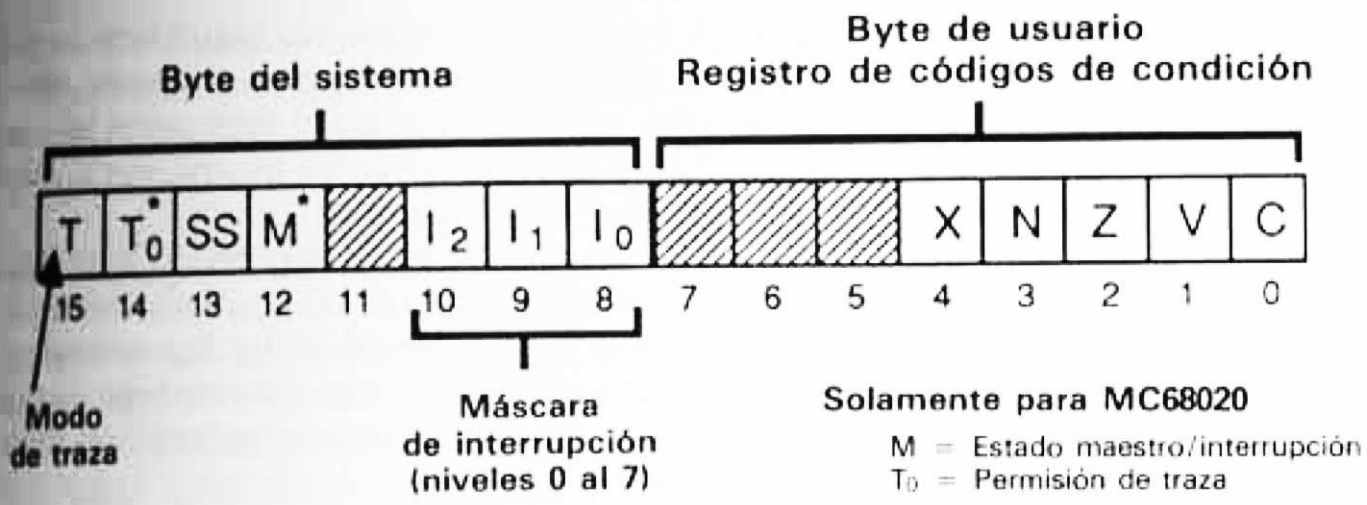


Figura 3.11
Registro de estado: byte del sistema/byte de usuario

En esta situación, la CPU ejecutará una única instrucción para, a continuación, saltar a una subrutina de comprobación suministrada por el propio usuario. El programa FIX del ordenador AlphaMicro AM-100™ es un excelente ejemplo de lo dicho. Con el FIX se puede visualizar el programa, ejecutarlo paso a paso, poner marcas y examinar el contenido de los registros en cualquier momento.

Indicador SS: Modos de usuario y supervisor

El 68000 puede funcionar en uno de dos modos (o estados) autoexcluyentes: el modo **usuario** o el modo **supervisor.** Para comprobar en cuál está en cada momento basta con evaluar el indicador SS (estado supervisor) del registro de estado. Si SS = 1, estamos trabajando en modo supervisor. Si SS = 0, estamos en el de usuario.

Un programa trabajando en modo supervisor tiene acceso a todos los recursos del sistema, incluyendo las áreas de memoria especiales, el puntero de pila supervisor y el byte del sistema, mediante instrucciones **privilegiadas.** Cuando se intenta usar estos recursos desde el modo de usuario, se desencadenan las subrutinas de error.

En un sistema típico, se conmutará frecuentemente entre uno y otro modo. De forma genérica, el *software* de las aplicaciones normales correrá en modo usuario, mientras que los sistemas operativos y los demás programas del sistema lo harán en el modo de supervisor.

Como veremos en el capítulo 6, hay muchos eventos que desencadenan estos saltos entre modos. Pueden ser programados deliberadamente o aparecer como resultado de excepciones inesperadas, llamadas *traps*[9].

La estructura de dos modos es la solución dada por Motorola al pro-

[9] Aquí, nuevamente, hemos conservado deliberadamente la expresión inglesa original por ser mucho mejor conocida que su posible traducción de "trampa" en español.

blema de la integridad del sistema de que nos ocupábamos en el capítulo 1. Mejor dicho, ofrece al diseñador del sistema un método para aumentar la seguridad del equipo. En particular, se pueden proteger las zonas de la memoria dedicadas al sistema operativo contra incursiones involuntarias o deliberadas de los programas de los usuarios.

El estado no se indica solamente en el *flag* SS del byte del sistema, sino que también es comunicado a todos los dispositivos exteriores a través de los tres terminales FC (control de función) del *bus* de control. Volviendo a la figura 1.3, podemos ver de qué forma estas señales permiten a la unidad de manejo de memoria establecer segmentaciones del total en memoria de usuario y de supervisor.

Resumen del modo supervisor

El modo supervisor permite el acceso a todos los recursos y, además, ofrece privilegios y recursos adicionales al diseñador del sistema para poder conseguir mayor eficacia y seguridad. Los programas normales correrán en modo usuario, pero las interrupciones, los *traps* y las excepciones son procesadas en modo supervisor.

Punteros de pila

Una pila es, sencillamente, una porción de memoria con un puntero que le permite **introducir** *(push)* datos o **sacar** *(pull),* según un esquema LIFO (el último en llegar es el primero en salir, *"Last In, First Out"*). En el capítulo 5 veremos con detalle cómo el 68000 maneja con facilidad las pilas con instrucciones de MOVE, que provocan aumentos y decrementos de los punteros.

Las pilas se utilizan, por regla general, para salvar toda clase de parámetros y palabra de estado cuando se salta a realizar otras tareas, tales como subrutinas, de las que, a su vez, se saltará en lo que se conoce como **anillamiento.** Incluso si se dispone de muchos registros para salvar y recuperar los datos durante los anillamientos, la pila es más útil para el programador, puesto que la secuencia de recuperación es la inversa, sistemáticamente, de la de almacenamiento.

Hay que tener cierto cuidado para no confundirse con la terminología de los punteros de pila del 68000. Cada cual es totalmente libre de crear sus propias pilas empleando el registro de dirección que más le convenga; sin embargo, estas pilas caen bajo su exclusiva responsabilidad.

Por su parte, el 68000 mantiene dos **pilas del sistema:** la pila del sistema de usuario (activa sólo en modo usuario) y la pila del sistema supervisor (activa sólo en modo supervisor). Algunas instrucciones del 68000 hacen un uso implícito de las pilas del sistema, mientras otras permiten que el programador las utilice mediante el uso del nemónico SP, o **puntero del sistema.** De hecho, SP es otro nombre de A7. Dado que no se puede estar simultá-

neamente en ambos modos, ambos punteros se indican con SP (= A7). Recuérdese, en todo caso, que el significado del SP viene determinado por el modo de trabajo del 68000 en cada momento.

Registro de códigos de condición

El registro único de 8 bits, CCR (registro de códigos de condición), forma el byte inferior (bits del 0 a 7) del SR (registro de estado).

Los 5 bits inferiores del CCR, como hemos visto, se emplean para señalar varias condiciones, resultado de las operaciones aritméticas y lógicas. Los tres más altos no se emplean actualmente.

Los cinco indicadores de condición (llamados a veces indicadores de estado) se designan por:

Bit	7	6	5	4	3	2	1	0
Indicador				X	N	Z	V	C

Donde:
X = eXtensión
N = Negativo
Z = Cero (en inglés, "Zero")
V = Rebose (en inglés, "oVerflow")
C = Acarreo (en inglés, "Carry")

En la anterior sección de los registros de datos analizamos los indicadores X, V y C. Los restantes indicadores son sumamente sencillos. El Z se pone a 1 si el resultado de la última instrucción ha sido 0, pasando a 0 en caso contrario. Un resultado distinto de cero obliga a que Z = 0. Un resultado cero hace Z = 1. El indicador N coincide con el MSB (bit más significativo), que también hemos llamado bit de signo. Así, en aritmética con signo, N = 1 para resultados negativos y N = 0 para resultados positivos o cero.

Conclusión

En este capítulo hemos desarrollado una visión más detallada de las instrucciones del 68000. En el capítulo 4 veremos los códigos más frecuentes de operación y, mediante ejemplos, veremos cómo funcionan y cómo emplearlos.

4

El set de instrucciones del 68000: Primeras etapas

"Las instrucciones crueles infectan al inventor al charlar."

(SHAKESPEARE, *Macbeth* I, vii)

El capítulo 4 nos explicará qué es una instrucción, introduciéndonos en las más sencillas y frecuentes del 68000. Los programas que se ofrecen como ejemplos no tratan, en absoluto, de ser completos o prácticos o listos para funcionar, aunque hemos tratado de hacerlos interesantes y directamente relacionados con el mundo real.

Instrucciones

En lo que al 68000 se refiere, una **instrucción** es un conjunto de palabras de 16 bits situadas en la memoria, y un **programa** es una secuencia de tales instrucciones, que llevará con éxito al sistema a la realización de algún trabajo útil.

Para las personas no preparadas, las instrucciones escritas a nivel de código máquina forman una desconcertante secuencia de unos y ceros, pero en cuanto son leídos y codificados en el *chip*, estas instrucciones son obedecidas de acuerdo con unas reglas muy precisas, constituyendo lo que llamamos **correr** un programa.

En este capítulo desmenuzaremos cada instrucción del 68000 para ver

cómo funciona y por qué se emplea. Mediante diagramas antes-y-después, ilustraremos el aspecto funcional de las instrucciones. El dónde y el cuándo usarlas es parte del arte de creación llamado programación en lenguaje máquina, de cuyas infinitas posibilidades sólo podemos insinuar un mínimo mediante ejemplos aislados de programas concretos.

No hay nada que pueda reemplazar la práctica propia: el apéndice E (recursos del 68000) ha sido preparado para que usted pueda comprender el *hardware*, el *sotfware* y cualquier documentación que pueda necesitar.

Si fuéramos criaturas binarias puras, podríamos referirnos a las instrucciones como "0111001000000001" (que realmente indica al 68000 que ponga el dato "1" en el registro de datos D1), o bien "0101111010010010" (que sumará un 7 a un registro de memoria), etc. Sin embargo, la vida es demasiado corta y los humanos no estamos hechos para comunicarnos así. Introducir un 1 o colocar erróneamente un 0 puede tener consecuencias desastrosas para el significado de una instrucción. Aún peor, algunas instrucciones tienen una o más palabras de **extensión** y pueden necesitar hasta 80 bits para ser totalmente escritas. Imaginemos lo que sería tener que aprender Morse con... ¡150 millones de códigos diferentes!

La forma razonable de trabajar consiste en disponer de símbolos reconocibles del inglés para cada instrucción, lo que, de hecho, constituye la manera en que los programadores en lenguaje ensamblador piensan, escriben y hablan sobre las instrucciones. Los símbolos empleados se llaman **mnemónicos**, puesto que nos ayudan a recordar qué hace cada instrucción.

Para los dos ejemplos antes citados, tenemos que para el código máquina 0111001000000001 su escritura es:

```
MOVEQ #1,D1
```

mientras que para el 0101111010010010 su escritura es:

```
ADDQ.L #7, (A2)
```

Incluso las versiones en inglés pueden parecer extrañas, pero debemos admitir que tienen mejor "aspecto" que las de unos y ceros. A medida que avancemos, irán apareciendo más hermosas y precisas las instrucciones con toda claridad —lo que hacen y cómo pueden combinarse para generar programas que funcionen es, después de todo, la razón para tratar de comprender sus secretos.

Debemos recordar siempre que el 68000 mismo nunca "verá" los códigos mnemónicos propiamente dichos. Se trata simplemente de una ayuda al aprendizaje y trabajo humanos. Como vimos en el capítulo 3, nuestras instrucciones simbólicas deben ser traducidas (ensambladas) al código máquina binario antes de que puedan ser ejecutadas. Las configuraciones binarias reales de cada instrucción se muestran en el apéndice D. Cada cual es libre de aprendérselas de memoria si quiere, pero seguimos pensando que la solución es un buen ensamblador de mnemónicos.

Formato de las instrucciones

Cada instrucción es una frase del inglés. De la misma forma en que se debe conocer el exacto significado de cada palabra, así deberemos conocer las reglas gramaticales o de sintaxis que impidan hacer combinaciones ilegales o carentes de sentido. Por ejemplo, "mordió perro hombre" contiene tres palabras perfectamente comprensibles, pero el conjunto es casi incomprensible. Así como los idiomas de cada país varían de unas regiones a otras levemente, veremos que hay distintas formas de escribir las instrucciones del 68000 mediante versiones levemente distintas con formatos algo diferentes para cada una de ellas. Afortunadamente, hay un estándar natural y, no sorprendentemente, fue inventado y promocionado por Motorola —razón por la que haremos un uso extensivo del mismo. Si usted observa alguna desviación en su máquina, ya sabe a quién debe reclamar.

Puesto que nuestro objetivo es aprender las reglas básicas del set de instrucciones del 68000, evitaremos todos los tecnicismos del ensamblador, adoptando la versión de Motorola del siguiente plan "vainilla".

Sintaxis de las instrucciones

Hay tres posibles disposiciones de las instrucciones del 68000:

1. Sin operandos: El código de operación es autoexplicativo.
2. Con un operando: Código de operación, seguido del operando.
3. Con dos operandos: Código de operación, seguido en primer lugar del operando fuente y después por el de destino.

El **código de operación** es un mnemónico, tal como JMP, MOVE, ADD, SUB, que nos indica lo que hace la instrucción. El 68000 dispone de alrededor de 60 códigos de operación básicos, y muchos de ellos tienen hasta ¡500 variaciones posibles! Incluso los más experimentados programadores del 68000 no pueden recordar todas estas posibilidades. En su lugar, lo más lógico es tratar de comprender los principios rectores de funcionamiento y consultar las hojas de referencia cuando se presenta alguna duda sobre una instrucción en particular. Por ello, hemos arreglado varios apéndices útiles que interrelacionan los códigos de operación de diversas formas para procurar ayudarle [1].

Los **operandos** son registros o posiciones de memoria que contienen el elemento sobre el que actúa el código de operación. Siguiendo con nuestro símil del idioma, el código de operación puede ser considerado como el

[1] Además de estas hojas, uno de los elementos más útiles para la programación es la llamada vulgarmente (y con mucho sentido) "chuleta", que contiene un resumen de dichas hojas y de la que el libro contiene un ejemplar. Por otra parte, no debe olvidarse que aquí las frases emplean verbos y complementos en inglés, tales como ADD, etc.

verbo, mientras que los operandos serían los complementos directos e indirectos de la frase.

Para empezar a conocer los formatos de las instrucciones, veamos algunos ejemplos, aunque sin profundizar demasiado en un esquema de funcionamiento.

Sin operando

RTS — El código de operación es *ReTurn from Subroutine* (regreso de subrutina). No precisa de operando para saber qué hay que hacer.

Con un operando

ASL.W (A0) — El código de operación es *Arithmetic Shift Left. Word.* (desplazamiento aritmético a la izquierda. Tamaño de palabra). El operando único indica al 68000 dónde encontrar el dato a desplazar, en este caso la palabra situada en la dirección marcada por el contenido del registro A0.

Con dos operandos

MOVE.B D3,D4 — El código de operación es MOVE. Byte (mover un dato de tamaño de byte), que precisa de dos operandos, es decir, aquí del operando **fuente**, el registro de datos D3, y del operando **destino**, el registro de datos D4. Este ejemplo indica que hay que llevar el byte bajo de la fuente (origen) al byte bajo del destino, esto es, *desde* el de origen *hasta* el de destino.

Dado que la mayoría de las instrucciones corresponde al tipo de dos operandos, revisemos esta disposición con más detalle (véase la figura 4.1). Independientemente de qué dos operandos se trate, siempre habrá una coma separándolos. El operando origen (fuente) aparece en primer lugar siempre; esto se debe a que es ahí donde la instrucción toma el dato. El destino aparece tras la coma, e indica dónde puede encontrarse el resultado de la operación.

Los operandos origen no son alterados por las instrucciones
Los operandos destino son alterados

Así, MOVE.B D3,D4 no cambia el contenido de D3 y sustituye el byte bajo de D4 por el de D3. Más adelante veremos algunos ejemplos particulares de MOVE y sus posibles variaciones.

He aquí otro ejemplo de una instrucción muy empleada llamada ADD.

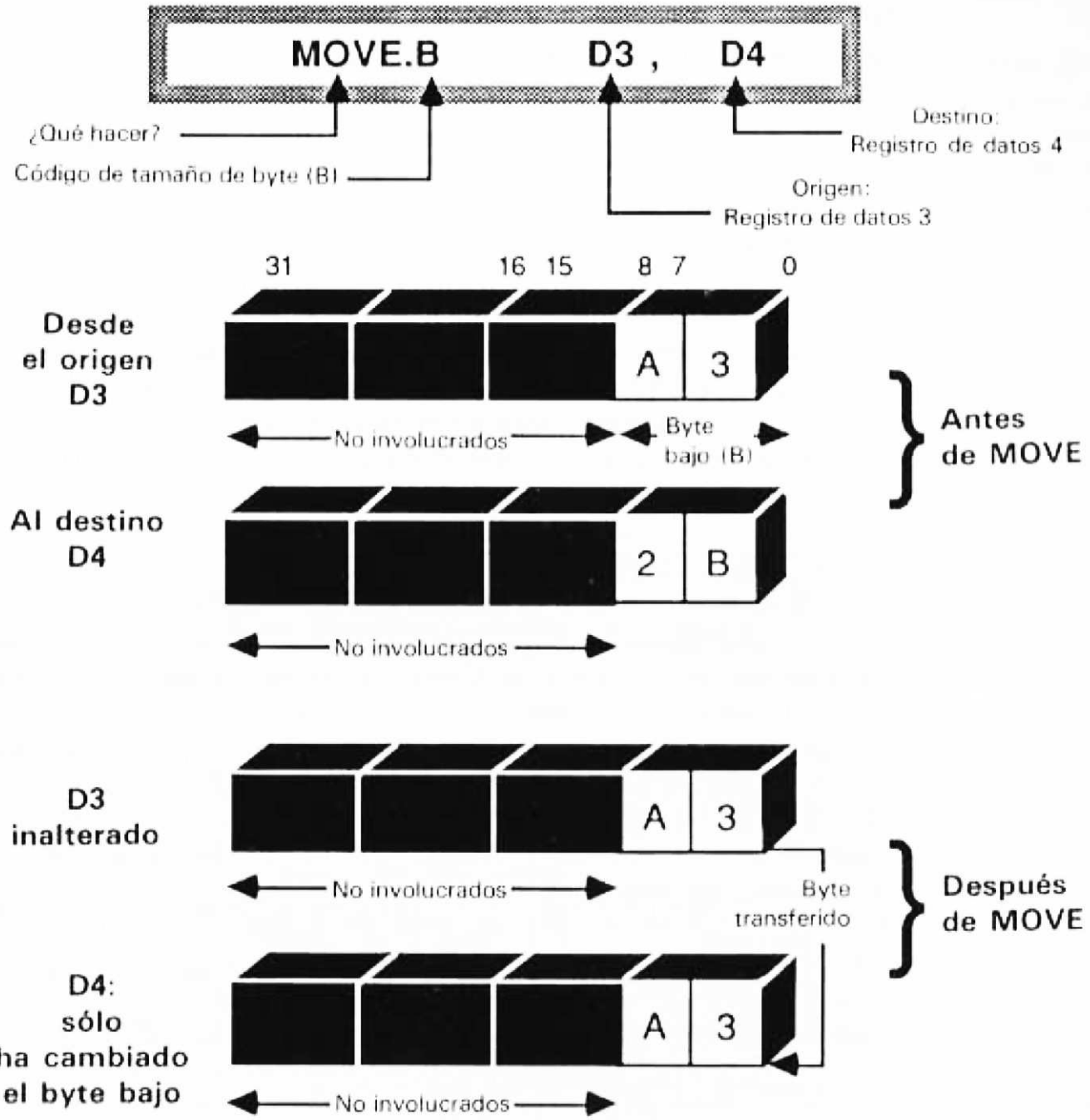


Figura 4.1
Instrucción con dos operandos: MOVE.B D3,D4

ADD.L D6,D7 — ADD. Doble palabra indica: sumar (adicionar) los 32 bits del origen, D6, a los 32 bits del destino, D7, y colocar el resultado en el destino, D7. Vemos, otra vez, que el destino es el receptor del resultado de la operación.

Códigos de tamaños de los datos: L, W, B

En los anteriores ejemplos hemos visto que algunos códigos de operación llevan aparejada una letra. A ésta se la denomina **código de tamaño**

del dato, porque indica cuántos bits del origen (fuente) y del destino están involucrados en la operación. Las reglas son muy simples y se aplican tanto para datos en registros como en memoria:

L significa doble palabra: opera sobre los 32 bits.
W significa palabra: opera sobre los 16 bits.
B significa byte: opera sobre los 8 bits inferiores.

La mayor parte de las instrucciones puede funcionar con los tres tipos de datos, por lo que en adelante emplearemos una notación simplificadora, por ejemplo ADD.z, donde "z" puede representar L, W, o B.

Ahora ya sabemos bastante para poder escribir un programa real, por lo que vamos a presentarle una situación hipotética con que comprobar sus bríos de programador.

APLICACION PRACTICA

Problema: Estamos corriendo un programa de nóminas a fines de marzo y necesitamos actualizar las horas trabajadas (YTD ; year-to-date) en el presente año, incluyendo las del mes de marzo. Emplee ADD y MOVE y coloque el resultado en el registro de datos D3.

Datos:

1. El número de horas YTD trabajadas en enero y febrero están en el registro D1.
2. Las horas trabajadas en marzo están en el registro de datos D2.

Caso práctico:

Horas YTD = 320 en D1
Horas en marzo = 138 en D2
Nuevas horas YTD = 458 a colocar en D3

Solución: Programa 4.1

```
MOVE.L  D1,D3
ADD.L   D2,D3
```

Ahora, D3 contiene la suma (D2 + D1) = 458. D1 y D2 permanecen inalterados.

Tamaño de los datos y rango de los resultados

La instrucción ADD realiza una interesante adición binaria, dependiendo de usted el decidir cuándo los resultados tienen un signo y cuándo no. En este pequeño programa estamos tratando con pequeños números posi-

tivos totalmente comprendidos en el rango de un registro de 32 bits, por cuanto la cuestión es redundante.

Merece la pena notar que el 68000, *usualmente*, tiene más trabajo que hacer cuando realiza una operación en 32 bits (.L) que cuando lo hace en 16 (.W). De forma similar, las operaciones en 16 bits (.W), *usualmente*, precisan de más pasos internos de la CPU que las variantes de 8 bits (.B). Si alguien pregunta: ¿cuán rápida es una instrucción de ADD en el 68000?, lo único que podemos ofrecer como respuesta es un rango de valores (casos de mejor y peor) para las posibilidades .B, .W, y .L. Incluso estos datos dependerán del modelo de *chip* y de su reloj (que, tipicamente, varia entre 125 y 60 nanosegundos). No, en realidad no estamos esquivando la pregunta, ADD puede, en realidad, ocupar entre 0 (sí, CERO) y 30 ciclos de reloj, dependiendo de "dónde, cuándo y qué" estemos sumando. El milagro efectivo de cero ciclos es, en resumen, consecuencia del solapamiento de instrucciones que realiza el 68020.

En cualquier caso, no existe una forma sencilla para determinar la cronología de las operaciones .B, .W, y .L. Por ejemplo, rara vez las operaciones en .W ocupan realmente el doble que sus versiones en .B; sin embargo, el sentido común nos indica que cuando podamos escoger el tamaño del dato, deberemos optar por el menor posible que cubra nuestras necesidades.

Desde luego, en el ejemplo anterior podrían haberse ahorrado algunos ciclos de reloj empleando MOVE.W y ADD.W, puesto que los números cabían en el rango de las operaciones de 16 bits (con y sin signo). Sin embargo, no hubiera funcionado con MOVE.B y con ADD.B, puesto que los valores sobrepasan el límite de 255 (11111111), correspondiente al máximo dato sin signo permitido en un byte. La situación más común en la práctica es trabajar con números con signo (en complemento a 2), de forma que el programa anterior, por ejemplo, permitiría valores negativos en D2 para ajustar las horas de YTD (ADD un negativo es SUBstraer).

A medida que se vaya desarrollando el programa de las nóminas, se irán encontrando valores con distintos rangos y signos, por lo que será necesario usar la letra de tamaño apropiada para cada caso. También veremos cómo utilizar el CCR (registro de códigos de condición) para comprobar el correcto funcionamiento de nuestra aritmética.

Comentarios

Una interesante posibilidad ofrecida a todos los niveles de programación es la capacidad de añadir títulos, fechas, números de revisión, comentarios, notas y recordatorios en el texto del programa para uso personal solamente. Tales comentarios son ignorados por los ensambladores y compiladores, pero pueden resultar notablemente útiles cuando usted (o sus colegas) vayan a leer su listado de programa varios meses después y se pregunten: "¿qué ocurría aquí?", o "¿por qué hacía yo esto?"

La adición de comentarios breves es una buena costumbre, especialmente para los lenguajes de bajo nivel, donde el propósito de cada línea no re-

sulta evidente. Los convenios de comentarios de Motorola son sencillos. Permiten tanto líneas individuales como comentarios en la propia línea de la instrucción ("on line"), como sigue:

```
* En cualquier sitio indica una linea de sólo comentario.
```

de forma que el ensamblador ignora todos los caracteres que siguen al * (asterisco) hasta que se lea la siguiente línea de código fuente. De forma alternativa, se puede incluir un comentario a la derecha de cualquier línea de programa:

```
<CODIGO OP> <operando(s)>     Aquí el comentario de la linea
```

para lo que basta con incluir un espacio (o tabulación) entre los operandos y el comentario.

Desarrollemos estas buenas costumbres añadiendo algunos comentarios al primer programa de nuestro ejemplo.

```
* Programa 4.1 con comentarios
* Actualización YTD 4 rev. 1 SKB

        MOVE.L  D1,D3    Antiguo YTD ahora en D3 (32 bits)
        ADD.L   D2,D3    Suma las horas de marzo al viejo YTD

* Ahora, D3 contiene las horas actualizadas del YTD
```

CCR: El registro de códigos de condición

En el capítulo 3 vimos de qué forma los indicadores del CCR supervisaban varias condiciones del procesador, alertándole frente a posibles errores. Es importante conocer de qué forma cada código de operación afecta a los cinco indicadores del CCR, para lo que emplearemos la siguiente notación:

Indicador del CCR	X	N	Z	V	C
MOVE	—	*	*	0	0
ADD	= C	*	*	*	*

Donde:

—	indica no alterado
0	indica siempre puesto a 0 (borrado)
1	indica puesto siempre a 1
*	significa que se pondrá a 0 o a 1 según el resultado de la instrucción
U	significa indefinido, es decir, el indicador no contiene información útil
= C	puesto a 0 o a 1, según esté el indicador C

En la figura 4.2 se muestran cómo cambian los registros y el CCR durante la ejecución del programa 4.1. Adornemos nuestro programa comprobando el estado del CCR. Esto impondrá algunas nuevas instrucciones para **ramificaciones** *(branching)*[2] y una construcción conocida como **etiqueta** para indicar el lugar al que deseamos que el programa se dirija.

```
* Programa 4.2: Comprobación del CCR
* Actualización YTD 5 rev. 2 SKB
          MOVE.L  D1,D3       Antiguo YTD, ahora en D3 (32 bits)
          ADD.L   D2,D3       Suma las horas de marzo al viejo YTD
* Ahora D3 contiene las horas YTD actualizadas
          BVS     ERROR.1     Salta a la etiqueta ERROR.1 si hay rebose
          BEQ     IDLE        Salta a la etiqueta LABEL si es cero
          <resto del programa>
          *       *       *
          BRA     OVER        Salta siempre a OVER
ERROR.1   <tomar acción: rebose de D3>   Se detectó error
          *       *       *
          BRA     OVER        Salta siempre a OVER
IDLE      <tomar acción: horas YTD = 0>   D3 = 0, Posible error
          *       *       *
OVER      <programa de finalización>
```

Ramificaciones y etiquetas

Tal y como prometimos, nuestro revisado programa introduce dos conceptos nuevos, directamente interrelacionados: ramificaciones y etiquetas. Salvo que se le indique otra cosa, el procesador avanzará secuencialmente de una instrucción a la siguiente, quizá de forma parecida a esto:

1. Tomar la dirección del PC (contador de programa).
2. Leer la primera palabra de la instrucción en la dirección de memoria apuntada por PC.

---

[2] Ahora hemos optado por la traducción más correcta al español, puesto que este es un caso extremo de deformación de nuestro idioma. En efecto, la expresión original inglesa de ***branching*** ha provocado la "aparición" del término "brancheo", que juzgamos fuera de lógica a todas luces y excesivamente forzado, a pesar de que ya forma parte de la jerga profesional al caso. En cualquier caso, su significado de derivación desde la tarea en ejecución en un momento dado se corresponde con la acepción de **ramificación**, clásica en nuestros manuales y tratados. A partir de ahora, el nemónico BRA y sus derivados deberán evocarnos este significado.

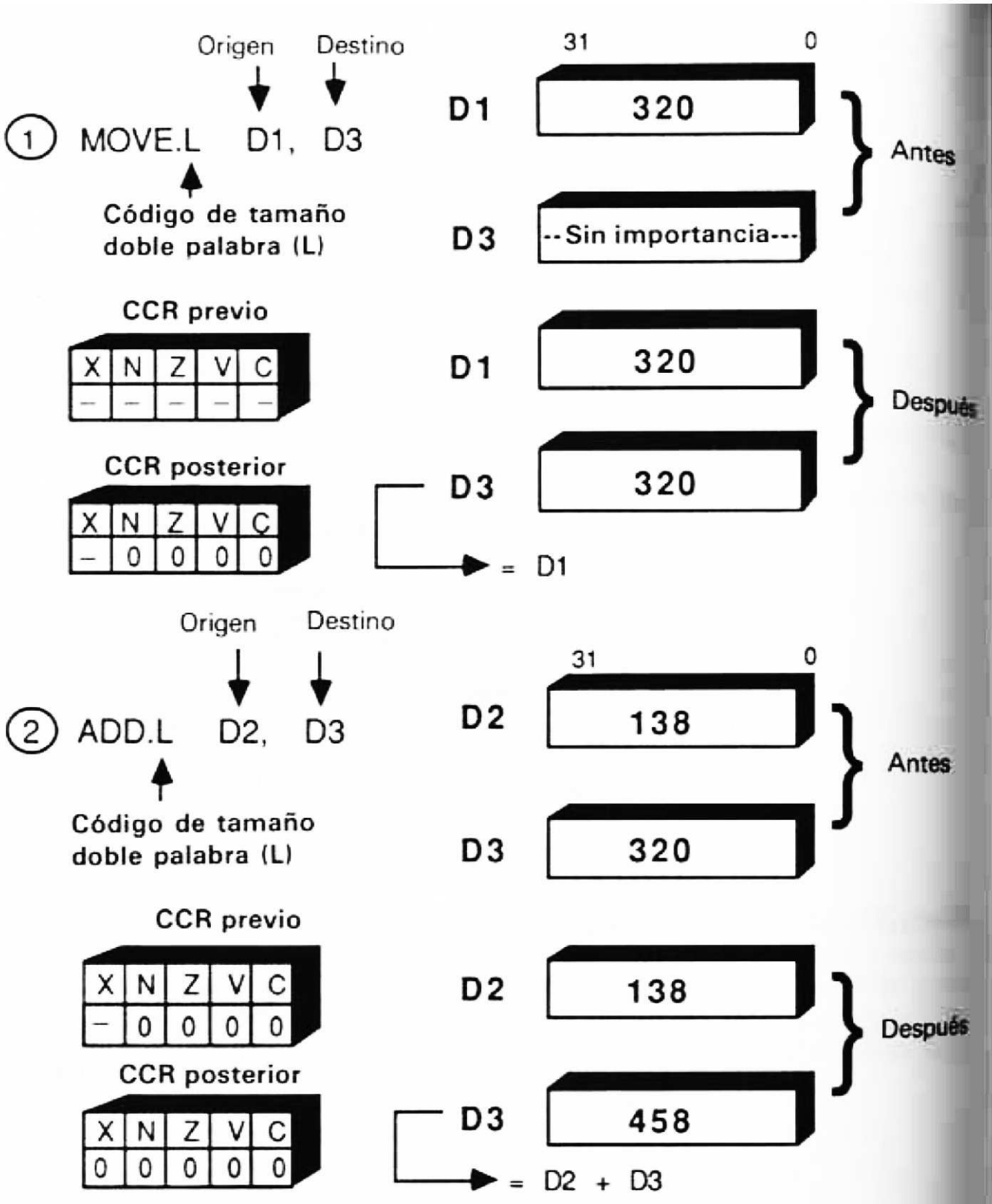


Figura 4.2
Actualización de horas YTD

3. Decodificar esta palabra-instrucción y, si es necesario, leer las palabras adicionales (o extensiones) precisas para poder decodificar la instrucción completa.
4. Ejecutar los pasos necesarios para completar la instrucción en curso.
5. Incrementar el valor del PC para apuntar a la próxima instrucción.
6. Repetir todo el ciclo desde el punto 1.

Esta secuencia lineal de lectura-ejecución de la instrucción (véase la figura 4.3) puede alterarse tanto de forma incondicional como bajo condiciones, mediante el uso de las instrucciones de ramificación y de las etiquetas.

Ramificación incondicional: BRA etiqueta

La instrucción BRA *(BRanch Always:* saltar siempre) es una ramificación incondicional, y su efecto es tan dramático como que altera el punto 5 de la secuencia normal antes desarrollada. Lo que ocurre es que el contador de programa PC, en lugar de incrementarse para apuntar a la siguiente instrucción, se carga con una nueva dirección que depende de la etiqueta empleada, y se pasa el control del programa a la línea de programa afectada de dicha etiqueta. Así, el código BRA OVER de nuestro ejemplo provoca el salto del programa a la línea marcada con OVER —y desde entonces se vuelve a seguir la secuencia normal de ejecución desde el punto 1 al 5, de forma normal, por lo menos hasta que encontremos otra orden de ramificación.

Veamos a continuación de qué manera se altera la secuencia normal al encontrar la instrucción BRA:

1. Tomar la dirección del PC (contador de programa).
2. Leer la primera palabra de la instrucción en la dirección de memoria apuntada por el PC.
3. Decodificar esta palabra de la instrucción y, si es necesario, leer las palabras adicionales (o extensiones) precisas para decodificar la instrucción completa.
4. Si la instrucción es BRA OVER, el punto 3 habrá generado un número con signo, llamado **desplazamiento** de la ramificación, que depende de la posición de la etiqueta OVER en el listado del programa.
5. **Adicionar** este desplazamiento al valor del PC y devolver la **suma** al PC. Ahora, el PC está apuntando a la instrucción marcada con OVER.
6. Ahora, el procesador está ya preparado para proceder al punto 1 anterior, donde leerá, decodificará y ejecutará la instrucción marcada con OVER.

Podemos ver que los puntos del 1 al 3 son siempre los mismos, pero que SI en el punto 3 se decodifica una instrucción de BRA, en lugar de incrementarse para apuntar a la siguiente instrucción; el PC se autoajustará para apuntar a la línea marcada con la etiqueta OVER (véase la figura 4.4). Por el momento, no debemos preocuparnos demasiado por la forma en que se calcula el desplazamiento de la ramificación para conseguir el valor correcto que añadir al PC para apuntar exactamente a OVER. Los principales puntos a tener en cuenta son:

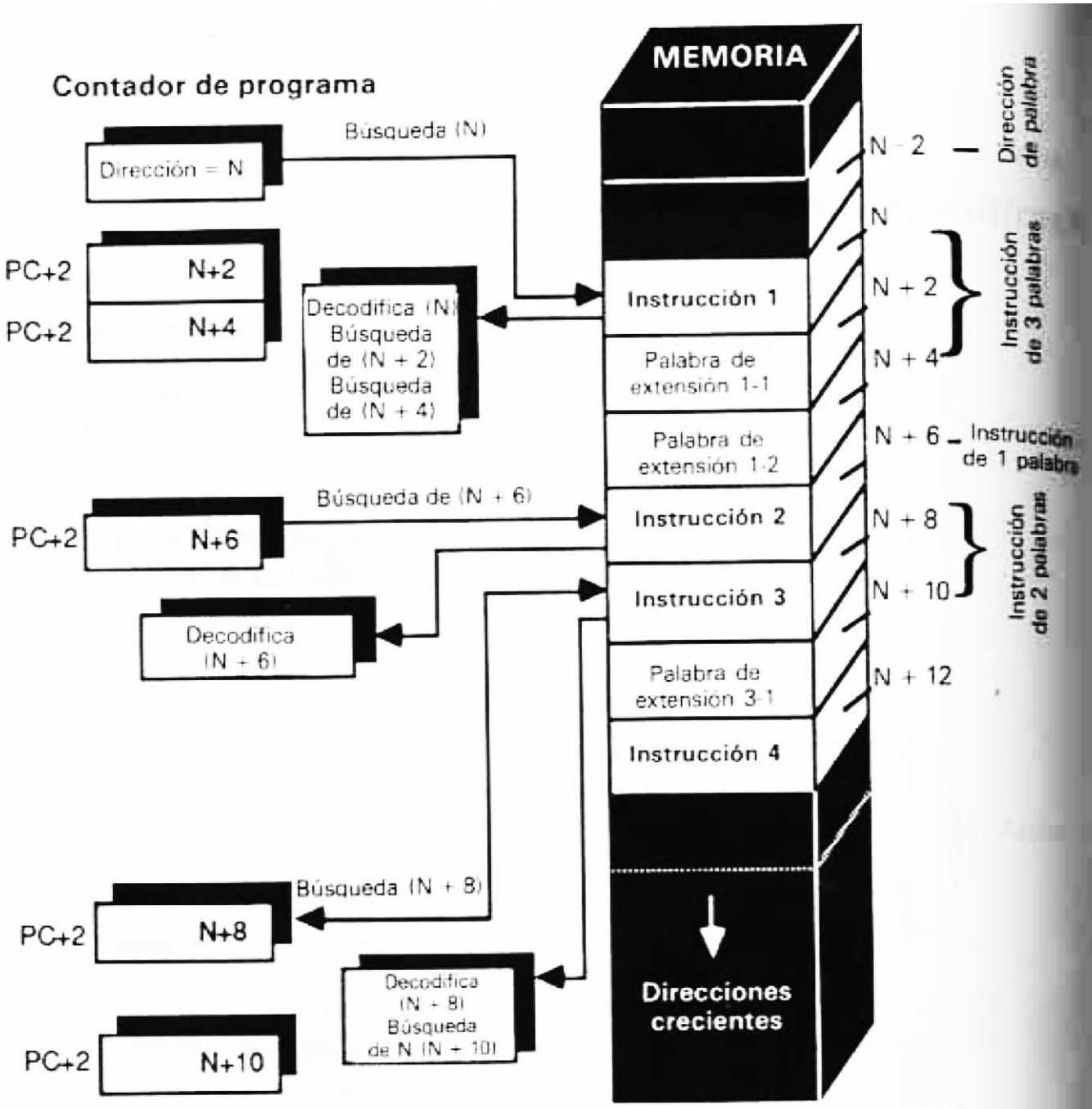


Figura 4.3
Contador de programa (PC) y la secuencia de instrucciones

- La ramificación supone una ruptura en la secuencia normal de "pasar a la siguiente instrucción" mediante el ajuste del valor de la dirección contenida en el PC.
- Esencialmente, las etiquetas son **direcciones de instrucción** que el sistema interpreta de forma que el nuevo valor del PC es ahora (PC + desplazamiento), dándonos la dirección de la instrucción situada en la línea etiquetada.
- Dado que el valor del desplazamiento tiene signo (positivo o negativo), el salto podrá hacerse hacia delante o hacia detrás. Si la etiqueta está situada más adelante en el programa, más allá de la línea del **BRA**, el desplazamiento será positivo, incrementando el PC para poder saltar

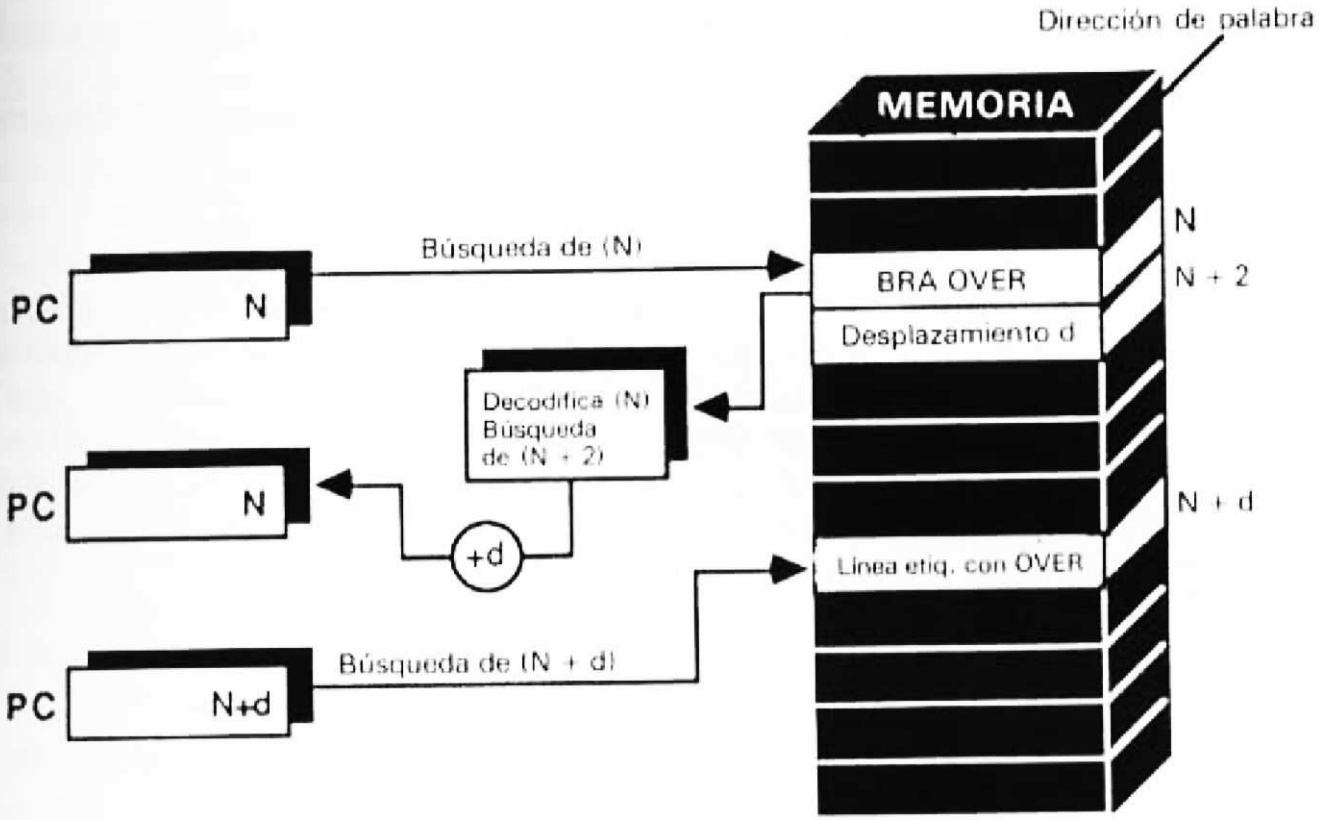


Figura 4.4
El contador de programa y la secuencia de la instrucción de ramificación

hacia delante. Si la etiqueta aparece antes que la línea del BRA, desplazamiento será negativo, con lo que el PC se decrementará par. ramificarse hacia atrás.

Los lectores que sepan BASIC reconocerán la similitud entre el BRA y el GOTO. La gran diferencia está en el uso de etiquetas con nemónicos alfabéticos, en lugar de números de líneas empleados en el BASIC. Tal como hemos indicado, entre bastidores, la etiqueta OVER actúa de forma muy parecida al número de línea. Al ensamblar, cargar y ejecutar el anterior programa, la etiqueta OVER es traducida a la dirección de memoria de esa instrucción en particular, y es esa dirección la que se coloca en el PC al decodificar la instrucción BRA OVER. La manera en que esto se hace será analizada con mayor detalle cuando hablemos del direccionamiento absoluto y del relativo.

El programador es libre de emplear cualquier conjunto razonable de caracteres como etiquetas, pero, naturalmente, deberá procurarse siempre evitar el duplicar etiquetas con el mismo nombre en un mismo programa. Además, deben respetarse las reglas de la sintaxis: las etiquetas han de colocarse al extremo izquierdo de las líneas, y hay que disponer de espacios, o tabulaciones, entre las etiquetas y los códigos de operación.

La ramificación incondicional no es muy útil. Veamos ahora el cómo y el porqué de la condicional, para así poder entender el programa 4.2.

Un total de 14 instrucciones de ramificación de estructura

```
Bcc   <etiqueta>
```

le permiten efectuar la ramificación solamente si se cumple la condición indicada por el código "cc". Las dos letras "cc" son mnemónicos referidos a condiciones particulares de los indicadores del CCR, que el 68000 comprobará antes de decidir si se debe efectuar la ramificación o no. En caso negativo, se ejecutará la siguiente instrucción. Si se satisface la condición estipulada, el 68000 salta a la línea marcada con la etiqueta, empleando la misma estrategia que la descrita para el BRA.

Así, Bcc <etiqueta> puede leerse como:

Si se cumple cc, hacer un BRA <etiqueta>.
Si no se cumple cc, ejecutar la siguiente instrucción.

La comprensión de Bcc es, aproximadamente, el 150 por 100 del camino al dominio del 68000

¿Qué clases de condiciones pueden comprobarse con cc? En el programa 4.2 se analizan sólo dos casos sencillos en los que solamente se comprueba un único indicador del CCR. Para BVS, se hace cc = VS, con lo que se comprueba el valor del indicador V: ¿Está a 1 o no? Para BEQ, se toma cc = EQ, de forma que se observa el indicador Z. ¿Es el resultado cero o no? Es decir, ¿es Z = 1 o no? BVS ERROR.1 significa lo siguiente: Se ramificará (a la línea marcada con ERROR.1) solamente si el indicador de rebose (V) está a 1. Si la suma de D2 y D3 provoca V = 1 (rebose) es que la respuesta contenida en D3 es errónea, y deberemos hacer algo al respecto. Por ello, se comprueba el estado del indicador V mediante BVS. Si V = 0, todo va bien (por ahora), y no necesitamos ramificar el programa. Si V = 1, ha habido rebose y usted ha tenido el valor y la prudencia de prepararse contra ello con BVS.

La parte del programa que se inicia en la línea marcada con ERROR.1 es la encargada de ocuparse del hecho de que ha habido rebose y de que el resultado almacenado en D3 es aritméticamente incorrecto para números con *signo*. Para nuestro ejemplo, eso significaría que las horas YTD han sobrepasado el rango de −2.147.483.648 a + 2.147.483.647, lo que indica la presencia de errores, bien en los datos de partida, bien en el bloque del programa. Así, BVS puede ser también un buen método de detectar errores durante el desarrollo y prueba de los programas, pero, de manera primordial con números grandes (o potencialmente grandes), BVS es una llave esencial contra errores.

BEQ IDLE significa que se ramificará (a la línea marcada con IDLE) solamente si el resultado es igual (EQual) a cero (Zero), es decir, sólo si el

indicador Z está puesto (= 1). Aquí, el mnemónico cc = EQ no es tan evidente (¿por qué no poner BZS, que significaría "saltar si está puesto el indicador de cero [*Branch if Zero Set*]?) [3], pero de todas formas lo emplearemos así. Este programa comprueba si D3, que contiene las horas YTD es cero o no. Si D3 no es cero, se sigue normalmente con el programa. Si D3 es cero, es que queremos hacer algo al respecto, por lo que saltaremos a la línea marcada con IDLE. Hemos escogido un mnemónico sugestivo, como siempre, para aumentar la legibilidad de nuestro programa. BEQ se emplea, a menudo, para saltarnos las partes innecesarias de un programa. Como un ejemplo frecuente valga éste: ¡No enviar nunca una factura de 0'0 pesetas! En nuestro caso, el salto a IDLE, si el número de horas YTD es nulo, no indica, necesariamente, un error (aunque pueda exigir un cuidado especial), sino que en un buen programas de nóminas se puede obviar con toda tranquilidad el coste de impuestos sobre el salario si el empleado no ha estado trabajando.

Bcc y el CCR

Bcc sirve para comprobar el CCR, pero no para alterar los indicadores del mismo: así, se pueden realizar varias pruebas sucesivas de Bcc, tales como BVS seguidas de BEQ, tal como en el programa 4.3. Si BVS no produce el salto, se comprueba inmediatamente el cero con BEQ.

Este punto pone de manifiesto la importancia del conocimiento de cómo cada instrucción afecta al CCR. Veamos un pequeño ejemplo de codificación cuidadosa que puede evitar muchas horas de frustración. Supongamos que después de estar funcionando con toda normalidad durante algunos años, decidimos añadir una inocente línea al programa 4.2 como la mostrada en el programa 4.2A (tales añadidos se conocen como **parches de una línea**):

```
* Programa 4.2A: Cambio incorrecto del programa 4.2
* Actualización YTD —4 rev. 3— Salvar D3 en D4 para uso posterior

        MOVE.L  D1,D3       El anterior YTD en D3 (32 bits)
        ADD.L   D2,D3       Añade las horas de mayo al anterior YTD
        MOVE.L  D3,D4       + + parche de línea

* D3 y D4 contienen, ambos, las horas YTD actualizadas.
```

[3] La razón de este mnemónico, como de otros muchos, está en su origen. En efecto, este tipo de denominaciones apareció ya en el pionero 6800, con el que Motorola entraba en el "ruedo" de los micros con una preciosa máquina muy apta para efectuar cálculos matemáticos no demasiado complejos para nuestros días y, sobre todo, controles de datos en entrada y/o salida de forma sencilla y eficaz. Así, la razón de BEQ estaba en la comprobación de la igualdad entre dos magnitudes: *Branch if they are EQual* (salta si son iguales).

```
        BVS     ERROR.1   Salta a ERROR.1 si hay rebose???
        <resto del programa como en 4.2>
        *       *       *
ERROR.1 <tomar acción de rebose de D3>   Se ha detectado rebose???
        *       *       *
```

Hemos introducido un MOVE.L entre el ADD y el BVS, tal como indica el comentario + + . ¿Dónde está el error? En el hecho de que el indicador V se borra con la instrucción MOVE, por lo que ahora el BVS carece de sentido. Nunca podremos saltar a ERROR.1 aunque ADD ponga el V a 1.

Bcc: Todas las posibilidades

Hasta ahora hemos visto dos de los 14 códigos Bcc. Las demás 12 Bcc comprueban otras condiciones del CCR, tanto de indicadores únicos como de combinaciones de ellos. Posponemos la discusión sobre las distintas comprobaciones de los indicadores hasta que no veamos el CMP (CoMPare) en el capitulo 6. He aqui los Bcc de comprobación de un único indicador de la forma BVS, o BEQ.

BVC: Salta si el indicador de rebose está a cero (V = 0)

Como se ve, BVS y BVC son pruebas **complementarias;** si una se cumple, la otra no.

BNE: Salta si no es cero, es decir, si Z = 0

También BEQ y BNE son complementarios —nuevamente, si se cumple uno, no se cumple el otro.

BCC: Salta si el indicador de acarreo está a cero (C = 0)
BCS: Salta si el indicador de acarreo está a uno (C = 1)

Una vez más, BCS y BCC son dos pruebas complementarias. Hay que tener cuidado y no confundir CC (Carry Clear) con cc (cualquier condición).

BPL: Salta si es positivo (Plus), esto es, si N = 0
BMI: Salta si es negativo (Menos), esto es, si N = 1

Si, ya lo ha adivinado usted, BPL y BMI también son complementarios.

Esta variedad de elección (recuérdese que aún quedan varios más complejos) nos permitirá una considerable flexibilidad en la comprobación de los resultados y en la toma de decisiones apropiadas mediante el salto (ramificación) a la linea de programa convenientemente marcada. Como he

mos visto, aparecen por parejas opuestas que pueden parecer, a primera vista, un derroche. Por ejemplo:

```
 BPL ANS__PLUS    Ejecutar el programa P si la respuesta es + ve
<programa M>      Ejecutarlo si la respuesta es — ve
*     *     *
      ANS__PLUS   <Programa P>
*     *     *
```

También puede escribirse de la forma:

```
BMI  ANS__MIN     Ejecutar programa M si el resultado es — ve
<programa P>      Ejecutar programa P si el resultado es + ve
*     *     *

      ANS__MIN    <Programa M>
*     *     *
```

Ambos programas llevan al mismo resultado, por lo que usted sobrevivirá igualmente con BPL que con BMI. En la práctica, sin embargo, resulta útil hacer una elección previa: salen programas más legibles empleando el Bcc más natural y apropiado para cada situación, provocando la ramificación, por ejemplo, cuando se produzca el evento menos probable.

Resumen de las ramificaciones

- BRA ETIQUETA provoca un salto incondicional a la etiqueta.
- Bcc ETIQUETA provoca el salto a la etiqueta sólo cuando se cumple la condición cc. Si no, se sigue con la siguiente instrucción.

Ahora que hemos visto ya algunas instrucciones sencillas que emplean los registros de datos como operandos, podemos pasar al estudio de otros casos en que los operandos empleados son combinaciones de registros y/o direcciones de memoria, conocidos como modos de direccionamiento.

Modos de direccionamiento

Imaginemos una instrucción que nos diga: "De acuerdo, ya sé que se supone que debo llevar (MOVE) o sumar (ADD) o lo que sea, pero dígame ahora dónde debo buscar el operador de origen y el de destino". Para ello, cada operando se expresa por medio de un determinado formato, uniéndolo, si es necesario, con la suficiente información para encaminar la instrucción a los datos correctos. Estos formatos de operandos se denominan **modos de direccionamiento,** de los que hay dos grupos principales: **modos directos a registros** y **modos de memoria.** Como veremos, cada uno de estos

grupos se subdivide, a su vez, en subgrupos menores de modos de direccionamiento. Asimismo veremos que la mayor parte de las instrucciones permiten elegir entre muchas combinaciones para el direccionamiento tanto del operador origen como del de destino.

Habiendo dejado claro que el camino para la comprensión del funcionamiento del 68000 depende en un 150 por 100 del dominio de la instrucción Bcc, podemos añadir ahora que el restante 150 por 100 es el relativo a los modos de direccionamiento.

Modo directo de registros

Hasta ahora, en nuestros programas, el dato inicial estaba disponible en los registros de datos D1 y D2, y se almacenaba el resultado de la suma D2 + D3 en un tercer registro de datos D3. Así, todos nuestros operandos estaban expresados de forma directa mediante un sencillo formato denominado modo directo de registros. Para este modo, hay dos variantes:

Modo de direccionamiento	*Símbolo*	*¿Qué operando usa?*
Directo de registro de datos	Dn	El valor de Dn
Directo de registro de direcciones	An	El valor de An

donde Dn puede ser cualquier registro de datos (del D0 al D7) y An cualquier registro de direcciones (del A0 al A7). Para simplificar, nos referiremos a estos dos modos por "modo Dn" y por "modo An".

En el capítulo 3 vimos la forma en que los registros de direcciones pueden almacenar direcciones de 16 ó 32 bits, de forma que, debido a esto, el modo An tiene ciertas restricciones, en el sentido de que está prohibido el manejo de tamaños de bytes con ellos. Aparte de estas diferencias, que más adelante detallaremos, los modos Dn y An son muy similares.

Estos dos modos directos se emplean cuando los operandos origen sean valores (datos o direcciones) que estén disponibles en los registros. Cuando se emplea el modo directo para el destino, hay que informar a la CPU del registro de que se trate para recibir el resultado de las operaciones. El próximo apartado trata de los operandos de memoria.

Modos de direccionamiento de memoria

En la mayor parte de los casos prácticos, los datos iniciales de un programa se han cargado previamente en la memoria de usuario (RAM) a través de algún dispositivo externo, tal como un disco o un teclado. El resultado de las distintas operaciones del programa se almacenará también en la RAM, para desde ahí ir a los dispositivos externos de salida, tales como los discos, impresoras o pantallas. Para conseguirlo se necesitan los modos de

direccionamiento de memoria, que le "dicen" al código de operación dónde localizar en la memoria los operandos de origen y destino. El 68000 tiene 10 modos distintos de direccionamiento de memoria, mientras el 68020 tiene 16, tal como se indica en el apéndice B.

El direccionamiento de memoria y la dirección efectiva

Los formatos de direccionamiento de memoria van desde el caso extremamente sencillo ("Aquí está la dirección de la memoria donde está el operando") hasta el más complicado ("¡Para conseguir saber la dirección de la memoria donde se halla el operando es preciso sumar dos números: para formarla, vaya a tal dirección de la memoria, donde encontrará otra dirección; súmele tal número y ésa es la dirección de su operando!"). Afortunadamente, el principiante puede ponerse a trabajar ya con los modos más sencillos. El caso complicado ha sido seleccionado para poner de manifiesto un hecho fundamental respecto a los modos de direccionamiento de memoria:

El modo de direccionamiento permite que la instrucción calcule la dirección efectiva (real) < ea > del operando

Cualquier operando situado en la memoria está absolutamente definido y localizado mediante su #<ea>, su dirección efectiva. En todo modo de direccionamiento, la CPU realiza el cálculo de la < ea > previamente a acceder al operando origen situado en dicha < ea >. Similarmente, antes de que se pueda escribir un operando resultado en la memoria, debe calcularse su < ea >. Estos cálculos (y sus accesos a memoria de programa necesarios) pueden ocupar entre 0 y 24 ciclos de reloj. La importancia de tales cálculos queda puesta de manifiesto por el hecho de que Motorola ha dispuesto de dos ALU adicionales para realizar el cálculo de la < ea > aritméticamente, mientras que la ALU principal se dedica al trabajo de aritmética con los datos normales.

¿De qué forma puede el 68000 saber cómo calcular el valor de la <ea>? Sin entrar en detalles, podemos decir que cada palabra de instrucción, a nivel de código máquina, tiene ciertos bits encargados de la codificación sin indeterminaciones de cada posible modo de direccionamiento para los operandos origen y destino. Al decodificar la palabra de la instrucción, el 68000 sabe tanto qué operación debe ejecutar como la forma de calcular las direcciones de los operandos.

En el capítulo 2 vimos que una de las ventajas de pasar de 8 a 16 bits es que las palabras de las instrucciones de 16 bits permiten mayor riqueza en el set de instrucciones y que eso conlleva los modos más potentes de direccionamiento.

¿Por qué tantos modos de direccionamiento?

Precisamente, esta abundancia en modos de direccionamiento, que antes sólo podía encontrarse en macro o miniordenadores, es la clave del éxito del 68000. Para alguien no avezado en el tema, puede parecer paradójico, pero esta aparente complejidad en los modos de direccionamiento es lo que realmente simplifica la programación del 68000, con esa enorme capacidad de direccionamiento tan útil para los cada vez más numerosos y sofisticados sistemas operativos multiusuario, compiladores, bases de datos relacionales y aplicaciones gráficas. Todos ellos requieren una rápida manipulación de las complejas estructuras de los datos almacenados en memoria, tales como listados anillados, "árboles" que crecen en todas las direcciones, disposiciones y tablas multidimensionales, pilas y colas. Buena parte del manejo de estos datos, especialmente en aplicaciones comerciales, como contrarios a los científicos, son más generación de direcciones que cálculo de datos.

Para localizar los operandos complejos con los anteriores micros de 8 bits de pocos modos de direccionamiento, el programador se veía obligado a escribir códigos muy específicos, a menudo tediosos, para el cálculo de la <ea>. Los avanzados modos del 68000 han reducido drásticamente este esfuerzo. En efecto, el programador ha cedido la responsabilidad del cálculo a las rápidas ALU interiores.

En esencia, los modos de direccionamiento del 68000 (y los modernos ensambladores que hacen uso de ello) ofrecen una especie de lenguaje de alto nivel para el acceso a las complejas estructuras de datos para grandes volúmenes de RAM.

Como veremos, cada modo de direccionamiento ha sido pensado para resolver un tipo particular de acceso a memoria y de manipulación de datos. Nuestro primer ejemplo fue, probablemente, el del modo más sencillo de direccionamiento, llamado **inmediato,** que sólo se utiliza para los operandos origen.

Modo de direccionamiento inmediato

En muchas ocasiones deseamos emplear constantes numéricas —es decir, números predeterminados que no son el resultado de ninguna operación—. Supongamos, por ejemplo, que en el programa 4.2 queremos obtener el número de empleados parados: todos aquellos de número de horas YTD cero. Asignemos el registro de datos D4 a contener dicho número. Cada vez que saltamos a la etiqueta IDLE, incrementaremos (le sumaremos 1) D4. Al terminar las nóminas, podemos sacar el valor de D4. El programa 4.3 muestra la nueva versión (con las adiciones en negrita).

```
* Programa 4.3: Cuenta de empleados parados
* Actualización de YTD 4 rev. 3 SKB
```

```
                MOVE.L  D1,D3       Anterior YTD, ahora en D3 (32 bits)
                ADD.L   D2,D3       Añadir las horas de mayo al anterior YTD
                CLR.W   D4          Poner los 16 bits inferiores de D4 a cero

* Ahora D3 contiene las horas actualizadas de YTD.

                BVS     ERROR.1     Saltar si hay rebose
                BEQ     IDLE        Saltar si es cero
                <resto del programa>
                *       *       *
ERROR.1         <tomar la acción de rebose en D3>   Rebose detectado
                *       *       *
                BRA     OVER        Saltar siempre a OVER
IDLE            ADD.Q   #1,D4       Incrementar D4 = cuenta de parados
                *       *       *

OVER            <programa de finalización>
                <escribir el valor de D4>
```

Notas al programa 4.3

En el programa revisado hay dos códigos nuevos, CLR (Borrar = CLeaR) y ADDQ (Suma rápida = ADD Quick). CLR es una instrucción sencilla pero muy útil, con una sola palabra y la sintaxis siguiente:

```
CLR.z   <operando>
```

que borra parte o todo el operando, según la letra .z, identificadora del tamaño del dato. CLR.L borrará los 32 bits. CLR.W borrará sólo los 16 bits menos significativos, dejando los más altos sin tocar, y CLR.B borrará tan sólo los 8 bits menos significativos, dejando los 24 más altos sin tocar.

En nuestro ejemplo hemos decidido emplear justamente los 16 bits más bajos de D4 como "contador de parados", dejándole a usted, si quiere, usar los 16 altos para cualquier otra misión. Para nóminas pequeñas (de menos de 255 empleados) se puede considerar en utilizar solamente el byte bajo de D4 (sin signo): el código de tamaño de datos .z permite una gran flexibilidad a la hora de utilizar los registros. Se hace CLR.W D4 para asegurarnos de que el contador de 16 bits de D4 está a cero antes de empezar a contar. Recuérdese que CLR.W no afecta a los 16 bits más altos de D4. Es un error sorprendentemente muy común el olvidar "CLRear" —borrar— los contadores [4].

CLR provoca los cambios esperables en el CCR:

[4] A veces no podemos evitar caer en la tentación de imitar la útil costumbre americana de formar verbos a partir de sustantivos, sobre todo cuando esto ayuda al lector español a familiarizarse con el texto.

Indicador del CCR	X	N	Z	V	C
CLR	–	0	1	0	0

puesto que el operando ahora es 0 (Z = 1), no negativo (N = 0), y no hay ni rebose ni acarreo (V = 0, C = 0). El indicador X permanece inalterado, tal como dijimos en la instrucción MOVE anterior.

Como ADD, ADDQ es una instrucción con dos operandos. Su formato general es:

```
ADDQ.z    #<dato>,<operando destino>
```

donde <dato> es un número comprendido entre 1 y 8. Esto significa que es un ADD (suma) del número de <dato> al destino (en L, W o B, según sea el código de tamaño) y colocar la suma en el destino. La "Q" de ADDQ indica que se trata de la forma rápida (*Quick*, en inglés) del código de operación de ADD. El dato inmediato de origen, escrito siempre con el símbolo delante, se suma, sencillamente, al destino. En nuestro caso,

```
ADDQ.W    #1,D4
```

incrementa en 1 la palabra baja de D4.

La razón de que sea rápido estriba en que el procesador no tiene que ir a ninguna parte a buscar el dato de origen. De hecho, el dato origen está grabado en 3 bits sin signo en la propia palabra del código de ADDQ, lo que explica también el porqué del pequeño rango de estas constantes, que sólo va desde 1 hasta 8.

Ahora he aquí una pequeña y rápida comprobación de lo que hemos aprendido. ¿Cómo pueden almacenarse los números del 1 al 8 en 3 bits? Pues, ¿no es cierto que tres bits sirven para codificar los números del 0 al 7? Pues, porque el 68000 ¡hace trampas! Es muy instructivo observar el truco, porque podremos penetrar "dentro" de una simple palabra de instrucción y analizar su configuración. La palabra ADDQ es de la forma

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	1	d	d	d	0	z	z	m	m	m	r	r	r

Donde:

ddd	—	bits del 9 al 11 especifican el #<dato>
zz	—	bits del 6 y 7 especifican el tamaño del dato
mmm	—	bits del 3 al 5 especifican el modo de destino
rr	—	bits del 0 al 2 especifican el registro de destino

Los bits 8 y del 12 al 15 identifican la instrucción como ADDQ.

Los códigos de modo y registro (mmm y rrr) son los bits que, como mencionábamos antes, indican al 68000 los cálculos necesarios para cono

cer la <ea>; en este caso, mmm y rrr están codificados para determinar el registro de destino correspondiente a esta <ea>. En este momento no nos interesa su formato, salvo poner de manifiesto el hecho de que, por regla general, se dedican a este propósito 6 de los 16 bits de la palabra de la instrucción. En el apéndice B se listan todos los códigos de las instrucciones.

El código del tamaño de los datos es simple: 00 = byte, 01 = palabra, 10 = doble palabra (dejando el código 11 sin usar, como una útil posibilidad para el futuro). En cuanto al "truco", la forma de codificar los números del 1 al 8 en tres bits, tenemos que es:

#1	Para ddd = 001
#2	= 010
#3	= 011
#4	= 100
#5	= 101
#6	= 110
#7	= 111
#8	= 000

Así, el decodificador de la instrucción se convierte ddd = 000 no a #0 (lo que sería una pérdida de tiempo y esfuerzo) sino a #8. Esta pequeña digresión nos ha servido para tomar conciencia de lo que significa la decodificación de las instrucciones. En las instrucciones más complicadas multipalabras que más adelante encontraremos será preciso saber en qué forma están dispuestas en las palabras de los códigos de los datos (incluyendo las direcciones) y en sus extensiones.

Esperamos que con esto hayamos conseguido ganarnos su respeto hacia la gente que ha escrito los ensambladores. Recuérdese que el ensamblador ha de convertir

```
ADDQ    #<dato>,<destino>
```

del programa fuente en la configuración de bits que hemos visto (entre otras cosas).

ADDQ y el CCR

ADDQ actúa sobre el CCR como un ADD normal:

Indicador del CCR	X	N	Z	V	C
ADDQ	= C	*	*	*	*

con lo que se obtienen los mismos indicadores que con la aritmética normal. Pero, ¿qué ocurre si se necesita sumar una constante mayor que 8? Sigamos leyendo lo que viene a continuación.

ADDI: Suma inmediata

Si se necesita sumar (ADD) constantes mayores que #8, debe emplearse la instrucción ADDI, que veremos ahora por su similitud con ADDQ. Su estructura es de la forma:

```
ADDI.z    #<dato>,<operando destino>
```

ADDI nos permite sumar un dato inmediato de hasta 32 bits a una doble palabra, de 16 bits a una palabra o de 8 bits a un byte. Por ejemplo,

```
ADDI.L    #$FFFFF,D0
```

incrementará D0 en $FFFFF = 11111111111111111111, pero

```
ADDI.W    #$FFFFF,D0
```

incrementará solamente la palabra inferior de D0, y lo hará en la cantidad de $FFFF = 1111111111111111, y, de la misma forma,

```
ADDI.B    #$FFFFF,D0
```

solamente incrementará el byte inferior de D0, haciéndolo en $FF = 11111111.

ADDQ es casi el doble de rápida que ADDI, por la sencilla razón de que ADDI necesita una o dos palabras de extensión para contener el <dato> inmediato de gran tamaño que maneja. Los tres bits del ADDQ#<dato> caben en la propia palabra de código, mientras que ADD.W y ADD.B precisan de una palabra de extensión, y ADD.L de dos. Veamos a continuación en qué forma.

Codificación de la instrucción ADDI

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1.ª Palabra	0	0	0	0	0	1	1	0	z	z	m	m	m	r	r	r

2.ª Palabra W #<dato>: 16 bits, o B #<dato>:8 bits

3.ª Palabra L #<dato>: aqui los otros 16 bits hasta el total de 32

Recuérdese que estas dos palabras de extensión están almacenadas en la memoria y deberán ser leídas como cualquier otro operando. Hay, sin embargo, una ventaja al respecto: sus direcciones de memoria están a continuación inmediatamente detrás de la palabra de la instrucción. Ahora ya sabemos que el PC (contador de programa) contiene la dirección de la instrucción y que después de la decodificación se habrá incrementado en 2. Así, en cuanto se decodifica el ADDI, el PC contiene ya la dirección donde se halla el dato inmediato en byte, palabra o doble palabra. Con el modo

de direccionamiento inmediato, no hay necesidad de ciclos extra de procesador o de cálculos para conocer la dirección efectiva de la memoria donde está el operando. El dato origen forma parte de la propia instrucción y está disponible en la misma secuencia de lectura y decodificación.

Por ahora hemos usado el modo inmediato solamente para las instrucciones ADDQ y ADDI, pero se puede encontrar en muchas otras más.

Aplicaciones generales del modo inmediato

He aquí un sencillo ejemplo:

```
MOVE.z      #<dato>,<operando de destino>
```

En este caso, sencillamente se lleva el dato inmediato (L, W, o B) al destino. La extensión correspondiente al #<dato> se almacena en memoria de la misma manera que con las extensiones del ADDI. No existe MOVEI como tal, sino que basta con un MOVE con formato de inmediato.

Obsérvese que las dos instrucciones siguientes son funcionalmente equivalentes:

```
CLR.z       D1
MOVE.z      #0,D1
```

Puesto que ambas cargan la parte L, W, o B del registro D1 con 0. ¿Podría usted adivinar cuál de las dos es más rápida? Una pista: CLR.z no precisa de palabras de extensión.

Otra versión muy popular del modo inmediato es:

```
MOVEQ.L     #<dato>,Dn
```

que es un transporte rápido de datos reservado para dobles palabras y usado sólo con registros de datos. MOVEQ permite llevar un dato de 8 bits con signo a un registro de datos, por lo que suele escribirse

```
MOVEQ.L     #<d8>,Dn
```

como regla mnemotécnica. Hay que tener en cuenta que para llevar un dato de 8 bits como si fuera una doble palabra, el 68000 realiza primero la **extensión de signo** del #<d8> a 32 bits.

Los bits de #<d8> forman parte del código de la propia instrucción, por lo que ésta representa una superrápida forma de llevar números del rango comprendido entre −128 y +127 con signo al total de los 32 bits de Dn.

He aquí una aplicación típica:

* Programa 4.4: Cuenta de ciclos

```
        MOVEQ.L  #52,D0    Se pone el contador D0 a + 52
LAZO    <programa>
        *    *    *
        SUBQ.L   #1,D0     Se decrementa el contador en 1
        BNE      LAZO      ¿Está el contador a 0? Si no, repítase el LAZO
                           Si es que sí, abandónese el LAZO
        <resto del programa>
```

En este caso hemos supuesto que queríamos ejecutar el programa 52 veces, como 52 es menor que 127, hemos empleado MOVEQ para preparar nuestro contador en D0. Cada vez que se ejecuta el <programa> en el lazo, se SUBstrae 1 del registro D0 y se comprueba el estado del CCR con la instrucción BNE (Saltar —*Branch*— si No Es cero). Mientras D0 no sea 0, se volverá a la etiqueta LAZO y se repetirá el <programa> una vez más. Después de exactamente 52 saltos, D0 alcanzará el valor 0, y no se saltará más como consecuencia del BNE. Por tanto, se abandonará el <programa> y se seguirá con el <resto del programa>. La instrucción SUBQ.L es nueva, pero resulta de sentido obvio:

```
SUBQ.z    #<dato>,<operando destino>
```

funciona exactamente como ADDQ, excepto en que se substrae el dato inmediato origen (valores del 1 al 8 sin signo) del destino, colocando el resultado en el destino. El código z del tamaño tiene el significado que ya nos es habitual: SUBQ.L lo restará del total de los 32 bits, SUBQ.W lo hará sólo de la palabra inferior y SUBQ.B solamente del byte menos significativo del operando de destino. Corresponde a las variedades de ADD y ADDI, también hay versiones del SUB:

```
SUB.z     <operando origen>,<operando destino>
SUBI.z    #<dato>,<operando destino>
```

Todas las variantes del SUB alteran el CCR en la misma forma que las variantes del ADD:

Indicador del CCR	X	N	Z	V	C
SUB	= C	*	*	*	*

de forma que se pueden comprobar las situaciones de cero, negativo y rebose de la forma usual. El indicador de C (acarreo) en realidad indica acarreo negativo, pero su funcionamiento es el mismo. ¿Acaso se equivoca la aritmética sin signo? Por ejemplo:

```
MOVEQ.L   #0,D6
SUBQ.L    #1,D6
```

dejará el CCR de la forma

Indicador del CCR	X	N	Z	V	C
	1	1	0	0	1

puesto que ahora D6 contiene el valor $FFFFFFFF = −1 (con signo), que es correcto (no hay rebose), pero incorrecto desde el punto de vista de la aritmética sin signo.

Resumen del modo de direccionamiento inmediato

El modo inmediato es limpio y rápido para manejar valores origen fijos. Existen ciertas variantes Quick #<dato>, del código de operación para datos suficientemente pequeños como para caber en el código de la propia instrucción, y hay versiones del inmediato para grandes valores del #<dato> que precisan de una o dos palabras de extensión situadas a continuación de la instrucción (en las direcciones de memoria PC + 2 y PC + 4).

A continuación veremos modos de direccionamiento que nos permitirán acceder a datos situados en cualquier parte de la memoria, no solamente los de palabra del código de instrucción o sus extensiones.

Direccionamiento absoluto

Desde luego, el direccionamiento inmediato está limitado al caso en que los valores del operando origen sean fijos y conocidos de antemano, antes de escribir la instrucción.

Sin embargo, en la mayor parte de los casos, los operandos son variables situadas en la memoria (desde un teclado o a través de un acceso a los ficheros de un disco). La situación más simple se presenta cuando se intenta conocer la dirección de memoria real, conocida también como **dirección absoluta,** de los datos que se desean manejar. Escribamos de nuevo el programa 4.1, suponiendo que los valores iniciales están en la memoria, en lugar de los registros de datos. Como hicimos en el capítulo 3, emplearemos el símbolo $ para denotar los números hexadecimales.

APLICACION PRACTICA

Problema: Calcular el total actualizado de horas YTD (desde enero hasta marzo), empleando operandos absolutos de memoria. Colocar el resultado en la doble palabra de dirección $6008.

Datos:

1. El total de horas YTD desde enero y febrero está en la dirección $6000.
2. Las horas trabajadas en marzo están en la dirección $6004.
3. Los valores están en tamaño de dobles palabras.

Ejemplos:

Horas YTD = 320 en la dirección $6000.
Horas marzo = 138 en la dirección $6004.
Horas YTD actualizadas = 458 en la dirección $6008.

Solución: Programa 4.5

```
MOVE.L  $6000,D3     D3 = horas YTD
ADD.L   $6004,D3     D3 = horas YTD + horas marzo
MOVE.L  D3,$6008     Se salva D3 en la memoria $6008
```

* La dirección $6008 ahora contiene la suma ($6000) + ($6004) = 458

Nótese el uso de los paréntesis para indicar que se está trabajando con los datos contenidos en esta dirección. Así, en la dirección ($6000) tenemos el valor ($6000) = 320). En el próximo apartado hablaremos de esta notación y veremos con detalle cómo las direcciones pueden acceder a dobles palabras, palabras y bytes de memoria.

El modo de direccionamiento absoluto indica simplemente la dirección real del origen, o del destino: no debe confundirse con el modo inmediato. Comparemos los siguientes casos:

```
MOVE.L    #$6000,D3
```

y

```
MOVE.L    $6000,D3
```

El pequeño símbolo "#" del dato inmediato marca drásticamente la diferencia. La primera línea significa: Reemplazar el contenido de D3 por el número $6000. La segunda línea significa: Reemplazar el contenido de D3 por el número que está en la dirección de memoria $6000.

Direccionamiento absoluto: Versiones larga y corta

La dirección absoluta que damos al especificar el operando origen o el de destino puede ocupar una palabra o dos, de acuerdo con el formato de la instrucción, dando lugar a las variedades **corta** y **larga** del modo de direccionamiento absoluto [5]. En la versión larga, la dirección absoluta es un número de 32 bits almacenado como dos palabras de 16 bits de extensión, lo que permite acceder a todo el espacio direccionable por el 68000.

El precio que debemos pagar por ello es que el procesador debe efectuar

[5] Obsérvese que, en este caso, dar la dirección absoluta de la memoria donde está el dato coincide con dar su <ea>, de la que ya se ha hablado anteriormente.

una lectura de las dos palabras de extensión en la memoria y combinarlas para formar la dirección de 32 bits antes de poder acceder al operando. Cuando no se desea poder acceder al espacio total de direccionamiento, se puede ahorrar tiempo empleando la versión corta de 16 bits. Con esta versión, el procesador extiende el bit de signo hasta el total de los 32 bits de la dirección final, lo que es mucho más rápido que tener que tomar una segunda palabra de la memoria. La variedad de modo corta permite acceder a direcciones comprendidas en el rango desde $000000 hasta $007FFF (los 32 Kbytes más bajos) y desde $FF8000 hasta $FFFFFF (los 32 Kbytes más altos), según que el bit de signo sea "0" o "1". (Cuando se hable del bit de signo de una dirección, desde luego no estamos sugiriendo que haya direcciones negativas: todas las direcciones absolutas son números positivos. Se entiende por bit de signo al colocado en la posición 15 de la palabra, que resulta ser "1" para direcciones superiores a $7FFF.)

Hay distintos modos de tratar las direcciones absolutas cortas y largas según los distintos ensambladores. Algunos generan automáticamente el mejor modo para nosotros (por ejemplo, nuestro $6000 sería tomado como corto), mientras otros requieren la letra del código L (largo) o W (corto, es decir, una sola palabra de extensión) tras la dirección.

Etiquetas como direcciones absolutas

Ya hemos empleado las etiquetas para las instrucciones de ramificación, y hemos explicado brevemente que, cuando el programa esté ensamblado y cargado en memoria, cada símbolo de etiqueta se convierte en la dirección de la instrucción a la que se desea saltar. Desde el punto de vista de un programador poco avezado, será suficiente considerar las etiquetas como direcciones y, como tales, emplear los mnemónicos no sólo para saltar, sino también como direcciones absolutas de operandos. La ventaja obvia es su legibilidad y facilidad de programación. Para verlo, reescribamos el programa 4.5 de la forma siguiente:

```
* Programa 4.5A: Uso de etiquetas como Direcciones Absolutas

* Datos como en el programa 4.5, con las etiquetas siguientes:
* HRSYTD  = dirección $6000, conteniendo 320 horas
* HRSMAR  = dirección $6004, conteniendo 138 horas
* NEWHRS  = dirección $6008 = destino de la suma

        MOVE.L  HRSYTD,D3       D3 = (HRSYTD)
        ADD.L   HRSMAR,D3       D3 = (HRSYTD) + (HRSMAR)
        MOVE.L  D3,NEWHRS       Salvar D3 en la dirección NEWHRS

* Ahora la dirección NEWHRS = $6008 contiene la suma ($6000) + ($6004) = 458
* Nótese, otra vez, el uso de paréntesis para indicar que se trata del contenido
* de una dirección. Así, tendremos que HRSYTD = $6000; pero (HRSYTD) = 320
```

Sin lugar a dudas, la versión 4.5A es mucho más comprensible inmediatamente que la 4.5 (quizá, algo más parecida al BASIC), y esto ayuda a escribir, modificar y comprobar los códigos. No es ninguna exageración decir que difícilmente veremos líneas como ésta:

```
MOVE.L    $6000,D3
```

salvo en ejemplos didácticos. Desde luego, hay que informar al sistema de lo que son las etiquetas HRSYTD, HRSMAR y NEWHRS, y para ello se precisa de cierta ayuda por parte del ensamblador. Una vez "asignadas" las direcciones a las etiquetas, se puede programar en términos de HRSYTD, etcétera, en lugar de ocupar nuestra memoria humana con números hexadecimales carentes de significado. La colocación de las direcciones en los campos de datos es casi tan simple como la etiquetación de los números de las instrucciones para la ramificación.

Directivas del ensamblador

Para indicar al ensamblador lo que deseamos, necesitamos hacer uso de unas pocas **directivas del ensamblador**, también conocidas a veces por **pseudocódigos**, porque, a primera vista, parecen instrucciones del tipo de las del 68000. Sin embargo, las directivas se limitan a dirigir y controlar el proceso de ensamblado y, a diferencia de las instrucciones "de verdad", no generan lenguaje en código máquina. Los ensambladores modernos disponen de centenares de diferentes directivas con muchas variantes no estandarizadas, la mayor parte de las cuales se sale del objetivo de esta instrucción.

Afortunadamente, solamente se necesitan tres o cuatro de tales directivas para dar sentido a los modos de direccionamiento, por lo que las presentaremos empleando la sintaxis estándar "vainilla" de Motorola. Una vez descritas, podremos completar el programa 4.5A, de forma que HRSYTD represente realmente la dirección que tenemos en la cabeza.

Directiva de origen absoluto: ORG

La siguiente línea

```
ORG       <dirección>       Cargar el programa en <dirección>
```

indica simplemente al cargador/ensamblador que se desea que las líneas de programa que vienen a continuación sean ensambladas y, eventualmente, cargadas a partir de una dirección específica dada de forma absoluta. Para nuestros actuales propósitos, solamente necesitaremos una directiva de ORG al principio del programa:

```
ORG       $6000             El programa empieza en la dirección $6000
```

A partir de aquí, cada línea del programa fuente será traducida al lenguaje máquina de instrucciones, algunas con una sola palabra, otras hasta cinco, siendo colocadas todas en la dirección apropiada para cada palabra a medida que el ensamblador vaya incrementando su **contador de posición:** un simple contador que empieza en el $6000 y se incrementa en uno o en (hasta) cinco, según las imposiciones de cada instrucción. Cuando se encuentra una etiqueta, inmediatamente ya sabemos (mejor dicho, ya sabe el ensamblador) su dirección. Antes vimos cómo funcionan las etiquetas de salto, ahora podemos entender cómo lo hacen las de datos y las directivas empleadas para definirlas.

Areas de datos de memoria con DS y DC

Hay dos directivas básicas, DS y DC, que nos permiten colocar etiquetas identificativas de zonas de datos en memoria:

```
ETIQUETA    DS.z        <número>    Define Almacenamiento
ETIQUETA    DC.z        <dato>      Define Constante
```

DS sirve para reservar un determinado<número>de posiciones de memoria (z = L, W, o B) en la dirección = Etiqueta, mientras que DC colocará, en la memoria que sea precisa, los<datos>que se escriban en la columna de la derecha a partir de la dirección Etiqueta.

Ejemplos de almacenamiento de datos

```
NEWHRS    DS.L        1    define almac. = 1 doble palabra en NEWHRS
```

sirve para reservar una doble palabra "vacía" para almacenamiento de datos en la dirección NEWHRS, que es exactamente lo que se necesita para el programa 4.5A. Las líneas siguientes realizan la misma función:

```
NEWHRS    DS.W        2    define almac. = 2 palabras en NEWHRS
NEWHRS    DS.B        4    define almac. = 4 bytes en NEWHRS
```

Las palabras ***buffer*** o ***buffer de datos*** (directamente empleadas desde el original inglés por su aceptación corriente en nuestra lengua) se usan para denotar áreas de memoria asignadas, por medio de DS, a contener información futura. A menudo nos encontraremos, por ejemplo, con líneas como ésta:

```
DSKBUF    DS.B        512    Disponer 512 bytes para buffer de disco
```

o similar, para definir un área general en la que cargar un bloque de información desde un disco.

Se puede, desde luego, emplear DS para reservar sitio para cualquier número de dobles palabras, palabras, o bytes, con tal de que se tenga cuidado de evitar las direcciones impares para las palabras, o las dobles palabras. Por ejemplo:

```
ETIQUETA1  DS.B  5   Situar 5 bytes desde "Etiqueta1".
ETIQUETA2  DS.L  2   ¡Ojo! Si Etiqueta1 es par, entonces Etiqueta2 es impar y no pueden almacenarse dos dobles palabras a partir de dirección impar.
```

La conclusión es que el ensamblador reserva el espacio de memoria solicitado, incrementa su contador de posición para apuntar tras el área asignada, dando a la siguiente etiqueta la dirección convenientemente incrementada. Así, aquí tratará de dar a Etiqueta2 el valor correspondiente a [Etiqueta1 + 5 bytes], que puede ser, o no, legal. Podría ponerse un DS.B en Etiqueta2 (las direcciones de los bytes pueden ser pares o impares), pero un DS.L, o un DS.W, provocarán un error de dirección.

Suponiendo que no hayamos forzado las reglas de par/impar, la dirección absoluta asignada a un DS dependerá, naturalmente, de dos factores: ORG, la dirección de partida del programa, y la situación de la etiqueta dentro del mismo.

Antes de pasar a ilustrar esto con un programa, analicemos la otra directiva de etiquetado de datos, DC, con más detalle.

Datos constantes: Ejemplos

La sintaxis de datos constantes (DC) es sólo un poco distinta, pero esta pequeña diferencia es de importancia galáctica:

```
HRSYTD    DC.L      320   Define constante = 320 en HRSYTD
```

no posiciona 320 dobles palabras de memoria.

El 320 representa un único campo de<datos>y nos basta con una doble palabra (debido al DC.L) para contener el binario equivalente al decimal 320 en la dirección que el ensamblador asigne a HRSYTD. Se puede emplear DC para almacenar cualquier cantidad de datos. A menudo se llama **tabla** a una lista de datos relacionados y almacenados consecutivamente, tal como las tablas de logaritmos, o las tablas trigonométricas, al final de los antiguos libros de texto.

```
TABLA     DC.B  $10,$2A,$F4,$09   Definición de las cuatro constantes
                                  que forman TABLA
```

* El byte de dirección TABLA contiene ahora $10
* El byte de dirección TABLA + 1 contiene ahora $2A

```
* El byte de dirección TABLA + 2 contiene ahora $F4
* El byte de dirección TABLA + 3 contiene ahora $09
```

El formato general es:

```
ETIQUETA  DC.z  <dato>,<dato>,...
```

DC.z dispondrá del área suficiente para contener los <dato>,<dato>,... que se indican, escribiéndolos sobre cualesquiera que hubiera en las direcciones marcadas por la etiqueta. El <dato> puede venir expresado en cualquier forma habitual: binario, decimal, hexadecimal, o ASCII.

Merece la pena recordar, una vez más, aquí, que ORG, DS y DC son seudocódigos, no instrucciones verdaderas del 68000. No serán traducidas en palabras de instrucciones de código máquina como un MOVE, o un ADD. Sin embargo, afectarán la posición del programa, la forma en que lo escribiremos y los valores que encontraremos en las palabras de extensión de las instrucciones después del ensamblado y la carga.

Además, los seudocódigos y las directivas han sido un ingrediente esencial de los ensambladores durante muchos años, lo que ha ejercido una considerable influencia en los diseñadores de microprocesadores cuando deben decidir el tipo y formato del set de instrucciones.

Etiquetas de datos en acción

Vamos ahora a reescribir el programa 4.5 para poner de manifiesto el funcionamiento de las directivas:

```
* Programa 4.5B. Etiquetas de datos con DS y DC
* Revisión del programa 4.5A
*   HRSYTD = dirección $6000, conteniendo 320 horas
*   HRSMAR = dirección $6004, conteniendo 138 horas
*   NEWHRS = dirección $6008 = destino de la suma

          ORG     $6000     Inicio en $6000 absoluto

* Definición de las áreas de datos
* Primera etiqueta HRSYTD = dirección $6000
* Después de almacenar 32 bits ahí, la siguiente etiqueta, HRSMAR,
* será = $6004, y así sucesivamente

HRSYTD    DC.L    320       Almacenar 320 en HRSYTD
HRSMAR    DC.L    138       Almacenar 138 en HRSMAR
NEWHRS    DS.L    1         Reservar 1 doble palabra en NEWHRS

* Resto del programa como en 4.5A

          MOVE.L  HRSYTD,D3     D3 = (HRSYTD) = ($6000)
```

```
ADD.L   HRSMAR,D3     D3 = (HRSYTD) + (HRSMAR)
MOVE.L  D3,NEWHRS     Salvar D3 en la memoria en la dirección
                      NEWHRS = $6008

* Ahora la dirección NEWHRS = $6008 contiene la suma ($6000) + ($6004) = 458
```

En este ejemplo hemos empleado ORG para fijar un área de memoria, empezando en la dirección absoluta $6000, que contiene no solamente nuestros datos, sino también el propio programa. La primera instrucción, MOVE.L, será colocada en la dirección [$6008 + 4 bytes] = $600B, justo detrás de NEWHRS.

Es perfectamente posible, y a menudo preferible, separar los datos del programa en memoria. La forma más sencilla de hacerlo consiste en emplear una segunda directiva ORG<dirección> para definir el principio del programa. A continuación tenemos el programa 4.5C con una línea más, que hace precisamente esto:

```
* Programa 4.5C: separación de las áreas de datos y programa
* Revisión del programa 4.5B. Iguales datos que en 4.5B
*   HRSYTD  = dirección $6000, conteniendo 320 horas
*   HRSMAR  = dirección $6004, conteniendo 138 horas
*   NEWHRS  = dirección $6008 = destino de la suma

        ORG     $6000   Los datos empiezan en la $6000 absoluta

* Definición del área de datos
* La primera etiqueta HRSYTD será = dirección $6000
* Después de almacenar 32 bits ahí, la siguiente etiqueta, HRSMAR,
* será = $6004, y así sucesivamente

HRSYTD  DC.L    320     Guardar un 320 en HRSYTD
HRSMAR  DC.L    138     Guardar un 138 en HRSMAR
NEWHRS  DS.L    1       Reservar 1 doble palabra en NEWHRS

        ORG     $8000   El programa empieza en la dirección
                        absoluta $8000

* La primera instrucción estará en la $8000
* El programa y los resultados igual que en el 4.5B

        MOVE.L  HRSYTD,D3     D3 = (HRSYTD) = ($6000)
        ADD.L   HRSMAR,D3     D3 = (HRSYTD) + (HRSMAR)
        MOVE.L  D3,NEWHRS     Salvar D3 en la memoria en la dirección
                              NEWHRS = $6008

* La dirección NEWHRS = $6008 ahora contiene la suma de ($6000) + ($6004) = 458
```

En su nueva posición, el programa sigue trabajando como antes, puesto que, cuando hace referencia a HRSYTD y las demás etiquetas, todavía

toma las direcciones de memoria definidas por ORG $6000 y nuestras etiquetas de datos DC y DS. HRSYTD estará definido como $6000, independientemente de donde coloquemos nuestro programa.

La ventaja inmediata de separar datos de programa es la posibilidad de que varios usuarios con distintos programas compartan una misma tabla en una zona común de memoria mutuamente aceptada como tal. En las instalaciones reales se encuentran variantes sin fin de esa separación entre áreas de datos y de programas. El asunto ahora es que hemos ganado una considerable flexibilidad, independientemente de donde pongamos las cosas.

Resumen del direccionamiento absoluto

Los datos de memoria pueden ser accedidos (leídos), o almacenados (escritos), empleando direcciones absolutas como operandos origen o destino. La dirección absoluta puede ser especificada explícitamente por $6000, o por $FFFFFF, por ejemplo, o mediante etiquetas simbólicas convenientemente definidas.

El direccionamiento absoluto, incluso con etiquetas, no es lo bastante flexible para la mayoría de las aplicaciones. Una forma más conveniente es mediante los registros de direcciones, tal como se expone en el próximo apartado.

Direccionamiento absoluto por registros

La principal misión de los registros de dirección, como su propio nombre indica, es la de ofrecer las direcciones de memoria de los operandos. En este contexto, es importante la idea del **puntero.** Si en el registro A3 está el valor $3000, diremos que A3 apunta a la dirección de memoria $3000.

Para distinguir entre el puntero A3 y el operando apuntado por él, usaremos la sintaxis estándar de Motorola: A3 es el puntero y (A3) es el operando apuntado por A3. Los paréntesis de (A3) representan lo que denominamos *indirección.* A3 es un registro de direcciones directo, pero (A3) es una dirección sobre registro de dirección.

En la sección del "modelo de memoria" del capítulo 3 vimos cómo el 68000 usa direcciones de bytes, palabras y dobles palabras; entonces, ¿qué apunta A3 realmente? Si A3 contiene un número impar, tal como $3001, no hay error posible: A3 apunta al byte situado en la dirección $3001; pero si es par, tal como $3000, puede estar apuntando a cualquiera de los tres casos posibles. En esta posición podremos encontrar los siguientes valores:

Un byte en la dirección de byte A3 = $3000 sería (A3) = $E2
Una palabra en la dirección de palabra A3 = $3000 sería (A3) = $E278
Una doble palabra en la dirección de doble palabra A3 = $3000 sería (A3) = $E278B01C

Los valores del ejemplo carecen de importancia frente al modo en que están identificados. Así, antes de contestar a la pregunta: "¿Qué es (A3)?" deberemos conocer el tamaño del dato involucrado, sea este L, W, o B. Por ejemplo:

```
MOVE.L    (A3),D7 llevará $E278B01C a D7
MOVE.W    (A3),D7 llevará $E278 a la palabra inferior de D7
MOVE.B    (A3),D7 llevará $E2 al byte inferior de D7
```

Como podemos observar, el operando origen (A3) se comporta de forma muy similar a un registro de datos: los códigos B, W, o L determinan qué parte del operando es la afectada. Las diferencias importantes son: las operaciones sobre L y W en memoria exigen siempre direcciones pares. Las operaciones sobre B pueden tener dirección par o impar. Aquí debemos recordar las reglas establecidas sobre "dirección baja-byte alto, dirección alta-byte bajo" (véase el capítulo 3).

Rehagamos ahora el programa 4.5, empleando el modo indirecto con registros de dirección en lugar del absoluto.

APLICACION PRACTICA

Problema: Calcular el total de horas YTD trabajadas (desde enero hasta marzo) empleando operandos indireccionados en memoria. Colocar el total actualizado en la doble palabra de dirección $6008.

Datos:

1. El total de horas YTD (de enero y febrero) es una doble palabra situada en la dirección $6000.
2. Las horas trabajadas en marzo están en una doble palabra situada en la dirección $6004.

Solución: Programa 4.6

```
* Poner las direcciones en los registros de dirección

        MOVEA.L  #$6000,A1    A1 contiene la dirección de YTD
        MOVEA.L  #$6004,A2    A2 contiene la dirección de horas marzo
        MOVEA.L  #$6008,A3    A3 contiene la dirección de nuevo YTD

* Cálculo

        MOVE.L   (A1),D3      D3 = horas YTD
        ADD.L    (A2),D3      D3 = horas YTD + horas marzo
        MOVE.L   D3,(A3)      Poner D3 en la memoria de dirección A3 = $6008

* Ahora (A3) contiene la suma (A2) + (A1) = 458
* A1 y A2 permanecen inalterados
```

MOVEA: Llevar a registro de dirección

El programa 4.6 introduce un nuevo código de operación, MOVEA (MOVE Address), que no es más que una versión del MOVE empleada cuando el operando de destino es un registro de dirección. El formato general es:

```
MOVEA.L    <origen>,An
```

o

```
MOVEA.W    <origen>,An
```

Las direcciones del 68000 son valores de 32 bits (incluso aunque nuestro 6800X emplee nada más que 20 ó 24), de forma que no está permitido un MOVEA.B. Incluso el MOVEA.W es de 32 bits, puesto que el valor de 16 bits siempre es extendido a 32, con el bit de signo, según se explicó en el capitulo 3. Además, MOVEA, como las demás operaciones sobre registros de dirección, no afectan al CCR, puesto que no estamos interesados en acarreos, reboses, negativos, ceros, o positivos cuando manipulamos direcciones.

Aquí hemos empleado MOVEA con un origen de datos inmediato para preparar los registros de dirección. Una vez que A1, A2 y A3 están preparados, el programa utiliza (A1) y (A2) como operandos origen y (A3) como de destino.

Restricciones del ADD (suma)

Usted, quizá, se asombrará que hayamos usado D3 en el anterior programa. ¿Por qué no ahorramos una linea (y un registro) haciendo

```
MOVE.L    (A1),(A3)      Correcto
ADD.L     (A2),(A3)      Ilegal
```

La primera línea es correcta; llevará el contenido de la dirección $6000 a la memoria de dirección $6008. La segunda, por el contrario, es ilegal, porque ADD debe tener, por lo menos, un registro de datos, y SUB debe tener, también por lo menos, un registro de datos. El 68000 no permite emplear ADD, o SUB, con dos operandos de memoria. Así, podremos hacer

```
ADD.z     Dn,Dm          Correcto
SUB.z     Dn,Dm          Correcto

ADD.z     An,Dm          Correcto para z = L o W solamente
SUB.z     An,Dm          Correcto para z = L o W solamente
```

```
ADD.z     Dn,(Am)     Correcto
SUB.z     Dn,(Am)     Correcto

ADD.z     (Am),Dn     Correcto
SUB.z     (Am),Dn     Correcto
```

Pero no se puede hacer

```
ADD.z     (Am),(An)   Incorrecto
SUB.z     (Am),(An)   Incorrecto

ADD.z     Dn,Am       Incorrecto
SUB.z     Dn,Am       Incorrecto
```

Las reglas anteriores prohíben emplear An en ADD ni en SUB como destino: entonces, ¿cómo puede aumentarse o disminuirse una dirección en un registro de direcciones? Hay una forma, llamada ADDA (sumar a un registro de dirección: *ADD Address*). Abordemos el programa 4.6 desde un ángulo diferente, para mostrar la forma en que trabaja ADDA:

```
* Programa 4.6A: Solución alternativa del 4.6, empleando ADDA
* Preparación de las direcciones en los registros de dirección

MOVEA.W   #6000,A0    A0 contiene la dirección de YTD
MOVEA.W   A0,A1       A1 también la tiene ahora
ADDA.W    #4,A0       A0 = $6000 + 4 = $6004
MOVEA.W   A0,A2       A2 contiene la dirección de horas marzo
ADDA.W    #4,A0       A0 = $6004 + 4 = $6008
MOVEA.W   A0,A3       A3 tiene la dirección de nuevas YTD

* Cálculo: igual que 4.6

MOVE.L    (A1),D3     D3 = Horas YTD
ADD.L     (A2),D3     D3 = Horas YTD + Horas marzo
MOVE.     D3,(A3)     Poner D3 en la memoria de dirección A3 = $6008
```

Así, para sumar (ADD) algo a un registro de direcciones, empleamos ADDA de la misma forma que usábamos MOVEA para llevar algo a un registro de direcciones. El formato general es:

```
ADDA.L    <origen>,An
ADDA.W    <origen>,An
```

mientras que para substraer una cantidad de An tenemos:

```
SUBA.L    <origen>,An
SUBA.W    <origen>,An
```

Nótese, nuevamente, que el hecho fundamental en la vida del 68000 es que no están permitidas las operaciones de bytes con los registros An. Como en el caso de MOVEA, la manipulación con An no provoca alteraciones del CCR.

El punto clave del programa 4.6A es la forma en que hemos usado ADDA con el puntero A0. Añadiendo 4 a un registro de dirección que inicialmente contenga un valor par, conseguimos apuntar a la próxima doble palabra de la memoria. De forma similar, si añadiéramos 2 ó 1, el puntero se "desplazaría" para apuntar a la siguiente palabra o al siguiente byte, respectivamente. Es muy frecuente encontrarse con que los datos que se manejan están almacenados en la memoria como secuencias o tablas; de forma que ADDA o SUBA son útiles para "posicionar" los punteros de los registros de dirección con objeto de barrer las tablas de datos en ambos sentidos. Esta forma de actuar es tan común que el 68000 ofrece dos variantes especiales del modo de direccionamiento en (An) para simplificar el barrido de las direcciones consecutivas. Los modos nuevos incrementan o decrementan automáticamente el puntero An. Veamos, en primer lugar, el modo indirecto de registro de dirección con posincremento.

Direccionamiento indirecto con posincremento: (An)+

El modo de direccionamiento indirecto con posincremento, que se escribe (An)+, se explica mejor con un ejemplo:

```
* Programa 4.6B: Solución alternativa a $-6, usando (An)+
* Preparación de dirección del primer valor de la tabla
        MOVEA.W  #$6000,A1      A1 contiene la dirección de las horas YTD, es decir,
                                apunta a horas YTD
* Cálculo
        MOVE.L   (A1)+,D3       D3 = ($6000) = horas YTD, y después se suma 4 al puntero
                                A1
        ADD.L    (A1)+,D3       Se suma ($6004) a D3 y después se suma 4 al puntero A1
        MOVE.L   D3,(A1)        Se guarda D3 en la memoria de dirección A1 = $6008
```

En lugar de emplear tres registros de dirección para los operandos, esta solución emplea solamente A1, de forma que el posincremento se encarga de apuntar a la doble palabra siguiente tras cada operación. El autoincremento es más "económico" que la instrucción ADDA #4,A1 y, además, nos evita la complicación de tener que saber de cuánto ha de ser el incremento, que él hace automáticamente.

(A1)+ incrementará A1 en 4, 2 ó 1, según sea el código del tamaño de dato empleado en el propio código de la operación. Por ejemplo:

```
MOVE.W    (A2)+,D5    Hacer D5 = palabra (A2), y después sumar (ADD) 2 a A2
```

acabará haciendo que A2 apunte a la siguiente palabra en (A2 + 2), y

```
MOVE.B    (A2)+,D5    Hacer D5 = byte (A2), y después A2 = A2 + 1
```

hará que A2 apunte al siguiente byte en (A2 = 1).

Tempus Fugit: Un intervalo dedicado a los cronogramas

Dado que ya hemos visto varias maneras de realizar la misma sencilla operación de sumar las horas YTD, resultará muy útil compararlas desde el punto de vista de los tiempos (cronogramas) empleados en ello. En nuestro sencillo contexto, no pararemos mientes en unos pocos microsegundos de más o de menos, pero, desde una perspectiva amplia y práctica, es importante preguntarse sobre cómo lo hace el 68000 (en la siguiente tabla deberá tenerse en cuenta que el *bus* de 8 bits del MC68008 precisará de más ciclos, mientras que los 32 del *bus* de 68020 le permitirán realizarlo en menos).

Programa 4.1

Directo de registros:

Tomar los datos de los registros: muy rápido, sin acceso a memoria. Almacenar los resultados en un registro: muy rápido, pero, ¿cómo se llevan los datos a esos registros? Y ¿cómo se imprimirán los resultados? Antes o después necesitaremos acceder a memoria.

Programa 4.5
(+ variantes)

Direccionamiento absoluto:

Tomar la dirección de los datos de las palabras de extensión: significa una o dos lecturas en memoria. Después, tomar el propio dato: significa una o dos lecturas de memoria. Tomar la dirección para almacenar el resultado: significa una o dos lecturas de memoria. Almacenar la doble palabra resultado: significa dos escrituras en memoria.

Programa 4.6

Direccionamiento indirecto:

Disponer tres registros con datos inmediatos: implica de tres a seis lecturas de memoria. Tomar

las direcciones de An: muy rápido. Entonces, leer los datos: significa una o dos lecturas en memoria. Almacenar el resultado: emplea dos escrituras en memoria.

Programa 4.6A Direccionamiento indirecto mediante ADDA:

Preparar un registro de direcciones con un dato inmediato: significa una o dos lecturas en memoria. Ejecutar un ADDA inmediato dos veces: supone dos lecturas en memoria. Tomar las direcciones de An: muy rápido. Entonces, tomar los datos: supone una o dos lecturas en memoria. Almacenar la respuesta: significa dos escrituras en memoria.

Programa 4.6B Direccionamiento indirecto con posincremento:

Preparar un registro de direcciones con un dato inmediato: significa una o dos lecturas en memoria. Generar la dirección con (An) + : muy rápido. Entonces, tomar el dato: supone una o dos lecturas en memoria. Almacenar el resultado: implica dos escrituras en memoria.

Así, el modo indirecto con posincremento parece ofrecer el mejor método global, con tal de que los datos estén convenientemente colocados de forma secuencial en la memoria. Una situación típica en que los datos ocupan de forma natural direcciones sucesivas es la del procesamiento de textos, donde se deben manejar largas **ristras** o secuencias de caracteres ASCII, cada uno de los cuales exige un byte de memoria. A menudo se critica al 68000, arguyendo falta de instrucciones explícitamente dedicadas al manejo de ristras de datos. El ejemplo siguiente demuestra lo contrario.

Manejo de ristras de datos con (An) +

He aquí un ejemplo de la potencia de (An) + . El problema resultará familiar a todos aquellos lectores que hayan tenido que mover o copiar un bloque de texto en un procesamiento de textos.

```
* Programa 4.7: Copiar una ristra de caracteres desde una posición de memoria a otra

* A1 apunta al primer carácter ASCII del bloque de texto almacenado en la memoria
* Se supone que el último carácter es el ASCII "nulo" (0)
* Se quiere copiar en el bloque de texto, incluyendo el nulo final en otra parte de la memoria
* de dirección inicial contenida en A2
* Si hay algún daño anterior en (A2), puede ser borrado
* ¡A2 es mayor que <A1 + tamaño del bloque>!
```

```
* Si el bloque está vacío (esto es, si empieza con el carácter NULO), no nos molestamos
* en moverlo

LAZO  TST.B    (A1)          ¿Hemos alcanzado un Nulo?
                             Prueba del byte en (A1) = 0
      BEQ      FINI          Si es así, saltar a FINI
      MOVE.B   (A1)+,(A2)+   Llevar el byte de A1 al byte de A2. Incrementar
                             A1 y A2 en 1 para apuntar al siguiente byte
      BRA      LAZO          Volver a LAZO para analizar el siguiente byte
FINI  <resto del programa>

* ATENCION: recuérdese que A1 y A2 habrán cambiado su contenido salvo que el primer
* carácter en (A1) sea un NULO
```

Podemos observar cómo (A1)+, como byte origen, y (A2)+, como destino, avanzan como punteros (el puntero A1 como "enviador" y el A2 como "receptor") en cada carácter de la ristra. Si no se hiciera una comprobación para detectar el final de la ristra, MOVE.B(A1)+,(A2)+, continuaría por toda la memoria disponible con resultados realmente extraños. Así, el sencillo TST.B introducido anteriormente nos evitará el alterar todo el contenido de la RAM.

```
TST.z     operando
```

comprueba si el tamaño z = L, W, o B del operando es negativo o zero (el zero no es error, es para poner en evidencia su sentido), y, según sean, se pondrán los respectivos indicadores Z y N del CCR. El indicador N carece de importancia en este ejemplo particular. Así,

```
TST.B     (A1)
```

hace la pregunta: "¿Es el byte en la memoria de dirección A1 = 0?". Si es así, el *flag* Z se pone a 1; de lo contrario se pone a 0. Para el 68000, los códigos ASCII carecen de interés por sí mismos: es nuestro problema el interpretar los 8 bits de cada byte de nuestra ristra.

El TST.B funciona bien suponiendo que el bloque se termina con un carácter ASCII NULO que en binario es 00000000 (llamado a veces **blanco**, que no debe confundirse con el "espacio" ASCII, que es 00100000, o con el "cero" ASCII, que es 00110000). Todo lo que debe recordarse es que el NULO es un carácter ASCII como otro cualquiera y que ocupa un byte de nuestra preciosa memoria. Por ello, nuestra TST.B busca un 0.

La instrucción BEQ tras la TST.B comprueba el valor del CCR y provoca un salto a FINI solamente si el indicador Z está a 1, es decir, si el byte de (A1) es un NULO.

Obsérvese también que hemos efectuado un TST.B al principio del programa. Si en el primer byte de (A1) es un NULO, saltaremos inmediatamente a FINI sin efectuar ningún movimiento de datos. Los programadores se entretienen ponderando la cuestión de si merece la pena copiar una ristra

vacía, es decir, aquella que empieza (y termina) con un NULO. Evidentemente, debe distinguirse entre un bloque vacío y ningún bloque en absoluto. El programa 4.7 ignora los bloques vacíos, puesto que se salta *antes* de copiar el NULO. No tiene ninguna dificultad el reescribir el programa 4-7, de forma que se copie el NULO de (A1) en (A2). En muchas facetas de la vida podría ser tachado de estrafalario o metafísico, pero en programación de ordenadores tales detalles pueden ser de vital importancia. Una buena razón para no copiar el NULO en (A2) podría ser que se puede desear copiar más texto detrás del actual o, dicho de forma más correcta, con palabras impresionantes, **concatenar** con otros bloques. Si, a pesar de todo, usted insiste en una copia exacta de un bloque no vacío que incluya el NULO final, aquí tiene el programa 4.8:

```
* Programa 4.8: Copiar una ristra de caracteres desde una posición de memoria a otra
* A1 apunta al primer carácter ASCII del bloque de texto de memoria
* Se supone que el último carácter del bloque es el ASCII NULO
* Se desea copiar todo el bloque, incluyendo el NULO final en otra parte de la memoria
* que empieza en la dirección A2. Se puede borrar cualquier dato que hubiera en (A2)
* A2 es mayor que <A1 + tamaño del bloque>
* Si la ristra en A1 esta vacia (es decir, empieza con un NULO), no se moleste en moverlo

        TST.B   (A1)            ¿Es el primer byte un NULO?
        BEQ     FINI            Si lo es, saltar a FINI porque el bloque está vacío
LAZO    MOVE.B  (A1)+,(A2)+     Llevar el byte de A1 a A2. Incrementar A1 y A2
                                para apuntar al siguiente byte en memoria
        BNE     LAZO            Si el byte movido no es el NULO, hay que seguir
                                copiando
FINI    <resto del programa>

* Recuérdese que A1 y A2 pueden resultar alterados
```

Resumen del modo (An)+

El modo (An)+, tanto como origen o como destino, o como ambos, es la forma más eficaz de manipular posiciones sucesivas ordenadas de menor a mayor. Se prepara An para apuntar a la dirección inicial y, escogiendo el tamaño de la operación (L, W, o B), se permite al 68000 que vaya incrementando correctamente el puntero.

Antes prometimos dos formas de barrer direcciones consecutivas, por lo que, habiendo visto cómo (An)+ va *hacia delante*, ahora presentamos el modo inverso, −(An), para acceder a la memoria *hacia atrás*.

Direccionamiento indirecto de registros de dirección con predecremento: –(An)

El modo indirecto de registros de dirección con predecremento está íntimamente relacionado con el modo (An)+, y se escribe –(An). El puntero se reduce o **decrementa** por 4, 2 ó 1 antes de que se realice la operación. De la misma forma que en (An)+, el cambio viene determinado por el código de tamaño empleado de los datos en la instrucción. He aquí un ejemplo sencillo:

```
* A5 contiene $8008 al principio

    CLR.L   –(A5)   Reducirá A5 en 4, y entonces borrará la doble palabra ($8004)
```

Sin embargo,

```
    CLR.W   –(A5)   Reducirá A5 en 2, y entonces borrará la palabra ($8006)

    CLR.B   –(A5)   Reducirá A5 en 1, y entonces borrará el byte ($8007)
```

Empleando –(An), se pueden barrer las tablas desde el final hasta el principio (lo que a veces resulta más rápido), con tal de que nos acordemos de preparar el registro para que apunte *exactamente más allá* del final de la tabla para que funcione el *predecremento*. (An)+ y –(An) funcionan de forma muy similar a poco que lo pensemos. (An)+ deja el puntero en posición, tras el anterior paso, un lugar más allá del final: listo para la búsqueda contraria con –(An). Esta idea se emplea en los procesadores de textos tales como el WordStar, que le permite una búsqueda adelante y hacia atrás en el documento.

Resumen del modo–(An)

El modo –(An), tanto como destino u origen, o como ambos, es la forma más eficaz de manipular posiciones sucesivas de memoria en direcciones de mayor a menor. Se prepara An de forma que apunte justamente más allá de la posición mayor y, tras haber escogido el tamaño de los datos (L, W, o B), se deja al 68000 que vaya decrementando correctamente el puntero antes de cada operación.

Conclusión

Concluimos este capítulo con un breve repaso a los modos de direccionamiento que hemos visto. En el capítulo 5 discutiremos los usos avanzados de estos modos e introduciremos nuevos modos e instrucciones.

Descripción de los modos:

Dn	Directo a registro de datos.	Llamados ambos directo a registro.
An	Directo a registro de dirección.	
(An)	Indirecto de registro de dirección.	
(An) +	Indirecto a registro de dirección con posincremento.	
−(An)	Indirecto a registro de dirección con predecremento.	
Inmed	Operando inmediato: también escrito #(dato).	
Abs.W	Dirección absoluta corta (16 bits con extensión de signo): también se escribe xxx[.W] o etiqueta.	
Abs.L	Dirección absoluta larga (32 bits): también se escribe xxx[.L] o etiqueta.	

5

Set de instrucciones del MC68000: Conceptos avanzados

En este capítulo emplearemos las instrucciones y modos de direccionamiento descritos en el capítulo 4. El primer paso consiste en enfrentarse a las labores de *housekeeping* [1], es decir, mantener un control real de la información contenida en la memoria o los registros de datos y direcciones. Esta materia proporciona una ocasión de explorar de forma práctica las instrucciones y las posibilidades del M68000.

Preservando el valor de los registros: Cómo y por qué

Con los modos (An), (An)+ y −(An) nos enfrentamos al problema que se presenta en un programa largo, cuando empiezan a acabarse los registros disponibles. Aunque el M68000 proporciona 16 registros muy versátiles, lo que es bastante más de lo que ofrecen la mayoría de los microprocesadores,

[1] Hemos conservado el término *housekeeping* por varias razones. En primer lugar, su traducción por "labores domésticas" o algo similar nos parece excesivamente forzada. En segundo lugar, este término aparece con frecuencia en los manuales de referencia técnica para indicar un conjunto de rutinas que realizan labores generalmente repetitivas, pero fundamentales para el mantenimiento del sistema. Finalmente, el término puede ilustrar el particular sentido del humor que los americanos han esparcido sobre toda la jerga informática.

se puede alcanzar una situación en la que los registros de D0 a D7 contengan resultados intermedios importantes y los registros de A0 a A7 almacenen determinados punteros que no deseamos perder. Supongamos, por ejemplo, que nos embarcamos en una operación para copiar una cadena de caracteres mediante una instrucción:

```
MOVE.B (A3)+,(A0)+
```

De este modo, como vimos en el capítulo 4, perderemos los valores iniciales de A3 y A0, puesto que al final de la operación A3 y A0 apuntarán a una dirección determinada por la cadena de caracteres, que a menudo será impredecible. Obviamente, podemos salvar los valores de los registros A0 y A3 escribiendo su contenido en una posición de la memoria y restaurarlo al finalizar el proceso leyendo ésta, como se demuestra en el programa:

```
* Programa 5.1
* Queremos copiar la cadena (A3) → (A0) sin perder los punteros en A3 y A0

        MOVE.L  A3,$4004        Salvamos A3
        MOVE.L  A0,$4000        Salvamos A0
                                <copiamos la cadena de caracteres>
        MOVE.L  $4000,A0        Restauramos A0
        MOVE.L  $4004,A3        Restauramos A3
```

El mismo "truco" puede emplearse para salvar y restaurar los valores de los registros Dn. Esto nos permite lograr nuestro propósito, pero hay algunos "pequeños problemas". En primer lugar, este truco puede plantear problemas en programas largos ("¿Dónde puse yo A3 y D7...?"). Podemos, además, borrar inadvertidamente los valores almacenados (al escribir sobre ellos durante alguna operación en el programa). Por último, hay que prestar una notable atención a las direcciones con las que trabajamos (pares o impares) según salvemos dobles palabras, palabras o bytes.

El M68000 proporciona dos métodos para simplificar las operaciones anteriores: la instrucción MOVEM y la pila de usuario.

MOVEM: Mover varios registros

MOVEM es una versión especial de MOVE que permite salvar varios registros de una forma rápida y fácil en un grupo de posiciones consecutivas de la memoria y restaurarlos más tarde, cuando se necesiten.

Para salvar registros se emplea el formato:

```
MOVEM.Z   <lista de registros>,<destino>
```

y para restaurarlos se emplea el formato:

```
MOVEM.Z   <fuente>,<lista de registros>
```

Nótese que Z sólo indica L o W, de modo que no está permitido emplear MOVEM.B.

La <lista de registros> puede indicar hasta 16 registros diferentes (de A0 hasta A7 y de D0 hasta D7) para salvar o restaurar, mientras que el <destino> y el <origen> indican la posición de la memoria en que comienza la lista. MOVEM.L transfiere 32 bits (una doble palabra completa), mientras que MOVEM.W sólo transfiere la palabra menos significativa, empleando determinados convenios de extensión del signo al restaurar los registros. El programa 5.1 puede escribirse:

```
* Programa 5.1A
* Salvar y restaurar varios registros en una dirección empleando MOVEM
* Queremos copiar la cadena (A3) → (A0) sin perder los punteros en A3 y A0

        MOVEM.L  A0/A3,$4000    Salvamos A0 y A3 en $4000 y $4004
                                <copiamos la cadena de caracteres>
        MOVEM.L  $4000,A0/A3    Restauramos A0 y A3 desde $4000 y $4004
```

Nótese cómo los registros en la <lista de registros> se separan empleando una barra (/). Para salvar registros consecutivos se puede emplear el formato:

```
        MOVEM.L  D0-D5/A4-A6,$6000
```

que salvará 9 registros: los 6 registros de datos de D0 a D5 y los 3 registros de direcciones de A4 a A6. Los 9 registros se almacenarán en la memoria, ocupando 9 dobles palabras en las posiciones $6000 a $6020. Para restaurarlos emplearemos el formato:

```
        MOVEM.L  $6000,D0-D5/A4-A6
```

Hasta ahora, hemos estado salvando los valores de los registros en una dirección absoluta, pero también se puede emplear un puntero (siempre que quede algún registro de direcciones An libre), empleando el modo de direccionamiento indirecto con predecremento –(An), en cuyo caso debemos restaurarlos empleando el modo (An) + , o direccionamiento indirecto con posincremento. De nuevo observamos como estos dos modos de direccionamiento son complementarios. El programa 5.1B emplea esta variante:

```
* Programa 5.1B
* Salvar y restaurar varios registros empleando los modos de direccionamiento
* indirecto –(An) y (An) +
* Se desea mover una cadena de caracteres sin perder por ello los valores
* de los punteros A0, A3. A5 apunta a la última dirección empleada en el área
* de la memoria destinada a almacenar varios valores temporalmente
* Suponer que A5 = $4008

        MOVEM.L  A0/A3,–(A5)
```

```
* Con la instrucción anterior reducimos A5 en 4. Salvamos A3, comenzando en $4004
* Reducimos de nuevo A5 en 4. Ahora A5 = $4000
* Ahora salvamos A0
* Nótese cómo MOVEM invierte el orden en que se efectúa el salvamiento!

        <copiar la cadena de caracteres>

        MOVEM.L  (A5)+,A0/A3

* Restauramos el valor de A0 desde $4000. Incrementamos A5 en 4 y restauramos A3
* desde $4004. Incrementamos A5 en 4. A5 recupera su valor inicial: $4008
```

La instrucción MOVEM con los modos −(An) y +(An) es muy flexible. Como se verá en la sección siguiente, es muy similar al concepto de pila.

Pilas

Emplear la pila como solución para el problema de salvar y restaurar registros requiere un corto preámbulo para hablar de la jerga de las pilas y de la mística de las mismas.

Como se muestra en las figuras 5.1 y 5.2, el registro de direcciones A7 se denomina USP (puntero de la pila de usuario) y se emplea para apuntar a una determinada zona de la memoria, denominada pila de usuario. Esta pila "crece" hacia abajo a partir de su base, desde las direcciones más altas de la memoria hacia las más bajas, a medida que salvamos datos; y se reduce hacia arriba, hacia la base de la pila, desde las direcciones más bajas hacia las más altas.

Quizá ayude este párrafo si se lee con atención. Todo esto nos recuerda aquella famosa caja que llegó desde Dublín (Irlanda) con la siguiente inscripción: "Esta caja debe permanecer siempre invertida. Para evitar cualquier confusión, la parte inferior se ha marcado como 'arriba'."

El término más empleado para salvar datos en la pila es el de "meter" los datos en la pila, mientras que la acción de restaurarlos se indica mediante "sacar" los datos de la pila [2]. Las pilas son dispositivos LIFO (el último en entrar es el primero en salir), mientras que una cola es un dispositivo FIFO (el primero en entrar es el primero en salir).

No hay ninguna duda acerca del lugar en que se está salvando un dato en la pila. A7, el puntero de la pila, siempre apunta al último dato salvado, que es el primer candidato a ser restaurado. La secuencia para meter, por ejemplo, D1 en la pila es:

---

[2] Hemos optado por traducir *stack* por pila, *push* por meter y *pull* por sacar, que, aunque no sean las traducciones más correctas, reflejan claramente los conceptos que queremos ilustrar. Desgraciadamente, los términos ingleses en este campo están impregnados de un humor muy particular (como ya se ha indicado), así como de dobles sentidos muy difíciles de traducir.

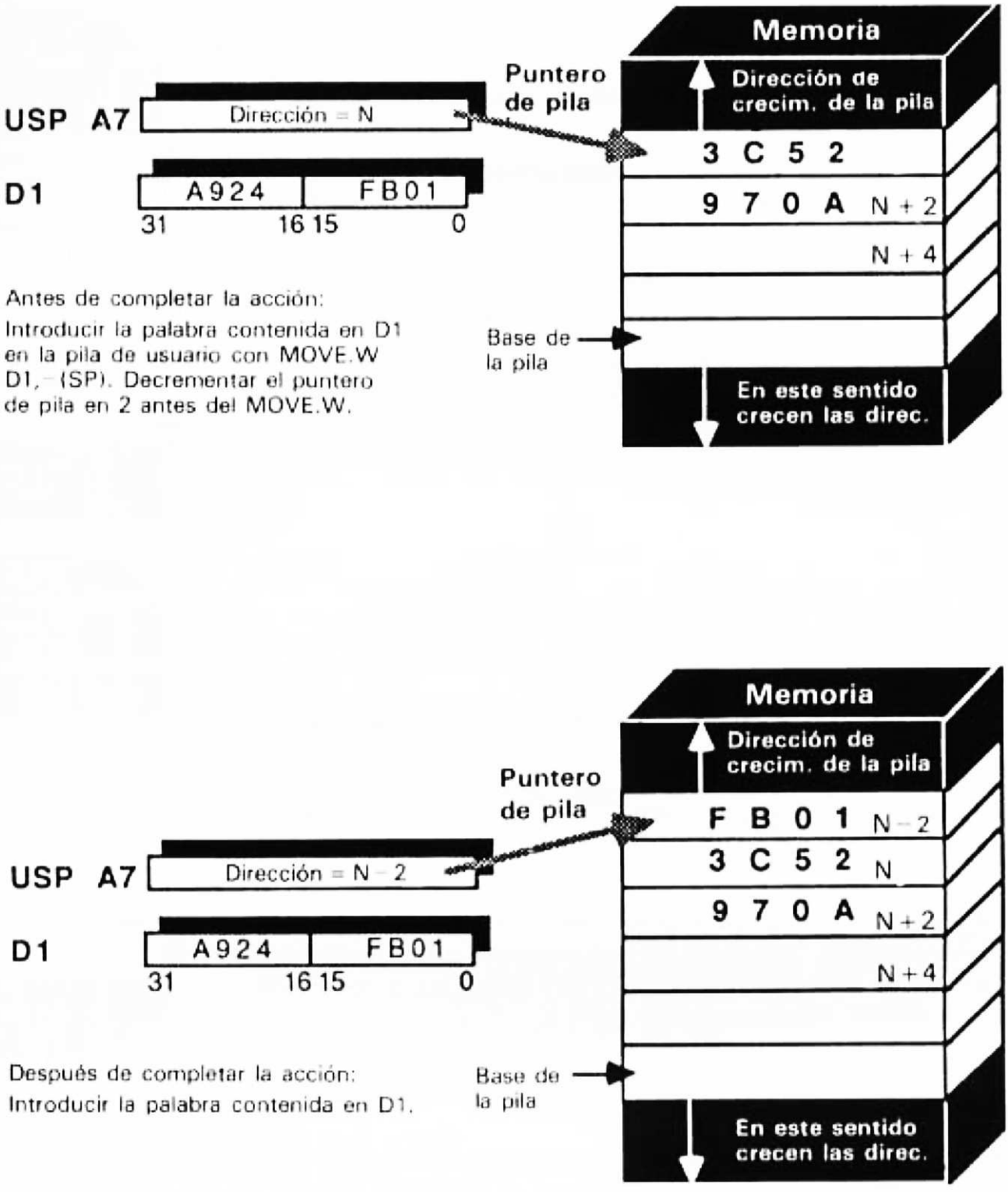


Figura 5.1
Manejo de la pila: introducir datos

MOVE.L D1,–(A7) Meter D1 en la pila

* Predecrementamos A7 (en 4, pues estamos tratando con palabras dobles)
* oprque nos movemos hacia abajo en la memoria
* (Recordemos que la pila crece hacia abajo)
* Entonces metemos D1 en la nueva dirección citada por A7;
* el puntero de la pila apunta ahora al elemento recién salvado

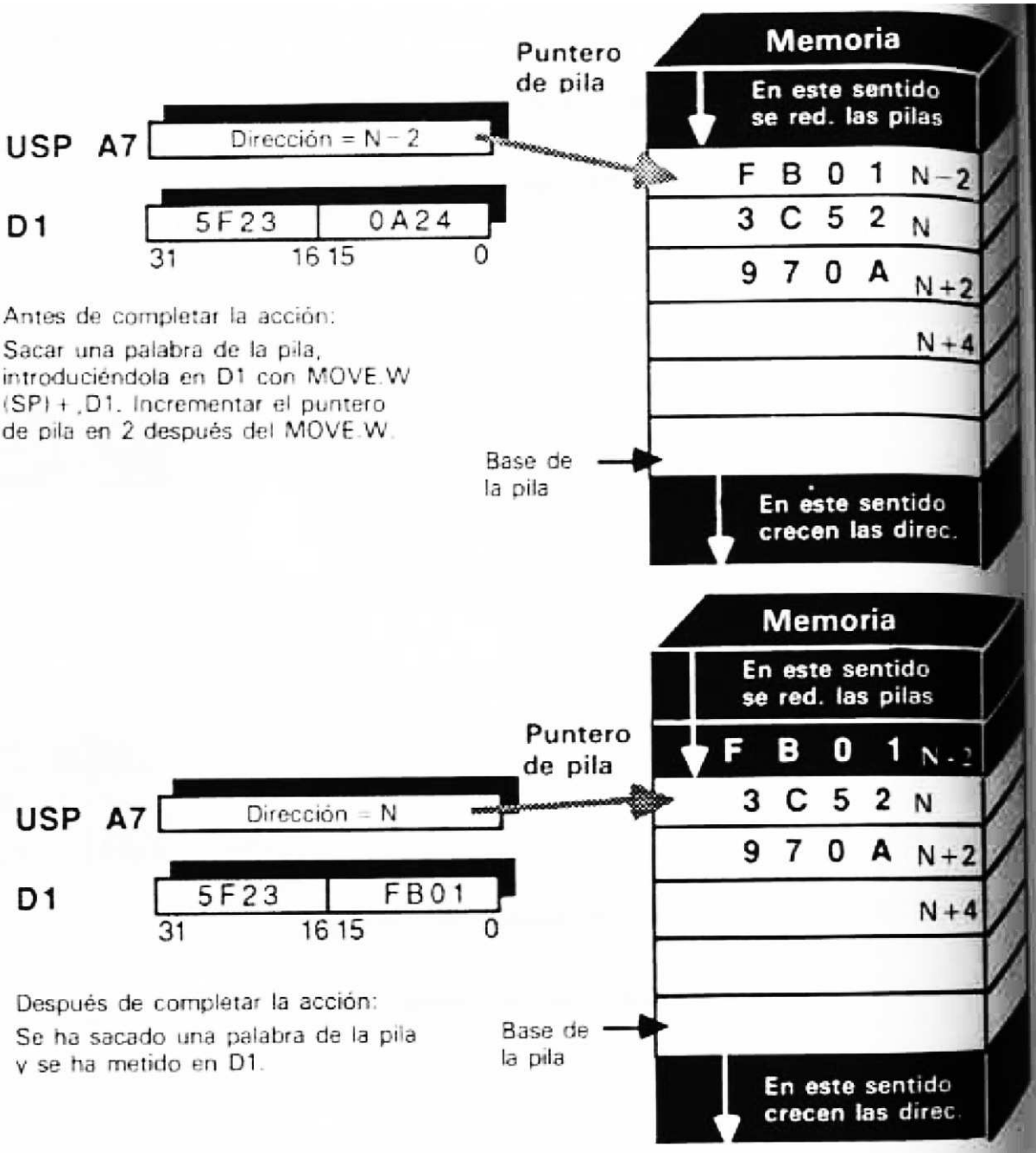


Figura 5.2
Manejo de la pila: sacar datos

Para sacar D1 de la pila se emplea:

```
MOVE.L  (A7)+,D1    Restaurar D1 desde la pila
```

* El valor previo de D1 está almacenado en (A7); así pues, se lleva (A7) a D1
* y se incrementa A7 en 4 para reducir la pila. (A7) apunta ahora al dato
* (si es que hay alguno) que entró antes de D1

La sintaxis estándar permite emplear el mnemónico SP (*Stack Pointer*: puntero de pila) en lugar de A7:

```
MOVE.L  D1,-(SP)    Meter D1 en la pila
MOVE.L  (SP)+,D1    Restaurar D1 desde la pila
```

MOVEM y la pila

Se puede emplear MOVEM para salvar varios registros en la pila:

```
MOVEM.L  D0-D3/A0-A6, -(SP)
```

de este modo, salvamos los valores de 11 registros, y con

```
MOVEM.L  (SP)+,D0-D3/A0/A6
```

los recuperamos.

MOVEM, sin embargo, permite salvar sólo los registros (L o W), mientras que MOVE permite salvar datos de la memoria en la pila, incluyendo bytes.

Metiendo bytes en la pila

Cuando se introduce un byte en la pila, por ejemplo:

```
MOVE.B  D2, -(SP)    Salvar el byte menos significativo de D2 en la pila
```

el M68000 tiene implementado un interesante truco que impide que los números impares entren en juego si más tarde deseamos meter una palabra o una doble palabra en la pila. La figura 5.3 muestra cómo funciona este truco.

Normalmente, la instrucción MOVE.B predecrementa y posincrementa An en 1, pero cuando trabaja con el puntero de la pila SP (= A7), el procesador cambia el puntero en 2 unidades para mantener las direcciones como números pares. Todos los datos de la pila quedan alineados como si de palabras se tratara. Cuando metemos un byte en la pila, éste se aloja en el byte más significativo de la palabra de la pila, mientras que el menos significativo se desperdicia.

Una vez que hemos visto cómo funciona la pila del M68000, vamos a ver una situación en la que el M68000 la emplea por su cuenta, sin hacer uso explícito de una instrucción MOVE. Primero necesitamos entender el concepto general de subrutina.

Subrutina: Una definición breve

Una subrutina es una parte del programa construida de modo que puede llamarse desde cualquier parte del programa principal y, una vez que ha concluido la tarea que tenia que realizar, devuelve el control al lugar del programa desde donde se la invocó. Las subrutinas tienen una etiqueta que las identifica, y llamar a una subrutina es muy parecido a efectuar una bifurcación *(branch)*. Sin embargo, a diferencia de las bifurcaciones, que

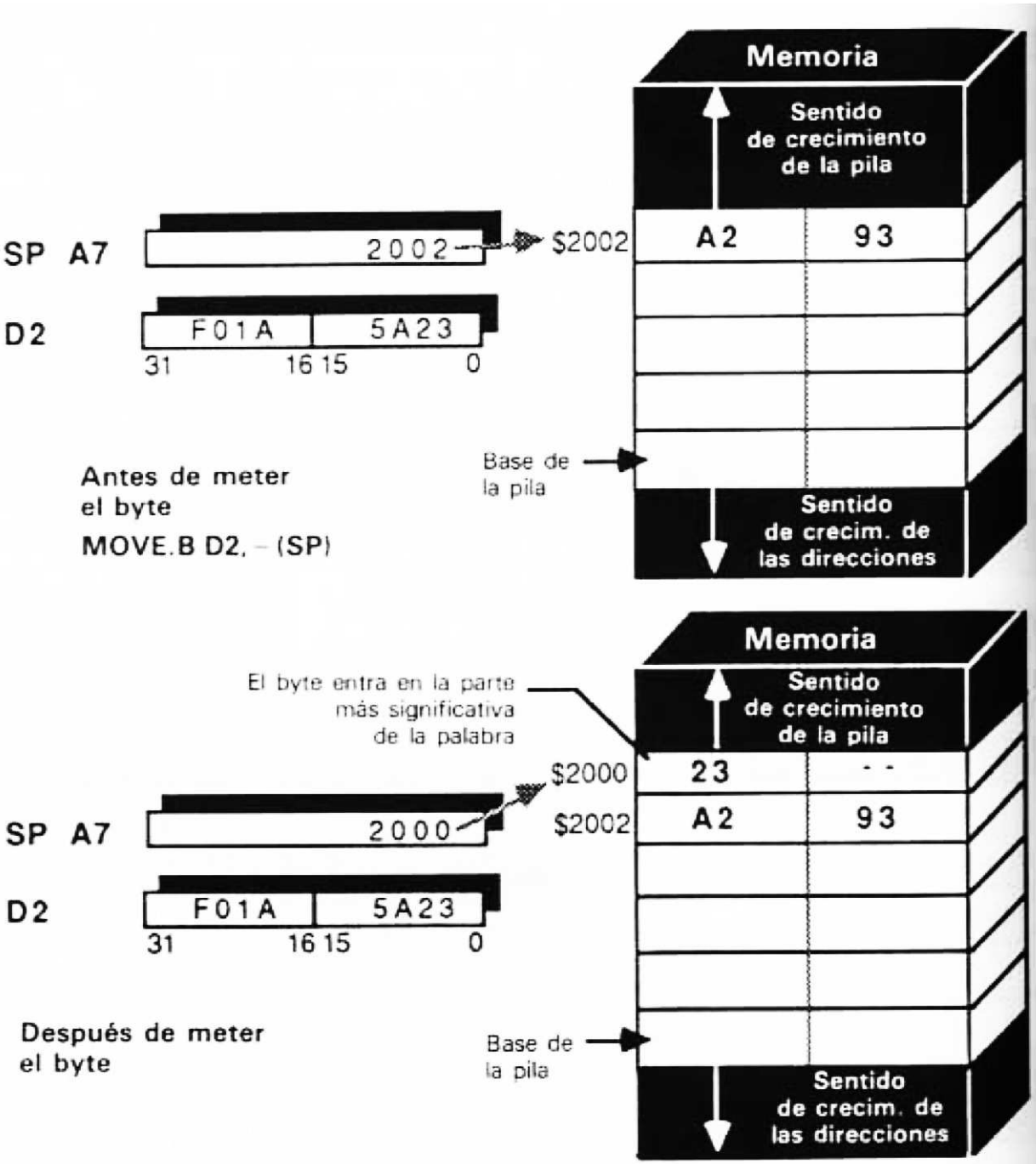


Figura 5.3
Funcionamiento de la pila con bytes

simplemente nos llevan a un lugar en el programa, llamar a una subrutina requiere un mecanismo que permita al sistema recordar la línea en la que se produjo la llamada, de modo que, cuando la subrutina concluye, el sistema sepa adónde retornar.

Las subrutinas son vitales para reducir la cantidad de código que hay que escribir y depurar. Cualquier secuencia de instrucciones que se emplee varias veces puede codificarse en forma de subrutina y luego llamarse tan a menudo como se necesite desde cualquier línea en el programa principal. Veamos cómo se llama a una subrutina y cómo el M68000 involucra en ello de forma automática a la pila, para asegurar que la dirección de retorno queda a salvo.

BSR: Saltar a una subrutina

Las subrutinas se llaman con una instrucción BSR. El formato empleado para BSR es el mismo que para la instrucción Bcc:

BRS <etiqueta> Saltar a la subrutina que comienza en la <etiqueta>

En la línea de la <etiqueta> encontraremos la subrutina codificada como cualquier otra parte del programa, pero concluida siempre con un RTS (retorno desde la subrutina). Aquí está, paso a paso, la secuencia que inicia un BSR:

1. Calcular la dirección de la siguiente instrucción y meterla en la pila de usuario.
2. Saltar (incondicionalmente, como en el caso de BRA) a la instrucción marcada con la <etiqueta>, haciendo que el PC apunte a la dirección de la <etiqueta>.
3. Se ejecutan, normalmente, las instrucciones que comienzan en la <etiqueta> hasta encontrar un RTS.
4. Se carga el PC, sacando de la pila la dirección que se salvó en el paso 1. De hecho, el procesador efectúa internamente un MOVE.L (A7) + ,PC.
5. El procesador ejecuta la instrucción cuya dirección se encuentra en el PC, de modo que el control se ha devuelto a la instrucción que seguía a la instrucción BSR.

El trío BSR, <etiqueta> y RTS se combinan para construir una técnica a la que nos referiremos como llamar a una subrutina. La etiqueta debe ser un mnemónico, puesto que normalmente hablaremos en términos de llamar a la subrutina "etiqueta". Una subrutina ahorra trabajo de programación y reduce el tamaño del programa, ahorrando, por tanto, memoria.

Ni la instrucción BSR ni la instrucción RTS afectan al estado de los indicadores del CCR, aunque seguramente las instrucciones contenidas en la subrutina los emplearán y alterarán. Veamos cómo funciona la instrucción BSR:

```
* Programa 5.2. Llamar a la subrutina ACUMU
* Programa para sumar varios números que están en la memoria
* D1 se empleará como acumulador. La subrutina ACUMU sumará D0 y D1

COMIEN  CLR.L   D1      Comenzamos. Poner a cero el acumulador
        *   *   *       <otras cosas>
PRINCI  MOVE.L  (A1),D0 Programa principal. Ponemos el valor de (A1)
                        en D0
        BRS     ACUMU   Llamamos a la subrutina ACUMU
        MOVE.L  (A2),D0 Ponemos el valor de (A2) en D0
        BRS     ACUMU   Llamamos a la subrutina ACUMU otra vez
```

```
        <imprimimos en el total que hay en D1>
        <terminar>
                *       *       *

* La sección de las subrutinas comienza aqui

ACUMU   ADD.L   D0,D1       Esta es una subrutina de una sola linea
                             llamada ACUMU
        RTS
* ACUMU entrada = doble palabra D0 que no cambia
        salida  = doble palabra D1 = D1 + D0
<posiblemente otras subrutinas siguen aqui>
                *       *       *
```

Normalmente las subrutinas son más largas y más útiles que el ejemplo que se da aquí. Sin embargo, esta subrutina nos sirve para ilustrar los principios básicos de su manejo.

Subrutinas: Parámetros de entrada y salida

Normalmente se asignan valores a los parámetros de una subrutina mediante MOVE antes de saltar a la misma con una instrucción BRA. Cada subrutina debe tener un conjunto de parámetros de entrada. ACUMU tiene sólo un parámetro de entrada D0 y otro de salida D1. Para optimizar el uso de las subrutinas, estos parámetros y su uso deben estar perfectamente documentados. Las subrutinas se denominan programas de propósito general o utilidades. Una vez que se han comprobado, pueden añadirse a una librería de subrutinas accesible a cualquiera que emplee el sistema. Muchos ensambladores permiten que dicha librería sea escrutada durante el ensamblado; cualquier subrutina llamada durante el programa se copia automáticamente. El propósito que se persigue es el de "no reinventar la rueda". Una vez que se comprende lo que hace una subrutina particular, se emplea ésta como si de una sola instrucción se tratase, sin preocuparse por los detalles internos de la misma.

Efectos secundarios de las subrutinas sobre los registros

Una subrutina de propósito general bien diseñada debe proteger al usuario de efectos secundarios no deseados. Una subrutina compleja puede hacer uso de muchos registros y, a menos que se tomen medidas, sus valores se perderán. La pila de usuario parece un lugar apropiado para salvar, y más tarde restaurar, estos valores aun a pesar de que BSR y RTS empleen la pila para salvar y recuperar la dirección de retorno de la subrutina (pasos 1 a 4, citados anteriormente). La filosofía LIFO de la pila permite realizar sucesivos salvamentos y restauraciones siempre que éstos se efectúen en el

orden correcto. Seguidamente se da un ejemplo de programa que emplea la pila en una subrutina:

```
* Programa 5.3: Subrutina PONER0

* A0 apunta a un área de memoria que deseamos poner a cero
* D0 contiene el número de palabras que hay que poner a cero
* La subrutina PONER0 no debe alterar A0 ni D0

PRINCI  <comienzo del programa>
        BSR     PONER0  Salvar la dirección de retorno
                        Saltar a PONER0
        <el programa continúa aquí>
        *   *   *

* Sección de las subrutinas

PONER0  MOVE.L  D0,-(SP)    Salvar D0 en la pila
        MOVE.L  A0,-(SP)    Salvar A0 en la pila
BUCLE   CLR.W   (A0)+       Poner (A0) a cero
                            Incrementar A0 en 2.
        SUBQ.L  #1,D0       Decrementar el contador
        BNE     BUCLE       Cuando el contador sea cero,
                            abandonar el bucle
        MOVE.L  (SP)+,A0    Restaurar A0
        MOVE.L  (SP)+,D0    Restaurar D0
        RTS                 Recobrar la dirección
                            de retorno y volver
        <aquí vendrán otras subrutinas>
        *   *   *
```

Cuando se restauran valores de la pila (implícita o explícitamente), se hace en orden inverso a como se salvaron en la misma, dejándola igual que se encontraba antes del BSR.

Efectos secundarios de las subrutinas en el CCR

Es casi seguro que los indicadores del CCR se alterarán durante la ejecución de una subrutina, y esto puede producir problemas en el programa principal. Muy a menudo se prueba un valor para llamar a una de las tres subrutinas, según éste resulte positivo, negativo o cero. Al volver puede desearse probar estos valores de nuevo empleando los valores originales del CCR.

Preliminares

Como ya se ha visto, puede quererse llamar a una subrutina determinada en muchas condiciones diferentes, y una subrutina que cambia el con-

texto del programa principal tiene un coste elevado en flexibilidad y puede producir errores difíciles de hallar.

Muchos sucesos, provocados por el sistema o por el usuario, pueden interrumpir un programa para realizar otras tareas, de modo que el concepto general de "preservar el contexto" es fundamental en todas las operaciones de un ordenador.

Por contexto entendemos aquí la lista de todos los registros e indicadores del procesador (incluyendo el PC) que necesitamos para salvar en algún lugar, de modo que cuando llegue el momento de regresar al programa principal (tras una interrupción o una llamada a una subrutina) pueda reiniciarse la tarea que se ejecutaba de forma inequívoca y correcta. Una gran parte de los errores de programación [1] se puede atribuir a un manejo pobre de los contextos del programa. El M68000 tiene varias instrucciones, MOVEM es un buen ejemplo, orientadas a simplificar este problema. Dependiendo de la situación, "salvar el contexto" puede ser bien responsabilidad del programador, bien una tarea asignada al sistema operativo. Más adelante veremos cómo el M68000 mantiene una pila independiente accesible sólo en modo supervisor mediante el punto de pila del modo supervisor (SSP). El modo supervisor es un modo de trabajo privilegiado definido para proteger ciertos contextos vitales para el sistema, empleando la pila en modo supervisor.

Salvando el CCR

Cuando se emplean subrutinas definidas por el usuario, se puede emplear la pila para salvar los indicadores del CCR del mismo modo que haríamos para preservar los valores de los registros. La forma de hacer esto difiere, sin embargo, entre el MC68000 y los M68010/M68020 (por razones que se discutirán en el capítulo 7). En el MC68000 se emplea:

```
MOVE.W  SR,-(SP)      Salvar el SR en la pila
```

De modo que, aunque sólo necesitamos salvar el CCR, nos vemos obligados a salvar ambos bytes del registro de estado. Los procesadores M68010/68020 permiten emplear otro método más simple:

```
MOVE.W  CCR,-(SP)     Salvar el CCR en la pila
```

que sólo salva el byte del CCR en la pila (el otro byte se pone a cero). En cualquier caso, para restaurar el CCR se emplea:

```
MOVE.W  (SP)+,CCR
```

[1] Hemos optado por traducir *bug* (bicho) por error, pues éste es su significado correcto. También hemos empleado el término depurar para *debugging*, a pesar de que ambos términos se han incorporado a la jerga de uso común en informática.

En todos los ejemplos de este capitulo se empleará la versión del M68000 (MOVEr desde el SR).

RTR: Retornar y restaurar el CCR

Una forma más simple de restaurar los valores del CCR es emplear una versión especial de la instrucción RTS, denominada RTR (retornar de la subrutina y restaurar los valores de los indicadores). La instrucción RTR al final de una subrutina restaurará los valores del CCR y luego devolverá el control al programa principal. Usar RTR cuando no se han salvado previamente los valores del CCR es un grave error, la pila no estará "sincronizada" y por todas partes surgirán errores.

Anidando subrutinas

Una vez comprendido el carácter LIFO de la pila de usuario, se ve inmediatamente que las subrutinas pueden llamar a otras subrutinas y así sucesivamente; este concepto se conoce como anidar. Simplemente confiamos en que la pila nos devuelva en último lugar lo que entró primero. El máximo número de subrutinas anidadas que pueden tenerse depende únicamente del sistema operativo y de la memoria disponible. Muchos sistemas no asignan un área fija a la pila, de modo que ésta puede crecer (hacia abajo) hasta que "choque" con una zona de memoria en uso.

Vamos ahora a introducir otros modos de direccionamiento. Excluyendo los modos de direccionamiento especiales del MC68020, tenemos cuatro modos que discutir en este capitulo. Todos ellos son variantes del direccionamiento indirecto.

Direccionamiento indirecto por registros con desplazamiento

Este modo se describe d16(An), donde d16 (un número de 16 bits) representa el desplazamiento en bytes que se suma al registro An antes de tomar el operando de la memoria. A diferencia de los modos –(An) y (An)+, el valor de An no cambia al emplear este modo de direccionamiento.

El desplazamiento puede ser cualquier número con signo de –32.768 hasta +32.767 y se empleará la notación d16 para recordar este hecho. Por tanto, este modo de direccionamiento permite acceder a la memoria en un intervalo de 32 Kbytes por arriba o por debajo de la dirección contenida en el registro An. Se emplea principalmente para trabajar con datos en una tabla cuya dirección base (es decir, la dirección de comienzo de la tabla) está contenida en An. Es útil expresar el modo d16(An) en términos de cálculo de una dirección efectiva, <ea>, es decir: <ea> = d16 + An. Esta

fórmula expresa cómo el procesador determina la dirección del operando. El desplazamiento d16 se almacena en realidad como una palabra de extensión, como ya se vio en el caso de datos inmediatos. El tiempo de cálculo de una dirección efectiva puede variar desde 0 ciclos, en el modo directo, hasta 17 ciclos,en modos complicados de direccionamiento indirecto. Veamos como funciona el modo d16(An).

Modo con desplazamiento: Aplicaciones

En la figura 5.4, A2 = $6000 apunta a una tabla de datos que consiste en 20 dobles palabras ($6000), ($6004), ($6008), ($600C), etc. Para poner D3 al valor de la cuarta entrada necesitamos efectuar un:

```
MOVE.L  12(A2),D3
```

puesto que la dirección efectiva de la fuente es:

$$A2 + (3 \text{ dobles palabras}) = A2 + 12 \text{ bytes} = \$6000 + \$C = \$600C$$

tras el MOVE A2 = $6000. Si deseamos cambiar el orden de las segunda y tercera entradas de la tabla, podríamos emplear el método:

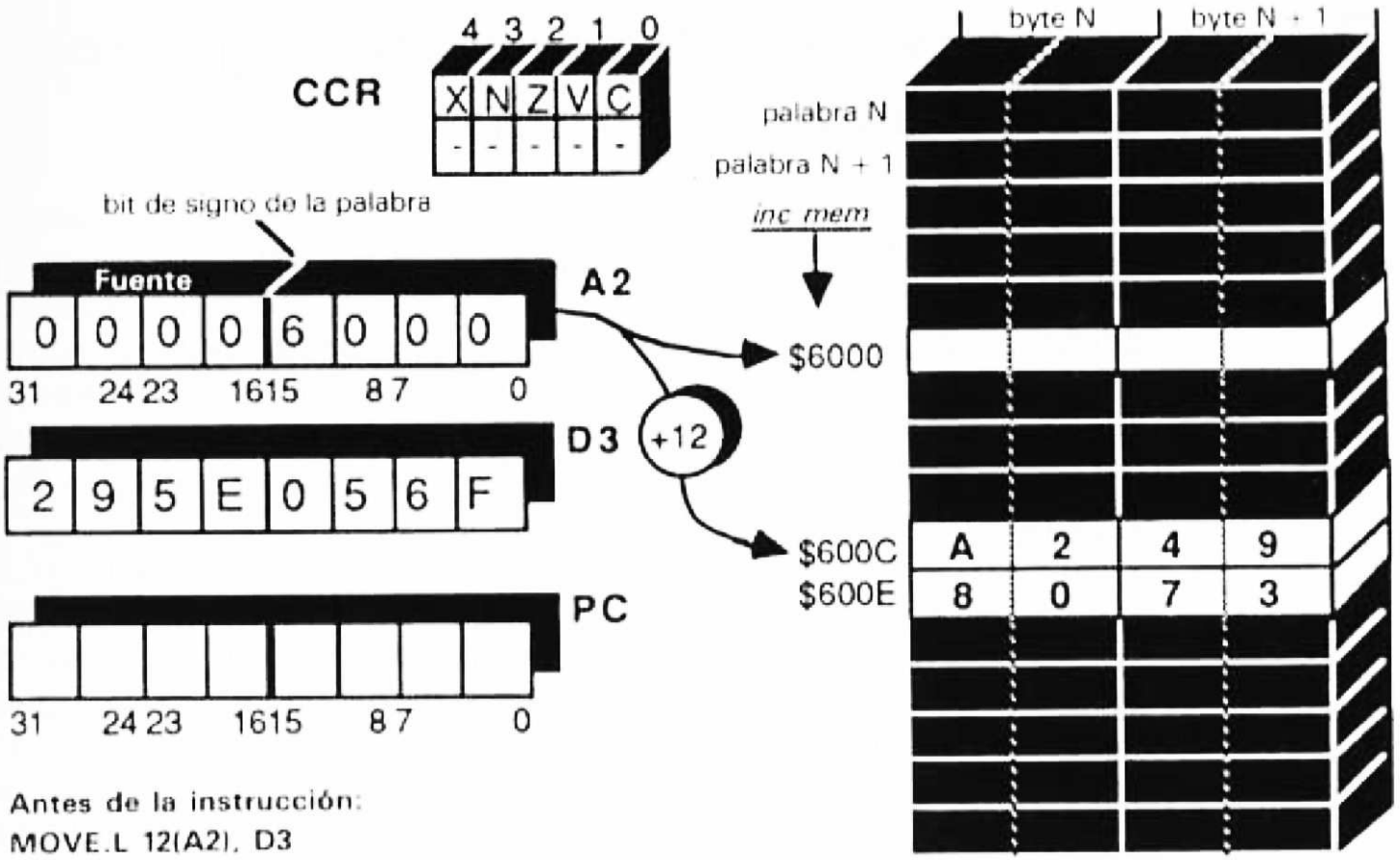
```
* Programa 5.4: Cambiar el orden en una tabla
* Ilustraremos el uso de d16(An) tanto en origen como en destino
* A2 contiene la base de la tabla
* Invertimos el orden de la segunda y tercera entradas

MOVE.L  4(A2),D0        Tomamos la segunda entrada
MOVE.L  8(A2),4(A2)     Movemos la tercera entrada al lugar de la segunda
MOVE.L  D0,8(A2)        Ponemos la segunda entrada en tercer lugar
```

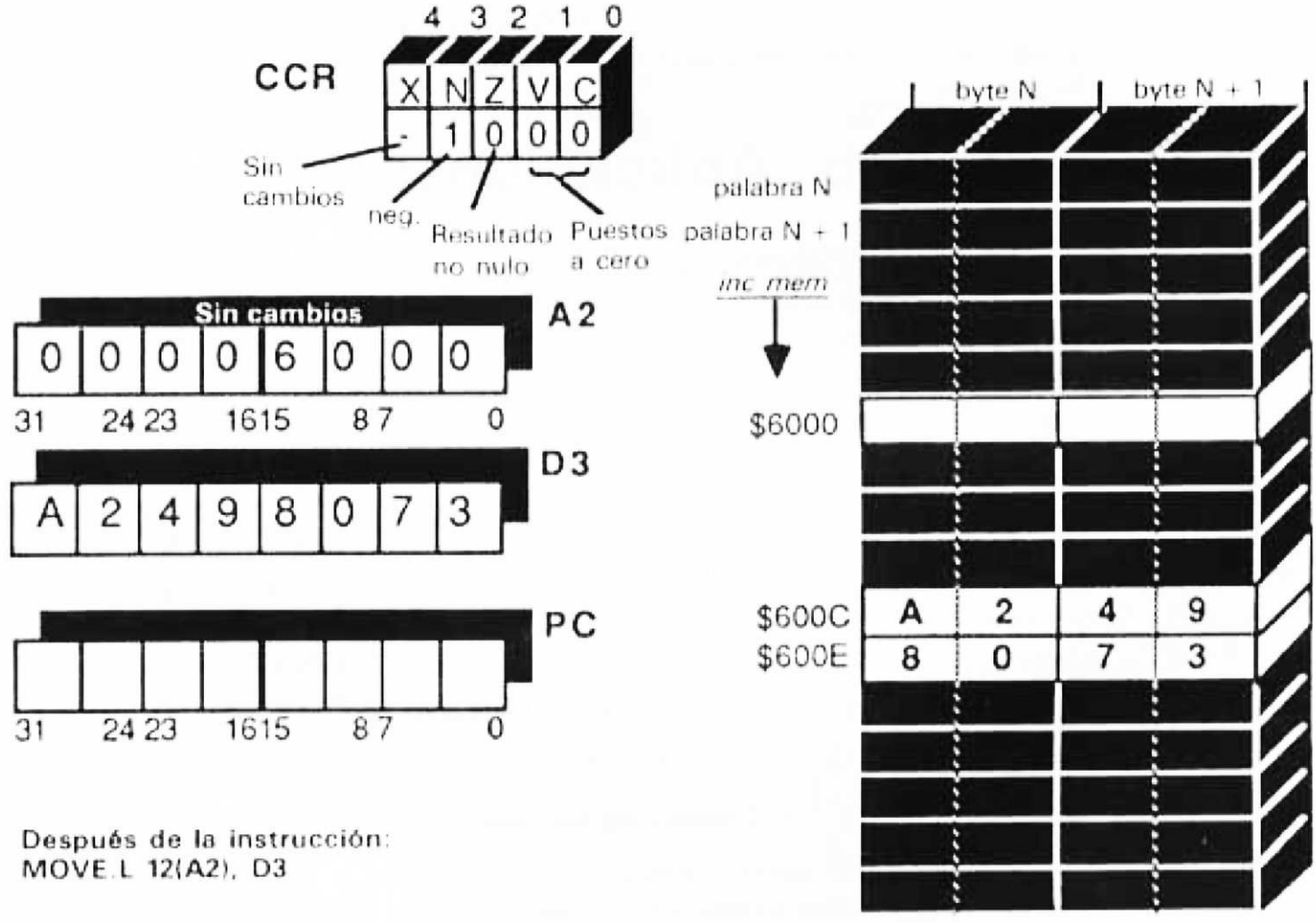
Los tres puntos a tener en cuenta cuando se emplea este modo son:

1. Para operandos tipo L o W, la suma (d16 + An) debe ser un número par. Para operandos tipo B, esta suma puede ser par o impar.
2. No deben confundirse –(An) con –1(An). Por ejemplo, si A2 = = $2001, tanto MOVE.B –1(A2),D3 como MOVE.B –(A2) mueven el byte de menor orden de D3, pero –1(A2) deja A2 = $2001. El modo predecrementado (A2) reduce A2 a $2000.
3. Este modo aparece normalmente como BLA(An); por ejemplo, donde BLA es un desplazamiento (es preferible que BLA tenga carácter mnemotécnico) al que se le asigna un valor durante el ensamblado.

Hay muchos casos en los que se necesita la flexibilidad de un desplazamiento variable en lugar del desplazamiento fijo de d16(An). El próximo modo de direccionamiento soluciona este problema.



Antes de la instrucción:
MOVE.L 12(A2), D3



Después de la instrucción:
MOVE.L 12(A2), D3

Figura 5.4
MOVE.L 12(A2),D3

Direccionamiento indirecto por registros con desplazamiento e índice

La extensión del modo d16(An) permite un desplazamiento variable extra, conocido como indice, para sumarlo desde un registro. Llamaremos modo indexado a este tipo de direccionamiento por brevedad. Se escribe d8(An,Xi.Z), donde d8 representa el desplazamiento en bytes que se efectuará (d8 es un número de 8 bits con signo desde −128 hasta +127), An contiene la dirección base como en el caso anterior y Xi es cualquier registro (D0-D7, A0-A7), conocido aquí como registro índice, y Z es un código de tamaño: L o W.

La dirección efectiva, <ea>, construye con tres elementos diferentes: <ea>= An + d8 + Xi.Z. En otras palabras, el procesador toma el número d8 y suma el número de bytes indicado por éste con la dirección An, y después toma del registro Xi.Z el número de bytes que debe emplear para efectuar el desplazamiento final relativo a An. El registro Xi.L aportará bien los 32 bits en el caso de Xi.L, bien los 16 menos significativos en el caso de Xi.W. La velocidad de cálculo de la dirección efectiva <ea> es la misma para los casos L y W.

A menudo abreviaremos el modo índice a d(An,Xi) si el desplazamiento fijo d8 es 0, este modo propiciará un modo indexado sencillo, como se encuentra en muchos microprocesadores de 8 y 16 bits.

Modo indexado: Aplicaciones

La aplicación principal del modo d(An,Xi) se encuentra al acceder a matrices multidimensionales de datos en la memoria. Con dos desplazamientos, por ejemplo, se puede asignar a An la dirección base de una hoja de cálculo. Si se asigna al desplazamiento d el valor del número de línea y al registro índice Xi el número de la columna, entonces d(An,Xi) es la dirección efectiva de la celda en la memoria, es decir, el contenido de la celda que se encuentra en la intersección de la fila y la columna indicada.

Aunque el desplazamiento fijo d8 ofrece un intervalo que va desde −128 hasta +127 solamente, el registro índice Xi.L es una amplia compensación, pues puede cubrir un rango de 2 Gigabytes a cada lado del valor del puntero An.

Veamos el uso del modo indexado empleando los datos del programa 5.4, con d8 = 0 para mostrar un direccionamiento indexado simple.

```
* Programa 5.5: Cambiar las posiciones de los registros en una tabla
* empleando modo indexado
* Datos iniciales como en el programa 5.4
* Ilustrar el uso del modo d8(An,Xi.Z) tanto en origen como en destino
* A2 apunta a la base de una tabla de dobles palabras
* Invertir el orden de las segunda y tercera entradas
```

```
MOVEQ.L   #4,D1              Se pone el registro índice D1 a 4
MOVE.L    0(A2,D1.W),D0      Se salva la segunda entrada de la tabla
MOVEA.W   #8,A0              Se pone el registro índice A0 a 8
MOVE.L    0(A2,A0.W),0(A2,D1.W)  Se mueve la tercera entrada al segundo
                                 lugar
MOVE.L    D0,0(A2,A0.W)      Se mueve la segunda entrada al tercer
                                 lugar
```

* Nótese que A2 permanece sin cambios

En el programa 5.5 hemos empleado tanto los registros D1 como A0 para demostrar que se puede emplear cualquier tipo de registro como índice. Los registros de datos son más apropiados que los de direcciones, principalmente, porque se pueden realizar más operaciones aritméticas sobre ellos. Cuando se está trabajando con una tabla, a menudo, es mejor operar (sumar, restar, multiplicar e incluso dividir) sobre el desplazamiento en Xi.

En el programa 5.5 la tabla está constituida por dobles palabras, de modo que hay que ajustar los desplazamientos en múltiplos de 4. Para tablas de palabras, los desplazamientos deben ser de 2. Naturalmente se puede pedir al M68000 que efectúe estos cálculos por nosotros. Para obtener múltiplos, ¡multipliquemos!

Multiplicación

Hay dos instrucciones básicas de multiplicación que emplean el mismo formato:

```
MULS  <fuente>,Dn    Multiplicación con signo
MULU  <fuente>,Dn    Multiplicación sin signo
```

Ambas instrucciones multiplican dos palabras de 16 bits, proporcionando un resultado de 32 bits. No se necesita un código de tamaño, pues implícitamente se asigna el código W. La fuente puede venir dada por cualquier modo de direccionamiento, excepto por An (direccionamiento directo por registro). La multiplicación se efectúa de acuerdo con la regla:

(palabra en la fuente) × (palabra de orden más bajo en Dn) =
= (resultado de 32 bits en Dn)

Como en el caso de ADD o SUB, la fuente permanece inalterada, pero el resultado destruye el valor previo del operando destino. Sin embargo, a diferencia de ADD y SUB, la multiplicación con signo (MULS) o de multiplicación sin signo (MULU). Cuando se emplean ADD o SUB, los indicadores

V (rebose) y C (acarreo) advertían de posibles errores en el signo del resultado. La multiplicación es diferente.

Las inviolables leyes de la aritmética binaria indican que, al multiplicar dos números sin signo de 16 bits, la respuesta cabe siempre en 32 bits sin acarreo ni rebose. Multiplicar un número con signo por otro sin signo no es un ejercicio con resultados brillantes, pues el M68000 asume que se conocen bien los tipos de los números que se pretenden multiplicar. Se debe elegir entonces entre MULU y MULS para obtener una respuesta correcta. Ambas instrucciones afectan a los indicadores del CCR:

Indicador	X	N	Z	V	C
MULS/MULU	–	*	*	0	0

(Recordemos lo que nuestra notación indica: X no cambia; N se pone al valor del bit de signo —bit 31—; Z se pone a 1 si el resultado es cero; C y V siempre se ponen a 0.) MULS, desde luego, maneja correctamente los signos de los resultados, por ejemplo: $(-6) \times (-2) = +12$ y $(-6) \times (+2) = -12$, de modo que el indicador N indica literalmente positivo o negativo. MULU se emplea para multiplicar sin considerar el signo, entonces N sólo indica qué valor tiene el bit más significativo (el bit en la posición 31) de Dn tras la multiplicación.

Ilustraremos MULS con un sencillo (pero esencial) ejemplo tomado de una nómina:

```
* Programa 5.6: Horas × Precio de la hora = Paga
* Como en el programa 4.1, la palabra D2 contiene el número de horas
* que se trabajó en el mes de marzo = 138
* La palabra en D4 contiene el precio por hora = 699
* Calcular el salario de marzo; si es positivo, salvarlo en la palabra larga
* en la dirección absoluta $A200. Preservar los valores de D2 y D4

        MOVE.W  D4,-(SP)    Salvar la palabra D4 en la pila
        MULS    D2,D4       D4 es ahora D4 = D2 × D4 = Horas × Precio = Salario
                            (D2 no cambia, D4 cambia)
        BMI     DEUDA       Si el resulta es negativo, ir a deuda
        BEQ     NOPAGA      Si el resultado es cero, ir a NOPAGA
        MOVE.L  D4,A$2000   Salvar el salario en la memoria
        MOVE.W  (SP)+,D4    Restaurar el valor de D4
        <continuar con el programa>
        *   *   *
        BRA     FINAL       Saltar a la sección final
DEUDA   MOVE.W  (SP)+,D4    Restaurar el valor de D4 (véase nota 2)
        <comprobar la situación de salario negativo>
        *   *   *
        BRA     FINAL       Saltar a la sección final
NOPAGA  MOVE.W  (SP)+,D4    Restaurar el valor de D4 (véase nota 2)
        <comprobar la situación de salario negativo>
        *   *   *
```

```
FINAL    <sección final; concluir el programa>

* Salario de marzo = 96462 almacenado en $A200
* D2 no ha cambiado, se restaura D4
```

1. Se ha empleado MULS en lugar de MULU, porque en la mayor parte de las aplicaciones financieras aparecerán cantidades negativas (devoluciones, ajustes, etc.). MULS permite emplear BMI con sentido, puesto que BMI comprueba el indicador N.
2. Tenemos que mantener la pila ajustada. Puesto que hemos metido D4, hay que sacarlo antes o después. No se puede sacar antes del BMI o el BEQ, porque el MOVE afecta los indicadores N y Z.
3. Tenemos un uso típico de BRA (salto incondicional) para saltar por encima de secciones del programa que no nos interesan. Sin el primer BRA FINAL, el programa habría entrado en la sección de DEUDA una fuente común de errores.

División

Como en el caso de la multiplicación, existen dos instrucciones para la división:

```
DIVS  <fuente>,Dn    División con signo
DIVU  <fuente>,Dn    División sin signo
```

Ambas instrucciones dividen el operando destino, Dn (32 bits), entre el operando fuente (16 bits) para dar el resultado (16 bits) en la palabra más significativa. La fuente puede tener cualquier modo de direccionamiento, excepto An, de modo que podemos dividir un registro de datos entre otro o entre una palabra en la memoria. Como ya se ha visto en otras operaciones aritméticas, la fuente (divisor) permanece inalterada, mientras que el destino (dividendo) se reemplaza por el resultado. La división emplea, pues, la siguiente regla:

(Destino —los 32 bits de Dn—) / (Fuente —una palabra de 16 bits—) = (Resto —palabra más significativa de Dn—) (Cociente —palabra menos significativa de Dn—)

He aquí un pequeño ejemplo:

```
* Programa 5.7
* Ganancias diarias promedio calculadas empleando DIVS
* Empleando los datos del Programa 5.6, D4 contiene el salario de marzo = 96462
* Calcular las ganancias diarias medias y almacenarlas en la segunda palabra
```

```
* de una tabla que emplea como puntero de la base el registro A2. Salvar el resto
* en la sexta palabra de esta tabla. Conservar el valor de D4

        MOVE.L  D4,D7       Se salva D4 en D7
        DIVS    #31,D4      Se divide D4 entre 31 (división en modo inmediato).
                            Palabra significativa de D4 = cociente. Palabra más
                            significativa de D4 = resto
        MOVE.W  D4,2(A2)    Se salva el cociente en la tabla
        SWAP    D4          Se cambia el orden de las palabras en D4. El resto
                            es ahora la palabra menos significativa y el cociente
                            la más significativa
        MOVE.W  D4,10(A2)   Se salva el resto en la tabla
        MOVE.L  D7,D4       Se restaura D4

* Ahora 2(A2) = 3111 y 10(A2) = 21
* 96462/31 = 3111, con un resto de 21
* Almacenarlas en la segunda palabra de una tabla que emplea como puntero de la base
  el registro A2. Salvar el resto en la sexta palabra de esta tabla. Conservar el valor de D4

        MOVE.L  D4,D7       Se salva D4 en D7
        DIVS    #31,D4      Dividir D4 entre 31 (división en modo inmediato).
                            Palabra significativa de D4 = cociente. Palabra más
                            significativa de D4 = resto
        MOVE.W  D4,2(A2)    Se salva el cociente en la tabla
        SWAP    D4          Se cambia el orden de las palabras en D4. El resto
                            es ahora la palabra menos significativa y el cociente
                            la más significativa
        MOVE.W  D4,10(A2)   Se salva el resto en la tabla
        MOVE.L  D7,D4       Restauramos D4

* Ahora 2(A2) = 3111 y 10(A2) = 21
* 96462/31 = 311, con un resto de 21
```

SWAP Dn es una instrucción simple, pero poderosa, que invierte el orden de las palabras más y menos significativas en los registros de datos. Nótese que en el programa 5-7 no se puede emplear directamente la instrucción MOVE para salvar el resto, puesto que esta instrucción sólo mueve la palabra menos significativa, y MOVE.L mueve toda la palabra. Más adelante estudiaremos manipulaciones más extrañas en los registros, incluyendo desplazamientos y rotaciones; pero ahora ya hemos visto la necesidad de estas operaciones: aislar una parte de un registro para poder acceder a ella.

Se ha empleado DIVS por la misma razón que se empleó MULS en el programa 5-6. DIVS proporciona la respuesta correcta; así, por ejemplo,

$$(-24)/(-2) = (+12) \text{ y } (+24)/(-2) = (-12), \text{ etc.}$$

Los indicadores N y Z del CCR reflejan el estado del cociente, porque el resto tiene el mismo signo que el dividendo (a menos que el resto sea cero). Por ejemplo:

$(+25)/(-2) = (-12)$ resto $= +1$
$(-25)/(-2) = (+12)$ resto $= -1$
$(-25)/(+25) = (-1)$ resto $= +0$

A diferencia de MULS, DIVS y DIVU pueden presentar dos problemas. El primero de ellos lo constituyen las "divisiones por cero". Si el programa no comprueba esta posibilidad, el M68000 disparará un mecanismo (TRAP) para evitar que el sistema caiga en un ciclo infinito. Este tipo de mecanismos pertenece a una clase de excepciones, que bien se encuentran bajo el control del programador o bien son competencia del sistema, que llevan al M68000 al modo supervisor (que se ha mencionado en el capítulo 3). En este modo privilegiado, el sistema inicia la acción de recuperación apropiada.

Brevemente, el mecanismo que se oculta tras un TRAP, como el que se dispara en el caso de una división por cero, lleva al procesador a una tabla en la memoria reservada al sistema ($000-$3FFF), denominada tabla de vectores de excepción, donde se encuentra la dirección de la rutina capaz de controlar la situación. El M68000 es, pues, tremendamente flexible a la hora de controlar una situación que en otros *chips* menos elaborados llevaría al caos, la destrucción o a cosas peores.

El segundo problema que puede aparecer es que DIVU y DIVS pueden llevar a situaciones de rebosamiento cuando el dividendo sea muy grande en relación con el divisor, de modo que se exceda la capacidad de 16 bits (de −32767 a 32768 en el caso de aritmética con signo y 65536 en el caso de aritmética con signo). Como medida de protección, el procesador pondrá el indicador de rebose, V, a 1 en el caso de que se produzca esta situación.

Si los números que se están empleando pueden llevar a esta situación, se debe comprobar el estado del indicador V inmediatamente después de la división con un BVS (bifurcar si se produce rebosamiento) inmediatamente después de la división, tal como anteriormente se hizo con ADD.

DIVS y DIVU afectan al CCR según la siguiente tabla:

Indicador	X	N	Z	V	C
DIVS/DIVU	—	*	*	*	0

que coincide con la de MULS/MULU, excepto en el caso del indicador V.

En el programa 5-7 empleamos para el divisor el número de días de marzo, el modo inmediato: #31. ¡Un programa más práctico permitirá emplear otros meses! Por ejemplo, podríamos construir una pequeña tabla con los días que tiene cada mes. Una tabla así puede ser útil en muchas aplicaciones financieras: para calcular los días transcurridos entre dos fechas a efectos de amortización de intereses y cosas parecidas. El modo indexado es adecuado a estos propósitos. En el próximo ejemplo veremos cómo se pueden efectuar cálculos aritméticos en el registro índice para simplificar la localización de los datos en una tabla.

```
* Programa 5.8
* Ganancias diarias promedio en cualquier mes
* El número del mes M (Enero = 1, Febrero = 2, etc.) se encuentra en la palabra menos signi-
* ficativa de D0. D4 contiene la paga de ese mes. A0 apunta a la base de la tabla de 12 pa-
* labras
* DIAMES (A0) = 31, 2(A0) = 28, 4(A0) = 31,... 24(A0) = 31
* Por tanto, el número de días del mes M se encuentra en la palabra
* {2 × <M—1>}(A0). Ignoraremos a los años bisiestos por el momento
* Calcular el promedio diario de ganancias con una cifra decimal y almacenar
* el resultado en la palabra menos significativa de D6
* Conservar los valores de D4 y D6

MOVEM.L  D0/D4,-(SP)      Los guardamos en la pila
SUBQ.W   #1,D0            D0 = M-1
MULU     #2,D0            D0 = 2 × <M-1>
MULS     100,D4           Multiplicar las ganancias por 100
DIVS     0(A0,D0.W),D4    Dividimos D4 entre los días del mes M
MOVE.M   D4,D6            Se ignora la palabra más significativa de D4 (el
                          resto)
                          La palabra menos significativa contiene el prome-
                          dio por 100
MOVEM.L  (SP)+,D0/D4      Restaurar los valores de D0 y D4

* Para el conocido mes de marzo, M = 3, de modo que D0 = 2 × {3—1} = 2
* El operando fuente para DIVS es la palabra en la dirección 0 + A0 + 4 = 4(A0),
* es decir, la tercera entrada en la tabla DIAMES, 31. D6 contiene ahora el promedio
* por 100. Podemos redondearlo a un decimal más tarde
```

Un método alternativo para la multiplicación es emplear desplazamientos, tal como se discute a continuación.

ASL: Desplazamiento aritmético a la izquierda

Hasta ahora hemos empleado MULU porque estamos tratando con números positivos conocidos y pequeños y porque es ligeramente más rápida que MULS. De hecho, existe una forma mucho más rápida de multiplicar por 2, 4, 8 o cualquier potencia pequeña de 2.

Se puede usar:

```
ASL.z  #<d3>,Dn
```

que realiza un desplazamiento aritmético a la izquierda. El número de lugares a los que afecta el desplazamiento viene dado por el número de 3 bits <d3>, lo que nos proporciona un contador de desplazamiento con un valor comprendido entre 1 y 8. El código de tamaño determina qué porción de Dn se ve afectada por el desplazamiento, B, W, o L. La figura 5.5 muestra cómo puede emplearse la instrucción ASL para multiplicar por 2. Cada

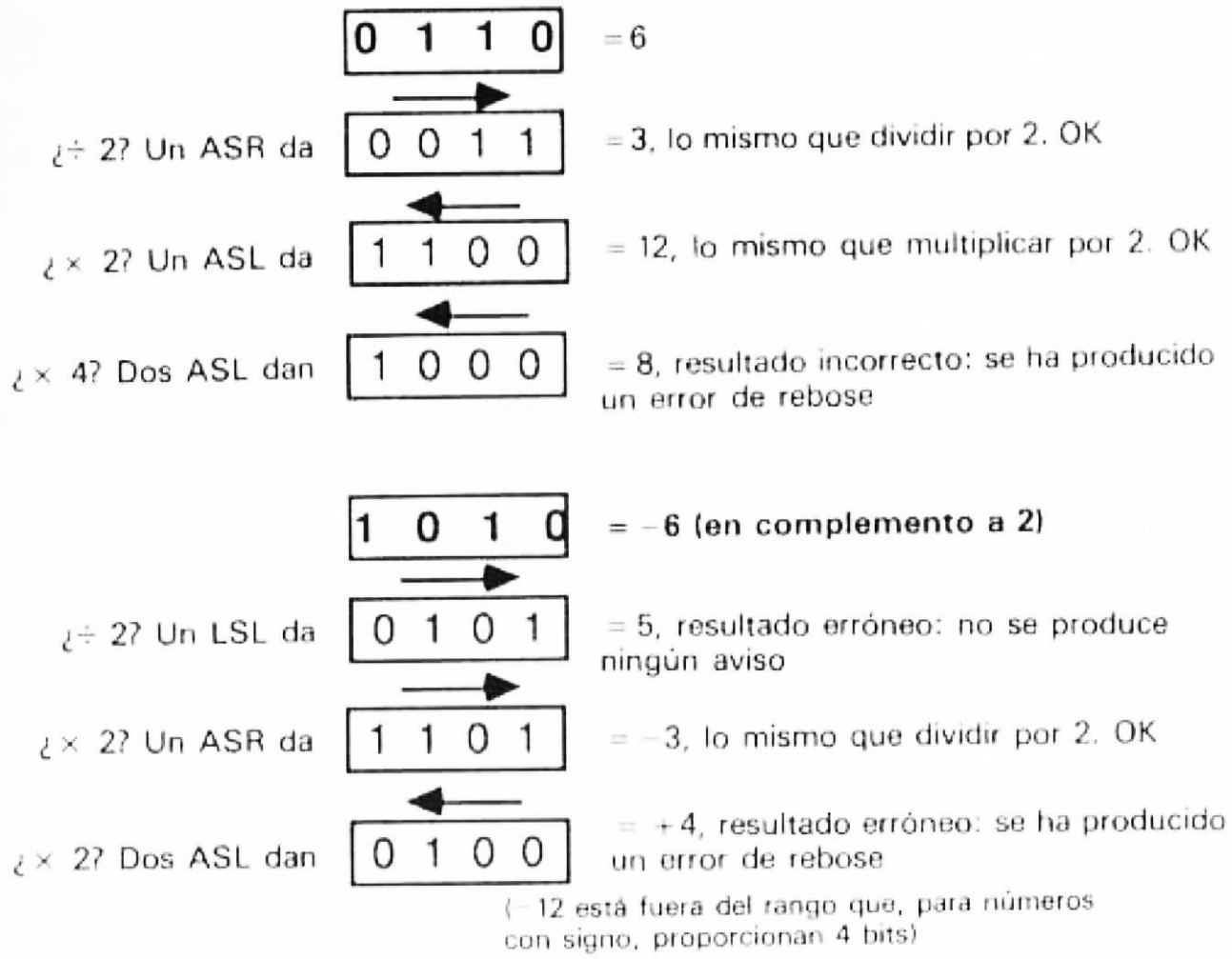


Figura 5.5
Uso de los desplazamientos para multiplicar y dividir

desplazamiento aritmético de la secuencia de bits en la porción L, W, o B de Dn es equivalente a duplicar el valor de dicha parte de Dn. De este modo, un ASL de 1 puede duplicar sólo el byte menos significativo de Dn sin afectar al resto del registro. A medida que el desplazamiento aritmético tiene lugar, se van metiendo ceros en Dn desde la derecha y los bits van saliendo por el otro extremo.

En el programa 5.8 podemos reemplazar:

```
MULU   #2,D0    Palabra larga en D0 = {palabra en D0 × 2}
```

por cualquiera de los siguientes:

```
ASL.L  #1,D0    Palabra larga en D0 = {palabra en D0 × 2}
ASL.W  #1,D0    Palabra en D0 = {palabra en D0 × 2}
ASL.B  #1,D0    Byte en D0 = {byte en D0 × 2}
```

Tanto MULU como cualquiera de los tres ASL funcionan igualmente bien en nuestro ejemplo, puesto que el valor máximo de D0 es 22, dentro de los límites de la capacidad máxima del byte menos significativo de D0 para aritmética con signo. Para números mayores, desde luego, habrá que de-

cidir cuál de las instrucciones (ASL/MULS/MULU) es la que proporciona una codificación más segura. A diferencia de MULS/MULU, ASL puede producir reboses y hay que comprobar el CCR para verificar que no se hayan excedido las diferentes capacidades de cálculo (L, W, o B).

ASL es de tres a cinco veces más rápida que MULU o MULS.

Para realizar un desplazamiento de más de 8 lugares, o para realizar un desplazamiento variable, se puede emplear el formato:

ASL.z	Dm,Dn	Desplazar Dn a la izquierda las veces que indique Dm (máximo, 64), donde los 6 bits menos significativos de Dm contiene el contador de desplazamiento

También se puede someter a desplazamiento un byte en la memoria, pero el contador de desplazamiento está restringido a 1:

ASL.z	<operando en memoria>	Desplazamiento simple a la izquierda de un operando en memoria

En el capítulo 6 se tratarán con más detalle las instrucciones de desplazamiento y rotación, de modo que sólo mencionaremos ahora una variante del ASL, el ASR.

ASR: Desplazamiento aritmético a la derecha

Como se muestra en la figura 5.5, desplazar un registro a la derecha es lo mismo que dividir por 2. Para conservar el bit de signo del registro sometido a desplazamiento, ASR introduce los bits a partir de éste hacia la derecha.

Los formatos para el ASR son los mismos que los del ASL:

ASR.z	#<d3>,Dn	Desplazar Dn a la derecha d3 veces
ASR.z	Dm,Dn	Desplazar Dn a la derecha las veces que indique Dm (máximo, 64)
ASR.z	<operando en memoria>	Desplazamiento simple a la derecha de un operando en memoria

Resumen del modo índice

El modo índice permite establecer punteros para manejar estructuras de datos en la memoria. Se pueden establecer desplazamientos positivos o negativos respecto a la dirección de la base mantenida en un registro de direcciones, calculándolos de diferentes formas, mediante operaciones aritméticas en cualquier registro designado como índice, Xi, en el modo d(An,Xi).

Terminamos ahora nuestro examen de los modos básicos de direccionamiento del M68000 estudiando los modos relativos.

Modos relativos: Motivaciones

En los ejemplos previos hemos hecho uso de datos almacenados en posiciones fijas de la memoria. Por ejemplo, "las horas YTD están almacenadas en la dirección $6000 o HRSTYD". En las aplicaciones reales rara vez se saben por adelantado las direcciones absolutas de los datos o rutinas, y a menudo tendremos considerables problemas para asegurarnos que los programas funcionarán al cargarlos en cualquier zona de la memoria. Tales programas se denominan programas relocalizables o programas independientes de la posición de memoria. De hecho, algunos sistemas operativos (como el caso del AMOS de Alpha Micro, empleado en el ordenador AM100L basado en el MC68000) requieren que todos los programas sean relocalizables. Otros sistemas operativos presentan restricciones o exigen ciertos programas especiales para cargar *software* no relocalizable).

En un programa relocalizable, la mayoria de las referencias a la memoria debe hacerse con relación a ciertas direcciones que no se conocen por anticipado. Excepciones obvias a esta regla son las localizaciones fijas, como las que se emplean para la entrada/salida, o las áreas reservadas por el sistema, como las tablas de vectores de excepción.

Cuando un programa se carga y se ejecuta, las instrucciones deben contener suficiente información como para permitir que el procesador calcule la dirección efectiva de cada operando, de modo que pueda localizarlo y trabajar con él correctamente.

El papel del contador de programa

La clave de todo lo expuesto anteriormente es el contador del programa, PC, que, como ya hemos visto, es un registro de direcciones muy especial que contiene la dirección absoluta de la instrucción que en ese momento está ejecutando el procesador. Un programa puede seguirse hasta que termina, mirando el PC de la misma forma que seguiríamos las flechas en un sitio público para llegar a cierto destino.

De modo que tenemos una forma de referirnos a los operandos en la memoria de una forma relativa, que no absoluta, indicando, por ejemplo, "420 bytes después (o antes) de la actual posición contenida en el PC se encuentran las horas YTD de marzo". Con este tipo de codificación sí es posible construir un programa relocalizable.

Direccionamiento relativo: Direccionamiento por contador de programa con desplazamiento

Este modo de direccionamiento se escribe simbólicamente como d16(PC): —lo que nos recuerda mucho al modo d16(An), con el PC en el lugar del

An—. Parece obvio, pues, que d16 indica un número de 16 bits con signo que se empleará como desplazamiento. Este número de 16 bits nos ofrece un rango de 32 Kbytes por encima o por debajo de la dirección contenida en el PC. La dirección efectiva se calcula según la expresión: <ea> = d16 + PC. El desplazamiento se almacena como una única palabra de extensión siguiendo a la instrucción, y, estrictamente hablando, esta palabra de extensión es la que representa el valor del PC cuando se calcula la dirección efectiva <ea>. Esto es francamente extraño, pues el procesador tiene que tomar la palabra d16 de la memoria, de modo que el PC ha pasado ya la dirección de la propia instrucción.

Aunque hasta aquí el modo d16(PC) parece igual al modo d16(An), hay una excepción fundamental:

Los modos de direccionamiento relativos sólo pueden emplearse para operandos fuente

Cualquier combinación de código/operando en la que el destino es una dirección alterable en la forma d16(PC) es ilícita. La razón es simple: es una forma de poner trabas a una alteración inadvertida del programa. Ampliaremos esto más adelante.

Los operandos en modo relativo no pueden alterarse

El modo d16(An) puede emplearse como fuente o destino, pero sólo puede emplearse el modo d16(PC) como fuente. Ahora puede parecer que el modo d16(PC) es inútil en la práctica: ¿cómo demonios se asigna un valor a d16? ¿Cuántos bytes delante o después de la instrucción actual está la dirección efectiva, <ea>, del operando que quiero emplear? La solución para esta pregunta se encuentra en las etiquetas. Empleando operandos etiquetados, podemos delegar en el ensamblador para que sea él quien realice los cálculos de los desplazamientos pertinentes. Este truco es similar al que empleamos con la combinación Bcc <etiqueta> en el capítulo 4.

Etiquetas como operandos relativos

Se ha visto en el capítulo 4 que se puede emplear la directiva ORG para forzar al ensamblador a asignar una dirección absoluta a las etiquetas.

Existe otra directiva del ensamblador, denominada RORG (origen relativo). En una sección del código fuente presidida por una RORG, el ensamblador asignará de forma automática a las etiquetas de los operandos el modo d(16)PC.

En esta fase, el ensamblador no sabe qué valor contendrá el PC cuando se decodifiquen las instrucciones durante una ejecución del programa, pero no le hace falta. Todo lo que el ensamblador necesita saber es el desplazamiento relativo, la "distancia" en bytes entre cualquier instrucción que emplee una etiqueta y la etiqueta misma.

La mayoría de los ensambladores son multipaso, es decir, recorren el código fuente varias veces para permitir que se establezcan las posiciones relativas de todas las etiquetas, y así calcular y asignar los correspondientes desplazamientos d16.

Veamos una situación típica, para lo cual introduciremos la instrucción JMP, que es simplemente una versión más versátil de la bifurcación incondicional BRA. También existe el equivalente de BRS, JRS. Existe, de todos modos, una diferencia técnica entre BRA y BSR y sus equivalentes JMP y JSR, dado que estas últimas admiten una clase más amplia de direcciones que las primeras, como veremos al final de esta sección.

```
        RORG            La siguiente sección es relativa
        *     *     *   <parte del programa viene aquí>
        JUMP  BUCLE     Saltar a BUCLE
        *     *     *   <aquí hay instrucciones que en el total ocupan 600 bytes>
BUCLE   <aquí continúa el programa>
        *     *     *
```

Las dos palabras correspondientes a la instrucción JMP, una vez ensambladas, tendrán el siguiente aspecto:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
											m	m	m	r	r	r	
JMP	0	1	0	0	1	1	1	0	1	1	1	1	1	0	1	0	
d16	0	0	0	0	0	0	1	0	0	1	0	1	0	1	1	0	(d16 = +598)

Los bits del 6 al 15 codifican la instrucción JMP en sí, mientras que los bits del 0 al 5 indican el modo de direccionamiento d16(PC). La palabra de extensión contiene el desplazamiento relativo de la etiqueta BUCLE. El desplazamiento es positivo, porque el salto ha sido hacia adelante.

Al decodificar esta instrucción durante la ejecución, el procesador, siguiendo las rígidas reglas del modo d16(PC), toma la palabra de extensión y calcula la dirección defectiva del operando según la regla: <ea> = PC + + 598 bytes. Puesto que el PC contiene la dirección de la palabra de extensión, tenemos que: <ea> = dirección del JMP + 600 bytes, que es la dirección de la línea etiquetada <BUCLE>. La instrucción JMP ahora asigna al PC el valor de la dirección <ea> calculada. Así, el procesador toma la próxima instrucción de la zona de memoria marcada como BUCLE. En otras palabras, hemos saltado a BUCLE.

El ensamblador calcula automáticamente el desplazamiento, en bytes, d16 = "<etiqueta>−PC" bytes, mientras que el procesador, para obtener la dirección real de la <etiqueta>, calcula: <ea> = PC + desplazamiento d16, donde ahora PC representa una dirección real y conocida.

Las etiquetas, en una sección presidida por un ORG, son iguales a aquellas empleadas en modo absoluto. Las etiquetas en una sección PC

equivalen a emplear el modo de direccionamiento relativo por contador de programa con desplazamiento.

Raramente se empleará el modo d16(PC) directamente; lo normal será emplear una directiva RORG en una determinada sección del programa, sabiendo que es una forma encubierta de emplear el modo d16(PC).

Además de emplear las etiquetas como marcas para los saltos y bifurcaciones, se pueden emplear las etiquetas de muchas formas. RORG permite emplear las directivas DC (definir constantes) y DS (definir un almacenamiento) con etiquetas relativas al PC. Así, las áreas de datos son relocalizables, porque se ha empleado el modo relativo al PC. Por ejemplo:

```
        RORG

TABLA   DC.W    $34A2,$00B7
```

define dos palabras de datos en la dirección TABLA que puede emplearse como operando fuente, sin importar dónde se encuentre el programa en la memoria. De este modo:

```
        MOVE.W  TABLA,D1
```

llevará $34A2 a la palabra menos significativa de D1. Para conseguir esto, el ensamblador define un desplazamiento de 16 bits en una palabra de extensión asociada a la instrucción MOVE y el operando fuente se codifica en el modo d16(Pc). Cuando la instrucción se decodifica y se ejecuta el desplazamiento sumado al PC, proporciona la dirección efectiva de los datos almacenados en la TABLA.

Vamos a estudiar ahora otro método de acceder a la memoria que emplean las importantes instrucciones LEA (cargar la dirección efectiva) y PEA (meter en la pila la dirección efectiva).

LEA es una instrucción de dos operandos con el formato:

```
LEA  <fuente>,An
```

LEA calcula la dirección efectiva del operando fuente y la almacena en An. Todos los bits de An resultan afectados, aun cuando el *chip* emplee un menor número de bits de dirección.

PEA es una instrucción con un solo operando con el formato:

```
PEA  <fuente>
```

PEA efectúa los mismos cálculos que LEA y almacena la dirección en la pila que en ese momento esté empleando el sistema (pila de modo usuario o de modo supervisor). PEA es equivalente, de hecho, a:

```
LEA     <fuente>,An
MOVE.L  An,-(SP)
```

(aunque cuando se emplea PEA no se involucra a ningún registro).

Podemos ilustrar el uso de LEA con áreas de datos etiquetados:

```
* Programa 5.9
* Uso de la instrucción LEA
* Calcular la media de dos números almacenados en la TABLA

          RORG
TABLA     DC.W     $34A2,$00B6    Definimos un área de datos en la dirección TABLA
          CLR.L    D1             Poner a cero el registro empleado como acumulador
          LEA      TABLA,A3       Cargar la dirección efectiva de la TABLA en A3
          MOVE.W   (A3)+,D1       D1 contiene ahora la primera de las dos palabras
                                  a sumar
          ADD.W    (A3),D1        Sumar la segunda palabra
          ASR.W    #1,D1          Dividir la suma por 2

* La respuesta final se encuentra en la palabra menos significativa de D1
* <$34A2 + $00B6>/2 = $1AAC
```

Emplear LEA suele revertir en un ahorro de tiempo. Si tenemos que acceder a un operando complejo varias veces en el curso del programa, conviene emplear previamente una instrucción LEA para calcular su dirección efectiva y almacenarla en un registro de datos. Por ejemplo, suponer que es necesario realizar complicados cálculos para establecer los valores de An y Xi antes de emplear el modo d8(Xn,Xi) para acceder a una matriz. Cada vez que empleamos d8(An,Xi) como operando, empleamos de 8 a 14 ciclos del procesador para calcular la dirección efectiva <ea>. Pero si empleamos:

```
LEA  d8(An,Xi),Am    Cargar Am con la <ea>; <ea> = d8 + An + Xi
```

Ahora ya tenemos un puntero para los subsiguientes cálculos.

Por otra parte, si vamos a llamar a una subrutina que necesita el operando d8(An,Xi), podríamos emplear un:

```
PEA  d8(An,Xi)       Meter d8 + An + Xi en la pila antes de un BSR o un JRS
```

La pila contendría entonces:

en la dirección del puntero	dirección de retorno de la subrutina
en la dirección del puntero + 4	la <ea> que metimos en la pila con PEA

Durante la subrutina, podemos recuperar esta <ea> usando una instrucción

```
MOVEA.L  $(SP),Am    Cargar Am desde SP + 4
```

Conceptualmente, LEA es idéntica al inverso de los paréntesis de indirección ( ).

An apunta a los datos en (An).
La dirección efectiva, <ea>, de (An) es An.
LEA (An),Am pone en Am el valor de An.

Veamos ahora el modo que nos queda: desplazamiento relativo al PC con desplazamiento e índice.

Direccionamiento relativo: Contador de programa con desplazamiento e índice

Por brevedad, denominaremos direccionamiento por PC indexado a este modo de direccionamiento. Este modo se indica d8(PC,Xi.Z), de donde se deduce que sigue las reglas dadas para el modo d8(An,Xi.Z), reemplazando An por el PC. La dirección efectiva del operando se calcula como:

$$<ea> = d8 + PC + Xi.Z$$

donde d8 representa un desplazamiento con signo de 8 bits (de −128 a + 127 bytes) y Xi.Z representa a cualquier registro seleccionado como registro índice. El código de tamaño Z puede ser L o W, y éste determina si se emplean los 16 ó 32 bits (se espera un signo en el caso de 32) del registro Xi (con signo) como desplazamiento índice adicional para el PC. El desplazamiento d8 y los códigos que especifican el registro Xi a emplear y su código de tamaño se almacenan en una única palabra de extensión. Para simplificar la escritura, muchas veces se denotará este modo como d(PC,Xi). Repetimos el aviso que dimos para el desplazamiento relativo al PC, d16(PC):

Los modos de direccionamiento relativos sólo pueden emplearse para operandos fuente

Cualquier combinación de código/operando en la que el destino es una dirección alterable en la forma d16(PC) es ilícita.

De nuestra discusión acerca de los códigos relocalizables se deduce que el direccionamiento por PC indexado permite definir y acceder a tablas y matrices. Vamos a reformular el programa 5-9 para mostrar un uso sencillo de este modo de direccionamiento con nuestra TABLA de datos.

* Programa 5.10
* Programa 5.9, empleando direccionamiento por PC indexado
* Calcular la media de dos números almacenados en la TABLA

```
            RORG
            CLR.L     D1                    Poner a cero el registro empleado como
                                            acumulador
            MOVE.Q    30,D0,A3              Ponemos el registro indice a cero
            MOVE.M    TABLE(PC,D0.W),D1     D1 contiene ahora la primera de las dos
                                            palabras a sumar
            ADDQ.W    #2,D0                 Incrementar el indice en 2
            ADD.W     TABLE(*PC,D0.W),D1    Sumar la segunda palabra
            ASR.W     #1,D1                 Dividir la suma entre 2
            <resto del programa>

* Se define un área de datos relativa a la dirección TABLA

TABLA       DC.W    $34A2,$00B6,$56F9,$11CC
            <continuar la tabla>

* La respuesta final se encuentra en la palabra menos significativa de D1:
* <$34A2 + $00B6>/2 = $1AAC
```

Como en los ejemplos anteriores de uso de la pareja RORG/etiqueta, el ensamblador establece la distancia entre PC y TABLA. El desplazamiento, sin embargo, está limitado a un valor con signo de 8 bits, de modo que el programa 5-9 funcionará correctamente sólo si la etiqueta TABLA está entre −128 y + 127 bytes de las instrucciones MOVE y ADD. La tabla, sin embargo, puede ser tan larga como se necesite; teniendo esto en cuenta, se ha modificado la posición de la tabla en programa 5-10. Se emplea el registro D0 como índice de la tabla, de modo que podemos extraer cualquier valor de la tabla.

Antes de la instrucción

```
ADD.W  TABLA(PC,D0.W),D1
```

D0 se ha puesto a 2, de modo que la dirección efectiva del operando fuente es:

$$\begin{aligned} \langle ea \rangle &= PC + \quad d8 \quad + D0 \\ &= PC + \langle TABLA - PC \rangle + 2 \\ &= TABLA + 2 \end{aligned}$$

de modo que sumamos la segunda palabra de la TABLA.

Modos relativos: Restricción de su uso a operandos fuente

Como ya se ha indicado dos veces, los modos relativos no pueden emplearse para operandos destino. Esta es una decisión deliberada de Moto-

rola para reducir el riesgo de que un programa, especialmente aquellos codificados de forma relocalizable, se autodestruya escribiendo inadvertidamente encima de sí mismo. Por ejemplo, si

```
MOVE.W  D0,8(PC)      ¡No permitido!
```

fuera una instrucción válida, reemplazaría la instrucción (o parte de la misma), que se encontrará 4 palabras (8 bytes) más adelante con lo que hubiera en la palabra menos significativa de D0. Hay una gran probabilidad de que se produzca el caos cuando el procesador alcanza una instrucción y ciegamente intenta ejecutarla. Por supuesto, se puede ser enormemente hábil y asignar deliberadamente a D0 una cadena de bits que correspondan a una instrucción válida. Los programas que se automodifican juegan un papel importante, pero el M68000 obliga a emplear una acción especial, para asegurarse de que el programa sabe lo que está haciendo.

Un efecto secundario de esta restricción es el gran cuidado que hay que tener al emplear etiquetas:

```
MOVE.W  D0,TABLA    Válido en secciones ORG
MOVE.W  D0,TABLA    Inválido en secciones RORG
```

Aunque estas dos instrucciones parecen lo mismo, sus modos de direccionamiento son diferentes. En el primer caso, TABLA es una dirección absoluta y, por tanto, válida. En el segundo, RORG es una dirección relativa, d16(PC), y, por tanto, inválida. Para evitar la restricción que se nos impone en las secciones RORG, empleamos:

```
LEA     TABLA,A1    Ponemos la dirección efectiva de la tabla en A1
MOVE.W  D0,(A1)     Movemos la palabra en D0 a la TABLA
```

Modos de direccionamiento: Resumen total

Hasta ahora hemos explorado los 12 modos de direccionamiento básicos del MC68000 (el MC68020 tiene seis más que estudiaremos en el capítulo 8). Ahora vamos a dar, es necesario, un resumen para ofrecer una visión de conjunto. Al describir algunas de las instrucciones, se ha indicado ocasionalmente que existían algunas restricciones en los modos de direccionamiento de los operandos origen y/o destino. Estas limitaciones puede parecer arbitrarias o producir confusión, incluso entre los programadores más experimentados. Gran parte de la literatura técnica contribuye a este sentir empleando una nomenclatura inadecuada para los diferentes modos de direccionamiento.

En el apéndice B, parte del cual se reproduce a continuación, se ha in-

tentado clasificar de forma lógica los modos de direccionamiento, lo que, esperamos, contribuirá a esclarecer cuándo determinados modos son válidos y por qué. El apéndice C lista todos los códigos de operación con los modos permitidos para los operandos fuente y destino.

Modos de direccionamiento del M68000

Cada modo de direccionamiento pertenece a todos o a alguno de los nueve grupos siguientes:

<ea> = Cualquier dirección efectiva
<rea> = Dirección efectiva de un registro
<dea> = Dirección efectiva de los datos
<mea> = Dirección efectiva de la memoria
<cea> = Dirección efectiva de control
<aea> = Dirección efectiva alterable (datos o memoria)
<adea> = Dirección efectiva alterable de datos
<amea> = Dirección efectiva alterable de la memoria
<acea> = Dirección efectiva alterable de control

Un * en la tabla indica los grupos para cada modo (y los modos para cada grupo).

Modo	*ea*	*rea*	*dea*	*mea*	*cea*	*aea*	*adea*	*amea*	*acea*
Dn	*	*	*			*	*		
An	*	*				*			
(An)	*		*	*	*	*	*	*	*
(An)+	*		*	*		*	*	*	
−(An)	*		*	*		*	*	*	
d(An)	*		*	*	*	*	*	*	*
d(An,Xi)	*		*	*	*	*	*	*	*
Abs.W	*		*	*	*	*	*	*	*
Abs.L	*		*	*	*	*	*	*	*
d(PC)	*		*	*	*				
d(PC,Xi)	*		*	*	*				
Inmed	*		*	*					
bd(An,Xi)	*		*	*	*	*	*	*	*

Modo	*ea*	*rea*	*dea*	*mea*	*cea*	*aea*	*adea*	*amea*	*acea*	
bd(PC,Xi)	*		*	*	*					MC6802
[bd,An],Xi,od	*		*	*	*	*	*	*	*	MC6802
[bd,An,Xi],od	*		*	*	*	*	*	*	*	MC6802
[bd,PC],Xi,od	*		*	*	*					MC6802
[bd,PC,Xi],od	*		*	*	*					MC6802

En el capitulo 8 se tratarán más detalladamente los modos de direccionamiento del M68020.

Descripción de los modos:

Dn	Registro de datos directo.
An	Registro de direcciones directo.
(An)	Registro de direcciones indirecto.
(An) +	Registro de direcciones indirecto con posincremento.
–(An)	Registro de direcciones indirecto con predecremento.
d16(An)	Registro de direcciones indirecto con desplazamiento; también se escribe d(An).
d16(An,Xi.Z)	Registro de direcciones indirecto con desplazamiento e índice; también se escribe d(An,Xi).
Abs.W	Dirección absoluta corta; también se escribe como xxx.W o como una etiqueta.
Abs.L	Dirección absoluta larga; también se escribe como xxx.L o como una etiqueta.
d16(PC)	Contador de programa con desplazamiento (modo relativo); también se escribe como d(PC) o como una etiqueta.
d8(PC,Xi.Z)	Contador de programa con desplazamiento e índice (modo relativo); también se escribe como d(PC,Xi) o como una etiqueta y (PC,Xi).
Inmed	Operando inmediato; también se escribe como #<datos>.

Modos para el M68000 solamente (véase el capítulo 8 para más detalle):

bd(An,Xi.Z*s)	Registro de direcciones indirecto con desplazamiento de la base e índice [similar al d(An,Xi.Z*s), pero bd puede ser d16 o d32].

bd(PC,Xi.Z*s)	Contador de programa con desplazamiento de la base e indice [similar al d(PC,Xi.Z*s), pero db puede ser d16 o d32].
[db,An],Xi.Z*s,od	Posindexado indirecto en la memoria.
[bd,An,Xi.Z*],od	Preindexado indirecto en la memoria.
[db,PC],Xi.Z*s,od	Contador de programa indirecto posindexado en la memoria.
[db,PC,Xi.Z*s],od	Contador de programa indirecto preindexado en la memoria.

Abreviaturas:

Dn	Cualquier registro de datos, D0-D7.
An	Cualquier registro de direcciones, A0-A7.
Xi	Cualquier An o Dn empleado como registro indice.
z	Indicador del tamaño de los datos (B, W, o L).
Z	Indicador del tamaño de los datos (L o W).
s	Factor de escala (1, 2, 4 ó 8).
PC	Contador de programa (20, 24 ó 32 bits).
SR	Registro de estatus.
CCR	Registro de códigos de condición.
d	Un desplazamiento extendido o en complemento a 2: d16, d8, d3, etc., indican el número de bits.
bd	Un desplazamiento de la base en complemento a 2 (16 ó 32 bits).
od	Un desplazamiento exterior en complemento a 2 (16 ó 32 bits).
xxx	Cualquier dirección absoluta válida.

Grupos de modos: Definiciones

Tomemos algunos grupos y modos para indicar por qué están asociados como se ha mostrado.

<dea> *dirección efectiva de los datos:* Como se ha visto, An permite solamente una aritmética restringida, no está clasificada como un operando de datos verdaderos.

<mea> *dirección efectiva de la memoria:* Excluye los modos de direccionamiento directo por registros en <rea>, Dn y An, que no son operandos de la memoria.

<adea> *dirección efectiva alterable de datos:* Todas las <dea> que pueden ser destinos válidos sujetas a cambiar por una instrucción. Obviamente, los datos inmediatos no son direcciones alterables, de forma

que el modo inmediato es una <dea> pero no una <adea>. De la misma manera hemos visto que los dos modos relativos no son alterables. Finalmente, An no es una <adea>, puesto que no es una <dea>.

<amea> *dirección efectiva alterable de la memoria:* Cualquier <mea> que pueda actuar como destino sujeto a cambios.

<aea> *dirección efectiva alterable:* Cualquier combinación de <adea> y <amea>, junto con An.

<cea> *dirección de control efectiva:* Un subconjunto de <mea> que representa sólo aquellas direcciones de la memoria a las que puede transferir el control, por ejemplo, mediante un JMP o un JRS.

Formatos de las instrucciones empleando grupos de modos

Esta agrupación de modos permite especificar las combinaciones de códigos de operación/operando de forma precisa y concisa. Se dan a continuación los formatos para algunas instrucciones que ya se han estudiado (para una lista completa, véase apéndice C):

MOVE.Z <ea>,<adea>

Fuente	<ea>	Todos los modos de direccionamiento son legales.
Destino	<adea>	Sólo se permiten direcciones efectivas alterables de datos. Excluido An, todos los modos relativos e inmediatos.

MOVEA.Z <ea>,An

Fuente	<ea>	Todos los modos de direccionamiento son legales.
Destino	An	Sólo se permite direccionamiento directo por registro.

ADD.z <ea>,Dn
ADD.z <ea>,Dn

Admite dos formatos:

Fuente		Todos los modos.
Destino		Sólo Dn.

o bien:

Fuente		Dn.
Destino	<amea>	Sólo se permite el modo de direccionamiento efectivo alterable. Excluido An, todos los modos relativos e inmediatos.

ADDI.z #<datos>,<adea>

Fuente	#<datos>	Sólo modo inmediato.
Destino	<adea>	Sólo se permiten direcciones efectivas alterables de datos. Excluido An, todos los modos relativos e inmediatos.

ADDQ.z #<datos>,<aea>

Fuente	#<datos>	Modo inmediato solamente.
Destino	<aea>	Solamente las direcciones efectivas alterables son legales. z = L, W sólo para An. Excluir todos los modos relativos e inmediatos.

MULS <dea>,Dn

Fuente	<dea>	Modos de direccionamiento por direcciones efectivas de datos; todos los modos excepto An.
Destino	Dn	Sólo registros de datos directos.

BRA ETIQUETA

Fuente	Ninguna	
Destino	ETIQUETA	Sólo se admiten los modos relativos d(PC) y (dPC,Xi).

JMP <cea>

Fuente	Ninguna	
Destino	<cea>	Modos que emplean direcciones efectivas de control.

El esquema anterior puede emplearse para cubrir instrucciones que, como MOVEM, tienen operandos poco corrientes.

Operandos implícitos

Para completar este esquema, notemos que algunas instrucciones hacen uso de registros del sistema sin hacer mención especifica del campo de operandos. Algunos ejemplos que ya se han visto son:

Instrucción		*Operando implícito*
BRA	Bifurcación incondicional	PC
JMP	Saltar siempre	PC
Bcc	Bifurcación condicional	PC
BSR	Saltar a una subrutina	PC,SP

Instrucción		*Operando implícito*
JSR	Saltar a una subrutina	PC,SP
RTS	Retorno desde una subrutina	PC,SP
RTR	Retornar y restaurar	PC,SP,CCR
MOVE to CCR		CCR
MOVE from CCR		Sr

Conclusión

Hasta ahora hemos expuesto el conjunto básico de instrucciones del M68000 y creemos que el lector tendrá una idea global del mismo. En los próximos capítulos cubriremos algunos grupos de instrucciones diversas, así como una discusión del MC68010 y del MC68020.

6

Otras instrucciones del M68000

En este capítulo estudiaremos otras instrucciones del M68000 agrupadas por funciones. Se empleará la agrupación por modos dada en el capítulo 5 y en el apéndice B para simplificar nuestra discusión de los modos de direccionamiento permitidos.

NOP: No opera

La instrucción NOP es una instrucción de una palabra que avanza el PC a la siguiente instrucción. No afecta a ninguno de los indicadores y no hay reglas complicadas para los operandos fuente y destino, puesto que éstos no existen. Sin embargo, esta instrucción es muy interesante de conocer si se está desarrollando un programa en ensamblador, sobre todo si las facilidades de edición/depuración de que se dispone son rudimentarias. A menudo es útil reservar espacio en el programa (cada NOP es una palabra) para subsecuentes inserciones; de manera similar se pueden borrar instrucciones, reemplazándolas por NOP. El código en lenguaje máquina para la instrucción NOP es $4E71, y en algunos sistemas se puede borrar insertando $4E71 directamente en el código objeto sin necesidad de reensamblado.

Manipulación de bits

Este primer gran grupo de instrucciones permite manipular bits y grupos de bits dentro de los registros, registros especiales y la memoria.

Como ya se ha visto, la mayoría de las instrucciones presenta variantes L, W y B para manipular partes específicas del operando. Hay muchas ocasiones en que es necesario aislar parte del operando, por ejemplo, cuando se necesita acceder a la parte superior o intermedia de una doble palabra. A menudo también es interesante disponer de nuestros propios registros de estatus o condición; por ejemplo, en un programa de nóminas podemos encontrar un "byte de estatus del empleado", con sus 8 bits, representando sexo (1 bit) estado civil (2 bits), etc. El sistema operativo también se comunica a menudo con el programa a base de cambiar el estatus de ciertos indicadores en determinadas localizaciones.

Muchas de las instrucciones para el manejo de bits del M68000 están orientadas a simplificar la idea de probar, poner a uno o a cero bits, que tienen un significado lógico más que aritmético. De modo que revisemos antes los operadores lógicos básicos.

TABLA 6.1

Resumen de las instrucciones lógicas

Instrucción	*Operando*	*Efectos sobre el CCR*
AND.L/W/B OR.L/W/B	<dea>,Dn o Dn,<amea>	X__N*Z*V0C0
NOT.L/W/B	<adea>	X__N*Z*V0C0
EOR.L/W/B	Dn<adea> (*)	X__N*Z*V0C0
ANDI{.B} ORI{.B} EORI{.B}	#xxx,CCR	X*N*Z*V*C*
ANDI{.W} ORI{.W} EORI{.W}	#xxx,SR (**)	X*N*Z*V*C*

(*) No se admite un dirección de la memoria como operando fuente.

(**) Instrucción privilegiada.

<dea> = Modos de direccionamiento por dirección efectiva: todos menos An.

<amea> = Direccionamiento por direcciones alterables efectivas: (An), (An)+, −(An), d(An), d(An,Xi), Abs.W, Abs.L.

<adea> = Modos de direccionamiento alterables por los datos: <amea> + Dn.

{%} indica que el tamaño de los datos va implícito.

Notación para el CCR: _ indica sin cambios, * indica cambios según las reglas del CCR, 0 indica que el indicador siempre se pone a 0.

Operaciones lógicas

El M68000 dispone de cuatro instrucciones lógicas básicas: NOT, AND, OR y EOR, que se resumen en la tabla 6.1. Estas instrucciones se emplean en muchas ocasiones, tales como poner a cero o uno determinados indicadores bits, y para enmascarar o extraer campos de datos de los registros o de la memoria. Vamos a recapitular lo que hace cada una de las instrucciones y después las veremos en acción.

NOT

El NOT lógico significa volver cada 1 en un 0 y cada 0 en un 1 en el operando designado. Matemáticamente, esto es equivalente a formar el complemento a 1 del operando, de modo que "NOT 01011010" → 10100101. NOT requiere sólo un operando que sirve como fuente y destino. El formato es:

```
NOT.z    <dda>
```

donde <dda> significa dirección de datos alterables, es decir, cualquier modo de direccionamiento excepto An, d(PC), d(PC,Xi) e Inmediata. Como es usual, el código para z dicta cuáles de los 32, 16 u 8 bits del operando resultan afectados por el NOT.

Por ejemplo,

```
NOT.B    D1
```

en la figura 6.1 invierte el byte menos significativo de D1 sin afectar a los otros tres bytes. El CCR cambia como en el caso de un MOVE, según se muestra en la tabla 6.1.

AND

AND requiere operandos fuente y destino. La base para un AND lógico se obtiene de la siguiente tabla de verdad:

Fuente	0	0	1	1	
Destino	0	1	0	1	
AND	0	0	0	1	→ Nuevo destino

En otras palabras, AND opera bit a bit, comprobando los valores de los bits en la fuente y el destino y formando un nuevo bit en el destino de acuerdo con las reglas dadas más arriba. A menos que ambos, fuente y destino,

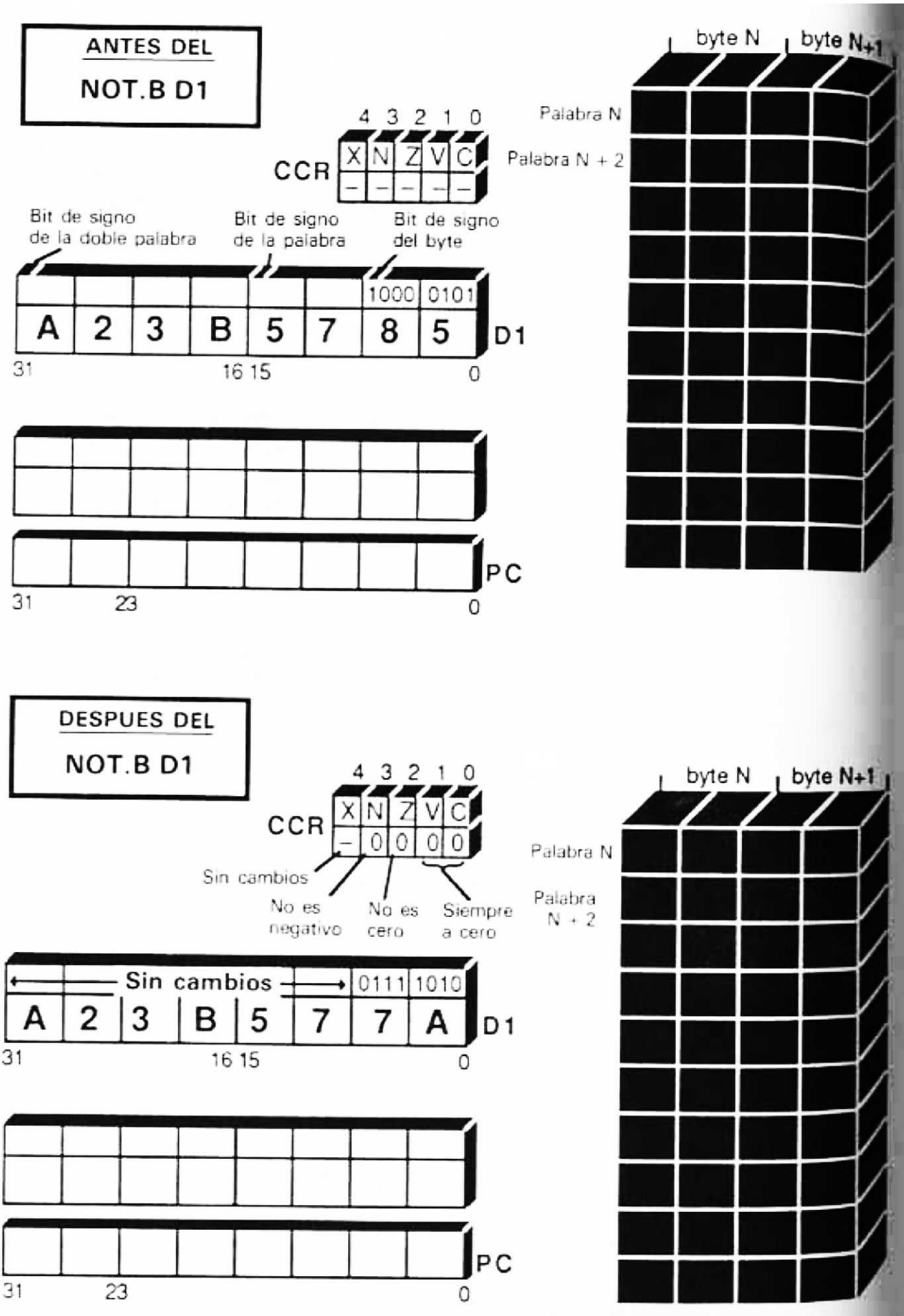


Figura 6.1
NOT.B D1

tengan un 1 en la misma posición, AND pondrá un cero en dicha posición. Hay dos formatos permitidos:

```
AND.z   <ded>,Ds
AND.z   Dn,<dem>
```

Donde <ded> significa dirección efectiva de datos, es decir, cualquier modo de direccionamiento, excepto An, y <dem> significa cualquier dirección efectiva de la memoria, es decir, cualquier dirección de datos alterables, excepto Dn.

Nótese que la fuente o el destino deben ser registros de datos. También puede suceder que ambos sean registros de datos.

Las figuras 6.2 y 6.3 muestran dos ejemplos. AND cambia el CCR como en el caso de un MOVE, como se ve en la tabla 6.1.

OR

El OR lógico se refiere al OR inclusivo. Aparte de emplear la tabla de verdad que se muestra más abajo, el OR funciona como el AND, empleando dos operandos y los mismos modos de direccionamiento y tamaños de datos; además, OR cambia el CCR de la misma forma que AND y MOVE.

Fuente	0	0	1	1	
Destino	0	1	0	1	
AND	0	1	1	1	→ Nuevo destino

Nótese que un OR produce un 1 siempre que la fuente o el destino sean un 1, de aquí el nombre de OR inclusivo.

Los formatos permitidos son, como en el caso del AND:

```
OR.z    <ded>,Ds
OR.z    Dn,<dem>
```

Las figuras 6.4 y 6.5 muestran al OR en acción sobre diferentes operandos.

EOR

EOR es el OR exclusivo, como se muestra en la tabla de verdad del EOR:

Fuente	0	0	1	1	
Destino	0	1	0	1	
AND	0	1	1	0	→ Nuevo destino

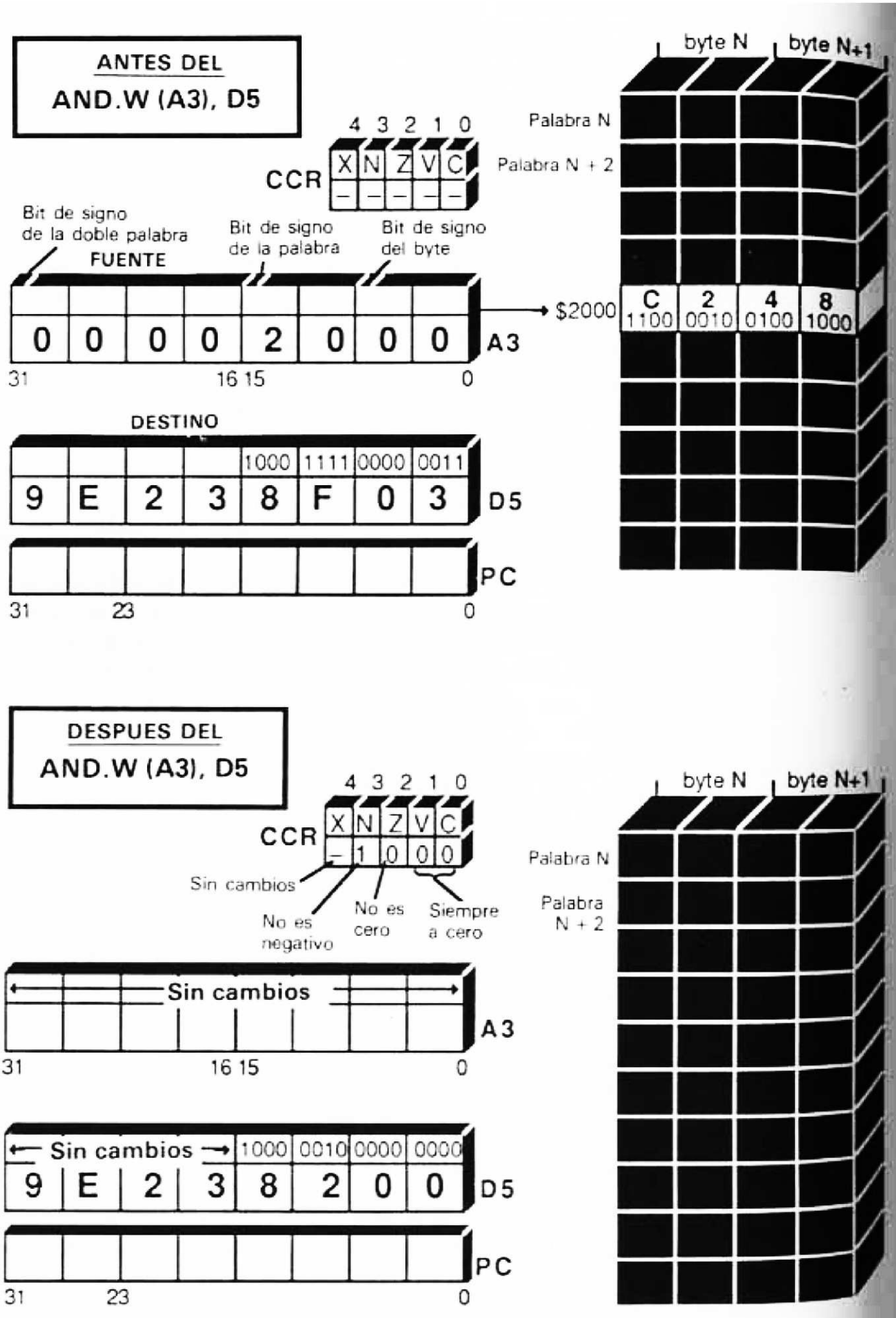


Figura 6.2
AND.W (A3),D5

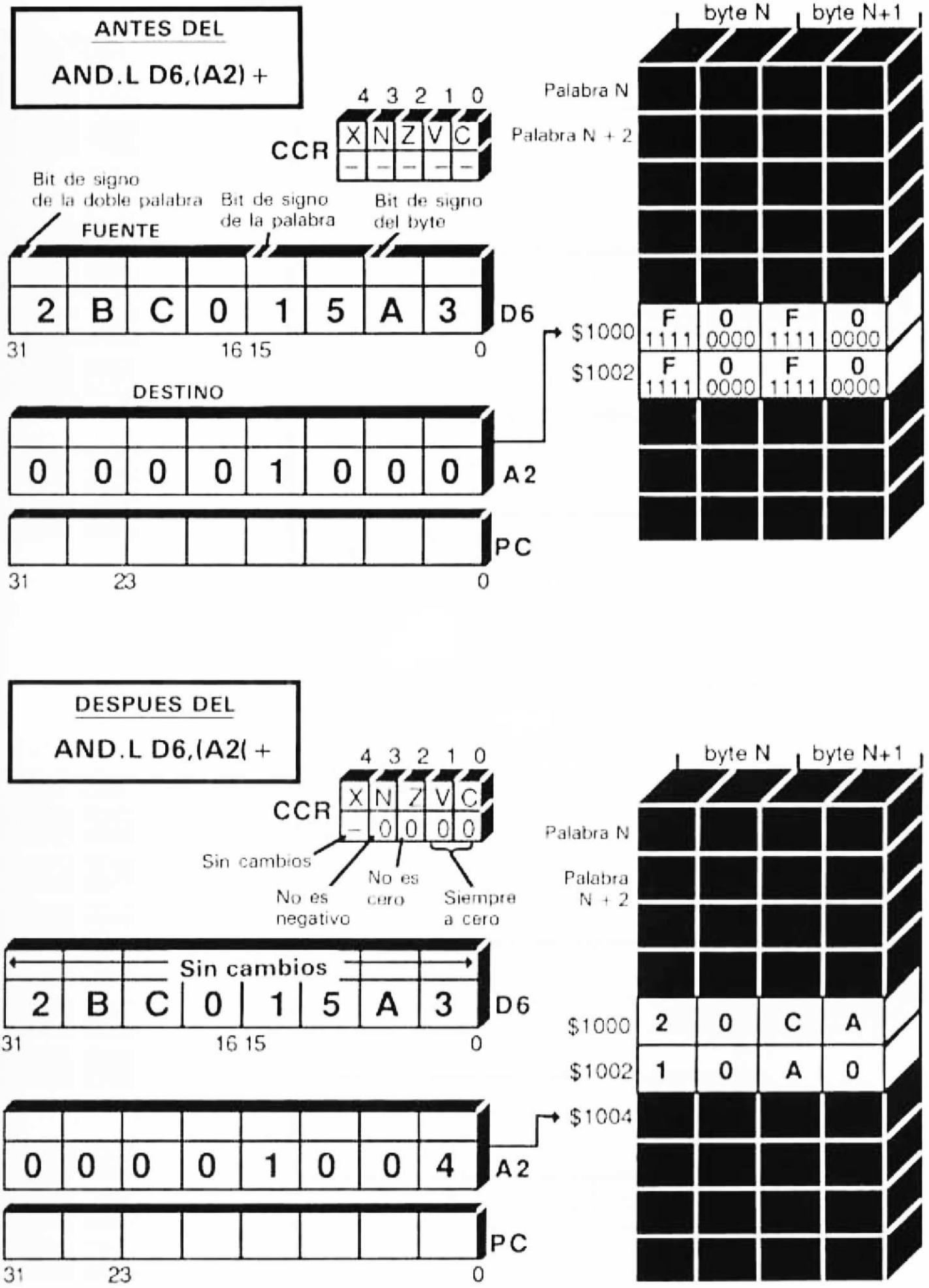


Figura 6.3
AND.L D6,(A2) +

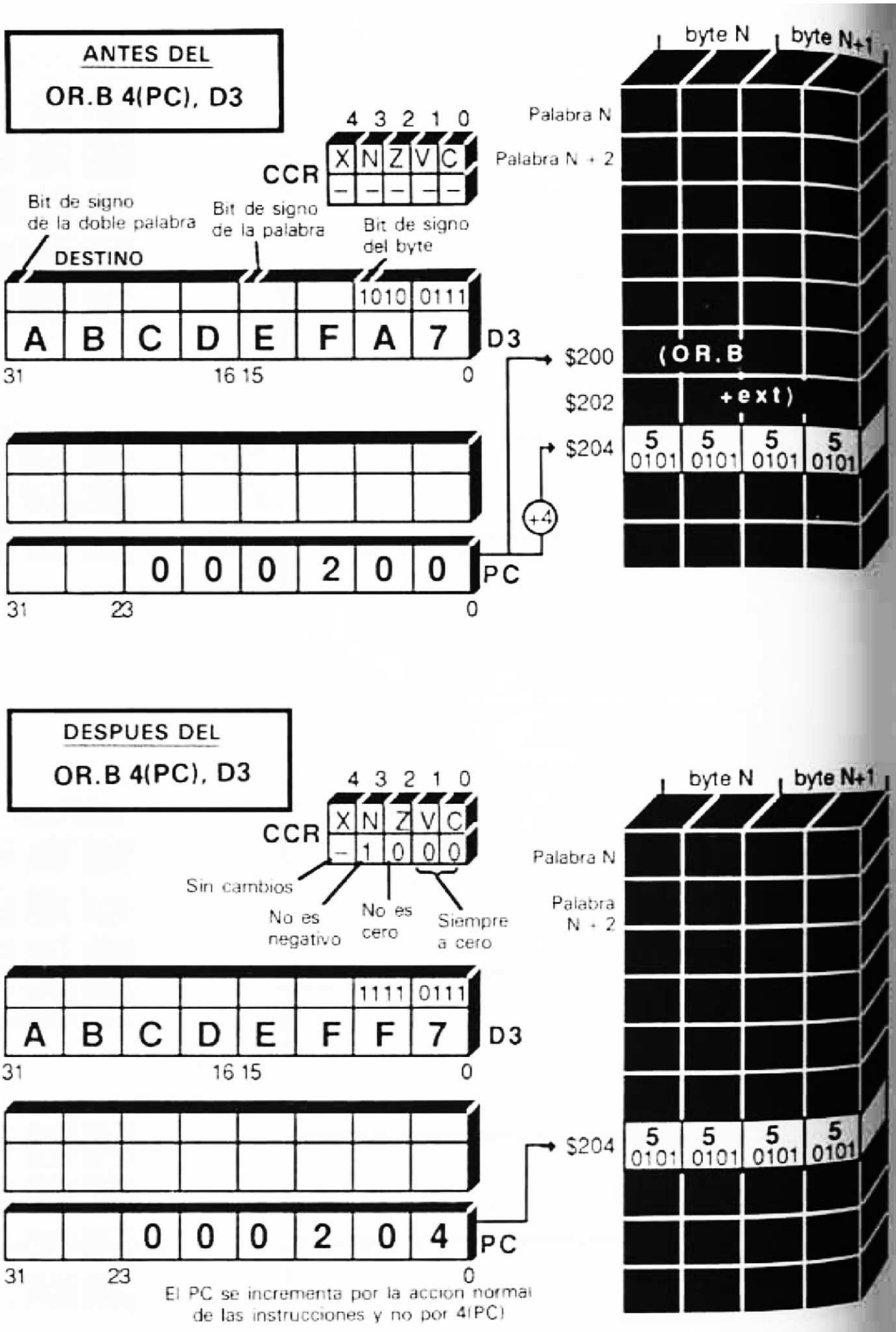


Figura 6.4
OR.B 4(PC),D3

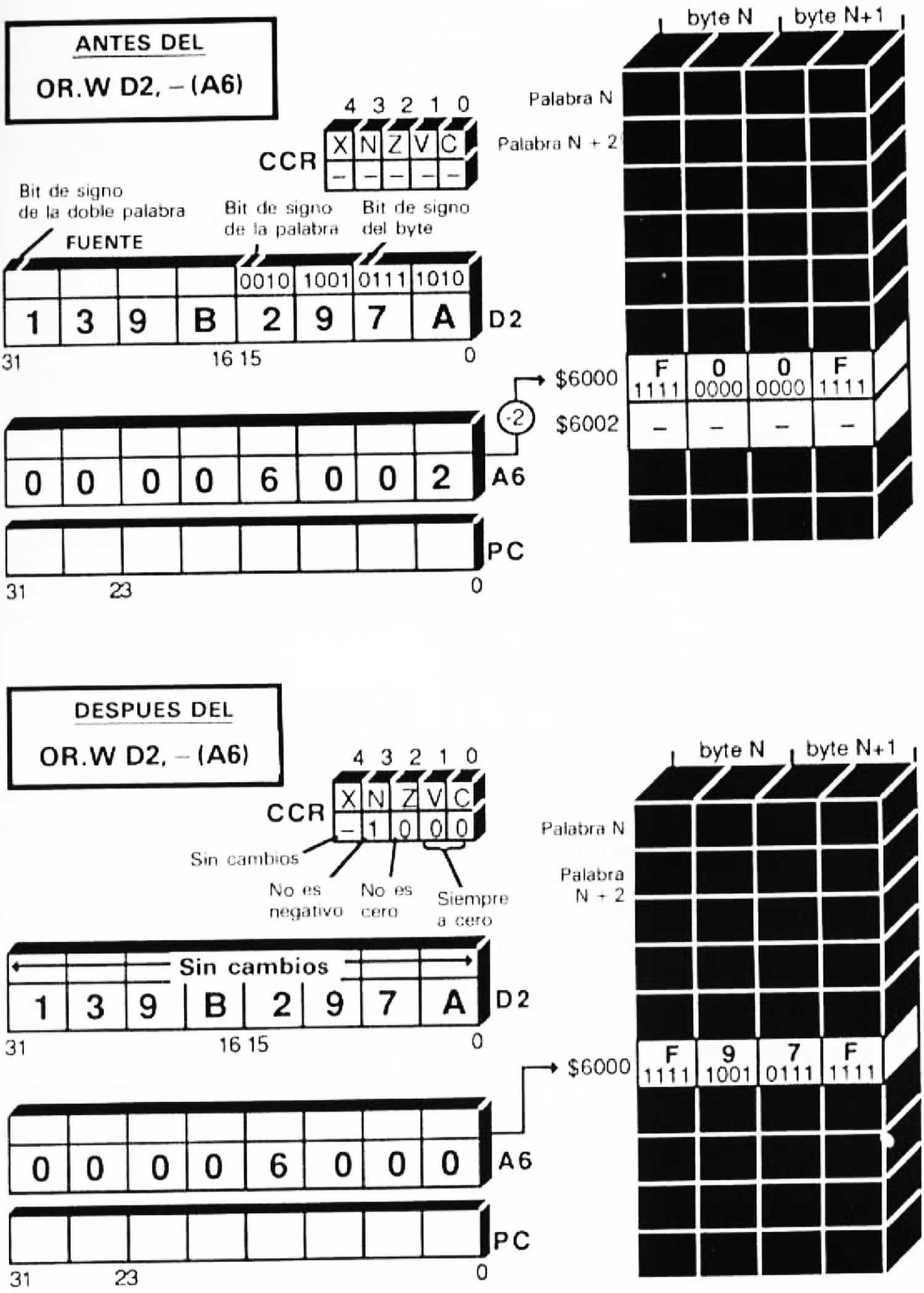


Figura 6.5
OR.W D2,–(A6)

La diferencia entre el OR y el EOR es el cero en la última columna. Un EOR busca en uno, pero no en los dos operandos, cuando da valores a los bits de destino. Su principal utilidad, como veremos, es que puede emplearse para invertir bits escogidos dentro de un campo sin molestar a los demás, a diferencia del NOT, que invierte todos y cada uno de los bits. A diferencia de AND y OR, EOR sólo permite un formato:

```
EOR.z    Dn,<dda>
```

No se puede emplear una dirección de la memoria como fuente del EOR

EOR cambia el CCR de la misma forma que AND, OR y MOVE, según se muestra en la figura 6.6.

Instrucciones lógicas: Variaciones en modo inmediato

Con la excepción de NOT, las instrucciones lógicas admiten formatos fuente en modo inmediato: ANDI, ORI y EORI, que presentan todos la misma forma:

```
ANDI.z
ORI.z    #<dato>,<dda>
EORI.z
```

El tamaño del dato en #<dato> debe ser d32, d16, d8, dependiendo del código de tamaño empleado para # (L, W, o B). La instrucción toma una o dos palabras para almacenar el dato inmediato.

El uso del modo fuente con AND es muy común. Para enmascarar o aislar un determinado operando se crea una máscara, #<masc>, con unos en las posiciones seleccionadas y ceros en las posiciones descartadas. Dado que #3 = 00000011, el ANDI.B, en la figura 6.7, pone a cero todos los bits, excepto los dos menos significativos en D2.

Con EORI.W, la mácara, #<masc>, se elige con unos y ceros en las posiciones que se desea invertir y con ceros en aquellas que se desea dejar intactas. En la figura 6.8, los bits en el byte menos significativo de la memoria en (A1) resultan todos invertidos por el "FF", mientras que el byte más significativo permanece inalterado.

Cambiando el CCR

Un formato especial permite cambiar cualquiera o todos los indicadores en el byte del CCR:

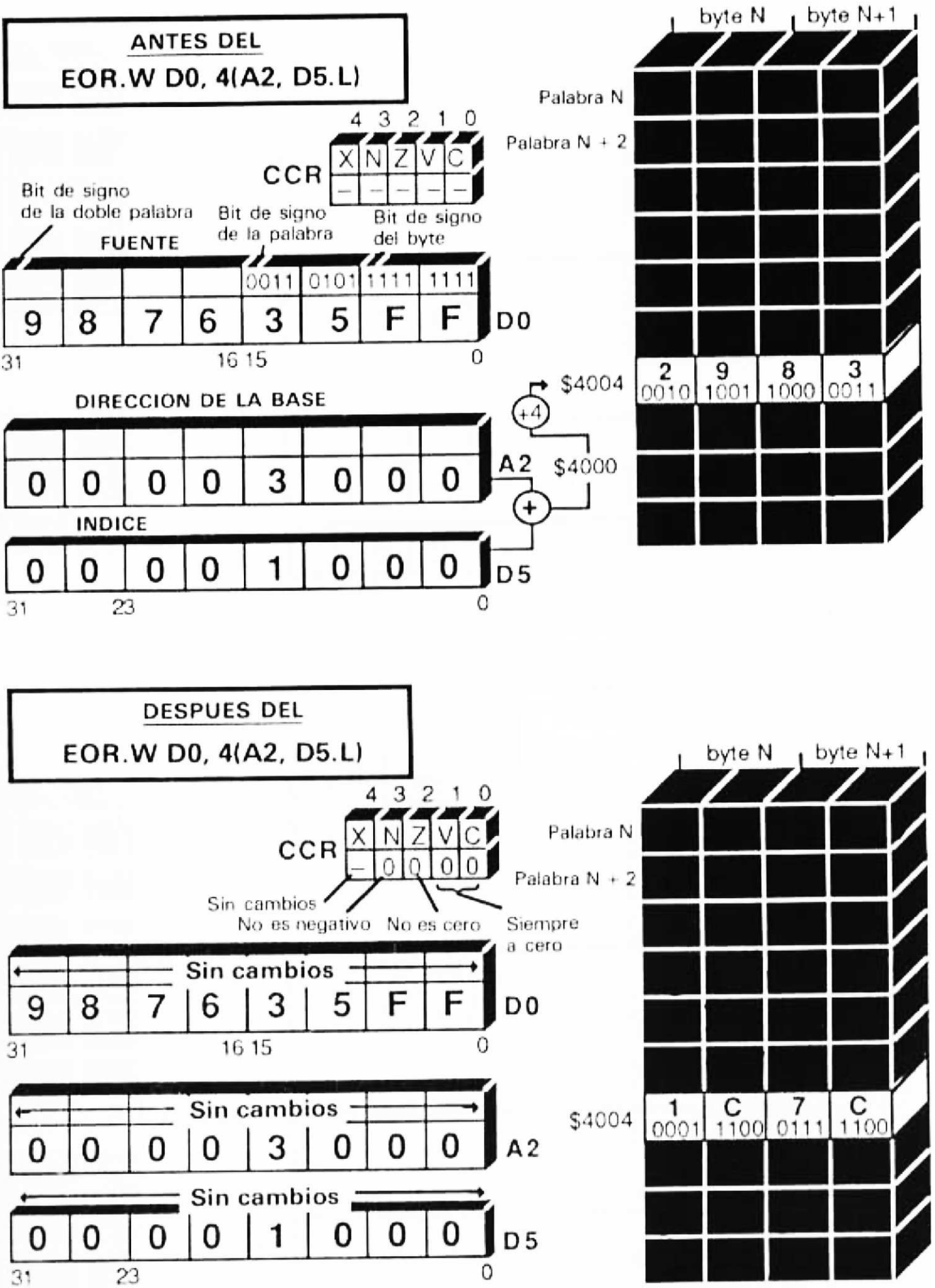


Figura 6.6
EOR.W D0,4(A2,D5.L)

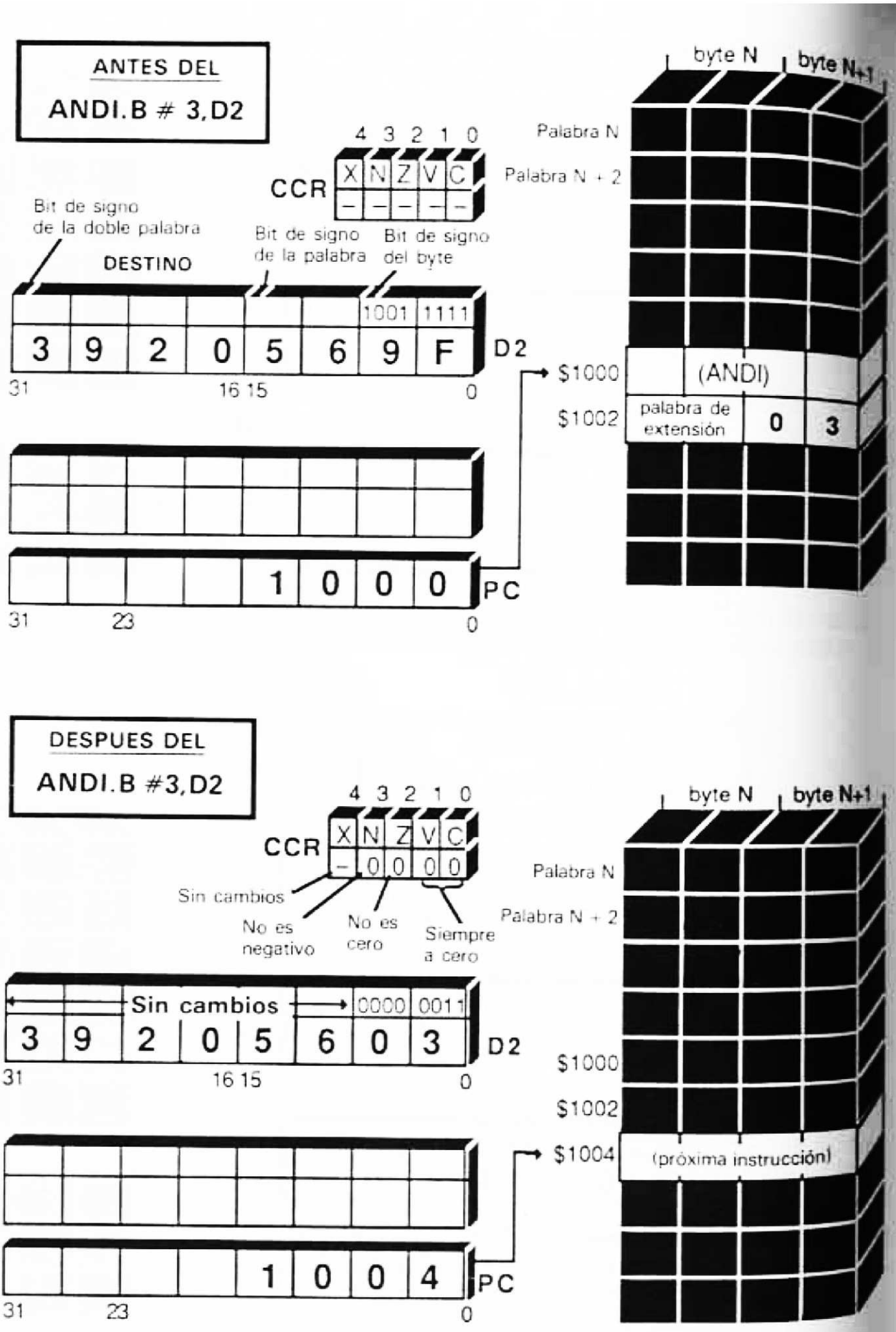


Figura 6.7
ANDI.B #3,D2

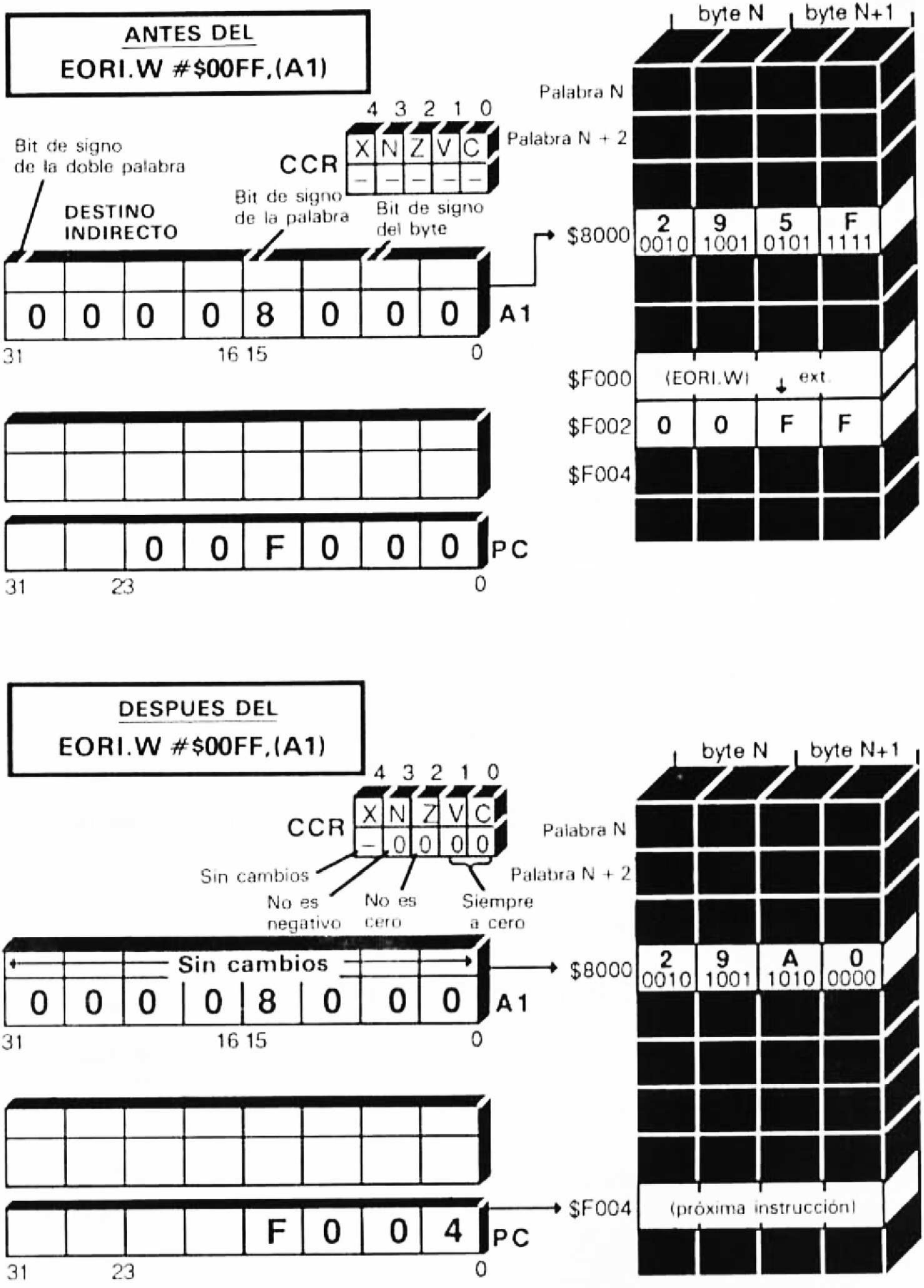


Figura 6.8
EORI.W #$00FF,(A1)

```
ANDI.B
ORI.B     #<dato>,CCR
EORI.B
```

Nótese que el destino se indica simplemente mediante CCR y que sólo se permite el modo byte. La B es opcional, pero aquí se usará para remarcar lo que realmente está sucediendo. Para usar este formato conviene recordar que:

Bit 0 = C *flag*
Bit 1 = V *flag*
Bit 2 = Z *flag*
Bit 3 = N *flag*
Bit 4 = X *flag*
Bits 5-7 no se emplean

Una aplicación común es la de poner a cero el indicador X sin cambiar el resto de los *flags* del CCR. Para esto se emplea:

```
ANDI.B    #$EF,CCR
```

puesto que $EF = 11101111. Esto es obligado antes de emprender cálculos que involucren aritmética extendida, por razones que se explicarán más tarde en este capítulo, en la sección de matemáticas de multiprecisión. Es instructivo comparar el método de cambiar el CCR mediante un ANDI con aquel que emplea un MOVE para el mismo fin. La instrucción:

```
MOVE.W  #$EF,CCR      Sólo se mueve un byte en lugar de toda la palabra
```

pondrá, efectivamente, a cero el *flag* X, pero cambiará todos los otros *flags* a 1.

Cambiando el SR (registro de estado)

En modo supervisor, y sólo en él, se pueden cambiar ambos bytes del SR, el byte más significativo, o byte del sistema, y el CCR (byte menos significativo). Para esto se emplean los formatos:

```
ANDI.W
ORI.W     #<d16>,SR    Instrucción privilegiada. Sólo en modo supervisor
EORI.W
```

El byte del sistema contiene los *flags* de ST (estado y traza), así como la máscara de interrupciones de tres bits. ¡De aquí la necesidad de protección

que supone ser una instrucción privilegiada! (MOVE-a-SR está protegida de modo similar.) Se tratará este particular en detalle más tarde, en la sección "Más acerca del privilegio".

Operaciones lógicas: Resumen

Eligiendo apropiadamente la máscara en operando fuente, se pueden alterar determinados bits o *flags* en el operando destino. Las reglas son:

NOT	Se invierten todos los bits en el destino.
AND	0 en la fuente: Pone a 0 el bit seleccionado en el destino. 1 en la fuente: Selecciona los bits que permanecerán inalterados en el destino.
OR	0 en la fuente: Selecciona los bits que permanecerán inalterados en el destino. 1 en la fuente: Pone a 1 el bit seleccionado en el destino.
EOR	0 en la fuente: Selecciona los bits que permanecerán inalterados en el destino. 1 en la fuente: Selecciona los bits que serán invertidos en el destino.

APLICACION PRACTICA

Problema: Poner a cero el octavo bit (el bit en la séptima posición) de cada byte de una cadena de caracteres ASCII en la memoria.

Preliminares: En algunos sistemas, el alfabeto ASCII de siete bits se extiende usando el octavo bit (el bit en la séptima posición) para aplicaciones de control no estándar. En otras ocasiones se emplea este bit para funciones de paridad (véase el problema 6.2 como ejemplo). Este octavo bit es en ocasiones un engorro y debe ser suprimido.

Datos: Se da una cadena de caracteres ASCII no nula que comienza en la posición de la memoria almacenada en A6, es decir, el primer byte de la cadena es el byte (A6). El fin de la cadena lo señala el byte cero $00.

Solución: Programa 6.1

```
BUCLE   ANDI.B  #$7F,(A6)+   La fuente inmediata es "01111111".
                              A6 se incrementa al "siguiente byte"
        TST.B   (A6)          ¿Es el nuevo byte $00?
        BNE.S   BUCLE         No: repetimos
        <resto del programa>  Si: fin de la cadena
```

Notas al programa: Estamos empleando $7F como máscara, #<masc>, tiene unos en todas las posiciones que no se desee cambiar (del 0 al 6) y un 0 en la posición que deseamos cambiar. La lógica subyacente es:

1 AND x = x (x sin cambios)
0 AND x = 0 (x se pone a cero)

Nótese la potencia del modo de direccionamiento (A6)+ cuando se usa en conjunción con el byte nulo como fin de la cadena. Esta es una técnica muy empleada cuando se trabaja con entes de longitud variable como cadenas de caracteres. La linea ANDI.B pone a cero el octavo bit en la dirección A6 y entonces incrementa A6 en 1 para obtener la dirección del siguiente byte. TST.B simplemente comprueba el nuevo byte (sin incrementar) y BNE indica bifurcación si no es cero. Así se continúa examinando la cadena de caracteres hasta alcanzar su final.

Instrucciones de desplazamiento y rotación

Hay ocho instrucciones de desplazamiento y rotación que permiten mover los bits dentro de los registros, o la memoria, hacia la derecha o hacia la izquierda. Dando un valor al contador de desplazamiento, que determina el número de desplazamientos, se puede cambiar la posición de los bits dentro de bytes, palabras y dobles palabras. Los desplazamientos también tienen funciones aritméticas.

Se ha visto en el capítulo 4 que en aritmética binaria simple desplazar los bits de una cadena una vez a la derecha es equivalente a multiplicar por 2, mientras que desplazarlos a la izquierda es equivalente a dividir por 2. De modo que un uso evidente de las instrucciones de desplazamiento es emplearlas como métodos rápidos de multiplicación y división por potencias de 2.

Sin embargo, antes de comenzar con los desplazamientos a izquierda y derecha de cadenas de bits, debemos saber si éstas representan números con signo o sin signo. Recordemos que en números con signo el bit significativo es el bit de signo. Si olvidamos este factor cuando realicemos desplazamientos, podemos obtener resultados erróneos. Debido a este problema, el M68000 ofrece dos tipos de desplazamientos: desplazamiento lógico y aritmético. La diferencia entre ellos estriba en cómo tratan el bit de signo. Veamos primero los desplazamientos lógicos, que son los más sencillos.

Instrucciones de desplazamiento lógico

Un desplazamiento lógico se emplea principalmente sobre números sin signo. El desplazamiento actúa moviendo una cadena de bits un determi-

nado número de posiciones a la izquierda (LSL) o a la derecha (LSR), insertando ceros en el extremo correspondiente. Cuando se inserta un cero, podemos imaginar cómo resulta desplazado el resto de los bits de la cadena uno detrás de otro, con un pobre bit al final saliéndose por el extremo. Las figuras 6.9 y 6.10 muestran dos desplazamientos lógicos en una palabra.

El número de desplazamientos efectuados se denomina cuenta de desplazamiento y se especifica mediante un operando inmediato, indicado como siempre por el símbolo #. Así, en nuestros ejemplos, el contenido de la palabra de orden más bajo de D1 sufre tres desplazamientos a la izquierda y dos a la derecha. El contador de desplazamiento inmediato puede tomar valores de #1 a #8. Esto significa, obviamente, que el máximo valor del contador de desplazamiento inmediato es #8.

Para efectuar más de ocho desplazamientos se necesita el formato

```
LSL.z    Dm,Dn
```

o

```
LSR.z    Dm,Dn
```

Donde el registro de datos Dm contiene el contador de desplazamiento. Con este formato se pueden realizar de uno a 64 desplazamientos a la derecha o a la izquierda. Sólo los 6 bits menos significativos de Dm (bits del 0 al 5) se emplean para el contador de desplazamiento (lo que explica el límite de 64 para el contador de desplazamiento en este formato). El término correcto para esto, que nos ahorrará mucha palabrería posteriormente, es: registro de datos del contador de desplazamiento = Dm módulo 64 (que a menudo se abrevia a Dm mod 64). Por ejemplo, si:

Dm = 3 ó 67 ó 131, entonces el contador de desplazamiento será = Dm mod 64 = 3,

o si

Dm = 63 ó 127 ó 191, entonces el contador de desplazamiento será = Dm mod 64 = 63.

La regla es dividir Dm entre 64, hasta que el resto sea menor de 64. La mayoría de los relojes funciona en base al concepto (horas mod 12), de modo que este concepto es omnipresente.

Usar Dm como contador de desplazamiento proporciona una mayor flexibilidad que emplear el contador de desplazamiento inmediato: por ejemplo, el contador de desplazamiento puede variarse de una manera dinámica en un programa. Los desplazamientos inmediatos son para pequeños desplazamientos fijos. Si el contador de desplazamiento se hace cero, por accidente o diseño, no se produce ningún desplazamiento, pero el CCR queda afectado (véase más abajo).

El código de tamaño de datos z especifica cuántos bits del registro des-

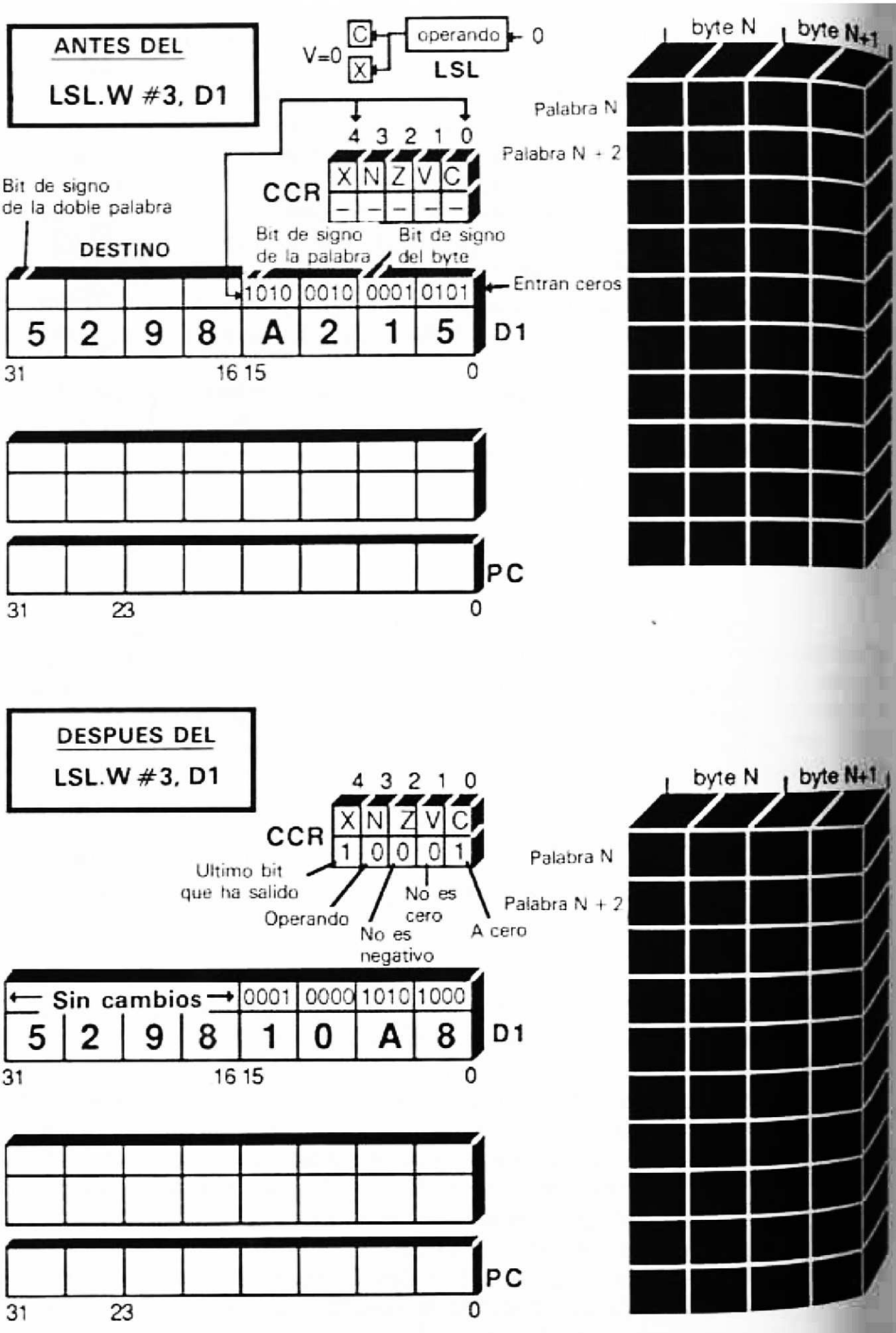


Figura 6.9
LSL.W #3,D1

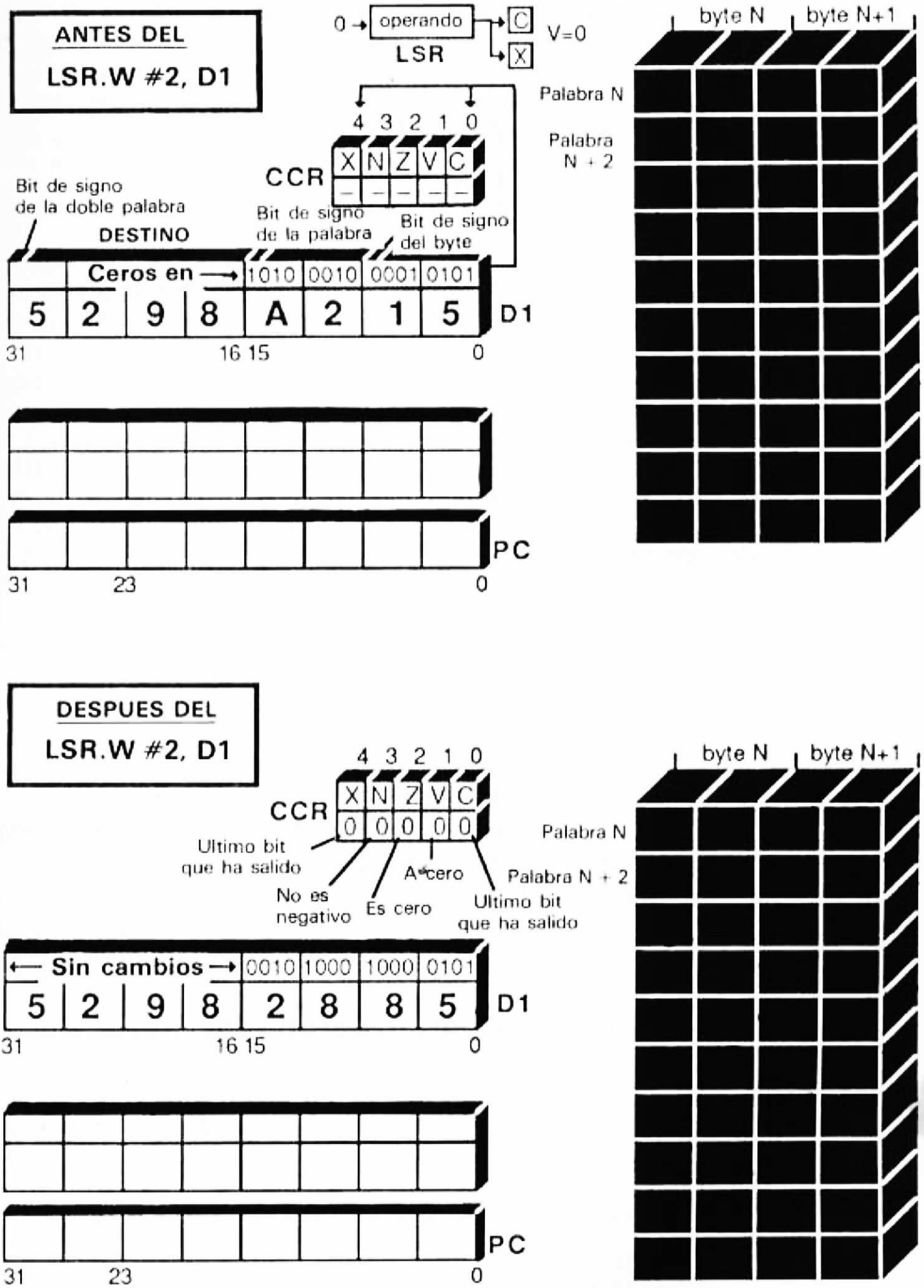


Figura 6.10
LSR.W #2,D1

tino resultarán afectados por el desplazamiento. En el ejemplo LSL.W, los 16 bits menos significativos, es decir, la palabra de menor peso, son los afectados por el desplazamiento. Si hubiéramos empleado L o B, el desplazamiento habría afectado al total de los 32 bits o habría quedado confinado a los 8 bits menos significativos de D1.

Desplazamientos lógicos y el CCR

¿Qué es lo que ocurre con el bit que sale del registro cuando se produce un desplazamiento? Como se muestra en las figuras 6.9 y 6.10, se almacenan tanto en el indicador C (acarreo) como en el indicador X (extendido) del CCR (registro de códigos de condición). Si miramos en los indicadores C o X después de que se complete una instrucción de desplazamiento, obtendremos el valor 0 ó 1 del último bit que salió del registro afectado por el desplazamiento. Los indicadores N (negativo) y Z (cero) indican si los bits en el registro D1 después del desplazamiento representan un número negativo o un cero. Recordemos que si está tratando con números sin signo, el indicador N sólo indica el estado del bit más significativo y no el signo del operando. El indicador V siempre se pone a 0. La siguiente tabla resume los cambios del CCR:

Indicador X: Se pone a 1 si Dn resulta negativo tras el desplazamiento; de lo contrario se pone a 0.
zamiento es 0, es decir, si no se produce desplazamiento alguno.

Indicador N: Si se pone a 1 si Dn resulta negativo tras el desplazamiento, de lo contrario se pone a 0.

Indicador Z: Se pone a 1 si Dn resulta cero tras el desplazamiento.

Indicador V: Siempre se pone a 0.

Indicador C: Resulta afectado del mismo modo que el indicador X, pero se pone a 0 si el contador de desplazamiento es 0, es decir, si no se produce desplazamiento alguno.

Desplazamientos lógicos para operandos de la memoria

Los desplazamientos lógicos de cadenas de bits de la memoria sufren dos restricciones. Sólo se pueden desplazar palabras en la memoria y el contador de desplazamiento debe ser 1.

El formato para un desplazamiento en la memoria es:

```
LSL.W   <dem>
LSR.W   <dem>
```

donde <dem> indica dirección efectiva de la memoria.

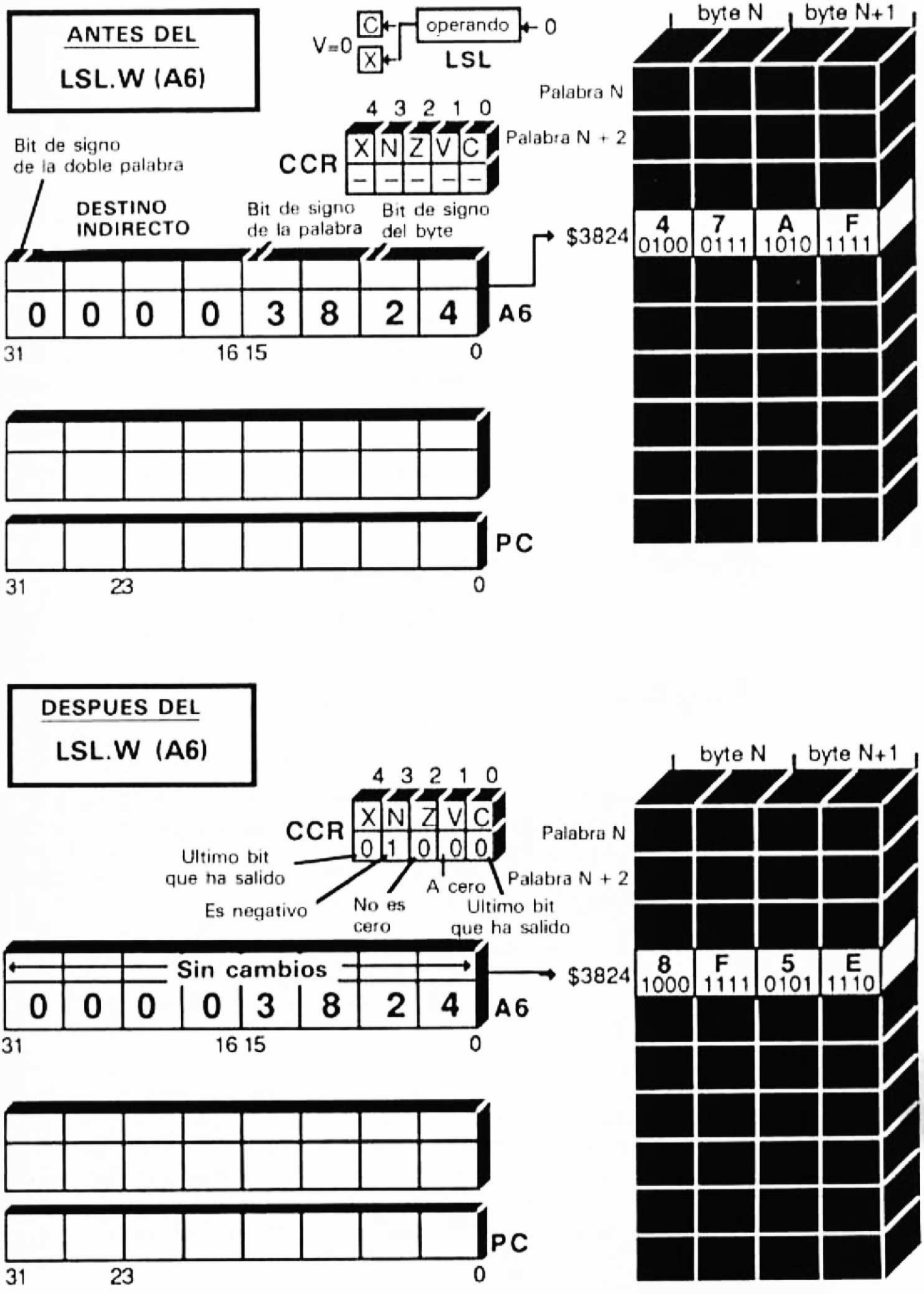


Figura 6.11
LSL.W (A6)

El contador de desplazamiento es siempre 1, por lo que no se lista explícitamente como operando.

El CCR cambia como en un desplazamiento lógico en un registro.

Indicador X: Se pone al valor del último bit que salió de la palabra destino.

Indicador N: Se pone a 1 si Dn resulta negativo tras el desplazamiento, es decir, si el bit en la posición 15 es 1; de lo contrario se pone a 0.

Indicador Z: Se pone a 1 si Dn resulta cero tras el desplazamiento.

Indicador V: Siempre se pone a 0.

Indicador C: Resulta afectado del mismo modo que el indicador X (dado que el contador de desplazamiento no es nunca 0).

En la figura 6.11 desplazamos la palabra situada en (A6) uno a la izquierda. Nótense los cambios en el CCR.

APLICACION PRACTICA

Problema: Comprobar la paridad de un carácter ASCII de 8 bits. Contar los unos en un código ASCII y poner el byte D3 a cero si la paridad es par; poner el byte D3 a uno si la paridad es impar (condición de error).

Preliminares: El código ASCII estándar de 7 bits (bits de 0 a 6) que se encuentra en el apéndice F asigna caracteres a cada una de las 128 combinaciones de 7 bits existentes desde $00 hasta $7F. Para comprobar la exactitud de una transmisión, se incluye en ocasiones un octavo bit (bit 7) a cada código. Las normas de asignación de este bit son:

Si el número de unos en el código de 7 bits es impar,
el número se pone al valor 1.

Si el número de unos en el código de 7 bits es par,
se pone a 0.

Por tanto, todos los códigos ASCII de 8 bits válidos tendrán un número par de bits 1, y se denomina a este esquema de comprobación **paridad par.** A medida que se recibe cada carácter se puede comprobar si algún bit se ha perdido o añadido durante la transmisión. Existen métodos de comprobación más elaborados, pero el método de paridad par-impar es adecuado en muchas circunstancias. Su mayor ventaja es que cualquier número impar de cambios transformará un código ASCII válido en uno inválido y, por tanto, se podrá detectar el error. La peor clase de errores es aquella que no deja huellas inmediatas.

Por ejemplo, el código ASCII para el número 5 es $35; si el bit 0 cambia de 1 a 0, recibiremos el código $34, que es el número 4. Este tipo de error

podría pasar inadvertido hasta que el interesado comprobase su libreta de cuentas. El código ASCII de 8 bits detectaría rápidamente este error:

0110101 = $35 = "5"	0110100 = $34 = "4"	ASCII de 7 bits
Perder el bit 0 de "5" da	0110100 = $34 = "4"	
00110101 = $35 = "5"	10110100 = $B4 = "4"	ASCII de 8 bits
Perder el bit 0 de "5" da	00110100 = $34 = INVALIDO	(paridad (errónea)

Datos: Un carácter ASCII de 8 bits en el byte de menor orden de D1.

Ejemplos:

Byte D1 = $A2 = 10100010 dará D3 = 1 (3 unos en D1 = impar)
Byte D1 = $47 = 01000111 dará D3 = 0 (4 unos en D1 = par)

Solución: Programa 6.2.

```
        CLR.B   D3          Poner el byte D3 a 0
DESP    TST.B   D1          ¿Es el byte D1 0?
        BEQ.S   HECHO       Si D1 es 0 hemos acabado
        LSL.B   #1,D1       Si D1 no es 0 hacemos un desplazamiento lógico
                            a la izquierda
        BCC.S   DESP        ¿Ha salido un 1? Si no es así, no hay acarreo
                            y vamos a DESP; si ha salido un 1 lo anotamos en D3
        EORI.B  #1,D3       Invertimos el bit 0 en D3, es decir, si D3 = 0,
                            hacemos D3 = 1, y si D3 = 1, hacemos D3 = 0
        BRA.S   DESP        Volvemos a DESP
HECHO   TST.B   D3          ¿Es D3 0?
        BNE     ERROR       No. Vamos a ERROR
        <resto del programa para paridad correcta>
        *   *   *
        BRA     OTRA_COSA   Continúa el programa
ERROR   <iniciar las acciones adecuadas para recuperar el error de paridad>
        *   *   *
```

Notas al programa: Algunas de las instrucciones de salto tienen la opción cortos (de −128 a 127 bytes). Esta opción ahorra una palabra en el ensamblado.

Instrucciones de desplazamiento aritmético

La figura 6.12 ilustra el desplazamiento aritmético. Como puede verse, los desplazamientos aritméticos son muy similares a los desplazamientos lógicos de la sección anterior. De hecho, emplean los mismos formatos para las cadenas de bits origen y destino, y los desplazamientos a derecha e izquierda se producen de acuerdo al tamaño del dato y al contador de desplazamiento (ASL equivale a desplazamiento aritmético a la izquierda y ASR a desplazamiento aritmético a la derecha). La diferencia es que cuando se desplazan aritméticamente números con signo, el procesador protege el bit

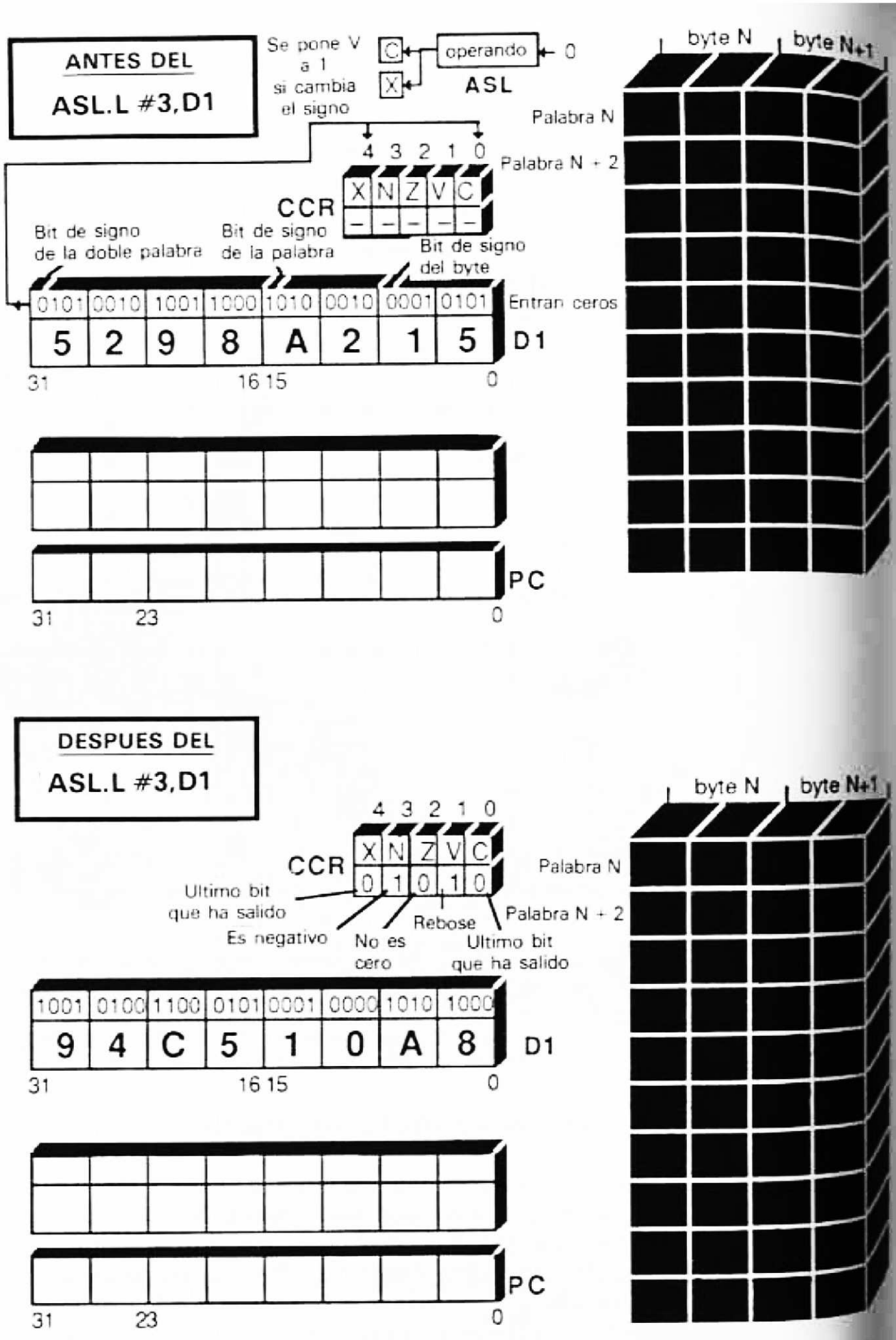


Figura 6.12
ASL.L #3,D1

más significativo contra posibles cambios que podrían llevar a resultados erróneos. Por ejemplo, si se desea dividir 4 entre 2, emplear el desplazamiento lógico llevaría a resultados erróneos. Veamos por qué:

En complemento a 2, −4 = **11111100** (en formato byte)
Desplazamiento lógico a la derecha de −4 = **01111110** = +126
El desplazamiento aritmético correcto a la derecha sería −2 = **11111110**

La situación es peor cuando se intenta dividir −4 entre 4, empleando desplazamiento lógico a la derecha con un contador de desplazamiento de 2.

El problema es que cuando un LSR introduce un cero a la izquierda (el valor más significativo) de cualquier número negativo, no sólo altera el bit de signo, también lo mueve a la posición 6 y la respuesta resultante carece de sentido en lo que concierne a la aritmética de signo. El LSR funciona bien con números de signo positivo, pero es claro que se necesita un desplazamiento a la derecha que funcione bien con todos los números con signo.

El ASR consigue esto introduciendo un 0 o un 1, dependiendo del signo que hay que desplazar. Tomemos el ejemplo de "−4 entre 2" de nuevo:

En complemento a 2, −4 = **11111100** (en formato byte)
El desplazamiento aritmético a la derecha de −4 = **11111110** = −2, que es correcto

Dado que −4 tiene el bit de signo a 1, ASR introduce un 1 a la izquierda, preservando el signo del dividendo.

Similarmente, multiplicar por 2 empleando desplazamientos a la izquierda puede llevar a resultados erróneos con números con signo:

En complemento a 2, +72 = **01001000** (en formato byte)
El desplazamiento a la izquierda de +72 = **10010000** = −122 con signo o +144 sin signo

Aquí la respuesta es correcta en aritmética sin signo, pero incorrecta con signo. El problema no se debe al desplazamiento en sí, sino al hecho de que +144 excede la capacidad de un byte de 8 bits (de −126 a +127). Hay que vivir con este hecho. Como se vio en la instrucción ADD, lo mejor que se puede hacer es vigilar el indicador de rebose V en el CCR: éste es nuestra defensa contra la aritmética de signo. Podemos recordar que el LSL siempre pone a cero el indicador V en el CCR. Luego el LSL es peligroso si se desea emplear números con signo. La solución es emplear el ASL cuando se desea emplear números con signo, porque el ASL afecta al indicador V. ASL introduce ceros desde la derecha como el ASL, pero, si se detecta un cambio de signo en cualquier momento durante el desplazamiento, se pondrá a 1 el indicador V. Si no ocurren cambios de signo, el indicador V

se pondrá a 0. Nótese que en desplazamientos múltiples el indicador V puede cambiar varias veces para terminar con el mismo valor con el que empezara. Sin embargo, si V se pone a 1, permanecerá en este valor hasta que termine el desplazamiento. Como en el resto de la aritmética con signo, es responsabilidad del programador comprobar el estado del indicador V: un valor V = 1 significa peligro.

Resumamos las diferencias entre los desplazamientos lógicos y aritméticos.

Desplazamientos lógicos y aritméticos: Diferencias

El LSR desplaza hacia la derecha, insertando ceros desde la izquierda.

El ASR desplaza hacia la derecha, copiando el bit de signo desde la izquierda.

Tanto el LSL como el ASL desplazan hacia la izquierda, insertando ceros desde la derecha; pero el LSL pone a cero el indicador de rebose en el CCR.

El ASL pone el indicador V a 1 si ocurre cualquier cambio de signo durante el desplazamiento.

Veamos ahora las semejanzas entre los desplazamientos lógicos y aritméticos.

Desplazamientos lógicos y aritméticos: Semejanzas

Los desplazamientos lógicos y aritméticos comparten los modos de direccionamiento.

```
ASL.z   #<d3>,Dn    Cuenta de desplazamiento inmediato entre 1 y 8
ASL.z   Dm,Dn       Cuenta de desplazamiento Dm mod 64
ASL.W   <amea>      Cuenta de desplazamiento = 1

ASR.z   #<d3>,Dn
ASR.z   Dm,Dn
ASR.W   <amea>
```

Tanto los desplazamientos lógicos como aritméticos copian el bit que sale desplazado en los indicadores C y X, y ambos afectan a los indicadores N y Z de la misma manera. Las figuras 6.12 y 6.13 muestran dos típicos desplazamientos aritméticos.

Desplazamientos aritméticos y el CCR

Resumamos los cambios en el CCR:

Indicador X: Se pone al valor del último bit que salió de la palabra destino. No resulta afectado si el contador de desplazamiento es 0, es decir, si no se produce desplazamiento alguno.

Indicador N: Se pone a 1 si el destino resulta negativo tras el desplazamiento.

Indicador V: Se pone a 1 si ocurre algún cambio en el bit de signo en cualquier momento durante el desplazamiento.

Indicador C: Resulta afectado del mismo modo que el indicador X, pero se pone a cero si el contador de desplazamiento es 0, es decir, si no se produce desplazamiento alguno.

APLICACION PRACTICA

Problema: Calcular la media aritmética de dos números con signo D0 y D1 al entero más próximo y almacenar la respuesta en la palabra de orden más bajo de D3. Señalar un error si se excede el rango de los números con signo.

Preliminares: La media aritmética de un par de números, a veces conocida como promedio, se obtiene sumándolos y dividiendo por 2. La media está exactamente a medio camino entre ambos números. Una aplicación muy común se encuentra en los procesos de búsqueda binaria en ficheros ordenados. Se localiza el registro buscado dividiendo el fichero en dos partes iguales. Comparar el objeto de la búsqueda con el registro intermedio de cada una de las partes indica cuál de ellas lo contiene. Entonces se divide esa mitad del fichero deseado y se repite el proceso hasta encontrar el registro deseado. En cada paso de la búsqueda es necesario calcular la media para hallar el registro intermedio.

Datos: Dos números de 16 bits en D0 y D1.

Ejemplos:

D0 = 2979 = $0BA3 D1 = 4261 = $10A5
D0 + D1 = 7240 = $1C48 (no hay rebose)
D3 = 1/2*7240 = 3620 = $0E24
D0 = −3 = $FFFD D1 = −5 = $FFFB
D0 + D1 = −8 = $FFF8 (no hay rebose)
D3 = 1/2*(−8) = −4 = $FFF4
D0 = 43981 = $ABCD D1 = 26341 = $66E5
D0 + D1 = 70322 = $112B2

Solución: Programa 6.3.

```
        MOVE.W  D0,D3          La palabra D0 está ahora en D3
        ADD.W   D1,D3          La palabra en D3 es D0 + D1
        BVS     ERROR          Saltar a ERROR si el indicador V se ha puesto a 1
        ASR.W   #1,D3          Dividir D3 entre 2. Ignorar el resto. D3 contiene ahora
                               la media de D0 y D3 (el entero más próximo)
        <resto del programa>
        *       *      *
        BRA     OTRA_COSA      Continúa el programa
ERROR   <señalar un error e iniciar las acciones de recuperación apropiadas>
        *       *      *
        FIN
```

Notas: Si D0 + D1 es par, la media obtenida en D3 será correcta; pero si D0 + D1 es impar, la media en D3 será 0,5 menor que la real. Nuestro problema simplificado pedia el entero más próximo, de modo que se ha ignorado cualquier resto al dividir por 2. Hay una forma fácil de diferenciar estos dos casos. Cuando se efectúa un ASR sobre un número impar, el bit que sale es un 1. Puesto que este bit se copia en los indicadores X y C del CCR, se puede comprobar fácilmente este hecho mediante un BCS o un BCC y emprender la acción adecuada.

La tabla 6.2 da un resumen de todas las instrucciones de desplazamiento y da como resulta afectado el CCR.

TABLA 6.2

Resumen de las instrucciones de desplazamiento

Instrucción	*Operando*	*Efectos sobre el CCR*
ASL.L/W/B ASR.L/W/B	Dm,Dn o #<d3>,Dn	X*N*Z*V*C*
ASL.W ASR.W	<amea>	X*N*Z*V*C*
LSL.L/W/B LSR.L/W/B	Dm,Dn o #<d3>,Dn	X*N*Z*V0C*
LSL.W LSR.W	<amea>	X*N*Z*V0C*

<amea> = Direccionamiento por direcciones alterables efectivas: (An), (An)+, −(An), d(An), d(An,Xi), Abs.W, Abs.L.

#<d3> = 3 bits que se emplean como datos inmediatos. Permiten un contador de desplazamiento de 1 a 8.

Notación para el CCR: _ indica sin cambios, * indica cambios según las reglas del CCR, 0 indica que el indicador siempre se pone a 0.

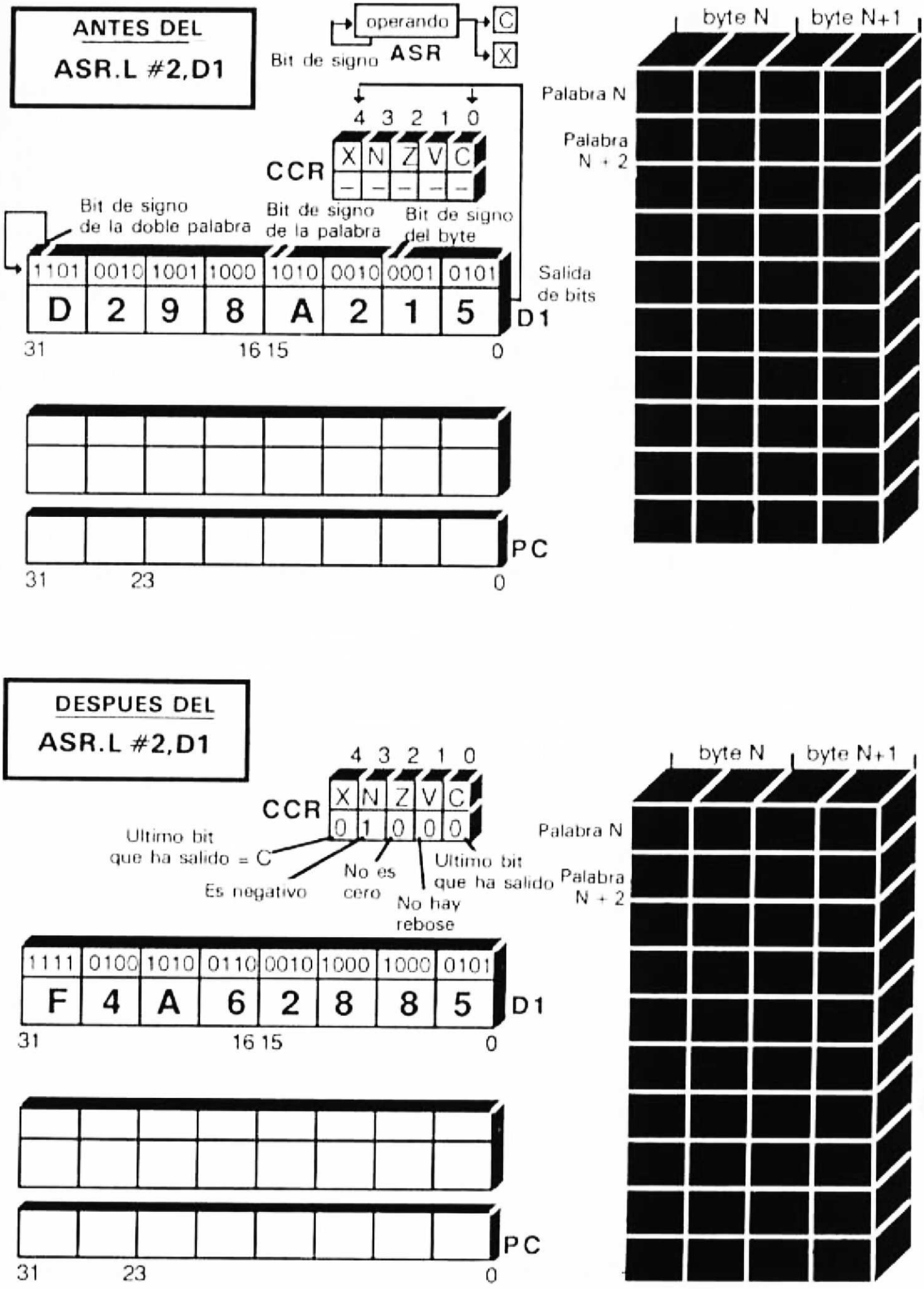


Figura 6.13
ASR.L #2,D1

Rotaciones

Rotar los bits en un registro es muy parecido al desplazamiento lógico descrito más arriba, excepto que los bits que salen por uno cualquiera de los extremos entran por el otro. Como la palabra "rotación" sugiere, uno puede imaginarse las cadenas de bits moviéndose en el sentido de las agujas del reloj (rotación a la izquierda) o en sentido contrario a éstas (rotación a la derecha). Como en el caso de los desplazamientos, se puede especificar cuántas veces se rotan los bits empleando un dato inmediato o un registro como contador de desplazamiento. Los formatos de la instrucción de rotación para la fuente y el destino son idénticos a aquellos empleados para los desplazamientos. La gran diferencia es lo que sucede con los bits a medida que rotan. La tabla 6.3 muestra las cuatro variantes para las instrucciones de rotación.

ROR	Rotación a la derecha
ROL	Rotación a la izquierda
ROXR	Rotación a la derecha empleando el indicador X
ROXL	Rotación a la izquierda empleando el indicador X

TABLA 6.3

Resumen de las instrucciones de rotación

Instrucción	*Operando*	*Efectos sobre el CCR*
ROL.L/W/B ROR.L/W/B	Dm,Dn o #<d3>,Dn	X_N*Z*V0C*
ROL.W ROR.W	<amea>	X_N*Z*V0C*
ROXL.L/W/B ROXR.L/W/B	Dm,Dn o #<d3>,Dn	X*N*Z*V0C*
ROXL.W ROXR.W	<amea>	X*N*Z*V0C*

<amea> = Direccionamiento por direcciones alterables efectivas: (An), (An)+, −(An), d(An), d(An,Xi), Abs.W, Abs.L.

#<d3> = 3 bits que se emplean como datos inmediatos. Permiten un contador de desplazamiento de 1 a 8.

Notación para el CCR: _ indica sin cambios, * indica cambios según las reglas del CCR, 0 indica que el indicador siempre se pone a 0.

Estas instrucciones aceptan los tres formatos dados para los desplazamientos, que son:

ROR.z	Dm,Dn	Rotar Dn{.z} a la derecha tantas veces como indique (Dm mod 64)
ROR.z	#<d3>,Dn	Rotar Dn{.z} a la derecha d3 veces (1 – 8)
ROR.z	<dem>	Rotar la memoria{.z} a la derecha una vez solamente

La figura 6.14 muestra cómo el bit desplazado siempre se copia en el indicador C del CCR. En las variantes ROR/ROL los bits desplazados entran directamente por el otro extremo y el indicador X permanece inalterado.

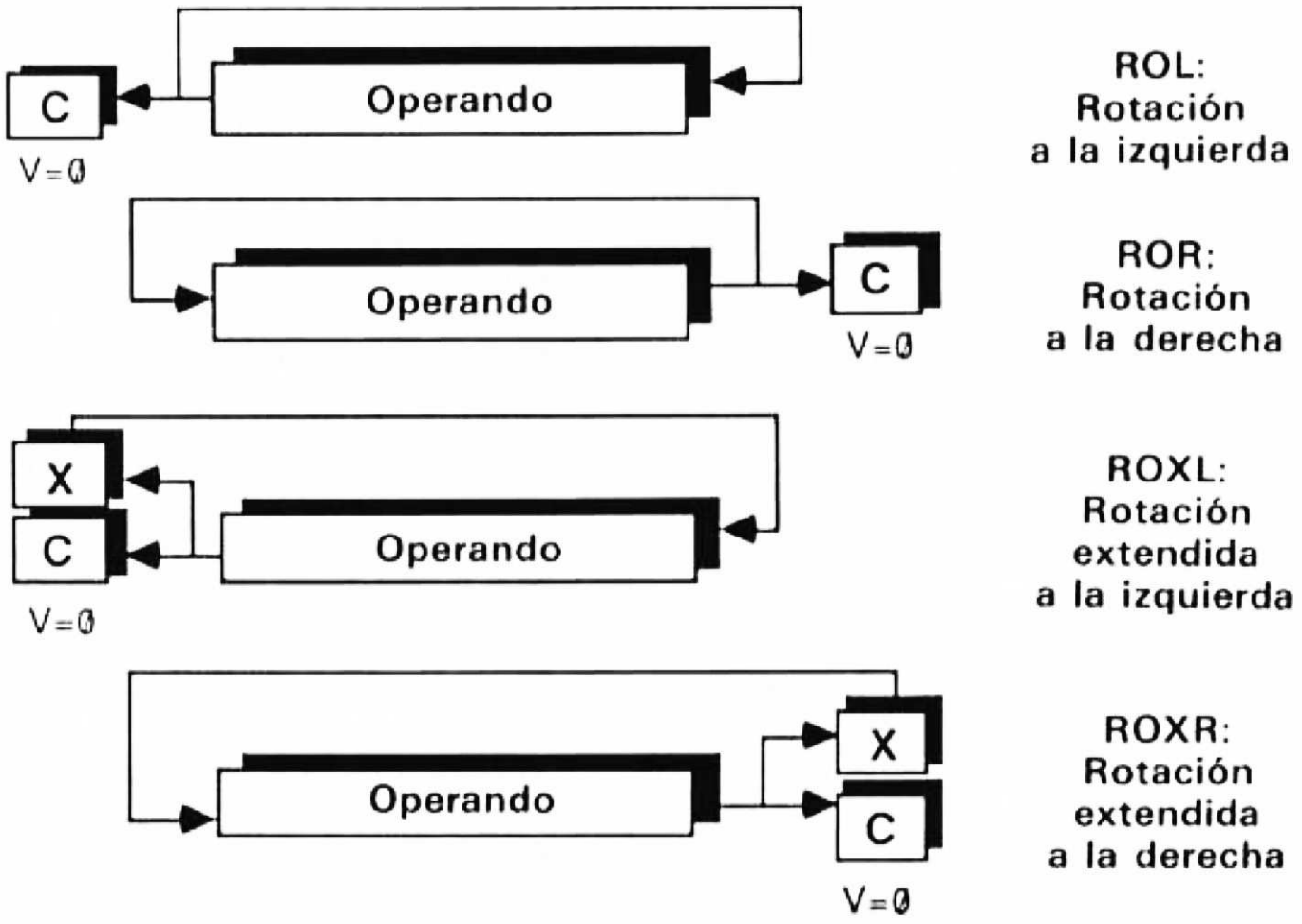


Figura 6.14
Esquemas generales para las instrucciones de rotación

En las variantes ROXR/ROXL, el bit desplazado se copia tanto en el indicador C como en el X. El antiguo valor del indicador X se introduce en el registro. El indicador está, como de costumbre, jugando el papel de bit adicional del registro, de modo que:

ROXR/ROXL.L	rotan 33 bits (una doble palabra y X)
ROXR/ROXL.W	rotan 16 bits (una palabra y X)
ROXR/ROXL.L	rotan 9 bits (un byte y X)

Todas las rotaciones, como los desplazamientos lógicos, ponen a cero el indicador V, de modo que no se indican los posibles errores en aritmética con signo.

Aquí se muestra una aplicación para iluminar la naturaleza de las rotaciones.

```
* Programa 6.4: Sumar los 4 bytes con signo de una doble palabra
* D0 contiene 4 bytes representando cada uno un número con signo
* Byte 1 = bits 0-7 (byte menos significativo de la palabra menos significativa)
* Byte 2 = bits 8-15 (byte más significativo de la palabra menos significativa)
* Byte 3 = bits 16-23 (byte menos significativo de la palabra más significativa)
* Byte 4 = bits 17-32 (byte más significativo de la palabra más significativa)
* La suma de los cuatro números se almacena en D1
* Se emplea D3 como registro temporal para cálculos
* Se mantiene los valores de D0 y D3
        MOVE.L  D3,-(SP)    Salvamos el registro temporal D3
        CLR.L   D3          Ponemos a 0 la doble palabra D3
        CLR.L   D1          Ponemos a 0 la doble palabra D1
        MOVE.B  D0,D1       Primer byte a D1
        ROR.L   #8,D0       Movemos el byte 2 al byte menos significativo de la palabra
                            menos significativa. El byte 3 pasa al byte más significativo
                            de la palabra menos significativa. El byte 4 pasa al byte
                            menos significativo de la palabra más significativa. El byte 1
                            pasa al byte más significativo de la palabra más significativa
        MOVE.B  D0,D3       Segundo byte a D3
        ADD.W   D3,D1       Sumamos los dos primeros bytes y lo almacenamos en D1
        ROR.L   #8,D0       Movemos el byte 3 al byte menos significativo de la palabra
                            menos significativa. El byte 4 pasa al byte más significativo
                            de la palabra menos significativa. El byte 1 pasa al byte
                            menos significativo de la palabra más significativa. El byte 2
                            pasa al byte más significativo de la palabra más significativa
        MOVE.B  D0,D3       Tercer byte a D3
        ADD.W   D3,D1       Sumamos el byte 3 a D1
        ROR.L   #8,D0       Movemos el byte 4 al byte menos significativo de la palabra
                            menos significativa. El byte 1 pasa al byte más significativo
                            de la palabra menos significativa. El byte 2 pasa al byte
                            menos significativo de la palabra más significativa. El byte 3
                            pasa al byte más significativo de la palabra más significativa
        MOVE.B  D0,D3       Cuarto byte a D3
        ADD.W   D3,D1       Sumamos el byte 4 a D1
        ROR.L   #8,D0       Movemos el byte 1 al byte menos significativo de la palabra
                            menos significativa. El byte 2 pasa al byte más significativo
                            de la palabra menos significativa. El byte 3 pasa al byte
                            menos significativo de la palabra más significativa. El byte 4
                            pasa al byte más significativo de la palabra más significativa
        MOVE.L  (SP)+,D3    Restauramos el valor de D3 copiándolo desde la pila
* D0, tras 32 rotaciones, ha vuelto a su valor original
* D1 contiene la suma de los bytes 1-4
```

Hemos empleado D3 y ADD.W para evitar los peligros del rebose. Sumar cuatro bytes con signo, no puede exceder el rango de los 16 bits con signo; pero puede sobrepasar el rango de asignado a los números de 8 bits con signo, de modo que no se puede emplear

```
ADDB.  D0,D1
```

inmediatamente después del ROR a menos que comprobemos si existe rebose.

Hay, desde luego, muchas formas de separar los bytes o las palabras en una doble palabra. SWAP, por ejemplo, se usa conjuntamente con una rotación (veremos más acerca del SWAP más tarde en este capitulo). Vamos a repetir el programa 6.4 empleando SWAP para sumar sólo los bytes 1, 3 y 4.

```
* Programa 6.5: Sumar los bytes 1, 3 y 4 con signo de una doble palabra
* D0 contiene 4 bytes representando cada uno un número con signo
* Byte 1 = bits 0-7 (byte menos significativo de la palabra menos significativa)
* Byte 2 = bits 8-15 (byte más significativo de la palabra menos significativa)
* Byte 3 = bits 16-23 (byte menos significativo de la palabra más significativa)
* Byte 4 = bits 17-32 (byte más significativo de la palabra más significativa)
* La suma de los bytes 1, 3 y 4 se almacena en D1
* Se emplea D3 como registro temporal para cálculos
* Se mantiene los valores de D0 y D3

MOVEM.L  D0/D3,-(SP)   Almacenar estos valores en la pila
CLR.L    D3            Poner a 0 la doble palabra D3
CLR.L    D1            Poner a 0 la doble palabra D1
MOVE.B   D0,D1         Byte 1 a D1
SWAP     D0            Invertir las palabras en D0. El byte 3 es ahora el byte
                       menos significativo de la palabra menos significativa
                       El byte 4 es el byte más significativo de la palabra
                       menos significativa. (El byte 2 es el byte más
                       significativo de la palabra más significativa.
                       El byte 4 es el byte más significativo
                       de la palabra más significativa)
MOVE.B   D0,D3         Byte 3 a D3
ADD.W    D3,D1         Sumar bytes 1 y 3 y almacenar el resultado en D1
ROR.L    #8,D0         Movemos el byte 4 al byte menos significativo
                       de la palabra menos significativa. (El byte 1 pasa
                       al byte más significativo de la palabra menos
                       significativa. El byte 2 pasa al byte menos significativo
                       de la palabra más significativa. El byte 3
                       pasa al byte más significativo de la palabra
                       más significativa)
MOVE.B   D0,D3         Byte 4 a D3
ADD.W    D3,D1         Sumar byte 4 a D1
MOVEM.L  (SP)+,D0/D3   Restaurar los valores de D0,D3 según la pila
* D1 contiene la suma de los bytes 1, 3 y 4
```

Asignación y comprobación de valores de bits

En el capitulo 4 encontramos la instrucción TST.z, que comprueba un operando completo (L, W, o B) para poner a 0 ó 1 los indicadores del

CCR, según proceda. En el próximo grupo de instrucciones redefiniremos esta idea para incluir la posibilidad de asignar o comprobar los valores de un bit en concreto en una amplia variedad de situaciones.

BTST: Comprobar un bit

BTST permite comprobar cualquier bit dentro de un registro de datos, o cualquier bit en un byte de la memoria. El resultado de esta comprobación, como en el caso de un TST, se refleja en el indicador Z del CCR:

Si el bit comprobado = 0, se pone el indicador Z a 1
Si el bit comprobado = 1, se pone el indicador Z a 0

El resto de los indicadores del CCR permanecen inalterados.

TST, como se vio en el capítulo 4, comprueba si todo un byte, palabra o doble palabra es cero, mientras que BTST comprueba un bit individualmente. BTST tiene los siguientes formatos:

BTST.L Dm,Dn	Comprueba el bit (Dm mod 32) de Dn
BTST.L #<d5>,Dn	Comprueba el bit d5 de Dn
BTST.B Dm,<dem>	Comprueba el bit (Dm mod 8) byte <dem>
BTST.B #<d3>,<dem>	Comprueba el bit d3 del byte <dem>

Donde <d5> representa un número de 5 bits, es decir, de 0 a 31, y <de> un número de 3 bits: de 0 a 7.

Nótese que <dea> excluye aquí el modo inmediato como una dirección válida. Puesto que el destino no se altera por un BTST, los destinos direccionados en modo relativo sí se permiten.

El operando fuente, que indica la posición del bit a comprobar, puede ser un número en un registro de datos o una constante inmediata. Recuérdese que el bit en la posición cero es el primero y menos significativo en todos los casos.

El rango en que puede moverse el valor de la posición del bit a comprobar es, obviamente, de 0 a 31 para los registros de datos y de 0 a 7 para los bytes de la memoria. Si se intenta comprobar una posición de un bit fuera de estos rangos, el procesador simplemente la reducirá en módulo 32 o en módulo 8, como se ha indicado más arriba.

Puesto que sólo los bytes de la memoria pueden comprobarse mediante BTST, es necesario en algunas ocasiones moverse desde ésta a un registro de datos para comprobaciones más elaboradas.

Los códigos de tamaños están implicados en el formato, de modo que son opcionales en la mayoría de los ensambladores. Aquí se usarán para clarificar lo que se está haciendo.

Aquí se muestran dos ejemplos típicos del uso de BTST.

```
* Programa 6.6: Comprobar si un número es par o impar empleando BTST
* D3 contiene un número sin signo
* Si es par dejarlo tal y como está
* Si es impar sumarlo uno para transformarlo en par

        BTST.L  #0,D3   ¿Es el bit 0 de D3 = 0?
        BEQ     PAR     Si es así saltar a PAR
        ADDQ.L  #1,D3   Si no es así es impar, sumarle uno para hacerlo par
        BCS     ERROR   ¿Demasiado grande para ser un número con signo
                        de 32 bits? Se ha producido un acarreo
PAR     <resto del programa>
        *   *   *
ERROR   <tratar el error de rango>
```

El programa 6.6 descansa en el hecho de que un número par tiene su bit menos significativo a 0. Transformar un número de impar a par es un truco útil para redondear las direcciones que usa el M68000 a la palabra más próxima.

```
* Programa 6.7: Comprobar el estatus de un empleado, empleando BTST
* A0 apunta al registro de un empleado en la memoria
* La primera palabra contiene la identificación del empleado
* El byte de orden más bajo de la segunda palabra es el byte de estatus del empleado
* Bit 0 = 0 para varones, 1 para mujeres
* Bit 1 = 0 para jornada a tiempo completo (TC),
* 1 para jornada a tiempo parcial (TP)
* Bit 2 = 0 para personal de oficina (PO), 1 para directivos (DI)
* Incrementar el contador D6 en 1 si el empleado es mujer/TC/PO
* Cuando se entra en este segmento del programa, D6 contiene el subtotal
* de tales empleados

        BTST.B  #0,2(A0)    Comprobar el bit de sexo
        BEQ     IGNORAR     Ignorar a los varones, es decir, bit 0 = 0
        BTST.B  #1,2(A0)    Comprobar el bit de TP/TC
        BNE     IGNORAR     Ignorar a los empleados a tiempo parcial,
                            es decir, bit 1 = 1
        BTST.B  #2,2(A0)    Comprobar el bit de DI/PO
        BNE     IGNORAR     Ignorar directivos, es decir, bit 2 = 1
        ADDQ.L  #1,D6       Añadir 1 al conjunto de empleados mujer/TC/PO
IGNORAR <resto del programa>
* Esto formaría parte de un programa para contar el número de empleados
* en cada categoría
```

Aquí estamos comprobando el byte de la memoria en la localización A0 + 2; por tanto, el operando destino es 2(A0). Para comprobaciones repetidas sobre 2(A0), probablemente encontraríamos más rápido mover 2(A0) a un registro de datos, puesto que nos ahorraríamos cálculos de direcciones efectivas y búsquedas en la memoria.

Comprobar y cambiar un bit

Hay tres variantes de BTST, que son:

BCLR : comprobar un bit y ponerlo a 0
BSET : comprobar un bit y ponerlo a 1
BCHG: comprobar un bit y cambiarlo

Estas variantes no sólo comprueban el valor de un bit alterando el estatus del indicador Z como BTST, sino que después proceden a cambiar el valor del bit comprobado según indique el nemónico.

Los formatos difieren ligeramente del de BTST, puesto que sólo se permiten operandos alterables, lo que es natural si se considera que BCLR/BSET/BCHG alteran realmente el destino.

```
BCLR.L  Dm.Dn          Comprueba el bit (Dm mod 32) de Dn
BCLR.L  #<d6>,Dn       Comprueba el bit (d6 mod 32) de Dn
BCLR.B  Dm,<dem>       Comprueba el bit (Dm mod 8) del byte <dem>
BCLR.B  #<d3>,<dem>    Comprueba el bit (Dm mod 8) del byte <dem>
```

Una vez que se ha puesto el indicador Z al valor correspondiente, **BCLR** pone el bit destino indicado a 0.

```
BSET.L  Dm.Dn          Comprueba el bit (Dm mod 32) de Dn
BSET.L  #<d6>,Dn       Comprueba el bit (d6 mod 32) de Dn
BSET.B  Dm,<dem>       Comprueba el bit (Dm mod 8) del byte <dem>
BSET.B  #<d3>,<dem>    Comprueba el bit (Dm mod 8) del byte <dem>
```

Una vez que se ha puesto el indicador Z al valor correspondiente, **BSET** pone el bit destino indicado a 1.

```
BCHG.L  Dm.Dn          Comprueba el bit (Dm mod 32) de Dn
BCHG.L  #<d6>,Dn       Comprueba el bit (d6 mod 32) de Dn
BCHG.B  Dm,<dem>       Comprueba el bit (Dm mod 8) del byte <dem>
BCHG.B  #<d3>,<dem>    Comprueba el bit (Dm mod 8) del byte <dem>
```

Una vez que se ha puesto el indicador Z al valor correspondiente, **BCHG** invierte el bit indicado: 0 → 1 ó 1 → 0.

Las tres instrucciones para probar y asignar valores a ciertos bits suelen emplearse únicamente para asignar valores a éstos, ignorándose los aspectos de comprobación. Empleando estos datos en el programa 6.7, podemos rebajar la categoría de un empleado cambiando su byte de estatus.

```
* Programa 6.7: Cambiar el estatus de un empleado, empleando BCLR
* A0 apunta al registro del empleado en la memoria
```

```
* La primera palabra contiene la identificación del empleado
* El byte de orden más bajo de la segunda palabra es el byte de estatus del empleado
* Bit 0 = 0 para varones, 1 para mujeres
* Bit 1 = 0 para jornada a tiempo completo (TC), 1 para jornada a tiempo parcial (TP)
* Bit 2 = 0 para directivos (DI), 1 para el resto del personal
* Rebajamos la categoria del empleado poniendo el bit 2 a 0

          BCLR.B  #2(A0)    Ponemos el bit 2 de 2(A0) a 0
          BEQ     ERROR     El empleado ya estaba en la categoria
          <resto del programa>
          *    *    *
ERROR     <comprobar los registros>

* BCLR comprueba el bit 2 antes de ponerlo a 0.
* BEQ producirá un salto si el bit 2 era ya 0, revelando un posible error
* en el conjunto de registros con el que se está trabajando
```

BCHG es útil para controlar la actividad durante los bucles, actuando como un balancín o conmutador. Por ejemplo:

```
* Programa 6.8A: Cambiando los trabajos mediante BSET y BCHG

          BSET.L  #0,D2     Comenzamos poniendo el bit 0 de D2 a 1
BUCLE     BCHG.L  #0,D2     Cambiar 0 → 1 y 1 → 0
          BEQ     TAREA0    Realizar la TAREA0 para números pares
          <Aquí comienza la TAREA1>
          BRA     BUCLE
TAREA0    <Aquí comienza la TAREA0>
          BRA     BUCLE

* Cada vez que se alcanza el comienzo del BUCLE comprobamos el valor del bit 0 de D2
* y se pone a 0 ó 1 el indicador Z. Después se cambia el valor del bit de 0 a 1
* o de 1 a 0. Entonces se comprueba el valor de Z para realizar la TAREA0
* o la TAREA1. De este modo se alternan las tareas a realizar. Para evitar un ciclo
* infinito se supone que una de las dos tareas contiene algún test para salir del BUCLE
```

Scc: Poner a uno condicionalmente

Scc representa un conjunto de códigos de un solo operando, que debe ser un byte, y que tiene el formato:

```
Scc{.B}    <adea>
```

El byte en <adea> se pone a $FF (todo unos) si la condición cc es cierta y se pone a $00 (todo ceros) si la condición cc es falsa. Dado que Scc altera el destino, sólo se permiten los modos de direccionamiento de tipo <adea>.

Scc tiene 16 posibles variantes correspondientes a los diferentes mnemónicos de condición representados por cc. Cada condición viene determinada

TABLA 6.4

Códigos de condición para Bcc, Dbcc y Scc

Mnemónico para el cc	*Condición*	*Fórmula booleana*	*Válido en aritmética*
CC	Acarreo a cero	~C	Sin signo
CS	Acarreo a 1	C	Sin signo
EQ	Es igual a	Z	Todas
F	Falso	0	Todas*
GE	Mayor o igual que	(N ^ V) + (~N ^ ~V)	Con signo
GT	Mayor que	(N ^ V ^ ~Z) + (~N ^ ~V ^ ~Z)	Con signo
HI	Mayor que	~C ^ ~Z	Sin signo
LE	Menor o igual que	Z + (N ^ ~V) + (~N ^ V)	Con signo
LS	Menor o igual que	C + Z	Sin signo
LT	Menor que	(N ^ ~V) + (~N ^ V)	Con signo
MI	Es negativo	N	Con signo
NE	No es igual a	~Z	Todas
PL	Es positivo	~N	Con signo
T	Verdadero	1	Todas*
VC	Se ha producido un rebose	~V	Con signo
VS	No se ha producido un rebose	V	Con signo

Leyenda: ~ = NO lógico; + = O lógico; ^ = Y lógico.
* F y T no se utilizan con Bcc.

por el estado de los indicadores en el CCR en el momento en que se efectúa la comprobación. Ya se han visto instrucciones de este tipo en el capítulo 4, en la sección en que se habló de las instrucciones Bcc (saltos condicionales). La tabla 6.4 muestra la lista completa de los códigos cc empleados con Scc, Bcc y DBcc (instrucción esta última de la que se hablará más adelante).

La función esencial de un Scc es la de almacenar el resultado de una comprobación efectuada sobre el CCR, de modo que pueda usarse el resultado obtenido más tarde, una vez que el CCR haya cambiado. Ya se ha visto que la mayoría de las instrucciones altera el CCR, y esto puede ser una molestia si se desea diferir una acción condicional.

Funcionamiento de las instrucciones tipo cc

Las diferentes condiciones representadas por un código cc varían desde comprobar el estado de un único indicador del CCR hasta evaluar complicadas expresiones booleanas que involucran varios de estos indicadores.

Las condiciones que afectan a un solo indicador se explicaron ya cuando se habló del Bcc. Estas ocho condiciones básicas dependen sólo del estado de los indicadores N, Z, C, o X. Estos indicadores pueden considerarse como variables booleanas que toman el valor 1 para significar verdad y el valor 0 para indicar lo contrario. Se pueden combinar, como se muestra, para indicar condiciones más complicadas. La tabla 6.4 muestra los cálculos lógicos efectuados por el M68000 para determinar la veracidad o falsedad de cuestiones típicas referentes a las relaciones entre dos números: mayor, menor, igual, etc.

Disección de una condición cc que afecta a varios indicadores

Tomemos como ejemplo la condición HI (mayor que), que se lista en la tabla como una condición sin signo. Si queremos comparar dos números sin signo en D0 y D1, podemos escribir:

```
SUB.L   D0,D1
```

Esto restará D0 de D1, cambiará los valores de los indicadores del CCR y reemplazará D1 por la diferencia entre D1 y D0. O, como se detallará más adelante, podemos escribir:

```
CMP.L   D0,D1
```

Para responder a la pregunta: "¿Es el número sin signo en D1 mayor que el número sin signo en D0?", tenemos que mirar los indicadores C y Z después de un SUB o un CMP. Si Z = 1, tenemos una diferencia nula, de modo que D0 = D1 y, por tanto, D0 no es mayor que D1. Por otra parte, si C = 1, tenemos un acarreo negativo, implicando que D1 es menor que D0; de nuevo, esto indica que D0 no es mayor que D1. Por tanto, la condición "ser mayor que" sólo se verifica si (C = 0 y Z = 0), expresión que nos lleva a la fórmula booleana: HI = −C&−Z, que se lee NO-C y NO-Z.

El efecto de

```
SHI  D5    Poner a 1 el registro D5 si se verifica la condición "ser mayor que"
```

por ejemplo, es:

Si (C = 0 y Z = 0), poner el byte de orden más bajo de D5 a $FF.
En otro caso poner el byte de orden más bajo de D5 a $00.

De este modo almacenamos el resultado de la comprobación "ser mayor que" en D5 para su uso posterior. Similarmente:

```
BHI  <etiqueta>   Saltar a la <etiqueta> indicada si se verifica
                  la condición "ser mayor que"
```

indica que el procesador salte a la <etiqueta> sólo si (C = 0 y Z = 0).

El procesador realiza estas pruebas independientemente de los anteriores pasos de programa. Es, de hecho, el programador quien tiene que dotar de significado a la condición HI ("ser mayor que"), empleando previamente un paso CMP o SUB que emplee los números que se desea comparar.

Cada una de las otras condiciones puede ser analizada de la misma forma estudiando los diferentes indicadores tras un CMP o un SUB que emplee los números que se están comparando.

Comparaciones cc con y sin signo

Está claro que las cuestiones del tipo "¿es mayor que?" o "¿es menor que?" sólo pueden resolverse si se sabe qué modo de numeración se está empleando: ¿con signo o sin signo? ¿Es "10000000" mayor que "00000111"? La respuesta es sí para números con signo y no para números sin signo.

Por otra parte, la cuestión "¿son iguales?" o "¿es cero?" puede contestarse con independencia del modo de numeración. La columna final de la tabla 6.4 indica qué condiciones se aplican a cada modo. Notar que Motorola ha elegido la forma "higher/lower/same" (más/menos/lo mismo) para las comparaciones sin signo y la forma "greater/less" (mayor/menor) para comparaciones con signo.

Se pueden comprobar las fórmulas booleanas restando varios números con y sin signo y comprobando los indicadores en el CCR. Entonces se sustituyen los valores de los indicadores 0 ó 1 en las fórmulas booleanas, aplicando las siguientes reglas:

$0 + 0 = 0$	Falso O Falso = Falso
$1 + 0 = 1$	Verdadero O Falso = Verdadero
$0 \wedge 0 = 0$	Falso Y Falso = Falso
$0 \wedge 1 = 0$	Falso Y Verdadero = Falso
$1 \wedge 1 = 1$	Verdadero Y Verdadero = Verdadero
$\sim 0 = 1$	NO Falso = Verdadero
$\sim 1 = 0$	NO Verdadero = Falso
$\sim(A + B) = (\sim A \wedge \sim B)$	NO (A O B) = (NO A) Y (NO B)
$\sim(A \wedge B) = (\sim A + \sim B)$	NO (A Y B) = (NO A) O (NO B)

Cada una de las condiciones se reduce a 1 (verdadero) o 0 (falso) y el Scc tomará nota de este hecho en cualquier registro o localización de la memoria previamente elegida.

Las reglas expresadas en álgebra booleana merecen un estudio más detallado, porque equivalen a capítulos enteros acerca del significado del acarreo y el rebose. Una vez que se ha convencido de que todas las reglas funcionan, relájese y deje que el M68000 las evalúe por usted, ¡lo hace realmente bien!

T y SF

Dos de las posibilidades para cc actúan de manera incondicional:

ST <adea>	Siempre pone <adea> a $FF (verdadero)
ST <adea>	Siempre pone <adea> a $00 (falso)

Nótese que no se emplea ni la variante T ni la variante F con Bcc; empleamos BRA para indicar una bifurcación incondicional, mientras que BF (no bifurcar) es más una abstracción de la programación estructurada que una instrucción real.

TAS: Probar y poner a 1 de manera indivisible

La última instrucción dentro del grupo de instrucciones de manipulación de bits es la instrucción TAS (probar y poner a 1 el operando), que esconde un sucio truco bajo su simple apariencia:

TAS{.B} <adea>

Esta linea primero prueba el byte en <adea> y asigna los valores correspondientes a los indicadores N y Z del CCR (Z = 1 si el byte es 0, N = 1 si el bit de signo es 1). Finalmente, TAS pone incondicionalmente el séptimo bit del byte destino a uno, transformándolo en negativo.

El raro truco que esta instrucción emplea es que la operación efectuada por TAS es indivisible; con esto queremos señalar que TAS emplea un ciclo de lectura/escritura especial, que no puede ser interrumpido, y ningún otro programa, dispositivo o procesador en el sistema, puede acceder al byte destino hasta que TAS ha finalizado. Incluso las rutinas normales del error del *bus* se ven alteradas para mantener TAS indivisible. ¿Para qué todo este jaleo si sólo se trata de probar y cambiar un byte? La razón se encuentra en la necesidad de suministrar medios de control y sincronización para diversas situaciones delicadas, en las que se puede encontrar el M68000 en los sistemas multiproceso y multitarea de hoy en dia. La idea general es que un recurso cualquiera —un fichero en disco, un banco de memoria, un dispositivo de entrada/salida (E/S) o incluso un procesador completo— puede ser compartido por diferentes grupos de usuarios. Para regular este acceso múltiple a un dispositivo se emplean tanto métodos *software* como métodos *hardware*. Diversos indicadores, semáforos y algoritmos de control de prioridades y colas se emplean para determinar quién y durante cuánto tiempo tiene acceso a un dispositivo determinado. Normalmente, cuando un programa emplea un dispositivo, lo marca como "en uso", poniendo un indicador de recursos, que puede ser un determinado bit o byte, a un cierto valor. Este recurso se devuelve al sistema, poniendo de nuevo a cero (o a un valor predeterminado) dicho indicador de recursos, de modo que otras tareas tengan acceso a él.

Supongamos, por ejemplo, que se ha asignado al byte en la posición $1000 el siguiente valor para todos los programas:

($1000) = $00	El fichero de empleados está libre; se permite el acceso para funciones de actualización.
($1000) = otro valor	El fichero de empleados está siendo utilizado; no se permite el acceso.

Un byte como éste puede tener un nombre pomposo como "byte de estatus del fichero de empleados". Hemos sugerido una dirección absoluta de modo que hay un sitio fijo donde cualquier programa pueda comprobar el estado actual del fichero de empleados.

Sin emplear TAS, un programa podría quedar así:

```
* Programa 6.9: Protección de ficheros sin TAS

WAIT  TST.B  $1000   ¿Está libre el fichero?
      BNE.S  WAIT    No, seguir intentándolo
      ST     $1000   Tomar el control del fichero poniendo el byte ($1000) a $FF
                     Esto indicará que el fichero está ocupado

* ST es equivalente a MOVE.B #$FF,$1000.

      <rutinas de proceso del fichero>
      CLR.B  $1000   Liberar el fichero para que otros puedan usarlo,
                     poniendo $00 en ($1000)
      <resto del programa>
```

Esto parece correcto, pero, ¿qué pasaría si se produce una interrupción justo antes del ST $1000? El programa que ha producido la interrupción podría hacer un TST.B $1000 y, encontrando el fichero libre, tomar su control, poniendo $1000 as $FF, y proceder a realizar las actualizaciones oportunas. Cuando el programa que nos ha interrumpido termine su tarea y devuelva el fichero al sistema, nuestro programa continuará el proceso de actualización de un fichero ya modificado, con consecuencias probablemente desastrosas.

Veamos cómo puede ayudar TAS:

```
* Programa 6.10: Protección de ficheros con TAS

WAIT  TAS    $1000   Comprobar ($1000) y asignar valores al CCR según proceda
                     Después poner ($1000) a un valor negativo
      BNE    WAIT    Según los valores del CCR el fichero estaba ocupado.
                     Seguir intentándolo
      <rutinas del proceso del fichero>
      CLR.B  $1000   Liberar el fichero para que otros puedan usarlo,
                     poniendo $00 en ($1000)
      <resto del programa>
```

El CLR.B es vital. Sin él las tareas podrían esperar indefinidamente la autorización para acceder al fichero; además, cuanto antes se produzca este CLR.B, mejor. El empleo del TAS, a diferencia del método que empleaba TST/ST, asegura que la secuencia de prueba y asignación de valores no puede ser interrumpida. Es importante notar que la secuencia BNE/WAIT está realmente comprobando el byte de estatus tal y como estaba antes de que TAS alterara el bit de signo. Cualquier interrupción o excepción que

ocurra después del TAS, pero antes del BNE, no afectará al CCR, dado que éste se salva y restaura siempre como parte del programa interrumpido.

En una aplicación real de tiempo compartido, el procedimiento de protección simple de ficheros que se ha mostrado aquí sería más complicado, empleando conceptos como los de ficheros de sólo lectura, públicos y de acceso restringido, protección de ficheros por registros, etc. Puesto que TAS afecta únicamente al bit 7 (el bit de signo) del operando (byte de estatus) para señalar ocupado, los otros 6 bits del byte pueden emplearse para indicar otras características del recurso compartido. Si es necesario el BNE (bifurcar si no es cero), puede sustituirse por BMI (bifurcar si es negativo), que prueba el indicador N.

Comparaciones con la familia de instrucciones CMP

El próximo grupo de instrucciones permite comparar los operandos fuente y destino. La noción de base relativa a la familia de instrucciones CMP es que el procesador se comporta como si estuviera efectuando un SUB, es decir, restando la fuente del destino, pero sin alterar este último. Recordar que un SUB sustituye el destino por la diferencia entre fuente y destino. Todo lo que el CMP hace es cambiar los indicadores del CCR (N, Z, V y C) como si hubiera efectuado una substracción. Tras un CMP en cualquiera de sus variantes, se puede emplear cualquiera de las instrucciones condicionales que dependen de los cc (códigos de condición), según se muestra en la tabla 6.4.

Hay dos aspectos relativos a los cc y el CMP que hay que tener en cuenta. Primero, siempre estamos haciendo preguntas en la forma "destino <condición> fuente", donde <condición> significa mayor que, menor que, etcétera. No es raro que los programadores tergiversen lo anterior, dado que las instrucciones se escriben en el formato fuente, destino.

Hay cuatro formatos para las instrucciones CMP, dependiendo del tipo de operandos que se estén comparando:

CMPA.z <ea>,Dn	Comparar con el valor de Dn (empleando el tamaño adecuado)
CMPA.z <ea>,An	Comparar con el valor de An (sólo L o W)
CMPI.z #<dat>,<adea>	Comparar un dato inmediato con el destino
CMPM.z (Am)+,(An)+	Comparar posiciones de memoria sucesivas

CMP cambia el CCR como lo hace un SUB, excepto que no afecta al indicador X.

Aquí se dan algunos ejemplos:

* Programa 6.11: Comparar dobles palabras sin signo

```
          CMP.L    D3,D4       Comparar las dobles palabras en D3 y D4
          BEQ      IGUAL       Saltar a IGUAL si D3 = D4. De aquí en adelante
                               se sabe que D3 y D4 son diferentes
          BHI      D4MAS       Saltar a D4MAS si el número con signo en D4
                               es mayor que el número con signo en D3
          BCS      D3MAS       Saltar a D3MAS si el número con signo en D3
                               es mayor que el número con signo en D4
          <nunca debemos llegar aquí>
          <ver comentario más adelante>
IGUAL     <rutinas para el caso D3 = D4>
          *    *    *
D4MAS     <rutinas para el caso en que D4 sea mayor>
          *    *    *
D3MAS     <rutinas para el caso en que D3 sea mayor>
          *    *    *
```

* Un BLS comprobará si los operandos mantienen una relación de "menor o igual";
* un BNE comprobará si la relación es de "desigualdad", de modo que hay bastante
* redundancia en las opciones que se pueden tomar tras un CMP sobre números
* sin signo
* En la secuencia de saltos condicionales que se ha desarrollado más arriba
* se han cubierto todas las posibilidades. A menudo se puede evitar un salto
* condicional, dado que todos los anteriores han excluido todas las posibilidades
* restantes excepto una. En el listado anterior podría reemplazarse el BCS
* por un BRA. Recordar que el indicador C tras un CMP o un SUB significa
* un acarreo negativo y no un acarreo, de modo que un BCS comprueba
* que la fuente sea mayor que el destino

* Programa 6.12: Comparar bytes con signo en modo inmediato

```
          CMPI.B   #-1,$4000   ¿El byte en $4000 es mayor/menor/igual que -1?
          BLE      MENOR       Saltar a MENOR si es menor o igual
          <acciones para el caso de que ($4000) sea mayor que -1>
          *    *    *
MENOR     <acciones para el caso de que ($4000) sea menor o igual que -1>
          *    *    *
```

* Las comparaciones con signo emplean BEQ/BGE/BGT/BLE
* como se muestra en la tabla 6.4
* Los datos de fuente de CMP se almacenarán en una o dos palabras de extensión,
* dependiendo del tamaño del dato

* Programa 6.13: Comparación de dos cadenas de caracteres ASCII en la memoria

* La cadena de caracteres 1 tiene como puntero a A1
* La cadena de caracteres 2 tiene como puntero a A2
* Ambas cadenas terminan con el carácter ASCII NUL ($00).
* Se pondrá el byte D6 a $00 si son diferentes y a $FF si son iguales.
* La palabra D7 contendrá el número de caracteres que coinciden.

```
        ST      D6          Poner el byte de indicación D6 a $FF
                            (seamos optimistas)
        CLR.W   D7          Poner a 0 el contador de caracteres
                            que coinciden
BUCLE   CMPM.B  (A1)+,(A2)+ Comparar los bytes, incrementar el puntero
        BNE     DISTIN      Los bytes son diferentes, se acabó.
                            Saltar a DISTIN
        TST     -1(A1)      ¿Se acabó el fin de la cadena?
        BEQ     FIN         Si, hemos terminado
        ADDQ.W  #1,D7       Incrementar el contador de caracteres
                            que coinciden
        BRA     BUCLE       Vamos a por otro caracter
DISTIN  CLR.B   D6          Indicar que no coinciden
FIN     <resto del programa>

* Los bytes se comparan sólo para comprobar si son diferentes (NE),
* ahora no nos importan consideraciones de signo. Si estamos interesados en secuencias
* lexicográficas, es decir, si la cadena de caracteres 1 está antes que la cadena 2
* en el diccionario podemos probar con un BHI o BLS (sin signo) e indicarlo
* de alguna forma en D6 cuando algún par de caracteres no coincida
* Nótese que TST.B -1(A1) comprueba el byte que se acaba de comparar;
* puesto que (A1)+ ha incrementado el puntero en 1 (byte), tenemos que movernos
* hacia atrás 1 byte también para estar en el sitio correcto. No confundir -1(A1)
* con -(A1), que desplazará hacia atrás el puntero, ocasionando resultados
* desastrosos
```

CMPA: Comparar direcciones

La variante de CMP, CMPA, como la variante de SUB, SUBA, se emplea sólo cuando el destino es un registro de direcciones. Sin embargo, hay una sutil pero importante diferencia entre CMPA y SUBA. SUBA, como todas las operaciones puras de aritmética que actúan sobre un registro de direcciones, no afecta al CCR, pero un CMPA no tiene sentido hasta que se reflejan en el CCR los cambios sobre los indicadores Z, V, N y C. Sin estos cambios no se podrían comprobar las condiciones definidas por los cc, es decir, si dos direcciones son iguales o si una es mayor/menor que la otra. Así pues, CMPA rompe las normas y afecta a los indicadores del CCR.

CMPA no permite códigos de tamaño byte, y cuando se realiza un CMPA.W, la palabra fuente se extiende, con signo, a 32 bits antes de que la comparación sea efectiva. Por otra parte, recordar que las direcciones son esencialmente números positivos sin signo y, como se verá en el próximo ejemplo, normalmente sólo se usarán pruebas sin signo tras un CMPA.

Uno de los principales usos del CMPA se encuentra cuando se desea comprobar si una dirección se encuentra dentro de ciertos límites. Ya se han visto varios ejemplos en los que, de diferentes formas, An se incrementa/decrementa empleando (An)+, -(An), ADDA, SUBA u operaciones con la pila. Hemos de tener la certeza de que An no excede, por exceso o por defecto, debajo de determinados límites. Veamos el siguiente ejemplo:

```
* Programa 6.14: Empleo de CMPA para comprobar los límites de una pila de usuario
* Se ha establecido una pequeña pila que emplea como puntero de pila a A2
* A0 apunta a la base de esta pila. A1 apunta al tope de la misma
* La pila crece hacia abajo en la memoria a partir de A0 y hacia A1 a medida
* que se introducen nuevos valores en ella. Diseñemos un método sencillo
* para comprobar que A2 no sobrepasa los límites
* Normalmente, A0>=A2>=A1

        MOVE.W  D1,-(A2)    Esta es la forma típica de entrar en una pila
        CMPA.L  A1,A2       ¿Está llena la pila?
        BEQ     LLENA       Sí, está a tope. Vámonos
        BCS     ERROR2      Estamos por debajo del valor del límite. Error
        <todo va bien, continuar>
        MOVE.L  (A2)+,D7    Esta es la forma típica de salir de una pila
        CMPA.L  A2,A0       ¿Hemos alcanzado la base?
        BCS     ERROR1      Estamos encima de la base
LLENA   <indicar que la pila está llena>
        * * *
ERROR1  <tomar el último valor que entró en la pila y ajustarla>
        * * *
ERROR2  <comprobar el último valor que salió de la pila y ajustarla>

* El primer CMPA efectúa la diferencia A2-A1, de modo que el indicador Z
* se pone a 1 sólo si A2=A1, indicando que la pila está exactamente llena
* C se pone a 1 sólo en el caso de un acarreo negativo, es decir, si A1 es mayor que A2,
* indicando que se ha sobrepasado el límite (como la pila "crece" hacia abajo
* estamos "por debajo" del límite). Un BEQ solo no sería suficiente,
* puesto que introducir valores en la pila puede decrementar A2 en 2 o en 4
* El segundo CMPA efectúa la diferencia A2-A1, de modo que el BCS indica
* si A2 es mayor que A0, es decir, ¡nos las hemos ingeniado para sacar más cosas
* de las que hemos metido
* Estar por encima de la base de la pila implica un error de programación,
* mientras que estar por debajo del límite sólo indica un abuso de la capacidad
* de la pila
```

Asociada muy de cerca al CMPA y a los saltos condicionales, existe otra instrucción, denominada DBcc, que da al M68000 otra ventaja más sobre la competencia. Veamos por qué.

DBcc: Probar los cc, decrementar y efectuar un salto condicional

El formato es:

```
DBcc  Dn,<etiqueta>   Comprobar los cc (códigos de condición), decrementar Dn
                      y condicionalmente saltar a la <etiqueta> indicada
```

Hay tres elementos que definen la instrucción DBcc. El primero es el familiar código de condición cc, análogo a los empleados para Scc. Un resumen de estos códigos se encuentra en la tabla 6.4. Asi nos encontramos con DBEQ, DBHI, etc. El segundo elemento es el contador de ciclos Dn. Dn representa aquí la palabra de orden más bajo de Dn y se denomina contador de ciclos del bucle o simplemente contador de ciclos. El tercer elemento es la <etiqueta>, que define el comienzo del bucle que se desea realizar. Como en la versión larga de Bcc, la <etiqueta> actúa como un PC de desplazamiento relativo de 16 bits, que se almacena en una palabra de extensión. Sin embargo, DBcc sólo permite saltos hacia atrás desde DBcc hasta la <etiqueta>, con una longitud máxima de $7FFE (32766) bytes. La <etiqueta> debe estar situada antes del DBcc.

Secuencia de sucesos en la ejecución de un DBcc

Cuando se alcanza un DBcc, la secuencia de sucesos es la siguiente:

Si la condición cc es cierta, ejecutar la siguiente instrucción.
Si la condición cc es falsa, decrementar Dn en −1, es decir,
Dn → Dn −1.
Comprobar ahora Dn. Si Dn es −1, ejecutar la siguiente instrucción.
Si Dn no es −1, saltar a la <etiqueta>.

A medida que vamos asimilando todo lo anterior y que va tomando un sentido DBcc, se revela como una instrucción compuesta extremadamente útil. Un gran tanto por ciento del trabajo que se realiza al programar en ensamblador va dirigido a la tediosa tarea de establecer controles para los diferentes bucles: contar el número de ciclos, controlar las condiciones que pondrán fin al bucle o ambas tareas exigen una atención permanente por parte del programador. DBcc reduce este trabajo, combinando todas las acciones que se han descrito anteriormente en una poderosa instrucción.

DBcc en acción

Vamos a ver la versión más simple de DBcc empleando DBF (DBF indica que el código de condición cc = Falso).

```
* Programa 6.15: Un bucle que no hace nada empleando DBF
* Ejecutar la tarea A 24765 veces y después continuar

        MOVE.W  #24764,D0     Poner D0 al valor de contador de ciclos −1
BUCLE   <la tarea A comienza aquí>
        DBF     D0,BUCLE      Volver a BUCLE hasta que D0 = −1
        <resto del programa>
```

```
* La primera vez que nos encontramos con el DBF, la condición es falsa por definición,
* de modo que hacemos D0 = 24763 y volvemos a BUCLE (puesto que D0
* no vale aún −1). Si seguimos este proceso descubriremos que se efectúa
* un total de 24765 veces antes que D0 alcance el valor −1
* Dado que la tarea A se ejecuta una vez cuando se entra en el bucle por primera vez,
* y puesto que repetimos el proceso hasta que D0 valga −1, hemos de poner el contador
* de ciclos del bucle a 1 menos del número de ciclos necesitados
```

Cuando se emplea DBF no se utilizan los aspectos de prueba de condiciones de DBcc, sino que sólo nos quedamos con el aspecto de cuenta. Incluso en este caso, al comparar el bucle anterior con un bucle normal en el que no se use DBcc, se puede apreciar un considerable ahorro de esfuerzo en la programación.

También existe la instrucción DBT, pero si seguimos la secuencia anterior es fácil darse cuenta que la instrucción

```
DBT  Dn,<etiqueta>
```

nos llevará de inmediato fuera del bucle. Es decir, DBT no hace nada en absoluto, se cita solamente para subrayar el funcionamiento lógico de las instrucciones DBcc. Si tiene alguna duda, relea la secuencia de funcionamiento de las instrucciones DBcc.

Cuando tenemos una condición "auténtica" como PL (positivo), CC (no hay acarreo), las instrucciones DBcc resultan equivalentes a la apreciada estructura de la programación EJECUTAR - HASTA QUE LA <condición> SEA VERDAD [1]. Dentro de esta estructura tenemos, además, una condición adicional que viene dada por la condición EJECUTAR - HASTA QUE Dn SEA −1, que nos permite poner un limite al número de iteraciones del bucle.

Los dos programas próximos aclararán todo esto.

```
* Programa 6.16: Hallar el primer valor negativo de una tabla de números

* A2 apunta a la base de la tabla de 100 números con signo
* Recorrer la tabla y poner el primer número negativo la palabra de orden más bajo
* de D4. Anotar su posición en la tabla en D5. Si no se encuentra ningún número
* negativo, poner D4 a −1 y D5 a 0

        CLR.W   D4          Ponemos la palabra D4 a 0
        MOVE.W  #99,D5      Ponemos el contador a (100 −1)
BUSCAR  TST.W   (A2)+       ¿Es negativa la palabra en (A2)?
        DBMI    D5,BUSCAR   BUSCAR hasta que encontremos un número
                            negativo o hasta que se acabe la tabla
        TST.W   D5          Es D5 = −1
        BMI     NOHAY       Si es así es que todas las palabras de la tabla
                            son positivas. Ir a NOHAY con D5 = −1
```

[1] Esta estructura se conoce en la jerga de programación por su denominación inglesa, DO-UNTIL, que por estar incorporada a la sintaxis de muchos lenguajes de programación es intraducible.

```
                MOVE.W   -2(A2),D4      Poner el valor de la primera palabra negativa
                                        en A2
                SUBQ.W   #100,D5        D5 = D5 - 100
                NEG.W    D5             D5 = -D5 = 100 - D5; ahora D5 contiene
                                        la posición de la primera palabra negativa
NOHAY           <resto del programa>

* TST.W pone los indicadores Z y N a los valores correspondientes a la palabra (A2)
* CMPI.W #0,(A2)+ haría lo mismo, pero empleando algún ciclo más
* Cada número no negativo (MI = falso) que se lea decrementará D5 y comprobará
* si D5 = -1, lo que significará el fin de la tabla y que NOHAY números negativos
* El primer número negativo (MI = verdadero) nos sacará del bucle y nos llevará
* a comprobar si D5 = -1 mediante un TST.W D5, es decir, si hemos recorrido
* la tabla sin éxito. Si D5 está en el rango 0-99, se calcula su posición en la tabla
* hallando primero D5 - 100 y "multiplicándolo" por -1 (emplear NEG.W
* es un truco habitual para hallar {a-Dn} a partir de {Dn-a}. Notar que el valor
* del primer número negativo se obtiene con un -2(A2), dado que (A2)+
* ya ha avanzado A2 a la siguiente palabra
```

Se puede emplear este tipo de programa, incorporando un DBcc, para localizar todos los números de cualquier tipo de una tabla. Por ejemplo:

```
BUSCAR  CMPI.W   #3000,(A2+)
        DBLE     D5,BUSCAR      BUSCAR hasta que (A2)< = 3000
        *        *      *
```

localizará todas las entradas de una tabla que sean mayores o iguales que 3000.

Las tablas o cadenas de longitud variable se pueden recorrer de muchas formas. Ya nos es familiar la idea de encontrar una cadena de longitud variable que termina siempre con el mismo carácter (el carácter ASCII NUL, por ejemplo), de modo que sólo tenemos que buscar este carácter en el bucle. Otra idea común es la de mantener la longitud de la cadena o tabla en la cabecera de la misma, de modo que a medida que se insertan o borran caracteres de la cadena o elementos de la tabla se actualiza esta cabecera. Veamos el programa 6.17 para ilustrar esta técnica:

```
* Programa 6.17: Encontrar la última entrada con valor cero en una tabla
* de longitud variable empleando DBEQ

* A2 apunta a la base de la tabla. La primera palabra de la tabla contiene
* el número de palabras con signo que siguen
* Recorrer la tabla buscando la última palabra cero. Anotar su posición en D5
* Si no se encuentran ceros poner D5 a -1.
* Si la tabla está vacía poner D5 = $00 y D0 = $FF. Si la tabla no esta vacía
* poner D0 = $00

        MOVE.W   (A2)+,D5       D5 contiene el número de elementos en la tabla.
                                Ahora A2 apunta al primer elemento de la misma
        SEQ      D0             Poner D0 a $FF si la tabla está ocupada
```

```
        BEQ       VACIA      La tabla estaba vacia, vámonos
        MOVE.W    D5,D4      Ahora D4 también contiene el número
                             de entradas de la tabla
        ASL.W     #1,D4      D4 = D4 × 2
        ADDA.W    D4,A2      A2 apunta ahora a después de la última entrada
        SUBQ.W    #1,D5      Poner el contador para el bucle a D5−1
BUSCAR  TST.W     −(A2)      ¿Es la palabra en (A2) cero?
        DBEQ      D5,SCAN    Continuar buscando hasta que (A2) sea cero
                             o hasta que se acabe la tabla
        TST.W     D5         ¿Es negativo D5?
        BMI       NOHAY      Si es así, no hay entradas nulas, nos vamos
                             a NOHAY con D5 = −1
        ADDQ.W    #1,D5      D5 = D5 + 1. Ahora D5 contiene la posición
                             de la última entrada cero en la tabla
NOHAY   <resto del programa>
        *    *    *
VACIA   <Salto; no hay elementos en la tabla>

* SEQ D0 comprueba el CCR después del MOVE a D5. Si D5 es $00 entonces
* (EQ es verdad) y ponemos D0 a $FF. Si no se verifica lo anterior, SEQ
* pondrá D0 a $00. Más tarde, en el programa, cuando A2, D5 y el CCR,
* podríamos referirnos al valor de D0, que recordará el resultado del SEQ
* Notar el uso de −(A2) para recorrer la tabla desde su final. Se ha tenido cuidado
* de dejar A2 apuntando 1 palabra después de la última de la tabla antes de entrar
* en el bucle de búsqueda. También se ha tenido que multiplicar por 2 el número
* de palabras para obtener el número correcto de bytes para A2
* Como en el programa 6.18, se ha puesto el contador de ciclos a uno menos
* de los necesarios
```

Comentarios generales acerca de DBcc

Sólo se ha podido dar un breve resumen de la enorme potencia que ofrecen las instrucciones de la familia DBcc, de modo que concluiremos con algunos comentarios:

- El valor que queda en el contador de ciclos del bucle es útil, como se ha demostrado en los dos últimos programas. Nos dice cómo y por qué ha terminado el bucle.
- Se puede emplear el contador de ciclos Dn como un registro índice de decremento automático.
- En algunas ocasiones puede ser útil entrar en el bucle directamente en la línea que contiene la instrucción DBcc, en vez de entrar en el bucle desde su comienzo:

```
        MOVE.W    #<contador>,Dn   Asignar un valor al contador de ciclos
        BRA       DBCC             Vamos a la línea en que está el DBcc
        *    *    *
BUCLE   <aquí empieza el bucle>
        *    *    *
DBCC    DBcc      Dn,BUCLE
```

Esto es perfectamente válido siempre que el contador de ciclos del bucle tenga un valor correcto. Es fácil equivocarse, dado que en este caso el valor que hay que asignar es el del número de interacciones que se necesitan y no uno menos.

- Tomando las debidas precauciones se puede cambiar el valor de Dn durante el bucle, acortando o alargando así la duración de éste.
- El M68010 tiene un modo particular de empleo de la instrucción DBcc para efectuar los bucles *(special loop mode)* que altera la secuencia de comprobación/decremento sin afectar al conjunto de la instrucción. Sin embargo, la velocidad de ejecución de los bucles pequeños se incrementa manteniendo la instrucción DBcc y los desplazamientos asociados a la misma en un esquema de precarga de las dos palabras necesarias, reduciendo así el número de accesos a la memoria.

Operaciones matemáticas

Ya se han visto las cuatro operaciones matemáticas del M68000, ADD (sumar), SUB (restar), MULU/MULS (multiplicar) y DIVU/DIVS (dividir) y algunas de sus variantes simples (como ADDQ, ADDI, ADDA, etc.). Vamos a estudiar ahora otras instrucciones que realizan varias funciones aritméticas.

NEG: Negación

Ya se ha usado NEG en un ejemplo anterior sin demasiadas explicaciones. De hecho:

NEG.z <adea> Niega el operando destino

simplemente reemplaza el destino por su complemento negativo a 2, es decir, efectúa {0−<adea>}, empleando la parte del destino indicada por z (L, W, o B). El destino debe ser una dirección efectiva de datos, lo que excluye los modos de direccionamiento An, d(PC), d(PC,Xi) e inmediato.

Si D0 contiene $12345678, entonces

NEG.L D0 da D0 =	$EDCBA988	(negación de los 32 bits)
NEG.W D0 da D0 =	$1234A988	(negación de los 16 bits menos significativos, el resto permanece sin cambios)
NEG.B D0 da D0 =	$12345688	(negación de los 8 bits menos significativos, el resto permanece sin cambios)

Una excelente razón para recordar que existe esta instrucción es que no se puede escribir:

```
SUBI.z  Dn,#<datos>    ¡ILEGAL!
```

de modo que para calcular {#<datos>−Dn} escribimos:

```
SUBI.z  #<datos>,Dn    Dn = Dn−#<datos>
NEG.z   Dn             Dn = −Dn
```

que sí está permitido.

Nótese también que a veces queremos {Dm−Dn}, sin alterar el valor de Dm. SUB.z Dn,Dm, nos da {Dm−Dn}, pero perdemos Dm. ¿Por qué no usar entonces SUB.z Dm,Dn seguido por un NEG.z Dn?

Otro empleo de NEG.z se encuentra tras un Scc para convertir la equivalencia VERDADERO = $FF en VERDADERO = 1, que a veces es más conveniente:

Scc	**<adea>**	Pone el byte en <adea> a $FF si es verdadero y a $00 si es falso
NEG.B	**<adea>**	Pone el byte en <adea> a 1 si es verdadero y a 0 si es falso

NEG afecta al CCR de la misma forma que SUB, aunque la información que obtenemos de los indicadores es menos útil:

X Se pone al mismo valor que C
N Se pone a 1 si el resultado es negativo
Z Se pone a 1 si el resultado es cero
V Se pone a 1 si se produce un rebose
C Se pone a 1 si se produce un acarreo negativo (siempre a menos que el operando sea cero)

EXT: Extensión del signo

En muchas ocasiones hemos observado que el M68000 extiende automáticamente el signo de un registro al pasar de 16 a 32 bits, por ejemplo. La instrucción EXT permite al programador ejecutar este truco de dos formas diferentes:

EXT.W	Dn	Extiende el signo de un byte a palabra, es decir, copia el bit 7 en el bit 15
EXT.L	Dn	Extiende el signo de una palabra a una palabra larga, es decir, copia el bit 15 en el bit 31

Si tenemos que D0 = $12345678, entonces EXT producirá los siguientes efectos:

```
EXT.W   D0      D0 = $12340078
EXT.L   D0      D0 = $00005678
```

dado que el bit 7 es 0 y el bit 15 es 0 también.

EXT es útil para conservar los signos de los datos cuando se cambian los tamaños de los mismos durante un programa. En ocasiones es fácil olvidar que tenemos un número negativo en D0, por ejemplo, y escribir:

```
MOVE.B  D0,D1
```

de modo que ahora sólo el byte menos significativo de D1 es "negativo". Más tarde, en el programa, podemos escribir:

```
ADD.W   D1,D3
ADD.L   D1,D3
```

y obtendremos resultados incorrectos. El uso de EXT resuelve este problema:

```
MOVE.B  D0,D1
EXT.W   D1      D1 tiene ahora el signo correcto
ADD.W   D1,D3
```

o bien, podemos escribir:

```
MOVE.B  D0,D1
EXT.W   D1      La palabra en D1 tiene ahora el mismo signo que el byte en D1
EXT.L   D1      La doble palabra en D1 tiene ahora el mismo signo
                que la palabra en D1
ADD.L   D1,D3
```

EXT altera el CCR de la misma forma que MOVE:

X No se altera
N Se pone a 1 si el resultado es negativo
Z Se pone a 1 si el resultado es cero
V Siempre a 0
C Siempre a 0

Matemáticas de multiprecisión

Hasta ahora sólo hemos hablado por encima del indicador X, pero no hemos entrado en detalles acerca del significado y funciones del indicador X ni en por qué en unas ocasiones cambia como el indicador C y en otras permanece inalterado. Hemos visto que el indicador X actúa más allá de los límites propios del bit más significativo de un registro. En el caso de ROXL (rotar a la izquierda empleando el indicador X) hemos visto cómo los bytes rotaban en un campo de 9 bits empleando el indicador X, las palabras lo hacían en un entorno de 17 bits y lo mismo pasaba con las dobles palabras, que empleaban 33. Necesitamos algunas instrucciones más que nos aclaren el funcionamiento de este misterioso indicador.

Instrucciones aritméticas extendidas

Introduciremos ahora las instrucciones extendidas ADDX, SUBX y NEGX, que emplean el indicador X de forma numérica, permitiendo que anteriores acarreos y acarreos negativos se incorporen a una aritmética multirregistro:

ADDX	Dm,Dn	Sumar (Dm + Dn + X), almacenar el resultado en Dn
ADDX	−(Am),−(An)	Sumar (fuente + destino + X); almacenar el resultado en el destino
SUBX	Dm,Dn	Resta (Dn − Dm − X); almacenar el resultado en Dn
ADDX	−(Am),−(An)	Restar (destino − fuente − X); almacenar el resultado en el destino
NEGX	<adea>	Sustituir <adea> por {0 − <adea> − X}

En los ejemplos que siguen debe recordarse que siempre que se produce un acarreo o un acarreo negativo, durante una operación aritmética, tanto los indicadores C como X, se ponen a 1. Las instrucciones de M68000 están cuidadosamente estudiadas, de modo que algunas instrucciones, como MOVE, afecten sólo al indicador C y no al X. De este modo, el indicador X "recuerda" un acarreo o un acarreo negativo hasta que necesitamos emplearlo mediante un ADDX o un SUBX, que puede estar varias líneas de programa después de que el indicador X adquiriera su valor. Sumemos dos números de 64 bits sin signo para ilustrar lo anterior. Cuando trabajamos con números de más de 32 bits, nos encontramos en el campo de la aritmética de precisión múltiple.

```
* Programa 6.18: Sumando números sin signo de 64 bits
*
* Cada uno de los números sin signo de 64 bits emplea 2 registros de datos:
* Número A = {D0}{D1}
* Número B = {D2}{D3}
* D0 contiene los 32 bits más significativos de A, D2 contiene los 32 bits
* más significativos de B, etc.
```

* La suma de A + B se almacenará en {D4}{D5}.
* El CCR reflejará los resultados relativos a los números de 64 bits;
* así, cuando C = X = 1, estaremos ante un acarreo de 64 bits,
* Z = 1 si el resultado (64 bits) es cero, etc.

```
MOVE.L   D1,D5
MOVE.L   D0,D4
ADD.L    D3,D5     D5 = D1 + D3. El acarreo viene ahora marcado por X
ADDX.L   D2,D4     D4 = D0 + D2 + X, donde X es el "acarreo" anterior.
                   ADDX creará ahora un nuevo X
```

* El resultado de A + B está ahora {D4}{D5}. El CCR reflejará cualquier acarreo de D4.
* Pero, ¿qué representa el indicador Z?, ¿qué pasa si D4 = 0 y D5 <> 0?
* La sección siguiente contestará esta pregunta

Aritmética extendida y el CCR

En la discusión siguiente, los comentarios relativos a ADDX son igualmente válidos para SUBX y NEGX, dado que afectan al CCR de la misma forma.

En el programa anterior, el comportamiento del ADDX recordaba totalmente al comportamiento de un ADD normal al que le hubiéramos incluido el indicador X. Un acarreo "tipo X" representa realmente 2^{32} (bit 32). La suma obtenida en D4 representa la doble palabra más significativa del resultado de 64 bits, porque lo hemos planeado así. El procesador, desconociendo, desde luego, la interpretación que planeamos darle a la instrucción ADDX.L, cambiará el CCR como si de adiciones de 32 bits se tratara (D0 + D2 + X). Si se produce un acarreo, lo encontraremos reflejado en los indicadores X y C (el anterior valor del indicador X se ha perdido, pero, puesto que ya lo hemos usado, no debe importar). El nuevo valor del indicador X representa realmente 2^{64} y puede que deseemos crear una aritmética de 80, 96 o incluso 128 bits. De este modo somos libres de encadenar una gran variedad de MOVE y SWAP, sabiendo que el registro X está a salvo. De nuevo esto indica la importancia de conocer cómo afectan las diferentes instrucciones del CCR.

Así, ADDX pone X = C como ADD. Para actuar correctamente con los números con signo en multiprecisión ADDX, también afecta a los indicadores N y V como la instrucción ADD.

Trucos con el indicador Z

Con el indicador Z se necesita un truco especial. Si una instrucción ADDX produce un resultado no nulo, el indicador Z se pondrá a 0, como en el caso de ADD; pero si el resultado es 0, y aquí está la trampa, Z también se mantendrá a 0. Siempre hay una razón para que Motorola prepare estas trampas, veamos cuál es.

Volviendo al último programa, supongamos que el resultado de ADDX.L D2,D4 resultara nulo, lo que es perfectamente posible (suponer D0 y D2 cero y que no se produce ningún acarreo en D1 + D3). En condiciones normales, ADD pondría Z = 1, haciéndonos creer que la suma de números de 64 bits ha resultado nula. Está claro que no podemos determinar si un número de 64 bits es nulo mirando solamente a los 32 bits más significativos. Esta es la razón para que ADDX ponga el indicador Z a 0 en el caso de un resultado no nulo y lo mantenga inalterado en caso de un resultado nulo. Por tanto, en nuestro ejemplo, si D4 fuera cero, el indicador Z reflejaría el resultado de la suma de los 32 bits menos significativos: D5 = D1 + D3. Si D5 es cero, Z = 1, y esto indicaría que el resultado de la suma de 64 bits es cero, mientras que si D5 no es cero, Z = 0, indicando que D4 o D5 (o ambos) no son nulos.

Ahora ya sabemos el porqué de este truco. Una consecuencia importante de todo lo anterior es que antes de embarcarnos en cualquier operación aritmética de multiprecisión debemos estar seguros de que Z = 1 (bit 2 del CCR) y X = 0 (bit 4 del CCR). Como sabemos que 4 = 00000100, una forma sencilla de garantizar estas condiciones es:

```
MOVE.X  #4,CCR
```

Vamos a ver cómo funciona SUBX restando números de 64 bits:

```
* Programa 6.19: Restando números sin signo de 64 bits
* Los datos se disponen como en el programa 6.18
* Cada uno de los números sin signo de 64 bits emplea 2 registros de datos:
* Número A = {D0}{D1}
* Número B = {D2}{D3}
* D0 contiene los 32 bits más significativos de A, D2 contiene los 32 bits
* más significativos de B, etc.
* La diferencia de A-B se almacenará en {D4}{D5}
* El CCR reflejará los resultados relativos a los números de 64 bits;
* así, cuando C = X = 1, estaremos ante un acarreo negativo de 64 bits;
* Z = 1 si el resultado (64 bits) es cero, etc.

        MOVE.L  D1,D5
        MOVE.L  D0,D4
        SUB.L   D3,D5     D5 = D1 - D3. El acarreo negativo viene ahora marcado por X
                <cualquier movimiento aqui no alterará X>
        SUBX.L  D2,D4     D4 = D0 - D2 - X, donde X es el "acarreo" negativo anterior

* El resultado de A-B está ahora {D4}{D5}. El CCR reflejará cualquier acarreo de D4
* El indicador Z refleja los valores {D4}{D5} y no sólo los de {D4}
* Nótese que el acarreo negativo X se resta
```

Multiplicaciones de multiprecisión

Las instrucciones MULU/MULS del MC68000 permiten multiplicar dos valores de 16 bits para obtener un resultado de 32 bits. El MC68020 ofrece

versiones que permiten realizar el producto de 32 × 32 bits = 64 bits con una sola instrucción (también existen nuevas instrucciones para la división), y el coprocesador matemático MC68881 nos lleva mucho más lejos, con operaciones de punto flotante de 80 bits. En las "peores" versiones del M68000 hay que programar estas extensiones, pero con la ayuda de ADDX y SUBX no es muy difícil. Vamos a esbozar brevemente los pasos necesarios para multiplicar dos dobles palabras A y B para obtener un producto de 64 bits. Supongamos A = (palabra1)(palabra2) y B = (palabra3)(palabra 4). Primero necesitamos los productos de las cuatro palabras, empleando MULU o MULS.

1. (palabra2) × (palabra4) = (prod11) (prod12) 32 bits
2. (palabra1) × (palabra4) = (prod21) (prod22) 32 bits
3. (palabra2) × (palabra3) = (prod31) (prod32) 32 bits
4. (palabra1) × (palabra3) = (prod41) (prod42) 32 bits

Donde cada producto (prod) es una palabra de 16 bits. Hasta ahora no nos hemos tenido que preocupar de los acarreos.

Como en la escuela primaria, podemos disponer estos productos en columna para sumarlos de derecha a izquierda:

```
                          (prod11)  (prod12)
                (prod21)  (prod22)            +
                (prod31)  (prod32)            +
(prod41)        (prod42)                      +
-------------------------------------------------
(suma 4)  (suma 3)  (suma 2)  (suma 1)
```

donde cada una de las sumas es:

suma 1 = prod12
suma 2 = prod11 + prod22 + prod23 con el acarreo X1
suma 3 = prod21 + prod31 + prod42 + X1 con el acarreo X2
suma 4 = prod41 + X2 con el acarreo X3

Con un uso juicioso de ADDX, SWAP y ASL, se obtiene finalmente el producto de dos dobles palabras.

Matemáticas extendidas con operandos de la memoria

Hasta ahora no hemos empleado la forma:

ADDX.z –(Am),–(An)

y alguien podría preguntarse para qué se ofrece esta extraña opción. Una respuesta puede hallarse mirando el ejemplo de multiplicación en multiprecisión anterior. Si algunos de los componentes de 16 ó 32 bits se almacenan hábilmente en la memoria, se puede evitar una gran parte de los manejos con los registros sumando directamente en la memoria y empleando el predecremento para recorrer la lista de operandos.

BCD (decimal codificado en binario)

El M68000 puede manejar un tipo de datos conocido como BCD (*Binary Coded Decimal* = decimal codificado en binario). Para ciertas tareas, especialmente para las financieras con gran cantidad de cálculos, los errores de las conversiones decimal a binario pueden ser un problema, incluso disponiendo de facilidades de punto flotante de alta precisión. La solución ofrecida por el BCD tiene un coste en memoria y velocidad, pero mantiene y manipula todos los números en el formato decimal exacto empleando un *nibble* (4 bits) completo para codificar cada dígito decimal. Puesto que un *nibble* tiene capacidad para codificar 16 números sin signo, del 0 al 15, y el BCD sólo emplea 10, del 0 al 9, puede verse la ineficiencia de este método (véase capitulo 1).

Los datos en BCD se manejan de forma similar a como lo haríamos con un ADDX, SUBX o NEGX:

ABCD.B	Dm,Dn	Sumar (Dm + Dn + X); almacenar el resultado en Dn
ADDX.B	−(Am),−(An)	Sumar (fuente + destino + X); almacenar el resultado en el destino
SUBX.B	Dm,Dn	Restar (Dm − Dm − X), almacenar el resultado en Dn
ADDX.B	−(Am),−(An)	Restar (destino − fuente − X); almacenar el resultado en el destino
NEGX.B	<adea>	Sustituir <adea> por {0 − <adea> − X}

El primer aspecto a notar es que todas las operaciones en BCD se realizan con bytes, lo que significa que podemos sumar, restar o negar dos números decimales con una sola instrucción. La mayoría de los ensambladores utilizará la opción B por omisión, pero aquí la mantendremos como recordatorio. Los números en BCD normalmente se encontrarán formando grupos de 4 por palabra o de 8 por palabra doble. Las secuencias más largas deben ser consideradas como cadenas en la memoria, de ahí la opción −(Am), −(An), que ya se vio con ADDX y SUBX. Supongamos que deseamos almacenar en la posición de la memoria, a la que apunta A0, el número decimal 564728. La cadena de números BCD de 3 bytes (6 *nibbles*) tendría este aspecto:

Cadena BCD A = 564728

Dirección del byte	*Byte almacenado*	*Equivalente decimal*	
A0	(0101)(0110)	(5) (6)	Más significativo
A0 + 1	(0100)(0111)	(4) (7)	
A0 + 2	(0010)(1000)	(2) (8)	Menos significativo

Nótese la secuencia de *nibbles* dentro de cada byte y la secuencia de bytes dentro de la cadena. Los dígitos menos significativos están más arriba en la memoria, de modo que si situamos el puntero al final de la cadena (A0) + 3, el formato con predecremento de ABCD sumará automáticamente con acarreo decimal y en la secuencia aritmética correcta 28, 47 y 56 con otra secuencia BCD especificada.

Operaciones en BCD y el CCR

Puesto que el acarreo determinado por el indicador X se suma de la misma forma que en el caso del ADDX, todo lo dicho anteriormente se aplica aquí. En particular, el truco para el indicador Z (ponerlo a 0 en el caso de un resultado no nulo, pero mantenerlo a su valor si el resultado es nulo) se mantiene como en el caso anterior. El acarreo X puede ser 1 ó 0, pero recordar que un acarreo decimal de 1 representa al número decimal 100 como resultado del acarreo producido por el byte menos significativo (cuyo rango va de 0 a 99). Es particularmente importante poner X = 0 y Z = 1 antes de comenzar ninguna tarea en BCD.

Las diferencias fundamentales en la forma en que las instrucciones AADX y ABCD afectan al CCR derivan del hecho de que los bytes BCD carecen de signo, o al menos éste no se obtiene como el complemento a 2, de modo que los indicadores N y V carecen de valor y permanecen indefinidos durante el tiempo que se empleen operaciones BCD.

BCD y el CCR. Resumen

X Se pone al mismo valor que C
N Indefinido
Z 0 si el resultado no es cero, no se altera en otro caso
V Indefinido
C Se pone a 1 si se produce un acarreo o un acarreo negativo

Sumando cadenas BCD en la memoria

Definamos otra cadena de valores BCD, denominada B, y escribamos un programa para sumar esta cadena y la definida anteriormente:

Cadena BCD B = 390112

Dirección del byte	*Byte almacenado*	*Equivalente decimal*	
A1	(0011)(1001)	(3) (9)	Más significativo
A1 + 1	(0000)(0001)	(0) (1)	
A1 + 2	(0001)(0010)	(1) (2)	Menos significativo

```
* Programa 6.20: Sumando cadenas BCD en la memoria empleando ABCD
* A0 y A1 apuntan, respectivamente, a los dígitos más significativos
* de las cadenas de 3 bytes A y B
* Sumar A y B. Almacenar el resultado en B. Señalar cualquier error de acarreo,
* es decir, si la suma excede 999999

        MOVE.W   #4,CCR      Poner X = 0 y Z = 1 antes de cualquier
                             operación BCD
        MOVEQ.W  #2,D0       Poner el contador de ciclos a 3
        ADDQ.W   #3,A0       Mover el puntero de la cadena A al final +1
        ADDQ.W   #3,A1       Mover el puntero de la cadena B al final +1
BUCLE   ABCD.B   -(A0),-(A1) Sumar un byte de A con uno de B,
                             empleando el indicador X (A + X) + B
        BDF      D0,BUCLE    Repetirlo 3 veces
        BCS      ERROR       Ir a ERROR si el acarreo está a 1
        <resto del programa>
        *   *   *

ERROR   <iniciar las acciones apropiadas>

* A1      (1001)(0101)  (9)(5)  Más significativo
* A1 + 1  (0100)(1000)  (4)(8)
* A1 + 2  (0100)(0000)  (4)(0)  Menos significativo
* El CCR tendrá X = 0, Z = 0. A0 y A1 volverá a tener sus valores originales
```

Números BCD negativos

La instrucción NBCD simplifica el manejo de los números BCD negativos, pero, puesto que no disponemos de indicadores N, X, o C, debemos ser cuidadosos para tener la certeza de que la suma se está realizando correctamente. NBCD actúa normalmente formando el complemento a 10 del operando (un byte) {0−<adea>}, pero si las operaciones han puesto X = 1, entonces NBCD lo tiene en cuenta y proporciona el complemento a 9 {0−<adea>−1}.

Esto permite negar una cadena BCD correctamente, como muestra el ejemplo siguiente:

```
* Programa 6.21: Negar una cadena BCD en la memoria empleando NBCD
* Con los mismos datos que en el programa 6.20, sustituir la cadena A
* por su opuesta (complemento a 10)
```

```
* Cadena A = 564728
* A0        (0101)(0110)    (5)(6)    Más significativo
* A0 + 1    (0100)(0111)    (4)(7)
* A0 + 2    (0010)(1000)    (2)(8)    Menos significativo

          MOVE.W    #4,CCR      Poner X = 0 y Z = 1 antes de cualquier
                                operación BCD
          MOVEQ.W   #2,D0       Poner el contador de ciclos a 3
          ADDQ.     #3,A0       Mover el puntero de la cadena A al final + 1
BUCLE     NBCD.B    -(A0)       Poner en -A0 {0-byte BCD-X}
          BDF       D0,BUCLE    Repetirlo 3 veces
          *    *    *

* Ahora la cadena A = 435272, que es el complemento a 10 de 564728
* A0        (0100)(0011)    (4)(3)    Más significativo
* A0 + 1    (0101)(0010)    (5)(2)
* A0 + 2    (0111)(0010)    (7)(2)    Menos significativo
* El CCR tendrá X = 0, Z = 0. A0 volverá a tener sus valores originales
```

El valor de X al principio era X = 0, de modo que cuando negamos el primer byte obtenemos {0−28} = 72 con acarreo negativo, de modo que X = 1. La siguiente negación nos da {0−47−1} = 52 con X = 1, y análogamente ocurre con la siguiente. Si más tarde, en el programa, necesitamos la cadena A, obtendremos la cadena 435272, que es tan positiva como la cadena 564728, que es la que se ha negado. Existen muchas soluciones, pero todas tienen que ser programadas, puesto que ninguna está implementada. Un método consiste en añadir a cada cadena un byte conteniendo el signo de la misma, por ejemplo los caracteres ASCII más (+) y menos (−), y almacenar siempre el valor absoluto de la cadena BCD.

Como ejemplo final para subrayar la mecánica BCD, restemos dos bytes BCD supuestos positivos que se encuentran en sendos registros de datos.

```
* Programa 6.22: Restar dos números BCD empleando SBCD
* Dados dos bytes BCD conteniendo números positivos en D0 y D1, calcular D0-D1
* Almacenar el resultado en D0
* Si el resultado es positivo, poner en A1 el signo ASCII "+";
* en caso contrario, poner en A1 el signo ASCII "-"

          MOVE.W    #4,CCR      Poner Z = 1, X = 0
          SBCD.B    D1,D0       Guardar en D0 el resultado de D0-D1
          BCS       NEG         Si C = 1 ir a NEG
          MOVE.B    #$2B,A1     Poner en (A1) el carácter ASCII "+"
          BRA       RESTO
NEG       ANDI.B    #$EF,CCR    Poner X = 0, el resto del CCR
                                permanece inalterado
          NBCD.B    D0          Negar el byte BCD en D0
          MOVE.B    #$2D,A1     Poner en (A1) el carácter ASCII "-"
RESTO     <resto del programa>
          *    *    *
```

Para obtener el valor absoluto del byte BCD negativo en D0 debemos poner X a 0 antes del NBCD. Supongamos, por ejemplo, D1 = (0000)(0010) y D0 = (0000)(0001), es decir, D1 = 2 y D0 = 1, entonces D0−D1 = −1 y el resultado en BCD {D0−D1} = (1001)(1001) con X = 1.

Si ahora efectuamos un NBCD con X = 1, obtendremos D0 = {0−99−1} = 0, que es incorrecto. Poniendo X = 0 tendremos D0 = {0−99} = 1, que es correcto.

Instrucciones de manejo de datos

Bajo este epígrafe incluiremos cinco instrucciones que cubren toda la escala de dificultades, desde lo más sencillo hasta lo más complicado:

SWAP{.W}	Dn	Intercambia el orden de las palabras en una doble palabra (la más significativa pasa a menos y viceversa
EXG{.L}	Rm,Rn	Intercambia el contenido de los registros
MOVEP.Z	Dm,d(An)	Mueve datos hacia los periféricos
MOVEP.Z	d(An),Dm	Mueve datos desde los periféricos
LINK	An,<tamaño>	Asigna pilas de usuario
UNLK	An	Libera pilas de usuario

En estas instrucciones, Z indica L o W (palabras dobles o simples).

SWAP: Intercambiar el papel de las palabras en un registro

SWAP Dn simplemente transforma la palabra más significativa de cualquier registro en la menos significativa y viceversa, de modo que los bits 0 a 15 pasan a ser los bits 16 a 31.

Un ejemplo de SWAP

Supongamos que tenemos la doble palabra D0 = $ABCDEF12 y necesitamos MOVEr la parte superior de la palabra, esto es, $ABCD a D6. Podríamos emplear:

SWAP	D0	La palabra más significativa de D0 es ahora $EF12 y la menos significativa es $ABCD

de modo que ahora D0 = $EF12ABCD. Tras esto podemos usar una instrucción MOVE:

MOVE.W	D6	D6 = $ABCD

Finalmente devolvemos a D0 su valor original con otro SWAP:

```
SWAP    D0          La palabra más significativa de D0 es ahora $ABCD
                    y la menos significativa es $EF12
```

De hecho, SWAP actúa como una rotación de 16 bits (a derecha o izquierda), sólo que mucho más rápidamente. La forma en que SWAP afecta a los indicadores del CCR es distinta a la forma en la que los altera una rotación ROR/ROL. SWAP afecta al CCR como un MOVE:

- X No resulta afectado (útil en los programas que emplean multiprecisión)
- N Se pone a 1 si el bit 31 es 1; se pone a 0 en otro caso
- Z Se pone a 1 si la doble palabra resulta cero; se pone a cero en otro caso (de modo que en realidad Z no cambia durante el SWAP, pero alguien puede necesitar probar para ver si es cero)
- V Siempre se pone a 0
- C Siempre se pone a 0

ROR/ROL afectan de forma diferente al indicador C a medida que los bits rotan.

EXG: Intercambia registros

EXG{.L} Rm,Rn emplea como operandos fuente y destino, indistintamente, registros de datos o de direcciones. Así, pues, son ejemplos válidos del empleo del EXG los siguientes:

```
EXG{.L}  Dm,Dn    El contenido de Dm pasa a Dn, y viceversa
EXG{.L}  Dm,An    El contenido de Dm pasa a An, y viceversa
EXG{.L}  Dm,An
```

El parámetro L es opcional, puesto que EXG implica el intercambio de una doble palabra (se intercambian siempre los 32 bits).

Todos los formatos para EXG son útiles y resultan equivalentes a la siguiente secuencia de MOVE:

```
MOVE.L  Rm,Rx
MOVE.L  Rn,Rm
MOVE.L  Rx,Rm
```

sin necesidad de emplear un tercer registro auxiliar Rx.

MOVEP: Intercambio de datos con los periféricos

MOVEP es una versión especial de MOVE para simplificar la interconexión con la anterior generación de dispositivos de 8 bits, que, por supuesto, se continúa empleando.

Aunque Motorola ha introducido muchos productos nuevos en la familia del 68000, ha decidido conservar una serie de características que permiten la compatibilidad, con periféricos y *chips* I/O [2], diseñados para microprocesadores de 8 bits, especialmente la familia de M6800.

Ya se ha mencionado que, de hecho, el M68000 puede interconectarse con dispositivos asincronos de alta velocidad de 16 bits, así como con periféricos de baja velocidad, normalmente dispositivos sincronos de 8 bits.

La instrucción MOVEP se ha pensado para facilitar la labor del programación a la hora de intercambiar información entre los registros de datos y los dispositivos I/O. El M68000 emplea un esquema de entrada/salida mediante un mapa de memoria *(memory-maped I/O)*, que para nuestros propósitos significa que podemos acceder a un periférico como si de una posición de memoria se tratara. En lugar de las instrucciones especiales de I/O que se encuentran en algunos sistemas, el M68000 realiza estos procesos de I/O con un MOVE y los operandos apropiados (desde luego, con alguna ayuda de dispositivos tan amables como controladores de discos, impresoras, terminales, etc.).

Puesto que los dispositivos periféricos de 8 bits se conectan preferiblemente a las 8 líneas de datos más significativas (o a las 8 menos significativas) del *bus* de datos del sistema (16 bits), los registros de control ocupan direcciones alternativas en el mapa de memoria del M68000, es decir, direcciones impares consecutivas o direcciones pares consecutivas.

Cuando se envien datos o instrucciones de control a uno de estos periféricos es necesario efectuar algunos cambios en el MOVE normal. Los modos (An)+ y −(An) funcionarán correctamente para transferencias entre posiciones contiguas de la memoria, pero una transferencia típica a un dispositivo I/O requerirá un esquema parecido al siguiente. Supongamos que D0 contiene los bytes (byte4)(byte3)(byte2)(byte1) y A0 contiene la dirección de entrada del dispositivo de I/O, entonces para transferir D0 a A0 haríamos algo así:

```
MOVEr byte4 a la dirección A0
MOVEr byte3 a la dirección A0 + 2
MOVEr byte2 a la dirección A0 + 4
MOVEr byte1 a la dirección A0 + 6
```

(Nótese la secuencia: el byte más significativo a la dirección más baja). Sin MOVEP, todo esto puede hacerse de varias formas, a cual más tediosa, por ejemplo:

---

[2] En adelante, nos referiremos a los dispositivos de entrada/salida por la abreviatura inglesa I/O *(input/output)*, que ya está incorporada a la jerga electrónica e informática castellana, dado que nuestro nivel ya nos lo permite y resulta más "realista".

```
MOVE.B   D0,6(A0)
ROR.L    #8,D0        Rotamos el byte2 a la posición menos significativa
MOVE.B   D0,4(A0)
ROR.L    #8,D0        Rotamos el byte3 a la posición menos significativa
MOVE.B   D0,2(A0)
ROR.L    #8,D0        Rotamos el byte4 a la posición menos significativa
MOVE.B   D0,(A0)
```

Empleando MOVEP, todo esto se reduce a una línea:

```
MOVEP.L  D0,0(A0)     Llevar D0 a (A0)...(A0 + 6)
```

MOVEP logra esto mediante el uso de rutinas propias para el posincremento del puntero de direcciones. MOVEP mueve los bytes comenzando por el más significativo y posincrementando en 2. Los formatos para MOVEP en salida son:

```
MOVEP.L  Dm,d(An)     Llevar los 4 bytes de Dm a posiciones de memoria
                      alternadas empezando en d(An)
MOVEP.W  Dm,d(An)     Como el anterior, pero empleando sólo 2 bytes
```

Vemos que L y W determinan el número de bytes a transferir, y que el destino debe ser un registro índice con desplazamiento, que se emplea para determinar la dirección en la que comience éste.

Motorola ha tenido una excelente razón para elegir d(An) como un operando. Normalmente se especifica un área de la memoria para zona de I/O. Se seleccionará un registro de direcciones para apuntar a la base de esta zona y se asignarán los desplazamientos simbólicos en el ensamblado fuente mediante un mnemónico, para distinguir las direcciones de los registros periféricos dentro del mapa de memoria de I/O. Por ejemplo, podemos encontrar:

```
MOVEP.W  D3,PIAD(A5)
```

De modo que A5 apuntará a la base de la memoria de I/O y PIAD será el desplazamiento que desde A5 se ha asignado, por ejemplo, a la PIA [3] 6821 de Motorola. En sistemas grandes, el uso de estos mnemónicos es indispensable.

La figura 6.15 muestra diferentes secuencias posibles para transferir diferentes tipos de datos a una dirección dada. Hasta ahora sólo hemos tra-

[3] PIA = *Peripheral Interface Adapter;* una traducción libre del término sería: "dispositivo para conexión y control de periféricos".

tado el procedimiento para enviar información a un periférico, los procesos para recibirla desde éstos son similares y nos permiten llevar los datos que el periférico envía a cualquier registro de datos:

MOVEP.Z	d(An),Dm	Cargar en Dm con 2 ó 4 bytes desde las direcciones alternadas que empiezan en d(An)

La entrada de datos con MOVEP funciona igual que la salida, sólo que en la dirección opuesta, asignando valores al registro elegido a partir de los datos que se encuentre en las direcciones alternas de I/O. Esta simetría en los procesos de I/O en el M68000 es otro intento de facilitar la tarea del programador.

Finalmente hay que notar que MOVEP no afecta al CCR, lo que es bastante sencillo. No estamos sino enviando o recibiendo una serie de bytes, y bajo esta base no existen criterios razonables para cambiar el CCR.

MOVEP: Resumen

MOVEP simplifica la transferencia de datos desde y hacia dispositivos I/O orientados al byte, incrementando automáticamente la dirección del operando en 2 por cada byte transferido. Esto permite que el *bus* de datos de 16 bits sea asignado de forma simultánea a dos dispositivos I/O de 8 bits.

Introducción al LINK/UNLK

Para entender las instrucciones LINK/UNLK se exige un buen conocimiento de cómo funciona la **pila;** de modo que vamos a repasar las características más notables de este dispositivo.

Como se vio en el capítulo 3, la pila puede emplearse tanto para salvar datos como para preservar el entorno de un programa durante una llamada a una subrutina. En particular, la pila se emplea para salvar la dirección de retorno de las subrutinas. La utilidad de la pila para almacenar datos reside en que emplea un mecanismo tipo LIFO[4], de modo que alterar los datos (introducir o extraer) de la misma no altera para nada la dirección misma de la pila, es decir, con la pila sólo hay que preocuparse de los datos, no de las direcciones de los mismos. Uno de los punteros de direcciones, el A7, que se conoce como SP (puntero de pila), apunta a la dirección de la última palabra que se guardó en la pila.

Hay muchas situaciones, sin embargo, en las que las subrutinas generan datos temporales o intermedios para a continuación llamar a otra subrutina, y así sucesivamente. Cuando nos introducimos varios niveles en lo que denominamos "subrutinas anidadas" puede ser una pesadilla para el progra-

[4] LIFO = *Last In First Out;* podemos traducirlo como: "sale primero el último que entró".

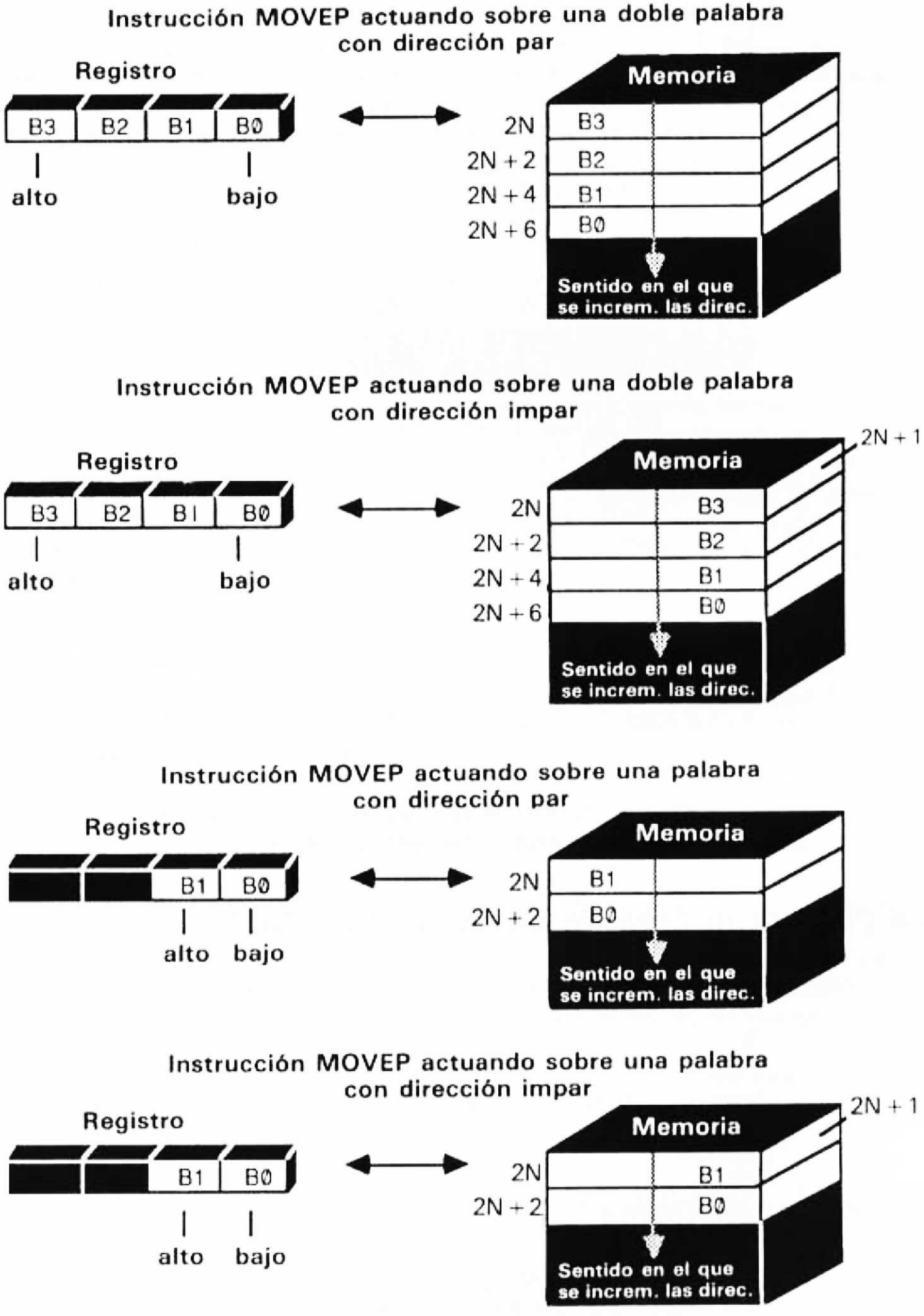


Figura 6.15
MOVEP.Z Dm,d(An) y MOVEP.Z d(An),Dm

mador mantener una relación de las direcciones de los datos temporales de los datos de cada subrutina. Una forma sencilla para salvar estos datos temporales sería reservar ciertas áreas de la pila para este fin, supuesto que podamos acceder a dichas áreas de datos sin alterar el funcionamiento normal de la misma. Es particularmente importante que bajo ninguna circunstancia perdamos una dirección de retorno de una subrutina.

Hasta ahora, simplemente, hemos entrado y sacado datos desde la cima de la pila con MOVE.z Dn,−(Sp) y MOVE.z (Sp)+,Dn, pero no existe ninguna ley que prohiba enredar en otras partes de la pila si se desea. Se puede tratar la pila como si de otra parte cualquiera de la memoria se tratase, empleando el SP (= A7) como otro registro de direcciones más. El programador es completamente libre de emplear el puntero de pila SP (= A7) con desplazamientos e índices, siempre que se observen dos normas:

1. No alterar jamás una dirección de retorno de una subrutina, que entrará en la pila automáticamente como un BSR (bifurcar a una subrutina) o JRS (saltar a una subrutina).
2. Tener la absoluta certeza de que cuando se produzca un RTS (retorno desde la subrutina) el puntero de la pila SP tendrá su valor correcto, porque RTS intentará obtener la dirección de retorno de la cima de la rutina. Si el SP no está apuntando al sitio que debe en la pila, RTS no encontrará la dirección de regreso y habrá comenzado el reinado del caos.

La idea subyacente en LINK/UNLK es la de ayudar al programador a asignar áreas de datos en la pila, sin violar las reglas o, al menos, reduciendo el riesgo.

Vamos a explicar cómo funcionan LINK y UNLK. Probablemente necesitará varios repasos hasta que quede claro.

La pila como área de datos: LINK/UNLK

Las instrucciones LINK y UNLK permiten al programador asignar y liberar áreas de datos temporales, denominadas tramas, sin perder ninguno de los elementos almacenados en la pila como valores de los registros o del CCR y direcciones de retorno de subrutinas. El mecanismo de LINK/UNLK permite además controlar todas las tramas anteriores asignadas por subrutinas previas.

¿Qué es una trama?

Una trama no es ni más ni menos que una parte de la memoria de la pila asignada a una subrutina que necesita espacio temporal de trabajo. El tamaño máximo de una trama es de 32 Kbytes y el número máximo de ellas de las que se puede disponer viene limitado solamente por la capacidad de

memoria disponible. Cada subrutina que emplea una instrucción LINK recibe una (y una sola) trama, que controlará de manera exclusiva. La trama permanecerá asignada hasta que sea designada con UNLK. Si, por ejemplo, tenemos cuatro subrutinas anidadas y cada una de ellas incluye una instrucción LINK, se dispondrá de cuatro tramas separadas en la pila cuando la cuarta subrutina esté en marcha. Cuando una subrutina termina, libera la trama antes de devolver el control a la subrutina llamante [5].

LINK	Asigna una trama de la pila
UNLK	Libera la trama de la pila

La idea básica es que un registro de direcciones cualquiera (excepto el A7, que ya actúa como SP) recibe la misión de actuar como puntero de la trama (FP) [6]. Durante la subrutina se puede manipular la trama empleando el FP con cualquiera de los modos de direccionamiento permitidos. El resto de la pila permanece inalterado (mientras que nos mantengamos dentro de la trama). A partir de este momento, considerar el FP como un puntero para los datos de una trama dentro de la pila.

Con estos preliminares, veamos el funcionamiento de LINK. Su formato general es:

```
LINK    An,#<-tamaño_del_bloque>
```

Donde An, que actuará como FP, puede ser cualquier registro de direcciones, excepto A7, que ya es el puntero de la pila. Una vez que se ha elegido An como FP, debe emplearse este registro de direcciones para todas las operaciones LINK/UNLK subsiguientes.

Veremos cómo la dirección contenida en An cambia a medida que nos movemos de una subrutina a otra, de modo que An siempre contiene el valor correcto para el FP de la trama activa.

El tamaño de la trama se indica mediante #<-tamaño_del_bloque>. El signo negativo se debe a que las pilas crecen hacia abajo en la memoria. El tamaño_del_bloque es un número de 16 bits sin signo (una palabra de extensión en la instrucción LINK), de modo que el máximo tamaño de la trama es de 32K. El tamaño_del_bloque debe venir dado por un número par de bytes.

Para hacer todo lo que se ha dicho hasta ahora, LINK realiza las tres funciones siguientes, que veremos primero rápidamente y luego con más calma para explicar los cómos y los porqués:

1. Salva An en la cima de la pila con un MOVE.L An,-(SP). El FP anterior está ahora a salvo en la pila.

---

[5] Quizá ahora resulte clarificador indicar que LINK puede significar atar, ligar, encadenar..., mientras que UNLinK puede traducirse por desatar. En este contexto, bien puede traducirse LINK/UNLK por asignar/liberar (una trampa, es decir, una zona de memoria).

[6] FP = *Frame Pointer:* "puntero de trama".

2. Salva el nuevo valor del SP en An con un MOVEA.L A7,An; nótese que A7 = SP. Ahora An es el nuevo puntero FP.
3. Pone SP a {SP menos tamaño_del_bloque}. Esto incrementa el tamaño de la pila en <tamaño_del_bloque> bytes más. La nueva trama ocupa ahora desde An = FP hasta SP.

Vamos a revisar ahora, paso a paso, el funcionamiento de LINK/UNLK, refiriéndonos a las figuras 6.16, 6.17 y 6.18.

Supongamos que estamos ejecutando una subrutina A que emplea A5 como FP para la trama de la subrutina A. El contenido de la trama A carece de importancia, pero nótese que la pila ha crecido (SP tiene ahora un valor menor) desde que se llamó a la subrutina A por primera vez.

Supongamos que en la subrutina A se alcanza un BSR que llama a la subrutina B. Como ya sabemos, BSR guarda la dirección de retorno de la subrutina B en la pila.

Ahora B va a comenzar su trabajo, si se desea podemos salvar los valores de CCR y de otros registros en la pila de la forma usual: quedarán allí, a salvo, hasta que B devuelva el control a A.

Supongamos que B necesita 512 bytes de memoria de trabajo (tablas, *buffers*, etc.); entonces la siguiente instrucción de B sería:

```
LINK  A5,#-512    Asignar mediante A5, 512 bytes
```

Esta línea está en realidad realizando las siguientes operaciones:

```
MOVE.L   A5,-(SP)    Salvar el puntero de trama (FP)
                     de la subrutina A en la pila
```

(Se salva el valor de A5 porque este registro de direcciones cambiará su valor durante la ejecución de LINK, y necesitamos recuperar su valor cuando ejecutemos un UNLK.)

```
MOVE.A   A7,A5       A7 = SP. A5 actúa como el puntero de trama (FP)
                     de la subrutina B
```

(Tras el paso 1, el puntero de la pila SP se deja apuntando al FP de A. Esta parte de la pila será el comienzo de la trama de B, de modo que copiamos el valor de SP en A5 para emplearlo como puntero de B.)

```
ADDQ.L   #-512,SP
```

(Sumando −512 a SP, incrementamos el tamaño de la pila en 512 bytes; cualquier valor que entre o salga de la pila durante la subrutina B lo hará en esta nueva parte de la pila. Podemos emplear A5 como FP para referirnos a cualquier parte de la trama de 512 bytes sin cambiar para nada el resto de

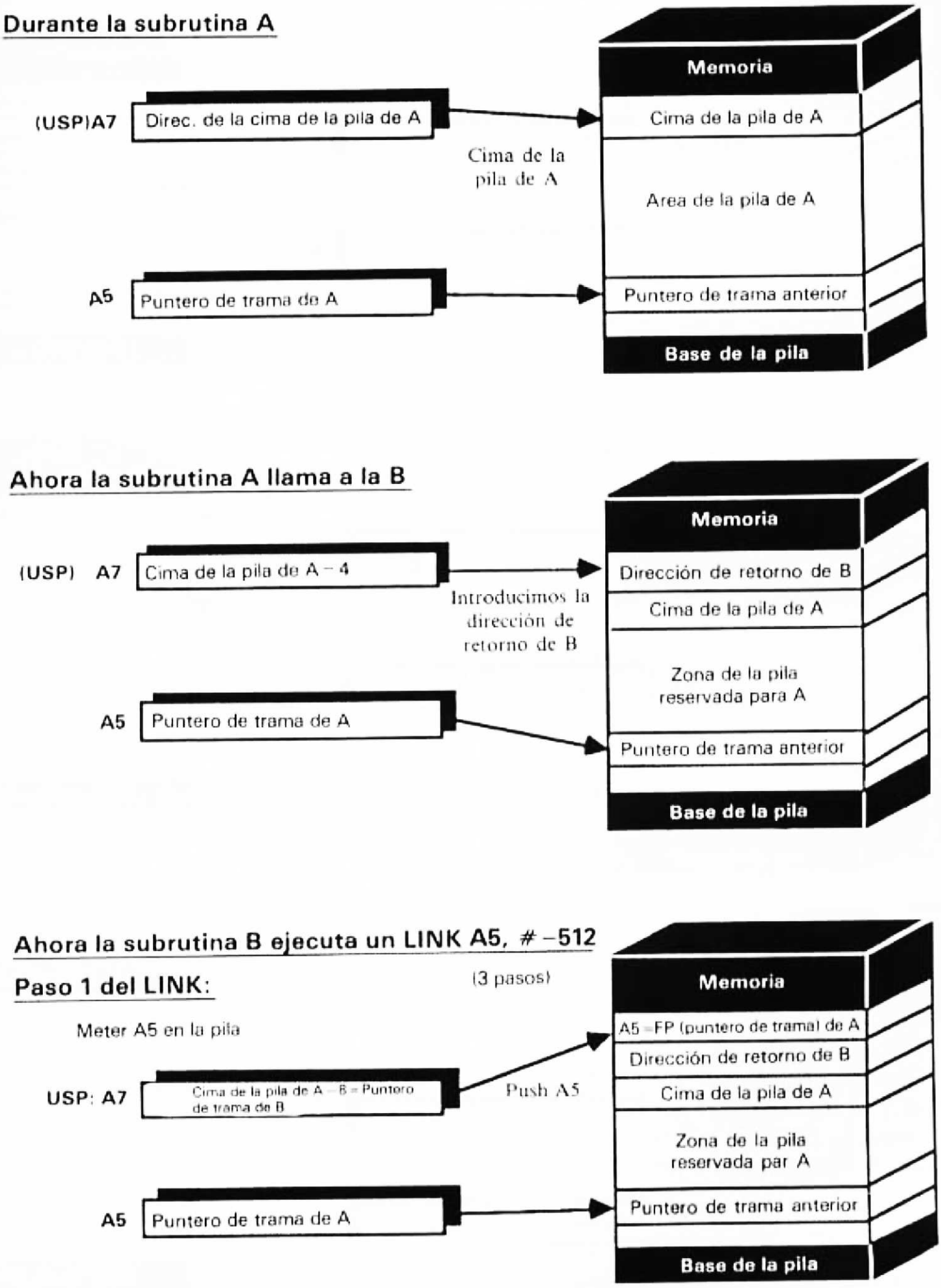
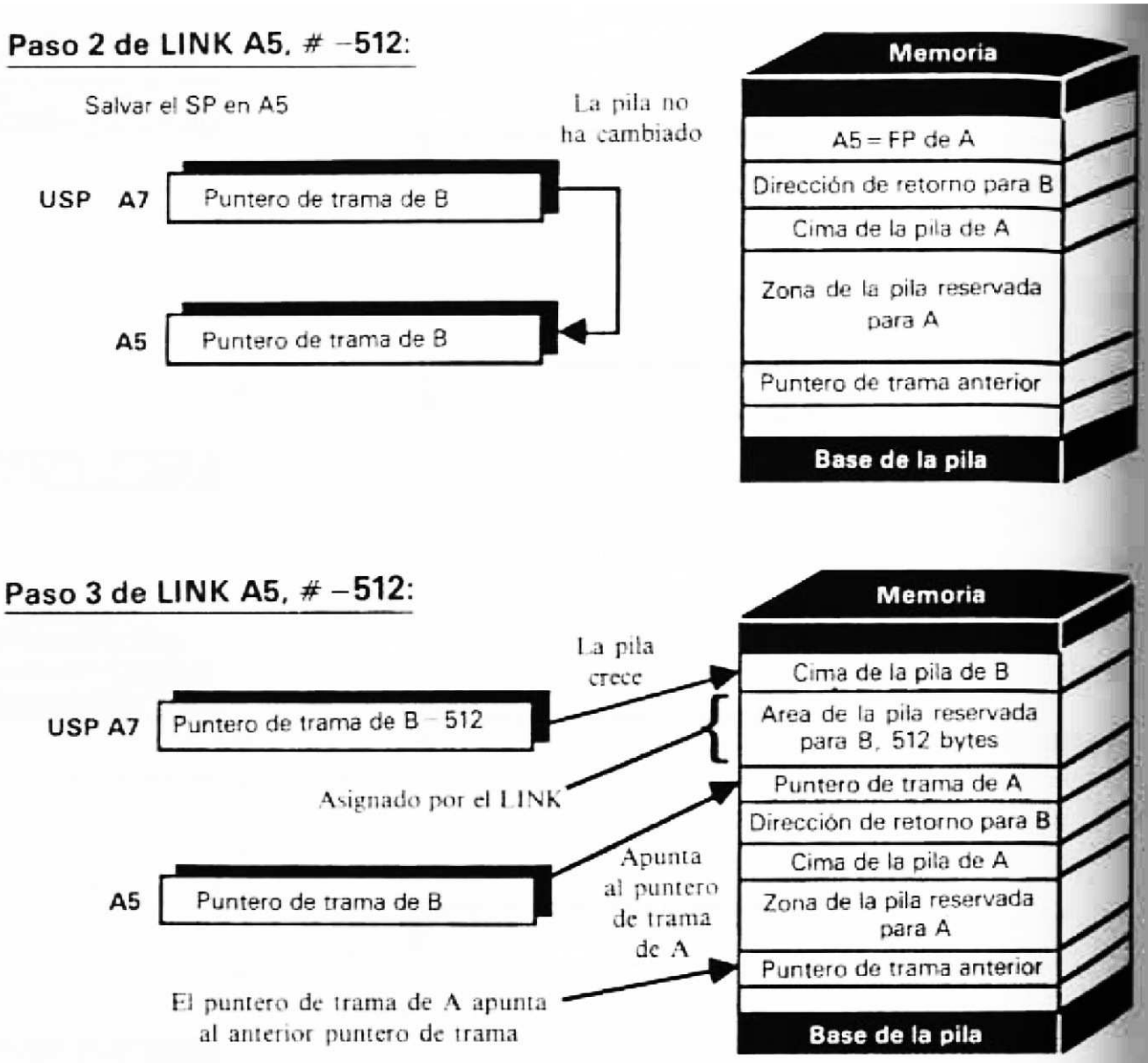


Figura 6.16
LINK y UNLK (primera parte)



La subrutina B ahora emplea el área que tiene reservada. El SP cambia (no importa). Eventualmente la subrutina B ejecuta un UNLK A5 (2 pasos)

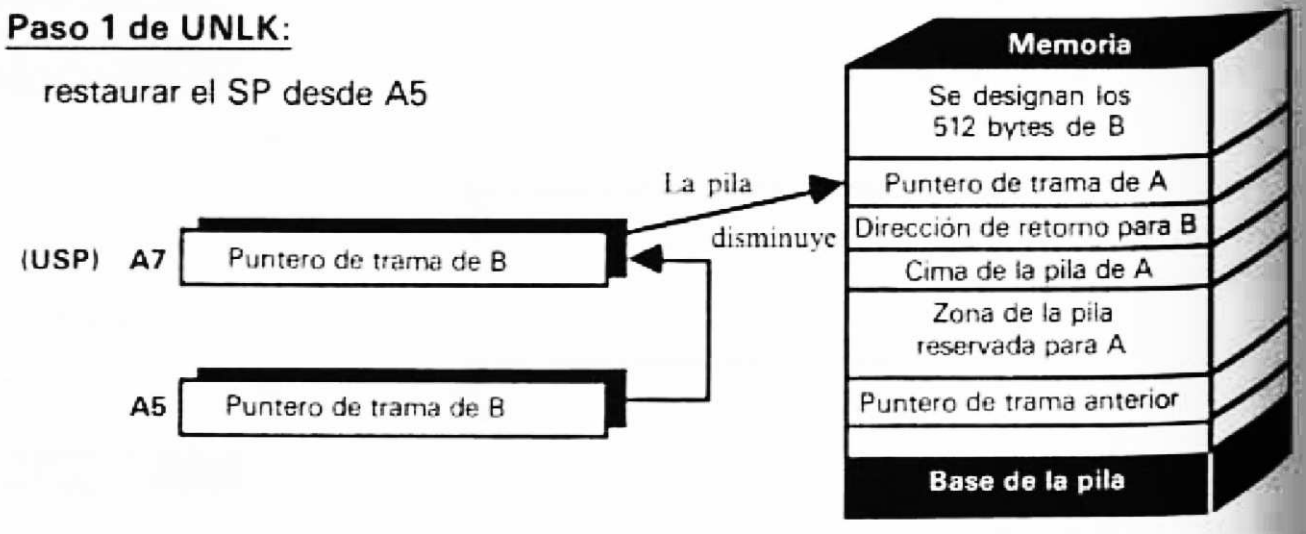
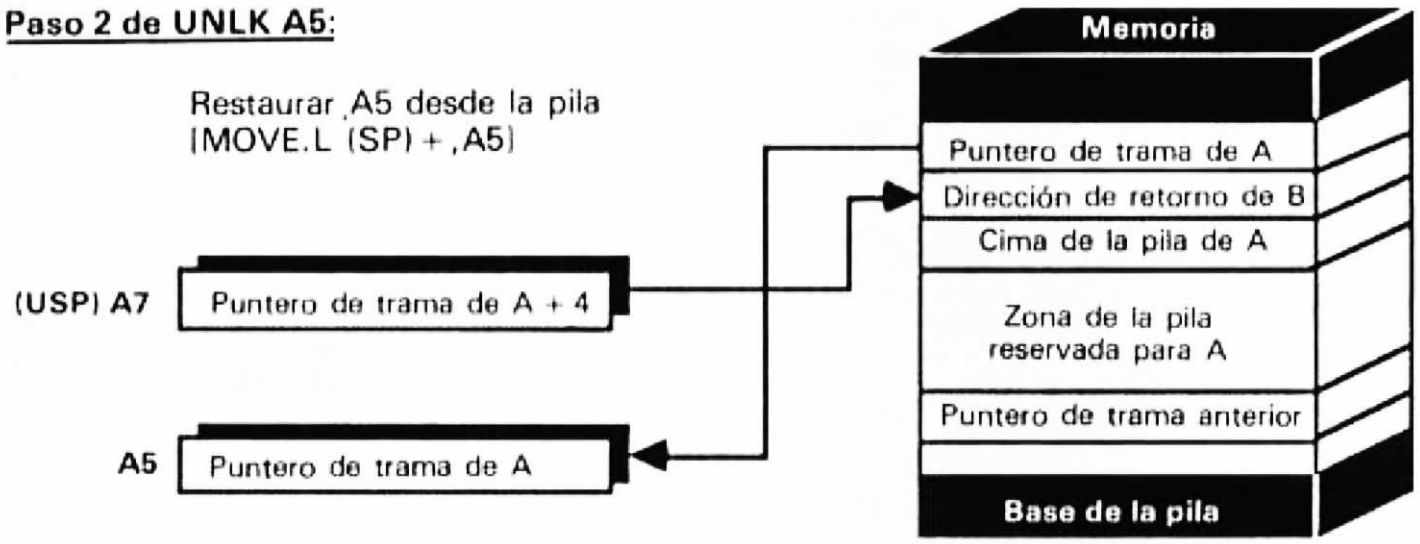
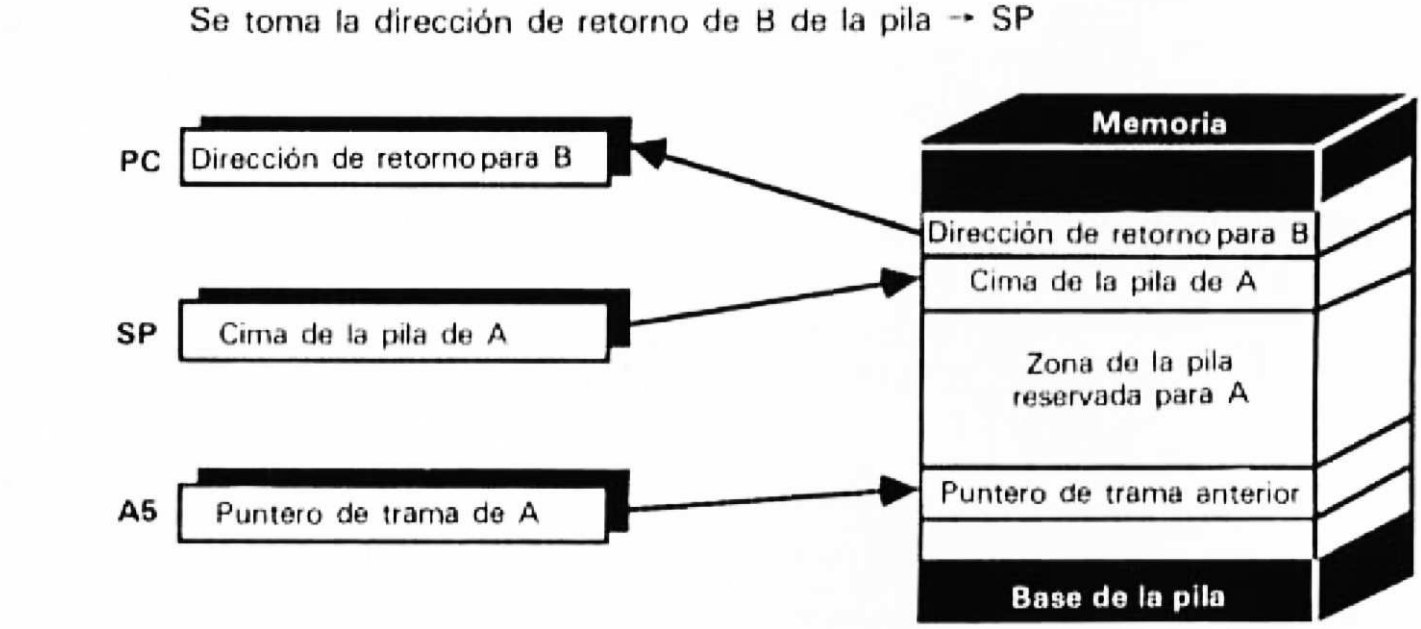


Figura 6.17
LINK y UNLK (segunda parte)

Paso 2 de UNLK A5:



Ahora la subrutina B ejecuta un RTS para volver a la subrutina A:



La subrutina retorna con la pila exactamente igual que al principio.

Figura 6.18
LINK y UNLK (tercera parte)

la pila; análogamente podemos emplear el puntero de la pila [SP = A7] para trabajar en ella, sin alterar la trama de B. Con esta disposición, la subrutina B puede emplear su propio espacio de trama de 512 bytes en la pila sin generar problemas. Justo antes de alcanzar un RTS o un RTR —suponiendo que se salvó el CCR—, que devolverá el control a la subrutina A, se necesita una instrucción:

```
UNLK    A5          Liberar la trama, restaurar los valores del SP
```

que liberará la trama y restaurará la pila automáticamente, empleando los dos pasos siguientes:

```
MOVEA.L  A5,A7      Restaurar el puntero de la pila
```

(Este es el paso inverso del paso 2 de LINK; ahora SP apunta a la posición en la que salvamos el FP de A en el paso 1 de LINK.)

```
MOVE.L  (SP)+,A5    Restaurar el valor de FP de A en A5
```

(Ahora recuperamos el valor del puntero de A de la pila y lo almacenamos en A5: esto cancela el paso de 1 de LINK.)

Si hubiéramos salvado el CCR en la pila, un RTR restauraría el valor de éste y nos llevaría a la subrutina A. Si no se han salvado los valores del CCR, un RTR nos devolverá a la subrutina A.

En cualquiera de los casos encontramos exactamente la misma disposición en la pila al retornar de la subrutina B que al abandonar la subrutina A. La pila y los punteros de trama tienen los mismos valores que al comienzo, y la zona de datos de B ha desaparecido.

Una vez que la subrutina A se haya completado, se liberará el área temporal asignada a ella y se retornará el control a la subrutina o programas llamantes. En el caso de que no existan más subrutinas anidadas, el control se devolverá al programa principal. En este momento, todas las tramas se habrán liberado, y A5 quedará disponible.

LINK/UNLK: Resumen

Se han descrito estas dos instrucciones en detalle porque nos muestran las líneas actuales de desarrollo de *software*. La elección del nombre LINK es significativa.

El primer elemento de una trama es siempre una palabra doble conteniendo la dirección de la trama previa. De modo que el puntero de una trama está apuntando a otra. Estas ideas de punteros apuntando a punteros no es tan complicada como pueda parecer. Esta es la idea de lo que denominamos listas encadenadas: de ahí el nombre de esta instrucción [7].

Más sobre los modos privilegiados

¿Qué es el modo privilegiado? En el sentido más amplio, la existencia de un modo privilegiado implica una estructura de niveles en un ordenador, estructura que va desde sistemas de acceso en los terminales, programas o ficheros protegidos por claves de acceso, hasta los niveles de protección especiales, donde el sistema operativo puede impedir el acceso, accidental o voluntario, a usuarios que podrían colapsar el sistema. El M68000 ofrece

---

[7] En este caso se emplea la acepción encadenar para traducir la expresión "linked list". Si bien en este contexto ésta es la traducción acertada al hacer referencia a zonas de la memoria, creemos que es más expresivo (sobre todo para las personas poco experimentadas) utilizar los términos asignar/liberar.

una combinación única de medios *hardware/software* que proporciona a diseñadores y programadores alguna ayuda en un área tan difícil como ésta.

Concluiremos el capítulo 6 revisando las instrucciones del modo privilegiado, algunas de las cuales ya se han visto brevemente.

Algunas de estas instrucciones realizan tareas tan familiares como MOVE y ANDI, y no hay ningún misterio en lo que hacen, pero, puesto que alteran (o pueden alterar) parámetros vitales del sistema, sólo se permite su uso en el estado supervisor del M68000. Intentar emplear una de estas instrucciones en el modo usuario produce el disparo de una señal TRAP, cuyo funcionamiento veremos más adelante.

El estado en que se encuentra el procesador se establece mediante el indicador S (bit 13) en el registro de estatus. Si S = 1, entonces el M68000 está en el modo supervisor (denominado también modo sistema o modo privilegiado). Si S = 0, el M68000 está en modo o estado usuario.

En otras palabras, el M68000 está en un modo o en otro, no hay modos intermedios. Como su nombre indica, los programas de usuario normalmente se ejecutan en modo usuario, mientras que el modo supervisor se reserva para el sistema operativo. Sentados ante un terminal no nos daremos cuenta, pero el indicador S estará constantemente cambiando de 1 a 0, y viceversa, a medida que el control pasa de una tarea a otra y al sistema operativo. Una notable excepción, por otra parte, es el Macintosh, de Apple Computer, que opera continuamente en modo supervisor.

El estado en que se encuentra el M68000 no sólo afecta al número o modo de las instrucciones que se pueden ejecutar, sino que también dicta qué registros se pueden acceder. El M68000 indica también el estado en que se encuentra (S = 0 o S = 1) mediante señales en las patitas de control de funciones FC, permitiendo que otros dispositivos, como *chips* de manejos de memoria, coprocesadores, etc., detecten su estado real y reaccionen de la forma adecuada. Una aplicación típica aquí es la de permitir a los diseñadores de sistemas controlar qué áreas de memoria están asignadas al usuario y cuáles al sistema.

Pilas en modo usuario y supervisor

Una consecuencia importante de estar en uno u otro de los modos es que el mismo símbolo A7 puede emplearse para acceder a dos registros físicamente diferentes sin ambigüedad. Cuando en modo supervisor se accede al registro A7, se está accediendo en realidad al SSP (puntero de pila del modo supervisor), mientras que si nos encontramos en modo usuario acceder al registro A7 significa acceder al USP (puntero de pila del modo usuario).

El M68000 mantiene, pues, dos punteros de pila diferentes y dos pilas, que en ocasiones se denominan pilas del sistema, para evitar confusiones con la multitud de pilas "privadas" que pueden mantenerse empleando los registros de A0 a A6.

Ahora, el usuario que trabaja en modo usuario es libre de acceder, ma-

nipular e incluso mutilar su propia pila empleando el registro A7 o el mnemónico asociado (SP). El daño, si se produce, quedará restringido a esa tarea en particular. Dado que el acceso a la pila es en modo supervisor, esto garantiza una cierta seguridad, que no absoluta, contra los programas mal diseñados. Como veremos, se emplea la pila en modo supervisor para salvar y restaurar constantemente información del sistema, de modo que un acceso accidental a A7 cuando A7 = SSP... ¡puede ser mortal!

Por otra parte, el OS [8] necesita acceder a la pila de usuario (por ejemplo, cuando se necesita salvar el USP al cambiar de tarea) y no podemos hacer esto accediendo a A7, porque A7 en modo supervisor indica SSP. Para evitar esto se emplean las instrucciones privilegiadas:

MOVE.L USP,An	Instrucción privilegiada. Copiar el puntero de la pila de usuario a An
MOVE.L An,USP	Instrucción privilegiada. Copiar An al puntero de la pila de usuario

En modo usuario, cualquiera de las instrucciones anteriores provocaría un *trap*. Una vez que el OS conoce el puntero de la pila de usuario, puede moverse con total libertad en la pila de usuario, empleándola para sus necesidades. La instrucción MOVE.L An,USP le permite restaurar dicho puntero cuando haya terminado.

Pilas y modo privilegiado: Resumen

Resumamos los aspectos de las dos pilas del sistema desde el punto de vista del privilegio. En modo usuario se puede manipular como se desee la pila de usuario, pero no puede accederse la pila del modo supervisor. En modo supervisor, ambas pilas son de libre acceso empleando MOVE y USP para acceder a la pila de usuario.

Los modos privilegiados y el registro de estado (SR)

El byte menos significativo del registro de estatus es el conocido CCR, del que tanto se ha hablado y que es accesible a todo el mundo. Se pueden probar, mover y modificar los indicadores del CCR en los estados usuario y supervisor. El byte más significativo del SR es totalmente diferente. En modo usuario, este byte es SOLO LECTURA mediante un MOVEr desde el SR (sin embargo, el MC68010/20 permiten sólo MOVEr desde el CCR;

[8] En adelante nos referiremos al sistema operativo por su abreviatura inglesa OS *(Operating System)*, que, como tantas otras, es ampliamente utilizada en castellano, incluso como nombre comercial de algunos sistemas operativos para ordenadores (DOS: Sistema operativo de disco).

véase el capítulo 8). En modo supervisor, el byte del sistema (SR) se puede leer y también alterar como MOVEr en los dos sentidos. Puede emplearse también ANDI/ORI/EORI.

Los indicadores del byte del sistema son:

Bits 8-10	Bits de la máscara de interrupciones IM (3 bits)
Bit 13	Indicador S (S = 1, modo supervisor; S = 0, modo usuario)
Bit 15	Indicador T (T = 1, modo traza activo; T = 0, modo traza inactivo)

Estos indicadores no pueden ser alterados en modo usuario, aunque sí que podemos leer su valor en ambos modos:

```
MOVE{.W}  SR,<adea>        Instrucción no privilegiada. Válida en ambos modos
```

lo que nos permitirá comprobar el estado de los indicadores IM, S y T (nótense las diferencias con respecto al MC68010/20 indicadas más arriba). Cuando se conecta el M68000 *(reset)*, éste comienza en modo supervisor, esto es natural y deseable, ya que algunos OS inician el sistema antes de permitir el acceso del usuario.

Pasar del modo supervisor al modo usuario no supone ningún problema, dado que, en modo supervisor, el OS siempre puede poner a 0 el indicador S con:

```
MOVE.W    #0,SR        Instrucción privilegiada. Se pone a 0 el indicador S
                       y todos los demás indicadores en el SR
```

o si no se desea alterar los otros indicadores del SR:

```
EORI.W    #$2000,SR    Instrucción privilegiada. Se pone a 0 el indicador S
                       sin alterar los demás indicadores
ANDI.W    #$DFFF,SR    Instrucción privilegiada. Se pone a 0 el indicador S
                       sin alterar los demás indicadores
```

También se puede emplear:

```
ORI       #<máscara>,SR    Instrucción privilegiada
```

para poner a 1 los indicadores del SR que se desee.

Podemos preguntarnos ahora cómo se pasa del modo usuario al modo supervisor si el indicador S no puede ser alterado en modo usuario. La respuesta está en el concepto de excepción. Una excepción en modo usuario llevará al procesador al modo supervisor poniendo S = 1, y dependiendo del

tipo de excepción, salvará o intentará salvar el contexto del procesador de diferentes formas.

Excepciones

En la jerga del M68000, las excepciones cubren una gran variedad de sucesos, algunos de los cuales quedan fuera del alcance de este capítulo. Los siguientes sucesos en modo usuario desencadenan una excepción:

— Errores que producen un TRAP
— Instrucciones TRAP
— Violaciones de privilegio
— Interrupciones: externas e internas
— Errores de *bus*
— *Reset*

Cada uno de estos sucesos pone el procesador en modo supervisor donde se procesan las excepciones. Vamos a ver los dos primeros tipos de excepciones, a modo de ejemplo, acerca del funcionamiento del procesador y de cómo, eventualmente, se devuelve el control al usuario.

Errores que producen un TRAP[9]

Al estudiar DIVU/DIVS, hemos visto que una división por cero se detecta automáticamente, produciendo un tipo especial de TRAP. Muy brevemente, esto es lo que ocurre en una división por cero:

1. Se entra en modo supervisor (S = 1)
2. Se salva el entorno del sistema en la pila del modo supervisor
3. Se salta al vector #5 en la tabla de vectores de excepciones
4. Se obtiene la dirección del programa de manejo de la excepción
5. Se ejecuta este programa que terminará con una instrucción RTE (retorno de una excepción)
6. Se restaura el entorno del sistema a partir del contenido de la pila del modo supervisor
7. Se continúa con el programa de usuario

[9] Podemos traducir TRAP por TRAmPa, es decir, cuando se produce un error, éste "cae en la trampa para errores", que disparará el proceso de excepciones, puesto que no nos parece oportuno emplear verbos como "trampear", y dado que TRAP aparecerá normalmente en conexión con el nombre de una instrucción, optamos por no traducir el término como norma general.

TABLA 6.5

Asignación de los vectores de excepción

Vector número	*Dec*	*Direc. hex.*	*Bloque*	*Función*
0	0	0000	SP	Reset: Inicialmente en el SSP[2]
1	4	004	SP	Reset: Inicialmente en el PC[2]
2	8	008	SD	Error del *bus*
3	12	00C	SD	Error de direcciones
4	16	010	SD	Instrucción ilegal
5	20	014	SD	División por cero
6	24	018	SD	Instrucción CHK
7	28	01C	SD	Instrucción TRAPV
8	32	020	SD	Violación del privilegio
9	36	024	SD	Traza
10	40	028	SD	Emulador de la línea 1010
11	44	02C	SD	Emulador de la línea 1111
12[1]	48	030	SD	(Reservado: sin asignación actual)
13[1]	52	034	SD	(Reservado: sin asignación actual)
14	56	038	SD	Error de formato[5]
15	60	03C	SD	Vector de interrupción no inicializado
16-23[1]	64	040	SD	(Reservado: sin asignación actual)
	95	05F		—
24	96	060	SD	Interrupción espuria[3]
25	100	064	SD	Autovector de interrupciones de nivel 1
26	104	068	SD	Autovector de interrupciones de nivel 2
27	108	06C	SD	Autovector de interrupciones de nivel 3
28	112	070	SD	Autovector de interrupciones de nivel 4
29	116	074	SD	Autovector de interrupciones de nivel 5
30	120	078	SD	Autovector de interrupciones de nivel 6
31	124	07C	SD	Autovector de interrupciones de nivel 7
32-47	128	080	SD	Vector de la instrucción TRAP[4]
	191	0BF		—
48-63[1]	192	0C0	SD	(No asignados, reservados)
	255	0FF		—
64-255	256	100	SD	Vectores de interrupción del usuario
	1023	3FF		—

[1] Los vectores números 12, 13, 16 al 23 y 48 al 63 están reservados por Motorola para futuras ampliaciones. No se debe asignar ningún periférico de usuario a estos números.

[2] El vector de reset (vector 0) requiere 4 palabras, a diferencia de los demás vectores, que sólo necesitan 2, y está localizado en el bloque de programas del modo supervisor.

[3] La interrupción espuria se produce cuando sucede un error de *bus* durante un proceso de interrupción.

[4] La instrucción TRAP #n emplea el vector número 32 + n.

[5] Sólo en el MC68010. Véase la sección de retorno desde las excepciones. Este vector está reservado sin asignación en el MC68000 y MC68008.

CHK: Disparar un TRAP si se exceden los límites

Se puede programar una TRAmPa similar para detectar si los valores obtenidos durante un cálculo exceden los límites prefijados; para ello, se emplea la instrucción:

```
CHK   <dea>,Dn    Disparar un TRAP si Dn es negativo o mayor que <dea>
```

Si Dn se mantiene dentro de los límites, ejecutaremos la próxima instrucción y si no es así, se disparará el proceso de excepciones, como en el caso de división por cero, pero empleando esta vez el vector #6 de la tabla de excepciones, para obtener la dirección del programa de manejo de excepciones.

TRAPV: Disparar un TRAP si se detecta un rebose *(Overflow)*

TRAPV generará un salto al vector #7 si se detecta un rebose.

Otros procesos TRAP

La instrucción:

```
TRAP  #<vector>   Salto al #<vector>
```

es un TRAP deliberado, de modo que existe una forma de que el programador en modo usuario entre en el modo supervisor; de hecho, TRAP #<vector> es una poderosa instrucción de programación para ampliar el repertorio de cualquier sistema.

La tabla 6.5 muestra la tabla de vectores de excepción, en la que se puede ver que hay 225 vectores independientes para rutinas de excepción. Algunos de estos vectores son fijos, como CHK y TRAPV; otros están reservados para rutinas presentes o futuras del sistema, tales como manejo de interrupciones. El resto puede emplearse para propósitos de usuario, como TRAP.

Como ejemplo simple, digamos que se puede asignar a la expresión COSH la instrucción TRAP #64, de modo que en la dirección $100 (la dirección del vector #64) se encuentra la dirección de la rutina COSH. Estas instrucciones, que no pertenecen al set del M68000, se conocen en ocasiones como llamadas al monitor o llamadas de servicio. Están al alcance de todos y pueden construirse para que parezcan instrucciones del set, con operandos que pueden pasarse a la rutina en cuestión empleando la pila.

Conclusión

En este capítulo hemos visto todas las instrucciones y modos de direccionamiento básicos del M68000. Los ejemplos propuestos han sido simples, para aislar la mecánica de cada grupo de instrucciones y operandos. Al mismo tiempo se ha pretendido con ellos dar una visión de las tendencias actuales de diseño, sus cómos y sus porqués. Ahora, por lo menos, ya conocemos el vocabulario elemental para entenderse con "el *microchip* de los ochenta" y tiene la preparación necesaria para enfrentarse con otros textos de nivel más elevado y con los ensambladores disponibles en el mercado. ¡Ojalá que el MOVE le acompañe! De todas formas, antes de que se encierre con sus máquinas, le invitamos a leer los dos últimos capítulos sobre el MC68010 y el MC68020.

7

El MC68010

En los capítulos precedentes hemos enfocado nuestra atención principalmente sobre el MC68000, el primer *chip* de la familia 68000 de Motorola. En este capítulo estudiaremos el siguiente miembro de esta familia, el M68010. A lo largo de este capítulo supondremos que el lector está familiarizado con las principales características del MC68000, es decir, que ha leído los capítulos 1 al 6. Puesto que las características del MC68000 son de carácter más avanzado, el material de este capítulo es más denso que el de los capítulos precedentes.

La clave del MC68010 se encuentra en sus capacidades de **emulación**, es decir, su capacidad para "simular cosas que no están realmente allí". En las emulaciones típicas, ciertas características *hardware* que realmente no existen son simuladas por *software*. Por ejemplo, algunas impresoras carecen de la capacidad de producir saltos de página. Esta capacidad puede emularse por *software*, contabilizando las líneas impresas en una página y luego produciendo suficientes saltos de línea hasta completar las 66 líneas de la página. Una de las aplicaciones más interesantes de las emulaciones se encuentra en los sistemas con memoria virtual. En estos sistemas, los programas pueden acceder a posiciones de memoria que están más allá del límite real de memoria disponible en *hardware*. La capacidad de manejar memoria virtual habría garantizado, por sí sola, un puesto en la historia para el MC68010, pero este *chip* todavía hace más cosas.

El MC68010 es capaz de emular sistemas operativos inexistentes, así como instrucciones (inexistentes en realidad) definidas por el usuario. Estas

características se conocen como **capacidades de máquina virtual.** Todas estas características simplifican el diseño de nuevos sistemas operativos, que es, por naturaleza, una tarea muy difícil. La máquina virtual permite a Motorola comprobar (emular) el comportamiento de nuevos *chips* de la familia del 68000 mucho antes de que estos *chips* estén realmente disponibles, incluso antes de que el diseño final de éstos se haya llevado realmente al papel. El proceso se realiza sobre un MC68010 en emulación. Por si esto fuera poco, el MC68010 realiza además otra función para aquellos usuarios que no puedan renovar su microprocesador con una nueva versión existente en la familia 68000, el MC68010 puede emularlo.

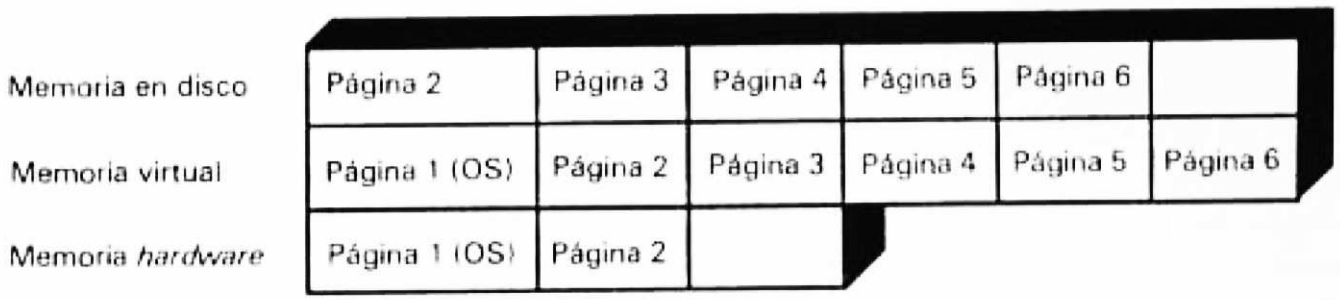
Es fácil comprender por qué Motorola ha implementado capacidades de emulación en el MC68010: permite reducir el coste de desarrollo de los *chips* de la familia 68000.

En este capítulo trataremos los conceptos de memoria virtual y máquina virtual, así como las características del MC68010, que le permiten soportar estas facilidades. Después discutiremos las diferentes formas de asignar áreas de la memoria y qué papel juegan éstas en el manejo seguro de la memoria virtual. Al final del capítulo discutiremos el modo lazo, que permite acelerar la ejecución de algunos bucles pequeños en un programa; finalmente discutiremos el MC68012, un pariente cercano del MC68010.

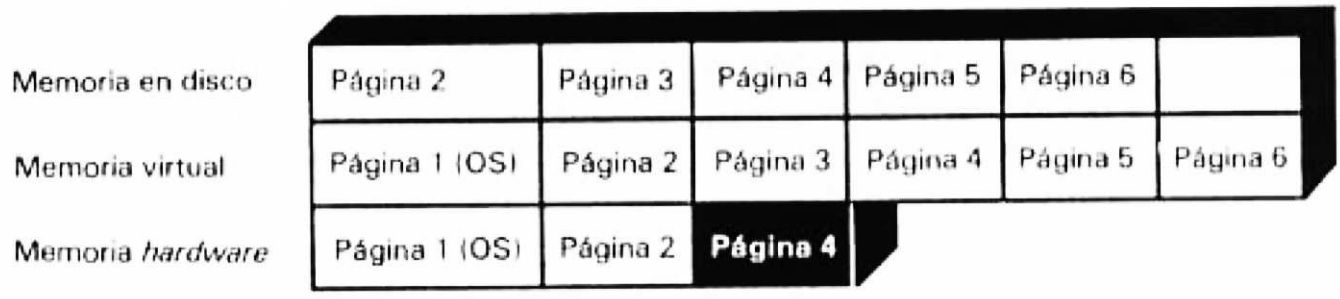
Memoria virtual

En cualquier ordenador hay una cierta cantidad de memoria real (memoria *hardware*) disponible. En la mayoría de los ordenadores, los usuarios están limitados a emplear esta memoria real, si bien es cierto que en algunos grandes ordenadores *(mainframes)* y microordenadores se empleaban algunos trucos de *software* para aparentar que había más memoria de la realmente disponible, es decir, estos sistemas empleaban memoria virtual. En las formas más desarrolladas de memoria virtual, el usuario puede acceder a la misma sin ningún tipo de restricciones y nunca sabe (ni tiene que molestarse en averiguarlo) cuál es la cantidad real de memoria *hardware* físicamente disponible. ¿Pero es esto el paraíso de los programadores? ¿Cómo se hace? ¿Cuál es la trampa? Como puede sospecharse, hay una trampa que conlleva importantes dificultades.

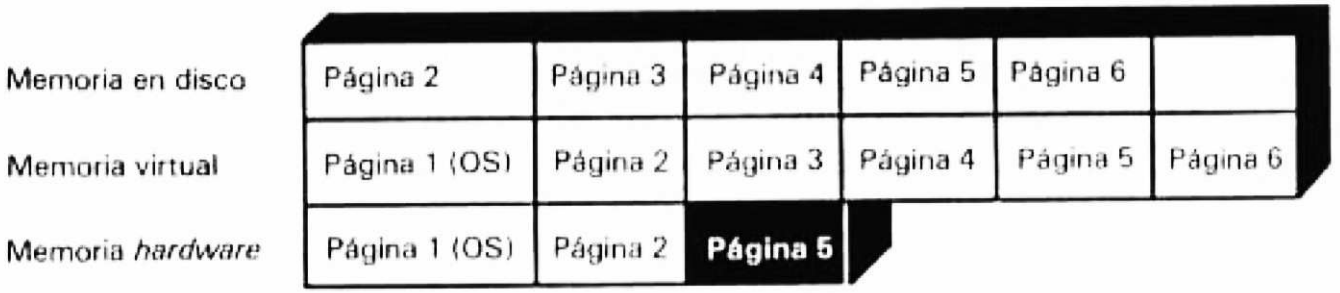
En cualquier instante, la memoria real contiene sólo una pequeña parte de la memoria virtual a la que se hace referencia. El resto de la memoria virtual se almacena mientras tanto en algún sitio, usualmente en un disco. Por ejemplo, supongamos un sistema de memoria virtual que permite a los programas direccionar hasta 384K de memoria, pero que tiene sólo 129K de memoria real. En este sistema, la memoria está dividida en páginas de 64K. De modo que en cualquier momento no puede haber más de tres páginas en la memoria real, aunque a la hora de programar se pueda disponer de seis. La página 1 está ocupada por el sistema operativo (OS) y tiene que estar siempre presente en la memoria real. Las páginas 2 a 6 constituyen el espa-



Configuración de la memoria al comenzar



Segunda configuración de la memoria



Tercera configuración de la memoria

Figura 7.1
Configuraciones de la memoria virtual en un 68000

cio para el programa que se está ejecutando y que reside realmente en el disco. La figura 7.1 compara las disposiciones de la memoria virtual. La memoria virtual no existe realmente en ninguna parte, excepto como un conjunto de punteros que direccionan el disco y la memoria real.

Cuando comienza la ejecución de un programa, la memoria se dispone como se indica en la figura 7.1a. El sistema de memoria virtual ha cargado, en este momento, solamente la primera página (página 2) del programa. Supongamos que ahora el programa necesita algunos datos que se encuentran en la página 4. Por lo que al programa se refiere, no se plantea ningún problema, simplemente se direccionan los datos de la página 4, pero el sistema de memoria virtual detecta una referencia a una página que no está en la

memoria real, de modo que la busca en el disco y la carga en memoria real. Ahora la configuración de la memoria (disco, real y virtual) se refleja en la figura 7.1b.

Cuando el programa del usuario hace referencia a las posiciones de la página 4, el sistema de memoria virtual las convierte en las posiciones correspondientes a la memoria real que ocupa esta página. Nótese que la página 4 reside, de hecho, en la página 3 de la memoria real; así, cuando el programa pide datos de la página 4, acaba obteniendo los datos pedidos, pero sacados de la página 3 de la memoria real.

Supongamos que el programa necesita ahora datos de la página 5, puesto que no hay memoria real libre, algunas de las páginas actualmente residentes en la misma deben eliminarse para hacer sitio a la página 5. El mejor uso de los recursos se consigue reemplazando la página 4 por la 5. Si la página 4 ha cambiando durante la ejecución, hay que volcarla al disco en su forma actual para no perder información. En cualquier caso, la memoria queda dispuesta como se indica en la figura 7.1c.

Decidir cuál de las páginas debe volcarse para hacer sitio a las demás es una función de muchas variables. De momento, digamos que si tiene una buena idea en este campo su futuro como programador será brillante. Puesto que los accesos a disco son hasta 100 veces más lentos que los accesos a la memoria, está claro que el precio de una mayor cantidad de memoria disponible se paga en velocidad de ejecución. Aún más, en los sistemas de memoria virtual sin restricciones y/o mal planeados, el uso de la memoria virtual puede llevar a situaciones en las que se produzcan demasiados accesos al disco. Por ejemplo, supongamos que el programa anterior entra en un ciclo en el que se llama alternativamente a las páginas 4 y 5, por ejemplo, 1.000 veces. Esto causaría 1.000 accesos al disco para leer los datos —una situación así supone un enorme desperdicio de recursos—. Es decir, hay que tener cuidado al emplear la memoria virtual.

Para implementar de manera eficiente un sistema de memoria virtual es necesario detectar (mediante *traps*) las referencias a posiciones ilegales de la memoria y traducirlas de memoria virtual a memoria real mediante *hardware* externo, normalmente una unidad de manejo de memoria (MMU). Sin esta capacidad, la CPU se vería obligada a desperdiciar su capacidad comprobando cada referencia a cada posición de memoria en todas las instrucciones.

Cuando se realiza una referencia a una posición ilegal de la memoria en un sistema sin capacidades de memoria virtual, ésta no es detectada inicialmente por el 68000. Sin pérdida de tiempo, el 68000 pasa la petición al *bus*, que, a su vez, la transmite a la memoria ("¡Hay una llamada para el señor Hex FFFFFFF0!"), donde se descubrirá que no es una posición permitida. El *bus* detecta este hecho y lo notifica al 68000, que generará una excepción debida a un error de *bus*, salvará (en la pila del sistema) el entorno del programa y llamará a las rutinas estándar de proceso de errores del sistema operativo. Normalmente se notificará un error de *bus* y se abortará la ejecución del programa.

¿Cómo se implementa la memoria virtual? Supongamos que alguna ins-

trucción, por ejemplo un MOVE, solicita una localización de la memoria (en memoria virtual) que está fuera de los límites de la memoria real. ¿Cómo es posible detectar este hecho y remediarlo de manera rápida? En un sistema con memoria virtual, habitualmente, se implementa una unidad de manejo de la memoria (MMU) entre el 68000 y el *bus*. Esta unidad intercepta cualquier llamada a la memoria y traduce las direcciones virtuales a direcciones reales, a continuación toma los datos y devuelve el control al 68000. Para el programador, y para el 68000, parece que los datos se han tomado de la memoria virtual. La MMU mantiene una tabla de las páginas que están en la memoria y de las posiciones que ocupan. Si la dirección virtual solicitada no se encuentra en la memoria real al ser llamada la MMU, generará un error de *bus*, que interrumpirá la instrucción en ejecución. En este momento se almacena la información del entorno del programa en la pila del sistema y las rutinas de error del *bus* la analizarán para determinar si se ha tratado de un error de memoria virtual; si es así, se emprenderán las acciones apropiadas para cargar esa parte de la memoria del disco en la memoria real (quizá sustituyendo a otra parte del programa que también se encontraba en la memoria real). Finalmente, una instrucción RTE devolverá el control al programa, y éste acabará de ejecutar la instrucción interrumpida.

¿Por qué es posible implantar sistemas de memoria virtual sobre el MC68010 y no sobre el MC68000? La respuesta se encuentra en que el MC68000, a diferencia del 68010, no almacena en la pila del sistema durante la interrupción que sufre la instrucción en curso, suficiente como para que un RTE concluya dicha instrucción. El MC68000 sólo almacena información suficiente para permitir un diagnóstico *software* de lo que ha ocurrido, de modo que se pueda informar al usuario del error del que ha abortado la ejecución del programa.

Máquina virtual

El M68010 no sólo soporta memoria virtual, sino que también está preparado para hacer frente al concepto, más amplio, de máquina virtual. Con un sistema de memoria virtual se consigue que un disco actúe como memoria real. Se pueden realizar otras emulaciones de *software* y *hardware* de una forma similar.

Por ejemplo, el concepto de *buffer* para un disco es exactamente el opuesto del concepto de memoria virtual; el usuario cree estar escribiendo en un disco, pero en realidad está escribiendo en la memoria real. Todos los accesos a disco se almacenan, de forma redundante, en la memoria, de modo que cuando se requieran estos datos, debido a que la velocidad de acceso a la memoria es varios órdenes de magnitud mayor que la de acceso al disco, los beneficios son obvios. La memoria real que se emplea para este uso se denomina memoria *cache*.

Otro ejemplo típico de emulación de *hardware* se encuentra en los *spools*. Un *spool* es un intermediario entre un dispositivo de alta velocidad

(la memoria real o *hardware*) y otro de baja velocidad (una impresora). Así, cuando un usuario cree que la salida de su programa se está volcando por una impresora, en realidad se está almacenando en un *spool* (que puede residir en un disco o cinta) para, quizá más tarde o quizá nunca, imprimirla. El empleo de *spools* puede ser muy útil en el caso de sistemas con una sola impresora (quizá de alta velocidad) y varios usuarios que simultáneamente intentan acceder a la misma. Sería absurdo tener a todos los usuarios esperando a que aquel que está imprimiendo termine; para evitar esta situación, todas las salidas de cada usuario se almacenan en el *spool* esperando su turno, mientras que cada uno de ellos continúa trabajando.

Más sofisticada es la capacidad de un sistema operativo (OS) de emular otro sistema operativo. ¿Qué es, en realidad, un OS? El OS es el programa más importante que se ejecuta en cualquier ordenador. Se carga cuando se enciende el ordenador y permanece allí hasta que se apaga, efectuando permanentemente funciones de control sobre la ejecución de todos los demás programas. En ocasiones se denomina a este sistema monitor [1]. En muchos ordenadores personales, el OS se almacena en ROM para evitar una destrucción accidental.

¿Cómo se generan los OS? Puede partirse de cero, pero sólo con un gran esfuerzo. Un método más sencillo es el de emplear un OS preexistente (OS-1), para controlar el desarrollo del nuevo OS (OS-2), hasta que el nuevo sistema esté lo suficientemente depurado como para sobrevivir por sí solo. Durante la fase de desarrollo es el OS-1 el que realmente controla el ordenador y emula al OS-2. Cuando se llega a alguna circunstancia especial se devuelve el control al OS-1, que es el que decide la acción a emprender y si debe o no devolver el control al OS-2. Esta situación requiere una habilidad especial. Por definición, un OS es una criatura omnipotente, el único y permanente guardián de todo lo que ocurre en un ordenador. Para emular el sistema que habrá de actuar sólo el OS-1 debe hacer creer al OS-2 que también él es omnipotente. En el M68010 esto se consigue ejecutando el OS-2 en modo usuario, en una situación de privilegio más bajo que el OS-1, que está en modo supervisor. Mientras el OS-2 ejecuta una instrucción, no está haciendo nada más allá de lo que haría un usuario normal, y la vida es fácil.

Cuando el OS-2 se encuentra con condiciones especiales (por ejemplo, interrupciones debidas al *hardware* externo o errores debidos a *bugs* en el OS-2), se genera una excepción y el control se retorna al OS-1, que decidirá la acción a emprender (normalmente resolverá la situación empleando alguna rutina *soft* hasta el extremo de poder retornar el control al OS-2). Uno de los recursos virtuales a los que el OS-2 debe tener acceso es al bit del sistema. El OS-2 debe tener acceso a aquellas instrucciones que permiten asignarse valores o comprobar su estado, así como aquellas instruccio-

[1] No debe confundirse el término monitor entendido como sistema con el monitor de algunos ordenadores personales. En estos últimos, el monitor está constituido por el conjunto de rutinas que el constructor implanta (en ROM). Las rutinas del monitor cubren un amplio espectro de funciones, desde rutinas para mostrar un determinado carácter en la pantalla, hasta las rutinas que inicialmente cargan el sistema operativo.

nes privilegiadas que requieran que éste tenga el valor 1. El OS-2 debe poder hacer todo esto, y hacerlo bien, mientras permanece en modo usuario. El truco no es muy distinto de otros descritos más arriba, consiste en generar la excepción adecuada y permitir que el OS-1 se encargue de todo y retorne el control al OS-2 cuando haya terminado.

En el M68000, el OS-2 puede determinar que él no está controlando el sistema mediante un MOVE SR,Dn y probando el bit del sistema. Esto funciona, porque el MOVE desde el SR no es una instrucción privilegiada en el MC68000. Por esto y por otros problemas, el MC68000 no puede realizar emulaciones correctamente.

En el MC68010 se resuelve el problema, pues el MOVE desde el SR es una instrucción privilegiada. Cuando en modo usuario se intenta ejecutar un MOVE desde el SR, se genera una excepción, el OS-1 toma el control y tiene la opción de pasar al OS-2 una versión alterada del SR en un registro Dn. El OS-2 no tiene modo de detectar qué es lo que realmente ha sucedido. La instrucción (no privilegiada) MOVE desde el CCR se ha añadido al MC68010, de modo que los códigos de condición pueden aceptarse sin generar una excepción.

Otro de los problemas que impiden al MC68000 realizar emulaciones con éxito es el hecho de que la información almacenada en la pila durante una excepción no es suficiente para completar la instrucción interrumpida, sino sólo para realizar un diagnóstico de lo que ha ocurrido, como ya se mencionó en la sección anterior al hablar de la memoria virtual.

¿Qué es lo que impide al omnipotente OS-2 (al menos eso se cree él) emular un OS-3 dentro aún del entorno de emulación? La respuesta es: nada. Si no fuera así, la emulación original no sería una verdadera emulación. Podría haber una larga cadena de sistemas emulados, cada uno de ellos creyéndose el principal y sin medios para determinar lo contrario.

La más sofisticada de las emulaciones consiste en simular una CPU mediante otra CPU. Esto permite que el *software* para la nueva CPU (por ejemplo, el MC68020) sea desarrollado y depurado mientras la nueva CPU está aún en desarrollo. La emulación en la familia del 68000 es posible, porque los sets de instrucciones se han diseñado para que sean compatibles. Por tanto, cualquier instrucción de un 68000 se ejecutará en cualquier procesador posterior de la familia; en los procesadores fabricados con anterioridad, bien se ejecutará normalmente o bien no se ejecutará, provocando una excepción debida a una instrucción ilegal. Por tanto, para emular un MC68020 con un MC68010 sólo es necesario cambiar las rutinas de manejo de instrucciones ilegales del OS para que, en caso de detectar una en particular, la emule en *software* antes de retornar.

Si la emulación es tan versátil, ¿por qué, simplemente, no se emulan CPU u OS en las aplicaciones diarias? La respuesta es eficiencia. El *software* emulado funciona mucho más lentamente que el *software* real, debido a los procesos de excepción y al uso de rutinas enteras para emular simples instrucciones.

Con esto terminamos nuestra discusión de las nuevas características del MC68010. En las tres próximas secciones examinaremos los nuevos regis-

tros e instrucciones que se emplean para soportar los procesos de emulación y la memoria virtual.

Registro vectorial de base

El MC68010 tiene un registro vectorial de base (VBR) que se emplea durante las emulaciones. Este registro no se encuentra en el MC68000. El VBR se emplea durante las transiciones entre el entorno del sistema operativo normal y el simulado. Para apreciar su significado necesita revisar algunos conceptos.

Cuando se genera una excepción, en el MC68000 ocurren varias cosas seguidas. Dependiendo de la clase de excepción, algunos registros especiales se modifican, se salva determinada información en alguna, o en ambas, de las pilas del sistema y, finalmente, se produce un salto a algunas de las 255 posibles rutinas de excepción. Lo que realmente nos importa aquí es el salto final que tiene lugar. Por ejemplo, una división por cero provoca el salto a la rutina número 5. Las direcciones de estas 255 rutinas se encuentran en las 256 dobles palabras primeras de la memoria (la primera de las dobles palabras no es una dirección, sino el puntero de la pila al encender el ordenador). Estrictamente hablando, la dirección para la enésima rutina se encuentra en la palabra localizada en la dirección 4n. La dirección de la rutina, por ejemplo, que maneja los errores que se producen al intentar una división por cero se encuentra en la dirección $14 (= 4 × 5). En la terminología habitual del 68000, se denomina **número del vector de excepción** al número n, **desplazamiento del vector** al número 4n, **vector de excepción** a la doble palabra almacenada en la dirección 4n y **tabla de vectores de excepción** a la tabla de 255 direcciones.

Imaginemos ahora un OS que controla la simulación de otro OS. Basándonos en la anterior discusión de este capítulo, es fácil entender que, durante la emulación, las excepciones se tratarán de una forma totalmente diferente a como se tratarían durante la operación normal del OS primario, puesto que este sistema estará controlando, de forma muy rígida, la situación, para que el sistema emulado no se dé cuenta de lo que realmente está sucediendo. En estas circunstancias se encontrará activo un conjunto alternativo de rutinas con puntos de entrada diferentes a los normales. Hay varias formas de alternar los dos conjuntos de rutinas, algunas de las cuales no requieren el uso de ningún registro nuevo, pero la más simple consiste en emplear el registro vectorial de base.

En el MC68010, las excepciones concluyen con un salto a la dirección contenida en la posición 4n + VBR de la memoria, donde VBR indica el número que está actualmente almacenado en el registro vectorial de base. Obsérvese que si el VBR es cero, esta dirección será la misma que en el caso del MC68000. De hecho, durante el encendido, el VBR está a cero, de modo que inicialmente la tabla de excepciones del MC68010 coincide con la del MC68000.

Cuando un OS establece un entorno en el que va a tener lugar una emu-

lación, carga un nuevo conjunto de rutinas de excepción y define una nueva tabla de vectores de excepción (que no se localiza en la posición cero de la memoria). Para habilitar el uso de esta tabla sólo es necesario cargar la localización de la tabla de vectores de excepción en el VBR con una instrucción MOVEC. Para volver al modo normal de operación basta con poner, de nuevo, el VBR a cero. El trabajo que hay que realizar para pasar de una tabla de excepciones a otra se reduce al mínimo, es decir, a una única instrucción MOVEC. Este método de alternar las tablas de excepción no sólo es rápido y ahorra pasos de programa, sino que elimina otros problemas potenciales. ¿Qué pasaría, por ejemplo, si durante un largo cambio de tablas de excepción se produce una interrupción? ¿Cómo se puede garantizar el uso de la rutina de interrupción correcta?

El siguiente ejemplo cambia el valor de VBR a $00100000 (hex):

```
MOVE.L  #$100000,D0     Fijamos el nuevo valor del VBR
MOVEC   D0,VBR          Cambiamos el VBR
```

La instrucción MOVEC se discute en la sección siguiente.

La figura 7.2 muestra una distribución típica de la memoria para un sistema operativo normal y un sistema emulado. Los pasos del 1 al 3 muestran lo que sucede durante un intento de división por cero cuando el control del sistema está en manos del sistema operativo real. Los pasos del 4 al 6 indican qué sucedería en el mismo caso, cuando el sistema emulado tiene el control. Para alternar las tablas de excepción basta con alternar el valor del VBR.

Las instrucciones MOVEC y MOVES

MOVEC es una instrucción privilegiada del MC68010 que mueve los datos desde y hacia los "registros de control". Una forma fácil de entender esta instrucción consiste en pensar que un MOVEC hace todo lo que no hace MOVE. La única excepción la plantea el USP, que se puede cambiar tanto mediante un MOVE como mediante un MOVEC. Si Motorola tuviera que comenzar, de nuevo, desde el principio, probablemente asignaría a la instrucción MOVEC las funciones de otras instrucciones MOVE especiales, como MOVE USP, MOVE SR y MOVE CCR. Con esto se conseguiría un conjunto de instrucciones más homogéneo.

El principal uso de MOVEC, en el MC68010, se encuentra a la hora de cambiar el VBR (ya discutido) o los registros de códigos de función SFC y DFC (que se discutirán más tarde).

La tabla 7.1 resume todas las instrucciones del 68000 que afectan a los registros de control. Para cada registro de control, la tabla indica si el registro se puede alterar con un MOVE o con un MOVEC y cuál fue el primer procesador en el que la instrucción se implementó. Un espacio en blanco indica que dicha instrucción no está disponible en ninguno de los procesadores existentes de la familia 68000. La columna del código hexadecimal

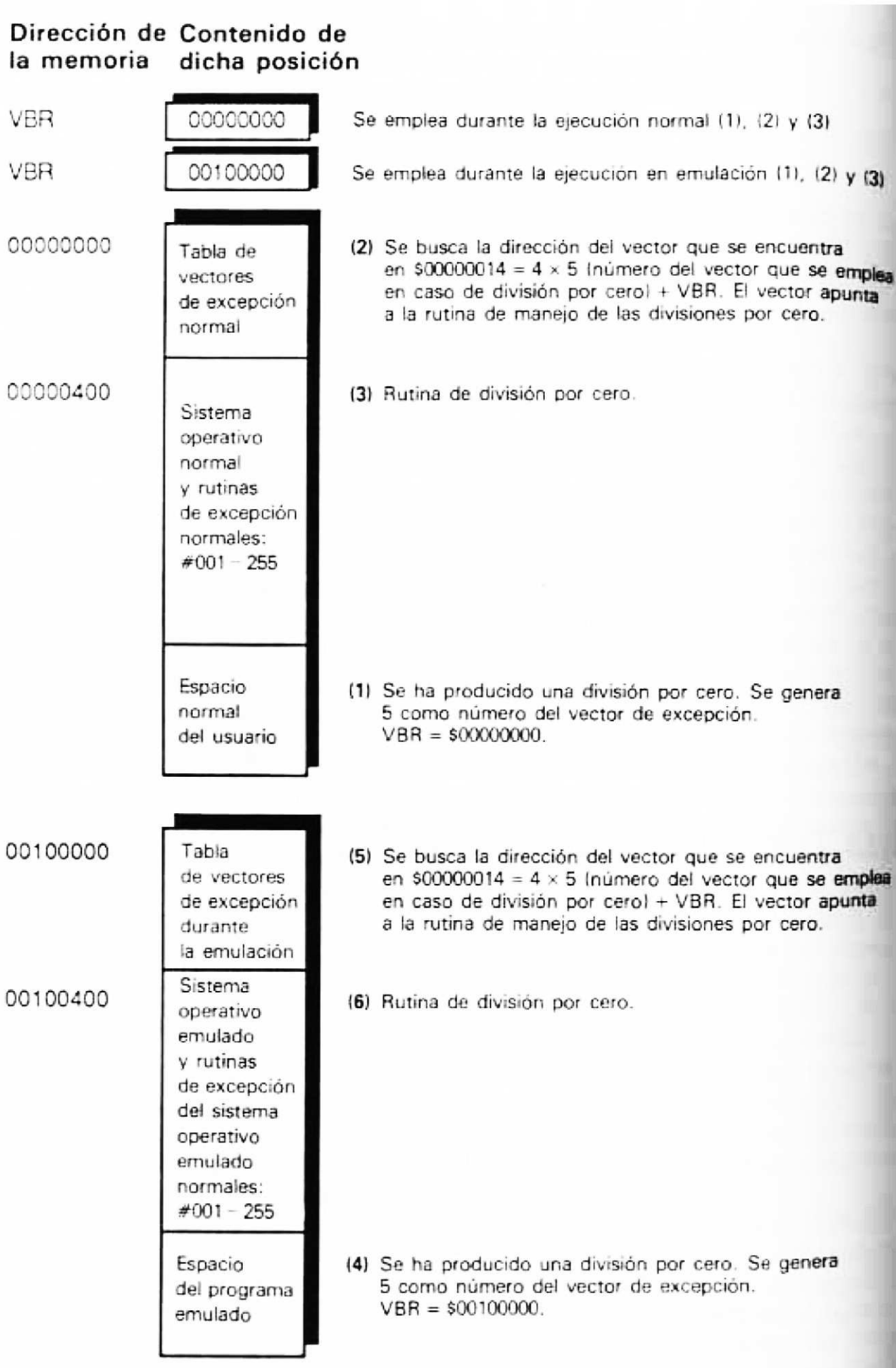


Figura 7.2
Uso del VBR durante una emulación

TABLA 7.1

Instrucciones MOVE para los registros de control

Registro de control	*Código hex.*	*Instr. MOVEC*	*Instr. MOVE*	*Situación de privilegio*
Al SR			68000	Privilegiado
Del SR			68000	No privilegiado
Del SR			68010	Privilegiado
Al CCR			68000	No privilegiado
Del CCR			68010	No privilegiado
USP	800	68010	68000	Privilegiado
VBR	801	68010		Privilegiado
SFC	000	68010		Privilegiado
DFC	001	68010		Privilegiado
MSP	803	68020		Privilegiado
ISP	804	68020		Privilegiado
CACR	002	68020		Privilegiado
CAAR	802	68020		Privilegiado

representa el código de tres dígitos hexadecimales empleados para representar el registro de control en la instrucción MOVEC. Se han incluido las instrucciones del MC68020 para completar la tabla.

Los registros SFC y DFC y los espacios de direcciones

Los registros de códigos de función origen y destino (SFC y DFC, respectivamente) son registros de 3 bits que designan diferentes espacios de direcciones. Sólo los programadores que trabajen al más bajo nivel en el desarrollo de sistemas tendrán necesidad de emplear, alguna vez, estos registros. Se emplean para controlar el manejo de la memoria y la seguridad del sistema. En un sistema que proteja ciertas partes de la memoria, el SFC y el DFC se emplean, en modo supervisor, para acceder a la memoria que normalmente es inaccesible. Para explicar cómo funcionan estos registros es necesario estudiar el *hardware* del 68000.

En un microprocesador de la familia 68000 pueden emplearse hasta 32 *pines* como líneas de direcciones. Cuando los datos se leen o se escriben en la memoria de estas líneas, transportan los 32 bits que afectan a la posición en cuestión. Se emplean 3 *pines* adicionales para determinar el código de función que establece qué tipo de memoria se está accediendo, ya se trate de memoria reservada para contener datos, programas o información del sistema (estos 3 *pines* proporcionan un máximo de 8 códigos de función, de los que 5 están actualmente implantados). Los códigos de función, también, indican un banco o un conjunto de direcciones a los que acceder. Téc-

nicamente hablando, es posible especificar hasta 8 bancos de direcciones (de memoria), cada uno con un rango de 32 bits y cada uno sobre su propia memoria *hard*. En la actualidad, sin embargo, sólo 4 bancos de 32 bits se emplean en la familia del 68000, y en las implantaciones más usuales se emplean estos 4 bancos en un solo bloque *hardware* de 32 bits, lo que reduce las esperanzas de aquellos que ansiaran un *bus* de direcciones de más de 32 bits.

La tabla 7.2 lista los 8 códigos de función y los bancos de direcciones que tienen asignadas actualmente. Cuando el 68000 se encuentra en modo usuario, es decir, si el bit del sistema está a 0, las instrucciones que se tomen de la memoria, para su posterior ejecución, tendrán un código de función 001. Cuando, sin salir del modo usuario, se intercambien datos con la memoria, el código de función será 010. En modo supervisor, los códigos serán 101 (instrucciones) y 110 (datos).

El quinto código de función actualmente implantado se denomina **código de función del banco de la CPU**. Este código se emplea en las comunicaciones con el *hardware* externo, es decir, durante las interrupciones, los *breakpoints* (cada vez que aparece una instrucción BKPT), en el nivel de control de acceso (durante las instrucciones CALLM y RTM del MC68020), y durante las comunicaciones con un coprocesador (MC68020). El banco de la CPU tiene un tratamiento diferente con respecto a los demás, puesto que no controla memoria *hard*, sino que se emplea para pasar diferentes parámetros a alguno de los destinos anteriormente citados.

Cuando el 68000 escribe (o lee) en la memoria, se manda una dirección de 32 bits junto con un código de función. El *hardware* externo decide entonces qué hacer con la señal. En las situaciones más simples, el *hardware* simplemente ignora el código de función y se ocupa de la dirección solicitada. En situaciones más sofisticadas, puede haber hasta 4 bancos de memoria *hardware*, uno para cada uno de los 4 primeros códigos de función. En este caso es posible que un OS, su tabla de datos, un programa y los datos de éste residan todos en la misma dirección numérica. Todos se acceden con la misma dirección, pero con un código de función diferente.

El código de función puede emplearse para implantar barreras de seguridad en el sistema. Esto puede lograrse mediante un dispositivo externo de manejo de la memoria. Por ejemplo, un OS puede permitir a los usuarios que ejecuten los programas que se encuentran en el área de programas del sistema (código de función 110), pero no que éstos tengan acceso a los programas en sí (código de función 101). Si los bancos para estos dos códigos de función corresponden realmente a áreas *hard* diferentes, el usuario no podrá nunca acceder al programa. El programa es, pues, invisible y está a salvo de miradas indiscretas. Si los bancos para los dos códigos de función corresponden a la memoria *hard*, el usuario podría acceder al programa, y es necesario emplear un dispositivo externo de manejo de memoria para evitar que el usuario examine el programa.

Supongamos que nos encontramos en uno de esos maravillosos sistemas con cuatro bloques diferentes de memoria. Supongamos que deseamos acceder a la posición 123456 en todos los bloques. Es fácil acceder a los ban-

TABLA 7.2

Códigos de función de los bloques de direcciones

Bits del código de función	*Bloque de direcciones*
000	No disponible: Reservado por Motorola para uso futuro
001	Bloque de datos del usuario
010	Bloque de programas del usuario
011	Reservado para definición por el usuario
100	Reservado para uso futuro de Motorola
101	Bloque de datos del modo supervisor (incluye los vectores de excepción del 2 al 255)
110	Bloque de programas del modo supervisor (incluye los vectores de excepción 0 y 1)
111	Bloque de la CPU (en el MC68000 sólo se emplea para confirmación de las interrupciones)

cos de datos (es decir, a las zonas *hard* reservadas para datos), pero los bancos de programas resultan inaccesibles. Inaccesibles a menos que se empleen las instrucciones MOVEC y MOVES. El siguiente ejemplo muestra cómo acceder a uno de los cuatro bancos, el banco de programas de usuario.

```
MOVE.L  #2,D0       Código de función del banco de programas de usuario
MOVEC   D0,SFC      Asignamos valores al código función fuente
MOVE.L  123456,D1   Se lee el valor de la doble palabra almacenada en el banco
                    de programas del usuario
```

Nota: Cambiar el SFC o el DFC sólo afecta a la instrucción MOVES, nunca a la forma en como se ejecutan otras instrucciones.

La capacidad de distinguir entre diferentes bancos de direcciones permite al MC68010 indicar al *hard* externo cuándo está accediendo bien a los programas del sistema, del usuario, o bien a los datos de cada uno de éstos. Resulta asi posible proteger una o varias de estas zonas del acceso no autorizado de los usuarios mediante un dispositivo *hard,* que eliminará chequeos redundantes por parte de la CPU, con el consiguiente ahorro de tiempo. Con el *hardware* adecuado, cada dirección banco de memoria se puede hacer corresponder con un bloque *hardware* diferente.

Las implementaciones típicas, sin embargo, usan, como ya se ha dicho, un solo bloque de memoria *hard.* Empleando los registros SFC, DFC y las instrucciones MOVEC y MOVES en modo supervisor, es posible conseguir que el OS acceda a los cuatro bancos.

Modo lazo

El MC68010 detecta automáticamente cuándo un bucle de instrucciones de tres palabras se ha repetido más de una vez, y entra en modo lazo. En

TABLA 7.3

Instrucciones que se pueden emplear en el modo lazo

Instrucciones	*Operandos*	
MOVE	lea,lea	o rea,lea
ADD SUB	lea,lea	o Dn,lea
CMP	lea,rea	o (Ax)+,(Ay)+
AND OR	lea,Dn	o Dn,lea
EOR	Dn,lea	
ABCD ADDX SBCD SUBX	−(Ax), −(Ay)	
CLR NEG NEGX NOT	lea	
TST NBCD	lea	
ASL ASR LSL LSR	lea	
ROL ROR ROXL ROXR	lea	

este modo, las instrucciones no se toman reiteradamente de la memoria, como ocurre en el modo normal, sino que se almacenan en la cola de preentrada *(pre-fetch queue)* y en el registro de decodificación de la CPU, y se ejecutan continuamente, sin tener que tomarlas de la memoria de nuevo. Si el modo lazo se interrumpe por una excepción, se saldrá del mismo tras retornar y ejecutar dos iteraciones más del bucle. Asi, los bucles simples, como mover un bloque de bytes, sumar una lista de números o desplazar un grupo de números, pueden ejecutarse a velocidades comparables a la de una única instrucción (por ejemplo, un movimiento de un bloque).

Los bucles que pueden entrar en modo lazo consisten en una instrucción de una palabra seguida por una instrucción DBcc. Las instrucciones permitidas se resumen en la tabla 7.3, donde lea indica una dirección efectiva de lazo: (An), −(An), (An)+, y rea indica An o Dn.

El MC68012

La única diferencia entre el MC68010 y el MC68012 es que el primero sólo puede direccionar hasta 16 MBytes de RAM (un *bus* de direcciones de 24 bits), mientras que el segundo puede direccionar bien 1024 MBytes (*bus* de 30 bits) o 2048 MBytes (*bus* de 32 bits). Para todos los demás efectos podemos considerarlos idénticos.

¿Por qué ha diseñado Motorola el MC68012? Si una aplicación necesita un *bus* de más de 24 bits de direcciones, parece apropiado emplear el MC68020, que permite emplear un *bus* de 32 bits. Hay dos razones que favorecen el empleo del MC68012: dinero y compatibilidad. El MC68020 es más caro que el MC68012, debido a las inversiones que ha habido que llevar a cabo para desarrollar muchas de sus nuevas facetas. El MC68020 no es compatible *pin* a *pin* con el resto de los anteriores miembros de la familia del 68000. Es posible desarrollar *software* para el MC68010 y más tarde

pasar a un *bus* de direcciones más ancho (30 bits) sin más que reemplazar el MC68010 por un MC68012. Un MC68020 necesitaria un nuevo zócalo.

Conclusión

A partir del MC68010, los miembros de la familia del 68000 pueden emular todas las instrucciones de otros procesadores de esta familia. Exceptuando el modo lazo, todas las nuevas características añadidas al MC68010 tienen como fin el soportar la posibilidad de estas emulaciones. Se proporciona capacidad suficiente al procesador para soportar otros tipos de emulación, como memoria virtual y máquinas virtuales.

8

El MC68020

En este capítulo se discutirán las nuevas características del MC68020 comparadas con las del MC68010. Se supone que el lector está familiarizado con las características generales de la familia del 68000, así como con las del MC68010 (capítulo 7).

Las nuevas posibilidades del MC68020 cubren un extenso campo de aplicación. Estas nuevas características incluyen un *bus* de direcciones de 32 bits y capacidades de *cache*[1] para acelerar la ejecución de las instrucciones; 7 nuevas instrucciones para comunicarse con los nuevos coprocesadores, como el coprocesador de punto flotante Motorola MC68881; 6 nuevos modos de direccionamiento para que sea más versátil, y un bit de *master* para trabajar en entornos de más de un sistema operativo.

Además, muchas de las antiguas instrucciones de propósito general se han ampliado, mientras que otras nuevas se han añadido, incluyendo 8 instrucciones para manipular campos de bits; algunos nuevos formatos para multiplicar y dividir, así como las instrucciones de desplazamiento y bifurcación, que se han ampliado.

Finalmente, el MC68020 presenta un tamaño dinámico de *bus*, lo que permite al procesador comunicarse con dispositivos de 8, 16 ó 32 bits, reali-

[1] Adoptaremos la postura de considerar *cache* como un término inducible, puesto que en este contexto tiene un significado propio, que difícilmente se puede condensar en una sola palabra. En este contexto puede traducirse *instruction cache* como acelerar una instrucción, aun cuando *cache* indica, originalmente, almacenar para su uso posterior.

zando transferencias de datos de 8, 16 ó 32 bits, en cualquier combinación e instante. Todas las restricciones de alineación de los datos se han eliminado, con excepción de aquella que establece que éstos deben encontrarse dentro de límites pares.

El microprocesador tiene ahora 120 *pines* dispuestos en la parte inferior de una estructura de planta cuadrada, en lugar de en los lados. Por tanto, actualizar un sistema incorporándole un MC68020 implica emplear un nuevo zócalo.

Instrucciones *cache*

El sistema de **instrucciones *cache***, o sistema *cache* del MC68020, es un mecanismo que acelera la ejecución de los programas con pequeños bucles. Es una característica del MC68020 que beneficiará a todos sus usuarios. Esta capacidad puede ser activada y desactivada fácilmente y no implica cambios en la forma en la que se escriben los programas, ni introduce variaciones a la hora de una ejecución normal de los mismos a cambio de sus ventajas.

Capacidades *cache* en la familia del 68000

Motorola ha empleado los resultados del análisis de sus experiencias previas para diseñar la familia 68000. Las capacidades *cache* del MC68020 son resultado de esta filosofía. Sus estudios demuestran que la mayoría de los programas escritos en ensamblador emplea la mayor parte del tiempo en bucles de pequeño y mediano tamaño. Cuando no se emplea ningún método *cache*, las instrucciones del bucle deben tomarse una y otra vez de la memoria; cuantas más veces se ejecute el bucle más veces se tomarán las mismas instrucciones de la memoria. Así es como trabaja el MC68000 y la mayoría de los microprocesadores.

El MC68010 introdujo una versión en pequeña escala del sistema *cache* denominada modo lazo. El modo lazo sólo permite un total de tres palabras de instrucciones y sólo se activa cuando las dos últimas palabras del bucle corresponden a una instrucción DBcc. En el capítulo 7 se encuentran más detalles acerca del modo lazo.

El MC68020 introduce una versión a gran escala de estas capacidades. Las instrucciones ya ejecutadas se almacenan en el propio procesador en una memoria interna de 256 bytes, es decir, una memoria *cache*[2] de 64 do-

[2] Nótese que ahora el significado correcto de *cache* no se corresponde con "acelerar", sino con "almacén", como ya se comentó anteriormente. *Cache* se refiere ahora a memoria en el sentido de memoria *hardware*, pues el *cache* es una pequeña memoria implantada en el microprocesador. A partir de ahora nos referiremos a esta acepción mediante el término "memoria *cache*", reservando para *cache* la acepción de "acelerar", y emplearemos el término "sistema *cache*" para referirnos al sistema que gestiona y controla todas estas capacidades.

TABLA 8.1

Acción del sistema cache *sobre los diferentes bloques de direcciones*

Código de función	*Bloque de direcciones*	*Control para el sistema* cache
001	Bloque de datos del usuario	No
010	Bloque de programas del usuario	Si
101	Bloque de datos del modo supervisor	No
110	Bloque de programas del modo supervisor	Si
111	Bloque de la CPU	No accede a la memoria

bles palabras de capacidad. La primera vez que se ejecuta el bucle no se obtiene ningún beneficio del sistema *cache*, cada instrucción se toma de la memoria, como se haría en un MC68010. A partir de la segunda ejecución del bucle comienzan los beneficios, pues el sistema *cache* detecta que las instrucciones están aún en la memoria *cache* y no intenta tomarlas de la memoria. El resultado neto es una ejecución más rápida del programa.

Funcionamiento del sistema *cache*

Se discutirá ahora el funcionamiento del sistema *cache*, pero antes es necesaria alguna preparación. Deberá revisar la discusión acerca de los códigos de función que se mantuvo en el capítulo 7. En esta discusión establecimos que una referencia a una determinada posición de memoria implicaba un total de 35 bits, de los cuales 3 constituían un código de función. La misión de este código es la de decidir de cuál de los bancos de memoria debe tomarse la dirección que indican los 32 bits restantes. Los cinco bancos de direcciones empleados en el 68000 y su uso en los sistemas *cache* se esboza en la tabla 8.1.

Nótese que las instrucciones sólo se toman de la memoria *(fetch)* cuando se emplean los códigos de función 010 ó 110; en general, podemos establecer que la memoria sólo se ve involucrada, en lo que a las instrucciones se refiere, cuando los códigos de función toman la forma f10. Los accesos a las demás zonas de memoria (con otros códigos de función) no se refieren a instrucciones y, por tanto, no se someten al sistema *cache*. Cuando el MC68020 toma de la memoria una palabra que contiene una instrucción, se carga en éste la doble palabra que la contiene. Por tanto, los dos últimos bits de la doble palabra que se cargan son siempre cero, de modo que podemos imaginar los 35 bits que se han empleado como:

Código de función (3 bits)	*Direcciones de la memoria* hard *(32 bits)*
f 1 0	ttttttt ttttttt ttttttt iiiiii00

TABLA 8.2a

Instrucciones del programa inicial

Instrucción	*Hex*
001000F4	aaaa
001000F6	bbbb
001000F8	cccc
001000FA	dddd
001000FC	eeee
001000FE	ffff
00100100	gggg
00100102	hhhh

TABLA 8.2b

Contenido de la memoria cache *al comienzo de la ejecución*

Indice	*Bit de validación*	*FC2*	*Tag*	*Datos*
00	0			
04-F4	0			
F4	0			
F8	0			
FC	0			

Los 24 bits t se denominan **tag del *cache*** y los 6 bits i se denominan **índice de *cache*.** El índice de *cache* determina cuál de las 64 posiciones de la memoria *cache* ocupará la doble palabra. Si dos instrucciones tienen el mismo índice de *cache,* sólo una de ellas podrá estar en el mismo en un determinado momento. Así, pues, dos instrucciones que se encuentren separadas 256 bytes exactamente (o un múltiplo de 256) en la memoria no podrán coexistir en la misma memoria *cache*. Nótese que este método de asignación de espacio en la memoria *cache* satisface dos criterios fundamentales: su ejecución es simple (y, por tanto, rápida) y garantiza que cualquier grupo de hasta 64 dobles palabras cabrá en la memoria *cache* en un momento dado.

Cuando el sistema *cache* actúa sobre una instrucción mantiene cinco parámetros para cada doble palabra de instrucciones:

Indice del *cache:* Definido por las direcciones [7:2] = 6 bits de la dirección de la doble palabra.

Tag del *cache:* Direcciones [31:8] = los 24 bits más significativos de la dirección de la doble palabra.

Cache FC2: Segundo bit del código de función (1 para el bloque supervisor y 0 para el bloque de usuario).

Bit de validación: 1 si el *cache* es válido y 0 si no lo es.

Datos del *cache:* Contenido de la posición de la memoria si el bit de validación es 1 (el contenido permanece indefinido si el bit de validación es 0).

El índice del *cache* es un número de 0 a 63 y define en cuál es las 64 posiciones de la memoria *cache* hay que trabajar. Las otras cuatro cantidades se almacenan en esa posición de la memoria *cache*.

Ejemplos de *cache*

Cuando se enciende un sistema basado en el MC68020, el procesador pone a cero todos los bits de validación. La tabla 8.2a muestra un programa ejemplo al arrancar y la tabla 8.2b muestra el estado inicial del sistema *cache*. Nótese que las instrucciones almacenadas en las posiciones indicadas en la tabla 8.2a están representadas, de forma figurada, por los números hexadecimales $AAAA, $BBBB, ..., $FFFF.

Las primeras instrucciones que se ejecutan no están sometidas al sistema *cache* y se toman todas de la memoria. Cada instrucción ejecutada se almacena en la memoria *cache* poniendo su bit de validación a 1 para indicar que hay datos válidos en la memoria *cache*. Además se almacenan los restantes datos relativos a cada instrucción *cache tag*, *cache* FC2 y datos del *cache*.

En la tabla 8.3a las instrucciones que se encuentran en las posiciones hexadecimales 001000F6 a 00100100 se han ejecutado como se indica mediante letra negrita. La tabla 8.3b muestra cómo la memoria ***cache*** ha sido convenientemente actualizada, es decir, que el bit de validación se ha puesto a 1 para indicar la presencia de datos válidos y los demás parámetros del

TABLA 8.3a

Instrucciones del programa después de que se han ejecutado seis de ellas

Dirección	*Hex*
001000F4	aaaa
001000F6	**bbbb**
001000F8	**cccc**
001000FA	**dddd**
001000FC	**eeee**
001000FE	**ffff**
00100100	**gggg**
00100102	hhhh

TABLA 8.3b

Contenido de la memoria cache *después de que se han ejecutado instrucciones*

Indice	*Bit de validación*	*FC2*	*Tag*	*Datos*
00	1	0	001001	gggghhhh
04-F4	0			
F4	1	0	001000	aaaabbbb
F8	1	0	001000	ccccdddd
FC	1	0	001000	eeeeffff

sistema *cache: cache* FC2, *cache tag* y datos del *cache,* se han actualizado también. Nótese que el sistema *cache* toma la doble palabra que se encuentra en la posición hexadecimal 001000F4, aunque sólo se empleará la palabra menos significativa. Esto se debe a que la memoria *cache* está estructurada en dobles palabras, de modo que al leer los datos como si de dobles palabras se tratara se asegura que estarán justamente en los márgenes. De la misma forma, el sistema *cache* toma la doble palabra, completa, que se encuentra en la posición $00100100, aunque sólo necesita la palabra de orden más bajo. Nótese también que los dos últimos dígitos del índice del *cache* son ahora 00 en lugar de FF, el valor que tenian cuando actuaban como los dos dígitos finales de la dirección de la memoria, debido al particular método que tiene el sistema de asignar posiciones dentro de la memoria *cache.*

Si una instrucción que se encuentra en la memoria *cache* debe ejecutarse de nuevo, se produce una señal. Cuando esta señal aparece, se toman los datos del *cache* en lugar de buscarlos en la memoria, con lo que no se necesita emplear ciclos del *bus* externo. Por ejemplo, en la figura 8.2, si la interrupción representada por los dígitos $ggg resulta ser un salto a la instrucción con el código $aaaa, que se encuentra en la posición 001000F4, se producirá una señal, porque esa instrucción está ya en la memoria *cache.* Nótese que la instrucción con el código $aaaa no se ha ejecutado nunca realmente, pues se cargó en la memoria *cache* a la vez que la instrucción $bbbb, ya que ambas forman parte de la misma doble palabra. Estrictamente hablando, se produce una señal cada vez que el índice del *cache,* el *cache tag* y el *cache* FC2 de una instrucción que se busca en la memoria coinciden con alguno de los valores previamente almacenados en la memoria *cache.*

Registros del sistema *cache*

Para soportar el sistema *cache* se han añadido dos nuevos registros de control al MC68020, denominados registros del control del sistema *cache*

(CACR) y el registro de direcciones del sistema *cache* (CAAR). Ambos son registros de 32 bits, aunque sólo 4 bits de CACR y 6 del CAAR están actualmente en uso. Además, la instrucción MOVEC se ha revisado para permitir el acceso a estos registros.

El CACR contiene 4 bits que permiten controlar el funcionamiento general del sistema *cache*. Excepto por el control que estos registros pueden ejercer sobre el sistema *cache*, éste es automático e inaccesible. Los 4 bits son:

Bit 1: Activa el sistema *cache* (E)
Bit 2: Congela el sistema *cache* (F)
Bit 4: Pone a cero las entradas del *cache* (CE); emplea el CAAR
Bit 8: Desactiva el sistema *cache*

Si el bit E está a 0, no se aplica el sistema *cache*, todas las instrucciones se toman de la memoria. Al encender el sistema, el bit E se pone a 0, de modo que debe ser puesto a 1 antes de que el sistema *cache* pueda entrar en acción; para ello se emplean las instrucciones:

```
MOVE.L  #1,D0       Asignar valores al bit E
MOVEC   D0,CACR     Poner el bit E a 1
```

Si el bit F está a 0, el sistema *cache* funciona como se ha descrito anteriormente; cuando el bit F se pone a 1, la memoria *cache* funciona como memoria de sólo lectura. Es decir, las señales se procesan como anteriormente, pero las nuevas instrucciones que no se encontraban ya en la memoria *cache* no entran en la misma. Esto puede ser interesante durante las emulaciones, cuando el programador desea que una determinada rutina de emulación no entre en la memoria *cache*. Puede emplearse para optimizar los resultados del sistema *cache*. Uno de estos casos se discute en la sección siguiente. Se asignan valores al bit F, según el siguiente ejemplo:

```
MOVE.L  #3,D0       Asignar valores a los bits E y F
MOVEC   D0,CACR     Poner los bits E y F a 1
```

O, también:

```
MOVEC   CACR,D0     Se leen los valores actuales del CACR
ORI     #2,D0       Se pone el bit F a 1 sin cambiar los demás bits
MOVEC   D0,CACR     Actualizar el CACR
```

Si leemos el bit C siempre lo encontraremos a 0. Sin embargo, si el bit C se encuentra a 1 pondrá toda la memoria *cache* a 0. Se puede poner a 1 el bit C de la forma siguiente:

```
MOVE.L  #5,D0       Asignar valores a los bits C y E
MOVEC   D0,CACR     Actualizar al CACR
```

El bit CE es similar al bit C, excepto que sólo pone a 0 una de las entradas de la memoria *cache*. Si leemos, el bit CE siempre se encontrará a 0. Si el bit CE se pone a 1, la posición de la memoria indicada en el CAAR se pone a 0. Esta posición la proporciona el índice del *cache* (bits [2:2]) del CAAR.

Limitaciones del sistema *cache*

El sistema *cache* es simple a la hora de la ejecución y de un tamaño moderado. Por tanto, hay situaciones en las que el programador debe tener en cuenta las limitaciones del sistema. En las situaciones descritas a continuación se supondrá que, cuando el sistema *cache* está activo, la velocidad de ejecución será al menos tan grande como cuando está inactivo. Por tanto, habilitar el sistema sólo se traduce en un incremento de la velocidad de ejecución. A continuación expondremos estas limitaciones.

Primera: Acelerar las instrucciones mediante el sistema *cache* puede no dar resultados cuando se trata de bucles largos. La memoria *cache* está limitada a 256 bytes. Si un bucle ocupa más de 256 bytes y se ejecuta muchas veces, la memoria *cache* no podrá albergar todas sus instrucciones, de modo que muchas de ellas tendrán que leerse repetidamente de la memoria.

Segunda: Las rutinas que se usan tanto en modo supervisor como en modo usuario tendrán que tomarse de nuevo en la memoria, incluso aunque ya estén en la memoria *cache*. El sistema *cache* considera que los accesos a la memoria en modo usuario son diferentes a los accesos a la memoria en modo supervisor. En una situación de *hardware* típica (como se explicó en el capítulo 7), todos los bloques de direcciones se refieren a la misma memoria *hardware*. En otros entornos puede haber cuatro bloques diferentes de direcciones. Puesto que todo esto se determina fuera del MC68020, éste no tiene forma de saber qué es lo que está sucediendo. Por tanto, tiene que asumir el peor caso, es decir, aquel en el que las memorias de programa de modo usuario y supervisor se encuentran en distintos bloques.

Si se accede a una instrucción en modo supervisor (código de función 110) e inmediatamente después se accede a la misma instrucción en modo usuario (código de función 010), el MC68020 no tiene forma de saber si las memorias de programa del modo supervisor y usuario se encuentran en el mismo bloque *hardware* y, por tanto, tiene que modificar el *cache*.

Tercera: El sistema *cache* no controla los accesos a las memorias de datos. Supongamos, por ejemplo, que se ejecutan las siguientes instrucciones:

```
MOVE.L  #3(A0)
MOVE.L  (A1),(A2)
```

El sistema *cache* controla las dos instrucciones, incluyendo el campo inmediato. Sin embargo, no controla las referencias a los bancos de datos. Por tanto, si estas instrucciones se ejecutan de nuevo, los accesos a los datos (A0) se volverán a efectuar. Esta situación está justificada por diversas razones. Una de ellas es que, para acceder a los bancos de datos, se duplicaría el espacio utilizado. Otra razón estriba en el hecho de que las áreas de datos están sujetas a cambios, y un funcionamiento correcto obligaría al sistema *cache* a controlar tanto las entradas como las salidas.

Cuarta: La memoria *cache* debe ponerse a cero en determinados instantes críticos. Por ejemplo, si se carga un programa en la memoria y el contenido previo de ésta se encuentra aún en la memoria *cache*, es necesario poner a cero la memoria *cache* (o al menos desactivarla). De otro modo se producirían falsas señales, que darían lugar a resultados desastrosos.

Quinta: El sistema *cache* puede fallar en sus intentos de acelerar varios bucles pequeños dentro de un programa. Consideremos una rara pero posible situación donde parte del bucle A reside entre las posiciones de memoria $xxxxxx00 y $xxxxxx7F y parte del bucle B reside entre las oposiciones $yyyyyy00 y $yyyyyy7F. Nótese que cada segmento ocupa 32 dobles palabras, es decir, la mitad de la memoria *cache;* sin embargo, ambas se almacenarán en la misma porción de la memoria *cache*, debido a las formas que tiene el sistema *cache* de asignar el espacio. Así, los bucles A y B se alternan en la memoria *cache*, y el sistema *cache* no producirá ningún efecto. En este momento el bit F viene al rescate. Si el bit F del CACR se pone a 1 tras la ejecución del bucle A, entonces el bucle B no obtendrá ningún beneficio, pero el bucle A sí. ¡Menos da una piedra!

Sexta: Si un programa que se modifica durante su ejecución se almacena en la memoria *cache* pueden obtenerse resultados erróneos. Los programas "automodificantes" están en contra de la filosofía de diseño de la familia de 68000, de modo que este problema no debe sorprendernos. Si una instrucción que entra en la memoria *cache* se modifica más tarde, en un programa automodificante, cambiará en la memoria, pero no en la memoria *cache*. En caso de una llamada posterior a dicha instrucción, se ejecutará la versión antigua, almacenada en la memoria *cache*, en lugar de la nueva, que se encuentra en la memoria externa. El problema aquí consiste en que el programador está tratando su programa como si de datos de salida se tratarán, y el sistema *cache*, ya lo hemos dicho, no está pensado para controlar las salidas de datos.

Como ejemplo, el siguiente programa halla el primer código de condición que es positivo en la comparación entre D0 y D1. Luego se modifica e inicia un bucle empleando la instrucción modificada.

```
BUCLE  CMP.L    D0,D1       Los dos registros permanecen inalterados
TEST   BHI      FIN         Esta instrucción se cambia a LS, CC, CS, etc.
       LEA      TEST,A0     Hallar la dirección de la instrucción de prueba
       ADD.W    #$100,(A0)  Se cambia el código de condición anterior
       BRA      BUCLE       Se realiza otra comparación y otra prueba
```

```
FIN     MOVE.W  (A0),D2     Se toma la instrucción Bcc
        ASR     #$8,D2      Se aisla el código de condición
        AND     #$F,D2      Se deja todo como estaba, excepto
                            los últimos 4 bits
```

Si el sistema *cache* está activo, la instrucción Bcc se ejecutará como si de un BHI se tratara. Si no está activo, no hay problema. La mejor solución consiste en no emplear programas automodificantes. Una solución muy elaborada podría consistir en sustituir la instrucción que comprueba cada uno de los 16 códigos de condición por 16 instrucciones separadas. Un poco de espacio extra resolvería un difícil problema. Otra solución, un poco peor en la lista de posible, consistiría en desactivar el sistema *cache* e impedir el acceso de otros usuarios mientras se ejecute esta peliaguda rutina. Una vez finalizada, se reactiva el sistema *cache* y se permite de nuevo el acceso de otros usuarios.

Nuevos modos de direccionamiento

El MC68020 introduce varios modos adicionales de direccionamiento que permiten más y más largos desplazamientos, un factor de escala y un nivel adicional de indirección. El programador principiante difícilmente encontrará situaciones (si es que las encuentra) en las que estos modos de direccionamiento sean útiles y acabará concluyendo que simplemente ahorran una instrucción y que por este motivo se han implantado. A medida que las situaciones que el programador tiene que resolver se van complicando, se encuentra una mayor utilidad para estos nuevos (y más complejos) modos de direccionamiento. Esperamos que los ejemplos expuestos en esta sección ayuden a comprender lo anteriormente expuesto.

Los nuevos modos de direccionamiento implantados se resumen en 6 variantes de 2 de los 12 modos de básicos del 68000. Tres son variantes del modo 110, conocido originalmente como direccionamiento indirecto con índice y desplazamiento de 8 bits; los otros tres son variantes del modo 110 011, conocido originalmente como direccionamiento indirecto por contador de programa con índice y desplazamiento de 8 bits. Estos dos conjuntos de variantes se implantan de forma paralela. Debido a este paralelismo, sólo necesitamos discutir en profundidad uno de estos modos, el otro es totalmente análogo.

Debido a la complejidad de estos modos de direccionamiento, debemos aclarar dos puntos antes de continuar.

Primero: Los nuevos modos de direccionamiento suman varios números, entre los que se incluyen números de 8, 16 y 32 bits, con y sin signo. En esta sección se supondrá que cuando se suman números de 8 y 16 bits se realiza primero una extensión de signo a formato de 32 bits. Esto se aplicará a todos los campos, así se trate de valores inmediatos, registros de los que sólo se emplea una palabra o byte o posiciones de la memoria.

Segundo: Los ensambladores disponibles para los diferentes ordena-

dores pueden presentar pequeñas variantes en la sintaxis de los modos de direccionamiento. Esto es especialmente cierto para los modos de direccionamiento que se describen a continuación, debido a los múltiples parámetros y operaciones involucradas. Estas diferencias sintácticas deben, sin embargo, ser de pequeña importancia y fácilmente traducibles de un ensamblador a otro.

En las próximas cinco secciones describiremos la forma original de uno de los modos de direccionamiento del MC68000, el direccionamiento indirecto con índice y desplazamiento de 8 bits, y las tres variantes que de este modo están disponibles en el MC68020. Para ilustrar el modo de direccionamiento original y sus variantes, emplearemos varios ejemplos (en realidad sólo trataremos con la dirección efectiva que se emplearía) construidos en torno al uso de tablas de conversión ASCII-EBCDIC. A medida que se compliquen las variantes se complicarán los ejemplos.

Direccionamiento indirecto por registro y memoria con índice

Este modo de direccionamiento se ajusta a la forma 110 rrr, donde rrr indica un número de 3 bits de un registro de direcciones (A0...A7). El nombre de direccionamiento indirecto por registro y memoria con índice se refiere a las cinco variantes existentes en el MC68020, incluyendo la forma original que se encontraba en el MC68000 y las tres variantes disponibles en el MC68020.

Como ya se ha mencionado anteriormente, la forma original de este modo disponible en el MC68000 se denominaba direccionamiento indirecto con índice y desplazamiento de 8 bits. Se representa por (d8,An,Rn.SIZE), donde:

d8	indica cualquier desplazamiento de 8 bits con signo (valores desde −128 a +127)
An	cualquier registro de direcciones
Rn	cualquier registro de direcciones o datos
SIZE	un código de tamaño que puede indicar bien una palabra, bien una doble palabra

La dirección efectiva se obtiene sumando d8, An y Rn.SIZE. Nótese que tanto d8 como Rn.SIZE sufren una extensión de signo antes de la suma.

Un ejemplo útil de la forma original se obtiene considerando (0,A0, D0.W). Si A0 contiene la base de una tabla de conversiones ASCII-EBCDIC y D0 contiene un byte ASCII, entonces (0,A0,D0.W) contiene la dirección efectiva del correspondiente byte EBCDIC.

Factor de escala

El MC68020 permite la inclusión de un factor de escala en la dirección efectiva, representado por (d8,An,Rn.SIZE*SCALE)[3], donde SCALE toma los valores 1, 2, 4 u 8.

La dirección efectiva se obtiene de forma similar a como se obtiene la forma original. La dirección efectiva se obtiene sumando d8, An y Rn.SIZE*SCALE. Nótese que tanto d8 como Rn.SIZE*SCALE sufren una extensión de signo antes de la suma.

Un ejemplo del uso del factor de escala se obtiene empleando (0,A0,D0.W*2). Como en el caso anterior, asumiremos que A0 contiene la base de una tabla de conversiones ASCII-EBCDIC. Ahora, sin embargo, la tabla contiene dos bytes por cada entrada: el primer byte indica si la conversión en el sentido ASCII-EBCDIC es posible (indicador de conversión) y el segundo indica el código EBCDIC correspondiente, cuando procesa. Así, si D0 contiene un byte ASCII, entonces (0,A0,D0.W*2) contiene la dirección efectiva del indicador de conversión y (1,A0,D0.W*2) la del byte EBCDIC.

En la forma original del MC68000 y en la forma escalada del MC68020, d8 puede tomar un valor nulo, pero d8, A0 y Rn deben estar presentes. Esto contrasta con las tres variantes descritas a continuación, en las que todos los registros y desplazamientos son opcionales.

Variante #1

Esta variante se denomina direccionamiento indirecto por registro con índice y desplazamiento de la base y se representa por (bd,An,Rn.SIZE*SCALE), donde bd es el desplazamiento de la base, de 0, 16 ó 32 bits.

Los tres parámetros son opcionales. Este carácter opcional es conveniente cuando uno de estos parámetros no se necesita y no se encuentra un registro libre para asignarle un valor nulo a la hora de calcular la dirección efectiva.

La dirección efectiva en esta variante se evalúa de forma similar a como se hace en la forma original. Si uno de los tres parámetros no está presente se le asigna un valor cero. La dirección efectiva se obtiene sumando bd, An y Rn.SIZE*SCALE. Rn.SIZE*SCALE sufre una extensión de signo antes de la suma.

Un ejemplo del uso de esta variante se obtiene considerando (des1,A0,D0.W*2). Esta vez supondremos que A0 es la base de una zona de datos cualquiera que contiene varias tablas, una de las cuales es nuestra vieja conocida tabla de conversión ASCII-EBCDIC. Si des1 es la distancia de la base de la tabla de conversión respecto a la base de la zona de datos y D0 contiene un byte ASCII, entonces (des1,A0,D0.W*2) y (des1 + 1,A0,D0.

[3] Hemos optado por no traducir los términos ingleses SIZE (tamaño) y SCALE (escala), pues forman parte de la sintaxis de un modo de direccionamiento del cálculo de la dirección efectiva.

W*2) contienen las direcciones efectivas del indicador de conversiones y del byte EBCDIC (como se indicó en el ejemplo del factor de escala).

Nótese que el modo de direccionamiento indirecto por registro de datos (Dn) se puede generar empleando esta variante. Esto se consigue omitiendo bd y An, y usando un registro de datos para Rn.

Variante #2

Esta variante se denomina direccionamiento posindexado indirecto por memoria, y se representa por ([bd,An],Rn.SIZE*SCALE,od), donde od indica un desplazamiento exterior de 0, 16 ó 32 bits.

La evaluación de la variante #2 es similar a la de la variante #1, excepto que se emplea un nivel extra de indirección en mitad del cálculo de la dirección efectiva. Los cuatro parámetros bd, An, Rn y od son opcionales. Si un parámetro no está presente, se le asigna un valor cero. La dirección efectiva se evalúa sumando primero los valores de bd y An y tomando, de la memoria, el contenido de la doble palabra a la que apunta la suma de ambas. Finalmente, se obtiene la dirección efectiva sumando esta doble palabra, Rn.SIZE*SCALE y od.

Veamos un ejemplo de esta variante empleando ([des1,A0],D0.W*2). Esta vez supondremos que las tablas de datos no están todas en el mismo sitio, sino distribuidas por doquier. Conocemos la dirección de cada una de las bases de cada tabla y, además, todas ellas se encuentran en una tabla maestra cuya base está almacenada en A0. La cantidad des1 indica el desplazamiento que hemos de efectuar en esta tabla maestra para hallar un puntero que nos lleve a la tabla de conversión deseada. La expresión [dis1, A0] nos lleva en realidad a la base de la tabla de conversión adecuada, mientras que ([dis1,A0],D0.W*2) es la dirección efectiva del valor EBCDIC correspondiente al byte ASCII almacenado en A0, si consideramos una tabla de conversión de dos entradas.

Variante #3

La tercera de las variantes se denomina direccionamiento preindexado por memoria, y se representa por ([bd,An,Rn.SIZE*SCALE],od).

Esta variante se evalúa de forma similar a como se hace con la variante #2, sólo que el nivel adicional de indirección tiene lugar en otro momento.

Los cuatro parámetros bd, An, Rn y od son opcionales. Si un parámetro no está presente, se le asigna un valor cero. La dirección efectiva se evalúa sumando primero los valores de bd, An y Rn.SIZE*SCALE y tomando, de la memoria, el contenido de la doble palabra a la que apunta la suma anterior. Finalmente se obtiene la dirección efectiva sumando esta doble palabra y od. Nótese que las variantes #2 y #3 difieren sólo en el momento en que el registro indice se suma antes (variante #3) o después (variante #2) de la referencia de la memoria.

Direccionamiento indirecto por contador de programa y memoria con desplazamiento e índice

Este modo de direccionamiento emplea el código 111 011. El nombre de direccionamiento indirecto por contador de programa registro y memoria con indice se refiere a las cinco variantes existentes en el MC68020, incluyendo la forma original que se encontraba en el MC68000, y las tres variantes disponibles en el MC68020.

La discusión precedente, referente al modo de direccionamiento indirecto por registro y memoria con indice y a sus tres variantes, se aplica por completo a este modo de direccionamiento y a sus variantes. La única diferencia entre ambos modos de direccionamiento estriba en que la segunda, es decir, la que actualmente discutimos, emplea el contador de programa (PC) en lugar de un registro de direcciones (An). Para aplicar lo anterior a este modo de direccionamiento basta con reemplazar "registro de direcciones" por "contador de programa", "An" por "PC" y "direccionamiento indirecto por memoria" por "direccionamiento indirecto por memoria relativo al PC".

Nótese que en cada uno de los ejemplos anteriores la tabla de conversión ASCII-EBCDIC podia encontrarse en cualquier lugar de la memoria. Si la tabla está localizada dentro del propio programa, es mejor emplear el direccionamiento relativo al PC (etil,PC,D0.W*2) en lugar del modo (desl. PC,D0.W*2). El segundo modo nos priva del registro A0, al que, además, hay que asignarle valores mediante una instrucción "LEAetil1,A0".

Las tablas 8.4 y 8.5 resumen los nuevos modos de direccionamiento y la sintaxis de sus direcciones efectivas.

Los bits de traza T0 y T1

Los bits de traza permiten que un programa controle a otro. Asi es posible que un programa maestro, P-1, controle la ejecución de otro programa esclavo, P-2, instrucción a instrucción. Esto se hace empleando el bit de traza T1, que es el bit 15 del registro de estatus. Cuando este bit se pone a 1, se produce una excepción de traza cada vez que termina una instrucción. Asi, P-1 debe poner a 1 el bit T1 empleando una instrucción privilegiada y comenzar la ejecución del programa P-2. Cada vez que se ejecuta una instrucción del programa P-2, el 68000 genera una excepción y retorna el control a P-1, que analiza lo sucedido y devuelve el control a P-2 con un RTE. El proceso se repite hasta que P-1 lo detiene.

Los bits de traza permiten crear programas capaces de efectuar un control exhaustivo de los efectos de cada una de las instrucciones. Estos programas incluyen facilidades de ensamblado y depuración, asi como programas que detectan la frecuencia de ejecución de cada instrucción.

En el MC68000, el bit de traza T0 no se emplea y está siempre a 0; además, el único bit de traza es el bit T1, que se denomina bit T. El funciona-

TABLA 8.4

Direccionamiento indirecto por registro con índice y memoria: Modo 110 rrr

Formato	*CPU*	*Sintaxis de la dirección efectiva*	*Parámetros*
Original	68000	(d8,An,Rn.SIZE)	Requeridos
Original	68020	(d8,An,Rn.SIZE*SCALE)	Requeridos
Variante #1	68020	(bd,An,Rn.SIZE*SCALE)	Opcionales
Variante #2	68020	([bd,An],Rn.SIZE*SCALE,od)	Opcionales
Variante #3	68020	([bd,An,Rn.SIZE*SCALE],od)	Opcionales

TABLA 8.5

Direccionamiento indirecto por registro con índice y memoria: Modo 110 011

Formato	*CPU*	*Sintaxis de la dirección efectiva*	*Parámetros*
Original	68000	(d8,PC,Rn.SIZE)	Requeridos
Original	68020	(d8,PC,Rn.SIZE*SCALE)	Requeridos
Variante #1	68020	(bd,PC,Rn.SIZE*SCALE)	Opcionales
Variante #2	68020	([bd,PC],Rn.SIZE*SCALE,od)	Opcionales
Variante #3	68020	([bd,PC,Rn.SIZE*SCALE],od)	Opcionales

TABLA 8.6

Bits de traza

T1	*T0*	*Funciones de traza*
0	0	Modo traza inactivo
0	1	Se activa el modo traza en las instrucciones que afectan al control del flujo (Bcc,JMP,DBcc)
1	0	Modo traza activo para todas las instrucciones
1	1	(Reservado por Motorola)

miento de los bits de traza es, pues, compatible, es decir, los programas de MC68000 que empleen funciones de traza funcionarán correctamente en el MC68020. La única excepción se planteará si algún programador se ha dedicado a jugar con el bit T0 en el MC68000: los programas que usen este bit no se ejecutarán correctamente en el MC68020.

Funcionamiento con coprocesador

El MC68020 tiene siete nuevas instrucciones para controlar las comunicaciones entre él mismo y sus coprocesadores. Los coprocesadores son procesadores que cumplen determinados requerimientos *hardware* especificados por Motorola. Uno de los requerimientos básicos es que el coprocesador debe tener ciertos registros para comunicarse con el procesador principal.

Una discusión completa de los coprocesadores incluye la descripción detallada de cada una de las instrucciones del 68000, de los requisitos *hardware* de interconexión y de las funciones particulares que cada coprocesador ofrece. En esta sección nos limitaremos a dar una visión general de las instrucciones de comunicación con los coprocesadores del 68000, al tiempo que mostramos cómo tienen lugar, realmente, estas comunicaciones. El propósito de esta sección es el de aclarar, al menos desde el punto de vista del programador, la forma en que el 68000 intercambia instrucciones y datos con los coprocesadores. Consideraremos también la situación que se presenta cuando un coprocesador no se encuentra físicamente presente en el sistema y es emulado por *software* hasta el momento en que se instale el dispositivo *hardware*.

Si los párrafos siguientes resultan oscuros, revise la sección de los códigos de función del capítulo 7.

Comunicaciones con los coprocesadores: Aspectos *hardware*

Veamos, en primer lugar, lo que ocurre a nivel *hardware*. Para entender correctamente las instrucciones de comunicación coprocesadoras del MC68020, un coprocesador debe tener un conjunto de 13 registros de interfaz estándar (registro de control, de órdenes, de condición, etc.) con 32 bytes en total. Emplearemos uno de estos registros, el registro de órdenes, en los ejemplos que siguen. Este registro se encuentra siempre en la posición 10 (decimal) en los 32 bytes. Cuando el 68000 se comunica con un coprocesador, lee o escribe en uno o más de los registros del coprocesador. Para lograr este propósito, el 68000 envía el código 111 por las tres líneas de códigos de función, indicando un bloque de memoria reservado para la CPU. Las 32 líneas de direcciones del 68000 envían 32 bits en el formato:

xxxx xxxx xxxx 0010 cccx xxxx xxxr rrrr

las 32 líneas de datos del 68000 envían 32 bits de datos.

Cuando el código de función es el 111, los 32 bits de las líneas de direcciones no se interpretan como una dirección de 32 bits (como en el caso de otros códigos de función), sino que se parte esta información en trozos más pequeños, que se emplean para determinar la posición final del bloque reservado para las funciones I/O de la CPU. De este modo, el código 0010

indica que nos encontramos ante una transferencia de información hacia/desde un coprocesador (en contraposición con otras transferencias de información al bloque de la CPU), cc es el código del coprocesador (de 0 a 7) que indica a qué coprocesador estamos accediendo y rrrrr es el registro de direcciones del coprocesador (de 0 a 31, en decimal). Las x indican bits que no se emplean hoy en dia.

Los códigos de coprocesador actualmente disponibles son:

000	MC68851, unidad de manejo de memoria paginada
001	MC68881, coprocesador de punto flotante
001-101	(Reservado por Motorola para su uso)
110-111	(Reservado para los usuarios)

Es necesario, para establecer una comunicación eficaz, que el procesador, o cualquier otro dispositivo *hardware* intermedio, pueda detectar cuándo salen las señales anteriormente descritas del 68000, para, al reconocerlas como señales de comunicación con un coprocesador, poder responder adecuadamente.

Si el 68000 está intentando enviar información al coprocesador, la respuesta adecuada del coprocesador será la de recibir los datos que le envian (a través de las 32 líneas de datos del 68000) y almacenarlas en el coprocesador ccc en el registro rrrrr. Si el 68000 está intentando leer información del coprocesador ccc, éste debe enviar al 68000 el contenido del registro rrr por las líneas de datos. Esto cubre el aspecto *hardware* de las comunicaciones con los procesadores. Vamos a ver ahora cómo el programador puede enviar los códigos de función y los bits adecuados de direcciones para acceder al coprocesador deseado.

Comunicaciones con los coprocesadores: Aspectos *software*

¿Cómo se comunica el programador normal con un coprocesador? En general, se empleará un ensamblador que soporte el coprocesador en cuestión. Se incluirán entonces en el programa algunas instrucciones del coprocesador, para comunicarse con él siguiendo las normas de la documentación del coprocesador. Así, finalmente, el ensamblador generará el código objeto adecuado con los parámetros del coprocesador. Nótese que la misma clase de comunicaciones con diferentes coprocesadores dará lugar a diferentes códigos objetos, debido a los diferentes códigos de identidad de los coprocesadores y dado que cada uno de ellos tiene su propio conjunto (lenguaje) de comandos.

Por ejemplo, supongamos un coprocesador denominado MC99999 que controla el consumo de potencia controlando cientos de medidores. El MC68020 se comunica con el MC99999 inicialmente para arrancarlo y el coprocesador emplea varios segundos o minutos en la tarea de control, re-

velando al MC68020 de la pesada carga. El MC99999 deja los resultados en la memoria e indica al MC68020 que ha terminado la misión que se le encomendó.

Una de las instrucciones de comunicación con los procesadores del MC68020 es la instrucción cpGEN. Esta instrucción envía un comando de 16 bits a un registro del coprocesador. Supongamos que el MC99999 tiene como código de procesador el número 2 y que la recepción de una instrucción de 16 bits, denominada POLL, en su registro de órdenes da lugar a que el MC99999 comience una sesión de medidas. Supongamos que en el lenguaje del MC99999 la instrucción POLL tenga un código $1234. Un ensamblador hipotético, que soportará el MC99999, aceptaría la instrucción:

```
cpGEN  METER,POLL
```

Nuestro hipotético ensamblador habría asociado el símbolo "METER" con el código del MC99999 (es decir, 2) y ensamblaría el código del comando POLL, asignándole el código binario adecuado (hex $1234). Una instrucción cpGEN se ensambla en dos palabras, que generalmente se representan como:

1	1	1	1	C	c	c	0	0	0	X	x	x	x	x	x
I	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i

Donde Ccc es el código del procesador, Xxxxxx es una dirección efectiva opcional (que no se emplea en el MC99999) y la segunda palabra es, realmente, la instrucción enviada al coprocesador. Entonces, la instrucción cpGEN queda, una vez ensamblada:

1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0

Ahora que hemos estudiado los aspectos *software* y *hardware* del funcionamiento con un coprocesador, vamos a ver qué pasa cuando falta alguno de estos elementos.

Emulación de las instrucciones de manejo de coprocesadores del MC68020

¿Es indispensable tener un MC68020 para acceder a los coprocesadores? No, simplemente es más fácil, más rápido y usa menos registros que el MC68010, pero no es indispensable.

La instrucción cpGEN del MC68020 discutida en la sección anterior es equivalente a las siguientes instrucciones del MC68010:

```
MOVE.L  #7,D0           Código de función del bloque de la CPU
MOVEC   D0,DFC          Movemos el código de función al registro
                        de códigos de función
MOVE.L  #$0002400A,A0   Ponemos el ccc = 2 y el rrrrr = 10 (decimal)
MOVE.L  #$1234,D1       Escogemos los datos a mover
MOVES   D1(A0)          Y movemos D1 al registro 10 del coprocesador 2
                        (decimal)
```

Nótese que en el MC68020 todo lo anterior se consigue con una sola instrucción, y sin emplear los registros D0, D1 y A0.

Puesto que todas las instrucciones anteriores forman parte del juego de instrucciones del MC68010, el aspecto *software* de las comunicaciones con un coprocesador puede emularse en el MC68000. Discutiremos ahora las capacidades de que están dotados el MC68010 y el MC68000 para comunicarse con los coprocesadores.

Coprocesadores y el MC68010

Si se conecta un MC99999 a un MC68010, es necesario emular las instrucciones de comunicación con los coprocesadores de que dispone el MC68020, empleando las instrucciones del MC68010.

Una solución más elegante sería, sin embargo, la siguiente: incluir las instrucciones de manejo de coprocesador en el programa, aunque éstas no formen parte del conjunto de instrucciones del MC68010; cambiar el ensamblador para que ensamble las instrucciones de la misma forma que lo haría un ensamblador del MC68020, o emplear un ensamblador para este último procesador, y, finalmente, cambiar las rutinas de excepciones para que al detectar el código 1111, no implementado, lo emule por *software*.

Coprocesadores y el MC68000

Desgraciadamente, el MC68000 es totalmente incapaz de comunicarse con los coprocesadores, no sólo porque carece de las instrucciones específicas del MC68020, sino también porque carece de la instrucción MOVEC y MOVES del MC68010. Lo mejor que el MC68000 puede llegar a hacer es emular el MC99999 en *software* y *hardware*.

La emulación de los coprocesadores por *software* no es siempre posible. En el caso de un coprocesador con una misión *hardware* altamente especializado, como el MC99999, la emulación puede ser imposible. Algunos coprocesadores pueden ser emulados implantando otros dispositivos *hardware*. Y para otros coprocesadores, como el MC68881, coprocesador de punto flotante, la emulación *software* es totalmente posible. De hecho, hay muchos sistemas basados en el 68000 que están empleando simulaciones *software* del MC68881.

Emular las características *hardware* en *software* presenta muchas ven-

tajas. Primera: Permite al fabricante del MC99999 realizar un modelo del comportamiento del *chip* antes de invertir en el desarrollo *hardware* del mismo. Segunda: Permite al fabricante del MC99999 instalar estas emulaciones antes de que el MC99999 esté realmente disponible. La emulación será varias veces más lenta que el MC99999 en si, pero en el mundo de los ordenadores una rutina lenta es mejor que no disponer de ninguna.

Tercera: Una vez que el MC99999 o el MC68881 estén disponibles, bastará con conectarlos al sistema MC68010 o MC68020 y el coprocesador funcionará sin necesidad de cambios en el *software*.

Coprocesadores no conectados en el MC68020

Si el MC99999 no se encuentra realmente presente y el MC68020 intenta comunicarse con él, se le notificará este hecho tras un ciclo del *bus* y se producirá una excepción motivada por una instrucción 1111 no implementada. Todo esto ocurre de forma automática.

La rutina de excepción que se dispara en este caso puede simplemente indicar que el MC99999 no se encuentra presente, o bien emularlo en *software* y devolver el control al programa. Como se ha discutido en la sección anterior, si se emula el MC99999 y éste se conecta más tarde, no serán necesarios cambios en el *software*.

Coprocesadores no estándar

¿Qué ocurrirá si el MC68020 se conecta a un coprocesador que no dispone de un interfaz estándar? Las instrucciones del MC68020 no serán capaces de conectarse con el coprocesador y, por tanto, no lo podrán em-

TABLA 8.7

Resumen de las instrucciones de control de coprocesadores

Mnemónico	*Descripción*
cpGEN	Envía una instrucción general al coprocesador
cpScc	Como Scc, pero emplea los códigos de condición del coprocesador
cpDBcc	Como Dbcc, pero emplea los códigos de condición del coprocesador
cpTRAPcc	Como TRAPcc, pero emplea los códigos de condición del coprocesador
cpBcc	Como Bcc, pero emplea los códigos de condición del coprocesador
cpSAVE	Salva el estatus del coprocesador (privilegiada)
cpRESTORE	Restaura el estatus del coprocesador (privilegiada)

plear. Habrá, pues, que sustituirlas por rutinas que empleen de forma explícita MOVEC, MOVES y MOVE para comunicarse con las direcciones adecuadas dentro del propio coprocesador. Estas rutinas serán similares, pero no idénticas, a las empleadas cuando el MC68010 emula las instrucciones de comunicación con los coprocesadores del MC68020.

Resumen de las instrucciones de manejo de coprocesadores del MC68020

Las siete instrucciones de manejo de coprocesadores del MC68020 se adaptan todas, para la primera palabra de instrucciones, al formato:

1 1 1 1 C c c I n s X x x x x x

Donde Ccc es un código de 3 bits que identifica al coprocesador, Ins depende de la instrucción y Xxxxxx depende de la misma. Nótese que los cuatro primeros bits de cada una de estas instrucciones son 1111. Como en otras instrucciones del 68000, algunas de éstas vienen seguidas de palabras adicionales en las que se incluyen desplazamientos, datos o códigos de condición.

El bit de *master*

El bit de *master* (M) se emplea en entornos donde varios sistemas operativos (*jobs*[4] privilegiados) están activos a la vez. El bit M distingue el sistema operativo maestro del resto de los sistemas. Debe notarse que excepto en el caso de desarrollo de sistemas operativos a muy bajo nivel, el programador normal no usará nunca el bit M.

Modos usuario y supervisor

El bit de *master* es una extensión del bit S, y el bit S es el mecanismo fundamental del 68000 para controlar las operaciones privilegiadas. Para clarificar la función de los bits S y M explicaremos, en primer lugar, los motivos para disponer de ambos.

Algunas funciones del procesador se consideran tan importantes que sólo algunos usuarios privilegiados o algunos programas privilegiados (como sistemas operativos) pueden tener acceso a ellas. En el 68000, estas funciones incluyen:

[4] No traduciremos el término *job*, que indica cada una de las tareas que se ejecutan bajo el control de cierto sistema operativo (programas, manipulación de ficheros, etc.), pues es un término incorporado a la jerga informática española.

- Una interrupción irreversible de todos los procesos en curso (RESET, STOP).
- Errores de cualquier tipo (errores de *bus*, errores de direcciones, errores de instrucción, divisor cero, errores de coprocesador).
- *Traps* de usuario.
- Interrupciones externas provocadas por el *hardware*.

El 68000 simplifica el concepto de privilegio al mecanismo más simple posible, es decir, en cualquier instante el 68000 está en modo privilegiado (supervisor) o en modo no privilegiado (usuario). Cuando se da cualquier condición de las listadas más arriba se abandona el programa en curso (ya estemos en modo usuario o supervisor) y se pasa el control al modo privilegiado (supervisor) y alguna rutina especial decide qué hacer ante la situación planteada. En algunas de estas rutinas la decisión a tomar dependerá de forma crítica del modo, supervisor o usuario, en que se encontraba el programa original cuando se produjo la situación anormal.

Una vez que el mecanismo de privilegios se ha activado de esta forma, aparecen dos funciones privilegiadas adicionales, a saber:

- Alternar entre los modos supervisor y usuario.
- Solicitar información acerca del modo activo (MC68010).

Llamar al modo supervisor es una función privilegiada y requiere una rutina privilegiada para decidir si se permite o no. Incluso solicitar información acerca del modo activo ("¿Estoy actualmente en modo supervisor?") es una función privilegiada. Esta situación de privilegio es necesaria para posibilitar situaciones en las que el sistema operativo principal emula otro sistema operativo. El sistema emulado se ejecuta en modo usuario, pero piensa que se encuentra en modo supervisor. Debe mantenerse en modo usuario, de modo que el sistema principal pueda interceptar cualquiera de las preguntas que, acerca de su estado, pueda plantear el sistema emulado, para devolver una respuesta adecuada.

Hay sólo dos niveles de privilegio en el sistema del MC68000, pero pueden implantarse más, de forma artificial, en el *software*. Simplemente mantengamos a todo el mundo en modo usuario (es decir, sin privilegios) y asignemos a cada cual un nivel de privilegio (por ejemplo, de 0 a 255). Cuando alguien intente emplear una función privilegiada, se pasará el control a una rutina adecuada, que comprobará el nivel de privilegio asignado y decidirá lo que deba permitir.

Implementaciones de los modos usuario, supervisor y *master*

En el MC68000, el bit 13 del registro de estatus se denomina bit S, o bit supervisor. El bit S se emplea para indicar cuándo el procesador se encuen-

tra en modo usuario (de bit S a 0) y cuándo se encuentra en modo supervisor (de bit S a 1). Algunas instrucciones, que se consideran privilegiadas, sólo se permiten cuando el sistema se encuentra en modo supervisor. Estas instrucciones son: instrucciones que cambian el bit S, que leen el bit S y MOVE USP, MOVEC, MOVES, RESET, STOP y RTE.

Cuando se intenta llevar a cabo una operación privilegiada en modo usuario se desencadena una excepción por violación de privilegio. El resto de las excepciones ocurre de la misma forma, esté el procesador en modo usuario o supervisor. Hay dos pilas, cada una con su propio puntero. El puntero de pila de usuario (USP) se emplea mientras el sistema está en modo usuario y el puntero de pila de modo supervisor (SSP) se emplea cuando el sistema se encuentra en modo supervisor. Siempre se hace referencia al puntero de uso mediante A7, ya se esté en USP o SSP. Cuando se produce una excepción se genera un bloque de información (denominado trama de pila), que se almacena en la pila de modo supervisor, y se llama a la rutina de excepción adecuada.

Cuando la rutina acaba, se elimina de la pila en modo supervisor trama de pila y se retorna el control al lugar donde ocurrió la excepción (a menos que ésta sea irrecuperable).

En el MC68020 se ha señalado otro bit para emplear conjuntamente con el bit S. Este es el bit 12 del registro de estatus, denominado M, o bit de *master*. En modo usuario (S = 0), las cosas suceden como en el MC68000. En modo supervisor (S = 1), existen, sin embargo, dos nuevos submodos. Cuando el bit M está a cero, el modo de interrupciones se considera activo y A7 se refiere al puntero de interrupciones (ISP), que apunta a la pila de interrupciones. Cuando el bit M está a 1, el modo *master* está activo y A7 se refiere al puntero de pila en modo *master* (MSP), que apunta a la pila en modo *master*. Siempre es posible referirse al puntero de pila activo mediante A77, ya sea éste USP, ISP, o MSP.

Cuando el bit M se pone a 0, todos los cambios de modo y pila funcionan como en el caso del MC68000, entendiéndose que la pila en modo supervisor actúa como pila de interrupciones. Cuando M se pone a 1, sin embargo, el tratamiento de las excepciones cambia de dos formas. Primera: las excepciones crean ahora tramas de pila en la pila en modo *master*, en lugar de en la pila en modo supervisor. Segunda: durante una excepción de interrupción por el *hardware* externo, la trama de pila se introduce en la pila en modo *master* y también se introduce en la pila del modo de interrupciones, al tiempo que se pone el bit M a 0, causando un cambio en el modo de interrupciones para subsiguientes procesos que empleen este bit. Así, un usuario normalmente opera en la pila en modo *master*, pero durante un proceso de interrupción provocado por el *hardware* externo opera en la pila de interrupciones, como en otros *jobs*.

Cada proceso de excepción termina normalmente con una instrucción RTE, que retorna el control al modo original, así como el acceso a la pila en dicho modo. En el caso de la doble trama de pila descrita más arriba, la instrucción RTE elimina ambas tramas de pila y retorna el control al modo original.

Las nuevas instrucciones del MC68020

Se han efectuado algunos cambios en el juego de instrucciones del MC68020. Muchas instrucciones antiguas han ampliado sus capacidades, al tiempo que se han añadido muchas instrucciones nuevas. En la sección final de este capítulo cubriremos todas las instrucciones que se han ampliado o añadido.

Instrucciones de bifurcación que admiten desplazamientos de 32 bits

Las instrucciones Bcc, BRA y BSR admiten ahora desplazamientos de 32 bits. Las instrucciones originales, con un formato de una única palabra, contenían 8 bits de código de operación y 8 bits de desplazamiento. Si el desplazamiento de 8 bits es nulo, la palabra siguiente se emplea como desplazamiento de 16 bits.

En el MC68020, si el desplazamiento de 8 bits es 255 (decimal), entonces las dos palabras siguientes se emplean como desplazamiento de 32 bits. El programador en lenguaje ensamblador generalmente no necesita estar al tanto de estas particularidades, porque el ensamblador decide automáticamente el formato a emplear.

Versión de 32 bits de la instrucción LINK

Originalmente, la instrucción LINK sólo admitía desplazamientos de 16 bits, mientras que ahora admite desplazamientos de 16 y 32 bits.

Extensión de bytes a dobles palabras

La instrucción EXTB.L amplía el signo de un byte a una doble palabra en un registro de datos. Las instrucciones originales del MC68000 son EXT.W, que amplía el signo de un byte a una palabra, y EXT.L, que amplía el signo de una palabra a una doble palabra. En los antiguos 68000 era necesario emplear las dos instrucciones EXT.W y EXT.L para lograr lo que ahora se hace con una sola instrucción.

Ampliación del límite superior a doble palabra para la instrucción CHK

La instrucción CHK del MC68000 comprueba que los valores de los registros de datos sean menos que el de una palabra dada. La instrucción CHK del MC68020 realiza la misma operación tanto con palabras como

con dobles palabras. No hay cambio aparente en la sintaxis de la operación, pero el ensamblador permitirá límites de 16 bits y decidirá automáticamente qué formato emplear.

CHK2 (instrucción nueva)

Se trata de una extensión posterior de la instrucción CHK. La instrucción CHK2 (comprobar dos límites) nos permite comprobar si un registro de datos o direcciones están dentro de unos límites que pueden ser bytes, palabras o dobles palabras. Todas las combinaciones están permitidas. La sintaxis es CHK2 ea,Rn, donde ea es la posición del límite inferior. El límite superior debe encontrarse inmediatamente después del inferior, a una distancia de 1, 2 ó 4 bytes, según el caso.

CMP2 (instrucción nueva)

La nueva instrucción CMP2 (comparar dos límites) opera como la instrucción CHK2, excepto que no produce un *trap* en el caso de que el valor dado se encuentre fuera de los límites, sino que pone los códigos de condición a los valores adecuados, permitiendo al usuario iniciar la opción oportuna. La sintaxis para esta instrucción es CMP2, ea,Rn y los indicadores de condición se ven afectados como sigue:

N: Indefinido
Z: Se pone a 1 si Rn coincide con cualquiera de los límites y a 0 en los demás casos
V: Indefinido
C: Se pone a 1 si Rn está fuera de los límites y a 0 en los demás casos
X: No se ve afectado

CAS y CAS2 (instrucciones nuevas)

CAS y CAS2 son instrucciones de comparación e intercambio, y ambas previenen contra el acceso de varios usuarios a las mismas áreas. Se pueden considerar como extensiones de la instrucción original TAS del MC68000. En un entorno multiusuario, la forma más sencilla de evitar que varios usuarios intenten actualizar la misma zona de datos se consigue estableciendo en algún lugar un indicador al que (por mutuo acuerdo) sólo puede asignar valores un usuario cada vez. La instrucción TAS permite a un usuario comprobar si un indicador está a 1 y, si no lo está, ponerlo a este valor. Además, TAS garantiza (y esto es lo más importante) que nadie tenga acceso al indicador durante el ciclo completo de lectura-modificación-escritura.

Por tanto, TAS satisface los requisitos mínimos necesarios para implantar con garantías un entorno multiusuario. Después de poner a 1 el indicador, el usuario es libre de ejecutar la rutina de actualización, sabiendo que sólo él tiene acceso a esa zona de datos. Los otros *jobs* deben abstenerse de ejecutar la rutina de actualización hasta que tengan el control del indicador. Por otra parte, el afortunado poseedor del indicador debe cooperar, liberándolo lo antes posible. Así, cuando un programa no consigue acceder al indicador, puede continuar probándolo, sabiendo que no caerá en un ciclo infinito, porque el indicador se liberará pronto.

Incluso en un entorno monousuario se necesita la instrucción TAS, porque existen otros usuarios en forma de interrupciones externas que pueden ocurrir en cualquier momento y con cualquier frecuencia. La posibilidad de restringir temporalmente el acceso a determinados recursos puede ser crítica.

CAS va más allá que TAS, y CAS2 más aún. Tienen los siguientes formatos:

```
CAS   Dc,Du,ea
CAS2  Dc1:Dc2,Du1:Du2,(Rn1:Rn2)
```

CAS compara Dc y ea y, si son iguales, Du reemplaza a ea. En el caso más simple, ea es un contador que cualquiera puede incrementar. El procedimiento consiste en copiar el valor de ea en Dc (el valor antiguo o comparado) y asignar a Du un nuevo valor (el valor nuevo o actualizado); entonces se ejecuta la instrucción CAS. Si falla, se ejecuta de nuevo. Precaución: si el contador Dc sufre una actualización continuada y rápida, probablemente sea más seguro emplear TAS; de otra forma, algunos usuarios pueden quedar bloqueados en sus intentos de actualización.

CAS2 es similar a CAS, salvo que realiza dos comparaciones. Si alguna de ellas falla, no se realiza la actualización. Nótese que allí donde CAS permite cualquier modo de direccionamiento por posición efectiva alterable de la memoria, CAS2 sólo permite modos de direccionamiento indirecto por registro (de cualquier tipo). Las precauciones que hay que tomar con CAS también son aplicables en este caso.

Nuevos formatos de las instrucciones DIV y MUL

Se han añadido algunos formatos nuevos para las instrucciones de multiplicación y división del MC68020. Todos los formatos usan 1 ó 2 registros de datos y una dirección efectiva para los operandos fuente, y almacenan los resultados en uno o más registros de datos. A continuación se indican todos los formatos, incluyendo los antiguos, que se encuentran disponibles en el MC68020.

Los distintos subíndices que se emplean con los registros de datos son: n para cualquier registro, r para los restos, q para los cocientes, h para la doble palabra más significativa y l para la doble palabra menos significa-

TABLA 8.8

Formatos para división y multiplicación

Instrucción	*Acción que realiza*				*Procesador*
DIVU.W	dea,Dn	Dn(32)	/ dea(16)	= Dn(16r:16q)	68000
DIVU.L	dea,Dq	Dq(32)	/ dea(32)	= Dq(32)	68020
DIVU.L	dea,Dr:Dq	Dr:Dq(64)	/ dea(32)	= Dr(32), Dq(32)	68020
DIVUL.L	dea,Dr:Dq	Dq(32)	/ dea(32)	= Dr(32), Dq(32)	68020
DIVS.W	dea,Dn	Dn(32)	/ dea(16)	= Dn(16r:16q)	68000
DIVS.L	dea,Dq	Dq(32)	/ dea(32)	= Dq(32)	68020
DIVS.L	dea,Dr:Dq	Dr:Dq(64)	/ dea(32)	= Dr(32), Dq(32)	68020
DIVSL.L	dea,Dr:Dq	Dq(32)	/ dea(32)	= Dr(32), Dq(32)	68020
MULU.W	dea,Dn	dea(16)	× Dn(16)	= Dn(32)	68000
MULU.L	dea,Dl	dea(32)	× Dl(32)	= Dl(32)	68020
MULU.L	dea,Dh:Dl	dea(32)	× Dl(32)	= Dh:Dl(64)	68020
MULS.W	dea,Dn	dea(16)	× Dn(16)	= Dn(32)	68020
MULS.L	dea,Dl	dea(32)	× Dl(32)	= Dl(32)	68020
MULS.L	dea,Dh:Dl	dea(32)	× Dl(32)	= Dh:Dl(64)	68020

tiva. Los números 16, 32 y 64 indican palabra, doble palabra y palabra cuádruple.

PACK y UNPK (instrucciones nuevas)

PACK realiza la conversión de números ASCII y EBCDIC a BCD, y UNPK realiza la conversión en sentido inverso. En una situación normal, se lee la secuencia ASCII o EBCDIC de un dispositivo de entrada (por ejemplo, una terminal, una cinta o un fichero en disco), se convierten internamente a BCD empleando una instrucción PACK y se realizan los cálculos en BCD empleando las instrucciones ABCD, SBCD y NBCD. Después se convierten de nuevo a ASCII o EBCDIC empleando la instrucción UNPK, enviándose finalmente a un dispositivo de salida. Los formatos disponibles son:

```
PACK   -(Ax),-(Ay), #margen
PACK   Dx,Dy, #margen
UNPK   -(Ax),-(Ay), #margen
UNPK   Dx,Dy, #margen
```

PACK toma una palabra del operando fuente, le suma el margen y escribe los dígitos segundo y cuarto en hexadecimal (bits [11:8] y [3:0]) del resultado en el byte de destino. Para una conversión ASCII, el desplazamiento es −$3030 = $CFD0. Para conversiones EBCDIC, el desplazamiento es −$F0F0 = $0F10. Una cadena de dígitos de cualquier longitud puede

convertirse empleando un bucle, que consiste en una instrucción PACK predecrementada y una instrucción DBcc. Así, por ejemplo, una cadena de dígitos de cualquier longitud podría convertirse como sigue:

```
        MOVE.L   #<tamaño-1>,D0      Establece el contador de dígitos
        MOVEA.L  #<finpack>,A0       Fin de los dígitos empaquetados
        MOVEA.L  #<finnopack>,A1     Fin de los dígitos no empaquetados
BUCLE   PACK     -(A0),-(A1),#$CFD0  Convertir a ASCII
        DBF      D0.BUCLE            Para llegar a -1
```

UNPK toma un byte del operando fuente, crea una palabra a partir de él, cuyos dígitos hexadecimales segundo y cuarto son los dos dígitos del operando fuente, y los dígitos primero y tercero son 0, le suma el desplazamiento y lo escribe en la palabra que se encuentra en la dirección de destino. Para conversiones ASCII, el desplazamiento es + $3030. Para conversiones EBCDIC, es + $F0F0.

Nuevos modos de direccionamiento para las instrucciones TST y CMPI

En el MC68000, las instrucciones TST (probar) y CMPI (comparar de modo inmediato) sólo permiten direcciones efectivas alterables de datos (adea). En el MC68020, TST.B y CMPI.B siguen restringidas a modos adea, mientras que TST.W, CMPI.W, TST.L y CMPI.L pueden operar con cualquier dirección efectiva. El efecto neto de este cambio es el de incluir los modos de direccionamiento por PC y directo por registro de direcciones.

TRAPcc (instrucción nueva)

TRAPcc es una nueva instrucción del MC68020 que genera un *trap* cuando se verifica un determinado código de condición. Se permiten los 16 códigos de condiciones, incluyendo "siempre *trap* (TRAPT)" y "nunca *trap* (TRAPF)". Opcionalmente, una palabra o una doble palabra puede seguir a la instrucción TRAPcc; el procesador no emplea esta palabra o doble palabra, pero queda a disposición de la rutina de *trap* del usuario. Los formatos disponibles son:

```
TRAPcc
TRAPcc.W  #d16
TRAPcc.W  #d32
```

Instrucciones de manejo de coprocesadores

Véase la sección de coprocesadores en otro lugar de este capítulo.

TABLA 8.9

Resumen de las instrucciones de manejo de campos de bits

Mnemónico	*Operando*	*Descripción*
BFEXTS	ea[offset:width],Dn	Extrae el bit de signo (extendido)
BFEXTU	ea[offset:width],Dn	Extrae el bit de signo para números sin signo (lo reemplaza por 0)
BFFFO	ea[offset:width],Dn	Encuentra el primer bit 1 en el campo de bits indicado
BFFINS	Dn,ea[offset:width]	Inserta bits (a partir de la posición menos significativa) en el campo indicado
BFCLR	ea[offset:width]	Pone todos los bits a 0
BFSET	ea[offset:width]	Pone todos los bits a 1
BFCHG	ea[offset:width]	Prueba un campo de bits, después lo complementa
BFTST	ea[offset:width]	Prueba un campo de bits

Nuevos registros de control accesibles mediante la instrucción MOVEC

La instrucción MOVEC puede acceder ahora a los registros CAAR y CACR. Véase el capítulo 7 para una discusión más amplia de la instrucción MOVEC.

Instrucciones de manejo de campos de bits (nuevas)

Hay 8 instrucciones de manejo de campos de bits en el MC68020. En cada una de ellas, el operando principal es un campo de bits con una extensión variable (de 1 a 32). Se hace referencia a este campo mediante un registro de datos a una posición de la memoria. Se considera que el bit situado más a la izquierda es el bit cero, y se define el campo de bits especificando, mediante un desplazamiento, el número de bits a contar desde el bit cero. El segundo de los operandos es un registro de datos.

En la tabla 8.9 tenemos:

- ea indica una dirección efectiva
- El desplazamiento es un valor inmediato que va de 0 a 31 o el valor de un registro de datos entre −2.147.483.647 y +2.147.483.648
- Longitud es la longitud del campo de bits entre 0 y 31 o el valor de un registro de datos en módulo 32. O representa un valor 32
- Dn es un registro de datos

A continuación se dan algunos ejemplos de instrucciones de manejo de campos de bits:

BFEXTU	D0[8:15],D1	Mover el bit 2 de D0 a D1
BFCLR	(A0)[2:5]	Poner a cero 4 bits de un byte de la memoria (A0)
BFFFO	(A0)[D0:32],D1	Encontrar el primer bit 1 comenzando en la posición D0 del byte (A0) de la memoria; buscar en 32 bits

Conclusión

Mientras las nuevas características del MC68010 están construidas en su práctica totalidad en torno a una única función (la emulación), el MC68020 introduce una gran variedad de nuevas funciones. Comparado con el MC68010, el MC68020 tiene un rango de direcciones 256 veces mayor, una mayor velocidad de ejecución, debido a su sistema *cache*, y es capaz de comunicarse de forma más rápida con los coprocesadores. Los nuevos modos de direccionamiento (6 en total), las instrucciones mejoradas y las nuevas instrucciones permiten programas más rápidos, que emplean menos memoria y más fáciles de desarrollar. Las instrucciones de bifurcación, que han sido mejoradas, permiten saltos de 32 bits. Las instrucciones de multiplicación y división soportan ahora operaciones de 64 bits. Finalmente, las nuevas instrucciones se manejo de campos de bits incrementan la velocidad y la potencia de las instrucciones que cambian los bits individualmente; esto ayuda en la actualización de los *bitmaps*[5] de los directorios de los discos y es de importancia crítica en el manejo de sistemas gráficos en tiempo real.

[5] El *bitmap* es una zona del disco (generalmente de los *diskettes* de 3 ó 5 pulgadas) en la que se indica qué partes del mismo están ocupadas. Generalmente se asigna a cada sector un bit, que se pone a 1 cuando está libre y a 0 cuando está ocupado (o viceversa).

Apéndice A

Instrucciones del M68000: Número de operandos

Sin operando:

NOP
ILLEGAL
RESET*
RTE*/RTR/RTS
TRAPV

Un operando:

ASL/ASR
Bcc/BRA/BSR
CLR
EXT
JMP/JSR
NBCD
NEG/NEGX
NOT
PEA
RTD
SCC
STOP*

SWAP
TAS
TRAP
TST
UNLK

Dos operandos:

ABCD
AND/ADDA/ADDI/ADDQ/ADDX
AND/ANDI/ANDI~CCR/ANDI~SR*
ASL/ASR
BCHG/BCLR/BSET/BTST
CHK
CMP/CMPA/CMPI/CMPM
DBcc
DIVS/DIVU
EOR/EORI/EORI~CCR/EORI~SR*
EXG
LEA
LINK
LSL/LSR~

MOVE/MOVE-desde-CCR/MOVE-al-CCR/MOVE-al-SR*/
MOVE-desde-SR**/MOVE-USP*/MOVEA/MOVEC**/MOVEM/
MOVEP/MOVEQ/MOVES**

MULS/MULU
OR/ORI/ORI-CCR/ORI-SR*
ROL/ROR/ROXL/RORX~
SBCD
SUB/SUBA/SUBI/SUBIQ/SUBX

Leyenda: ~ = Puede tener uno o dos operandos.
* = Privilegiada en el MC68000.
** = Privilegiada en el MC68010.

Apéndice B

Modos de direccionamiento del M68000

<ea> = Cualquier dirección efectiva
<rea> = Dirección efectiva de un registro
<dea> = Dirección efectiva de los datos
<mea> = Dirección efectiva de la memoria
<cea> = Dirección efectiva de control
<aea> = Dirección efectiva alterable (datos o memoria)
<adea> = Dirección efectiva alterable de datos
<amea> = Dirección efectiva alterable de la memoria
<acea> = Dirección efectiva alterable de control

Modo	*ea*	*rea*	*dea*	*mea*	*cea*	*aea*	*adea*	*amea*	*acea*
Dn	*	*	*			*	*		
An	*	*				*			
(An)	*		*	*	*	*	*	*	*
(An) +	*		*	*		*	*	*	
−(An)	*		*	*		*	*	*	
d(An)	*		*	*	*	*	*	*	*
d(An,Xi)	*		*	*	*	*	*	*	*
Abs.W	*		*	*	*	*	*	*	*
Abs.L	*		*	*	*	*	*	*	*
d(PC)	*		*	*	*				

Modo	*ea*	*rea*	*dea*	*mea*	*cea*	*aea*	*adea*	*amea*	*acea*	
d(PC,Xi)	*		*	*	*					
Inmed	*		*	*						
bd(An,Xi)	*		*	*	*	*	*	*	*	MC68020
bd(PC,Xi)	*		*	*	*					MC68020
[bd,An],Xi,od	*		*	*	*	*	*	*	*	MC68020
[bd,An,Xi],od	*		*	*	*	*	*	*	*	MC68020
[bd,PC],Xi,od	*		*	*	*					MC68020
[bd,PC,Xi],od	*		*	*	*					MC68020

Descripción de los modos de direccionamiento

Modos comunes a toda la familia:

Dn	Registro de datos directo
An	Registro de direcciones directo
(An)	Registro de direcciones indirecto
(An) +	Registro de direcciones indirecto con posincremento
−(An)	Registro de direcciones indirecto con predecremento
d16(An)	Registro de direcciones indirecto con desplazamiento; también se escribe d(An)
d8(An,Xi.Z)	Registro de direcciones indirecto con desplazamiento e índice; también se escribe d(An,Xi)
Abs.W	Dirección absoluta corta; también se escribe como xxx.W o como una etiqueta
Abs.L	Dirección absoluta larga; también se escribe como xxx.L o como una etiqueta
d8(PC)	Contador de programa con desplazamiento (modo relativo); también se escribe como d(PC) o como una etiqueta
d8(PC,Xi.Z)	Contador de programa con desplazamiento e índice (modo relativo); también se escribe como d(PC,Xi) o como una etiqueta y (PC,Xi)
Inmed	Operando inmediato; también se escribe como #<datos>

Variantes y adiciones para el MC68020:

bd(An,Xi.Z*s)	Registro de direcciones indirecto con desplazamiento de la base e índice [similar al d(An,Xi.Z*s), pero bd puede ser d16 o d32]
bd(PC,Xi.Z*s)	Contador de programa con desplazamiento de la base e índice [similar al d(PC,Xi.Z*s), pero db puede ser d16 o d32]
[bd,An],Xi.Z*s,od	Posindexado indirecto en la memoria
[bd,An,Xi.Z*],od	Preindexado indirecto en la memoria

[bd,PC],Xi.Z*s,od	Contador de programa indirecto posindexado en la memoria
[bd,PC,Xi.Z*s],od	Contador de programa indirecto preindexado en la memoria

Abreviaturas:

Dn	Cualquier registro de datos, D0-D7
An	Cualquier registro de direcciones, A0-A7
Xi	Cualquier An o Dn empleado como registro índice
z	Indicador del tamaño de los datos (B, W, o L)
Z	Indicador del tamaño de los datos (L o W)
s	Factor de escala (1, 2, 4 u 8)
PC	Contador de programa (20, 24 ó 32 bits)
SR	Registro de estado
CCR	Registro de códigos de condición
d	Un desplazamiento extendido o en complemento a 2: d16, d8, d3, etc., indican el número de bits
bd	Un desplazamiento de la base en complemento a 2 (16 ó 32 bits)
od	Un desplazamiento exterior en complemento a 2 (16 ó 32 bits)
xxx	Cualquier dirección absoluta válida

Apéndice C

Instrucciones del MC68000: Modos legales

Mnemónico	*Función*	*Modos permitidos*	*Tamaño del operando*
ABCD	Sumar en decimal	Dm,Dn o −(Am),−(An)	B
ADD	Sumar en binario	<ea>,Dn o Dn,<amea>	L,W,B
ADDA	Sumar direcciones	<ea>,An	L,W
ADDI	Suma inmediata	#<datos>,<adea>	L,W,B
ADDQ	Suma rápida	#<d3>,<aea>	L,W,{B}
ADDX	Suma extendida	Dm,Dn o −(Am),−(An)	L,W,B
AND	Y lógico	<dea>,Dn o Dn,<amea>	L,W,B
ANDI	Y inmediato	#<datos>,<adea>	L,W,B
ANDI → CCR	Suma inmediata con el CCR	#<d8>,CCR	B
ANDI → SR*	Suma directa con el SR	#<d16>,SR	W
ASL/ASR	Desplazamientos aritméticos	Dm,Dn o #<d3>,Dn	L,W,B (Dm mod 64)
ASL/ASR	Desplazamientos aritméticos	<amea>	W (contador de desplazamiento = 1)
Bcc	Condición de bifurcación	<etiqueta>	Desplazamiento de 16 bits
Bcc.S	Condición de desplazamiento corto	<etiqueta>	Desplazamiento de 8 bits

Mnemónico	*Función*	*Modos permitidos*	*Tamaño del operando*
BCHG	Comprobar/Cambiar un bit	Dm,Dn o #<d5>,Dn	L (Dm mod 32)
BCHG	Dm,<amea> o #<d3>,<amea>		
BCLR	Probar/Poner a cero un bit	Igual que BCHG	
BRA	Salto incondicional	<etiqueta>	Desplazamiento de 8 ó 16 bits
BSET	Comprobar/Poner a 1 un bit	Igual que BCHG	
BSR	Saltar a una subrutina	Igual que BRA	
BTST	Comprobar un bit	Dm,Dn o #<d15>,Dn	L (Dm mod 32)
BTST		Dm,<mea> o #<d3>,<mea>	B (Dm mod 8)
CHK	Comprobar los límites de los registros	<dea>,Dn	W
CLR	Poner a 0 un operando	<adea>	L,W,B
CMP	Comparar	<ea>,Dn	L,W,(B)
CMPA	Comparar direcciones	<ea>,An	L,W
CMPI	Comparación inmediata	#<dato>,<adea>	L,W,B
CMPM	Comparar con la memoria	(Am)+,(An)+	L,W,B
DBcc	Salto cond. decimal	Dm,<etiqueta>	Desplazamiento de 16 bits
DIVS/DIVU	División con y sin signo	<dea>,Dn	W
EOR	OR exclusivo	Dn,<adea>	L,W,B
EORI	OR exclusivo inmediato	#<datos>,<adea>	L,W,B
EORI>CCR	Códigos de condiciones de EORI	#<d8>,CCR	B
EORI>SR*	Registro de estado de EORI	#<d16>,SR	W
EXG	Intercambiar registros	Rm,Rn	L
EXT	Extensión de signo	Dn	L,W
ILLEGAL	No permitido	Sin operando	
JMP/JSR	Salto/Salto a subrutina	<cea>	Sin tamaño
LEA	Cargar una dirección efectiva	<cea>,An	L
LINK	Asignar espacio	An,#<d16>	Sin tamaño
LSL/LSR	Desplazamientos lógicos	Igual que ASL/ASR	
MOVE	Mover datos	<ea>,<adea>	L,W,(B)
MOVE>CCR	Mover al CCR	<adea>,CCR	W (palabra más baja)
MOVE>SR*	Mover al SR	<adea>,SR	W (* todos los modos)
MOVE<CCR	Mover desde el CCR	CCR,<adea>	W (palabra más baja)
MOVE<SR^^*		SR,<adea>	W (* sólo para el 68010)
MOVE>USP*	Mover al USP	USP,An o An,USP	L
MOVEA	Mover una dirección	<ea>,An	L,W

Mnemónico	*Función*	*Modos permitidos*	*Tamaño del operando*
MOVEC^^*	Mover a los registros de control	Rc,Rn o Rn,Rc	L
MOVEM	Mover varios registros	<lista,reg>,<acea>+, <cea>+, <lista.reg>	L,W [+ −(An)] L,W [+ +(An)]
MOVEP	Mover datos desde/ hacia los periféricos	Dn,d(An) o d(An),Dn	L,W
MOVEQ	Mover rápidamente	#<d8>,Dn	L (extensión de signo a 32 bits)
MOVES^^*	Mover espacios de direcciones	RN,DFC<amea> o SFC<amea>,Rn	L,W,B
MULS/MULU	Multiplicar con y sin signo	<dea>,Dn	W
NBCD	Negar en decimal	<adea>	B
NEG/NEGX	Negar/Negar en modo extendido	<adea>	L,W,B
NOP	No efectúa operación alguna	Sin operandos	Sin tamaño
NOT	Complemento lógico	<adea>	L,W,B
OR	OR lógico inclusivo	Igual que AND	
ORI	OR inclusivo inmediato	Igual que ANDI	
ORI>CCR	ORI con los códigos de condición	Igual que ANDI>CCR	
ORI>SR*	ORI con el registro de estado	Igual que ANDI>SR	
PEA	Obtener una dirección efectiva de control	<cea>	L
RESET*	Reinicia un dispositivo externo	Sin operandos	Sin tamaño
ROL/ROR	Rotar a la izquierda o a la derecha	Igual que ASL/ASR	
ROXL/ROXR	Rotación con extensión	Igual que ASL/ASR	
RTD^^	Retornar/Desasignar	#<d16>	Sin tamaño (extensión de signo a 32 bits)
RTE*	Retornar de una excepción	Sin operando	Sin tamaño
RTE^^*		Sin operando	Sin tamaño
RTR	Retornar y restaurar el CCR	Sin operando	Sin tamaño
RTS	Retornar de una subrutina	Sin operando	Sin tamaño
SBCD	Resta decimal	Igual que ABCD	
Scc	Poner a 1 condicionalmente	<adea>	B
STOP*	Carga un SR/STOP	#<d16>	Sin tamaño
SUB	Resta binaria	Igual que ADD	
SUBI	Resta inmediata	Igual que ADDI	
SUBQ	Resta rápida	Igual que ADDQ	
SUBX	Resta extendida	Igual que ADDX	

Mnemónico	*Función*	*Modos permitidos*	*Tamaño del operando*
SWAP	Intercambia registros	Dn	W
TAS	Comprueba y pone a 1 un operando	<adea>	B
TRAP	Trap	#<d4>	Sin tamaño
TRAPV	Trap cuando se produce un *overflow*	Sin operando	Sin tamaño
TST	Comprobar un operando	<adea>	L,W,B
UNLK	Desasignar	An	Sin tamaño

Apéndice D

Resumen de las instrucciones del M68000

Este apéndice constituye una referencia para todas las instrucciones implementadas hasta ahora en la familia del 68000, es decir, el MC68000, el MC68010 y el MC68020. Se incluyen las secuencias de bits para cada instrucción, así como las secuencias de bits que identifican a cada uno de los modos de direccionamiento. Este apéndice no describe cómo se ejecutan en realidad cada una de estas instrucciones, aspecto que constituye el capítulo 1 de esta obra. Aquí se contempla el set de instrucciones desde un punto de vista muy general y esperamos proporcionar algunos trucos que ayuden a memorizarlo.

Modos de direccionamiento del 68000

La familia del 68000 emplea un rico juego de modos de direccionamiento. Hay 12 modos de direccionamiento básico, además de algunos modos de direccionamiento implicados para algunas instrucciones. El MC68020, además, amplia las posibilidades de dos de los modos de direccionamiento básicos, proporcionando un total de 18 variaciones sobre los 12 modos de direccionamiento básicos. (Véase el capítulo 8 para ampliar más detalles.)

La tabla D.1 contiene información realtiva a estos modos de direccionamiento. La primera columna contiene el nombre completo o la descripción de estos 18 modos de direccionamiento. Nótese que algunas de estas descripciones son largas.

La columna "CPU" indica en qué microprocesador de la familia se encuentra disponible este modo de direccionamiento; si se encuentra en blanco, esto indica que el modo de direccionamiento se aplica a todos los procesadores de la familia del 68000; un "20" indica que el modo de direccionamiento sólo se encuentra disponible en el MC68020.

La columna marcada como "Mod Reg" contiene los códigos de 6 bits disponibles para cada modo de direccionamiento. Los números binarios que identifican los registros se representan por "rrr", que puede variar entre "000" y "111".

Las cuatro columnas siguientes definen cuatro categorías de modos de direccionamiento. Estas categorías son muy útiles para definir qué modos de direccionamiento están permitidos en cada una de las instrucciones del 68000.

Registros (REA): Estos modos de direccionamiento se refieren a los registros. Esto incluye los direccionamientos directos por registros de direcciones y datos.

Datos (DEA): Estos modos de direccionamiento se refieren a los operandos constituidos por datos. Esto incluye todos los modos, excepto el direccionamiento directo por registro de direcciones. Muchas instrucciones del 68000 que operan sobre datos están diseñadas de modo que no pueden alterar los registros de direcciones. Esto forma parte de la filosofía del 68000, basada en la creencia de que evitar errores en el manejo de los registros de dirección (que pueden dar lugar a grandes pérdidas de tiempo durante las fases de desarrollo de un programa), compensan los esfuerzos extra de programación que habrá que realizar para hacer las manipulaciones deseadas sobre los registros de datos.

Memoria (MEA): Estos modos de dirección se refieren a los operandos de la memoria. Esto incluye todos los modos, excepto los de direccionamiento directo por registro. Hay dos instrucciones con sus modos de direccionamiento restringidos a este tipo, porque se trata de instrucciones que realmente se ocupan del manejo de la memoria, éstas son TAS y MOVES. Las cuatro instrucciones de desplazamiento OR, AND y ADD también imponen restricciones MEA en algunos de sus formatos. Por ejemplo, ADD D0,D1 se permite en el formato ADD ea,D1, pero no se permite en el formato ADD ea,D1 (para evitar duplicaciones). Por tanto, el formato ADD D0,ea sólo permite MEA como destino, mientras que el formato ADD ea,D1 permite cualquier ea —dirección efectiva— como fuente.

Control (CEA): Estos modos de direccionamiento se refieren a posiciones de memoria sin especificar si se trata, por su tamaño, de dobles palabras, palabras o bytes. Esto se aplica a los comandos de salto (Jxx) y a aquellos de con destino de tamaño indefinido (LEA, PEA, MOVEM, BFxxxx).

TABLA D.1

Modos de direccionamiento efectivos

Nombre del modo de direccionamiento	*CPU*	*Mod*	*Reg*	*Registro (R)*	*Datos (D)*	*Memoria (M)*	*Control (C)*	*Alterable (A)*	*Sintaxis del ensamblador*	*Palabras de extensión*
Registro de datos directo		000	rrr	X	X	—	—	X	Dn	0
Registro de direcciones directo		001	rrr	X	—	—	—	X	An	0
Registro de direcciones indirecto		010	rrr	—	X	X	X	X	(An)	0
Registro de direcciones indirecto con posincremento		011	rrr	—	X	X	—	X	(An) +	0
Registro de direcciones indirecto con predecremento		100	rrr	—	X	X	—	X	−(An)	0
Registro de direcciones indirecto con desplazamiento		101	rrr	—	X	X	X	X	(d16,An)	1
Registro de direcciones indirecto con indice y desplazamiento de 8 bits		110	rrr	—	X	X	X	X	(d8,An,Rn)	1
Registro de direcciones indirecto con indice y desplazamiento de la base	20	110	rrr	—	X	X	X	X	(bd,An,Rn)	1-3
Memoria indirecta posindexada	20	110	rrr	—	X	X	X	X	([bd,An],Rn,od)	1-5
Memoria directa preindexada	20	110	rrr	—	X	X	X	X	([bd,An,Rn],od)	1-5
Absoluto corto		111	000	—	X	X	X	X	addr.W	1
Absoluto largo		111	001	—	X	X	X	X	addr.L	2
Contador de programa indirecto con desplazamiento		111	001	—	X	X	X	—	(d16.PC)	1
Contador de programa indirecto con indice y desplazamiento de 8 bits		111	011	—	X	X	X	—	(d8.PC,Rn)	1
Contador de programa indirecto con indice y desplazamiento de la base	20	111	011	—	X	X	X	—	(bd,PC,Rn)	1-3
Contador de programa indirecto posindexado	20	111	011	—	X	X	X	—	([bd,PC],Rn,od)	1-5
Contador de programa indirecto preindexado	20	111	011	—	X	X	X	—	([bd,PC,Rn],od)	1-5
Inmediato		111	100	—	X	X	—	—	#n	1-2
(Reservado por Motorola)	?	111	101							
(Reservado por Motorola)	?	111	110							
(Reservado por Motorola)	?	111	111							

Alterable (AxEa): Esto se refiere a operandos que pueden cambiarse. Esto excluye los modos inmediato y relativo al PC. Motorola toma la postura de no considerar los programas automodificantes. Realmente, un programa diseñado para el 68000 puede automodificarse, pero normalmente requerirá una instrucción extra (normalmente LEA) para lograrlo. Esta es la parte de la filosofía de diseño del 68000 que considera que los beneficios que se obtienen de evitar los errores de un programa que se automodifica (evitando así las consiguientes pérdidas de tiempo en la fase de desarrollo) compensan con creces los pequeños esfuerzos adicionales que habrá que realizar en la programación para legitimar este tipo de cambios. Si se planea realizar muchas referencias de carácter automodificante en un programa, basta con incluir una instrucción "LEA etiqueta1,An" al comienzo del mismo y realizar todas estas referencias empleando "etiqueta1-etiqueta1(An)".

Combinando las categorías anteriores se pueden crear otras, como datos alterables, memoria laterable y control alterable. Por ejemplo, las direcciones de datos alterables son aquellas que son tanto direcciones alterables y de datos.

La columna etiquetada "sintaxis de ensamblador" contiene una representación figurativa de los caracteres realmente empleados para codificar cada uno de los modos de direccionamiento en un programa del 68000. El ensamblador que se emplee en un determinado sistema puede tener su propia sintaxis que difiera de la presentada aquí. El símbolo an representa cualquier registro de A0 a A7, Dn cualquier registro de D0 a D1, Rn indica An o Dn, d8 cualquier número sin signo de 0 a 255 o cualquier número con signo de −128 a +127, d16 cualquier número sin signo de 0 a 65536 o con signo de −32768 a +32767, d32 cualquier número sin signo de 0 a 4294967295 o con signo de −2147483648 a +2147483647 y #n d8, d16 o d32 en las instrucciones relativas a bytes, palabras o dobles palabras.

Modos de direccionamiento permitidos y ortogonalidad

Una característica importante del diseño del set de instrucciones del M68000 es la ortogonalidad entre sus instrucciones y sus modos de direccionamiento. Una ortogonalidad completa en el direccionamiento indicaría que cada instrucción es capaz de emplear todos los modos de direccionamiento en cada uno de sus operandos (fuente, destino, contador, etc.), flexibilizando y facilitando la programación en ensamblador. Pocos microprocesadores, sin embargo, se han diseñado con una ortogonalidad completa, porque esto representaría también inconvenientes. La ortogonalidad completa se consigue normalmente pagando un elevado precio en la potencia del set de instrucciones, ya que combinaciones irrelevantes de instruc-

ción/modo de direccionamiento emplean secuencias de bits que podrían emplearse para otras instrucciones más poderosas. El M68000 consigue una maravillosa optimización, logrando un set de instrucciones altamente ortogonal y muy potente.

Cada instrucción del M68000 consiste en un operador y cero o más operandos. Se relacionan a continuación algunas instrucciones con diferentes números de operandos:

RTS		0 operandos
BR	ETIQUETA	1 operando
OR.W	D0,4(A0)	2 operandos
CAS2.W	D0:D1,D2:D3,4(A0)	5 operandos

En la mayoría de las instrucciones, cada uno de los operandos sólo puede encontrarse en uno de los 12 modos de direccionamiento, o bien se le permite que tenga cualquiera de los 12 modos de direccionamiento permitidos. Es importante, para todos los programadores, comprender los criterios que Motorola ha empleado para definir los modos permitidos para una determinada instrucción. Estos criterios se han aplicado de una forma consistente en todo el set de instrucciones del 68000. Sin comprender estos criterios, parecerá que el 68000 carece de gran parte de su ortogonalidad; sin embargo, si estos criterios se entienden resultará fácil memorizar completamente el set de instrucciones del 68000.

Un ejemplo puede ayudar a aclarar todo lo anteriormente expuesto. La instrucción OR señalada más arriba forma parte de un grupo de instrucciones que tienen todas el mismo formato:

```
OR.tamaño    Dn,<destino>
```

y están representadas por una secuencia de 16 bits que corresponde con:

```
1 0 0 0 S d r 1 S z D e s t i n
```

No es necesario explicar las abreviaturas Sdr, Sz y Destin para esta descripción, pero pueden encontrarse más adelante.

Hay dos operandos en esta instrucción. El primero de los operandos debe ser siempre un registro de datos en modo directo, pero el primero puede ser cualquiera de los modos de direccionamiento permitidos. En este grupo de instrucciones OR, cinco de los 12 modos de direccionamiento permitidos son ilegales. Los cinco modos ilegales violan tres de las reglas de legalidad. Los modos ilegales, con las reglas violadas, son:

1. Modos relativos o inmediatos al PC: Este no puede ser nunca alterado.
2. An: Las operaciones sobre datos no están en general permitidas sobre un registro An.
3. Dn: Esto duplica la función de otro grupo de instrucciones.

La regla 1 es rigurosamente cierta para aquellas instrucciones que alteran el operando destino. La regla 2 es casi siempre cierta, pero se permiten algunas operaciones sobre An, como, por ejemplo, MOVE, ADD, SUB, ADDQ y SUBQ. Algunos ensambladores permiten emplear estas instrucciones sin restricciones. Algunos, sin embargo, requieren el empleo de MOVEA, ADDA y SUBA en lugar de MOVE, ADD y SUB cuando el destino es un registro de direcciones. El resultado neto es el mismo, pero es más difícil que se produzcan errores inadvertidos. Para comprender la regla 3 es necesario saber que hay otro grupo de instrucciones OR, que tienen la forma:

```
OR.tamaño    <destino>,Dn
```

y están representadas por una secuencia de 16 bits que corresponde con:

```
1 0 0 0 D d r 0 S z F u e n t e
```

Si ambos grupos se permitiesen en el caso:

```
OR,tamaño    Dn,Dm
```

la ortogonalidad del set de instrucciones sería mayor, pero éste sería menos potente, al reducirse el número de secuencias de bits que quedarían libres para otras instrucciones. De hecho, algunas de las secuencias de bits que la regla 3 deja libres se emplean para la instrucción NBCD. Un programador no necesita conocer la regla 3, pues cualquier ensamblador ensamblará una instrucción OR.tamañoDn,Dm en la forma correcta. Por otra parte, si se está diseñando un ensamblador o desensamblando un programa, la regla 3 es de capital importancia.

El ejemplo anterior hará, probablemente, más fácil de entender la siguiente lista de reglas que —junto con sus excepciones— gobiernan los modos de direccionamiento del 68000.

1. Los modos relativos al PC o los inmediatos son ilegales si se altera el destino. Esta es la causa de que exista una categoría de modos de direccionamiento "alterables" (AEA). Nótese que las instrucciones de comprobación no alteran el destino, aunque simulan una resta de éste. Las excepciones son: TST y CMPM; con estos modos de direccionamiento son ilegales, aunque no alteren el destino. En el MC68020, TST.W, TST.L, CMPI.W y CMPI.LL son legales en todos los modos, pero TST>B y CMPI.B no.
2. Las funciones duplicadas son competencia únicamente de una de las instrucciones. Por ejemplo, en todas las instrucciones de desplazamiento "op #1,Dn" está permitido, mientras que "op Dn" no lo está. Las excepciones a esta regla son CMP #c,Dn y CMPI #c,Dn,

que, aunque son equivalentes, se ensamblan en dos instrucciones diferentes (ambas con dos palabras). Lo mismo es cierto para los pares de instrucciones siguientes: ADD/ADDI, SUB/SUBI, AND/ANDI y OR/ORI. Nota: si 1 < = c < = 8, entonces ADDQ #c,Dn y SUBQ #c,Dn están también permitidos y son equivalentes a los casos anteriores, pero ocupan una palabra menos. Por tanto, ADDQ y SUBQ no se consideran duplicadas (o triplicadas).

3. Muchas de las operaciones de datos no se permiten con An (registros de direcciones). Esta es la razón para las categorías de direccionamientos tipo "datos". Las siguientes subreglas cubren todos los casos y se dan en orden creciente de categoría, es decir, las reglas posteriores se aplican en caso de conflicto:

   a) Un registro An jamás participa en una operación a nivel de bytes.

   b) An no interactúa con los registros especiales de datos. Excepción: La instrucción MOVES puede mover cualquier registro especial a cualquier registro de datos o direcciones.

   c) Un registro An puede participar en movimientos directos. Así, EXG y MOVE se permiten. MOVE.B queda excluido por la regla 3a. Nótese que An puede moverse desde/hacia los registros especiales de direcciones (al USP, por ejemplo), pero no desde/hacia otros registros de datos especiales (CCR, SR), según establece la regla 3b.

   d) Se pueden comprobar los valores de cualquier registro An (pero no alterarlos de forma permanente). Por tanto, están permitidos CMP (excepto CMP.B), CMPA y TST (excepto TST.B).

   e) Sobre los registros An se permiten las siguientes operaciones: ADD, An1,Dn2 y SUB An1,Dn2 (excepto en tamaño byte), así como ADDA, SUBA, ADDQ y SUBQ (de nuevo, excepto en tamaño byte). En las siguientes instrucciones no están permitidos ADD Dn1,An2 y SUB Dn1,An2. Nótese que ADDI #c,An y SUBI #c,An están excluidos por la regla 2.

4. Las instrucciones LEA, PEA, MOVEM, Jxx y BFxxxx utilizan únicamente direcciones de operandos de control. Excepciones: MOVEM permite (An)+ y −(An) como fuente y destino; BFxxxx permite Dn como un operando de bits.

5. Las instrucciones de cálculo xxxxxxxxxxxxxxxxxxxxxxxxxxxx no están permitidas. La excepción a esta regla la constituye la instrucción BTST #a,#b, que debe ser reemplazada por ANDI u ORI al CCR.

Tabla resumen de las instrucciones del MC68000: Preliminares

La tabla D.2 resume los sets de instrucciones de los cinco microprocesadores en la familia del 68000, es decir, el MC68008, el MC68000, el MC68010, el MC68012 y el MC68020. Los sets del MC68008 y el MC68000 son idénticos, del mismo modo que lo son los del MC68010 y MC68012, de modo que en la tabla sólo es necesario distinguir entre las instrucciones del MC68000, MC68010 y MC68020. Además hay que tener presente la estricta compatibilidad que ha presidido el diseño de la familia del 68000, es decir, la compatibilidad de las instrucciones de los microprocesadores más modernos respecto a las de aquellos diseñados con anterioridad. Por tanto, las instrucciones del MC68000 se ejecutan de la misma forma en los cinco microprocesadores, las instrucciones del MC68010 se ejecutan de la misma forma en el MC68010, MC68012 y MC68020. Las instrucciones del MC68020 son exclusivas de éste.

Por construcción, la tabla D.2 es muy compacta, emplea muchas abreviaturas y no incluye detalles exhaustivos de cómo funciona cada una de las instrucciones. Para detalles acerca de cada una de las instrucciones, véase el capítulo 4. Las siguientes secciones explican los símbolos empleados en cada columna de la tabla D.2.

Columna de instrucciones

La primera columna en la tabla D.2 proporciona los nemónicos estándar de Motorola para cada una de las instrucciones. Cada ensamblador puede, por otra parte, emplear aquellas pequeñas variaciones respecto a estos mnemónicos. Se emplean las siguientes abreviaturas:

Código	*Representa*
.s	Instrucciones de byte, palabra o doble palabra
.s2	Palabra
.s3	Carga de operando, operando de palabra y de palabra larga (sólo para TRAPcc y cpTRAPcc)
cc	Indica un código de condición (CC,CS, EQ, GE, GT, HI, LE, LS, LT, MI, NE, PL, VC, VS)

Columna de la CPU

Esta columna indica cuándo una instrucción es del MC68000 (en blanco), del MC68010 ("10") o del MC68020 ("20"). Véanse las notas anteriores acerca de la compatibilidad en la familia del 68000.

Columna de sintaxis

Esta columna proporciona la sintaxis general para las instrucciones, en la forma en que aparecerán en la familia del 68000. En los casos en los que aparecen los operandos fuente y destino, el operando fuente va en primer lugar. Así, en caso de duda, recordar las instrucciones "MULtiplicar el primer operando por (almacenándolo en) el segundo", "DIVidir el primer operando entre (almacenándolo en) el segundo".

Para algunas instrucciones se proporcionan dos sintaxis. En estas instrucciones siempre hay un bit D o Q (como se explica más abajo) en el código de instrucción. La primera de las sintaxis corresponde al caso en que este bit D o Q esté a 0 y la segunda al caso en que esté a 1.

Las abreviaturas que se emplean en esta columna son:

Código	*Representa*
Dn,Dn1,Dn2	Cualquier registro de datos (de D0 a D7)
An,An1,An2	Cualquier registro de direcciones (de A0 a A7)
Rn,Rn1,Rn2	Cualquier registro de datos o direcciones
PC	Contador de programa (doble palabra)
SR	Registro de estado (palabra)
CCR	Registro de códigos de condición (byte)
SSP	Puntero de pila en modo supervisor (doble palabra)
USP	Puntero de pila en modo usuario (doble palabra)
SP	Puntero de pila activo
RC	Cualquier registro de control (USP, MSP, ISP, VBR, SFC, DFC, CACR, CAAR)
d3	Cualquier número sin signo de 3 bits, es decir, un número de 0 a 7 (excepto para la instrucción BKPT, el 0 representa el valor 8)
d4	Cualquier número sin signo de 4 bits, es decir, de 0 a 15 (un vector *trap*)
d5	Cualquier número sin signo de 5 bits, es decir, de 0 a 31 (un número que identifica la posición de un bit)
d8	Cualquier número sin signo de 8 bits, es decir, de 0 a 255, o bien un número con signo de −128 a +127
d16	Cualquier número sin signo de 16 bits, es decir, de 0 a 65536, o bien un número con signo de −32768 a +32767
d32	Cualquier número sin signo de 32 bits, es decir, de 0 a 4294967295, o bien un número con signo de −2147483648 a +2147483648
#n	Indica #d8 para instrucciones de byte, #d16 para instrucciones de palabra y #32 para instrucciones de doble palabra

Código	*Representa*
ea,ea1,ea2	Direcciones efectivas (véase la tabla D.1 para los 12 modos de direccionamiento). En la mayoría de las instrucciones no todos los modos de direccionamiento están permitidos; las columnas FUE y DAT de la tabla D.2 indican cuáles de los modos son legales (véanse las indicaciones para estas columnas más abajo)
Dc,Dc1,Dc2	Cualquier registro de datos; se emplean en las comparaciones
Du,Du1,Du2	Cualquier registro de datos; se emplean en las actualizaciones
reglist	Cualquier lista de 0 a 16 registros (todos An o Dn) separados por comas
etiqueta	En el código fuente: cualquier etiqueta que se encuentre en cualquier lugar del programa. En el código objeto: un desplazamiento d8, d16 o d32 relativo a la posición indicada por el PC

Cualquier expresión entre llaves (por ejemplo, {#n}) es opcional, es decir, puede estar presente u omitirse. Las comas, paréntesis, puntos o guiones corresponden realmente al código que puede encontrarse en un programa.

Columna de códigos de instrucción

Esta columna contiene una descripción bit a bit de las instrucciones ya ensambladas. Cada bit viene representado por un 1, un 0 o como parte de un campo de bits. Cada campo de bits comienza con una letra mayúscula y puede tener una longitud entre 1 y 32 bits. Los campos de bits empleados se describen a continuación.

Sz	Un campo de 2 bits que indica el tamaño de los operandos (00 = byte, 01 = palabra, 10 = doble palabra).
S	Un campo de 1 bit que indica tramaño (0 = palabra, 1 = doble palabra). Excepción: en el caso de la instrucción CHK se invierte el significado (véase la nota "a" en la tabla D.2).
Siz	Un operando de 3 bits que indica tamaño (010 = palabra, 011 = doble palabra, 100 = ninguna de las dos).
Q	Un indicador de tamaño de 1 bit para las instrucciones MULx y DIVx (0 = doble palabra, 1 = palabra cuádruple).

Fuente	Un indicador de 6 bits que informa del modo de direccionamiento aplicado al operando fuente. Pueden codificarse 12 modos de estos 6 bits (véase la tabla D.1 para un resumen de los mismos).
Destin	Un indicador de 6 bits que informa del modo de direccionamiento aplicado al operando destino.
Tindes	Igual que Destin, pero los tres primeros y los tres últimos bits han alterado sus posiciones. Sólo se emplea en la instrucción MOVE.
Sar	Un campo de 3 bits que indica un registro fuente.
Dar	Un campo de 3 bits que indica un registro de direcciones.
Sdr,Ddr	Un campo de 3 bits que indica un registro de datos (fuente, destino).
Sdar,Ddar	Un campo de 4 bits que indica un registro de datos o direcciones. Se indica un registro de direcciones mediante un 0 seguido de los 3 bits correspondientes al número que lo identifica; análogamente se hace con los registros de datos, pero sustituyendo el 0 por un 1.
D	Un campo de 1 bit que determina cuál de los diferentes campos posibles corresponde al operando fuente y cuál al operando destino. En general, cuando una instrucción contiene un campo D, todos los campos "desde" corresponden a operandos fuente y todos los campos "hacia" corresponden a fuentes. Todos los campos que se ven afectados por el bit D se incluyen en las próximas 8 entradas:
Fefadr	Un campo de 6 bits que indica una dirección efectiva "desde". Es fuente cuando D está a 0 y destino cuando D está a 1.
Tefadr	Un campo de 6 bits que indica una dirección efectiva "hacia". Es destino cuando D está a 0 y fuente cuando D está a 1.
Fdr	Un campo de 3 bits que indica un registro de datos "desde". Se toma como fuente cuando D está a 0 y como destino cuando D está a 1.
Tdr	Un campo de 3 bits que indica un registro de datos "hacia". Se toma como fuente cuando D está a 1 y como destino cuando D está a 0.
Far	Un campo de 3 bits que indica un registro de direcciones "desde". Se toma como fuente cuando D está a 0 y como destino cuando D está a 1.
Tar	Un campo de 3 bits que indica un registro de direc-

	ciones de "hacia". Se toma como fuente cuando D está a 1 y como destino cuando D está a 0.
Fdar	Un campo de 4 bits que indica un registro de direcciones o datos de "desde". Se toma como fuente cuando D está a 0 y como destino cuando D está a 1.
Tdar	Un campo de 4 bits que indica un registro de direcciones o datos "hacia". Se toma como fuente cuando D está a 1 y como destino cuando D está a 0.
Udr	Un campo de 3 bits que indica un registro de datos empleado para actualizaciones (para las instrucciones CAS y CAS2).
Cdr	Un campo de 3 bits que indica un registro de direcciones empleado para actualizaciones (para las instrucciones CAS y CAS2).
Hdr	Un campo de 3 bits que indica la palabra de orden más alto de un registro de datos (para MULx y DIVx).
Imm	Un campo de 3 bits de datos inmediatos.
Immediat	Un campo de 8, 16 ó 32 bits de datos inmediatos.
Displace	Un campo de 8, 16 ó 32 bits que contiene un desplazamiento inmediato.
Cnt	Un campo que contiene un contador de desplazamiento de 3 bits.
Argcont	Un campo de 8 bits que proporciona un argumento para CALLM (1 byte).
Bitno	Un campo de 3 ó 5 bits que indica la posición de un bit (para BTST, BCHG, BCLR y BSET).
Vec	Un campo de 3 bits que contiene un vector para la instrucción BKPT.
Vect	Un campo de 4 bits que contiene un vector para la instrucción TRAP.
Registerlistmask	Un campo de 16 bits que selecciona de 0 a 16 registros para la instrucción MOVEM. Los registros están asociados por orden, cada uno a un bit con D0 asociado al bit 0 y A7 al bit 15, excepto en el caso de direcciones en modo predecrementado en el que se invierte el orden.
T	Un campo de 1 bit que indica el si hay que considerar o no el signo en las instrucciones D0 MULx y DIVx (0 = sin signo, 1 = con signo).
R	Un campo de 1 bit que indica el sentido en que han de efectuarse las rotaciones (0 = derecha, 1 = izquierda).

Bitoff	Un campo de 6 bits que indica un desplazamiento, bien 0 n n n n n, si se emplea un valor de 5 bits, bien 1 0 0 n n n para el caso en que se emplee un registro de datos Dn.
Bitwid	Un campo de 6 bits que indica una distancia (emplea el mismo formato que Bitoff).
Coprocessorcommd	Un campo de 16 bits que contiene una instrucción de un coprocesador.
Cpi	Un campo de 3 bits que identifica a uno de los coprocesadores disponibles.
Cpcond	Un campo de 6 bits que contiene un código de condición de coprocesador.
Controlregis	Un campo de 12 bits que contiene la identidad de uno de los registros de control: 0000 0000 0000 = SFC (68010) 0000 0000 0001 = DFC (68010) 0000 0000 0010 = CACR (68020) 1000 0000 0000 = USP (68010) 1000 0000 0001 = VBR (68010) 1000 0000 0010 = CARR (68020) 1000 0000 0011 = MSP (68020) 1000 0000 0100 = ISP (68020)
Cond	Un campo de 4 bits que contiene un código de condición. Los códigos de condición definidos son: 0000 = Verdadero 0001 = Falso 0010 = Más grande 0011 = Más pequeño/igual 0100 = No hay acarreo 0101 = Acarreo 0110 = No es igual 0111 = Es igual 1000 = No hay rebose 1001 = Hay rebose 1010 = Positivo 1011 = Negativo 1100 = Mayor/igual 1101 = Menor que 1110 = Mayor que 1111 = Menor/igual

Cada una de las instrucciones del 68000 ocupa de 1 a 11 palabras y está formada por un número variable de palabras, de 1 a 3, que constituyen la instrucción básica que van seguidas, cuando procede, por 0 a 10 palabras de extensión. La tabla D.1 lista sólo las palabras básicas. Cualquier conjun-

to de 16 bits encerrado entre paréntesis es opcional; así, por ejemplo, cuando un campo inmediato puede estar formado por 16 ó 32 bits, siempre se mostrarán los 16 finales entre paréntesis. Para cada uno de los 12 modos posibles de direccionamiento, las palabras de extensión de la dirección tienen el mismo formato. La tabla D.1 lista el número de palabras de extensión que se necesitan para la dirección en cada uno de los modos de direccionamiento. Debido a su regularidad, las palabras de extensión de la dirección no se han incluido en la tabla D.2.

Las columnas SRC y DST

Las columnas SRC y DST resumen, en unos cuantos símbolos, los modos de direccionamiento permitidos para los operandos fuente y destino. Los símbolos son consistentes, concisos y fáciles de memorizar.

Si sólo uno de los 12 posibles modos de direccionamiento está permitido, se representa por uno de los siguientes símbolos: Dn, An, (An), (An)+, −(An), d(An), SP, (SP)+, −(SP) o I. Como puede verse, todos los símbolos son autoexplicativos excepto I, que indica un campo inmediato.

Una "Q" indica que el operando es un valor inmediato de 3 bits.

Si hay varios modos de direccionamiento para un operando, se indica esta condición mediante alguno de los siguientes símbolos: EA, REA, DEA, MEA, CEA, AEA, ADEA, AMEA, o ACEA. Estos símbolos ya se han explicado en este capítulo. EA indica que todos los modos de direccionamiento están permitidos.

Los registros especiales se indican mediante los símbolos: CCR, SR, PC y USP, que deben ser autoexplicativos. El símbolo "cr" representa cualquiera de los registros de control: USP, MSP, ISP, VBR, SFC, DFC, CACR y CAAR.

Unos cuantos casos requieren una notación especial:

(Dn)	Representa direccionamiento indirecto por registro de datos.
&	Se emplea para unir grupos diferentes; por ejemplo, "CEA&Dn" indicaría "cualquier dirección de control o cualquier dirección obtenida mediante direccionamiento directo por registro de datos".
~	Se emplea para eliminar partes de un grupo; así, "DEA~I" indica "cualquier DEA excepto direccionamiento inmediato".
,	Empleada para indicar dos operandos : "Dn,An" significa "un operando Dn y un operando An"; "Dn,Dn" significa "dos operandos Dn".

Probablemente, los programadores cuidadosos notarán que en aquellos operandos con sólo un modo de direccionamiento permitido el código del modo de direccionamiento a menudo coincide con los 6 últimos bits de la instrucción. Por ejemplo, la instrucción SWAP usa solamente direccionamiento directo por registros de direcciones, que corresponde al código

000rrrr. Los últimos 6 bits de SWAP son, de hecho, estos 6 bits. Pero, cuidado, estas coincidencias no se dan siempre.

Columna de códigos de condición

Esta columna indica qué códigos de condición se ven afectados por cada instrucción. Los símbolos que se emplean aqui son:

—	El código no se ve afectado por la instrucción.
0	El código se pone a 0.
1	El código se pone a 1.
*	Se cambia el código.
U	Se deja el código con un valor indefinido.

Columna de privilegio

Esta columna indica qué instrucciones son privilegiadas. Las instrucciones privilegiadas sólo se pueden ejecutar cuando el bit de privilegio se pone a 1. De otro modo se generará una instrucción.

Columna de notas

Las letras en esta columna se refieren a las notas a pie de página al final de la tabla.

TABLA D.2.

Resumen de las instrucciones del M68000

Instrucción	*CPU*	*Sintaxis*	*Código de instrucción*	*SRC*	*DST*	*Cód. de condición*	*Privilegio*	*Notas*
			FEDCBA9876543210			XNZVC		
ORI.s		#n,ea	00000000SzDestin Immediatxxxxxxxx (xxxxxxxxxxxxxxxx)	I	ADEA	-**00		
ORI.B (to CCR)		#n,CCR	0000000000111100 00000000Immediat	I	CCR	*****		
ORI.W (to SR)		#n,SR	0000000001111100 Immediatxxxxxxxx	I	SR	*****	X	
CMP2.s	20	ea,Rn	00000Sz011Source Ddar000000000000	CEA	REA	-U*U*		
CHK2.s	20	ea,Rn	00000Sz011Source Ddar100000000000	CEA	REA	-U*U*		
BTST		Dn,ea	0000Sdr100Destin	Dn	DEA	--*--		
BCHG		Dn,ea	0000Sdr101Destin	Dn	ADEA	--*--		
BCLR		Dn,ea	0000Sdr110Destin	Dn	ADEA	--*--		
BSET		Dn,ea	0000Sdr111Destin	Dn	ADEA	--*--		
MOVEP.s2		d16(An1),Dn2	0000Tdr1DS001Far	d(An)	Dn	-----		
		Dn1,d16(An2)	Displacexxxxxxxx	Dn	d(An)			
ANDI.s		#n,ea	00000010SzDestin Immediatxxxxxxxx (xxxxxxxxxxxxxxxx)	I	ADEA	-**00		
ANDI.B (toCCR)		#d8,CCR	0000001000111100 00000000Immediat	I	CCR	*****		
ANDI.W (to SR)		#16,SR	0000001001111100 Immediatxxxxxxxx	I	SR	*****	X	
SUBI.s		#n,ea	00000100SzDestin Immediatxxxxxxxx (xxxxxxxxxxxxxxxx)	I	ADEA	*****		

Instrucción	*CPU*	*Sintaxis*	*Código de instrucción*	*SRC*	*DST*	*Cód. de condición*	*Privilegio*	*Notas*
ADDI.s		#n,ea	00000110SzDestin Immediatxxxxxxxx (xxxxxxxxxxxxxxxx)	I	ADEA	*****		
RTM	20	Rn	000001101100Sdar	REA		*****		
CALLM	20	#d8,ea	00000110llSource 00000000Argcount	CEA		-----		
CAS.s	20	Dc,Du,ea	00001Sz011Destin 0000000Udr000Cdr	Dn	AMEA	-****		
CAS2.s	20	Dc1:Dc2,Du1:Du2,(Rn1:Rn2)	00001Sz011111100 Ddar000Udr000Cdr Ddar000Udr000Cdr	Dn	(An)&(Dn)	-****		[m]
BTST		#d5,ea	00001000000Destin 000000000000Bitno	I	DEA~I	--*--		
BCHG		#d5,ea	00001000001Destin 000000000000Bitno	I	ADEA	--*--		
BCLR		#d5,ea	00001000010Destin 000000000000Bitno	I	ADEA	--*--		
BSET		#d5,ea	00001000011Destin 000000000000Bitno	I	ADEA	--*--		
EORI.s		#n,ea	00001010SzDestin Immediatxxxxxxxx (xxxxxxxxxxxxxxxx)	I	ADEA	-**00		
EORI.B (to CCR)		#d8,CCR	0000101000111100 00000000Immediat	I	CCR	*****		
EORI.W (to SR)		#d16.SR	0000101001111100 Immediatxxxxxxxx	I	SR	*****	X	
CMPI.s	[e]	#n,ea	00001100SsDestin Immediatxxxxxxxx (xxxxxxxxxxxxxxxx)	I	ADEA	-****		
MOVE.s	10	ea,Rn Rn,ea	00001110SzFefadr TdarD000000000000	AMEA [q] REA	REA AMEA [q]	-----	X	
MOVE.B		ea1,ea2	0001TindesSource	DEA	ADEA	-**00		

Instrucción	CPU	Sintaxis	Código de instrucción	SRC	DST	Cód. de condición	Privilegio	Notas
MOVEA.L		ea,An	0010Dar001Source	EA	An	-----		
MOVE.L		ea1,ea2	0010TindesSource	EA	ADEA	-**00		
MOVEA.W		ea,An	0011Dar001Source	EA	An	-----		
MOVE.W		ea,ea2	0011TindesSource	EA	ADEA	-**00		
NEGX.s		ea	01000000SzDestin		ADEA	*****		
MOVE.W (from SR)		SR,ea	01000000 11Destin	CCR	ADEA	-----	[p]	
CHK.s2	[e]	ea,Dn	0100Ddr1S0Source	DEA	Dn	-*UUU		[a]
LEA		ea,An	0100Dar111Source	CEA	An	-----		
CLR.s		ea	01000010SzDestin		ADEA	-0100		
MOVE.W (from CCR)	10	CCR,ea	01000010 11Destin	CCR	ADEA	-----		[i]
NEG.s		ea	01000100SzDestin		ADEA	*****		
MOVE.W (to CCR)		ea,CCR	01000100 11Source	DEA	CCR	-----		[i]
NOT.s		ea	01000110SzDestin		ADEA	-**00		
MOVE.W (to SR)		ea,CR	01000110 11Source	DEA	CCR	-----	X	
NBCD		ea	0100100000Destin		ADEA	*U*U*		
LINK.L	20	An,#d32	0010100000001Dar Displacexxxxxxxx xxxxxxxxxxxxxxxx	I	An,Pc	-----		
SWAP		Dn	0100100001000Ddr		Dn	-**00		
BKPT	10	#d3	0100100001001Vec			-----		[g]
PEA		ea	0100100001Source	CEA	-(SP)	-----		
EXT.s2		Dn	0100100 01S000Ddr		Dn	-**00		
EXTB.L	20	Dn	0100100111000Ddr		Dn	-**00		
MOVEM.s2		reglist,ea ea,reglist	01001D001STefadr Registerlistmask	all REA CEA&(An)+	ACEA&-(An) all REA	-----		

Instrucción	CPU	Sintaxis	Código de instrucción	SRC	DST	Cód. de condición	Privilegio	Notas
TST.s		ea	0100101 0SzDestin		ADEA	-**00		[e]
TST.W/TST.L	20	ea		EA				
TAS		ea	0100101011Destin		ADEA	-**00		
ILLEGAL			0100101011111100			-----		[b]
MULS.L/MULU.L	20	ea,Dn	0100110000Source	DEA	Dn,Dn	-***0		
	20	ea,Dn1:Dn2	0DdrTQ0000000Hdr					
DIVS.L/DIVU.L	20	ea,Dn	0100110001Source	DEA	Dn,Dn	-***0		[v]
	20	ea,Dn1:Dn2	0DdrTQ0000000Hdr					
DIVSL.L/DIVUL.L	20	ea,Dn1:Dn2						
TRAP		#d4	010011100100Vect	[t]	[t]	-----		
LINK.W		An,#d16	0100111001010Dar	I	An,Pc	-----		
UNLK		An	0100111001011Dar	An,(SP)+	SP,An	-----		
MOVE (USP)		An,USP	0100111001 10DSar	An	USP	------	X	
		USP,An		USP	An		X	
RESET			0100111001110000			-----	X	
NOP			0100111001110001			-----		
STOP			0100111001110010		SR	*****	X	
RTE			0100111001110011	(SP)+	SR,PC	*****	X	[s]
RTD	10		0100111001110100	(SP)+ [r]	PC	-----		
RTS			0100111001110101	(SP)+	PC	-----		
TRAPV			0100111001110110	[t]	[t]	-----		
RTR			0100111001110111	(SP)+	CCR,PC	*****		
MOVEC	10	Rc,Rn	0100111001111101D	cr	REA	-----	X	[h]
		Rn,Rc	TdarControlregis					
JSR		ea	0100111010Destin	CEA	PC	-----		
JMP		ea	0100111011Destin	CEA	PC	-----		
ADDQ.s		#d3,ea	0101Imm0SzDestin	Q	AEA [k]	*****		[c]
Scc		ea	0101Cond11Destin		ADEA	-----		

Instrucción	*CPU*	*Sintaxis*	*Código de instrucción*	*SRC*	*DST*	*Cód. de condición*	*Privilegio*	*Notas*
DBcc		Dn,label	0101Cond11001Ddr Displacexxxxxxxx	Dn,I	PC	-----		[d]
TRAPcc.s3	20	{#n}	0101Cond11111Siz (xxxxxxxxxxxxxxxx) (xxxxxxxxxxxxxxxx)	[t]	[t]	-----		[b] [d]
SUBQ.s		#d3,ea	0101Imm1SzDestin	Q	AEA [k]	*****		
Bcc	[f]	label	0110CondDisplace (Displacexxxxxxxx) (xxxxxxxxxxxxxxxx)	I	PC	-----		[e]
BRA	[f]	label	01100000Displace (Displacexxxxxxxx) (xxxxxxxxxxxxxxxx)	I	PC	-----		[e]
BSR	[f]	label	01100000Displace (Displacexxxxxxxx) (xxxxxxxxxxxxxxxx)	I	PC	-----		[e]
MOVEQ(.L)		#d8,Dn	0111Ddr0Immediat	Q	Dn	-**00		
OR.s		ea,Dn Dn,ea	1000TdrDSzFefadr	DEA Dn	Dn AMEA	-**00		
DIVU.W/DIVS.W		ea,Dn	1000DdrT11Source	DEA	Dn	-***0		[v]
SBCD		Dn1,Dn2	1000Ddr100000Sdr	Dn	Dn	*U*U*		
SBCD		−(An1),−(An2)	1000Dar100001Sar	−(An)	−(An)	*U*U*		
PACK	20	Dn1,Dn2,#d16	1000Ddr101000Sdr Adjustmentxxxxxx	Dn	Dn	-----		
PACK	20	−(An1),−(An2),#d16	1000Dar101001Sar Adjustmentxxxxxx	−(An)	−(An)	-----		
UNPK	20	Dn1,Dn2,#d16	1000Ddr110000Sdr Adjustmentxxxxxx	Dn	Dn			
UNPK	20	−(An1),−(An2),#d16	1000Dar110001Sar Adjustmentxxxxxx	−(An)	−(An)	-----		
SUB.s		ea,Dn Dn,ea	1001TdrDSzFefadr	EA [k] Dn	Dn AMEA	*****		

Instrucción	*CPU*	*Sintaxis*	*Código de instrucción*	*SRC*	*DST*	*Cód. de condición*	*Privilegio*	*Notas*
SUBA.s2		ea,An	1001DarS11Source	EA	An	-----		
SUBX.s		Dn1,Dn2	1001Ddr1Sz000Sdr	Dn	Dn	*****		
SUBX.s		−(An1),−(An2)	1001Dar1Sz001Sar	−(An)	−(An)	*****		
User-defined			1010.					[w]
CMP.s		ea,Dn	1011Ddr0SzSource	EA [k]	Dn	-****		
CMPA.s2		ea,An	1011DarS11Source	EA	An	-****		
EOR.s		Dn,ea	1011Sdr1SzDestin	Dn	ADEA	-**00		
CMPM.s		(An1) + ,(An2) +	1011Dar1Sz001Sar	(An) +	(An) +	-****		
AND.s		ea,Dn Dn,ea	1100TdrDSzFefadr	DEA Dn	Dn AMEA	-**00		
MULU.W/MULS.W		ea,Dn	1100DdrT11Source	DEA	Dn	-***0		
ABCD(.B)		Dn1.Dn2	1100Ddr100000Sdr	Dn	Dn	*U*U*		
ABCD(.B)		−(An1),−(An2)	1100Dar100001Sar	−(An)	−(An)	*U*U*		
EXG(.L) (Dn)		Dn1,Dn2	1100Sdr101000Sdr	Dn,Dn		-----		
EXG(.L)		An1,An2	1100Sar101001Sar	An,An		-----		
EXG(.L) (Dn,An)		Dn1,An2	1100Sdr110001Sar	Dn,An		-----		
ADD.s		ea,Dn Dn,ea	1101TdrDSzFefadr	EA [k] Dn	Dn AMEA	*****		
ADDA.s2		ea,An	1101DarS11Source	EA	An	-----		
ADDX.s		Dn1,Dn2	1101Ddr1Sz000Sdr	Dn	Dn	*****		
ADDX.s		−(An1),−(An2)	1101Dar1Sz001Sar	−(An)	−(An)	*****		
ASR.s/ASL.s		#d3,Dn	1110CntRSz000Ddr	Q	Dn	*****		
ASR.s/ASL.s		Dn1,Dn2	1110SdrRSz100Ddr	Dn	Dn	*****		
LSR.s/LSL.s		#d3,Dn	1110CntRSz001Ddr	Q	Dn	***0*		
LSR.s/LSL.s		Dn1,Dn2	1110SdrRSz101Ddr	Dn	Dn	***0*		
ROXR.s/ROXL.s		**#d3,Dn**	**1110CntRSz010Ddr**	**Q**	**Dn**	***0*		

Instrucción	*CPU*	*Sintaxis*	*Código de instrucción*	*SRC*	*DST*	*Cód. de condición*	*Privilegio*	*Notas*
ROXR.s/ROXL.s		Dn1,Dn2	1110SdrRSz110Ddr	Dn	Dn	***0*		
ROR.s/ROL.s		#d3,Dn	1110CntRSz011Ddr	Q	Dn	-**0*		
ROR.s/ROL.s		Dn1,Dn2	1110SdrRSz111Ddr	Dn	Dn	-**0*		
ASR/ASL(.W)		ea	1110000R11Destin		AMEA	*****		
LSR/LSL(.W)		ea	1110001R11Destin		AMEA	***0*		
ROXR/ROXL(.W)		ea	1110010R11Destin		AMEA	***0*		
ROR/ROL(.W)		ea	1110011R11Destin		AMEA	-**0*		
BFTST	20	ea {offset width}	1110100011Destin 0000BitoffBitwid		CEA&Dn	-**00		
BFEXTU	20	ea {offset:width},Dn	1110100111Source 0DdrBitoffBitwid	CEA&Dn	Dn	-**00		
BFCHG	20	ea {offset:width}	1110101011Destin 0000BitoffBitwid		ACEA&Dn	-**00		
BFEXTS	20	ea {offset:width},Dn	1110101111Source 0DdrBitoffBitwid	CEA&Dn	Dn	-**00		
BFCLR	20	ea {offset:width}	1110110011Destin 0000BitoffBitwid		ACEA&Dn	-**00		
BFFFO	20	ea {offset:width},Dn	1110110111Source 0DdrBitoffBitwid	CEA&Dn	Dn	-**00		
BFSET	20	ea {offset:width}	1110111011Destin 0000BitoffBitwid		ACEA&Dn	-**00		
BFINS	20	Dn,ea {offset:width}	1110111111Destin 0SdrBitoffBitwid	Dn	ACEA&Dn	-**00		
cpGEN	20	[cp parameters]	1111Cpi000Destin Coprocessorcommd			-----		[n]
cpScc(.B)	20	ea	1111Cpi001Destin 0000000000Cpcond		ADEA	-----		
cpDBcc(.W)	20	Dn,label	1111Cpi001001Ddr 0000000000Cpcond DisplacexxxxxxxxX	Dn,I	PC	-----		

Instrucción	*CPU*	*Sintaxis*	*Código de instrucción*	*SRC*	*DST*	*Cód. de condición*	*Privilegio*	*Notas*
cpTRAPcc.s3	20	{#n}	1111Cpi001111Siz 0000000000Cpcond (xxxxxxxxxxxxxxxx) (xxxxxxxxxxxxxxxx)			-----		
cpBcc.s2	20	Dn,label	1111Cpi01SCpcond Displacexxxxxxxx (xxxxxxxxxxxxxxxx)	I	PC	-----		
cpSAVE	20	ea	1111Cpi100Destin		ACEA&–(An)	-----	X	
cpRESTORE	20	ea	1111Cpi101Source	CEA&(An)+		-----	X	

[a] El campo de bits S se invierte respecto a las demás instrucciones (0 = doble palabra, 1 = palabra). Esto se debe a que el formato de doble palabra es una característica del M68020 (no pertenece al diseño original).

[b] ILLEGAL tiene el único código de bits que siempre será ilegal. Todos los demás códigos de bits están reservados para futuras expansiones por Motorola. TRAPF no tiene ningún efecto en un programa del MC68000 (como un NOP, pero en el MC68010 causará una excepción de instrucción ilegal). Por tanto, Motorola recomienda colocar una instrucción TRAPF al comienzo de cualquier programa que, diseñado para el MC68020, sea incompatible con el MC68010. Si se intentara ejecutar el programa sobre un MC68010, no producirá problemas graves.

[c] Un valor inmediato de 0 se interpreta como 8 en las instrucciones de desplazamiento, así como en las instrucciones ADDQ y SUBQ.

[d] Los códigos de condición 0000 (T, verdadero) y 0001 (F, falso) no se encuentran disponibles en las instrucciones Bcc y cpBcc. Nótese que BRA tiene un código correspondiente a "BRT" y el de BSR se corresponde con el de "BRF". Dbcc, cpBDcc, Scc, cpScc y TRAPcc permiten el empleo de todos los códigos de condición.

[e] Algunas instrucciones tienen funciones extendidas en el MC68020. Bcc, BRA y BSR permiten desplazamientos de 32 bits; CHK permite extensiones de 32 bits (CHK.L está permitido en el MC68020); CMPI y TST admiten direccionamiento relativo al PC.

[f] Bcc, BRA y BSR permiten desplazamientos de 8, 16 y 32 bits, según sigue:
MC68000: Si el desplazamiento es 0, entonces sigue un desplazamiento de 16 bits.
MC68020: Si el desplazamiento de 8 bits coincide con el código hexadecimal FF, entonces sigue un desplazamiento de 32 bits.

[g] BKPT ha evolucionado como sigue:
MC68000: No implementada.
MC68010: Genera un ciclo de *breakpoint* en el *bus* con el código del bloque de direcciones 111 y una dirección de 32 bits.
MC68020: Genera un ciclo de *breakpoint* en el *bus* con el código del bloque de direcciones en las líneas de dirección A2, A3 y A4 y pone a cero las líneas A0 y A1. Se genera también una respuesta, bien una instrucción de 16 bits, bien una excepción de instrucción ilegal.

[h] MOVEC admite más registros de control en el MC68020 (véase la lista). Para una lista de los registros de control, véanse las explicaciones a la columna del código de instrucciones que preceden a la tabla D.2.

[j] MOVE desde CCR y MOVE al CCR son operaciones de palabras. Ambas requieren direcciones de palabra (pares), pero únicamente actúan sobre el byte CCR. Además, MOVE desde CCR borra el byte alto del destino.

[k] Las operaciones de byte son ilegales con los registros de direcciones.

[m] Nótese que CAS2 permite el direccionamiento indirecto con los registros de datos, es decir, (Dn).

[n] Los modos legales de direccionamiento y las alteraciones de los códigos de condición dependen del coprocesador.

[p] MOVE desde el SR no es una instrucción privilegiada en el MC68000, pero sí en MC68010 y MC68020. Es necesario para que el 68010 pueda soportar las emulaciones. Si un OS emulante funciona en estado de usuario, no debe estar enterado de ello. Para conseguirlo, basta con ejecutar un MOVE desde SR.

[q] MOVES actúa sobre diferentes espacios de direccionamiento, por lo que sus modos de direccionamiento están limitados.

[r] En la instrucción RTD se añade el desplazamiento al SP.

[s] Estrictamente, RTE regresa desde cualquier excepción. RTE recupera desde 4 hasta 44 palabras de SSp o ISp, recupera el PC y el SR y (cuando deba ser) restaura otros registros internos. Nota: Los errores de *bus* y de dirección, en el 68000/68008, generan un formato de pila que no encaja en el esquema general de vuelta del RTE. En este caso, deben restaurarse 4 palabras de la pila antes de proceder al RTE.

[t] Las instrucciones TRAP, TRAPcc y TRAPV generan excepciones y afectan a diversos registros.

[v] Los formatos de la división son:

DIVs.W ea,Dn	Dn(doble)	/ea(palabra)	→	Dn(pal. coc., pal. res)
DIVs.W ea,Dn	Dn(doble)	/ea(doble)	→	Dn(doble pal. coc.)
DIVx.L ea,Dn1:Dn2	Dn1:Dn2(cuadr.)	/ea(doble)	→	Dn1(doble resto), Dn2(doble cociente)
DIVxL.L ea,Dn1:Dn2	Dn1(doble)	/ea(doble)	→	Dn1(doble resto), Dn2(doble cociente)

[w] Las instrucciones que empiecen por los bits 1010 están reservadas para definición por el usuario. Generan interrupciones y reciben diversos nombres, tales como de emulación, definibles-por-usuario, ilegales e indefinidas. En el MC68000/MC68010, las instrucciones que empiecen por 1111 generarán interrupciones similares a las que empiecen por 1010, pero en el MC68020 se emplean para implementar las instrucciones del coprocesador.

Apéndice E

Recursos del M68000

Los siguientes símbolos a la derecha de la lista indican las categorias a las que pertenecen los articulos disponibles de un proveedor determinado (nótese que actualmente están apareciendo nuevas versiones de los productos anteriores basadas en el 68020):

K	*Kits.*
SBC	Ordenadores de una sola placa.
Sys	Sistemas completos.
S	*Software.*
X	Compiladores cruzados.
U	UNIX (o productos relacionados con éste).
Chip	Fuentes secundarias.

1. El FX-688 SBC incorpora un MC68008 y una USART 6551 a un precio de 350 dólares. Incluye 8K EPROM y 2K RAM. Ensambladores cruzados para Apple II/John B. Allen.
2. AM-100L/AM-1000/Work Stations y sistemas multiusuario basados en el MC68000 y bajo AMOS, PC-DOS y UNIMOS. De 5.000 a más de 10.000 dólares.
3. REGULUS OS (UNIX +)/A68KPM SBC, dotado de compatibilidad de *bus* con el LSI-11. Modelo básico de 256K a un precio de 2.900 dólares; el procesador APX cuesta 29.900 dólares y el procesador de I/O 2.345 dólares.

4. Macintosh/Mac Plus/XL/Lisa. Las impresoras láser están controladas por un MC68010 a 10 MHz e incluyen 1 Mb de RAM y 512 KB de ROM.
5. Modelo dual 68000/UNIX. El modelo 1124 puede funcionar sobre UNIX o RM/COS.
6. Dispone de un ordenador personal basado en el MC68000 a 8 MHz, el 520ST a un precio de 800 dólares, con color y sonido incorporado, 512 Kb RAM. Funciona sobre TOS o GEM (ambos son marcas registradas).
7. El PC7300 emplea un 68010, 512 Kb RAM, un *floppy disk* y un disco duro de 10 Mb. Incorpora un *modem* de 1.200 baudios. Precios entre 4.000 y 7.000 dólares.
8. Desarrolla *software* para la mayoria de los microordenadores: DEC/UMDS-10 OS/UMDS-30.
9. CCA EMACS = Editor de textos de pantalla en UNIX. Elisp = extensión del lenguaje *Lisp*.
10. Universe 68/05, dotado de un *bus* de 32 bits *(versabus)*/12,5 mh/MC68000/procesador de entrada-salida. Universe 2203 VME/UNOS = extensión en tiempo real del UNIX system V.
11. CGC 7800. Ordenador con gráficos en color funcionando sobre IDRIS OS.
12. Unisoft Uniplus + /CTS-300/Multibus Intel/MERLIN OS/MODEL 3300. Sistema multiusuario equipado con 84, 33 ó 12 Mb; cuesta 13.500, 9.600 ó 7.600 dólares.
13. AMIGA. Ordenador personal orientado a los negocios o el hogar basado en el MC68000. Dispone de gráficos y sonido. Sistema operativo: AMIGA DOS. Precio: 1.295 dólares.
14. Una amplia variedad de sistemas mono/multiusuario basados en CP/M-68 Kb.
15. Compilador/intérprete de UCSD Pascal 2.0 para el MC6800.
16. Sistema MegaFrame de 16 usuarios basado en un 68010 a 10 MHz, a un precio de 20.000 dólares. Sistema inicial MiniFrame, con un precio de 5.000 dólares. Dispone de UNIX V, Cobol, FORTRAN-77, BASIC, Pascal, C. Véase también el AT&T PC7300.
17. Varios sistemas sobre UNIX V, desde el sistema 100, con 4 Mb, hasta el sistema 300, con 16 Mb.
18. SBC basado en el MC68000/Z80 en kit o montado basado en CP/M 88.
19. CP/M 68K. Versión de CP/M para el M68000.
20. Instrumentación y procesos de control. Modelo 83/80 UNIX/68 KS-8 UNIX V7. Dispone de *Inisoft* en el S-100.
21. Macroensamblador cruzado para el IBM PC a 199 dólares. SBC a 695 dólares, con 16K EPROM/20K RAM/2 RS-232C/Monitor y *debuger*.
22. Segunda fuente del MC68000.
23. ECL-3211 OS/ensamblador/BASIC/C/FORTRAN/Pascal.
24. Micro concurrent Pascal para el M68000/intérprete-kernel 3.2 Kb.

25. Ensamblador cruzado A68K/*linker* L68K/ + LIB68K para CP/M y PC-DOS a un precio de 200-250 dólares. Código fuente para C con un precio de 700 dólares.
26. El FT-68K dispone de Xenix y un controlador de gráficos Multibus.
27. Compilador cruzado de Pascal.
28. Paquete de punto flotante/Compilador de Pascal.
29. Minibox = Unisoft Uniplus + /HK68 = SBC con DR CP/M 68K. PolyForth/Regulus/Multibus con seis *slots* de expansión.
30. HP 9826A/HP9000/200/HP-UX OS. Portable UNIX en el PC integral a un precio de 4.990 dólares. Incluye HP-UX, PAM, HP Windows.
31. Fuente secundaria autorizada para el M6800/68000. Se espera una versión CMOS pronto = HD68000 en el empaquetamiento normal de 64 *pines* DIP. Además se esperan versiones en empaquetamiento plano para microordenadores de bajo consumo/portátiles.
32. Sistema basado en el M68000 sobre Unisoft Uniplus + . Workstation = MicroSystems NX a un precio comprendido entre los 8.895-9.500 dólares.
33. IBC Ensign = Unisoft Uniplus + . Sistema multiusuario de 32 puestos/8 Mb.
34. Modelo 9000 básico con 128 Kb RAM/128 Kb ROM/compatible con el VERSABUS/interfaz IEEE-488/Disco duro opcional de 4 × 5 Mb o 4 × 10 Mb a un precio de 56.000 dólares.
35. Versión del Sinclair QL sobre el MC68008 con *modem*/teléfono,
36. Sistema multiusuario con 16 puestos a 9.995 dólares. OEM 8 usuarios/500K 40 Mb/Mirage OS/S-100. Placa del procesador MC6800 a 695 dólares. Versión del MC68010 a 795 dólares. Placa de entrada-salida Matching a 695 dólares.
37. Tech Publishing Systems TPS-2000 sobre el MC68010. Más de 37.000 dólares.
38. Ensambladores cruzados/pasacles/c/para todos los M68000. Corre sobre VAX/VMS/UNIX. También sobre Apollo/Sun.
39. MTOS-68K/ensamblador/C/Pascal/sistema monousuario de 8K. MTOS-68KF ROM para el SBC Omnibyte/Microbar MC68000.
40. Compilador de Lattice C para todos los MC68000.
41. Compiladores y compiladores cruzados de C para todos los sistemas MC68000 basados en UNIX.
42. Sistema operatico COHERENT OS (rm) multiusuario-multitarea commpatible con UNIX. Compiladores y compiladores cruzados en C para el M68000/PDP-11/Z8000/8086.
43. Micro/32 corriendo sobre el sistema operativo REGULUS OS.
44. LISP para crear sistemas expertos sobre el 68000.
45. *Miniframe* 68K sobre XENIX. Emplea la MMU MC6809. Existen planes para VENIX (versión VM, con memoria virtual).
46. Sistema de animación para programación visual/Sistema de depuración en COBOL.

47. XENIX = versión del UNIX v7/ensambladores/BASIC/C/FORTRAN-77/COBOL-74 para el M68000 y el 8086/8088.
48. OS-9 = Sistema operativo tipo UNIX. Funciona sobre la mayoría de los sistemas basados sobre el 6809 y 68000 en una amplia gama de tamaños, que van desde los sistemas basados en control por ROM hasta los sistemas de medio tamaño en tiempo compartido. C/Pascal/BASIC/COBOL.
49. TRICEP = con un precio de unos 2.500 dólares por usuario, es un sistema multiusuario de 4 a 8 usuarios/Unisoft Uniplus/UNIX V/512K, 4CRTs, disco duro de 16 Mb a 10.495 dólares. Con un disco de 32 Mb a 12.495 dólares.
50. Fuente secundaria autorizada.
51. VERSAdos/Macroensamblador estructurado/Pascal/FORTRAN-77. Un amplio rango de sistemas de desarrollo *software*.
52. Libros/Literatura técnica.
53. EXORciser —módulo de diseño MEX68KDM—. Monitor de 8K MACSbug. VERSAbus/VMM-micro monotarjeta Versamodule. VME bus/subconjunto del versabus.
54. S1: sistema operativo basado en el M68000 para DataMedia 932/IBM 9000/NCR Tower/Compupro 68K/Dual CPU/IBC Ensign/Stride/SORD 68/Pertec 32000/FORCE. Amplio rango de ensambladores cruzados y compiladores para el M68000 y para la mayoría de los micros.
55. *Software* LAN (redes de comunicación locales) para interconectar el M68000/DEC-VEXen/IBM-PC y más.
56. Herramientas de desarrollo cruzado, C y macroensambladores.
57. SBC OB68K1.
58. Pascal nativo (Pascal 2) y cruzado para todos los sistemas 68000 basados en UNIX.
59. Amplia variedad de sistema mono/multiusuario.
60. Modelo 3220 con multiproceso/5 usuarios/53 Mb.
61. OASIS OS, basado sobre el INTEL 8086/8088 y el M68000/32 usuarios.
62. Ensamblador cruzado para el Apple II/II + (32K) a MC68000 a 95 dólares. MINOS 1,0 OS para Apple II + /Apple//e y la placa 68000 AP proporcionan acceso directo al control del 68000.
63. *Pixel* 80, funcionando a 10 MHz, con UniSoft UniPlus + .
64. Modelo P/35 y un amplio rango de *workstations*.
65. Gran variedad de sistemas multiusuario para negocios y para fines científicos.
66. Ordenador personal TRS-16 para usos profesionales y de negocios.
67. INFORMAX/File-it!/C-ISAM en la mayoría de los sistemas que, basados en el M68000, funcionan sobre UNIX.
68. INGRES: Sistema de manejo de una base de datos relacional para el MC68000. Basado sobre UNIX.
69. ICEBOX/SPICE, sistemas de desarrollo *software*. Existe una gran

variedad de ensambladores y ensambladores cruzados para Intellec e iPDS.

70. Fuente secundaria autorizada.
71. Compilador RMCOBOL para los sistemas M68000, funcionando sobre UNIX.
72. SBE350/300/200, sistemas multiusuario que funcionan sobre el sistema Regulus OS (véase Alycon)/LEX68. Gran variedad de *software* y de sistemas SBC basados sobre el M68000. PROBUG para el M68K10 o M68K12 o M68CPU (rm). Placas SBC para el multibus IEEE796 con 1 MB de RAM.
73. Fuente secundaria autorizada.
74. *Workstation* gráfica, sobre UNIX y M68000.
75. Familia de accesorios UNIX = VAR/68K (MC68008 + Regulus OS). Precio aproximado, 7.900-25.000 dólares.
76. IPT (herramientas integradas de producción) para el NCR Tower, 1632 y Plexus P/53.
77. MC6800 Versabus/microtarjeta para emplear XENIX en un IBM XT. Precio, 1.995-2.995 dólares, incluyendo el *software*.
78. Serie Stride 400. Sistemas multiusuario, multisistema operativo/ VMEbus. Precio entre 2.900 y 60.000 dólares.
79. *Workstation* modelo 2/250 sobre UNIX. Equipa un M68000 a 10 MHz. Precio, menos de 10.000 dólares. Gran variedad de sistemas multiusuario VM (por ejemplo, modelo 2/120 = 1 Mb, modelo 2/170 = 2 Mb). Gráficos. Disponibilidad de red Ethernet a 10 Mb/segundo.
80. Serie 70/400 = multiusuario. Incluye un M68000 a 12 MHz sobre un sistema operativo MIRAGE OS.
81. Ensamblador cruzado/compilador de Pascal para el DEC PDP-11 y sistemas M68000.
82. Ensamblador cruzado y macroensamblador UNIflex/BASIC/Pascal/C.
83. Disco duro basado en el M68000. Puede conectarse a la red LAN AppleTalk y a la impresora ImageWriter, así como a algunos *modems*.
84. Sistemas de desarrollo para Pascal y ADA sobre M68000.
85. Sistema NU Machine = MC68010 a 10 MHz sobre UNIX. Datos de 32 bits, capaz de direccionar 37,5 Mb/s. *Bus* de banda ancha NuBUS con convertidor de Multibus bajo el sistema operativo de Motorola V/68 OS.
86. Editor de pantalla EMACS/MINIMACS. Editores multiventana para todos los sistemas basados en el M68000. Amsterdam compiler Kit = C; compiladores de Pascal y compiladores cruzados para el M68000 y el Intel 8086/8088 bajo UNIX.
87. Versión portable del UNIX implantada en unos 90 microordenadores de 60 fabricantes. El sistema Unisoft + está disponible para la mayoría de los sistemas basados en el M68000. Ensambladores ASM68 y ensambladores cruzados para el VAXen, etc. IP/TCP,

posibilidad de acceder a redes de comunicaciones, grabación de ficheros y control en exclusividad de los mismos.

88. Unisoft Uniplus + /el *bus* VME equipa un M68000 256K de RAM con doble puerto de entrada-salida. Emplea un sistema operativo VRTX.
89. Compiladores de Pascal y C para sistemas basados en el M68000, y sobre UNIX o Versados.
90. *Workstations* en UNIX. Sistemas multiusuario bajo Unisoft Uniplus + /MCS OS/150WS.
91. MAINSAIL = *software* de aplicación para el desarrollo de lenguajes para sistemas basados en el M68000 y UNIX (también admite otros sistemas operativos).

Apéndice F

Tabla de conversiones entre ASCII y distintos sistemas de numeración

DEC X_{10}	HEX X_{16}	OCT X_8	Binario X_2	ASCII
0	00	00	000 0000	NUL
1	01	01	000 0001	SOH
2	02	02	000 0010	STX
3	03	03	000 0011	ETX
4	04	04	000 0100	EOT
5	05	05	000 0101	ENQ
6	06	06	000 0110	ACK
7	07	07	000 0111	BEL
8	08	10	000 1000	BS
9	09	11	000 1001	HT
10	0A	12	000 1010	LF
11	0B	13	000 1011	VT
12	0C	14	000 1100	FF
13	0D	15	000 1101	CR
14	0E	16	000 1110	SO
15	0F	17	000 1111	SI
16	10	20	001 0000	DLE
17	11	21	001 0001	DC1
18	12	22	001 0010	DC2
19	13	23	001 0011	DC3

DEC X_{10}	HEX X_{16}	OCT X_8	Binario X_2	ASCII
20	14	24	001 0100	DC4
21	15	25	001 0101	NAK
22	16	26	001 0110	SYN
23	17	27	001 0111	ETB
24	18	30	001 1000	CAN
25	19	31	001 1001	EM
26	1A	32	001 1010	SUB
27	1B	33	001 1011	ESC
28	1C	34	001 1100	FS
29	1D	35	001 1101	GS
30	1E	36	001 1110	RS
31	1F	37	001 1111	US
32	20	40	010 0000	SP
33	21	41	010 0001	!
34	22	42	010 0010	"
35	23	43	010 0011	#
36	24	44	010 0100	$
37	25	45	010 0101	%
38	26	46	010 0110	&
39	27	47	010 0111	'
40	28	50	010 1000	(
41	29	51	010 1001	)
42	2A	52	010 1010	*
43	2B	53	010 1011	+
44	2C	54	010 1100	,
45	2D	55	010 1101	-
46	2E	56	010 1110	.
47	2F	57	010 1111	/
48	30	60	011 0000	0
49	31	61	011 0001	1
50	32	62	011 0010	2
51	33	63	011 0011	3
52	34	64	011 0100	4
53	35	65	011 0101	5
54	36	66	011 0110	6
55	37	67	011 0111	7
56	38	70	011 1000	8
57	39	71	011 1001	9
58	3A	72	011 1010	:
59	3B	73	011 1011	;
60	3C	74	011 1100	<
61	3D	75	011 1101	=

DEC X_{10}	HEX X_{16}	OCT X_8	Binario X_2	ASCII
62	3E	76	011 1110	>
63	3F	77	011 1111	?
64	40	100	100 0000	@
65	41	101	100 0001	A
66	42	102	100 0010	B
67	43	103	100 0011	C
68	44	104	100 0100	D
69	45	105	100 0101	E
70	46	106	100 0110	F
71	47	107	100 0111	G
72	48	110	100 1000	H
73	49	111	100 1001	I
74	4A	112	100 1010	J
75	4B	113	100 1011	K
76	4C	114	100 1100	L
77	4D	115	100 1101	M
78	4E	116	100 1110	N
79	4F	117	100 1111	O
80	50	120	101 0000	P
81	51	121	101 0001	Q
82	52	122	101 0010	R
83	53	123	101 0011	S
84	53	124	101 0100	T
85	55	125	101 0101	U
86	56	126	101 0110	V
87	57	127	101 0111	W
88	58	130	101 1000	X
89	59	131	101 1001	Y
90	5A	132	101 1010	Z
91	5B	133	101 1011	[
92	5C	134	101 1100	\
93	5D	135	101 1101	]
94	5E	136	101 1110	^
95	5F	137	101 1111	–
96	60	140	110 0000	`
97	61	141	110 0001	a
98	62	142	110 0010	b
99	63	143	110 0011	c
100	64	144	110 0100	d
101	65	145	110 0101	e
102	66	146	110 0110	f
103	67	147	110 0111	g

DEC X_{10}	HEX X_{16}	OCT X_8	Binario X_2	ASCII
104	68	150	110 1000	h
105	69	151	110 1001	i
106	6A	152	110 1010	j
107	6B	153	110 1011	k
108	6C	154	110 1100	l
109	6D	155	110 1101	m
110	6E	156	110 1110	n
111	6F	157	110 1111	o
112	70	160	111 0000	p
113	71	161	111 0001	q
114	72	162	111 0010	r
115	73	163	111 0011	s
116	74	164	111 0100	t
117	75	165	111 0101	u
118	76	166	111 0110	v
119	77	167	111 0111	w
120	78	170	111 1000	x
121	79	171	111 1001	y
122	7A	172	111 1010	z
123	7B	173	111 1011	{
124	7C	174	111 1100	|
125	7D	175	111 1101	}
126	7E	176	111 1110	~
127	7F	177	111 1111	DEL

Indice alfabético