

Arquitectura de Computadores

Lenguaje Máquina del Motorola 68000

Paco Aylagas (D-4416)

Tutorías: Mañanas

Tardes: (L y V)

Tel. 913 365 058

e-mail: paylagas@eui.upm.es

Bibliografía:

Microprocessor Systems Design (2nd. ed.)

Alan Clements

PSW-KENT Publishing Company

<http://www.dia.eui.upm.es>

Tablón de anuncios - Bloque IV, 4ª planta

- El contenido de este seminario comprende dos temas del programa, por lo que es materia de examen.
- No hay que saber el 68000 de memoria
- Hay conceptos nuevos que no se conocen. Se irán aprendiendo poco a poco.
- NO seguir los apuntes. El orden está mezclado.
- Se me puede interrumpir en cualquier momento.

Arquitectura de Computadores

Lenguaje Máquina MC68000

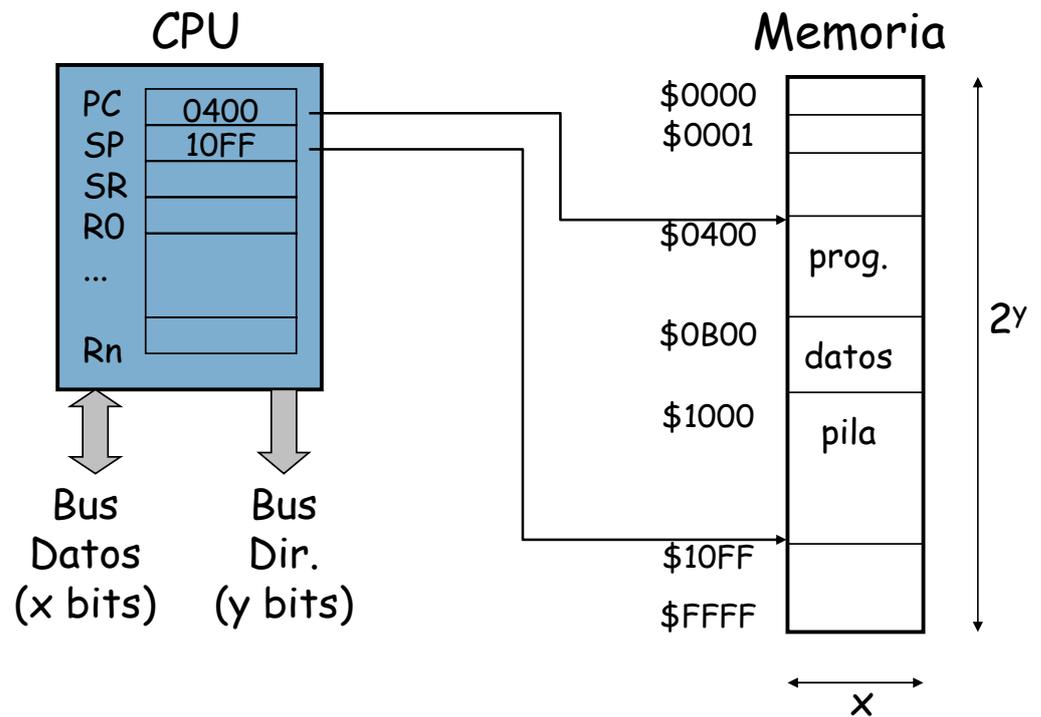
1. Estructura del MC68000
2. Formato de las Instrucciones
3. Modos de Direccionamiento
4. Juego de Instrucciones
5. Excepciones e Interrupciones
6. Ejemplos

En el capítulo anterior hemos visto la interfaz hardware que ofrece la CPU, esto es, los elementos que ofrece para conectarla con el resto de los componentes de un ordenador. Ahora abordaremos la interfaz software, o lo que es lo mismo, lo que necesitan saber los programadores o constructores de compiladores para poder escribir programas capaces de ejecutarse en la CPU.

Esta interfaz software muestra aspectos como el repertorio y formato de las instrucciones que ofrece, con sus variados modos de direccionamiento para acceso a los operandos. Pero para poder escribir programas para una CPU también se necesita conocer otros aspectos como los registros internos, estructura o visión de la memoria y los tipos de datos que maneja directamente la ALU.

Después de dar una introducción general a estos aspectos, se verá con cierto detalle un caso concreto: el procesador 68000 de Motorola.

Visión del Hardware



El programador más inmediato de un ordenador trabaja a muy bajo nivel, pues ve el hardware muy de cerca. Los programas los tiene que escribir en lenguaje máquina o, normalmente, en ensamblador, es decir, una representación nemotécnica del lenguaje máquina en la que cada instrucción ensamblador se corresponde con una instrucción máquina.

¿Qué tiene que conocer este programador de ensamblador o de lenguaje máquina para escribir programas para el ordenador? La respuesta es "Algo de hardware y algo de software".

1) Algo de **hardware**:

- Algunos detalles de la CPU:
 - Registros generales y de control (Contador de Programa, Puntero de Pila, Registro de Estado, ...).
 - Tamaño de los registros generales, para saber el rango de los números que se pueden manejar.
- Tamaño de la memoria

Visión del Software

Juego de Instrucciones

Nemotécnico	Cod. Máquina
move #3,R2	11000111 (\$C7)
add R2,R3	00110110 (\$36)
...	
...	
sub #1,R2	01100110 (\$66)

Pila

E1
50
2B
9A

LIFO | - Meter
- Sacar

2) Algo de **software**:

- El juego de instrucciones
- Organización de la pila (si la hay).

Estos elementos de hardware y de software son con los que cuenta el programador para escribir sus programas, o lo que es lo mismo, ésta es la visión del programador del ordenador básico.

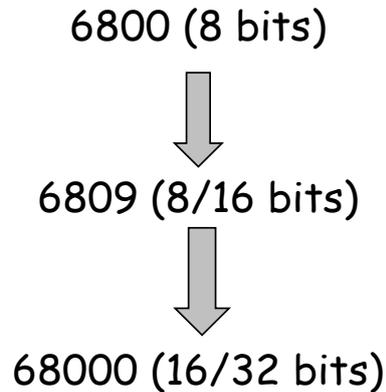
En resumen, cuenta con una memoria donde almacena su programa (con las instrucciones y los datos) y una CPU que es capaz ejecutar las instrucciones del programa, ayudándose para ello de registros internos a los que trae los operandos desde la memoria principal, y de una pila para guardar ciertos datos temporales.

Puesto que ya conocemos la estructura básica de una CPU, en este capítulo vamos ocuparnos del lenguaje máquina, es decir, del formato que pueden tener las instrucciones máquina, de los modos de direccionamiento disponibles para acceder a los operandos indicados en las instrucciones y, por supuesto, de los distintos tipos de instrucciones que suelen ofrecer los procesadores de propósito general (al menos los de tipo CISC). Al final del capítulo particularizaremos todo esto para el microprocesador de Motorola MC68000, comentando su estructura general, organización de la memoria, su pila y, por supuesto, su juego de instrucciones.

MC 68000

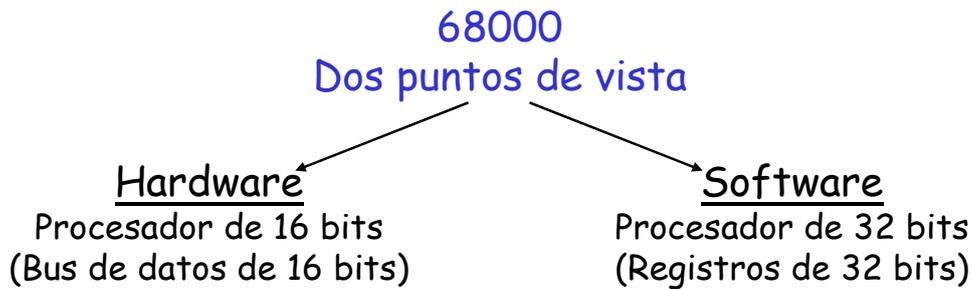
Una Familia de Microprocesadores

OBJETIVO: Compatibilidad Hw. y Sw.



El 68000 es el primer componente de una familia de procesadores, la 68000, en la que se consigue compatibilidad (hacia arriba) tanto en el hardware como en el software.

A su vez, la familia 68000 está diseñada para ofrecer cierta compatibilidad con los componentes de la anterior familia, la 6800.



Año	Chip	Reloj (MHz)	Bus Datos	Bus Direcciones	Capacidad de Memoria
1979	68000	8-20	16	24	16 Mb
1982	68008	8	8	20/22	1/4 Mb
1983	68010	12,5	16	24	16 Mb
1983	68012	12,5	16	30/31	1/2 Gb
1984	68020	12-33	32	32	4 Gb
1987	68030	16-50	32	32	4Gb
1989	68040	25-40	32	32	4 Gb
1994	68060	50	32	32	4 Gb

Desde el punto de vista hardware, el **68000** es un procesador de 16 bits, pues es el ancho de su bus de datos. No obstante, al programador se le ofrece realmente un procesador de 32 bits, que es el tamaño de sus registros generales. Para simular la gestión de datos de 32 bits, el 68000 realiza dos ciclos de lectura o escritura de 16 bits. Con sus 24 bits de direcciones puede direccionar hasta 16 Mbytes. Le sucedió el **68010**, que incluía un cierto soporte para la gestión de memoria virtual.

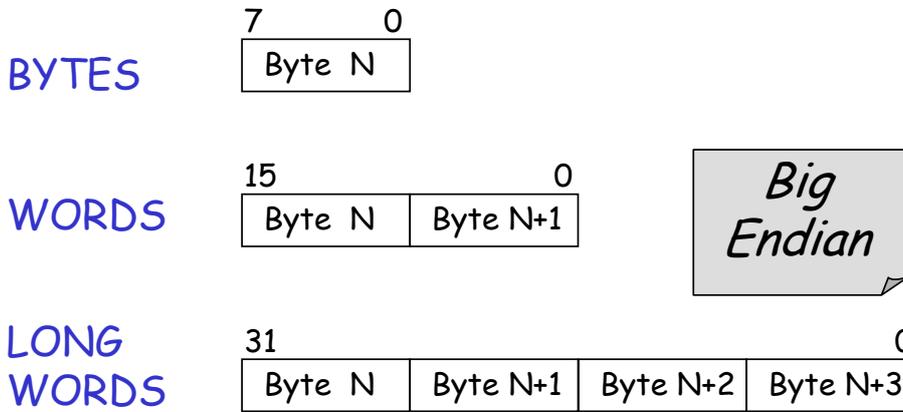
Un hito importante en la familia fue el **68020**, un procesador de 32 bits tanto en el bus de direcciones como de datos, y una caché de instrucciones de 256 bytes. La mejora de este modelo fue el **68030**, que incluía una MMU (Unidad de Gestión de Memoria) completa. Disponía de dos cachés separadas para instrucciones y para datos.

El **68040** añadió un coprocesador para la aritmética de coma flotante, y el tamaño de las cachés se aumentó a 4 Kbytes.

A lo largo de los años se ha ido evolucionando hasta que la saga terminó con el **68060** en el año 1994. El 68060 es un procesador superescalar (dos unidades de ejecución) y segmentado. Las cachés de instrucciones y datos son de 8 Kbytes cada una.

Posteriormente Motorola dejó la fabricación de microprocesadores, interviniendo únicamente en la alianza con IBM y Apple para la construcción del PowerPC.

La Memoria Principal es una serie de ceros "0" y unos "1" agrupados en:



... formando un espacio lineal de direcciones

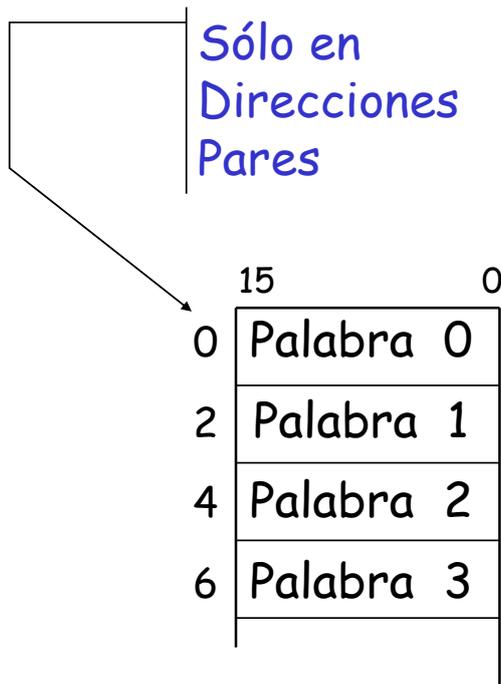
Dir. Hex	Memoria Principal
000000	0 B
1	2 0
2	3 A
3	F 4
4	F C
5	F 1
6	E 0
7	C 8
...	
...	
FFFFFFC	??
FFFFFFD	??
FFFFFFE	??
FFFFFFF	??

La memoria que ve el 68000 es una colección de unos y ceros agrupados formando datos de distintos tamaños.

Aunque el 68000 es un procesador orientado a palabras (*words*) de 16 bits, y con posibilidad de acceder a datos de 32 bits (dobles palabras o *long words*), su memoria es direccionable a nivel de byte. Esto quiere decir que puede direccionar y trabajar con unidades de 8, 16 y 32 bits con la misma facilidad

En cualquier caso, el 68000 ve un espacio lineal de direcciones, donde cada dirección corresponde, al menos, a un byte, aunque también puede corresponder a una palabra o a una doble palabra.

Como se muestra en la diapositiva, Motorola utiliza el protocolo de ordenación en memoria conocido como **big endian**, según el cual, para las palabras y las dobles palabras, se almacena el byte más significativo del número en la dirección más baja de memoria. Esto es equivalente al orden normal de escritura de los lenguajes occidentales.



Para acceder a esta memoria, dispone de:

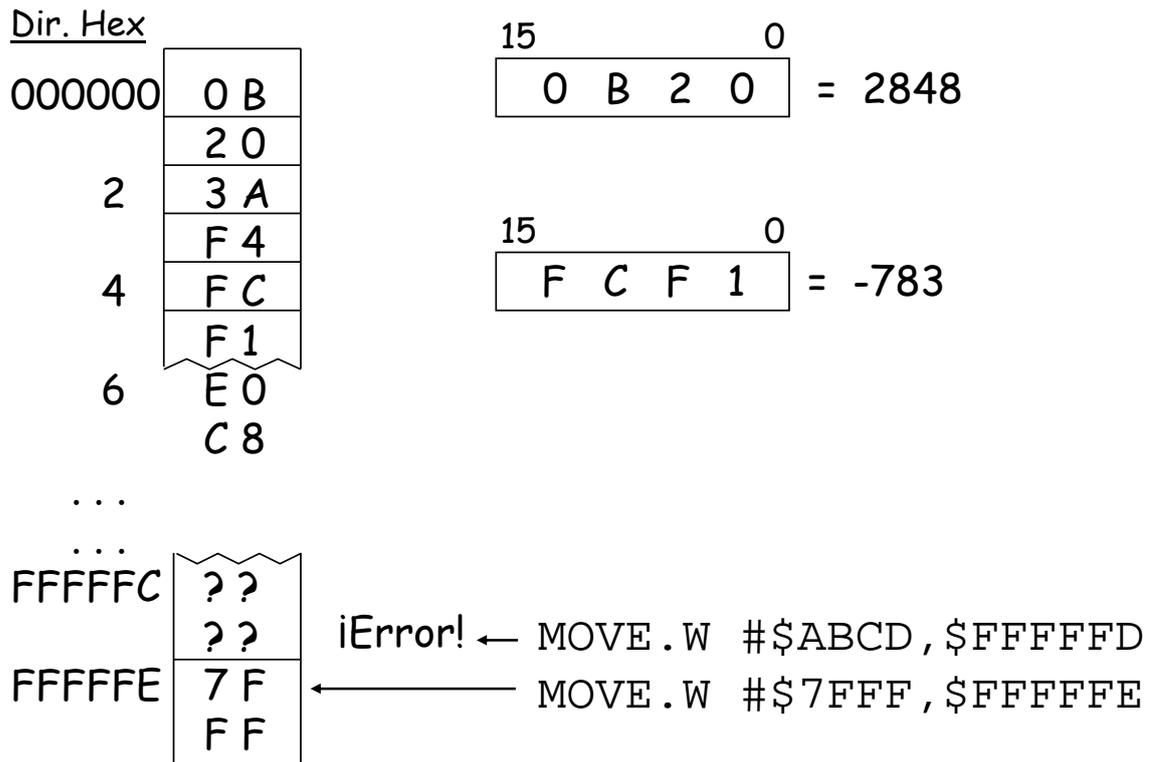
☞ 23 hilos de direcciones (A_1-A_{23})

8 Mpalabras de 16 bits

Las palabras están formadas por dos bytes consecutivos que **obligatoriamente deben comenzar en una dirección par** de memoria.

Aunque los registros de direcciones del 68000 son de 32 bits, solamente puede direccionar un espacio de 16 Mbytes de memoria (2^{24}) y por eso el Contador de Programa (PC) es de 24 bits.

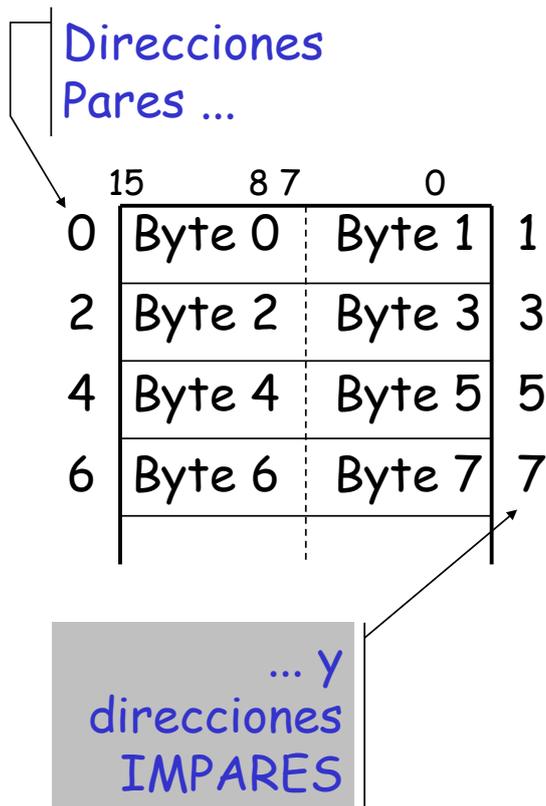
Por su orientación al trabajo con palabras, este procesador solamente cuenta con 23 patas de direcciones (A_1-A_{23}). Ya que las palabras solamente pueden estar en direcciones pares, se considera que la señal A_0 está siempre a cero. Esto quiere decir que, por ahora, el 68000 ve hasta 8 Mpalabras (2^{23} palabras) de 16 bits.



En la diapositiva podemos ver los contenidos de algunas palabras (words) de memoria. En la dirección 0 está el valor binario 0B20, cuyo equivalente en decimal (en complemento a 2) es 2.848.

El valor binario de la palabra ubicada en la dirección 4 es FCF1, que interpretado como un valor en complemento a 2 equivale a -783 en base 10.

Obsérvese cómo una instrucción que intenta escribir una palabra en la dirección FFFFFD (impar), produce un error.



Para acceder a esta memoria, dispone de:

- ☞ 23 hilos de direcciones (A_1-A_{23})
-
- 8 Mpalabras de 16 bits
- ☞ 2 hilos de selección de byte (UDS, LDS)

Ya habíamos dicho que la resolución del direccionamiento del 68000 es de byte, es decir, que a cada byte de memoria le corresponde una de las hasta 2^{24} direcciones posibles.

Sin embargo, hemos visto que mientras que los registros de direcciones y el Contador de Programa (PC) pueden formar direcciones de 24 bits, el 68000 solamente cuenta con 23 hilos de direcciones: A_1-A_{23} . Entonces ¿Cómo se puede seleccionar de forma individual cada uno de los dos bytes de la palabra referenciada con las señales A_1-A_{23} ?

La respuesta está en dos hilos adicionales que salen del 68000: el UDS (*Upper Data Strobe*) y el LDS (*Lower Data Strobe*).

Cuando el 68000 hace referencia a un byte de dirección par, se activará automáticamente la señal UDS, si la dirección es impar, se activará LDS. Si se hace referencia a una palabra completa, se activarán las dos señales, UDS y LDS, simultáneamente.

Mientras que una palabra tiene que estar en una dirección par, **un byte puede estar en una dirección par o impar.**

Dir. Hex

000000	0 B
1	2 0
2	3 A
3	F 4
4	F C
5	F 1
6	E 0
7	C 8

$$\begin{array}{c} 7 \quad 0 \\ \boxed{0 \ B} = 11 \end{array}$$

$$\begin{array}{c} 7 \quad 0 \\ \boxed{F \ 1} = -15 \end{array}$$

...

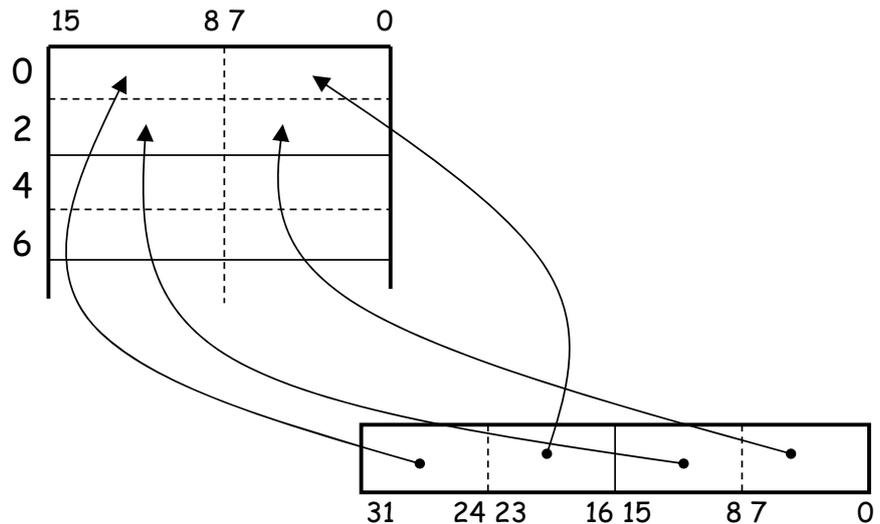
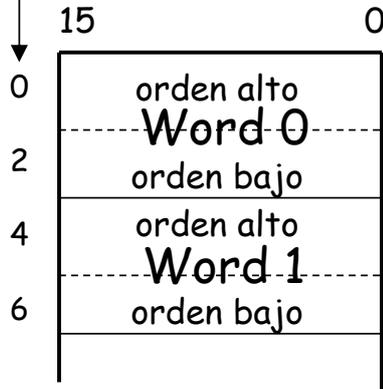
...

FFFFFFC	0 7	←	MOVE.B #7, \$FFFFFFC
FFFFFFD	? ?		
FFFFFFE	? ?		
FFFFFFF	F B	←	MOVE.B #\$FB, \$FFFFFFF

El contenido de un byte consta de 8 bits, por lo que puede tomar valores entre 0 y 255. Estos valores binarios puros se deben interpretar según lo indique el programa en el que se utilizan, para darles su verdadero significado. Así, el contenido F1 interpretado como un entero sin signo corresponde al valor decimal 241, mientras que si se interpreta como un entero con signo en complemento a 2, indica el valor -15.

Obsérvese que los valores de los bytes pueden ubicarse en cualquier dirección de memoria, tanto par como impar.

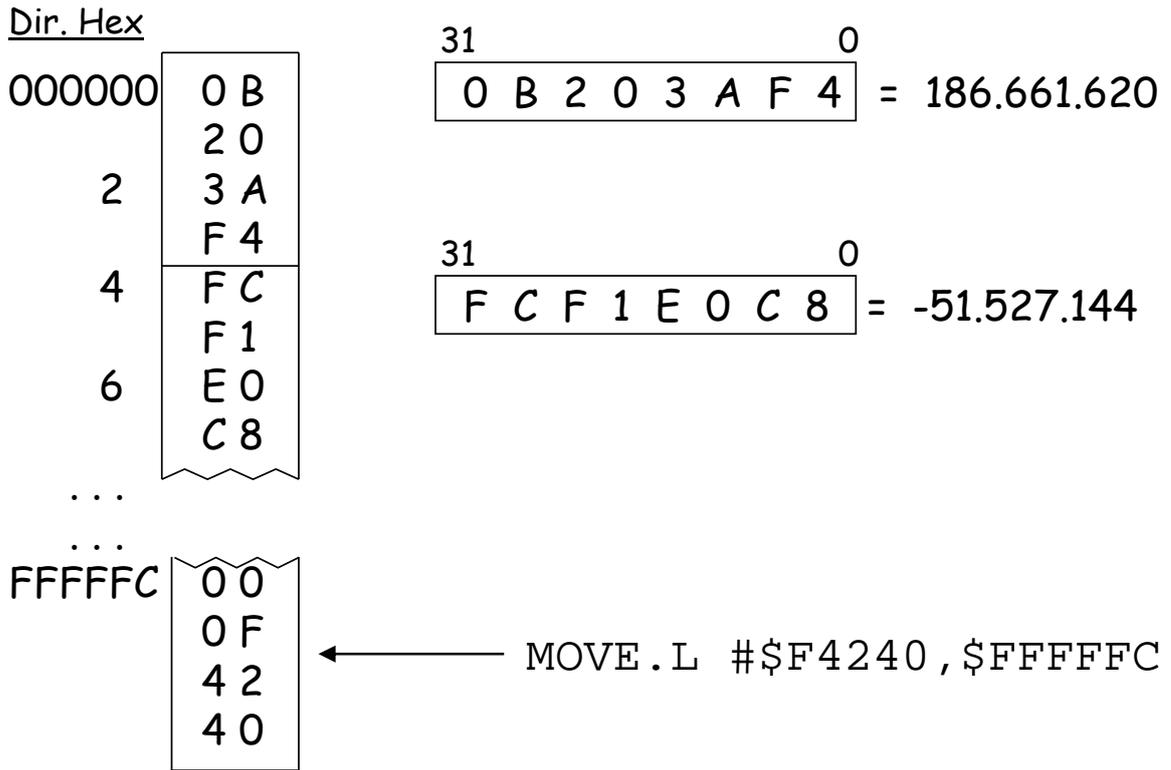
En cualquier dirección PAR



Almacenamiento de una Doble Palabra en memoria

Aunque las dobles palabras están formadas por cuatro bytes consecutivos, su dirección no tiene que ser múltiplo de 4, sino que basta con que esté situada en cualquier dirección par (excepto la última palabra de la memoria, que sí tendrá que estar ubicada en una dirección múltiplo de 4).

Como se puede ver en la transparencia, debido al orden *Big Endian*, dentro de cada doble palabra, la palabra de dirección más baja es la de mayor peso, y el byte más significativo de cada palabra está en la dirección más baja de memoria.



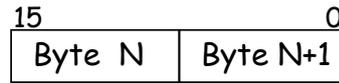
Aunque en los ejemplos de la transparencia se muestran algunos valores negativos (en complemento a dos) almacenados en memoria, en realidad simplemente está almacenada una ristra de bits, y su interpretación como número entero o sin signo depende de cada instrucción.

La Memoria Principal es una serie de ceros y unos agrupados en

BYTES

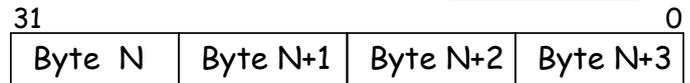


WORDS



Big Endian

LONG WORDS



Dir. Hex Mem. Pr.

000000 0 B 0 B

1 2 0

2 3 A

3 F 4

4 F C

5 F 1

6 E 0

7 C 8

...

...

FFFFFFC 0 0 0 0 F 4 2 4 0

FFFFFFD 0 F

FFFFFFE 4 2

FFFFFFF 4 0

... formando un espacio lineal de direcciones, donde coexisten distintos tipos de datos

En el espacio de memoria pueden coexistir simultáneamente distintos tipos de datos (*bytes*, *words* y *long words*).

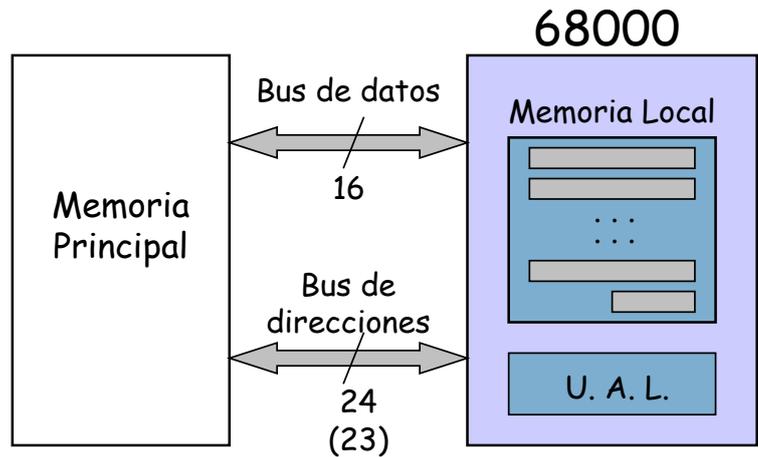
Recuérdese que las dobles palabras (*long words*) no tienen por qué estar ubicadas necesariamente en direcciones múltiplos de 4, sino que pueden estar situadas en cualquier dirección par.

☞ Es Memoria Alterable Local al Chip

- Muy pequeña (74 bytes)
- De acceso muy rápido

☞ En Total, 19 Registros

- 18 de 32 bits
- 1 de 16 bits



Este procesador cuenta con una pequeña memoria interna (74 bytes) estructurada en 19 registros accesibles al programador. Estos registros son de lectura/escritura, aunque como ya veremos, en algunos de ellos el acceso está restringido en ciertas situaciones.

Estos 19 registros tienen diversos cometidos, por ello hay:

- Registros de datos
- Registros de direcciones
- Registros de propósito especial:
 - Contador de Programa (PC)
 - Registro de Estado (SR)

Pasemos a tratar con cierto detalle cada uno de estos tipos de registros.

8 Registros de Datos (D0 .. D7)

	31	16 15	8 7	0
D0				
D1				
D2				
D3				
D4				
D5				
D6				
D7				

Para manipulación genérica de:

- Bytes
- Words
- Long Words

mediante el conjunto
de instrucciones de la CPU

La única diferencia entre ellos
es su nombre

EJEMPLOS

	D0	D1
	0 3 1 A 0 4 2 3	A 3 C 5 3 0 2 4
MOVE.B #8, D0	0 3 1 A 0 4 0 8	
MOVE.W #\$800, D0	0 3 1 A 0 8 0 0	
MOVE.L #\$1, D1		0 0 0 0 0 0 0 1
MOVE.W D1, D0	0 3 1 A 0 0 0 1	

El 68000 dispone de 8 registros de datos de 32 bits: **D0-D7**. La mayoría de las operaciones de manipulación de datos se efectúan con alguno de estos registros.

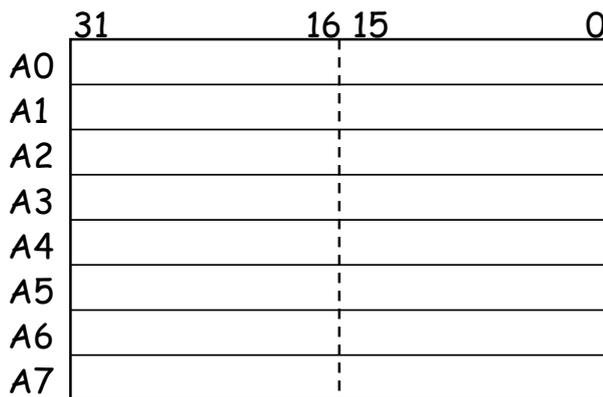
Los 8 registros son de uso general, en cuanto que cualquier operación permitida para uno de ellos, está igualmente permitida para cualquier otro registro de datos. Esto quiere decir que no hay un registro especial como el "Acumulador" de otras familias de procesadores.

Ya que los ordenadores con palabra de 32 o 16 bits no son apropiados para la manipulación de texto con caracteres que ocupan un byte, el 68000 permite operaciones con parte del contenido (8 o 16 bits) de los registros de 32 bits. Por esto, los registros están divididos en dos palabras y la palabra de menor peso está a su vez dividida en dos bytes. Se permiten operaciones con la doble palabra completa del registro, con la palabra de menor peso, o simplemente con el byte de menor peso.

Como se puede ver en los ejemplos de la parte inferior de la transparencia, las operaciones con dobles palabras, palabras o bytes, se denotan con los sufijos **.L**, **.W** y **.B** respectivamente, que se añaden a los códigos nemotécnicos de operación.

Recuérdese que cuando se opera con una sección de un registro (palabra o byte), la sección afectada es siempre la de menor peso del registro (bits 0 a 7 ó 0 a 15), y que el resto de los bits no se consideran y permanecen inalterados.

8 Registros de Direcciones (A0 .. A7)



A los registros de direcciones se les aplica un conjunto de instrucciones más reducido

El A7 se distingue del resto
PUNTERO DE PILA (SP)

Para contener las direcciones de los datos de la memoria principal a manipular

Con direcciones de 16 bits → ¡ Extensión de Signo !

	DO	A0
	8 3 C 2 4 C 5 F	F F F F F F F C
MOVE .W DO , A0		0 0 0 0 4 C 5 F
MOVE .W #\$FFDE , DO	8 3 C 2 F F D E	
MOVE .W DO , A0		F F F F F F D E

El 68000 cuenta con 8 registros de direcciones designados mediante **A0-A7**. Estos registros deben verse como “punteros”, ya que contienen las direcciones que indican la ubicación de los datos en memoria principal.

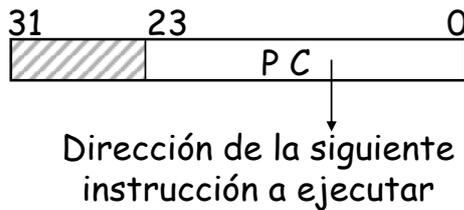
Todos los registros son de 32 bits, y aunque solamente se permiten las operaciones de palabra y de doble palabra, la operación sobre la palabra de menor peso afecta al registro completo. Esto significa que cuando se realiza una operación sobre la palabra de un registro de dirección, el bit de signo del operando se extiende por los bits 16 a 31 (por el registro completo). Esto es así porque el contenido de los registros de dirección son valores con signo en complemento a dos. Las direcciones negativas se utilizan como desplazamientos negativos relativos a una dirección dada.

Al igual que con los registros de datos, las operaciones que impliquen un registro de dirección pueden realizarse con cualquiera de ellos, es decir, todos tienen las mismas características. No obstante el registro A7 sí es un registro especial, pues además del papel que desempeñan el resto de los registros A0-A6, el A7 actúa como “Puntero de Pila” o *Stack Pointer* y se le puede designar mediante A7 ó SP indistintamente. Más adelante, al tratar los modos de direccionamiento, veremos la Pila y el papel que desempeña el registro A7 en su gestión.

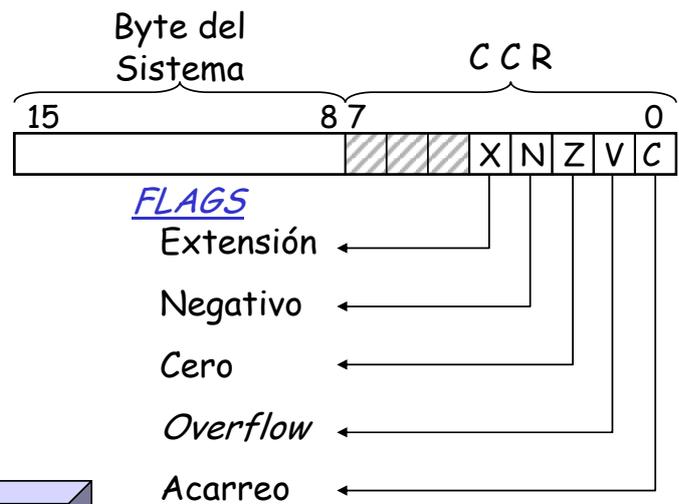
Aunque los registros de direcciones son de 32 bits, debe observarse que ya que el espacio de direccionamiento del 68000 es de 2^{24} bytes, solamente son significativos los 24 bits de menor peso de cada registro de direcciones.

Los registros de direcciones suelen utilizarse con un modo de direccionamiento de los operandos denominado “direccionamiento indirecto”. Más tarde, cuando tratemos este direccionamiento, veremos algún ejemplo de su utilidad.

☞ Contador de Programa (PC)



☞ Registro de Estado (SR)



El CCR (*Condition Code Register*) sirve para evitar la ejecución puramente secuencial de los programas

El 68000 cuenta con dos registros de propósito especial:

- Contador de Programa (PC)
- Registro de Estado (SR)

El **Contador de Programa (PC)** es el encargado de indicar siempre la dirección de la siguiente instrucción a ejecutar. Es de 32 bits, aunque ya que solamente se dispone un espacio de direccionamiento de 2^{24} , únicamente son significativos los 24 bits de menor peso.

El **Registro de Estado (SR)** indica, con sus 16 bits, cierta información del estado actual del procesador. Esta información está repartida en dos campos lógicos:

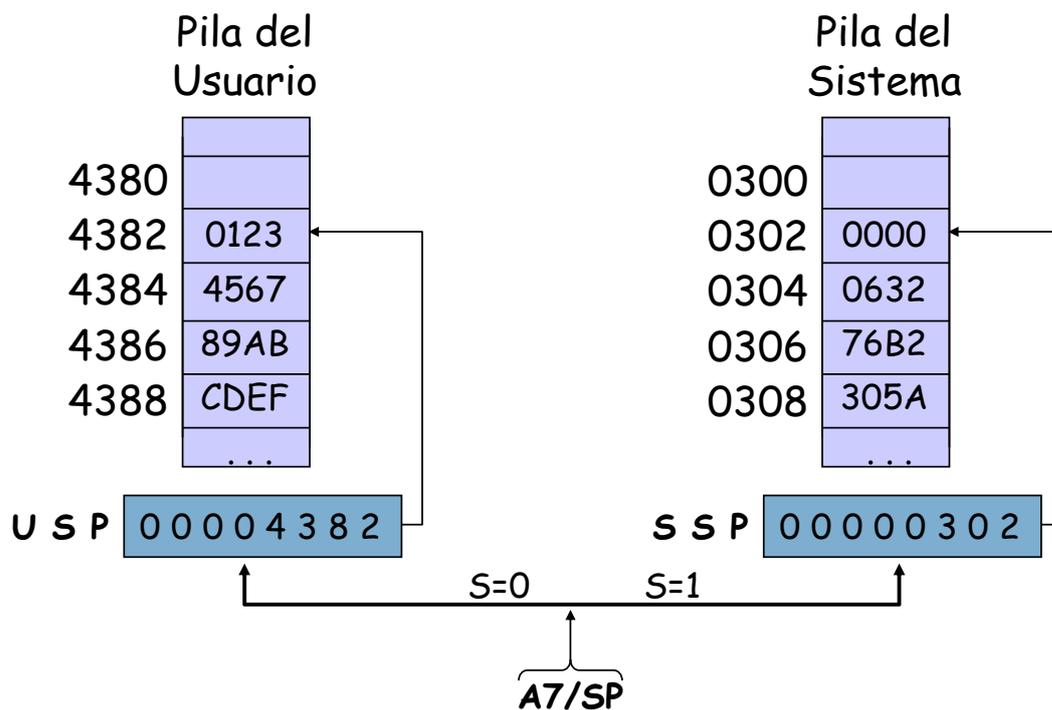
- El CCR (registro de códigos de condición)
- El Byte del Sistema

El CCR está en el byte menos significativo del registro, pero solamente utiliza los cinco bits de menor peso. Estos bits son los *flags* de estado del procesador, e indican alguna condición resultante de las últimas operaciones ejecutadas. La mayoría de las instrucciones realizan operaciones cuyo resultado afecta a los *flags* de condición. Por ejemplo, cuando el resultado de una suma es negativo, se activa un bit que así lo indica; cuando el resultado es cero, se activa otro, etc. Como vemos en la figura, los cinco *flags* son los siguientes:

- C: **Acarreo**. Se activa (se pone a 1) cuando se produce un acarreo en el bit de mayor peso del resultado de una operación lógica o aritmética.
- V: **Overflow**. Se activa cuando hay un desbordamiento en una operación aritmética en complemento a dos.
- Z: **Cero**. Se activa cuando el resultado de una operación es cero.
- N: **Negativo**. Este *flag* es una copia del bit más significativo (el de signo) del operando evaluado o de un resultado.
- X: **Extensión**. Este *flag* tiene el mismo significado que el de acarreo, pero solamente le afectan, básicamente, las instrucciones aritméticas con signo y las de desplazamiento y rotación.

Estos *flags* de condición sirven para poder alterar el flujo secuencial de ejecución de los programas basándose en el resultado de ciertas operaciones.

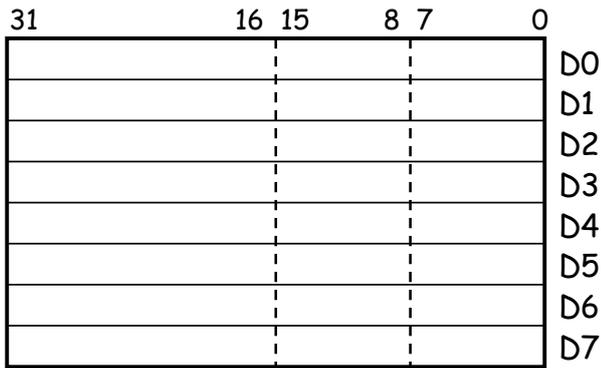
Existen dos pilas (y dos Punteros de Pila)



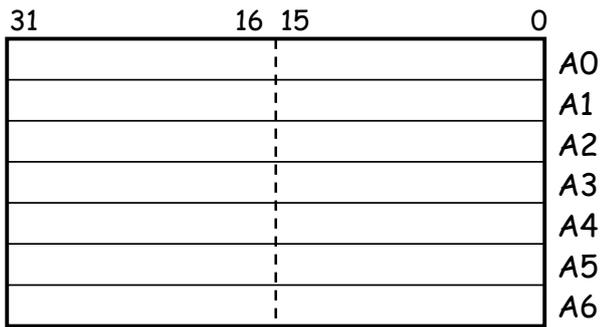
Como se puede ver en la diapositiva, **hay dos pilas de trabajo**, una del usuario y otra del sistema operativo o supervisor. Esto quiere decir que físicamente también hay dos registros A7 (punteros de pila), de tal manera que cuando el procesador está en modo usuario, el registro A7 ó SP se refiere físicamente al puntero de pila del usuario, mientras que si está en modo supervisor, el A7 ó SP se refiere al puntero de pila del sistema operativo.

El usuario no tiene forma de acceder al puntero de pila del sistema operativo, pero cuando se está en modo supervisor sí se puede acceder a ambas pilas. En modo supervisor se puede utilizar el USP (*User Stack Pointer*) para referirse al puntero de pila del usuario, y el SSP (*Supervisor Stack Pointer*) para referirse al puntero de pila del supervisor. Si se utiliza A7 o SP, se refiere al puntero de la pila correspondiente al modo actual de ejecución.

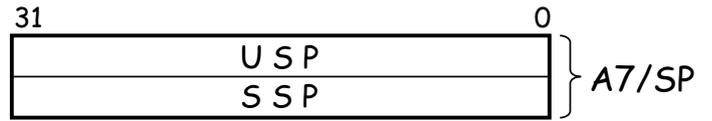
Para el usuario, el modo supervisor es transparente, es decir, es como si no existiera.



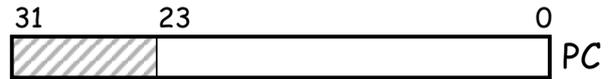
Registros de Datos



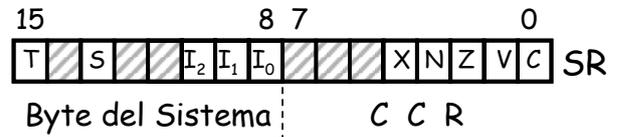
Registros de Direcciones



Punteros de Pila

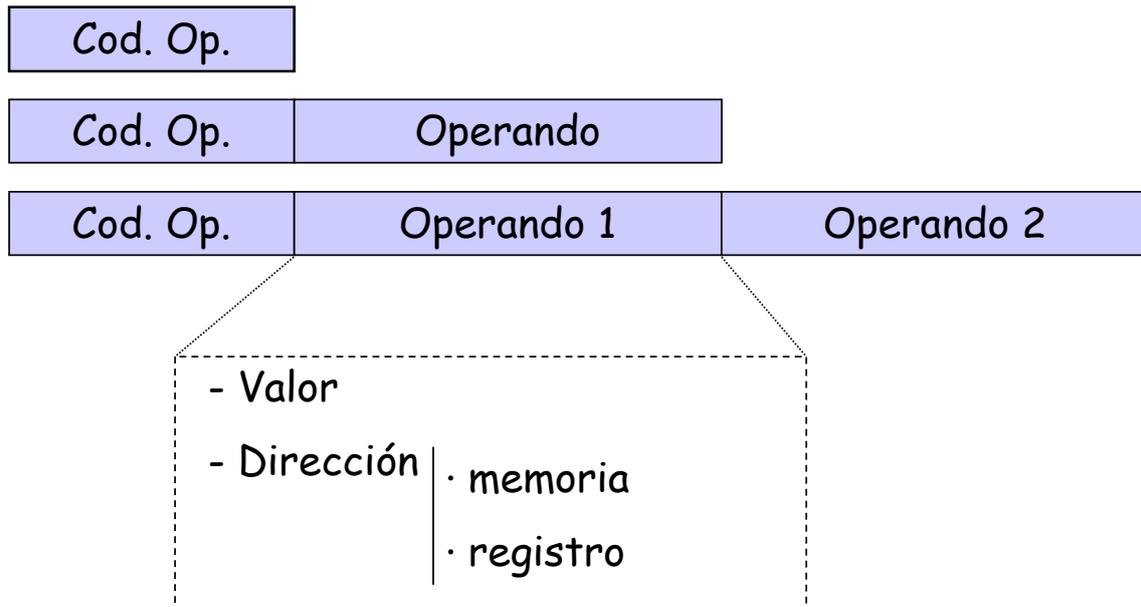


Contador de Programa



Registro de Estado

Aquí tenemos un resumen gráfico de los registros internos del 68000.



Una instrucción máquina es una secuencia de bits que indica una operación que debe realizarse y con qué operandos debe realizarse.

Los operandos son datos que están en alguno de los registros internos de la CPU, en la memoria principal o en algún controlador de entrada/salida.

Una instrucción siempre realiza alguna acción sobre algún o algunos operandos. Estos pueden estar indicados implícita o explícitamente en la instrucción máquina.

Un operando está implícito en una instrucción cuando el propio código de operación establece sobre qué actúa la operación. En este caso, tal operando es fijo, es decir, el programador no puede establecer que esa instrucción actúe con otro operando distinto.

Cuando una instrucción puede operar con diversos operandos posibles, a elegir por el programador, los operandos elegidos deben figurar de manera explícita (separados del código de operación) en la instrucción máquina.

Si una instrucción utiliza varios operandos, estos pueden ser una mezcla de operandos implícitos y explícitos.

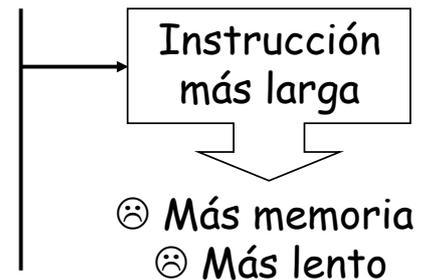
El formato de una instrucción máquina define la disposición y el reparto de sus bits entre los componentes de la instrucción, esto es, el código de operación y los operandos.

Una instrucción máquina debe incluir el código de operación y, si los tiene, uno o más operandos explícitos. A cada operando explícito se accede mediante uno de los distintos modos de direccionamiento que veremos en el siguiente apartado.

Como veremos más tarde (al comentar los distintos tipos de instrucciones que se suelen encontrar en los procesadores de propósito general) ya que hay instrucciones con uno, dos, tres o ningún operando explícito, es normal que los juegos de instrucciones estén compuestos por instrucciones con distintos formatos. En el formato de cada instrucción se suele indicar implícita o explícitamente el modo de direccionamiento de cada uno de sus operandos.

Longitud de Instrucción

- ✓ Muchos códigos de operación
- ✓ Muchos modos de direccionamiento
- ✓ Gran espacio de direcciones
- ✓ Long. instrucción múltiplo de palabra



Reparto de Bits

- ✓ ¿Cuántos códigos de operación?
- ✓ ¿Cuántos operandos?
- ✓ ¿Dónde están los operandos?
- ✓ ¿Cuántos registros hay?
- ✓ ¿Cuántos modos de direccionamiento?
- ✓ ¿Granularidad de las direcciones?

¡ COMPROMISO !

A la hora de diseñar un juego de instrucciones, se deben tener en cuenta ciertas **consideraciones** sobre dos cuestiones: la longitud de la instrucción y el reparto de los bits que la componen.

Longitud de la instrucción. Hay un claro compromiso entre el deseo de un rico y variado repertorio de instrucciones, y la necesidad de ahorrar espacio. Los programadores quieren muchos códigos de operación distintos (requiere muchos bits en el campo de código de operación) para disponer de operaciones que se ajusten lo más posible a sus necesidades, y así escribir programas más cortos. De igual manera, cuantos más modos de direccionamiento estén disponibles, más flexibilidad tendrán para manejar estructuras de datos complejas, como tablas o matrices. Por otro lado, puesto que cada vez se dispone de mayor cantidad de memoria, se requieren muchos bits para hacer referencia a un gran espacio de direccionamiento. Pero claro, una instrucción que ofrezca todo esto puede ocupar 32 bits, posiblemente, el doble que otra similar que no ofrezca tanta flexibilidad. Está claro que una instrucción larga ocupa más espacio en memoria que una corta, se tarda más tiempo en llevarla de memoria a la CPU, y se tarda más tiempo en decodificarla.

Se debe tener en cuenta que el ancho del bus de datos sea múltiplo de la longitud de un carácter, para evitar desaprovechar el espacio de almacenamiento. Y esta restricción afecta a la longitud de las instrucciones, que deben ocupar un número entero de palabras, o en una palabra debe caber un número entero de instrucciones. Por esto los tamaños de las instrucciones suelen ser múltiplos de 8. Un diseño con caracteres de 9 bits, instrucciones de 12 y palabras de 31 bits sería una catástrofe.

Reparto de los bits. Aquí el compromiso está en el reparto de los bits disponibles entre el número de códigos de operación y la versatilidad de los operandos. Esta versatilidad genera las siguientes preguntas: ¿cuántos operandos hay en la instrucción?, ¿de cuántos modos de direccionamiento se dispone?, ¿dónde están los operandos, en registros o en memoria?, ¿cuántos registros hay?.

El direccionamiento a memoria también plantea una pregunta: ¿cuál es la **granularidad de las direcciones**? Es decir, ¿a dónde apunta cada dirección, a un byte, a una palabra de 16 bits, a una palabra de 32 bits, ...? Si el grano es de 1 byte, para direccionar una memoria de 64 Kb se requieren 16 bits, mientras que si el grano fuera una palabra de 4 bytes, solo se necesitarían 14 bits, lo cual significa instrucciones más cortas, menor espacio para el programa, y un menor tiempo para alimentar la instrucción. Sin embargo, tiene la pega de que para obtener un carácter de memoria hay que acceder a la palabra completa, y luego en la CPU seleccionar la parte deseada, lo cual consume tiempo.

👉 ¿Dónde se Ubican los Operandos?

ADD #7, R4, \$FF1234

En la Instrucción

- ☺ Rápido y simple
- ☹ Sólo constantes
- ☹ No para resultados

En Memoria Principal

- ☹ Muchos bits

En Registros de la CPU

- ☺ Se requieren pocos bits
- ☺ Rápido acceso
- ☹ Pocos registros
- ☹ Hay que cargar los registros

El diseño cuidadoso de los códigos de operación es una parte importante del juego de instrucciones de una máquina. No obstante, la mayor parte de los bits de una instrucción se utilizan para especificar los operandos de la operación, por lo que también debe afrontarse con mucho cuidado el modo de direccionar los operandos de las instrucciones.

Vamos a considerar ahora dos factores sobre los operandos a tener en cuenta en el diseño del formato de las instrucciones:

- Dónde poner los operandos
- Cuántos operandos se indican en la instrucción

Los operandos pueden estar en tres sitios:

- En la propia instrucción
- En registros de la CPU
- En memoria principal

Con el operando en la misma instrucción, parece claro que el acceso a él es simple y rápido, no obstante estos operandos solamente pueden ser constantes, pues su valor se establece en tiempo de compilación; por esto mismo, tampoco se pueden utilizar como operandos de destino o de resultado. Así, si las variables están ubicadas en memoria principal, parece conveniente utilizar el campo de operando para indicar su **dirección en memoria principal**, sin embargo, para los procesadores actuales, es normal disponer de un bus de direcciones de 32 bits, lo que implica que el campo de operando en la instrucción requiere también 32 bits, y si consideramos una instrucción con tres operandos más el código de la operación, la longitud de la instrucción se dispara. Una alternativa puede ser **utilizar registros generales** para contener los operandos. Así, en una máquina con 8 registros se necesitan solamente 3 bits para indicar uno de ellos en el campo de operando. Esto tendría la mejora añadida de que el acceso a un registro es mucho más rápido que a memoria principal. Pero también tiene pegas. Una es que si se dispone de pocos registros generales y se ubican en ellos las variables, se pueden agotar enseguida. Por esto, las arquitecturas recientes tienden proporcionar un número generoso de registros. Otra pega es que para operar con operandos en registros, previamente hay que cargarlos desde memoria principal, lo que significa ejecutar instrucciones adicionales con direcciones de memoria (largas y costosas). Por esto, solamente merece la pena cargar los operandos en registros cuando se van a utilizar repetidamente (lo cual suele ser lo más habitual).

👉 ¿Cuántos Operandos se Indican en la Instrucción?

Todos

ADD \$FF1234, \$FFF123, \$FFFF12

↓ ☹ Requiere muchos bits

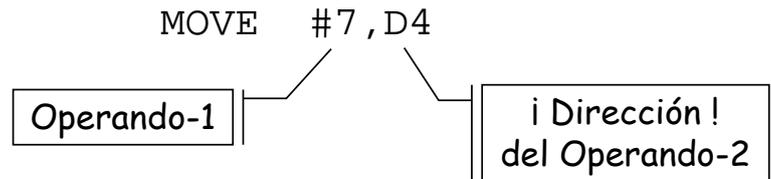
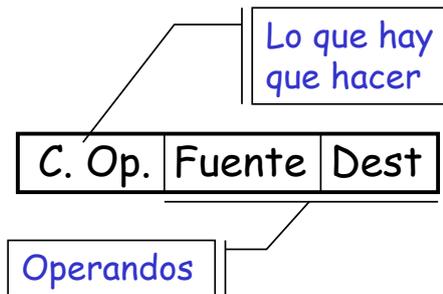
Reutilizar Alguno

ADD \$FF1234, \$FFF123

Omitir Alguno (Implícitos)

- En Acumulador: ADD \$FF1234
- En la Pila: PUSH R1

En cuanto al **número de operandos**, algunas instrucciones pueden requerir tres operandos, como la suma (dos sumandos y un resultado), lo cual puede resultar en un exceso de bits. Por ello, la mayoría de este tipo de instrucciones suele utilizar como resultado uno de los mismos operandos, ahorrando así espacio en la instrucción. Otra posibilidad consiste en no indicar los operandos explícitamente en la instrucción, utilizando directamente ciertos operandos de forma implícita. Así, algunas máquinas indican un único operando para realizar una suma, y lo que hacen es añadir el contenido de ese operando al de un registro especial llamado **acumulador**, y el resultado lo dejan también en el acumulador. El problema que presenta esta última técnica es que cuando hay muchos cálculos, hay que estar constantemente cargando y salvando el contenido del acumulador, lo cual requiere espacio de instrucciones y tiempo de ejecución. Otro modo de indicar operandos implícitos es con los direccionamientos **orientados a Pila** que veremos en las siguientes transparencias.



Con el Direccionamiento Inmediato (prefijo #) se expresa el valor del operando en la propia instrucción

¡ Pero solamente sirve para leer!
constantes

MOVE D4, #12
¡ Ilegal y sin sentido !

Para operar con **variables**, en la instrucción se debe especificar la **dirección** del operando, no su valor

El formato de una instrucción máquina consta de código de operación y de operandos (es una costumbre aceptada en informática el considerar al resultado o destino de la operación también como un operando).

Si el código de operación indica lo que hay que realizar, cada operando debería contener su valor. Así tenemos en el ejemplo una instrucción que copia el valor 7 al registro D4. En esta situación en la que **el valor del operando está en la propia instrucción** (el 7, en nuestro ejemplo), decimos que se tiene un **direccionamiento inmediato** al operando.

Pero si nos fijamos, en el operando de destino no está su valor, sino la dirección donde hay que dejar el operando fuente (D4 en nuestro ejemplo).

Se puede observar que con el direccionamiento inmediato el valor del operando es siempre el mismo (es una constante), puesto que se escribe en la propia instrucción en tiempo de compilación. Entonces ¿qué pasa si queremos operar con variables?

Está claro que **el direccionamiento inmediato no sirve para operar con variables**. En su lugar lo que se necesita es especificar la dirección de memoria donde está la variable, la cual tendrá valores distintos en distintas circunstancias.

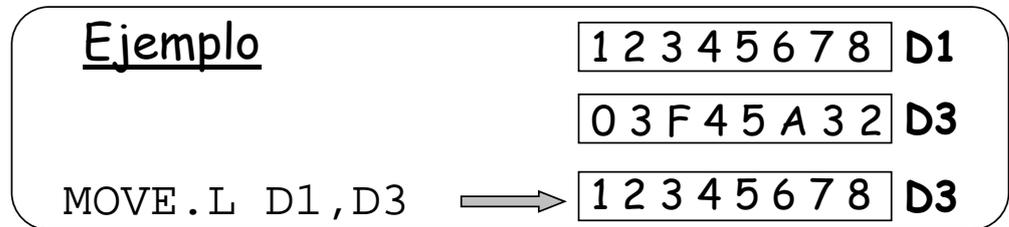
Hay diversos mecanismos o “modos de direccionamiento” para indicar la dirección de un operando en el 68000. Veámoslos a continuación.

El dato está en un registro

- de Datos
- de Direcciones

☞ Directo a Registro de Datos

Se expresa nombrando un registro de datos (D0-D7)



☞ Directo a Registro de Direcciones

Se expresa nombrando un registro de direcciones → (A0-A7, SP, USP, SSP)

Ej. MOVEA.L D3, A3 →

1	2	3	4	5	6	7	8	A3
---	---	---	---	---	---	---	---	----

Con **direccionamiento directo a registro**, el valor del operando se encuentra en uno de los **registros internos del 68000**. Por esto, en el campo de operando simplemente debe indicarse el nombre de tal registro.

Con este direccionamiento se puede acceder tanto a operandos fuente como de destino o de resultado, y como no implica accesos a memoria principal, resulta un modo rápido de direccionamiento.

Como registros de operandos puede utilizarse registros de datos (D0-D7) o de direcciones (A0-A7).

¿ Se puede acceder de forma explícita a los registros SR y CCR ?

→ MOVE.W D0,CCR Para poner el CCR
→ MOVE.W SR,D0 Para leer el SR

iNO se puede acceder al PC!

En algunas instrucciones también se permite acceder al registro de estado en su conjunto (SR), o solamente al CCR, aunque algunos de estos accesos solamente se permiten en modo supervisor.

No hay ninguna instrucción para acceder explícitamente al contador de programa, aunque implícitamente todas las instrucciones lo modifiquen para poder extraer la siguiente instrucción de memoria.

El operando de la instrucción indica la dirección de M.P. donde está su valor

La dirección se puede expresar

- En decimal 124
- En hexadecimal \$7C
- En binario %01111100

Dos Variantes

Absoluto
Corto

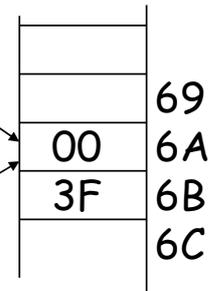
Absoluto
Largo

☆ EJEMPLOS

D1 1 2 3 4 0 0 3 F

MOVE.W D1, \$6A

MOVE.W D1, 106



MOVE.W D1, 6A ¡ ILEGAL !

A este modo de direccionamiento se le conoce como direccionamiento **absoluto, directo a memoria**, o simplemente **directo**. En este caso, **en el campo de operando de la instrucción se indica la dirección de memoria principal donde se encuentra el valor del operando o su destino**.

Este modo también es conocido como "absoluto" debido a que la dirección de la variable debe conocerse en tiempo de compilación y no varía nunca. Este direccionamiento solamente es válido para programas no reubicables, o bien para programas que siendo reubicables utilizan este direccionamiento solamente para acceder a dispositivos periféricos cuyas direcciones (siempre fijas) están *mapeadas* o representadas en el espacio de direccionamiento de la memoria principal.

Como se puede ver en los ejemplos, para denotar un operando en ensamblador, simplemente hay que poner una dirección (prefijado por \$ para base hexadecimal o % para base binaria) o la etiqueta de la variable.

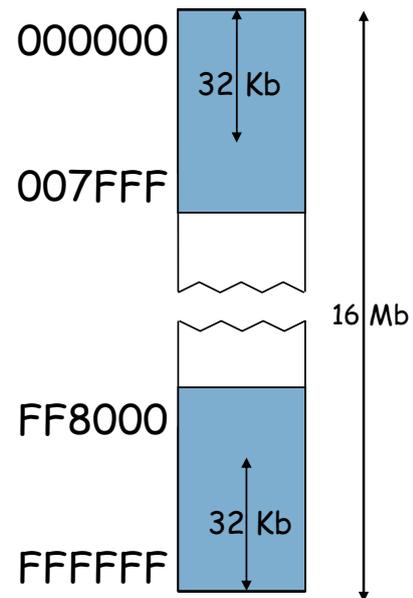
Este direccionamiento tiene dos variantes, dependiendo del número de bits que se requieren en el campo de operando de la instrucción para indicar la dirección. Veámoslos en la siguiente transparencia.

Absoluto Largo

Se utilizan 32 bits de la instrucción (24 efectivos) para expresar la dirección completa del operando en M. P.

Absoluto Corto

- ✓ La dirección expresada como operando ocupa 16 bits (se aplica extensión de signo)
- ✓ Genera instrucciones más cortas
- ✓ Sólo permite direccionar los 32 Kb más bajos y los 32 Kb más altos de la memoria principal



En el **direccionamiento absoluto largo**, el campo de operando consta de 32 bits en los que se indica la dirección absoluta y completa del valor del operando. Con los 32 bits se permite especificar cualquier área de memoria de todo el espacio de direccionamiento del 68000.

Con el **direccionamiento absoluto corto**, el campo de operando está formado por solo 16 bits. A partir de esta palabra se forma una dirección de 24 bits con extensión de signo. Con 16 bits con signo solamente puede accederse a dos áreas de la memoria:

\$000000 - \$007FFF (los 32 primeros Kbytes)

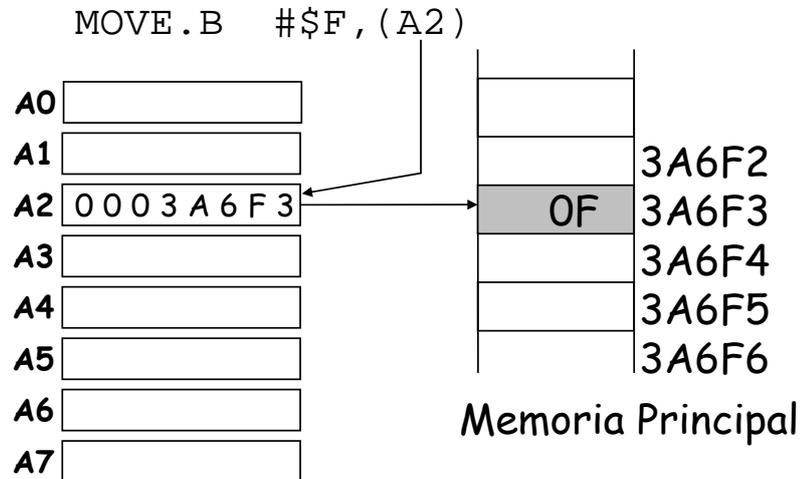
\$FF8000 - \$FFFFFF (los últimos 32 Kbytes)

A pesar de existir dos variantes del direccionamiento absoluto, el programador de ensamblador no tiene que preocuparse de esto, simplemente debe indicar el nombre de la variable o la dirección, pues el ensamblador selecciona automáticamente el modo más conveniente en cada caso.

El campo de operando indica un registro de direcciones cuyo contenido es la dirección del dato en M.P.

Se expresa nombrando un registro de direcciones entre paréntesis (**An**)

☆ EJEMPLO



Este direccionamiento es el Indirecto Puro
Existen otros modos derivados de éste

Direccionamiento indirecto por registro significa que **la dirección del operando se encuentra en un registro**. En el 68000, son los registros de direcciones los que se ocupan de esto (A0-A7).

Obsérvese que si en el direccionamiento absoluto el operando era una variable, pero su dirección era fija y constante por encontrarse en la propia instrucción, en el direccionamiento indirecto también es variable la dirección, pues ésta se encuentra en un registro de direcciones.

Como se muestra en los ejemplos, la indirección se denota encerrando entre paréntesis el registro que contiene la dirección de la variable. **An** hace referencia al contenido de An, mientras que **(An)** hace referencia al contenido de la dirección de memoria apuntada por An.

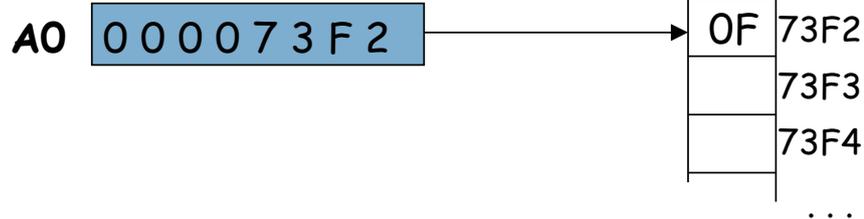
Podemos ir adelantando que los registros de direcciones suelen cargarse con instrucciones como MOVEA (*Move Address*), o LEA (*Load Effective Address*).

Esto que hemos comentado es el “direccionamiento indirecto por registro simple”. Pasemos a ver algunas variantes.

Un Ejemplo de Uso

```
MOVEA    #$73F2, A0
```

```
MOVE.B   #$F, (A0)
```



¿ Por qué no poner ?
 MOVE.B #\$F, \$73F2

Aquí mostramos un ejemplo de la utilidad del direccionamiento indirecto a través de los registros de direcciones.

Para escribir un valor en una dirección de memoria, se puede realizar cargando previamente un registro de direcciones con la dirección de memoria a escribir, y después escribir o copiar el dato mediante direccionamiento indirecto a través de dicho registro. Como se indica en la transparencia, parece más sencillo, y de hecho lo es, escribir el valor directamente mediante una operación de escritura en memoria.

En este caso, efectivamente, no tiene sentido utilizar un registro de direcciones, pero en la siguiente diapositiva tenemos un ejemplo donde aparece su utilidad.

¿Cómo pondríamos a cero todos los bytes
de las direcciones [73F0 .. 74F0] ?

Una Solución	Otra Mejor
MOVE.B #0,\$73F0	MOVEA.L #\$73F0,A0
MOVE.B #0,\$73F1	LOOP MOVE.B #0,(A0)
MOVE.B #0,\$73F2	ADDA.L #1,A0
. . .	CMPA #\$74F1,A0
. . .	BNE LOOP
MOVE.B #0,\$74F0	...

Para escribir un valor en una posición de memoria nos podemos servir de una instrucción de movimiento de un dato inmediato (contenido en la propia instrucción) hacia una dirección absoluta en memoria. Según esto, si queremos copiar tal valor a 256 posiciones consecutivas en memoria (rellenar un bloque de memoria) necesitaríamos otras tantas instrucciones.

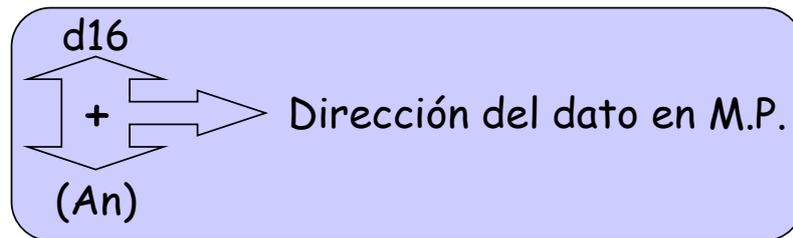
Una mejor solución puede venir de la mano de los registros de direcciones y del direccionamiento indirecto, pues como vemos en la "mejor solución", este relleno del bloque de memoria se consigue indicando en un registro la dirección de comienzo del bloque de memoria y, después, con ayuda de un bucle en el que se va incrementando este registro de dirección, se va escribiendo en direcciones sucesivas de memoria principal.

Indirecto con Desplazamiento

- ✓ Se expresa indicando un desplazamiento a la izquierda del registro de indirección:

d16 (An)

- ✓ El desplazamiento es un número de 16 bits con signo (-32768 . . 32767)



Como se puede ver en la transparencia, el **direccionamiento indirecto con desplazamiento** es muy similar al indirecto puro. En este caso, la dirección efectiva se obtiene no solo con el contenido del registro de indirección, sino añadiéndole además un número de 16 bits con signo (el desplazamiento) contenido en la propia instrucción (como una constante). Esto significa que a partir de una dirección cualquiera de memoria contenida en el registro de indirección, el desplazamiento puede modificar tal dirección en ± 32 Kbytes.

Obsérvese que para sumar un número de 16 bits con signo a otro de 32 bits, también con signo, al número de 16 bits se le aplica previamente una extensión del signo hasta los 32 bits.

...Indirecto con Desplazamiento

Muy útil para construir programas reubicables y para acceder a los campos de un registro

☆ EJEMPLO

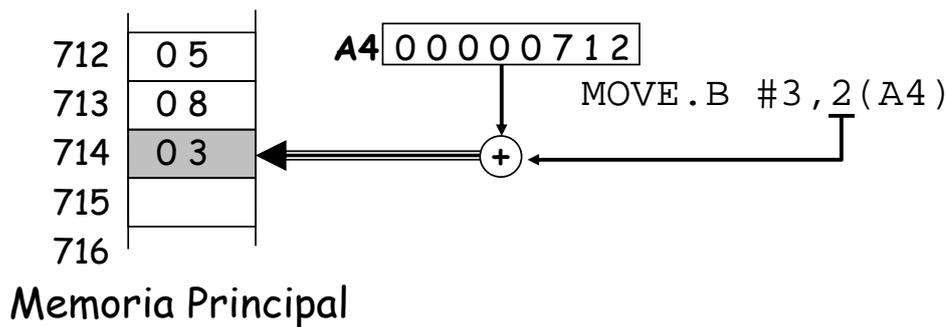
VAR Hora: RECORD

Horas,Min,Seg: BYTE

END;

5h 8' 3"

0 (A4)	→ Horas
1 (A4)	→ Min
2 (A4)	→ Seg



El direccionamiento indirecto con desplazamiento **es útil para la escritura de programas reubicables**, de tal manera que el registro se carga, en ejecución, con la dirección del comienzo de los datos, y los accesos a los operandos se calculan sumando a la dirección de comienzo de los datos (que es variable) el desplazamiento relativo (que es constante y establecido en tiempo de compilación).

Otra aplicación (también reubicable) consiste en cargar en el registro la dirección de una estructura de datos (como un registro de Pascal) y con el desplazamiento hacer referencia a cada uno de sus campos.

Aquí mostramos un ejemplo con un registro que contiene la hora (hora, minutos y segundos) y en el que cada uno de sus campos ocupa un byte. Con la dirección del comienzo del registro (712) cargada en el registro A4, se accede a cada uno de sus campos con los desplazamientos 0, 1 y 2.

Indirecto indexado

- ✓ Se expresa como: $d8 (A_n, X_i.z)$
- ✓ Siendo: $X_i = A0 \dots A7, D0 \dots D7$
 $Z = L \text{ ó } W \text{ (con extensión de signo)}$
- ✓ La dirección del dato en memoria se obtiene sumando:

- | | |
|--|---|
| | <ul style="list-style-type: none"> - El contenido del registro de dirección (A_n) - El desplazamiento de 8 bits (-128 .. 127) - El contenido del registro de índice |
|--|---|

El siguiente paso en el direccionamiento indirecto nos lleva al **direccionamiento indirecto indexado**. Es similar al indirecto con desplazamiento, pero en el caso indexado se dispone de un registro más (registro de índice X_i) cuyo contenido también debe sumarse en el cálculo de la dirección efectiva. Obsérvese que el registro de índice puede ser uno cualquiera de datos o direcciones, y que mediante el sufijo “.W” o “.L” puede considerarse la totalidad del registro o solamente la palabra de menor peso.

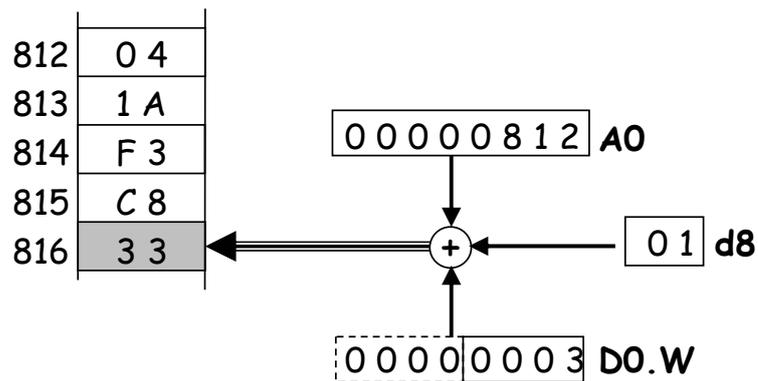
Como se observa en la transparencia, cuando el direccionamiento indirecto utiliza registro de índice, el desplazamiento es un número, con signo, de solo 8 bits, lo que quiere decir que permite una modificación constante en un rango de -128 a +127.

...Indirecto Indexado★ EJEMPLO

Muy útil para acceder a un elemento de un vector dentro de un registro

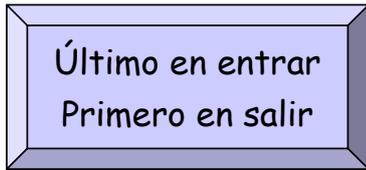
```
VAR Lista:RECORD
    Num_Elementos: BYTE
    Elementos: array [0..3]of BYTE
END;
```

```
MOVE.B #$33,1(A0,D0.W); Lista.Elem[3] := $33
```

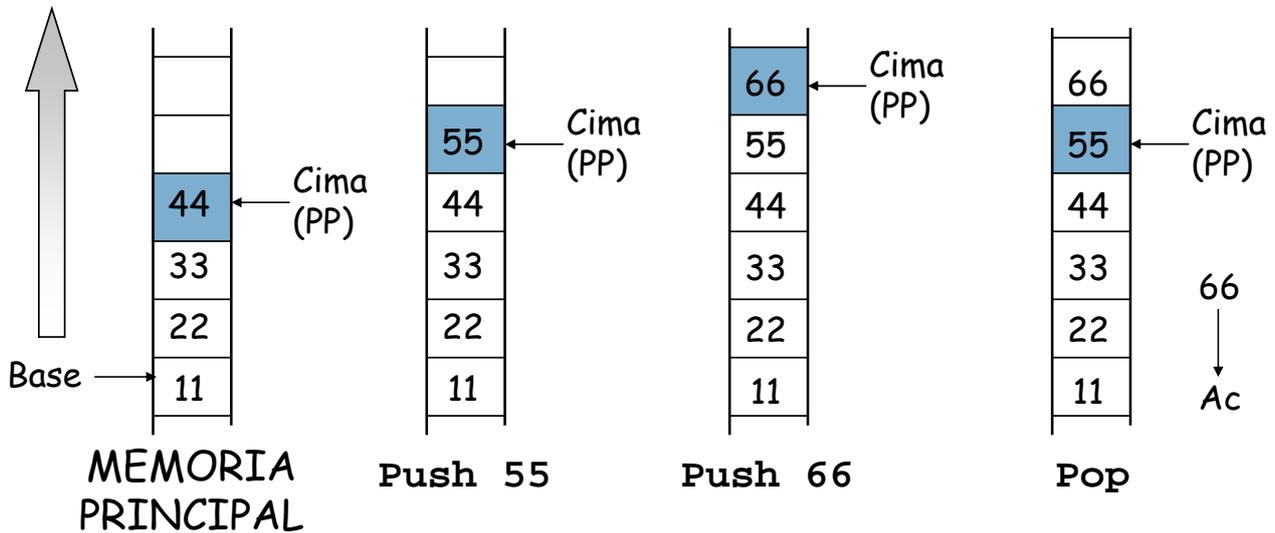


La utilidad de este direccionamiento está en la gestión de tablas de dos dimensiones, de vectores donde cada elemento de la tabla es un registro o estructura de datos, o para acceder a los elementos de un vector que es un campo de un registro.

Por ejemplo, si tenemos un vector cuyos elementos son estructuras de datos (registros), para acceder a los campos de un registro concreto, en el registro de indirección A_n se cargaría la dirección de comienzo del vector y en el registro de índice, el desplazamiento hasta un registro concreto (un elemento del vector). Sumando el desplazamiento de 8 bits se obtendría la dirección completa del campo deseado.



Direccionamiento | - Implícito (PP)
 - Explícito



Direccionamiento a Pila

La Pila es una estructura de datos que se mantiene en memoria principal. Básicamente es un vector de celdas de datos de direcciones secuenciales, en los que se meten y sacan datos con política LIFO (*Last In, First Out*), es decir, el último en entrar es el primero en salir.

Esta estructura es muy útil como ayuda en la ejecución de los programas, pues se utiliza para guardar en ella variables temporales, parámetros, direcciones de retorno en la llamada a procedimientos, etc.

El primer elemento que se mete en la pila se dice que esta en el fondo de la pila, y el último que se ha metido está en la **cima de la pila**. El interés de la cima de la pila es que siempre está en ella el primer dato disponible de la pila. Las operaciones con la pila son **meter** y **sacar** datos de ella, y operan utilizando un registro denominado **puntero de pila** que siempre contiene la dirección de la cima de la pila.

En algunos procesadores, como el 68000, el registro de puntero de pila es simplemente uno de los registros generales que se utilizan para direcciones, y se accede a la pila mediante instrucciones de movimiento de datos y el ya comentado direccionamiento indirecto por registro. Los procesadores de Intel disponen de un registro específico: el **Puntero de Pila**, que siempre está apuntando a la cima de la pila, y se ofrecen instrucciones específicas para **meter** y **sacar** datos de la pila (`PUSH` y `POP`) en las que ni siquiera se indica explícitamente el registro de indirección, pues está implícito por el código de operación.

Tanto en Motorola como en Intel, la Pila es una estructura de datos que, en contra de la intuición, crece de las direcciones altas hacia las bajas, es decir, que la cima de la Pila siempre está en una dirección más baja que la base de la Pila (salvo cuando la Pila está vacía, claro).

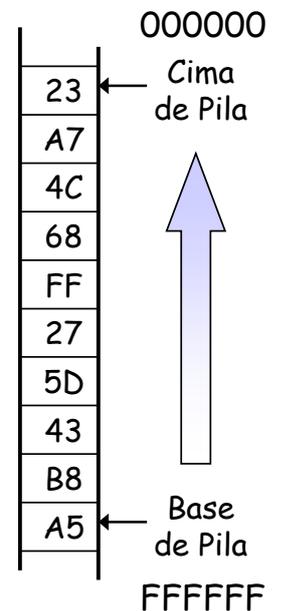
- ✓ Estructura de datos de tipo L.I.F.O. en memoria principal
- ✓ Con operaciones básicas para meter y sacar datos

Muy útil para

- Almacenamiento de variables temporales
- Evaluación de expresiones aritméticas
- Llamadas a subprogramas

En el 68000, la pila crece de direcciones altas hacia direcciones bajas

Siempre en direcciones PARES



La Pila es una estructura de datos que se mantiene en memoria principal. Básicamente es un vector de celdas de datos de direcciones secuenciales, en los que se meten y sacan datos con política LIFO (*Last In, First Out*), es decir, el último en entrar es el primero en salir.

Esta estructura es muy útil como ayuda en la ejecución de los programas, pues se utiliza para guardar en ella variables temporales, parámetros, direcciones de retorno en la llamada a procedimientos, etc.

El primer elemento que se mete en la pila se dice que está en el fondo de la pila, y el último que se ha metido está en la **cima de la pila**. El interés de la cima de la pila es que siempre está en ella el primer dato disponible de la pila. Las operaciones con la pila son **meter** y **sacar** datos de ella, y operan utilizando un registro denominado **puntero de pila** que siempre contiene la dirección de la cima de la pila.

En Motorola, la Pila es una estructura de datos que, quizás en contra de la intuición, crece de las direcciones altas hacia las bajas, es decir, que la cima de la Pila siempre está en una dirección más baja que la base de la Pila (salvo cuando la Pila está vacía, claro).

Indirecto con Pre-Decremento

- ✓ Se expresa como: $-(An)$
- ✓ El dato se obtiene como en el indirecto puro
- ✓ **Antes** de obtener el contenido de An ,
 An se **decrementa** en

1 → Si el dato es BYTE

2 → Si el dato es WORD

4 → Si el dato es LONG WORD

Muy útil para
meter un dato en la Pila

`MOVE.W #$305A, -(A7)`

Las dos últimas variantes del direccionamiento indirecto son las que permiten modificar el contenido del registro de indirección antes o después del cálculo de la dirección efectiva.

El **direccionamiento indirecto con pre-decremento** le resta al registro de indirección el tamaño del dato al que se accede, de tal manera que si la operación lleva el sufijo “.B”, se resta 1; si el sufijo es “.W”, se le resta 2 al registro, y si el sufijo es “.L”, se le resta 4. Una vez realizada la resta, se toma el contenido del registro como la dirección del operando. Esto quiere decir que si antes de la ejecución de esta instrucción, el registro de indirección An apuntaba a una cierta dirección de memoria, después de comenzar su ejecución, y justo antes de acceder al operando, el registro apunta a una dirección de memoria 1, 2 o 4 bytes más baja.

El registro A7 (o puntero de pila) supone una excepción, pues cuando se utiliza como registro de indirección con predecremento o postincremento, se incrementa o decrementa en 2 tanto para un operando de tipo *word* como para un byte, y en 4 para las dobles palabras.

Indirecto con Post-Incremento

- ✓ Se expresa como: $(An)+$
- ✓ El dato se obtiene como en el indirecto puro
- ✓ **Antes** de obtener el contenido de An ,
 An se incrementa en

1 → Si el dato es BYTE

2 → Si el dato es WORD

4 → Si el dato es LONG WORD

Muy útil para
sacar un dato en la Pila

MOVE.L (A7)+,D0

El **direccionamiento indirecto con post-incremento** se comporta de manera análoga al caso anterior, con la diferencia de que ahora, en primer lugar se toma el contenido del registro de indirección como la dirección del operando, y a continuación se le añade a dicho registro un valor igual al tamaño del operando.

Aunque pueda parecer un poco rebuscado, este direccionamiento es fundamental para la gestión de estructuras de datos de tipo "pila", muy comunes entre los procesadores actuales. Pasemos a la siguiente transparencia para comentar brevemente la utilidad de la pila y cómo se gestiona fácilmente con los dos modos de direccionamiento que acabamos de ver.

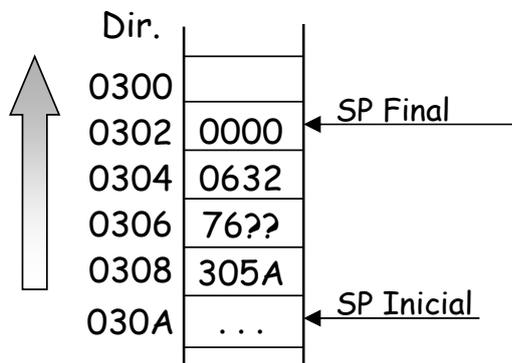
Estos dos direccionamientos también resultan útiles para tratar con vectores de datos cuyos elementos están almacenados consecutivamente en memoria. Por ejemplo, para sumar los n elementos (de una palabra de tamaño) de una tabla cuya dirección de comienzo está contenida en $A0$, basta con ejecutar n veces la siguiente instrucción:

`ADD.W (A0)+,D0`

👉 El registro SP (A7) apunta siempre a la cima de la Pila

①

Meter Datos



```
MOVE.W #$305A, -(SP)
```

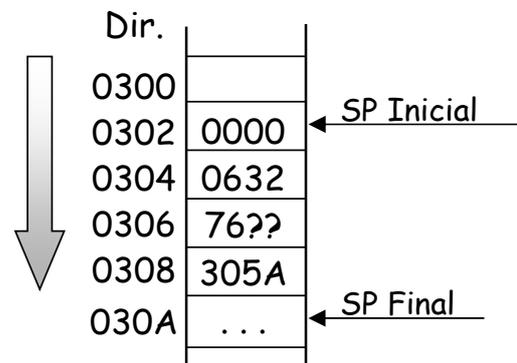
```
MOVE.B #$76, -(A7)
```

```
MOVE.L #$632, -(SP)
```

```
SP 00000302
```

②

Sacar Datos



```
SP 00000302
```

```
MOVE.L (A7)+, D0
```

```
MOVE.B (SP)+, D0
```

```
MOVE.W (A7)+, D0
```

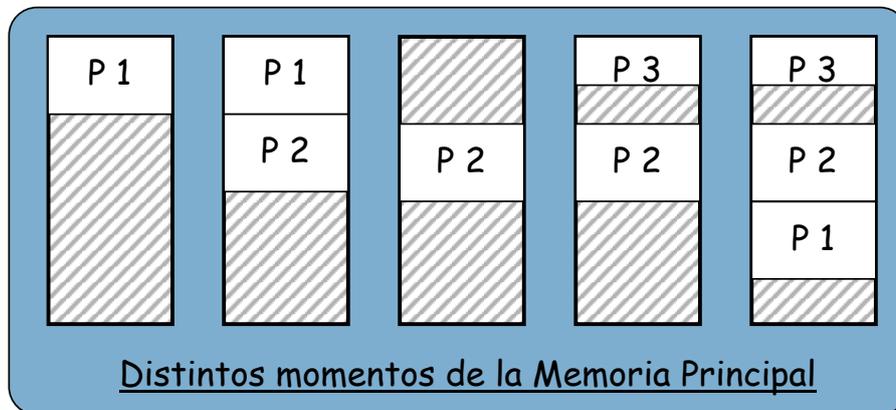
En el 68000, el puntero de pila (SP) puede ser cualquier registro de direcciones, pero algunas instrucciones que acceden implícitamente a la pila, como las de llamada y retorno de subrutina utilizan el registro A7, por lo que éste es el que se utiliza normalmente.

Así, el acceso a la pila se realiza mediante instrucciones de movimiento de datos y el ya comentado direccionamiento indirecto por registro. En la pila del 68000 los elementos de la pila ocupan siempre direcciones pares. Cuando se mete un byte en la pila, se deja inutilizado el byte adyacente para mantener las direcciones pares.

Sabiendo ya cómo es la pila del 68000 y sus operaciones, resulta fácil ver que el direccionamiento indirecto con pre-decremento es el apropiado para meter elementos en la pila, mientras que el indirecto con post-incremento se utiliza para sacar datos de ella.

Obsérvese el efecto de las operaciones de meter datos en la pila (①) partiendo de un Puntero de Pila con el valor 030A. Las operaciones de sacar datos (②) comienzan con un Puntero de Pila con el valor que dejó la última operación de meter datos, o sea, 0302.

- ✓ En un sistema multitarea puede existir más de un programa cargado en la Memoria Principal (ejecutándose)
- ✓ En tales sistemas, un mismo programa puede cargarse en zonas distintas de memoria cada vez que se ejecuta

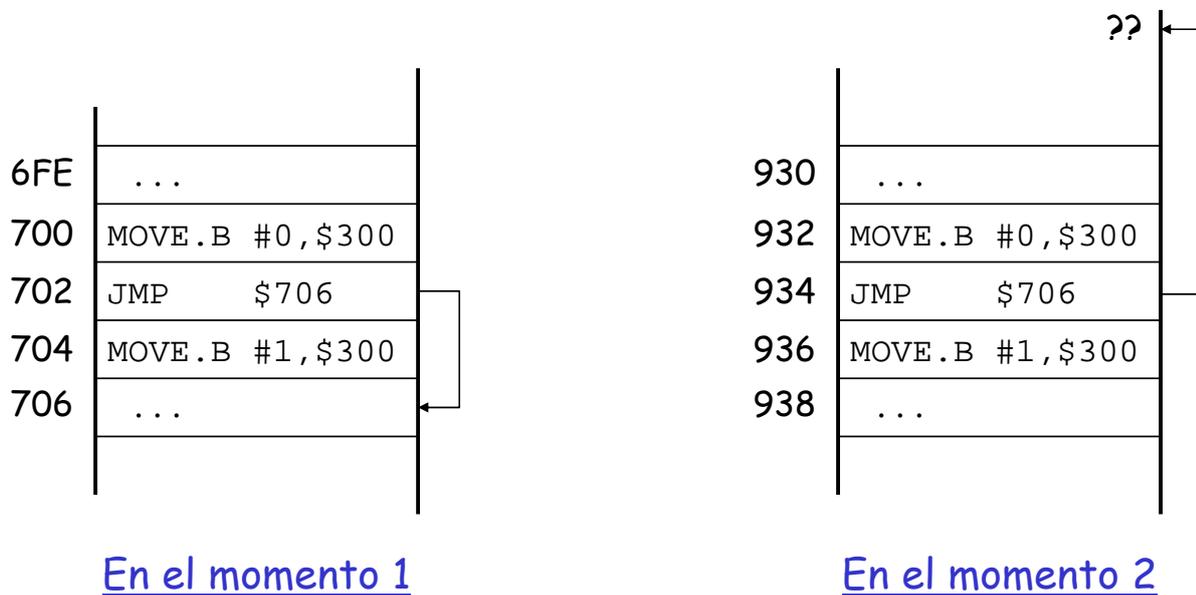


¡ESTO PUEDE SER UNA FUENTE DE PROBLEMAS!

En los sistemas multiusuario y en casi todos los sistemas operativos en general, suele ser normal la multitarea, según lo cual múltiples programas están cargados en memoria ejecutándose. Por esto, cuando se solicita la ejecución de un cierto programa no se sabe a priori el área de memoria que va a estar disponible para cargarlo.

Teniendo en cuenta esto, ya hemos visto que ciertos tipos de direccionamiento, como el absoluto o directo, no son apropiados para estos sistemas. También hemos visto que el direccionamiento indirecto soluciona bien el acceso a las variables.

☆ EJEMPLO de un posible problema



Como ya veremos, hay unas instrucciones de salto que alteran el flujo de ejecución secuencial de los programas, y en estas instrucciones se debe indicar la dirección de la siguiente instrucción a ejecutar. En ensamblador, esta dirección se suele indicar mediante una etiqueta que identifica la dirección de una instrucción.

Pues bien, la dirección de una instrucción a la que se desee saltar desde algún punto del programa también es variable, es decir, depende de la zona de memoria en la que se cargue el programa.

Como vemos en la diapositiva, en el *Momento 1* es posible que un programa esté cargado en una dirección de memoria tal que una instrucción de salto a la dirección \$706 se ejecute tal y como desea el programador. Sin embargo, en otro *Momento 2*, si este mismo programa se carga en otra dirección distinta, como se puede apreciar, la instrucción de bifurcación a la dirección \$706, no va a saltar a la instrucción deseada (la de la dirección \$938), sino a la instrucción de la dirección \$706; dirección que quizás cae incluso fuera del espacio del programa.

Como vemos, los saltos a direcciones absolutas suponen una pega para que un programa sea reubicable, es decir, que se pueda ejecutar en cualquier dirección de memoria.

Veamos a continuación cómo se soluciona esto en el 68000.

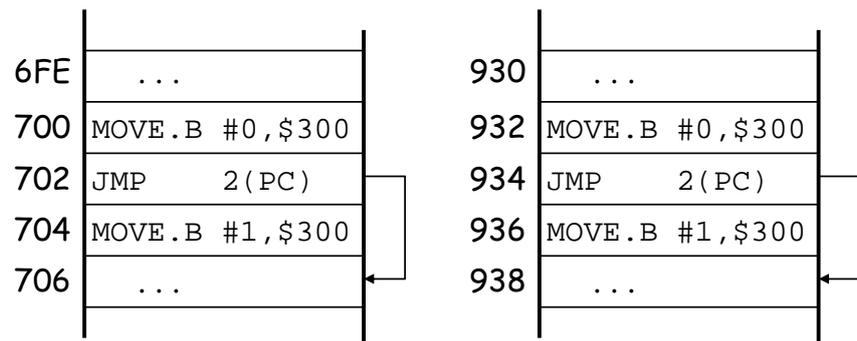
Relativo al PC con Desplazamiento

- ✓ Se expresa como: **d16 (PC)**
- ✓ Idéntico al **d16 (An)** excepto que sólo puede utilizarse para operandos fuente (no alterables)

```
MOVE.B 30(PC), $700
```

```
MOVE.B $700, 30(PC) ¡ ILEGAL !
```

- ✓ Permite escribir programas reubicables



Aunque el problema de la dirección de salto también se podría solucionar con el direccionamiento indirecto por registro que hemos visto anteriormente, en el 68000 se suele utilizar una variante en la que el registro de indirección es el Contador de Programa (PC), en lugar de uno de los registros generales de direcciones.

Este **direccionamiento relativo al Contador de Programa** tiene dos variantes:

- Relativo al PC con desplazamiento
- Relativo al PC e indexado

El formato del **direccionamiento relativo al PC con desplazamiento** es idéntico al indirecto con desplazamiento, con la ya mencionada salvedad de que el registro de indirección es el Contador de Programa, y que este direccionamiento solamente puede utilizarse para hacer referencia a operandos fuente, es decir, que no puede utilizarse para modificar un operando.

Relativo al PC e Indexado

- ✓ Se expresa como: $d8(PC, X_i.z)$
- ✓ Idéntico al $d8(An, X_i.z)$
- ✓ Con la misma restricción que el Relativo a PC con Desplazamiento

El **direccionamiento relativo al PC e indexado**, también es equivalente al indirecto indexado, y con la misma restricción que el relativo al PC con desplazamiento.

Modo	EA	REA	DEA	MEA	CEA	AEA	ADEA	AMEA	ACEA
Dn	X	X	X			X	X		
An	X	X				X			
(An)	X		X	X	X	X	X	X	X
(An)+	X		X	X		X	X	X	
-(An)	X		X	X		X	X	X	
d (An)	X		X	X	X	X	X	X	X
d (An, Xi)	X		X	X	X	X	X	X	X
Abs.W	X		X	X	X	X	X	X	X
Abs.L	X		X	X	X	X	X	X	X
d (PC)	X		X	X	X				
d (PC, Xi)	X		X	X	X				
Inmediato	X		X	X					

No todos los modos de direccionamiento que hemos visto pueden utilizarse en cualquier situación y con cualquier instrucción. Para cada una de las instrucciones que vamos a ver se indican en su formato, mediante una nomenclatura, los modos de direccionamiento permitidos.

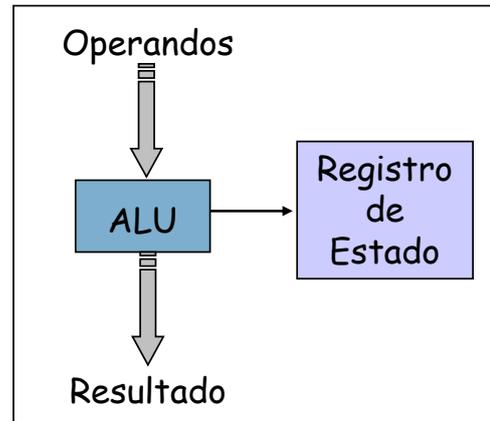
El cuadro adjunto muestra la nomenclatura utilizada para indicar los modos de direccionamiento permitidos en cada caso.

Como curiosidad, los nombres completos de los tipos de direccionamiento que se indican son los siguientes:

- EA *Effective Address*
- REA *Register Effective Address*
- DEA *Data Effective Address*
- MEA *Memory Effective Address*
- CEA *Control Effective Address*
- AEA *Alterable Effective Address*
- ADEA *Alterable Data Effective Address*
- AMEA *Alterable Memory Effective Address*
- ACEA *Alterable Control Effective Address*

- ✓ Transferencia de Datos
- ✓ Aritméticas
- ✓ Lógicas
- ✓ Transferencia de Control
- ✓ Control del Sistema

Ejecución de la Instrucción



Aunque el número de instrucciones que se ofrece varía de máquina a máquina, en todas ellas suelen encontrarse los mismos tipos de instrucciones:

- Transferencia de datos
- Aritméticas
- Lógicas
- Transferencia de Control
- Control del sistema

Muchas de estas instrucciones utilizan dos operandos, y dependiendo de la versatilidad u ortogonalidad de la máquina, pueden permitir que ambos operandos estén solamente en registros generales, uno en registro y el otro en memoria, o los dos en memoria. Obviamente, siempre hay un compromiso entre la máxima ortogonalidad (cualquier mezcla de modos de direccionamiento), con la correspondiente longitud de instrucción que ello conlleva.

En general, Intel y PowerPC suelen ofrecer instrucciones en las que no permiten todos los tipos de direccionamiento en una misma instrucción (por ejemplo, direccionamiento directo para ambos operandos), sobre todo en el caso del PowerPC, que es un procesador RISC. En cambio Motorola ofrece un juego de instrucciones más ortogonal, claro está, a cambio de tener instrucciones de hasta 10 bytes de longitud.

Otra cuestión que se debe tener en cuenta es que una gran parte de las instrucciones modifican o establecen los *flags* de estado del Registro de Estado, indicando si la última operación ha producido *overflow*, si un resultado es cero, positivo o negativo, etc.

Tres Formatos Principales

C. Op.			RTS
C. Op.	Dest		JMP \$400
C. Op.	Fuente	Dest	MOVE D4 , D6

El 68000 cuenta con diversos formatos de instrucción, pues aunque todas las instrucciones contienen un código de operación, el número de operandos es variable. Hay instrucciones con dos operandos, como las de movimiento; con un operando, como las de test o comprobación, y las que solamente contienen el código de operación. Al hablar del número de operandos nos referimos a operandos explícitamente indicados en la instrucción, porque muchas instrucciones también afectan a operandos de una manera implícita, como las de retorno de subrutina o de excepción, que toman automáticamente datos (operandos) de la pila.

En las instrucciones con dos operandos, debe quedar claro que, a efectos de nomenclatura, **el primer operando es el fuente y el segundo es el destino.**

☞ El tamaño de los operandos se indica añadiendo un sufijo al código de operación

- .B → Longitud de Byte (8 bits)
- .W → Longitud de Palabra (16 bits)
- .L → Longitud de Doble Palabra (32 bits)

☆ EJEMPLOS

	D5	D6
Contenido inicial →	FF A0 55 3A	00 CD 77 FE
MOVE .B D5, D6	FF A0 55 3A	00 CD 77 3A
MOVE .W D5, D6	FF A0 55 3A	00 CD 55 3A
MOVE .L D5, D6	FF A0 55 3A	FF A0 55 3A

Algunas instrucciones afectan a algunos *flags* de condición

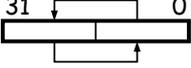
Las instrucciones pueden operar con datos de distintos tamaños, y para ello simplemente se debe acompañar un sufijo al código de operación. Como se ve en la transparencia, los sufijos “.B”, “.W” y “.L” denotan, respectivamente, operandos de tipo byte, palabra o doble palabra. Si no se indica ningún sufijo, se supone que los operandos son palabras.

Las instrucciones del 68000, de acuerdo a su cometido se pueden clasificar en los siguientes grupos:

- De movimiento o copia de datos
- Aritméticas
- De comparación y comprobación (o test)
- De manipulación de bits
- De desplazamiento y rotación
- Lógicas
- De control de programa

En [Clements 2.4] se detallan completamente todas las instrucciones del 68000, y se muestran ejemplos de su utilidad.

Debemos avanzar que la ejecución de muchas de las instrucciones que vamos a comentar afectan a los *flags* de condición del Registro de Estado (SR).

Código	Operandos	Operación
MOVE	<EA>, <ADEA>	fuente → destino
MOVEA	<EA>, An	
MOVEQ	#<d8>, Dn	
MOVEP	Dn, d(An) d(An), Dn	
MOVEM	<lista_reg>, <ACEA>+ <CEA>+, <lista_reg>	
LEA	<CEA>, An	dirección → An
PEA	<CEA>	dirección → -(SP)
EXG	Rn, Rm	Rn ↔ Rm
SWAP	Dn	

Ejemplos: MOVEM.L D0-D7/A0-A6, -(SP)
MOVEM.L (SP)+, D0-D7/A0-A6

A pesar del nombre genérico del grupo, la mayoría de las instrucciones de movimiento de datos simplemente copian información de un lugar a otro. Comentaremos algunos aspectos significativos de estas instrucciones.

MOVE copia información entre dos operandos, incluso permite acceder a los registros SR, CCR y USP, pero obsérvese que no sirve para copiar un dato a un registro de direcciones, para ello está la instrucción específica **MOVEA** (*move address*) que permite cualquier direccionamiento origen, pero solo admite un registro de direcciones como destino.

En cuanto a los registros especiales, se puede escribir el CCR, y se puede leer y escribir el SR, aunque la escritura en el SR solamente se permite en modo supervisor. El USP también se puede leer y escribir desde un registro de direcciones, pero solamente en modo supervisor.

MOVEQ (*move quick*) está prevista para copiar literales inmediatos de 8 bits con signo.

La instrucción **MOVEP** (*move peripheral*) está pensada para trabajar con los periféricos de la familia 6800, de 8 bits, por lo que la copia de operandos la realiza de 8 en 8 bits.

MOVEM (*move multiple*) le ofrece al programador la posibilidad de mover o copiar un grupo de registros a/desde memoria principal con una sola instrucción. Una lista de registros se define como Ai-Aj/Dp-Dq, lo cual indica todos los registros que van desde el Ai al Aj y del Dp al Dq.

La primera instrucción del ejemplo mete en la pila todos los registros generales excepto el A7, la segunda hace lo contrario, restaura el contenido de los registros guardados en la pila.

LEA (*load effective address*) carga en un registro de direcciones la dirección calculada en el primer operando. Esta instrucción solo admite operandos de 32 bits. Esta instrucción resulta útil para cargar registros de direcciones que se van a utilizar en direccionamientos indirectos. **PEA** (*push effective address*) es similar, mete en la pila la dirección calculada en el operando.

EXG intercambia el contenido de los dos registros indicados, mientras que **SWAP** intercambia las dos palabras de un registro de datos.

Código	Operandos	Operación
ADD	<EA>, Dn Dn, <AMEA>	fuerza + dest → dest
ADDA	<EA>, An	
ADDI	#<dato>, <A DEA>	
ADDQ	#<d3>, <AEA>	
ADDX	Dn, Dm -(An), -(Am)	fuerza + dest + X → dest
SUB	<EA>, Dn Dn, <AMEA>	dest - fuerza → dest
SUBA	<EA>, An	
SUBI	#<dato>, <A DEA>	
SUBQ	#<d3>, <AEA>	
SUBX	Dn, Dm -(An), -(Am)	dest - fuerza - X → dest

Las instrucciones aritméticas que ofrece el 68000 actúan sobre números enteros de 8, 16 y 32 bits, excepto la multiplicación, la división y las operaciones que tienen como destino a un registro de direcciones, que solamente trabajan con dobles palabras.

La instrucción **ADD** realiza la suma de dos operandos, donde uno de ellos debe ser un registro de datos. No se permite la suma de dos operandos en memoria.

Para añadirle algo a un registro de direcciones debe emplearse la instrucción **ADDA** (*add address*), que solamente admite operandos de tipo *word* o *long word*,

La suma de un literal inmediato se realiza con **ADDQ** o **ADDI**. **ADDQ** utiliza un campo de 3 bits en la instrucción para indicar una constante (de valor 1 a 8). En el caso de **ADDI**, el literal puede ser un byte, una palabra o una doble palabra. Obviamente, si la constante es pequeña (entre 1 y 8), interesa utilizar **ADDQ**, pues ocupa menos espacio y se ejecuta más rápidamente.

ADDX suma el contenido de los dos operandos, más el contenido del *flag* X del CCR. Esta instrucción puede servir para realizar operaciones con datos de 64 bits almacenados en dos registros.

El cometido y funcionamiento de las operaciones de sustracción es análogo al de las instrucciones de suma.

Código	Operandos	Operación		
MULS	<DEA>, Dn	fuelle x dest → dest 16 x 16 → 32		
MULU	<DEA>, Dn	fuelle x dest → dest		
DIVS	<DEA>, Dn	dest / fuente → dest Dn 32 / 16 → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>resto</td><td>cociente</td></tr></table>	resto	cociente
resto	cociente			
DIVU	<DEA>, Dn	dest / fuente → dest		
CLR	<ADEA>	0 → dest		
NEG	<ADEA>	0 - dest → dest		
NEGX	<ADEA>	0 - dest - X → dest		
EXT	Dn	Extiende signo en Dn		

Hay dos variantes para las operaciones de multiplicación y división, dependiendo de si se consideran los operandos con signo (MULS y DIVS) o sin signo (MULU y DIVU).

Las operaciones de multiplicación **MULS** y **MULU** (*multiplication signed* y *multiplication unsigned*) obtienen el producto de dos operandos de 16 bits, esto es, multiplica la palabra de menor peso del primer operando por la palabra de menor peso del segundo operando, y deja un resultado de 32 bits en el registro completo que figura como segundo operando.

Las instrucciones de división **DIVS** y **DIVU** (*division signed* y *division unsigned*) dividen la doble palabra del registro del segundo operando entre la palabra de menor peso del primer operando. El cociente de la división es un número de 16 bits que se deposita en la palabra de menor peso del registro del segundo operando, mientras que el resto se deja en la palabra de mayor peso.

La instrucción **CLR** (*clear*) pone un cero en el operando. Ya que entre sus modos de direccionamiento no están permitidos los registros de direcciones, para poner a cero un registro de direcciones puede hacerse con `SUBA.L An, An`.

NEG (*negate*) le resta a cero el operando y deja el resultado en el mismo operando. Simplemente realiza el complemento a dos del operando. Su variante **NEGX** (*negate with extension*) le resta a cero el operando y el contenido del *flag X* del CCR.

La instrucción **EXT** (*extension*) realiza una extensión de signo en un registro de datos. Si el operando es una palabra, extiende el signo del byte de menor peso a la palabra completa. Si el operando es una doble palabra, extiende el bit de signo de la palabra de menor peso al registro completo.

Código	Operandos	Operación
AND	<DEA>, Dn Dn, <AMEA>	} fuente \wedge dest \rightarrow dest
ANDI	#<dato>, <ADEA>	
OR	<DEA>, Dn Dn, <AMEA>	} fuente \vee dest \rightarrow dest
ORI	#<dato>, <ADEA>	
EOR	Dn, <ADEA>	} fuente \oplus dest \rightarrow dest
EORI	#<dato>, <ADEA>	
NOT	<ADEA>	$\overline{\text{dest}} \rightarrow \text{dest}$

El 68000 dispone de cuatro operaciones lógicas o “booleanas”: AND, OR, EOR (*exclusive OR*) y NOT. Todas ellas se pueden aplicar a operandos de tipo byte, palabra o doble palabra; incluso se permite modificar los registros CCR y SR (este último solamente en modo supervisor), aunque no se pueden aplicar sobre los registros de direcciones.

Las operaciones AND, OR y EOR suelen realizarse entre un operando y una “mascara de bits” que indica los bits del operando sobre los que se va a realizar la operación. Recordamos que la utilidad de la operación AND es la de “desenmascarar” o poner a cero selectivamente los bits del operando (los indicados con un 0 en la máscara); la instrucción OR realiza la función contraria, es decir, “enmascararlos” o ponerlos a uno (los indicados con un 1 en la máscara). La operación EOR afecta a los bits del operando indicados por los bits a 1 de la máscara, cambiando los ceros por unos y viceversa.

La instrucción NOT realiza el complemento a uno del operando completo.

Código	Operandos	Operación
ASL	<AMEA> Dn, Dm #<d3>, Dn	
ASR	igual que ASL	
LSL	igual que ASL	
LSR	igual que ASL	
ROL	igual que ASL	
ROR	igual que ASL	
ROXL	igual que ASL	
ROXR	igual que ASL	

En las operaciones de desplazamiento, todos los bits del operando (un registro de datos) se mueven uno o más lugares hacia la izquierda o la derecha.

Estas operaciones se pueden clasificar como lógicas, aritméticas y circulares o de rotación.

En las **operaciones de desplazamiento lógico**, entra un cero por un extremo, y el resto de los bits se desplazan hacia el extremo opuesto. El bit que sale por este extremo opuesto pasa al *flag* de acarreo (C) y al de extensión (X).

Las operaciones de **desplazamiento aritmético** sirven para realizar fácilmente multiplicaciones o divisiones por potencias de 2 (desplazamiento a la izquierda y a la derecha respectivamente). El desplazamiento aritmético a la izquierda se comporta igual que el desplazamiento lógico, con la única salvedad de que en la versión aritmética se puede activar el *flag* de overflow si cambia el signo del operando. El desplazamiento aritmético a la derecha difiere del lógico en que el bit que entra por la izquierda no es un cero, sino una copia del bit de mayor peso (el de más a la izquierda). Esta copia no significa otra cosa que una extensión del signo, de tal forma que aunque se va dividiendo por 2, se va manteniendo el signo en el resultado (el cociente).

En las **rotaciones**, al desplazar todos los bits, el bit que sale por un extremo entra por el opuesto, de tal manera que no se pierde ningún bit, solamente se altera el orden. Estas instrucciones de rotación pueden ser útiles para ir analizando los bits del operando, ya que el bit de mayor peso se copia al *flag* S del CCR, y los bits que salen por los extremos se copian a los *flags* de acarreo y de extensión.

Código	Operandos	Operación
BTST	#<d5>,Dn #<d3>, <MEA> Dn, Dm Dn, <MEA>	$\overline{\text{dest}[\text{bit}]} \rightarrow Z$
TAS	<ADEA>	1º comprueba dest, establece CCR 2º $1 \rightarrow \text{dest}[\text{bit } 7]$
BCLR	#<d5>,Dn #<d3>, <AMEA> Dn, Dm Dn, <MEA>	1º $\overline{\text{dest}[\text{bit}]} \rightarrow Z$ 2º $0 \rightarrow \text{dest}[\text{bit}]$
BSET	igual a BCLR	1º $\overline{\text{dest}[\text{bit}]} \rightarrow Z$ 2º $1 \rightarrow \text{dest}[\text{bit}]$
BCHG	igual a BCLR	1º $\overline{\text{dest}[\text{bit}]} \rightarrow Z$ 2º $\overline{\text{dest}[\text{bit}]} \rightarrow \text{dest}[\text{bit}]$

Las instrucciones de manipulación de bits actúan sobre un único bit del segundo operando, en lugar de hacerlo sobre el operando completo. El orden o peso del bit se indica en el primer operando. El operando sobre el que se actúa (el segundo) puede ser un byte, una palabra o una doble palabra.

En cada una de estas instrucciones, el complemento del bit seleccionado se copia al *flag* Z (cero) y a continuación se opera sobre el bit en cuestión (se deja como está, se pone a 0 ó se pone 1). El resto de los *flags* del CCR no resultan afectados por estas operaciones.

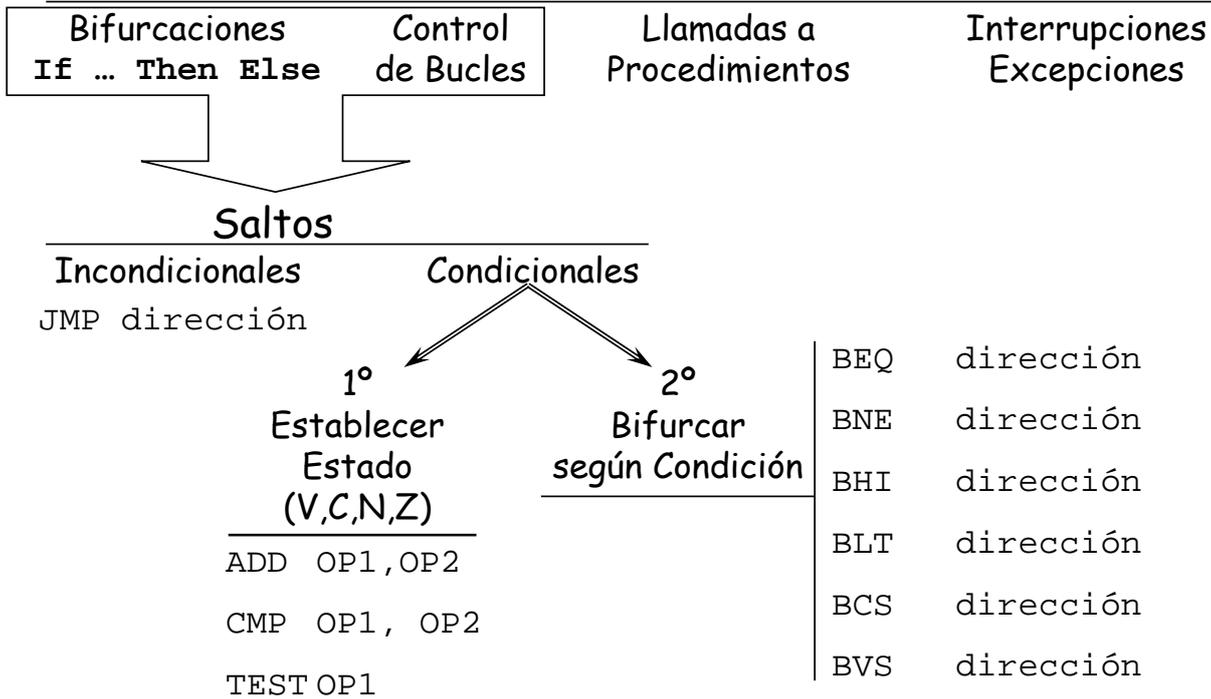
La instrucción **BTST** no afecta al contenido del operando, simplemente comprueba el valor del bit especificado y lo refleja en el *flag* Z del CCR (si el bit es 0, pone a 1 el *flag* Z).

Aunque la instrucción **TAS** no es estrictamente una instrucción de manipulación de bits, es fácil encuadrarla en este grupo. Esta instrucción opera sobre un byte y actúa en dos pasos. Primero comprueba si el valor del byte es cero o negativo y establece consecuentemente los *flags* Z y N. A continuación pone a 1 el bit más significativo del byte (el 7). El propósito de esta instrucción es facilitar la sincronización de procesos, y debe saberse que su ejecución completa se realiza sin perder la posesión del bus, por lo que no puede ser interrumpida por otro procesador (en un sistema multiprocesador).

BCLR y **BSET** comprueban primero el valor del bit en cuestión y lo reflejan en el *flag* Z; a continuación ponen el bit a 0 ó a 1 respectivamente.

Por último, **BCHG** primero también comprueba el valor del bit especificado para reflejarlo en el *flag* Z; a continuación cambia su valor (de 0 a 1, o de 1 a 0).

A veces hay que romper el flujo secuencial de las instrucciones



Transferencia de control

Ya sabemos que el flujo natural de ejecución de las instrucciones es el orden lineal: se ejecuta la instrucción de la siguiente dirección de memoria. No obstante, los programas tienen que modificar esa secuencialidad dependiendo de ciertas situaciones o de los datos de entrada. También se hace necesaria la ruptura del flujo secuencial de control cuando se realizan llamadas a procedimientos, que si bien mantienen una secuencialidad lógica, requieren un salto a una instrucción distinta de la siguiente en secuencia.

Por esto se hace necesaria la presencia de instrucciones que permiten variar el flujo de control, lo cual se realiza con instrucciones de salto o bifurcación.

Hay instrucciones de **salto incondicional**, que simplemente tienen un código de operación y un operando que indica la dirección de la siguiente instrucción a ejecutar. Pero en muchas ocasiones, los saltos se realizan solamente si se da una determinada circunstancia, es decir, necesitamos **saltos condicionales**.

Los saltos condicionales se realizan en función de los bits o *flags* de estado del registro de estado. Estos bits los establecen muchas operaciones de transferencia de datos, lógicas y aritméticas. Además se dispone también de **operaciones de comparación**, que toman dos operandos y establecen los bits de estado dependiendo de la comparación de los dos operandos. También hay **instrucciones de prueba o test**, que establecen los *flags* de estado considerando solamente el valor de un único operando (positivo, negativo, cero).

Algunos de los *flags* más comunes que suele tener el registro de estado son los que indican: *overflow*, acarreo, cero, negativo.

Como resumen, decir que los saltos condicionales están formados por pares de instrucciones, una que establece el estado, y otra que establece la condición y el salto; si el estado cumple la condición, se produce el salto. Las instrucciones más habituales para establecer el estado son las aritméticas, las de comparación y las de prueba o *test*.

Hay otros mecanismos para romper el flujo secuencial de ejecución, como son las corrutinas, las interrupciones y las excepciones (o interrupciones software). De éstas, en esta asignatura vamos a tratar solamente las interrupciones, pero lo haremos en el capítulo dedicado a los Sistemas de Entrada/Salida

Código	Operandos	Operación
JMP	<CEA>	dirección → PC
BRA	<etiqueta>	PC + despl → PC

En los programas, hay veces que se desea alterar el flujo de ejecución independientemente de los datos o la información que se tiene; para estos casos, el 68000 dispone de los **saltos incondicionales**. Se dispone de dos instrucciones de salto incondicional: `JMP` (*jump*) y `BRA` (*branch*). Mientras que `JMP` ofrece múltiples modos de direccionamiento para indicar cualquier dirección de memoria, `BRA` tiene como único operando un byte o una palabra que actúa como desplazamiento relativo al Contador de Programa, por lo que solamente permite saltos en un rango de ± 32 Kbytes sobre el PC. En ensamblador, basta con poner una etiqueta como operando y el ensamblador se encarga de calcular el desplazamiento.

Código	Operandos	Operación
<u>Comparación y Test</u>		
CMP	<EA>, Dn	
CMPA	<EA>, An	
CMPI	#<dato>, <ADEA>	
CMPM	(An)+, (An)+	
TST	<ADEA>	comprueba dest y pone CCR

Saltos Condicionales

Bcc	<etiqueta>	if cc then PC + d → PC
DBcc	Dn, <etiqueta>	if not cc then Dn := Dn -1; if Dn <> -1 then salta a <etiqueta> end if; end if;

Ejemplo:

```
CMP D1, D2
BEQ SALIR
```

Otras veces, la decisión de cuál es la siguiente instrucción a ejecutar sí se toma basándose en los datos que se tienen. En lenguaje máquina, los saltos condicionales se toman a partir del valor de los *flags* de condición del Registro de Estado. Como ya hemos comentado, hay muchas instrucciones de las que hemos visto cuya ejecución afecta a estos *flags*, tales como las aritméticas, lógicas, de desplazamiento, etc. Hay, además unas cuantas **instrucciones de comparación y comprobación** que sirven exclusivamente para establecer los *flags* de condición sin modificar los operandos, simplemente comparándolos o comprobándolos.

Las instrucciones de comparación tienen dos operandos. Al destino le restan el fuente y en función del resultado se establecen los *flags* de condición; el resultado no se deja en ningún sitio y los operandos permanecen inalterados. **CMP** realiza comparaciones con registros de datos, mientras que **CMPA** lo hace con los de direcciones. **CMPI** permite comparaciones de datos en memoria o en registros de datos con valores inmediatos. Para las comparaciones en las que ambos operandos están en memoria se utiliza **CMPM** (*compare memory*) la cual solamente permite el direccionamiento indirecto con post-incremento. Esta última instrucción es útil para comparar listas completas de datos o bloques de memoria elemento a elemento.

La instrucción básica de bifurcación condicional es **Bcc**, la cual toma la decisión del salto dependiendo del código de condición establecido (*cc*) y de los valores de los *flags* de condición. En la siguiente transparencia se muestra un cuadro con los distintos códigos de condición del 68000 y los valores de los *flags* que considera cada condición. Se debe resaltar que las direcciones de salto deben estar en el rango de ± 32 Kbytes de la dirección actual.

Así, por ejemplo, la instrucción **BEQ SALIR** salta a la etiqueta **SALIR** si como resultado de una comparación realizada inmediatamente antes, los dos operandos de la comparación eran iguales (*flag Z = 1*).

La instrucción **DBcc** (*decrement and branch*) está pensada para ayudar a los compiladores a implementar los mecanismos de control de bucles. Es poco común en otros procesadores y de dudosa utilidad real. No obstante, en la transparencia puede verse la lógica de su funcionamiento.

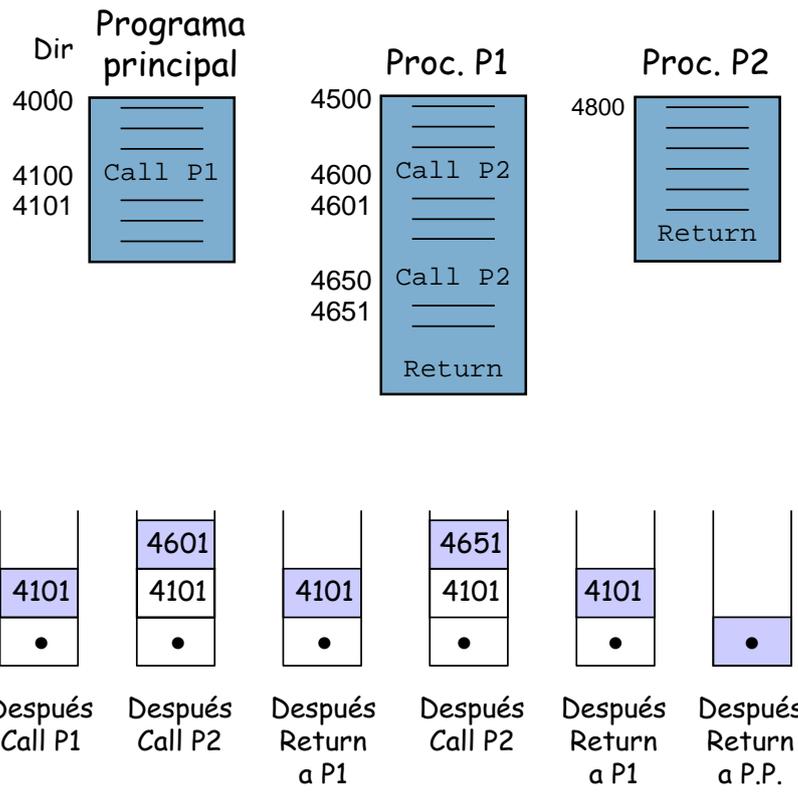
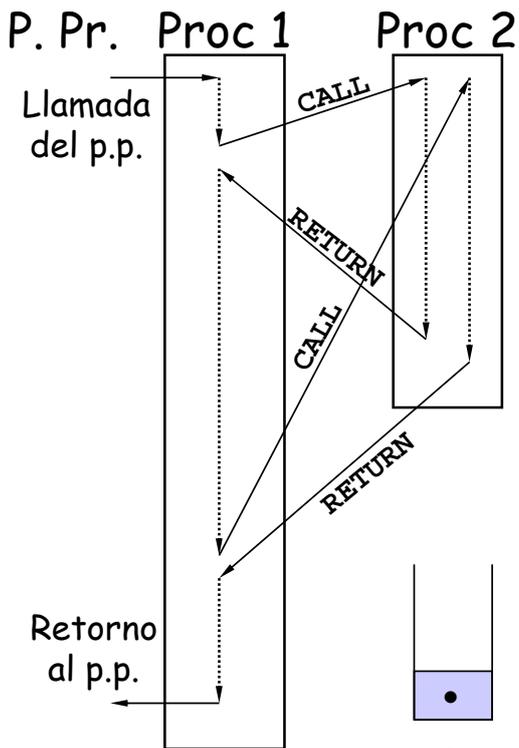
<u>Cod. Cond.</u>	<u>Significado</u>	<u>Expresión Evaluada</u>
EQ	Igual	Z
NE	Distinto	\bar{Z}
GE	Mayor o Igual	$(N \wedge V) \vee (\bar{N} \wedge \bar{V})$
HI	Mayor	$\bar{C} \wedge \bar{Z}$
GT	Mayor	$(N \wedge V \wedge \bar{Z}) \vee (\bar{N} \wedge \bar{V} \wedge \bar{Z})$
LT	Menor	$(N \wedge \bar{V}) \vee (\bar{N} \wedge V)$
LS	Menor o Igual	$C \vee Z$
LE	Menor o Igual	$Z \vee (N \wedge \bar{V}) \vee (\bar{N} \wedge V)$
PL	Positivo	\bar{N}
MI	Negativo	N
T	Cierto	Siempre cierto
F	Falso	Siempre falso
CC	Acarreo = 0	\bar{C}
CS	Acarreo = 1	C
VC	Overflow = 0	\bar{V}
VS	Overflow = 1	V

Aquí se muestran los códigos de condición (cc) que se utilizan como sufijo en las instrucciones de salto condicional Bcc y DBcc.

EJEMPLO

```
    . . .  
    MOVE.L    #$0100,A2    A2 Apunta a la 1ª dir.  
BUCLE SUBI.B    #5,(A2)    Resta 5 a la pos. de memoria  
    ADDI.L    #1,A2        Incrementa puntero  
    CMPA.L    #$0201,A2    ¿A2 en la última dirección?  
    BNE      BUCLE        Si no era la última, repite  
    . . .
```

Aquí tenemos un ejemplo de uso de la bifurcación, Se muestra un fragmento de programa en el que se resta el valor 5 a los contenidos de las posiciones de memoria comprendidas entre la dirección \$0100 y la \$0200, ambas inclusive.



Los procedimientos se utilizan asiduamente en los lenguajes de alto nivel, por lo que el lenguaje máquina debe proporcionar un soporte para que se ejecuten eficazmente. Así, veremos aquí el mecanismo que generalmente se utiliza para realizar las llamadas a procedimientos con el nivel que ofrece el lenguaje máquina.

Sabemos que un procedimiento o subrutina es un grupo de instrucciones que realizan alguna tarea y que puede ser llamado desde diversos puntos de un programa. A diferencia de los saltos, estas llamadas tienen la particularidad de que una vez terminado el trabajo del procedimiento, se devuelve el control a la siguiente instrucción del punto de llamada. Por esto, una de las primeras cosas que hay que hacer en una llamada a procedimiento es **guardar la dirección de retorno** en alguna parte. Veamos algunas posibilidades.

En un registro. Cuando se llama a un procedimiento se guarda siempre la dirección de retorno en un registro preestablecido, así cuando finaliza el procedimiento simplemente hay que tomar la dirección de retorno de ese registro. La pega que tiene es que si desde un programa principal se llama a un procedimiento P_1 , y éste llama a su vez al procedimiento P_2 , al realizar la segunda llamada se machaca la dirección de retorno de la primera llamada.

Este problema se produce porque se utiliza el mismo lugar para guardar la dirección de retorno de todos los procedimientos. Por esto, una mejora podría consistir en guardar la dirección de retorno **en una posición de memoria al comienzo de cada procedimiento**, de tal manera que si el programa principal llama al procedimiento P_1 , se guarda la dirección de retorno en una zona de datos al comienzo del procedimiento P_1 , y cuando éste llama al procedimiento P_2 , esta dirección de retorno se guarda en la zona correspondiente del procedimiento P_2 , con lo que no se machaca la primera dirección de retorno. El problema está en que no permite los procedimientos recursivos ni los reentrantes. (Un procedimiento es reentrante si en un momento dado puede tener activas varias instancias por haber sido llamado desde distintos procedimientos de distintos procesos).

El mejor lugar para guardar la dirección de vuelta es **la Pila**. Cuando se llama a un procedimiento se mete la dirección de retorno en la cima de la pila. (Téngase en cuenta que cada proceso tiene su propia pila de trabajo). Así, si un procedimiento se llama recursivamente, se van guardando sus direcciones de vuelta en la pila, que simplemente va creciendo a medida que se producen las llamadas, y empezará a decrecer según se vaya llegando al final de cada instanciación.

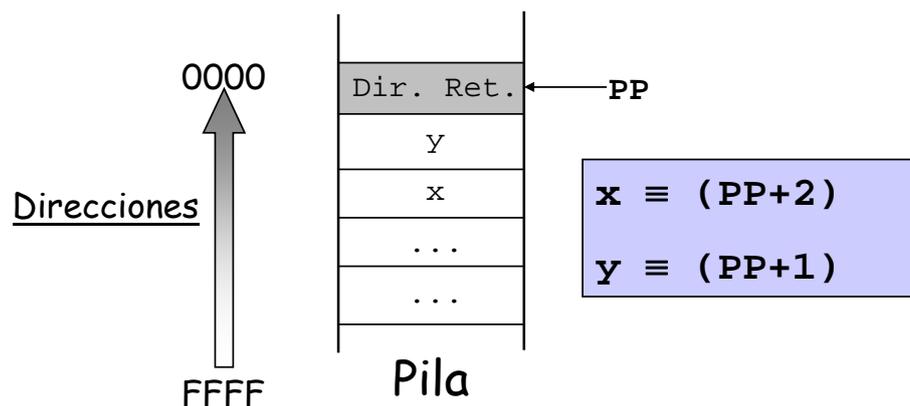
Por esto, todos los procesadores de propósito general disponen de instrucciones de llamada a procedimiento (CALL) que tienen un único parámetro, la dirección del procedimiento llamado, que automáticamente salva el CP en la pila antes de saltar al procedimiento llamado. Obviamente, también disponen de una instrucción de fin de procedimiento (RETURN) que saca el valor de la cima de la pila y lo pone en el CP (Contador de Programa).

```

procedure Ejemplo (x, y: integer) is
  a, b: integer;
begin
  ...
  b := (x * x) - (y * y);
  ...
end Ejemplo;

```

¿Qué pasa con los parámetros?



Una vez que sabemos guardar la dirección de retorno, debemos ocuparnos del **paso de parámetros**. Podríamos pensar en poner los parámetros en una dirección fija de memoria o en registros, pero nos encontraríamos con los mismos problemas que para salvar la dirección de retorno. Por esto, los parámetros también se van a poner en la pila.

Supongamos un programa que en un punto determinado realiza la siguiente llamada al procedimiento *P1*:
`CALL P_1 (x, y)`.

Esta es la secuencia de acciones que deben llevarse a cabo para realizar la llamada a un procedimiento:

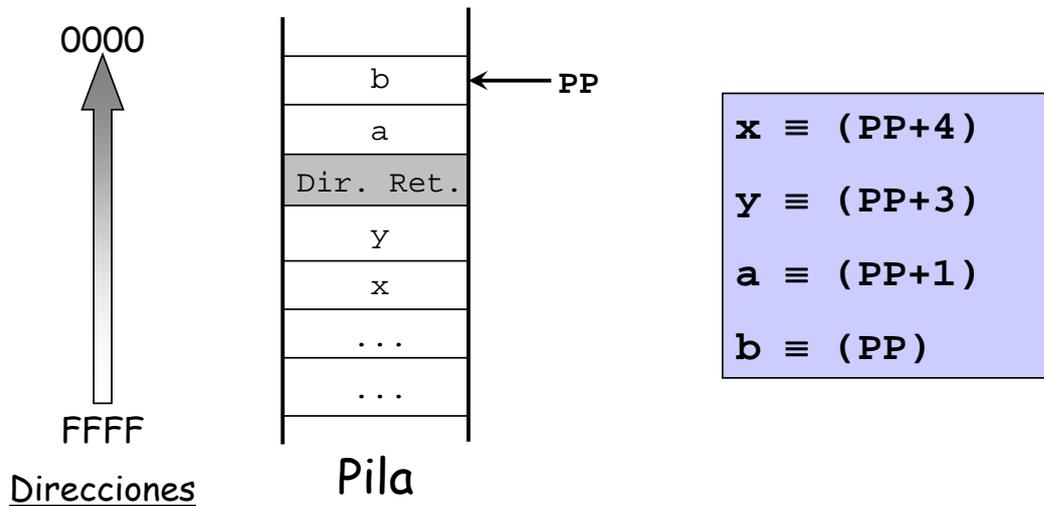
1. Meter en la pila el parámetro *x*.
2. Meter en la pila el parámetro *y*.
3. Meter en la pila el valor del *CP*.
4. Saltar a la dirección del procedimiento *P1*.

Una vez que toma control el procedimiento llamado, tiene en la cima de la pila la dirección de retorno, inmediatamente debajo el parámetro *y* y, por último, el parámetro *x*. Sabiendo esto, se puede acceder a los parámetros mediante un direccionamiento relativo al *PP* (Puntero de Pila): *PP+1*, *PP+2*, ...

Cuando en el procedimiento llamado se alcance la instrucción `RETURN`, su ejecución provocará el sacar de la pila la dirección de retorno y ponerla en el *CP*, con lo que se devolverá el control al procedimiento llamante. En éste, se deberá restaurar el estado de la pila a la misma situación que tenía antes de la llamada al procedimiento, es decir, habrá que sacar de la pila los dos parámetros que se encuentran situados en las dos posiciones más altas de la pila. Una vez que se haya dejado la pila en su situación original se continúa con la ejecución de la siguiente instrucción a la de la llamada al procedimiento.

Esto funciona "aparentemente", aunque, como veremos más adelante, hay situaciones en las que este mecanismo no es válido y necesita algún arreglo.

¡ ... y las variables locales ?



El último aspecto que nos queda por ver de la llamada a procedimientos es la **asignación de espacio para las variables locales**.

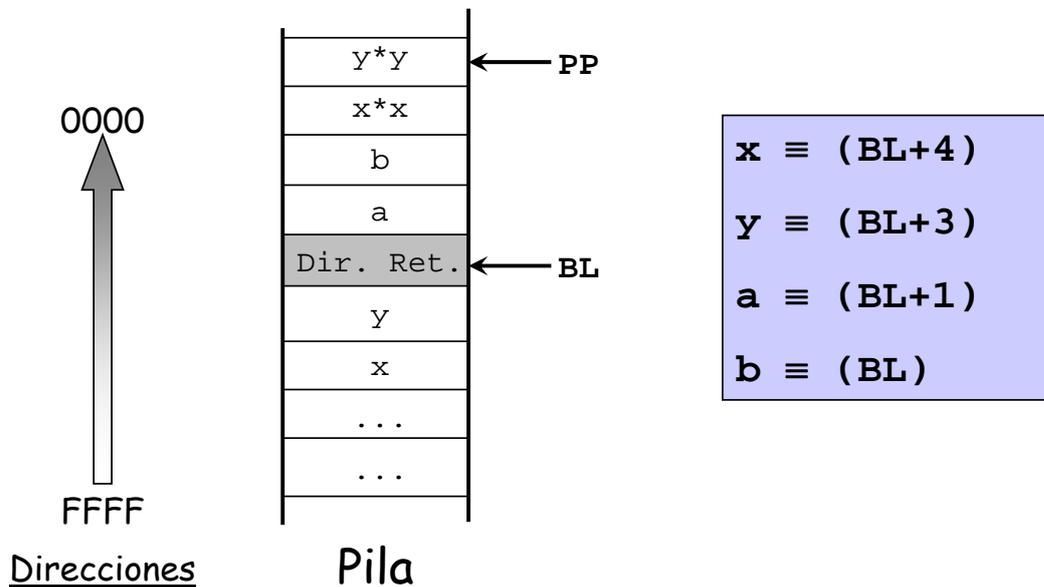
Las variables globales de un programa pueden guardarse o ubicarse en alguna zona fija de memoria, sin embargo, no puede hacerse lo mismo con las variables locales de un procedimiento, pues como ya hemos visto antes, se machacarían estos valores en procedimientos reentrantes o recursivos. Por lo tanto, las variables locales de un procedimiento deben guardarse también en la pila.

Hasta ahora hemos visto que la llamada a un procedimiento deja en la pila los parámetros y la dirección de retorno antes de ceder el control al procedimiento llamado. Ahora, una vez que toma control el procedimiento debe ocuparse de asignar espacio para sus variables locales. Si cada variable necesita una palabra de memoria, lo que se puede hacer es dejar tantos huecos en la pila como variables haya, haciendo corresponder cada hueco a una variable.

En este momento, después de haber reservado espacio para las variables locales, el escenario de la pila será el siguiente: el **PP** estará apuntando a la última de las variables locales, seguida del resto de las variables locales, en orden inverso al de su declaración. En nuestro ejemplo, el **PP** estará apuntando a la variable **b** y **PP+1** estará apuntando a la variable **a**; **PP+2** apuntará a la dirección de retorno, mientras que **PP+3** apuntará al parámetro **y** y **PP+4** al parámetro **x**.

El conjunto de valores constituido por los parámetros, la dirección de retorno y las variables locales se denomina **trama de pila**.

¿... y las variables temporales ?



Acabamos de ver el mecanismo para referenciar las variables locales de cada rutina. No obstante, todavía se pueden presentar problemas. Por ejemplo, supongamos una llamada al procedimiento de la transparencia, en el que se debe calcular la siguiente expresión

$$b := (x * x) - (y * y)$$

Es común que la evaluación de esta expresión se realice por partes, de tal manera que primero se calcula el valor de x^2 y se guarda en la pila, a continuación se calcula el valor de y^2 y también se guarda en la pila. Por último se toman los dos valores de la cima de la pila y se restan.

Pues bien, cuando se ha metido en la pila el valor de x^2 y vamos a acceder al parámetro y ¡resulta que ya no está en $PP+3$! Esto es porque la pila ha crecido, la cima de la pila se ha desplazado y por lo tanto el registro PP ha variado su valor.

Para evitar este problema lo que se necesita es un registro que esté continuamente apuntando a la trama de la pila en curso (puede haber otras tramas debajo si hay llamadas anidadas). A este registro lo vamos a llamar BL (Base Local). Así, lo primero que debe hacer el procedimiento llamado es poner el BL con el valor de PP actual y después reservar el espacio para sus variables locales. Así, posteriormente podrá direccionar los parámetros y las variables locales mediante direccionamiento relativo al BL ($BL-1$, $BL-2$, ...), con lo que ya no importa que el PP varíe su valor debido a variables temporales.

Código	Operandos	Operación
JSR	<CEA>	PC → -(SP); dir → PC
BSR	<etiqueta>	PC → -(SP); PC + despl → PC
RTS		(SP)+ → PC
RTR		(SP)+ → CCR; (SP)+ → PC
LINK	An,#d16	1.- SP-4 → SP 2.- An → (SP) 3.- SP → An 4.- SP-d16 → SP
UNLK	An	1.- An → SP 2.- (SP) → An 3.- SP+4 → SP

Los mecanismos que ofrece el 68000 para implementar llamadas y retornos de subrutinas se apoyan en la pila de trabajo del programa y en su puntero de pila (SP ó A7).

La instrucción básica de salto a subrutina es **JSR** (*jump subroutine*). Como vemos en la transparencia, tiene un único operando que es la dirección de la primera instrucción de la rutina a la que se desea saltar. La ejecución de esta instrucción se realiza en dos pasos:

- 1º. Se mete el contenido del PC (dirección de la siguiente instrucción en secuencia) en la Pila. Esto significa guardar la dirección de retorno, es decir, la siguiente instrucción que se debe ejecutar al término de la subrutina.
- 2º. Se lleva al PC la dirección indicada en el operando de la instrucción.

Además de esta instrucción, también se dispone de **BSR** (*branch subroutine*), pero al igual que su hermana **BRA**, tiene como único operando un byte o una palabra que actúa como desplazamiento relativo al Contador de Programa, por lo que solamente permite saltos en un rango de ± 32 Kbytes sobre el PC. En ensamblador, basta con poner una etiqueta como operando y el ensamblador se encarga de calcular el desplazamiento.

Al finalizar una subrutina, se debe ejecutar la instrucción de retorno al punto de llamada. Ésta suele ser **RTS** (*return from subroutine*), la cual carga en el Contador de Programa la dirección de retorno sacándola de la Pila.

En algunas circunstancias especiales el programador puede necesitar restaurar también los *flags* de condición que había en el momento de la llamada a la subrutina. Para ello, también debe salvarse el CCR en la Pila al comienzo de la ejecución de la subrutina mediante una instrucción como `MOVE CCR, -(A7)`.

Si se había salvado el CCR en la Pila, la instrucción de terminación de la subrutina debe ser **RTR** (*return and restore*), la cual primero saca de la Pila el antiguo CCR y lo copia a su parte del Registro de Estado, y por último saca la dirección de retorno para llevarla al Contador de Programa.

Para los programadores avanzados en ensamblador, el 68000 dispone de dos instrucciones que facilitan la llamada y retorno de subrutina: **LINK** y **UNLK**. Estas instrucciones resultan útiles cuando es necesario reservar espacio en la pila para las variables locales de la subrutina. Para la descripción detallada de su funcionamiento se requiere el conocimiento previo de los mecanismos de llamada a subrutina con paso de parámetros y reserva de espacio para variables locales. En el apartado 3.2 del texto de Clements (*Passing parameters to modules* y *Stack and local variables*) se describe muy bien esta técnica, así como la utilización de **LINK** y **UNLK**.

Son desviaciones del flujo normal de ejecución del programa provocadas por alguna circunstancia "excepcional":

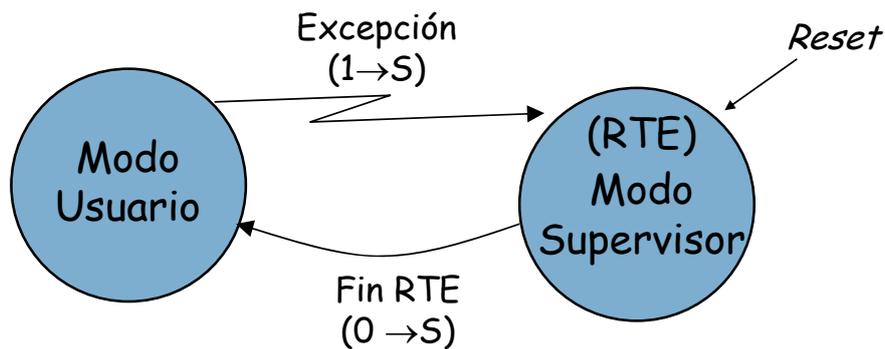
- ✓ *Reset*
- ✓ Violación de privilegio
- ✓ División por cero
- ✓ Interrupciones, etc.

Las excepciones se utilizan para indicar que se ha producido alguna situación especial. Estas situaciones especiales o "excepcionales" pueden generarse **externamente** por el hardware (errores de bus, *reset*, ..., interrupciones en general) y comunicadas a la CPU a través de algunas de sus patas, o **internamente**, por el software, durante la ejecución de una instrucción. Las excepciones también pueden generarlas los programas de usuario (ejecutándose en modo usuario) para llamar la atención del sistema operativo, es decir, para solicitarle un servicio.

Los mecanismos de tratamiento de excepciones están previstos para tratar las interrupciones de manera muy rápida o eficiente.

¿Qué sucede cuando se produce una excepción?

- ① Se salva cierta información de estado
- ② Se pasa a modo Supervisor
- ③ Toma control una Rutina de Tratamiento de la Excepción (RTE)

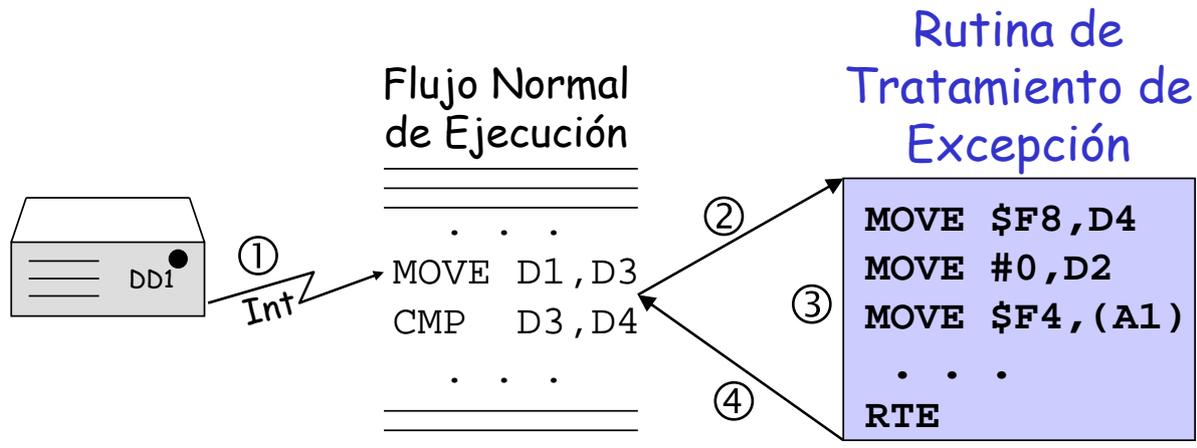


Cuando se produce una excepción hay que interrumpir el programa en ejecución y ceder el control de la CPU a la rutina de tratamiento correspondiente a esa excepción. Cuando finaliza el tratamiento de la excepción se debe continuar con el programa interrumpido; por ello, al producirse la excepción:

- Lo primero que debe hacerse es guardar la información básica del estado actual del procesador (el Contador de Programa y el Registro de Estado).
- A continuación se pasa a modo supervisor para tener control absoluto sobre el sistema.
- Por último se cede el control a la rutina de tratamiento de la excepción (RTE).

Al finalizar la RTE, debe restaurarse el estado que tenía el procesador en el momento de producirse la excepción y devolver el control al programa interrumpido.

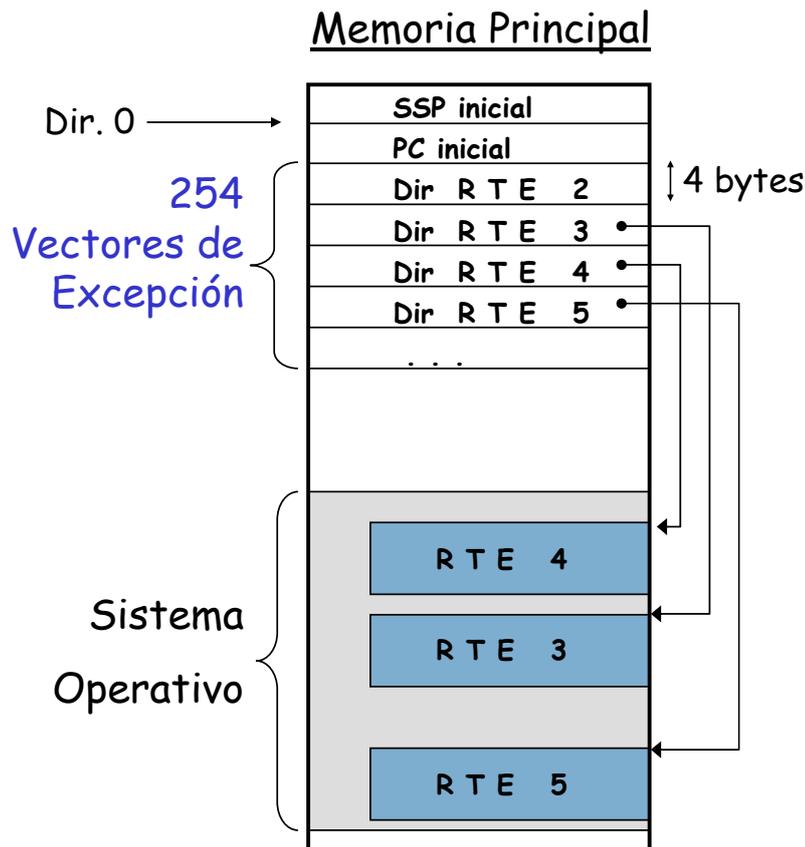
El arranque del 68000 se produce como una excepción (*RESET*), por lo que ya se arranca en modo supervisor. Cuando finaliza el proceso de inicialización del sistema operativo y se cede el control a los programas de usuario, se pasa a modo usuario.



Las excepciones se utilizan para indicar que se ha producido alguna situación especial. Estas situaciones especiales o “excepcionales” pueden generarse **externamente** por el hardware (errores de bus, *reset*, ..., interrupciones en general) y comunicadas a la CPU a través de algunas de sus patas, o **internamente**, por el software, durante la ejecución de una instrucción. Las excepciones también pueden generarlas los programas de usuario (ejecutándose en modo usuario) para llamar la atención del sistema operativo, es decir, para solicitarle un servicio.

Los mecanismos de tratamiento de excepciones están previstos para tratar las interrupciones de manera muy rápida o eficiente.

Cada excepción (o situación excepcional) tiene asociado un número de vector o **vector de excepción**. Un vector de excepción es una posición de memoria principal donde se encuentra la dirección de la rutina que sabe tratar adecuadamente la excepción o situación que se ha producido.



Aunque es en el capítulo dedicado a los Sistemas de Entrada/Salida donde estudiaremos con detalle las excepciones, aquí vamos a avanzar algunos datos referentes al 68000.

Cada excepción (o situación excepcional) tiene asociado un número de vector o **vector de excepción**. Un vector de excepción es una posición de memoria principal (ocupa una doble palabra) donde se encuentra la dirección de la rutina que trata adecuadamente la excepción o situación que se ha producido.

En este procesador, hay 256 vectores. Cada vector ocupa 4 bytes de memoria, pues se necesitan 24 bits para indicar la dirección de una rutina de tratamiento de excepción (RTE).

Cada excepción tiene su correspondiente vector de excepción. El tratamiento de la excepción RESET es especial, pues en el proceso de arranque se utilizan los 2 primeros vectores de excepción (vectores 0 y 1) para tomar de ahí los valores iniciales del puntero de pila del supervisor (SSP) y del contador de programa (PC). Por esto, la primera excepción "normal" es la correspondiente al vector 2, el cual está ubicado a partir de la dirección 8.

Al espacio de memoria utilizado para los 256 vectores de excepción se le conoce como "Tabla de Vectores de Excepción", y está situada en la dirección 0 de memoria principal.

De los 256 vectores, los 64 primeros (0 a 63) y el 255, están reservados para el sistema, el resto (64 a 254) son los disponibles para las excepciones definidas por el usuario.

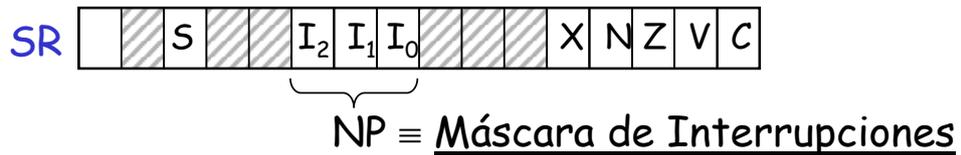
Las interrupciones reservadas para el sistema, están dedicadas a interrupciones predefinidas (0 a 15 y 24), autovectores (25 a 31) y traps (32 a 47). El resto aunque está sin asignar, no pueden ser utilizadas por el usuario.

El tratamiento de interrupciones se tratará con detalle en el capítulo dedicado a los Sistemas de Entrada/Salida.

INTERRUPCIONES
 Son excepciones asíncronas generadas
 por dispositivos externos a la CPU
 (por ej. al pulsar una tecla)

Cada interrupción tiene una prioridad o "nivel" (NI)	El programa en ejecución tiene un nivel de privilegio (NP)
Para aceptar una interrupción (NI > NP) ó (NI = 7) Al aceptar una interrupción: NI → NP	

☞ El 68000 contempla hasta 7 niveles de interrupción



Como ya hemos visto, una interrupción es una petición de servicio generada por un dispositivo periférico.

El 68000 dispone de dos sistemas para tratar interrupciones. Uno está diseñado especialmente para tratar con los periféricos de 16 bits de la familia 68000, mientras que el otro se utiliza para mantener la compatibilidad con los dispositivos anteriores de la familia 6800 (aunque también se puede utilizar con los dispositivos de 16 bits). Aquí comentaremos algunos aspectos del tratamiento de interrupciones específico para los periféricos de 16 bits.

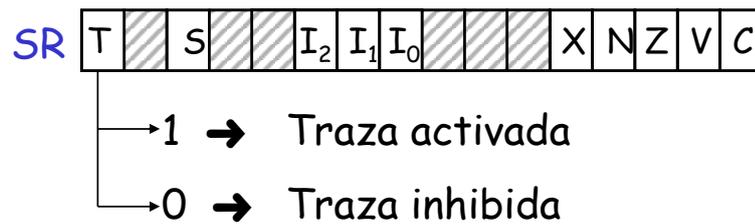
Las interrupciones tienen prioridades o niveles de privilegio, de tal manera que cuando se produce una interrupción, ésta tiene asociada una **prioridad entre 1 y 7**. Esta prioridad se le comunica al 68000 codificada como un número binario mediante las tres patas de *Interrupt Privilege Level* del procesador: IPL0, IPL1 e IPL2 (una prioridad de nivel 0 significa ausencia de interrupción).

Por otra parte tenemos que un programa en ejecución también tiene una prioridad, denominada **Nivel del Procesador**. Este Nivel del Procesador viene indicado por 3 bits en el Registro de Estado: I₀, I₁ e I₂, con lo que especifica también un privilegio de ejecución entre 0 y 7.

Cuando se produce una interrupción y se activan las patas IPL del 68000, para que éste acepte la interrupción, la prioridad de la interrupción debe ser mayor que el Nivel del Procesador. No obstante, si la prioridad de la interrupción es 7 (interrupción no enmascarable), ésta se atiende indefectiblemente.

Ya que el Nivel del Procesador indica qué interrupciones se aceptan y cuáles no, también se le conoce como "Máscara de Interrupciones".

Cuando se acepta una interrupción, se pasa a modo supervisor y se guarda en la pila el Contador de Programa y el contenido del Registro de Estado que había en el momento de producirse la interrupción. A continuación se identifica la interrupción producida y, mediante el mecanismo de los vectores de interrupción, se cede control a la rutina de tratamiento de interrupción correspondiente.



Si T=1, al final de la ejecución de cada instrucción se provoca la excepción TRAZA (y se pone T=0)

Útil para la ejecución de programas "paso a paso"

Para terminar de ver todos los bits del Registro de Estado, hablaremos del "**Bit de Traza**". Cuando este bit del SR está activado (se dice entonces que se está en "modo traza"), al término de la ejecución de cada instrucción, se produce la excepción de "traza". La rutina de tratamiento asociada a esta excepción suele ser parte de un monitor o depurador que se utiliza para poder seguir la ejecución de los programas "paso a paso" y ayudar en la depuración de errores.

Código	Operandos	Operación
Scc	<ADEA>	if cc then \$FF → dest
NOP		PC + 2 → PC
TRAP	#<d4>	PC → -(SSP) SR → -(SSP) vector #<d4> → PC
TRAPV		if V then TRAP 7
CHK	<DEA>, Dn	if Dn(0..15) < 0 or Dn(0..15) > [DEA] then TRAP 6
RTE		if S then (SP)+ → SR; (SP)+ → PC; else TRAP 8

Por último vamos a comentar algunas instrucciones variadas difíciles de encasillar en los grupos que hemos visto anteriormente.

scc (*set byte conditionally*) es una instrucción poco usual en otros procesadores. En la ejecución de esta instrucción, si se cumple la condición indicada por *cc*, se ponen a 1 todos los bits del operando; si la condición no se cumple, se ponen a cero todos sus bits.

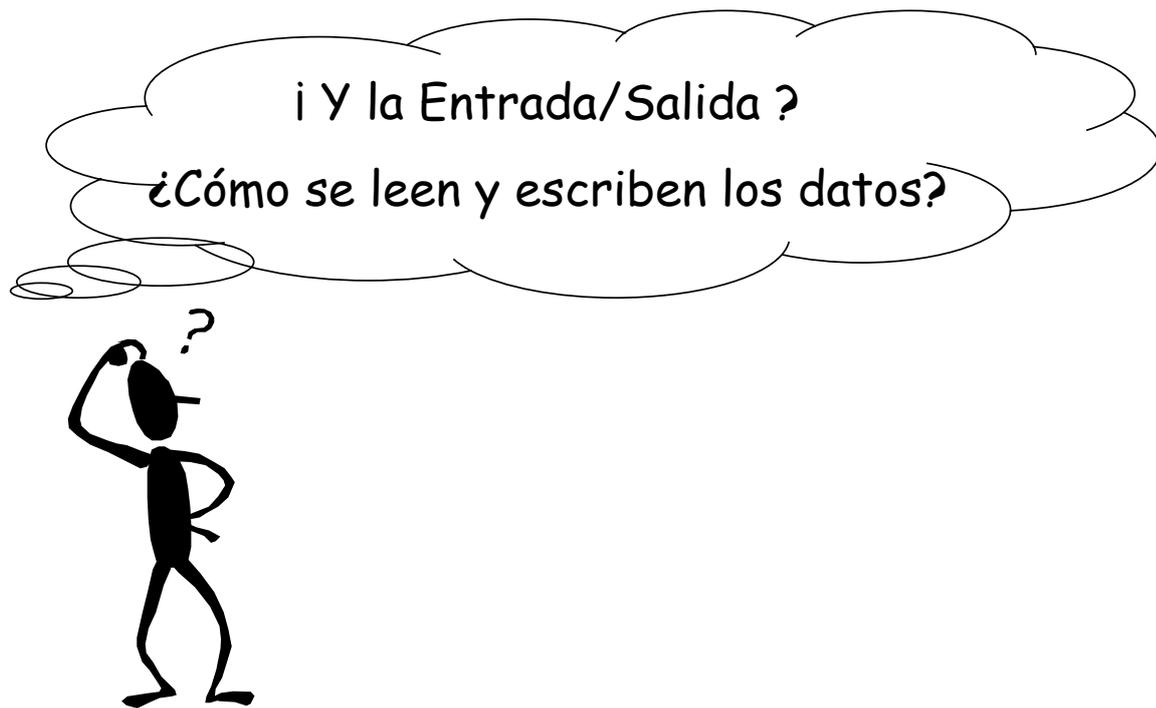
NOB (*no operation*) no tiene ningún efecto en la CPU más que incrementar el Contador de Programa para apuntar a la siguiente instrucción. Útil para la depuración de programas.

La forma de simular excepciones por programa es mediante **TRAP**. Esta instrucción da lugar a que se realicen las mismas acciones que si se hubiera producido la excepción cuyo número o vector se indica como operando. A las excepciones o interrupciones generadas mediante una instrucción explícita por programa también se las conoce como "interrupciones software". El parámetro de esta instrucción es un valor inmediato de 4 bits, ya que hay 16 posibles interrupciones software o traps (0 a 15). Los 16 vectores de excepción asociados a las interrupciones son los que van del 32 al 47; así, el trap 0 tiene asociado el vector 32; el trap 1, el 33, el trap 2, el 34; y así sucesivamente.

En un momento dado, si el bit *V* (*overflow*) del Registro de Estado está activado, al ejecutar la instrucción **TRAPV** se produce la excepción de *overflow* (la de vector 7). Esto debe dar control a una rutina que trata las situaciones de *overflow*. Si el bit *V* está a 0, la ejecución de **TRAPV** no tiene ningún efecto.

La instrucción **CHK** se utiliza para comprobar si el índice utilizado para acceder a un elemento de una tabla está dentro de los rangos válidos, es decir, de los límites de la tabla. Comprueba si la palabra de menor peso del registro *Dn* es menor que cero o mayor que el primer operando. De ser cierta cualquiera de estas dos condiciones ejecuta un **TRAP 6**.

Ya hemos comentado anteriormente que la última instrucción de una subrutina debe ser la instrucción **RTS**, la cual restaura en el Contador de Programa la dirección de retorno. Esto no es suficiente cuando la subrutina en cuestión es una rutina de tratamiento de excepción, pues como ya hemos visto, al producirse una excepción no solo se guarda en la Pila la dirección de retorno, sino también una copia del Registro de Estado (SR). Por esto, la última instrucción de las rutinas de tratamiento de excepciones debe ser **RTE** (*return from exception*), la cual, en primer lugar, restaura el SR sacando su valor de la cima de la pila, y a continuación restaura el Contador de Programa con el siguiente valor guardado en la pila.



¿Y cómo se realiza la entrada/salida?

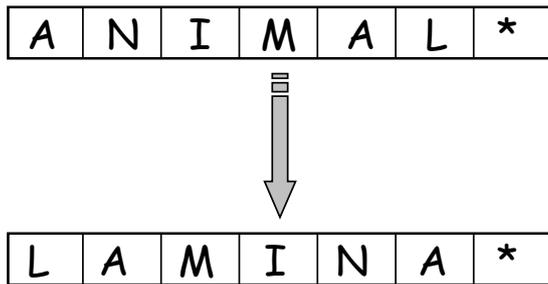
Por supuesto que faltan los mecanismos de E/S, pero es que sobre esto hay mucho que hablar, y lo veremos con detalle en el capítulo dedicado a los Sistemas de Entrada/Salida.

MOVE	<ea>, Dn	CMP	<ea>, Dn
MOVE	Dn, <ea>	CMP	Dn, <ea>
MOVEA	<ea>, An	CMPI	#<dato>, <ea>
ADD	<ea>, Dn	BEQ	dirección
ADD	Dn, <ea>	BNE	dirección
ADDI	#<dato>, <ea>	BRA	dirección
SUB	<ea>, Dn	MULU	<ea>, Dn
SUB	Dn, <ea>	CLR	<ea>
SUBI	#<dato>, <ea>	LEA	Valor, An
		LEA	Valor(An), An

Como hemos visto, son muchas y muy variadas las instrucciones del Motorola 68000 (como buen representante de una CPU CISC). No obstante, la mayoría de los programas convencionales pueden realizarse con unas cuantas instrucciones sencillas.

En el cuadro de la figura se muestran unas cuantas instrucciones de propósito general con las que pueden escribirse muchos programas convencionales.

La notación **Dn** se refiere a cualquier registro de datos, mientras que **An** lo hace a cualquier registro de direcciones.



```

ORG      $0
DC.L    $80000 ;SSP inicial
DC.L    START ;PC inicial
ORG     $400
TIRA   DC.B  'ANIMAL*'

START  LEA      TIRA,A5
MOVEA.W SP,A6
METE   MOVE.B  (A5)+,-(SP)
CMPI.B #'*',(A5)
BNE    METE

LEA    TIRA,A5
SACA   MOVE.B  (SP)+,(A5)+
CMPA.W SP,A6
BNE    SACA

BREAK ;parada de la CPU
END   ; fin del programa

```

Supongamos que nos encargan un sencillo programa en el que dada una tira de caracteres, cuyo final está marcado por el símbolo "*" (asterisco), hay que invertir el orden de los caracteres, dejando la tira resultante en la misma dirección de memoria que la tira original.

Para solucionarlo, vamos a aprovechar la estructura lógica de la pila. Primero copiaremos todos los caracteres de la ristra metiéndolos en la pila, y a continuación los empezamos a sacar de la pila copiándolos a partir de la dirección original de la tira en memoria.

En la solución que tenemos en la figura, las sentencias en negrita son instrucciones máquina, mientras que las demás son "directivas del ensamblador".

La directiva `ORG` indica la dirección de memoria en la que debe cargarse el código o los datos que vienen a continuación. `DC.L` significa "Define Constant con tamaño Long Word" y es una reserva de espacio en memoria que se inicializa con el valor que se indica como parámetro. Así, las tres primeras sentencias del programa indican que a partir de la dirección 0 deben reservarse dos palabras largas (de 4 bytes), la primera con el valor inicial \$80000, y la segunda con la dirección de la primera instrucción de nuestro programa, la que tiene la etiqueta `START`. A continuación, mediante `ORG $400` nos situamos en dicha dirección de memoria y se reserva una serie de bytes para la tira de caracteres 'ANIMAL*', a cuya dirección hace referencia la etiqueta `TIRA`.

Veamos ahora las instrucciones máquina. Comenzamos por cargar la dirección de la tira en un registro de direcciones con la instrucción `LEA`, y a continuación salvamos el valor actual del registro de pila en el registro de direcciones `A6`.

Ahora se entra en un bucle en el que se van copiando los caracteres a la pila mediante `MOVE.B (A5)+,-(SP)`. Después de esta instrucción, `A5` apunta al siguiente carácter de la tira (aún sin copiar). Con `CMPI`, comparamos el siguiente carácter con "*", y mientras no sea igual se bifurca al comienzo del bucle (`METE`) para seguir metiendo caracteres en la pila.

Una vez que se detecta el símbolo final de la tira, se debe entrar en otro bucle en el que se van sacando los caracteres de la pila (en orden inverso al que se metieron) y se copian empezando por la dirección original de la tira.

Otra vez, se empieza por cargar en `A5` la dirección original de la tira, y se entra en el mencionado bucle. Ahora, mediante `MOVE.B (SP)+,(A5)+` se saca un carácter de la pila, se copia a la dirección en memoria y se incrementan los registros de dirección utilizados. Para saber cuándo se ha acabado de sacar todos los caracteres de la pila se compara el valor actual del puntero de pila con el valor que tenía al comenzar el programa, y mientras no sea igual se salta al comienzo del bucle (`SACA`).

La instrucción `STOP` carga el operando en el registro de estado y pone a la CPU en estado de `HALT`. La última sentencia (`END`) no es una instrucción máquina, sino una directiva que le indica al ensamblador que el programa fuente ha terminado.

```

program Ejemplo;
{ Se transfiere el contenido de una tabla fuente a una
  tabla destino segun el valor de la constante Modo.
  - Si el bit 0 de Modo vale 0: se trasfiere hasta encontrar el valor 255.
  - Si el bit 0 de Modo vale 1: se transfiere toda la tabla.
}
const
  Modo      : byte := 0;
  LongTabla : byte := 10;
  TablaF    : array[0..LongTabla-1] of byte = (12,34,12,45,76,255,87,76,55,10);
var
  TablaD: array [0..LongTabla-1] of byte;
  i      : integer;

begin
  if (Modo and $01) = 0 then
  begin
    i := -1;
    repeat
      i:=i+1;
      TablaD[i] := TablaF[i];
    until TablaF[i] = $FF;
  end
  else
    for i := 0 to LongTabla-1 do
      begin
        TablaD[i] := TablaF[i]
      end;
  end.

```

PROGRAMA
PASCAL

Vamos a ver otro ejemplo con el que podemos comparar un programa Pascal y su correspondiente en ensamblador del 68000.

En la diapositiva de arriba se muestra un programa Pascal que debe transferir el contenido de una tabla fuente u origen (TablaF) a una tabla de destino (TablaD).

La transferencia puede realizarse de dos modos. Si el bit 0 de la variable `Modo` es 0, la transferencia se realiza hasta que se encuentre el valor 255 entre los valores de la tabla origen; si el `Modo` es 1, se transfiere incondicionalmente el contenido completo de la tabla origen a la tabla destino.

Veamos en la siguiente diapositiva cómo podría ser un programa equivalente en ensamblador del Motorola 68000.

```

*          P R O G R A M A       E N S A M B L A D O R

* ===== Vectores de interrupcion =====
          ORG          $0
          DC.L         $10000      ; Base de la pila del supervisor
          DC.L         begin      ; Contador de Programa inicial

* ===== Definicion de constantes =====
          ORG          $400
Modo      DC.B         1
TablaF    DC.B         0,1,2,3,4,$FF,6,7,8,9
LongTab   EQU          10

* ===== Definicion de variables =====
TablaD    DS.B         LongTab
* Para el índice i se utiliza el registro D0

```

En nuestro programa en ensamblador vamos a distinguir dos tipos de sentencias: las instrucciones (en negrita) y las directivas de ensamblaje. Las instrucciones son las que se van a traducir una por una a sus correspondientes instrucciones máquina, mientras que las directivas de ensamblaje le indican al programa traductor (el ensamblador) ciertas acciones que debe tomar.

Un programa se compone de código (instrucciones) y de datos, y ambos están situados en memoria principal. En nuestro ejemplo, vamos a situar primero los datos del programa y a continuación las instrucciones.

Con la directiva **ORG**, se le indica al ensamblador la dirección de memoria donde debe situar lo que viene a continuación. Así, vemos que se comienza con **ORG \$0** para indicar que se empieza en la dirección cero de memoria. Las dos directivas siguientes: **DC** (*define constant*) reservan un espacio de memoria para una palabra larga cada una (**.L**), y las inicializan con dos valores.

Las dos primeras palabras largas del espacio de memoria del 68000 le indican estas dos cosas:

1. Dirección de la base de pila (**SP**) del supervisor: \$10000
2. Valor inicial del Contador de Programa (**PC**): valor de la etiqueta **begin**

A continuación pegamos un salto hasta la dirección \$400 de memoria, donde situamos las constantes y las variables de nuestro programa. Como vemos cada una de ellas tiene su correspondiente etiqueta. La variable en la que manteníamos el índice de los bucles en el programa Pascal, la mantendremos ahora en el registro **D0**, por lo que no declaramos ninguna variable en memoria para ello.

También podemos ver aquí la definición de constantes literales. Vamos a definir aquí la longitud de las tablas, que es 10. Para ello utilizamos la directiva de ensamblaje **EQU**. Esta directiva simplemente asigna un símbolo a un valor, de tal manera que cuando se utiliza ese símbolo posteriormente en el programa, será exactamente equivalente a poner dicho valor en su lugar.

El espacio de memoria desde la dirección 0 hasta la \$3FF siempre debe dejarse reservada para los vectores de interrupción, pudiendo poner el programa de usuario a partir de la dirección \$400, donde nosotros hemos puesto la variable y constantes.

```

* ===== Cuerpo del Programa (begin) =====
begin
  ORG      $500      ; Siempre debe comenzar en dirección PAR
if        MOVE.B   Modo,D0
          AND.B    D0,$01
          BNE     else

then      MOVE.W   #-1,D0
          MOVEA.L #TablaF,A0
          MOVEA.L #TablaD,A1

repeat   ADD.W    #1,D0
          MOVE.B  0(A0,D0),D1
          MOVE.B  D1,0(A1,D0)

until    CMP     #$FF,D1
          BNE     repeat
          BRA     fin

else     MOVE.W   #0,D0      ; Inicializa el índice D0 a cero
          MOVEA.L #TablaF,A0
          MOVEA.L #TablaD,A1

for      MOVE.B  0(A0,D0),D1
          MOVE.B  D1,0(A1,D0)
          ADD.W   #1,D0
          CMP.W  #LongTab,D0
          BNE    for

fin      BREAK                    ; Detiene el procesador
          END                      ; Fin del ensamblaje

```

El cuerpo de nuestro programa (las instrucciones) las ponemos a partir de la dirección \$500 con la correspondiente directiva **ORG**. Hay que asegurarse de que **la primera instrucción máquina de un programa siempre esté en una dirección par**.

La última instrucción del programa es la instrucción **BREAK**, que detiene el procesador, por lo que deja de alimenta y ejecutar instrucciones. Esta instrucción siempre debe ser la última de nuestros programas de prácticas.

La última sentencia de un programa siempre debe ser la directiva **END**, que le indica al ensamblador que deje de traducir instrucciones, por lo que cualquier cosa que haya después de esta directiva, será ignorada.