CURSO DE PROGRAMACION LENGUAJE ENSAMBLADOR

Introducción.

Los traductores se dividen en dos grupos dependiendo de la relación entre lenguaje fuente y lenguaje objeto. Cuando una instrucción de un lenguaje fuente nos genera una única instrucción numérica máquina decimos que ese lenguaje fuente es Ensamblador.

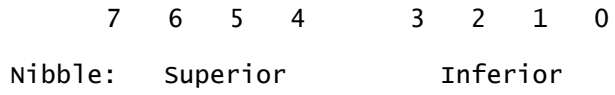
Cuando la instrucción simbólica de lenguaje fuente (como Basic, Cobol, Fortran, etc) nos genera varias instrucciones máquina o varias instrucciones simbólicas de otro lenguaje, decimos que el traductor que realiza la transformación es un compilador.

Las características fundamentales de un Ensambladores que cada una de sus sentencias es una codificación simbólica de una instrucción numérica máquina. Otra característica que presenta es que nos permite llegar a usar cualquier recurso del sistema, cosa que no nos permiten los lenguaje de alto nivel.

Programar en Ensamblador es como programar en un lenguaje máquina ya que hay una identificación entre lenguaje máquina de 0 y 1 y un lenguaje simbólico.

Longitud de los Datos.

Los tipos principales de datos permitidos por los micro-programa de Intel tiene una longitud de palabras de 1, 4, 8, 16 y 32 bits y se denominan, respectivamente, Bit, Nibble, Byte, Palabra, Doble Palabra.



Los números decimales se pueden almacenar de varias formas, como por ejemplo:

- Desempaquetado, donde cada byte contiene un dígito.

Ejemplo: 1434 → 01 04 03 04 → 0000 0001 0000 0100 0000 0011 0000 0100

- Empaquetado, donde cada byte contiene dos dígito.

Ejemplo: 1434 → 14 34 → 0001 0100 0011 0100

- Agrupaciones superiores al byte:

Palabra → 2 bytes.
Doble Palabra → 2 palabras
Cuádruple Palabra → 4 palabras
Párrafo → 16 bytes.
Página → 256 bytes (normalmente).
Segmento → 64k bytes (normalmente).

Origen y destino.

Los términos origen y destino se usan para distinguir la situación de los operandos especificados por las instrucciones de programación.

Ej: `MOV ax , bx ; BX es el operando origen y AX es el operando destino.`

Efectivamente, la instrucción significa... "mover el dato contenido en el operando origen (BX) al operando destino (AX)".

Familias de Procesadores 8086.

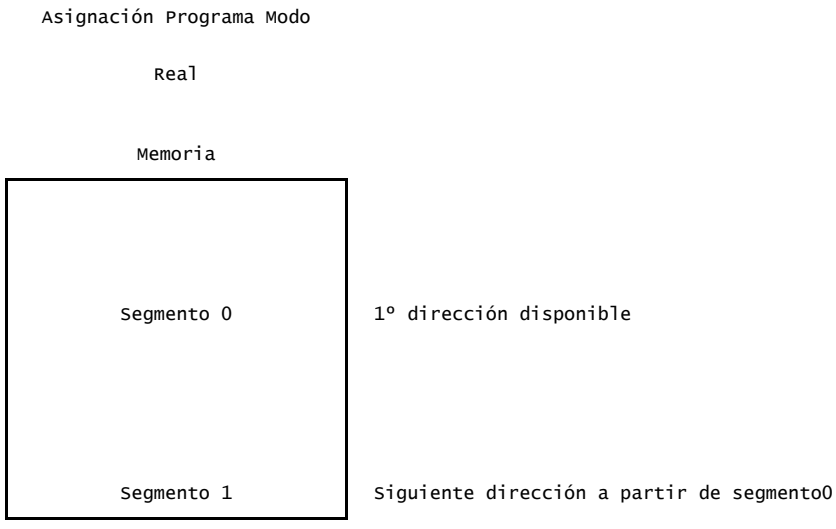
Procesador	Modos disponibles	Memoria Direccionable	Tamaño del Registro
8086 / 8088	Real	1 MegaB	16 bits
80186 / 80188	Real	1 MegaB	16 bits
80286	Real y Protegido	16 MegaB	16 bits
80386	Real y Protegido	4 GigaB	16 o 32 bits
80486	Real y Protegido	4 GigaB	16 o 32 bits

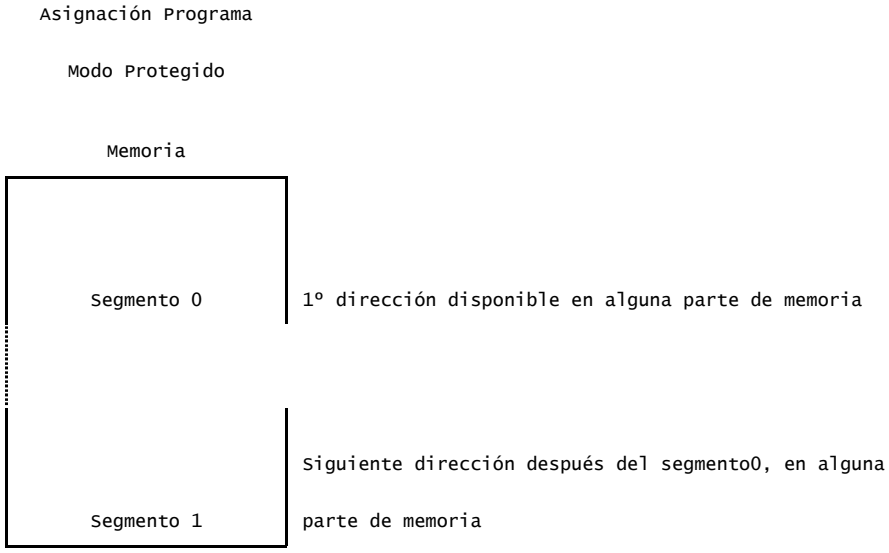
En modo Real solo se puede ejecutar a la vez un proceso. El sistema operativo DOS solo funciona en modo real. En el modo Protegido, más de un proceso pueden ser activados a la vez.

Arquitectura de Segmentos.

Vamos a definir registros como elementos con un número determinado de bits que usa el procesador para hacer unas determinadas operaciones. Vamos a definir segmento como una porción de memoria seleccionada por el procesador para realizar cierto tipo de operaciones.

Con la llegada de procesadores en modo protegido, la arquitectura de segmento consiguió que los segmentos puedan separarse en bloques diferentes para protegerlos de interacciones indeseables. La arquitectura de segmentos realizó otro cambio significativo con el lanzamiento de procesadores de 32 bits, empezando con el 80386, que minimizan las limitaciones de memoria de la arquitectura de segmentos de los 16 bits, siendo, además, compatibles con éstos de 16 bits. Ambos ofrecen paginación para mantener la protección de los segmentos. En DOS los segmentos se asignan normalmente adyacentes uno al otro.



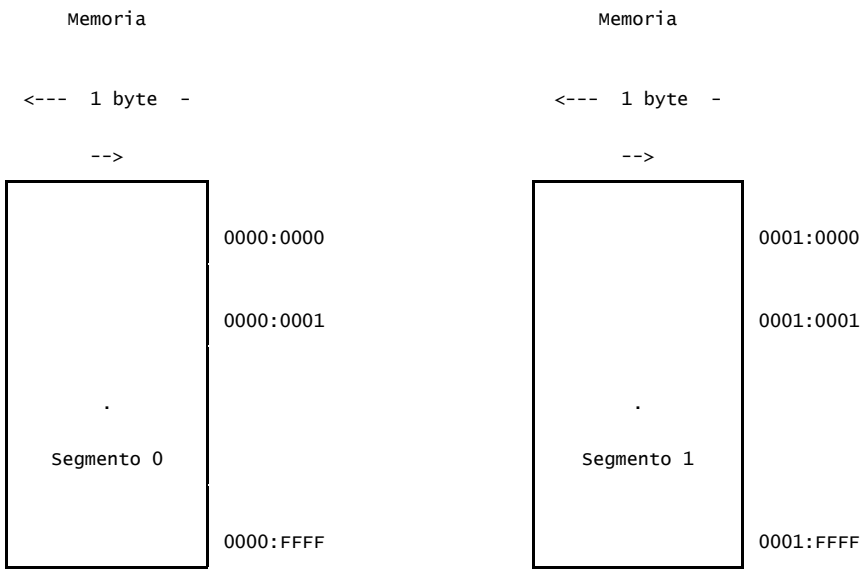


En modo Protegido los segmento estarían en cualquier parte de memoria. El programador no sabe donde están ubicados y no tiene ningún control sobre ellos.

Los segmentos pueden incluso moverse a una nueva posición de memoria o cambiarse al disco mientras que el programa se está ejecutando.

Direccionamiento de los segmentos.

Es un mecanismo interior que combina el valor del segmento y un valor de desplazamiento para crear una dirección. Las 2 partes representan una dirección 'segmento:desplazamiento'.



La porción del segmento es siempre de 16 bits. La porción del desplazamiento es de 16 y 32 bits. En modo real el valor del segmento es una dirección física que tiene una relación aritmética con el desplazamiento.

El segmento y el desplazamiento crean junto una dirección física de 20 bits, con la que se puede acceder a un MegaB de memoria (2^{20}), aunque, por ejemplo, el sistema operativo de IBM usa sobre 640k de memoria por programa.

Vamos a considerar, por defecto, que tratamos con un desplazamiento de 16 bits. El segmento seleccionará una región de 64k y usaremos el desplazamiento para seleccionar 1 byte dentro de esa región. La forma de hacerlo sería:

1º El procesador desplaza la dirección del segmento 4 posiciones binarias a la izquierda y la rellena con 0. Este funcionamiento tiene el efecto de multiplicar la dirección del segmento por 16.

2º El procesador añade esta dirección de segmento de 20 bits resultante a la dirección de desplazamiento de 16 bits. La dirección de desplazamiento no se cambia.

3º El procesador usa la dirección de 20 bits resultante, a menudo llamada dirección física, al acceder a una posición en el MegaB de espacio direccionado.

Ejemplo: Hexadecimal --> 5 3 C 2 : 1 0 7 A

Binario ------> 0101 0011 1100 0010:0001 0000 0111 1010

1) 0101 0011 1100 0010 0000

2) 0001 0000 0111 1010 +

———————————————

0101 0100 1100 1001 1010 → 5 4 C 9 A → Dirección Física

Ejemplo: Hexadecimal --> 1 3 F 7 : 3 8 A C

Binario ------> 0001 0011 1111 0111:0011 1000 1010 1100

1) 0001 0011 1111 0111 0000

2) 0011 1000 1010 1100 +

———————————————

0001 0111 1000 0001 1100 → 1 7 8 1 C → Dirección Física

Cuando trabajamos en Ensamblador la memoria la dividimos en 4 regiones de 64k. Estas serían:

- Segmento de Código que contiene los código de instrucción ⇒ el programa que se está ejecutando.
- Segmento de Datos que guarda las variables del programa.
- Segmento de Pila con información referente a la pila.

- Segmento Extra o área de datos complementario, usada generalmente con operaciones con cadenas.

Las dirección base actuales de cada segmento se guardan en registros punteros especiales de 16 o 32 bits, denominados Registro de Segmento.

Tipos de Registros.

Todos los procesadores 8086 tiene la mismo base de registros de 16 bits. Se puede acceder a algunos registros como 2 registros separados de 8 bits. En el 80386/486 se puede acceder a registros de 32 bits.

L → Low (Baja)

H → High (Alta)

Registro de Datos		
AH	AL	AX: Acumulador
BH	BL	BX: Base
CH	CL	CX: Contador
DH	DL	DX: Extra
8 bits	8 Bits	Total 16 bits

AX: Funciona como AC en algunas ocasiones. Realiza operaciones como entrada/salida de datos, multiplicación, división, operaciones con decimales codificados en binario, etc.

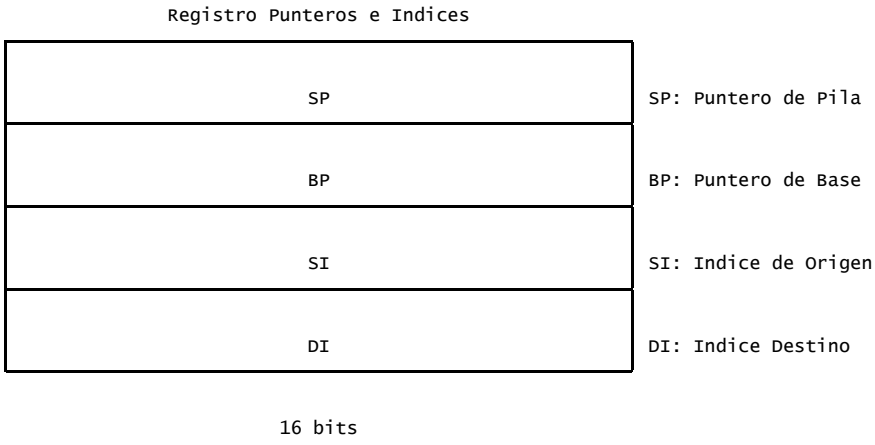
BX: Funciona como registro Base, en algunas ocasiones, para referenciar direcciones de memoria En estos casos mantiene la dirección de base, comienzo de tabla o matrices, en la que la dirección se determina usando valores de desplazamiento.

CX: Funciona como registro Contador, en algunas ocasiones, es decir, cuenta el número de bits o palabras en una determinada cadena de datos durante los operaciones con cadenas.

Ej: Si se va a mover de un área de memoria a otra n palabras, CX mantiene inicialmente el número total de palabras a desplazar llevando la cuenta de la palabra o byte que va siendo trasladada.

En las instrucciones de desplazamiento y rotación CL se usa como contador.

DX: Se usa en la multiplicación para mantener parte del producto de 32 bits o en las divis. para mantener el valor del resto. Y en operaciones de Entrada/Salida de datos para especificar la dirección del puerto de E/S usado.



Los registros Indices SI y DI y los registros Punteros SP y BP guardan los valores de desplazamiento empleados para el acceso a determinadas posiciones de memoria Una característica importante de los 4 registros es que se pueden usar operaciones arit méticas y lógicas de modo que los valores de desplazamiento que almacenan

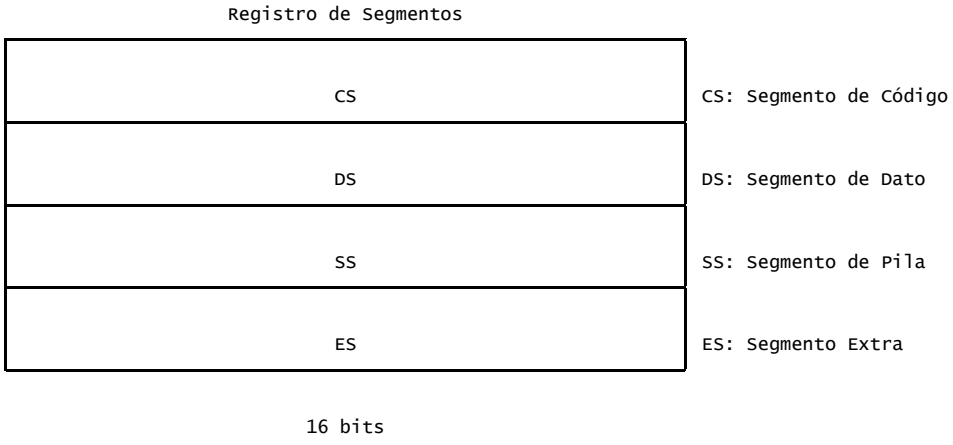
pueden ser el resultado de cálculos previos.

SP: Apunta a la posición de la cima de la pila del segmento de pila en memoria. Es un registro usado para guardar un valor de desplazamiento que direcciona la posición de un operando origen durante operaciones de tratamiento de cadenas.

BP: Apunta a una zona dentro de la pila dedicada al almacenamiento de datos.

SI: Es usado como registro índice en ciertos modos de direccionamiento indirecto. También puede guardar un valor de desplazamiento indirecto. Se usa para almacenar un desplazamiento que direcciona la posición de un operando origen durante operaciones de tratamiento de cadenas.

DI: También se usa como registro índice en determinados modos de direccionamiento indirecto. Además almacena un desplazamiento de dirección, la posición de un operando destino durante operaciones con cadenas.



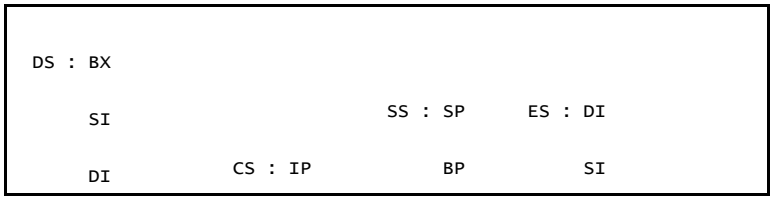
Las áreas de memoria asignadas al código de programa, datos y pila se direccionan por separado a pesar de poder solaparse. En un momento dado hay siempre 4 bloques disponibles de memoria direccionable denominadas segmento. Cada uno de los segmento suele tener una longitud de 64k.

Los registros de segmento CS, DS, SS y ES se usan para apuntar a las bases de los 4 segmento de memoria direccionbles:

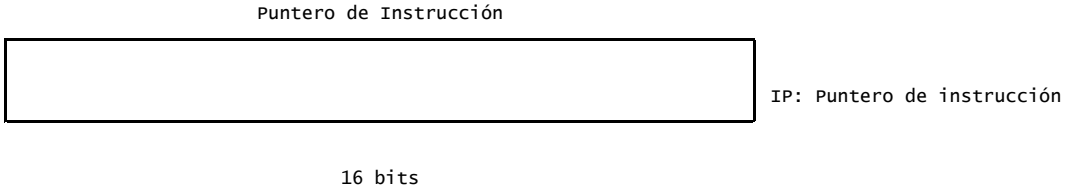
* **El segmento de código.**
* **El segmento de datos.**
* **El segmento de pila.**
* **El segmento extra.**

Para determinar una direcciones en el segmento de código tendremos que realizar el desplazamiento de 4 bits hacia la izquierda del registro CS poniendo a 0 los bits 0, 1, 2 y 3. Lo que equivale a multiplicar CS por 16. Sumando a continuación el valor de 16 bits almacenado en IP. La dirección dentro de los otro 3 registros se calcula similarmente.

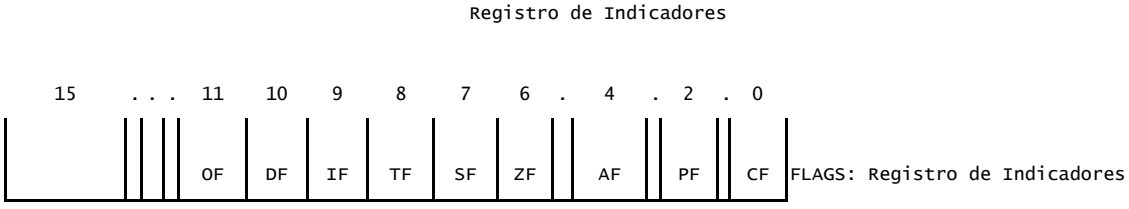
Las combinaciones de registro de segmento y desplazamiento depende de los tipos de operaciones que se esté ejecutando. Por omisión se asume que la dirección de un operando está en el segmento de datos y el registro de segmento a usar es por tanto DS con el desplazamiento BX, SI o DI.



Si el desplazamiento está almacenado en un registro puntero como SP o BP se asume que el operando está en el segmento de pila y, por tanto, el registro de segmento de pila SS se usa como base. Si la dirección del operando es el destino de una instrucción de cadena, el registro del segmento Extra ES constituye la base y el desplazamiento se almacena en DI o SI.



IP se usa para localizar la posición de la próxima instrucción a ejecutar dentro del segmento de código en curso. Como el registro CS contiene la dirección base del segmento de código, cualquier dirección de 20 bits dentro del segmento se localizará empleando cualquier IP como desplazamiento desde CS.



Los bits 0, 2, 4, 6, 7 y 11 son indicadores de condición que reflejan los resultados de operaciones del programa. Los bits del 8 al 10 son indicadores de control. Los indicadores de condición pueden comprobarse tras ejecutar determinadas operaciones usando el resultado de la comprobación en la toma de decisiones de vifurcación condicional.

Indicadores de Condición:

Bit 0. Indicador de acarreo (CF) → Se pone a 1 si en una operación de suma o resta se produce un acarreo por exceso o por defecto. Si una operación no produce acarreo estará a 0.

Bit 2. Indicador de paridad (PF) → Se pone a 1 si el resultado de una operación tiene un número par de bits a 1. Y se pone a 0 cuando el resultado tiene un número impar de bits a 1.

Bit 4. Indicador auxiliar de acarreo (AF) → Funciona igual que el anterior, pero se usa para señalar un acarreo por exceso o defecto de los 4 bits menos significativos en los valores de BCD (decimal codificado en binario).

Bit 6. Indicador de cero (ZF) → Se pone a 1 si el resultado de una operación es 0, esto ocurre, por ejemplo, después de usar una instrucción de resta o decremento o al hacer una comparación entre 2 número de igual valor. Para resultados distintos de 0 el indicador estará a 0.

Bit 7. Indicador de signo (SF) → Indica si un número es positivo o negativo en los términos de las aritméticas de complemento a 2. Se usa el bits más significativo de cualquier número en complemento 2 para indicar si dicho número es positivo cuando está a 0 o negativo cuando está a 1. Y se copia en el bit 7 del registro de indicadores.

Bit 11. Indicador de desbordamiento (OF) → Cualquier resultado que exceda los límites del tamaño de un operando provoca un desbordamiento (overflow) y activará este indicador a 1.

Registro indicador de Control:

Bit 8. Indicador de intercepción (TF) → Se pone a 1 para indicar que el modo de

intercepción (TRAP) está activado, haciendo que el micro-procesador ejecute la instrucción paso a paso. El procesador genera instrucción una detrás de otra. Un DEBUGGING puede usar este rango para procesar un programa instrucción a instrucción.

Bit 9. Indicador de interrupción (IF) → Se pone a 0 para desactivar la interrupción externa y a 1 para activarla. IF se controla con la instrucción CLI (desactiva interrupción externa) y STI (activa interrupción externa).

Bit 10. Indicador de dirección (DF) → Señala la dirección hacia la que se procesa la instrucción de cadena en relación con SI y DI. Se pone a 0 para la cadena que se procesa hacia arriba, o sea, hacia direcciones de memoria más altas y se pone a 1 para las cadenas que se procesan hacia abajo, o sea, hacia direcciones más bajas.

Solo para 80386 / 486.

Los procesadores de este tipo usan registros de 8 y 16 bits igual que el resto de la familia de los 8086. Todos los registros se extienden a 32 bits excepto los registros de segmento de 16 bits. Los registros extendidos comienzan con la letra E: el registro extendido de AX es EAX.

Los procesadores 386 / 486 tienen 2 registros de segmento adicionales: FS y GS.

Registro de Datos Extendidos

L → Low (Baja)

H → High (Alta)

H	L	
	AH AX AL	EAX: Acumulador
	BH BX BL	EBX: Base
	CH CX CL	ECX: Contador
	DH DX DL	EDX: Extra

8 b. 8 b.

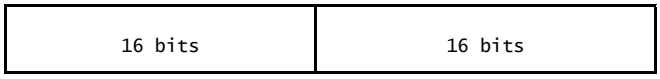
16 bits 16 bits Total 32 bits

Registro Puntero e Indice Extendido

L → Low (Baja)

H → High (Alta)

	SP	ESP: Puntero de Pila
	BP	EBP: Puntero de Base
	SI	ESI: Indice de Orden
	DI	EDI: Indice de Destino
H	L	Total 32 bits

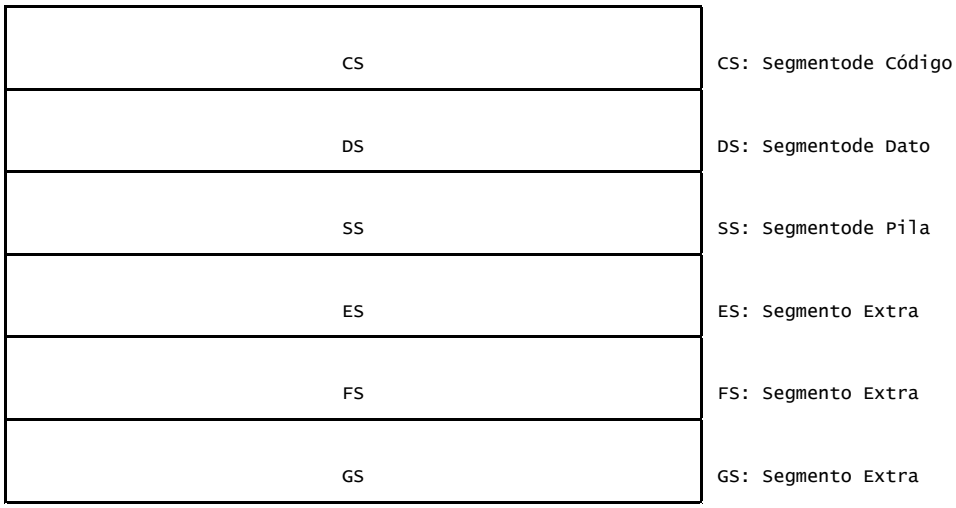


Puntero de Instrucción

Extendido

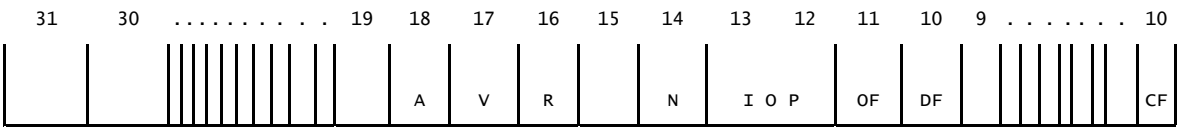
16 bits 16 bits Total 32 bits

Registro de Segmentos Extendidos



16 bits

Puntero de Bandera Extendida (EFlags)



<------- 386/486 ---- └--- Para todo los

--> <--- 286/386/486 ----> procesadores 8086 ->

IOP indica el nivel de protección para operaciones de Entrada/Salida.

El bit 14 (N) se usa en relación con procesos anidados.

- R → para reanudación de procesos.
- V → está relacionado con el modo 8086 virtual.
- A → está relacionado con el control de alineación.

Sentencias.

Una sentencia (o instrucción) en ensamblador puede tener la estructura siguiente:

[Nombre] [Operación] [Operandos] [;Comentario]

El **nombre** normalmente es una etiqueta. La **operación** indica la acción que se va a realizar con los operandos, que son la lista de uno o más items con los que la instrucción o directiva opera.

Ej:
```
principio: MOV ax, 7    ; movemos el valor 7 al reg ax
```

Una lista lógica puede contener como máximo 512 caracteres y ocupa 1 o más líneas física. Extender una línea lógica en 2 o más líneas física se realiza poniendo el carácter '\' como el último carácter, que no es un espacio en blanco, antes del comentario o fin de línea. Puede ponerse un comentario después de '\'.

Ej:
```
.if (x>0)              \; x debe ser positivo
    && (ax<0)          \; ax debe ser negativo
    && (cx=0)           ; cx debe ser cero
   MOV dx,20h
 .endif
```

Ej:
```
coment^          En este ejemplo se asignará este texto
    MOVE ax,0    y este código al comentario hasta que aparezca
                 el símbolo:
    ^ MOVE cx,1
```

Movimientos de datos.

En un micro-procesador 8086, el juego de registros es muy reducido. Evidentemente esto no es suficiente para llevar en registros todas las variables del programa Además, algunos de estos registros presentan peculiaridades (Ej: el registro de bandera) que los hacen inservibles para almacenar valores. Por todo esto, las variables de un programa se almacenan en memoria y se traen al registro sólo cuando son necesarias para algún calculo. Después, si es necesario actualizar su valor se vuelve a llevar a memoria el contenido del registro que proceda.

Algo ya mencionado es que muchos de los registros desempeñan papeles concretos en algunas instrucciones

Ejemplo: para un bucle, el contador se suele llevar casi siempre en el registro CX. O para una multiplicación es necesario cargar uno de los factores en AX).

Como ya vimos, para todas las operaciones de movimientos de datos existe una instrucción de lenguaje ensamblador denominada MOV, donde los operandos son 2, escritos separados por comas: el primero es el destino y el segundo el origen (Ej: MOV ax, bx).

Evidentemente, los operandos deben de tener la misma longitud (= número de bits). Pueden ser registros, posiciones de memoria, etc. Hacemos aquí un pequeño inciso para anotar que tradicionalmente los ensambladores no distinguen entre minúsculas y mayúsculas.

Tanto las instrucciones como los identificadores podemos escribirlos como queramos.

```
MOV ax,1
mov BX,2
MOV CX,3
mov dx,4
```

Estas instrucciones cargan AX con 1, BX con 2, ...

El dato que se carga en el registro va incluido en la propia instrucción siguiendo al código de operación que indica que se trata de una instrucción de transferencia y a que referencia hay que transferir.

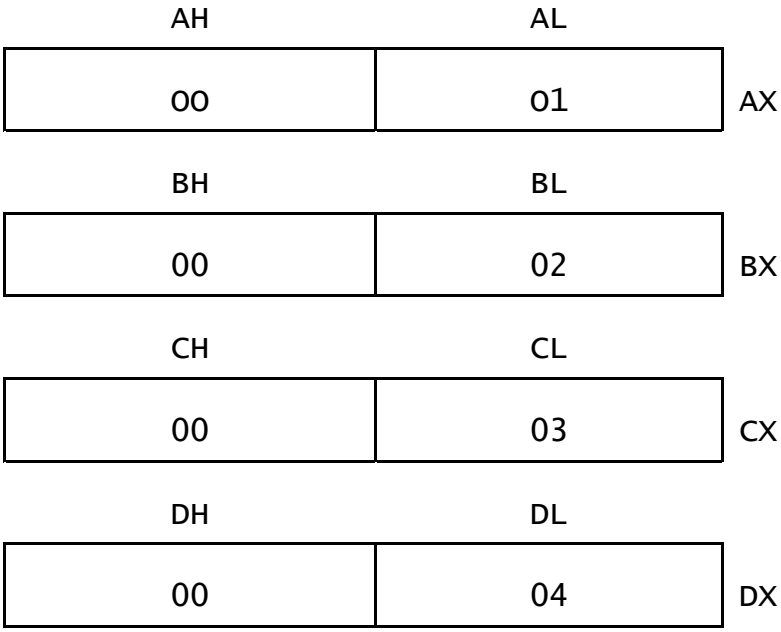
Así, los código de las 4 instrucciones en hexadecimal son:

	Código Operación
MOV ax,1	BB 0100
MOV bx,2	BB 0200
MOV cx,3	B9 0300
MOV dx,4	BA 0400

16 bits

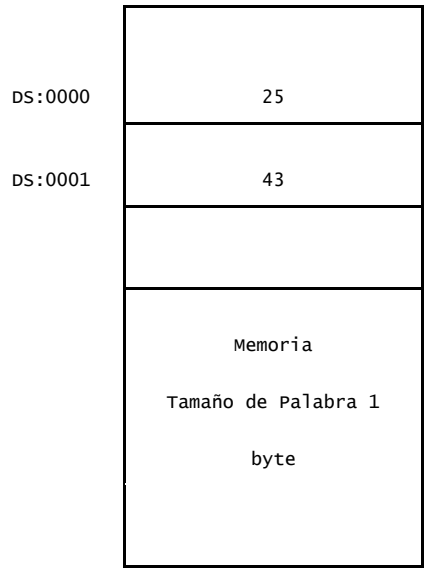
La estructura de las 4 instrucciones es igual. De principio el primer byte es el código de operación que identifica la instrucción que se está tratando. Ya que con este código de operación es necesario un valor de este tamaño (16 bits) para completar la instrucción, como podemos apreciar en los otros 2 bytes.

Estos bytes constituyen un valor de 16 bits almacenados en el orden INTEL, es decir, el bit menos significativo el primero y el más significativo después.

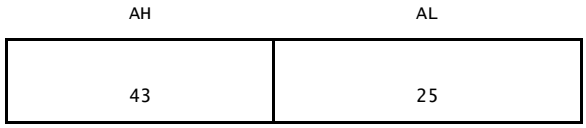


Cuando queremos especificar al ensamblador que un numero debe ser interpretado como una dirección de memoria a la que acceder escribimos el número entre corchetes. Así la siguiente instrucción carga en AX la palabra almacenada en la dirección DS:0000. Es decir, carga en AL el byte contenido en DS:0000 y en AH el contenido en DS:0001

```
MOV ax, [0]
```



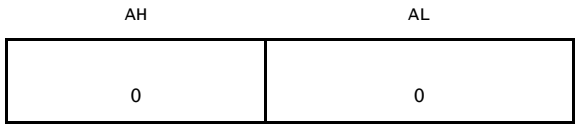
El valor de AX sería:



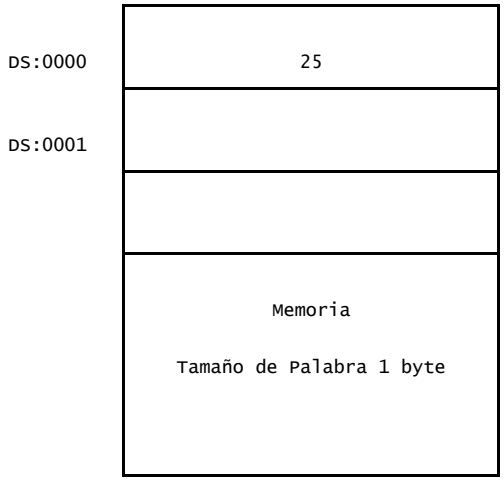
En cambio la siguiente instrucción lo que hace es almacenar el valor 0 en el registro AX.

```
MOV ax, 0
```

Es importante comprender la diferencia entre las 2 instrucciones. La primera lleva corchetes, se accede a la memoria y el valor recogido se almacena en AX. En la segunda se almacena en AX el valor 0 directamente sin acceder a memoria. Por lo tanto el valor de AX quedaría:



También podemos escribir en una posición de memoria el contenido de un registro. Con *MOV [0], al* almacenamos en el contenido de AL en DS:0000 Si AL es 25...



Aunque así podemos acceder a cual quier posición de memoria, en el segmento apuntado por DS, a menudo, nos interesa acceder a la posición de memoria indicada por el contenido de un registro. Por omisión se asume que la dirección de un operando está en el segmento de datos y el registro de segmento a emplear es DS. Con el desplazamiento almacenado en BX, SI o DI.

Las instrucciones que hemos visto no nos sirven porque la dirección a la que accede va incluida en los código de instrucción. Para ello existe un formato de la instrucción MOV que nos permite acceder a la posición de memoria indicada por BX. Así si BX contiene el valor 55AAh se accederá a esta posición de memoria mientras que si BX contiene el valor 1000h se accederá a la posición 1000h. Para esto escribimos entre corchetes BX el lugar de dar directamente el número.

```
MOV bx, 2345h
MOV ax, [bx]
```

Con esto almacenamos en AX el contenido de la dirección 2345 y 2346 ya que AX es una palabra. Si fuera AL o AH solo se almacenaría 2345. Por tanto, en AH almacena 2345h y en AL 2346h.

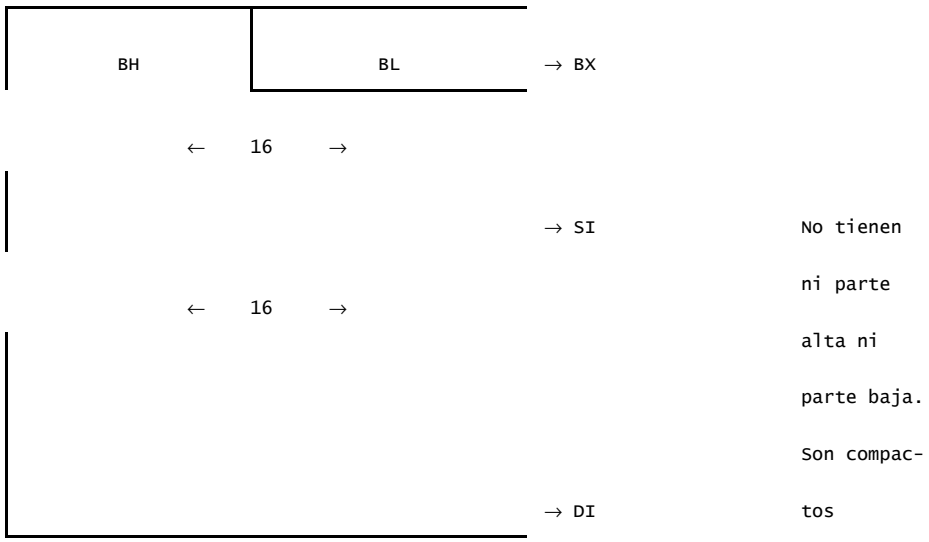
Por supuesto podemos acceder a memoria usando el registro BX para la escritura:

```
MOV bx, 2345h
MOV [bx], al
```

En este caso, solo se modifica el contenido de la dirección 2345h y no el de la 2346h, ya que estamos escribiendo el contenido de un registros de 8 bits.

Hemos usado en los ejemplos los registros de datos BX, esto se debe a que el juego de instrucción de los 8086 permiten únicamente acceder a la dirección contenida en el registro de datos BX. No a la contenida en el registros de datos AX, CX o DX. Ya que solo disponer de un solo registro para acceder a memoria resulta incómodo los 8086 incorporan otros 2 registros índices: SI y DI.

Estos dos registros permiten todas las instrucciones vistas para BX, tanto para transferencia entre registros como para transferencia entre procesador y memoria. Pero no hay ninguna forma de acceder por separado a los dos bytes que lo componen.



Otro registro que se puede usar como BX, SI y DI es el registro de puntero de base BP. Al igual que en SI y Di no se puede acceder por separado a los 2 bytes de mayor y menor peso. Pero la mayor diferencia es que por defecto se usa el registro de segmento SS para completar la dirección de 20 bits.

DS : BX SS : BP

SI

DI

Movimientos más generales realizados con MOV:

Formato de Instrucción		Ejemplo
MOV reg , reg	→ → →	MOV aex, ebx
MOV mem , reg	→ → →	MOV [bx], al
MOV reg , mem	→ → →	MOV ch, [40ffh]
MOV mem , inmediato	→ → →	MOV byte ptr [di], 25*80
MOV reg , inmediato	→ → →	MOV ebx, offffh
MOV segmentoreg , reg 16	→ → →	MOV ds, ax
MOV reg 16 , segmentoreg	→ → →	MOV ax, es
MOV mem , segmentoreg	→ → →	MOV [si], es
MOV segmentoreg , mem	→ → →	MOV ss, [1234h]

Hay varios puntos que debemos explicar de la tabla:

Reg: representa cualquier registro de 8, 16 o 32 bits pero con la restricción de que cuando la operación es MOV reg, reg ambos deben ser iguales.

Mem: se refiere a posiciones de memoria.

Inmediato: especifica que se debe tomar un valor incluido en los código de la propia instrucción como en el caso de MOV ax, 5 en donde 5 se denomina valor inmediato.

Segmentoreg: puede ser cualquier registro de segmento.

Reg 16: indica cualquier registro de 16 bits.

La exigencia en algunos formatos de registros de 16 bits se debe a que los registros de segmento normales y extendidos son de 16 bits → todas las operaciones de movimiento con estos registros se hacen con registros de 16 bits. Como vemos no es posible copiar directamente el contenido de un registro de segmento a otro. O cargar un registro de segmento con un valor inmediato, sino que es necesario pasar por un registro intermedio.

MOV es, ds	→	MOV ax, ds MOV es, ax
MOV es, 5	→	MOV ax, 5 MOV ds, ax

Especificar el registro CS como destino de la instrucción MOV está **prohibido**, ya que haría que la ejecución saltase a otro punto. Esto solo se permite a la instrucción de salto. La mayoría de los 8086 paran la ejecución del programa al encontrarla.

Uno de los formatos MOV interesantes es:

MOV mem, inmediato.

Permite introducir directamente en memoria un valor sin alterar ningún registro. Pero esto presenta un inconveniente, en los demás formatos siempre hay un registro como destino u origen de forma que el ensamblador deduce el tamaño del valor a mover. Así en las instrucciones.

```
MOV ax, [4220h]
```

```
MOV al, [4220h]
```

La primera instrucción genera un acceso a memoria de 16 bits que se descarga en un registro de ese tamaño. La segunda genera una lectura de 8 bits ya que Al solo tiene esta longitud.

```
MOV [4220h], 0
```

Nada indica al compilador si el 0 es de 1 o de 2 bytes. Hay dos posibles instrucciones máquina para esta línea de lenguaje ensamblador:

1ª Una instrucción de escritura de 16 bits poniendo a 0 los bits de las direcciones 4220h y 4221h.

2ª Una instrucción de escritura de 8 bits poniendo a 0 solo los bits de la dirección 4220h.

Por ello se hace necesario indicar al ensamblador de cual de las dos se trata.

La sintaxis para esto se comprende mejor así: El número 4220h es análogo a un puntero ya que es un valor que se interpreta como direcciones de memoria usándose en realidad como el desplazamiento que completa la dirección con DS. Este puntero puede ser de dos tipos:

- Puntero a 1 byte → se debe generar una instrucción de escritura de 8 bits.

- Puntero a 1 palabra → se debe generar una instrucción de escritura de 16 bits.

Así para aclarar esto se usa:

```
MOV  byte ptr [4220h], 0        ; pone a 0 solo el byte de la dirección 4220h

MOV word ptr [4220h], 0         ; pone a 0 los bytes de las direcciones 4220h y 4221h
```

Hemos visto que podemos acceder a la memoria componiendo las direcciones de diferentes formas, especificándolas directamente, en cuyo caso, el valor se incluye en la codificación de la instrucción por medio de los registros BX, SI, DI o BP. Pero en realidad hay muchas formas de indicar como se deben componer el desplazamiento.

Estas maneras de obtener la dirección se denominan Modos de Direccionamiento y cada uno tiene asignado un registro de segmento por defecto.

Modos de Direccionamiento

Denominación	Obtención Desplazamiento	Segmento	Ejemplo
Direccionamiento Absoluto	Inmediato	DS	MOV al, 0
Direccionamiento Indirecto con Base (BX)	BX+XX	DS	MOV cx, [bx+2]
Direccionamiento Indirecto con Indice	SI+XX	DS	MOV dx, [si+2]

(SI)			
Direccionamiento Indirecto con Indice (DI)	DI+XX	DS	MOV [di], es
Direccionamiento Indirecto con base (BP)	BP+XX	SS	MOV ax, [bp+4]
Direccionamiento Indirecto con Base (BX) e Indice (SI)	BX+SI+XX	DS	MOV [bx+si-2], cx
Direccionamiento Indirecto con Base (BX) e Indice (DI)	BX+DI+XX	DS	MOV dx, [bx+di]
Direccionamiento Indirecto con Base (BP) e Indice (SI)	BP+SI+XX	SS	MOV ds, [bp+si]
Direccionamiento Indirecto con Base (BP) e Indice (DI)	BP+DI+XX	SS	MOV ax, [bp+di]

Puede apreciarse que todos los modos admiten que después de obtener el valor de desplazamiento de un registro o de la suma de dos, se añada una constante a éste, antes de generar la lectura, indicada por XX. Si se omite el valor XX se generan instrucciones que no añaden ningún valor al desplazamiento obtenido por los registros.

Hemos visto que cada modo usa un registro de segmento por defecto para componer la dirección completa pero siempre es posible que le CPU use, no el registro de segmento por defecto, sino uno espe-

cificado por nosotros.

A nivel de lenguaje máquina lo que se hace es añadir a la instrucción un prefijo de 1 byte que modifica el comportamiento de la siguiente instrucción haciendo que use el registro de segmento correspondiente al prefijo en lugar del registro por defecto.

Hay 4 prefijos distintos, uno para cada registro de segmento y se denominan Prefijos de Segmento por convenio se añade el prefijo inmediato antes de la apertura de corchetes o después de PTR, si este aparece. Así las siguientes instrucciones acceden a posición de memoria dentro del segmento de código de datos extra y segmento de pila, respectivamente.

```
MOV al, cs:[bx+4]
MOV word ptr ds:[bp-0fh], 0
MOV es:[di], al
MOV word ptr ss:[bx], 0
```

Es interesante tener en cuenta que por ejemplo:

MOV [bx], ax y **MOV ds:[bx], ax** son equivalentes. Ya que si no se indica nada el registro por defecto es DS. También es interesante resaltar que para acceder a una posición de un vector, que haya sido previamente definido en el segmento de datos, usamos los corchetes.

Segmento de datos...

```
tabla word 100 dup (0) ; Vector de 100 posición, cada posición sería de 1 palabra con valor inicial 0.
```

Segmento de código...

```
MOV ax, tabla[si]      ; Donde SI actúa como índice del vector.
MOV bx, 0          ; Esto sería equivalente
```

```
  MOV ax, tabla[5]      ; todas las sentencias pasarían al regis-
tro AX
  MOV ax, tabla+5       ; el valor de la posición 5 de la tabla
pasa AX.
  MOV ax, tabla[bx]+5
  MOV ax, tabla[bx][di]      ; Todas estas instrucciones también
son equivalentes
  MOV ax, tabla[di][bx]      ; todas las sentencias mueven la
posición
  MOV ax, tabla[bx+di]       ; indicada por BX+DI dentro de la
tabla a AX.
  MOV ax, [tabla+bx+di]
  MOV ax, [bx][di]+tabla
```

Constantes.

Una constante entera es una serie de uno o más números seguidos de una base opcional especificada por una letra.

```
MOV ax, 25    MOV ax, 0b3h.
```

Los números 25 y 0b3 son constantes enteras. La h indica la base hexadecimal. Los distintos tipos de base...

"b" o "y"	Binario: 'b' si la base es menor o igual que 10
"o" o "q"	Octal
"d" o "t"	Decimal: 'd' si la base es menor o igual que 10
"h"	Hexadecimal

Pueden indicarse en mayúsculas o minúsculas. Si no se indica, el ensamblador la interpreta directamente con la base actual. Por defecto es la decimal, pero puede cambiarse con la sentencia RADIX...

```
radix 16  ; base=16 (hexadecimal)
 db 11    ; se interpreta como valor 11h=17
 db 11t   ; se interpreta como valor 11 (en decimal) =11
 db 01at  ; genera un error, por que a no es un dígito decimal

radix 2   ; base=2 (binaria)
 db 11    ; se interpreta como el valor 11 (binario) =3

radix 10  ; base=10 (decimal)
 db11     ; se interpreta como el valor 11d=11
```

Los números hexadecimales siempre deben comenzar con un dígito del 0 al 9. Si no es así y comienzan por alguna letra hay que añadir un 0 al principio para distinguir entre símbolo hexadecimal y etiqueta.

abch V.S. 0abch ⇒ abch es interpretado como una etiqueta y 0abch como un número hexadecimal

Los dígitos hexadecimales desde A a F pueden ser mayúsculas y minúsculas. Se pueden definir constantes enteras simbólicas por algunas de las siguientes asignaciones de datos o con EQU (o con el signo "=").

EQU.- Tiene como función la de asignar un nombre simbólico a valor de una expresión.

El formato es: ***nombre EQU expresión***. La utilidad de esta directiva reside en hacer más clara y legible las sentencias de un programa fuente realizado en ensamblador. Al contrario que en la directiva "=", el nombre no puede redefinirse, es decir permanece invariable la expresión asociada a lo largo de todo el programa fuente (al intentar redefinir crea un error).

"Expresión" puede ser una constante numérica, una referencia de direcciones, cualquier combinación de símbolos y operaciones que pueda evaluarse como un valor numérico u otro nombre simbólico. Si un valor constante, o sea un nombre, usado en numerosos lugares del código fuente necesita ser cambiado se debe modificar la expresión en un lugar solo, en vez de por todas las partes del código fuente.

Ejemplo:

```
col EQU 80   ; columna de una pantalla (col=80)
```

```
fil EQU 25   ; fila de una pantalla (fil=25)
pantalla EQU col*fil  ; tamaño de una pantalla (2000)
línea EQU fil
longitud dw 0   ; variable tipo palabra
byte EQU ptr longitud ; byte=primer byte de longitud
cr EQU 13  ; retorno de carro
cf EQU 10  ; principio de línea
```

Por defecto, el tamaño de palabra para expresiones de **MASM 6.0** es de 32 bits. Se puede modificar con:

- OPTION EXPRE 32 ; es erróneo cambiar el tamaño de palabra una vez ha sido fijada

- OPTION EXPRE 16 ; la dos últimas usan el tamaño fijo de palabra de 16 bits

- OPTION M510 ; → opción para operaciones muy específicas del ensamblador

Operadores.

Un operador es un modificador que se usa en l campo de operandos de una sentencia en ensamblador. Se puede usar varios operadores con instrucciones del procesador. `MOV ax, [bx+2]` La palabra reservada "MOV" es una instrucción y el signo "+" es un operador. Entre los tipos de operadores hay:

- Operador Aritmético, opera sobre valores numéricos.

- Operador Lógico, opera sobre valores binarios bit a bit.

- Operador Relacional, compara 2 valores numéricos o 2 direcciones de memoria del mismo segmento y produce como resultado: 0 → si la relación es falsa, 0ffffh → si es verdadera.

- Operador de Atributos, permite redefinir el atributo de una variable o etiqueta. Los atributos para variables de memoria pueden ser:
  - byte (1 byte)
  - sbyte (byte con signo)
  - word (palabra)
  - sword (palabra con signo)
  - dword (doble palabra)
  - sdword (doble palabra con signo)
  - fword (6 bytes)
  - qword (8 bytes)
  - tbyte (10 bytes)

Los atributos para etiquetas pueden ser:

near → cuando se puede referenciar desde dentro del segmento donde está definida la etiqueta.

far → cuando se puede referenciar desde fuera del segmento donde está definida la etiqueta.

El ensamblador evalúa expresiones que contienen más de un operando según las siguientes reglas:

1ª Siempre se ejecutan antes las operaciones entre paréntesis que las adyacentes.

2ª Se ejecutan primero las operaciones binarias de más prioridad.

3ª Las operaciones de igual prioridad se ejecutan de izquierda a derecha.

4ª Operaciones unarias de igual prioridad se ejecutan de derecha a izquierda.

El orden de prioridad de todos los operadores está en la tabla:

Prioridad	Operador
1	( ) [ ]
2	LENGHT SIZE WIDTH MASK
3	• (referencia un campo de un estructura)
4	: (usado en prefijos de segmento)
5	CROOFFSET OFFSET SEG THIS TYPE
6	HIGH HIGWORD LOW LOWWORD
7	+ - (en operaciones unarias)
8	* / MOD SHL SHR
9	+ - (en operaciones bianrias)
10	EQ EN LT LE GT GE

11	NOT
12	AND
13	OR XOR
14	OPATTR SHRT •TYPE

INSTRUCCIONES

Instrucciones Aritméticas: *ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC*

*** ADD:** Realiza la suma de dos operandos identificándolos como origen y destino, quedando el resultado de la suma en el operando destino (ADD destino, origen). Los dos operandos deben ser del mismo tipo.

Ejemplo:

```
ADD ax, bx
ADD si,  di
ADD [0], ax
ADD ah, [bx]
ADD byte ptr[di-2], 2
MOV ebx, 43981
ADD eax, ebx
```

*** ADC:** Se utiliza para realizar una suma de 32 bits con registros de 16 bits. La suma se realizaría igual, pero operando sobre 32 bits. En realidad, podemos descomponer una suma de 32 bits en dos sumas de 16 bits. Primero, deberíamos sumar los 16 bits de menor peso de cada operando, almacenando los 16 bits resultantes como palabras baja del resultado. Después, deberíamos sumar los 16 bits de mayor peso de cada operando, pero además deberíamos sumar el acarreo del último bit de la palabras de menor peso. Así, llevamos efectivamente el acarreo del bit 15 de los operandos al bit 16 (esto se realiza utilizando el flag de acarreo CF).

Así, en el siguiente ejemplo, se le sumaría a el valor de 32 bits contenido en DXAX, el valor de 32 bits contenido en

CXBX.

```
                    cx    bx
                        +
                    dx    ax
                    dx    ax
```

ax=ax+bx

si cf=1 (si hay acarreo), dx=dx+cx+1

sino dx=dx+cx

Ejemplo:

```
ADD  ax, bx  ; Suma los 16 bits más bajos, dejando el acarreo(si se produce) en CF
             ; (Flag de Acarreo) preparado para sumárselo a los 16 bits más altos.
ADC dx, cx ; Suma los 16 bits más alto, y a ese resultado se le suma el acarreo (si
   existe) producido por la suma de bx + ax. El resultado de la suma queda en  ax
   (la parte baja) y en dx (la parte alta).
```

Nota: La utilización de el Flag de acarreo (CF) puede producir a veces algún problema. Si fuese así, una solución sería que, después de ejecutar alguna operación que utilice este flag, ejecutar la instrucción CLC (Borra la bandera de acarreo (CF) sin afectar a ninguna otra bandera).

* **SUB:** Realiza la operación de resta de un operando origen sobre un operando destino. El resultado queda almacenado en el operando destino...

Formato SUB destino, origen; (destino = destino - origen).

Ejemplo:

```
SUB cl, dl   ; En Cl se almacena el valor resultante de CL - DL.
SUB es:[bx], dx  ; En la posición de memoria indicada, se almacena el valor
```

```
                ; resultante de restarle a la dirección indicada DX.
SUB al, [bp+4]
```

*** SBB:** Se utiliza para realizar una resta de 32 bits utilizando registros de 16 bits. El formato es el mismo que con ACD; debemos, descomponer los operandos en fragmentos de 16 bits y comenzar a restar por la derecha. En cada resta después de la primera, debemos calcular la resta y del resultado restar el contenido del flag de acarreo (en las restas el acarreo se representa mediante acarreo negativo). Así, en el siguiente ejemplo se le restaría a un valor de 32 bits compuesto por DXAX, un valor de 32 bits compuesto por CXBX.

```
                    dx    ax
                          -
                    cx    bx
                    dx    ax

          ax=ax-bx
          si CF=-1 (si hay acarreo) entonces DX=DX-CX-1
          sino DX=DX-CX
```

Ejemplo:

```
SUB ax, bx ; Resta los 16 bits más bajos, deja el acarreo preparado para restar
           ; los 16 bits más alto.
SBB dx, cx ; Resta los 16 bits más altos y además resta el acarreo dejado por
           ; la anterior instrucción.
           ; Resultado de 32 bits en DX(palabras alta) y AX (palabras baja).
```

*** MUL:** Formato MUL fuente.

Multiplica, sin considerar el signo, el acumulador (Al si el operando fuente es un byte, AX si el operando fuente es un número de 16 bits o EAX si el operando fuente es un número

de 32 bits) por el operando fuente. Si el operando fuente es de tipo byte, el resultado se almacena en AX. Si el operando fuente es de tipo palabra, el resultado se almacena en AX (palabra inferior) y DX (palabra superior).

Si el operando fuente es de tipo doble palabra, el resultado se almacena en EAX (doble palabra inferior) y EDX (doble palabra superior).

Si la mitad superior del resultado (AH para el caso de operando tipo byte, DX para el caso operando tipo palabra o EDX para el caso de operando tipo doble palabra) no es cero, se activan las banderas CF y OF, indicando que esta mitad superior contiene dígitos significativos del resultado.

(fuente tipo byte) * AL = AHAL (AX) (AH parte más alta y AL parte más baja)

(fuente tipo palabra) * AX = DXAX (DX parte más alta y AX parte más baja)

(fuente tipo doble palabra) * EAX = EDXEAX (EDX parte más alta y EAX parte más baja)

Ejemplo:

```
MOV al, 10 ; Movemos a AL 10.
MOV bl, 12 ; Movemos a BL 12.
MUL bl  ; Fuente de tipo byte; el resultado queda en AX (AX=AL*BL).
         ;La parte más alta queda en AH y la más baja en AL
MOV ax, 20
MOV bx, 5
MUL bx; Fuente de tipo palabra.
        ; El resultado queda en DXAX (DXAX=AX*BX).
        ; La parte alta queda en DX y la más baja en AX.
MOV eax, 5
MOV ebx, 2
MUL ebx ; Fuente de tipo doble palabra; el resultado queda en
```

```
        ; EDXEAX (EDXEAX=EAX*EBX). La parte alta queda en EDX
        ; y la más baja en EAX
```

* **IMUL:** Realiza la misma operación que MUL, solo que en este caso se contempla el signo.

* **DIV:** Formato DIV fuente.

  Divide, sin considerar el signo, el acumulador (AX (dividendo) si el operando fuente (divisor) es un byte, DXAX (dividendo) si el operando fuente (divisor) es un número de 16 bits o EDXEAX (dividendo) si el operando fuente (divisor) es un número de 32 bits) por el operando fuente.

  Si el operando fuente es de tipo byte el cociente se almacena el AL y el resto se almacena en AH.
  AX / fuente = AL (resto en AH)

  Si el operando fuente es de tipo palabra, el cociente se almacena en AX y el resto se almacena en DX.
  DXAX / fuente = AX (Resto en DX)

  Si el operando fuente es de tipo doble palabra, el cociente se almacena en EAX y el resto en EDX.
  EDXEAX / fuente = EAX (Resto en EDX)

Ejemplo:

```
MOV ax, 12 ; Movemos a AX 12.
MOV bl, 10 ; Movemos a BL 10.
DIV bl  ; fuente de tipo byte.
        ; el cociente queda en AL (AL=1), y el resto queda en AH(AH=2).
MOV ax, es:[si]
```

```
MOV bx, es:[si+2]; Movemos en DXAX un valor almacenado en memoria,
                  ; por ejemplo, 123567.
MOV bx, 65000
DIV BX  ; Fuente de tipo palabra; el cociente queda en AX y el resto
        ; queda almacenado en DX.
MOV eax, es:[si]
MOV ebx, es:[si+4] ; Movemos a EDXEAX un valor almacenado en memoria.
MOV ebx, 50000
DIV ebx    ; Fuente de tipo doble palabra; el cociente queda en EAX y
           ; el resto queda almacenado en EDX.
```

* **IDIV**: Realiza la misma operación que DIV, sólo que en este caso se contempla el signo.

* **INC**: Se utiliza para incrementar el contenido de un registro o de una posición de memoria.

Ejemplo:

```
MOV ax, 5  ; a AX se le pasa el valor 5.
INC ax   ; AX incrementa en una unidad su valor (AX=6).
INC byte ptr[bp+si]      ; EL byte al que apunta la suma de "BP + SI" en la memoria
                         ; se ve incrementado en una unidad.
INC word ptr[bp+si]      ; Lo mismo pero para una palabra.
```

* **DEC**: Se utiliza para decrementar el contenido de un registro o de una posición de memoria.

Ejemplo:

```
MOV ax, 5  ; a AX se le pasa el valor 5
DEC ax   ; AX decrementado en una unidad su valor (AX=4)
```

```
DEC byte ptr[bp+si]      ; El byte al que apunta la suma de "BP+SI" en la memoria
                         ; se ve decrementado en una unidad.
DEC word ptr[bp+si]      ; Lo mismo pero para una palabra.
```

Instrucciones Lógicas: *NEG, NOT, AND, OR, XOR*

* **NEG**: Esta instrucción lo que hace es calcular el complemento a dos del operando, y almacenando en el mismo lugar. Esto es, efectivamente, equivalente a cambiar de signo el operando de la instrucción.

Ejemplo:

```
MOV ax, 5    ; a AX se le pasa el valor 5.
NEG ax       ; Se haya el complemento a 2 de AX y se guarda en AX (AX= -5).
NEG byte ptr es:[bx+si+2]; Se haya el complemento a 2 de la posición de memoria
                         ; (dentro del Segmento Extra) indicada por el de "BX+SI+2"
```

* **NOT**: Se realiza el NOT lógico del operando bit a bit. El NOT lógico bit a bit consiste en invertir cada bit del operando (pasar los 0 a 1 y los 1 a 0; 10100 -> 01011)

Ejemplo:

```
NOT si    ; El valor que tenga SI pasa los 0 a 1 y los 1 a 0.
NOT word ptr es:[0]      ; Lo mismo pero en una posición de memoria.
```

* **AND**: Operación "y lógico" a nivel de bit entre los dos operandos. El resultado se almacena en el destino.
  Formato AND destino, fuente.

```
0  0 - 0
0  1 - 0
1  0 - 0
1  1 - 1
```

Ejemplo:

```
AND ax, bx ; AND lógico entre AX y BX. El resultado queda en AX.
AND es:[0], dx  ; Lo mismo pero con posiciones de memoria.
AND di, es:[si]
AND byte ptr[9], 3 ; Lo mismo pero con valores inmediatos.
```

* **OR**: Operación "o lógico exclusivo" a nivel entre los dos operandos. El resultado se almacena en el destino.
  Formato OR destino, fuente.

```
0  0 - 0
0  1 - 1
1  0 - 1
1  1 - 1
```

Ejemplo:

```
OR al, ah  ; Las mismas operaciones que con AND pero utilizando el OR.
OR [di], ch
OR cl, [bp+4]
OR byte ptr es:[si], 1
```

* **XOR**:Operación "o lógico exclusivo" a nivel de bit entre los dos operandos. El resultado se almacena en destino.
  Formato XOR destino, fuente.

```
0  0 - 0
0  1 - 1
1  0 - 1
1  1 - 0
```

Ejemplo:

```
XOR ax, ax ; El XOR entre dos bits con el mismo valor es siempre 0,
             ; independientemente del valor previo de AX (AX=0).
        ; Las ventajas de hacerlo así son dos: la ejecución de XOR reg, reg es más
        ; rápida que la de MOV reg, o que la de MOV ax,0 , y la codificación de la ;
        primera ocupa menos bytes que la segunda; Esta técnica no puede utilizar  ;
        se para poner a cero los registros de segmento.
XOR byte ptr[55aah], 4
XOR al, 00aah
```

* **XCHG**: Intercambia el contenido entre dos operandos. No pueden utilizarse registros de segmento como operandos.

Ejemplo:

```
XCHG si, di    ; Si SI tiene valor 45 y DI tiene valor 68, ahora, DI se queda con
               ; valor 45 y SI con 68.
XCHG al, [bx+4]
XCHG ss:[si], bx
```

* **CMP**: Formato CMP destino, origen. (destino - origen) Esta instrucción realiza una resta de un operando origen sobre un operando destino, pero con la particularidad de no almacenar el resultado y no modificar ninguno de los 2 operandos, pero si se modifican los bits de indicadores (Flags). Los operandos deben ser del mismo tipo.

  Esta modificación de los bits de indicadores, nos permitirá posteriormente, mediante la inspección de los mismos, poder realizar determinadas acciones. Normalmente después de una instrucción de comparación (CMP), hay una instrucción de salto.

Ejemplo:

```
CMP ax, bx ; Comparamos AX con BX
JL menor  ; Si AX es menor que BX saltamos a la etiqueta MENOR
                        .
                        .
MENOR:
CMP bl, cl
CMP bx, cx
CMP bl, byte ptr es:[si]
CMP word ptr es[si], bx
CMP bx, 30
CMP byte ptr es:[si], 01h       ; Normalmente, después de cada instrucción de
                                ;comparación, viene una instrucción de salto.
```

Instrucciones de Salto.

Vimos que en el funcionamiento de un microprocesador se reduce básicamente a los siguientes pasos:

- Recogida de la siguiente instrucción de la dirección CS:IP
- Incremento de IP en el número de bytes que componen la instrucción.
- Ejecución de la instrucción.

Una introducción de salto se reduce a cambiar el contenido de IP y, eventualmente el de CS.

Principalmente, existen dos tipos de instrucciones de salto: aquellas que especifican la dirección de salto inmediato después del cód. de operación, es decir, especifican la etiqueta a la que hay que saltar (denominados saltos directos), y aquellas que especifican una dirección de memoria de la que hay que recoger la dirección a la que saltar (denominadas saltos indirectos).

Los bytes que componen una instrucción de salto directo incluyen en el cód. la operación algunos bytes que especifican la dirección a la que se debe producir el salto.

Pero existen varios formatos posibles para la instrucciones de salto directo. El primero se denomina *short jump* (salto corto), y el único dato que incluye la instrucción después del cód. de operación es un byte, que representa en complemento a 2 el valor a añadir a IP para seguir la ejecución. Este byte se suma a IP, para lo que primero es necesario extenderlo en signo (que el signo del primer byte ocupe el segundo byte) a 16 bits. Así, el byte representa un desplazamiento entre -128 y +127 bytes (256 bytes), que es el rango que se puede especificar con un bytes en complemento a 2.

Si observamos el orden en el que el microprocesador lleva a cabo la ejecución de una instrucción, veremos que el desplazamiento se suma a IP después de haber incrementado éste. Por tanto, el desplazamiento se toma desde la dirección de comienzo de la siguiente instrucción al salto, y no desde la propia instrucción de salto.

El siguiente formato de salto directo es el *near jump* o salto cercano. Este formato, la instrucción incluye dos bytes que forman la palabra a sumar a IP, también en complemento a 2. Así, el rango de salto está entre -32768 y +32768 bytes (65535 bytes), que efectivamente permiten un salto a cualquier punto del segmento donde reside la instrucción de salto (en este formato CS tampoco es alterado por el salto). El ensamblador comprueba si el salto está en el rango (-128, +127) para realizar un salto corto y si no lo está genera un salto cercano.

El último tipo de salto se denomina *far jump* o salto lejano. Esta denominación se debe a que éste formato de salto, cambia tanto CS como IP, pudiendo saltar a cualquier punto del megabyte direccionable (2 elevado a 20). En éste formato de salto, la instrucción lleva dos palabras con el desplazamiento y el segmento de la dirección a la que hay que saltar (se utiliza para realizar un salto a otro segmento). Este tipo de salto copia directamente en IP y CS los valores dados por la instrucción, sin tener en cuenta el contenido previo de ambos.

Existen dos formatos de instrucciones de indirecto: el primero, denominado near jump o salto cercano, lee una palabra de la dirección de memoria especificada y carga el registro IP con ésta. Así, se puede saltar a cualquier punto del segmento donde resida la instrucción de salto. El otro tipo se denomina far jump o salto lejano, y toma de la dirección especificada dos palabras, la primera de la cuales se introduce en IP, y la segunda en CS (Ya que el

ordenamiento INTEL siempre se almacenan primero los elementos de menor peso). De ésta forma se puede saltar a cualquier punto de la memoria direccionable con un salto indirecto.

* **JMP:** El formato de la instrucción es JMP dirección. Provoca un salto incondicional, por lo que se utiliza para seguir la ejecución del programa en otro punto, que puede ser especificando una etiqueta (salto directo) o especificando una dirección (salto indirecto). Cuando incluimos instrucciones de salto en el programa, indicamos la dirección del destino, y en caso de que el salto necesite especificar un valor a sumar a IP, el ensamblador se encarga de calcular el desplazamiento desde el punto donde se ejecuta el salto. En una instrucción JMP; el propio ensamblador decide si debe generar un salto corto o lejano: en el caso de que el destino esté en el rango de un byte con signo, se genera un salto corto, en caso contrario, se genera un salto cercano.

Ejemplo:

```
ETIQUETA1:
            .
            JMP continuar
ETIQUETA2:
            .
            JMP continuar
            .
CONTINUAR:
```

Nota: Para los siguiente saltos, vamos a tener en cuenta significados de palabras inglesas que nos van a ayudar a definir el tipo de salto a realizar:

(Equal=igual, Not=no, Greater=mayor, Less=menor, Above=superior, Below=inferior, Carry=acarreo, Zero=cero, Overflow=desbordamiento,

Sign=signo, Parity=paridad)

*** JA:** (Salto si superior). Es equivalente a JNBE (Salto si no inferior ni igual). El formato es: JA etiqueta si tanto el flag de acarreo CF como el flag de cero ZF está a cero (CF=0, ZF=0). Si CF=1 o ZF=1 no se transfiere el control. No se considera el signo.

Ejemplo:

```
        CMP ax, bx ; Comparar AX con BX.
        JA etiqueta      ; Saltar (Bifurcar) a ETIQUETA si AX>BX
        .  ; (sin considerar signo).
        .
ETIQUETA:
```

*** JAE:**(Salto si superior o igual). Es equivalente a JNB (Salto si no inferior). El formato es: JAE etiqueta. Salta a la etiqueta si el flag de acarreo es cero (CF=0). No se considera el signo.

Ejemplo:

```
                CMP ax, bx    ; Comparamos AX con BX.
                JAE etiqueta  ; Bifurca a ETIQUETA si AX> o =BX
                   .          ; (sin considerar el signo).
                   .
        ETIQUETA:
```

*** JB:** (Salto si inferior). Es equivalente a JNAE (Salto si no superior ni igual) y a JC (Salto sin acarreo). El formato es: JB etiqueta. Salta a la etiqueta si el flag de acarreo es uno (CF=1). No se considera el signo.

Ejemplo:

```
            CMP ax, bx
            JB etiqueta   ; Bifurca a ETIQUETA si AX < BX
               .          ; (sin considerar el signo).
               .
      ETIQUETA:
```

* **JBE:**(Salto si inferior o igual). Es equivalente a JNA (Salto si no superior). El formato es: JBE etiqueta. Salta a la etiqueta si el flag de acarreo es igual a 1 o el flag de cero es igual a uno (CF=1 y ZF=1). Si CF=0 y ZF=0 no hay salto. No se considera el signo.

Ejemplo:

```
            CMP ax, bx
            JBE etiqueta  ; Bifurca a ETIQUETA si AX es = o < que BX
               .          ; (sin considerar el signo).
               .
      ETIQUETA:
```

* **JE:** (Salto si igual). Es equivalente a JZ (Salto si cero). El formato es: JE etiqueta. Salta a la etiqueta si el flag de cero es igual a uno (ZF=1). Se considera número con signo y sin signo.

Ejemplo:

```
            CMP ax, bx    ; Comparo AX con BX.
            JE etiqueta1  ; Bifurca a ETIQUETA1 si AX = BX.

            CMP ax, bx    ; AX=AX-BX
            JZ etiqueta2  ; Bifurca a ETIQUETA2 si AX es cero.
```

* **JG:** (Salto si mayor). Es equivalente a JNLE (Salto si no menor ni igual). El formato es: JG etiqueta. Salta a la etiqueta si el flag de cero es igual a cero y el flag de desbordamiento contiene el mismo valor que el flag se signo (ZF=0 y SF=OF). Si ZF=1 o SF<>OF, no hay salto. Se considera el signo.

Ejemplo:

```
            CMP ax, bx
            JG etiqueta   ; Bifurca a ETIQUETA si AX > BX
               .          ; (considerando el signo).
               .
      ETIQUETA:
```

* **JGE:**(Salto si mayor o igual). Es equivalente a JNL (Salto si no menor). El formato es: JGE etiqueta. Salta a la etiqueta si el flag de desbordamiento contiene el mismo valor que el flag de signo (SF=OF). Se considera el signo.

Ejemplo:

```
            CMP ax, bx
            JGE etiqueta  ; Bifurca a ETIQUETA si AX es > o = BX
               .          ; (considerando el signo).
      ETIQUETA:
```

* **JLE:**(Salto si menor o igual). Es equivalente a JNG (Salto si no mayor). El formato es: JLE etiqueta. Salta a la etiqueta si el flag de cero está a uno o el flag de desbordamiento y el de signo contiene valores distintos (ZF=1 o SF distinto de OF). Si ZF=0 y SF=OF no se produce el salto. Se considera el signo.

Ejemplo:

```
            CMP ax, bx
            JLE etiqueta  ; Bifurca a ETIQUETA si AX es < o = BX
               .          ; (considerando el signo).
                .
      ETIQUETA:
```

* **JNA, JNAE, JNB, JNBE, JNE, JNG, JNGE, JNL, JNLE:**
  Estas instrucciones comprueban exactamente las condiciones opuestas a sus análogas sin la letra N. En realidad no sería necesaria, porque son sinónimas de JBE, JB, JAE, JNZ, JLE, JL, JGE Y JE, respectivamente. Pero el lenguaje ensamblador estándar las incluye para facilitar el trabajo del programador.

* **JO:** (Salto si desbordamiento). Formato es: JO etiqueta. Salta a la etiqueta si el flag de desbordamiento está a uno (OF=1).

Ejemplo:

```
            ADD ax, bx    ; AX=AX+BX
            JO etiqueta   ; Bifurca a ETIQUETA si hay desbordamiento
               .          ; (Overflow).
      ETIQUETA:
```

* **JNO:**(Salto si no desbordamiento). El formato es: JNO etiqueta. Salta a la etiqueta si el flag de desbordamiento está a cero (OF=0).

Ejemplo:

```
                ADD al, bl     ; AL=AL+BL
                JNO etiqueta   ; Bifurca a ETIQUETA si no hay desbordamiento
                  .      ; (No overflow).
                  .
          ETIQUETA:
```

* **JS:** (Salto si signo). El formato es: JS etiqueta. Salta a la etiqueta si el flag de signo está a uno (SF=1).

Ejemplo:

```
                SUB ax, bx     ; AX=AX-BX
                JS etiqueta    ; Bifurca a ETIQUETA si signo, es decir, AX < 0
                  .            ; (en este caso, si AX es menor que BX).
                  .
          ETIQUETA:
```

* **JNS:**(Salto si no signo / si el signo en positivo). El formato es: *JNS etiqueta*. Salta a la etiqueta si el flag de signo está a cero (SF=0).

Ejemplo:

```
    SUB ax, bx ; AX=AX-BX
    JNS etiqueta ; Bifurca a ETIQUETA si no signo, es decir, AX > o = que BX
                  .            ; (en este caso, si AX es mayor o igual que BX).
                  .
ETIQUETA:
```

*** JP:** (Salto si paridad). Es equivalente a JPE (salto sin paridad par). El formato es: JP etiqueta. Salta a la etiqueta si el flag de paridad está a uno (PF=1).

Ejemplo:

```
            AND ax, bx     ; AX=AX AND BX
            JP etiqueta    ; Bifurca a ETIQUETA si paridad par, es decir
              .            ; si el número de "unos (1)" que hay en AX es par.
              .
      ETIQUETA:
```

*** JNP:** (Salto si no paridad). Es equivalente a JPO (salto sin paridad impar). El formato es: JNP etiqueta. Salta a la etiqueta si el flag de paridad está a cero PF=0).

Ejemplo:

```
       AND ax, bx     ; AX=AX AND BX
       JNP etiqueta   ; Bifurca a ETIQUETA si paridad impar, es decir
          .   ; si el número de "unos (1)" que hay en AX es impar.
          .
  ETIQUETA:
```

*** LOOP:** Esta instrucción permite realizar "bucles" utilizando el registro CX como contador (CX en un contador que va decrementándose). Un bucle es un conjunto de instrucciones que se ejecutan una serie de veces. Esta instrucción equivale al par: DEC CX // JNZ etiqueta.
El formato es: LOOP etiqueta.

Ejemplo:

```
     MOV cx, 15 ; CX=15; 15 sería el número de veces que se va a ejecutar el
      bucle.
```

```
ETIQUETA:
        ; Aquí estarían las instrucciones que están dentro del bucle.
        .
      LOOP etiqueta    ; CX=CX-1 y bifurca a ETIQUETA si CX es distinto a cero.
```

* **LOOPE:** Esta instrucción al igual que LOOP, permite realizar "bucles" utilizando el registro CX como contador (CX en un contador que va decrementándose) pero además el flag de cero debe estar a uno (ZF=1). Es equivalente a LOOPZ (Bucle si cero). Esta instrucción equivale al par: JNE FIN // LOOP OTRO. El formato es: LOOPE etiqueta.

Ejemplo:

```
          MOV cx, Length tabla ; CX=longitud de la TABLA.
          MOV si, inicio       ; Movemos a SI el inicio de la TABLA.
          DEC si         ; Esto se hace para poder realizar el bucle
                               ; que viene ahora
    OTRO: INC si         ; Movemos a SI su valor inicial.
          CMP tabla[SI], 0     ; Comparamos cada valor de la TABLA con cero
          LOOPE OTRO           ; si el valor de TABLA es igual a cero realiza un
                               ; LOOP normal, sino, no hace el LOOP.
```

* **LOOPNE**: Esta instrucción al igual que LOOP, permite realizar "bucles" utilizando el registro CX como contador (CX en un contador que va decrementándose) pero además el flag de cero debe estar a cero (ZF=0). Es equivalente a LOOP-NZ (Bucle si no cero). Esta instrucción equivale al par: JE FIN // LOOP OTRO.
  El formato es: LOOPNE etiqueta.

Ejemplo:

```
          MOV cx, Length tabla ; CX=longitud de la TABLA.
```

```
      MOV si, inicio        ; Movemos a SI el inicio de la TABLA.
       DEC si         ; Esto se hace para poder realizar el bucle
                             ; que viene ahora.
OTRO: INC si                 ; Movemos a SI su valor inicial.
       CMP tabla[SI], 0      ; Comparamos cada valor de la TABLA con cero
       LOOPNE OTRO           ; si el valor de TABLA es distinto a  LOOP normal,
                             ; sino, no hace el LOOP.
```

Instrucciones de Rotación y Traslación.

Este grupo de instrucciones nos permitirán tratar la información almacenada en registros o en memoria mediante el tratamiento unitario de sus bits.

*** RCL:**(Rotar a la izquierda con acarreo).

El formato es: RCL operando, contador. Rota a la izquierda los bits del operando junto con la bandera de acarreo, CF, el número de bits especificado en el segundo operando. Si el número a desplazar es 1, se puede especificar directamente (Por ejemplo: RCL AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:

```
MOV cl, 3      ; Rotar 3 bits
               ; AL = 0101 1110b, CF=0 (Flag de acarreo=0)
RCL al, cl           ; AL = 1111 0001b, CF=0
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 0101 1110b, CF=0	AL = 1011 1100b, CF=0
2	AL = 1011 1100b, CF=0	AL = 0111 1000b, CF=1
3	AL = 0111 1000b, CF=1	AL = 1111 0001b, CF=0

* **RCR**:(Rotar a la derecha con acarreo).

El formato es: RCR operando, contador. Rota a la derecha los bits del operando junto con la bandera de acarreo, CF, el número de bits especificado en el segundo operando. Si el número a desplazar es 1, se puede especificar directamente (Por ejemplo: RCR AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:

```
MOV cl, 3     ; Rotar 3 bits
              ; AL = 0101 1110b, CF=0 (Flag de acarreo=0)
RCR al, cl    ; AL = 1000 1011b, CF=1
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 0101 1110b, CF=0	AL = 0010 1111b, CF=0
2	AL = 0010 1111b, CF=0	AL = 0001 0111b, CF=1
3	AL = 0001 0111b, CF=1	AL = 1000 1011b, CF=1

* **ROR:**(Rotar a la derecha).

El formato es: ROR operando, contador. Rota a la derecha los bits del operando de tal forma que el bits del extremo derecho del operando destino para al bit extremo izquierdo de dicho operando y al mismo tiempo para el bit de acarreo (CF). Si el número a desplazar es 1, se puede especificar directamente (Por ejemplo: ROR AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:
```
MOV cl, 2    ; Rotar 2 bits
             ; AL = 0011 0011b, CF=0 (Flag de acarreo=0)
RCR al, cl   ; AL = 1100 1100b, CF=1
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 0011 0011b, CF=0	AL = 1001 1001b, CF=1
2	AL = 1001 1001b, CF=1	AL = 1100 1100b, CF=1

* **ROL:**(Rotar a la izquierda).

El formato es: ROL operando, contador. Rota a la izquierda los bits del operando de tal forma que el bits del extremo izquierdo del operando destino para al bit extremo derecho de dicho operando y al mismo tiempo para el bit de acarreo (CF). Si el número a desplazar es 1, se puede especificar directamente (Por ejemplo: ROL AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:
```
MOV cl, 2     ; Rotar 2 bits
              ; AL = 1100 1100b, CF=0 (Flag de acarreo=0)
RCR al, cl    ; AL = 0011 0011b, CF=1
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 1100 1100b, CF=0	AL = 1001 1001b, CF=1
2	AL = 1001 1001b, CF=1	AL = 0011 0011b, CF=1

* **SAL:** (Desplazamiento aritmético a la izquierda). Es equivalente a SHL (Desplazamiento lógico a la izquierda).
  El formato es: SAL operando, contador. SHL y SAL realizan la misma operación y son físicamente la misma instrucción. Copia en cada bit del operando el contenido previo del bit de su derecha. El bit de menor peso se pone a cero. El contenido previo del bit de mayor peso se copia en el flag de acarreo (CF). Es equivalente a multiplicar el operando por dos, tanto para números sin signo como para número en complemento a 2, siempre el resultado no se salga del rango. Si el número de bits a desplazar es 1, se puede especificar directamente (Por ejemplo: SAL AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:

```
MOV cl, 2      ; Desplazar 2 bits
               ; AL = 1100 1100b, CF=0 (Flag de acarreo=0)
SAL al, cl     ; AL = 0011 0000b, CF=1
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 1100 1100b, CF=0	AL = 1001 1000b, CF=1

2	AL = 1001 1000b, CF=1	AL = 0011 0000b, CF=1

* **SAR:** (Desplazamiento aritmético hacia la derecha con extensión de signo). El formato es: SAR operando, contador. Copia en cada bit del operando el contenido previo del bit de su izquierda. El bit de mayor peso mantiene su valor anterior. El contenido previo del bit de menor peso se copia en el flag de acarreo (CF). Es equivalente a dividir el operando por dos para números en complemento a 2. Si el número de bits a desplazar es 1, se puede especificar directamente (Por ejemplo: SAR AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:

```
MOV cl, 2     ; Desplazar 2 bits
              ; AL = 1100 1100b, CF=0 (Flag de acarreo=0)
SAR al, cl    ; AL = 1111 0011b, CF=0
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 1100 1100b, CF=0	AL = 1110 0110b, CF=0
2	AL = 1110 0110b, CF=0	AL = 1111 0011b, CF=0

* **SHR:** (Desplazamiento aritmético hacia la derecha).

El formato es: SAR operando, contador. Copia en cada bit del operando el contenido previo del bit de la izquierda. En el bit de mayor peso se almacena un 0. El contenido previo del bit de menor peso se copia en el flag de acarreo (CF). Es equivalente a dividir el operando por

dos para números sin signo. Si el número de bits a desplazar es 1, se puede especificar directamente (Por ejemplo: SHR AL, 1). Si es mayor que 1, su valor debe cargarse en CL y especificar CL como segundo operando.

Ejemplo:

```
MOV cl, 2     ; Desplazar 2 bits
              ; AL = 0011 0011b, CF=0 (Flag de acarreo=0)
SHR al, cl    ; AL = 0000 1100b, CF=1
```

Procedimiento:

Cuenta (CL)	Antes	Después
1	AL = 0011 0011b, CF=0	AL = 0001 1001b, CF=1
2	AL = 0001 1001b, CF=1	AL = 0000 1100b, CF=1

Ejercicios:

1.- Disponemos en memoria de una variable que nos ocupa una palabra, identificada con el símbolo UNO y que tiene el valor de 35 (dentro del segmento de datos: UNO DW 35), y disponemos de un byte identificado con el símbolo DOS y que posee un valor de 10 (dentro del segmento datos: DOS DB 10). Calcular la suma de estos datos.

2.- Acceder a un datos que está almacenado en la dirección 123Ah:0008 en una palabra de memoria dentro del segmento extra, y calcular lo siguiente:

a) Si los bits: 11, 9, 5 y 3 están a uno.
b) El número de bits a uno que tiene ese dato.
c) Si este dato es de paridad impar, debe saltar a una etiquea que se llama FIN.

3.- Supongamos que tenemos cargados en variables de memoria (UNO, DOS, TRES, CUATRO, CINCO, SEIS, SIETE de tipo byte), siete informaciones de ficheros que hemos leido de un disco. Cada información puede tener, en sus cuatro bits menos signifactivos, los siguientes atributos:

- Si bit 3 a 1 - Atributo de Lectura.
- Si bit 2 a 1 - Atributo de Sistema.
- Si bit 1 a 1 - Fichero Oculto.
- Si bit 0 a 1 - Fichero Borrado.

Se quiere saber cuantos ficheros de estos siete son: de lectura, de sistema, ocultos, borrados, de lectura y sistema, de lectura y oculto, de sistema y oculto y de lectura y sistema y oculto.

```
UNO   DB   1111 0111
                  DOS       DB   1111 1000
                  TRES      DB   1111 0101
```

```
CUATRO    DB  1111 1110
CINCO     DB  1111 1111
SEIS      DB  1111 0010
SIETE     DB  1111 1110
```

4.- Realiza una rutina que nos permita multiplicar dos cantidades que estén almacenadas en dos palabras de memoria, que conocemos con los simbolos UNO y DOS, considerando que dicha multiplicación debe realizarse mediante sumas sucesivas.

5.- Realizar una rutina que nos permita realizar la división de dos cantidades numéricas almacenadas en 2 dirección de memoria que vamos a conocer con los nombres UNO y DOS, considerando que dicha división se debe realizar mediante resta sucesivas.

6.- Disponemos de una cantidad almacenada en memoria identificada por 1 palabra mediante el símbolo UNO. Calcula el factorial del valor de la palabra.

La Pila.

La estructura de una PILA es similar a un montón de libros apilados: los elementos se van ordenando cada uno detrás del último en llegar (es decir, los libros se van apilando cada uno encima del anterio), pero al sacarlos de la estructura se empieza por el último en llegar, acabando por el primero (al retirar los libros se comienza por el superior, y se acaba por el que queda abajo del todo).

A la operación de introducir un elemento en una pila se le suele dar el nombre de empujar un elemento (**push** en inglés). La operación de extraer un elemento de una pila se le denomina **pop**.

Los elementos que puede almacenar la pila del microprocesador son valores de 16 bits, con lo cual el puntero de pila se debe incrementar o decrementar 2 unidades a la hora de sacar o introducir valores en la pila (a meter un valor de 16 bits en la pila el puntero de la pila se decrementa en dos unidades, y a la hora de sacar un elemento de la pila el puntero se incrementa en dos unidades; la pila crece hacia abajo en lugar de hacia arriba).

El microprocesador tiene dos registros que se utilizan para gestionar la pila: el SS (Segmento de Pila) y el SP (Puntero de Pila). El par SS:SP da la dirección donde se encuentra el último valor empujado en la pila.

* **PUSH:** Decrementa en 2 unidades el puntero de la pila, es decir, decrementa en 2 unidades el registro SP, y a continuación almacena en la cima de la pila la palabra especificada en el operando origen asociado a la instrucción. Formato PUSH origen

Ejemplo:
```
PUSH ax      ;es equivalente a:  SP = SP-2  //  MOV ss:[sp], ax
```

El operando origen no puede ser un operando inmediato (ni el registro de segmento CS).

* **POP:** Esta instrucción toma una palabra de la cima de la pila y la sitúen el operando destino asociado a la instrucción, incrementando, a continuación, en 2 unidades el puntero de la pila.
Formato POP origen

Ejemplo: `POP ax ; es equivalente a: AX = SS:[SP] // SP = SP + 2`

Cuando una instrucción PUSH o POP se ejecuta en un código de programa con el tamaño de registro de 32 bits (USE32), el ensamblador utiliza como valor de trasferecencia 4 bits en lugar de 2 bytes (una palabra), y las operaciones realizadas con ESP se efectúan sobre unidades de 4 elementos.

* **PUSHF:** Esta instrucción decrementa en 2 unidades el puntero de la pila y a continuación, almacena en la cima de la pila el registro de indicadores (FLAGS). No tiene ningún operando.

* **POPF:** Esta instrucción almacena en el registro de indicadores (FLAGS) la palabra situada en la cima de la pila aumentando en 2 unidades, a continuación, el puntero de la pila. No tiene ningún operando.

* **PUSHA y POPA:** Estas instruciones almacenan y sacan de la pila la información contenida en los registros siguientes y en el

orden siguiente: AX, CX, DX, BX, SP, BP, SI y DI. El valor de SP es guardado en la pila antes de que el primer registro sea guardado. En el caso de utilizar registros de 32 bits la instrucciones serían: PUSHAD y POPAD.

Todo lo que entra en la pila, tiene que salir de la pila. El orden de situar y sacar palabras de la pila es el siguiente:

```
PUSH ax
PUSH bx
PUSH cx
PUSH dx
.
Rutina del programa
.
POP dx
POP cx
POP bx
POP ax
```

Ejercicios:

1.- Se pide calcular los números comprendidos entre dos cantidades numéricas almacenados en palabras y que vamos a identificar con los símbolos UNO y DOS. Se debe utilizar de forma obligatoria instrucciones que manejen la pila.

Interrupciones.

Una interrupción es una señal que provoca la suspensión del programa que se estaba ejecutando y provoca el comienzo de ejecución de un programa de tratamiento que de solución a esa interrupción.

A ese programa se le conoce como RUTINA DE TRATAMIENTO de esa interrupción.

Este procesador nos presenta tres grupos de interrupciones:

a) Interrupciones Hardware o Interrupciones Externas, que son aquellas provocadas por los dispositivos periféricos, controladas por un procesador especial de interrupciones (8259) o IPC (Controlador de Interrupciones Programable), y la rutina de tratamiento está "cableada".

b)Interrupciones Internas, que son aquellas provocadas dentro del propio procesador por una situación anormal de funcionamiento de alguna de sus partes.

C) Interrupciones de Software, Son aquellas que son programables y que podemos cambiar. Las interrupciones de software podemos llegar a manejarlas y por ello el ensamblador nos proporciona una instrucción que nos permita poner en funcionamiento una determinada rutina de interrupción; esta instrucción es INT.

* INT. Formato INT núm_entero. Ese "núm_entero", asociado a la instrucción, es un identificativo que nos dice mediante la aplicación de un algoritmo, la posición de Memoria Interna donde se encuentra almacenada la dirección de comienzo de la rutina de tratamiento de esa interrupción.

El ensamblador permite, normalmente, identificar 256 interrup-

ciones. Una parte de ellas son las correspondientes a la ROM-BIOS y las proporciona el fabricante.

Otra parte de ellas forman del sistema operativo DOS, y otra parte de ellas queda libre para que el programador genere sus propias rutinas de interrupción.

Las interrupciones correspondientes a la parte de la BIOS y las correspondientes a la parte del DOS representas características similares.

Existe un flag denominado IF (Interrupción Flag, Flaf de Interrupción) que determina la reacción del microprocesador ante una interrpción. Si el flag está a uno, el rpocesador responde a la interrupción producida; pero si el flag IF está a cero, la petición de interrupción será ignorada completamente por el microprocesador.

En algunas secciones de código, resulta necesario deshabilitar las interrupciones (poner el flag IF a cero) durante algunos ciclos, y habilitarlas de nuevo después.

La familia 8086 provee dos instrucciones que realizan estas tareas:

STI (Activar flag de interrupciones): Pone el flag IF a i, de forma que se premiten las interrupciones.

CLI (Borrar flag de interrupciones): Pone el flag IF a 0, de modo que el microprocesador no responde a más interrupciones hasta que se ejecuta un STI o se altera el contenido de los flags (entre ellos el de IF) recuperándolos de la pila con POPF o IRET.

```
MOV ax, 8000h
CLI
MOV ss, ax
MOV sp, 2000h
STI
```

* IRET: Retorno de interrupción. Formato: IRET (no tiene operandos). Retorna de una rutina de servicio a la interrupción, extrayendo de la pila los nuevos valores de IP y CS, en este orden, y el contenido del registro de flags. La ejecución continúa en la instrupción siguiente a la que se estaba ejecutando cuando ocurrió la interrupción.

Ejemplos de interrupciones del DOS.

Vamos a ver unos ejemplos de interrupciones del DOS (Vamos a ver unas interrupciones donde el "nº entero" va a ser 21h. Esta interrupción presenta una gran cantidad de funciones diversas; por ello además de indicar el "nº entero", debemos indicar también el "nº función" que deseamos dentro de esa interrupción. Dicho número se almacena siempre el registro AH):

- *INT 21h. Función 01h*: Permite dar entrada a un carácter e teclado y al mismo tiempo dicho carácter aparece en pantalla, en la posición en la que se encuentre el cursor. El carácter tecleado queda almacenado en AL. Si no hay ningún carácter disponible, se espera hasta que haya alguno.

```
MOV ah, 01h
INT 21h      ; El carácter tecleado queda en AL
```

- *INT 21h. Función 02h*: Permite llevar un carácter desde el pocesador hacia la pantalla. Dicho carácter debe estar almacenado en el registro DL. Aparecerá en la posición donde se encuentre el cursor.

```
MOV dl, carácter
MOV ah, 02h
INT 21h
```

- *INT 21h. Función 08h*: Permite dar una entrada de un carácter desde el teclado pero sin que aparezca en pantalla. El carácter tecleado queda almacenado en el registro Al.

Si no hay un carácter disponible se espera hasta que lo haya.

```
MOV ah, 08h
INT 21h      ; El carácter tecleado queda en AL
```

- *INT 21h. Función 09h*: Visualización de una cadena de caracteres. Nos permite llevar una cadena de caracteres hacia la pantalla. Dicha cadena aparecerá a partir de la posición en la que se encuentre el cursor. Esta función necesita que en el registro DX se encuentre la dirección de comienzo de la cadena a presentar en pantalla.

```
MOV dx, offset cadena    ; En DX queda el desplazamiento que hay que hacer dentro
                         ; de DS para llegar a la posición donde se encuentra
                         ; "cadena" (DS:DX).
MOV ah, 08h
INT 21h
```

- *INT 21h. Función 4Ch*: Acabar el proceso con código de retorno. Permite realizar el retorno al Sistema Operativo. Acaba el proceso actual, enviando un código de retorno al programa original. Se trata de uno de los diversos métodos con los que se puede provocar una salida definitiva de un programa.

```
MOV ah, 4ch
INT 21h
```

Otras instrucciones

* **LEA**: Formato: LEA destino, fuente. Transfiere el desplazamiento del operando fuente al operando destino. El operando fuente debe ser un operando de memoria. El operando destino es un registro, pero no un registro de segmento. LEA permite especificar regitros índices en el operando fuente, al contrario que con la instrucción OFFSET.

Ejemplo:

```
LEA dx, [bx+si+20] ; En DX quedaría el desplazamiento que habría
                         ;que hacer dentro del segmento de datos (DS), para
                         ; acceder a la información
                         ; indicada por la suma de BX + SI + 20.
                         ; DX - BX + SI + 20
                         ; DS : DX = BX + SI + 20
(Segmento de Datos)
                 TEXTO db "ejemplo1$"
                 .
(Segmento de Código)
                 .
LEA dx, texto ; Queda en DX el desplazamiento que hay que hacer dentro del
              ; segmento de datos (DS), para acceder a la información que tiene
              ; la variable TEXTO DS:DX= dirección del texto.
```

Si situamos todos los datos que emplea nuestro programa en un sólo segmento, apuntando por DS, la localización de memoria de un dato se puede dar por un desplazamiento (offset), suponiendo implícitamente que reside en el segmento apuntando por DS.

Este tipo de punteros (un puntero es un valor que indica la

localización de otra variable) se denominan NEAR POINTERS (Puntero cercanos).

Pero si queremos especificar la dirección donde reside un dato que puede estar en cualquier lugar del megabyte direccionable, es necesario especificar tanto segmento donde se encuentra el dato como el desplazamiento dentro de dicho segmento; este tipo de punteros se denominan FAR POINTERS (Punteros lejanos).

Cuando almacenamos un puntero cercano en memoria, se almacena únicamente una palabra. Pero cuando alamacenamos un puntero lejano, se almacena el desplazamiento y el segmente en palabras consecutivas en memoria.

Al cargar un puntero cercano, por ejemplo, SI, se carga directamente de memoria con instrucciones como MOV SI, mem, de manera que el par DS:SI contiene el puntero deseado. Pero al cargar un puntero lejano, es necesario cargar tanto un registro con el desplazamiento (offest) como un registro de segmento del puntero (habitualmente se carga en ES, manteniendo DS siempre constante apuntando a los datos de uso habitual del programa).

Existen 2 instrucciones que cargan de una sola vez tanto el desplazamiento como el segmento: LDS y LES.

```
LDS reg, mem      LDS si, cs:[di+2]
LES reg, mem      LES ax, [bp+si]
```

Ambas instrucciones cargan en el registro especificado con reg la palabra contenida en la dirección dada por mem, y en el registro de segmento indicado (DS para LDS y ES para LES) la palabra contenida en la dirección indicada +2.

*** OFFSET:** Formato OFFSET variable o OFFSET etiqueta. Podemos utilizar la instrucción OFFSET para obtener el desplaza-

miento dentro de un segmento de una etiqueta cualquiera.

Ejemplo:

```
        (Segmento de Datos)
        TABLA db 'ejemplo 1$'
           .
        (Segmento de Código)
           .
        MOV ax, offeset tabla   ; En AX queda el desplazamiento que hay que hacer
                                ; dentro del segmento por defecto en curso para
                                ; acceder al contenido de TABLA.
                                ; AX = desplazamiento de TABLA.
```

Estructuras de programación. Directivas.

* IF: Las instrucciones que empiezan por "IF" son directivas condicionales. Sirven para que el ensamblador incluya o no las sentencias que vienen a continuación, según se cumpla o no una determinada condición.

El formato es:

```
.IF condicional
sentencias
[ .ELSELF condición2
sentencias ]
[ .ELSE
sentencias ]
.ENDIF
```

Ejemplo:

```
.IF cx == 20
   MOVE dx, 20
.ELSE
   MOVE dx, 30
.ENDIF
```

Algunos operadores utilizados para la comparaciones son:

```
== Igual
!= Distinto (no igual)
>  Mayor
>=Mayor Igual
<  Menor
<=Menor Igual
```

```
!    NOT Lógico
&&AND Lógico ("y")
||  OR Lógico ("o")
```

* **FOR:** Las instrucciones que empiezan por "FOR" son directivas de "ciclos".

El formato es:

```
FOR parámetro, <Lista de argumentos>
ENDM
```

Ejemplo:

```
MOV ax, 0
FOR arg, <1,2,3,4,5,6,7,8,9,10>
     ADD ax, arg
ENDM ; Esto lo que hace es sumar cada vez a AX el valor que
      ; aparece en la de argumentos.
      ; ( AX = AX + 1, AX = AX + 2, AX = AX + 3,...)
```

* **WHILE:** Las instrucciones que empiezan por "WHILE" son directivas de "ciclos".

El formato es:

```
.WHILE condición
     sentencias
.ENDW
```

Ejemplo:

```
(Segmento de Datos)
```

```
    buf1    BYTE        "Esto es una cadena", '$'
    buf2    BYTE  DUP (?)
(Segmento de Código)
    XOR bx, bx
    .WHILE  (but1[bx]!= '$')
            MOV al, buf1[bx]
            MOV buf2[bx], al
            INC bx
    .ENDW
```

*** REPEAT:** Las instrucciones que empiezan por "REPEAT" son directivas de "ciclos".

El formato es:

```
.REPEAT
     sentencias
.UNTIL condición
```

Ejemplo:

```
(Segmento de Datos)
    buffer  BYTE 100 DUP (0)
(Segmento de Código)
    XOR bx, bx
    .REPEAT
            MOV ah, 01h
            INT 21h
            MOV buffer[bx], al
            INC bx
    .UNTIL (al==13)     ; En este caso la interrupción 21h con la
```

función 01h deja en AL la tecla que se ha pulsado y la mete en "buffer". Si esta es ENTER (13) se sale del bucle, sino sigue en él.

Estructura de un programa en Ensamblador:

Para entrar directamente en materia, veamos un listado en lenguaje ensamblador de un programa corriente: se trata de un programa que muestra la cadena "Primer Programa" en el monitor y retorna al Sistema Operativo.

```
PILA SEGMENT STACK 'STACK'       ; Abre el segmento de PILA.
   DW 100h DUP (?)                     ; Reserva 100 palabras para la PILA.
PILA ENDS                        ; Cierra el segmento de PILA.
DATOS SEGMENT 'DATA'             ; Abre el segmento de DATOS.
  mensaje DB "Primer Programa", '$'     ; Mensaje a escribir.
DATOS ENDS                 ; Cierra el segmento de DATOS.
CODIGO SEGMENT 'CODE'            ; Abre el segmento de CODIGO.
ASSUME CS:CODIGO, DS:DATOS, SS:PILA
ENTRADA:
  MOV ax, DATOS                  ; Valor de segmento para DATOS.
  MOV ds, ax               ; Para acceder a "mensaje".
  MOV dx, OFFSET mensaje         ; Para la interrupción 21h, función 09.
  MOV ah, 09   ; Especifica el servicio o función 09.
  INT 21h      ; Invoca el servicio 09: Imprimir Cadena.
  MOV ax, 4C00h      ; Servicio (Función) 4Ch, con valor de retorno 0.
  INT 21h      ; Invoca servicio 4Ch: Retorno al DOS.
CODIGO ENDS          ; Cierra el segmento de CODIGO.
END ENTRADA          ; Final del modulo fuente y primera instrucción
                     ; desde donde debe empezarse a ejecutar el programa.
```

Veamos primero el formato de SEGMENT:

nombre SEGMENT [alineamiento] [READONLY] [combinación] ['clase']

.....

nombre ENDS

Sirve para indicarle al ensamblador que la información que venga a continuación, y hasta que no encuentre una directiva de fin de segmento (ENDS), corresponde al mismo segmento.

- nombre: Indica el nombre del segmento que vamos a utilizar. El "nombre" indicado en SEGMENT y el "nombre" indicado en ENDS deben ser el mismo.

- alineamiento: En "alineamiento" vamos a situar una información que le dirá al ensamblador las características que debe tener la dirección de memoria que elija para realizar la carga de ese segmento. Los valores que le podemos dar son:
  - BYTE: Puede colocar el segmento en cualquier dirección.
  - WORD: Debe colocar el segmento en una dirección múltiplo de 2.
  - DWORD: Debe colocar el segmento en una dirección múltiplo de 4.
  - PARA: Debe colocar el segmento en una dirección múltiplo de 16. Es el alineamiento por defecto.

- READONLY: Informa al ensamblador de un error producido cuando alguna instrucción del segmento que contiene la sentencia READONLY es modificada.

- combinación: La combinación nos dirá una serie de informaciones para el montador (LINKER). Le podemos situar las posibilidades:
  - PRIVATE: No combina el segmento con segmentos de otros

módulos, aún cuando tengan el mismo nombre.

- PUBLIC: Le dice al montador (LINKER) que este segmento y todos los que tengan el mismo nombre en 'clase' se concatenarán en un mismo segmento.
- STACK: Indica que estamos tratando con un segmento de pila. Esta identificación es obligatoria en el caso de que el segmento contenga la pila. Al menos dabe haber un segmento de pila para crear un módulo ejecutable con el montador (LINKER).
- AT expresión: Sirve para indicarle al montador en que dirección de memoria deberá situar la información correspondiente a éste segmento. La dirección de memoria viene especificada en el parámetro "expresión".
- COMMON: Indica que este segmento y todos los del mismo nombre ("clase") que procese el montador empezarán en la misma dirección, solapándose entre sí. La longitud asignada por el montador es la longitud máxima de todos los segmentos COMMON procesados.
- MEMORY: EL segmento se ubicará en una dirección de memoria superior a la de otros que aparecerán durante el montaje (LINK) del programa. Se puede aplicar, por ejemplo, para utilizar la memoria más allá de los límites del programa. Sólo puede haber un segmento de este tipo. Si existieran varios, solo se procesará el primero como tal, y el resto se procesará como COMMON.

- use (sólo 80386/486): Determina el tamaño del segmento. USE16 indica que el desplazamiento es de 16 bits de ancho, y USE32 indica que el desplazamiento es de 32 bits de ancho.

- 'clase': La 'clase' es un nombre que sirve para que el montador pueda unificar todos los segmentos que tengan asociados dicho nombre, si es que le damos la posibilidad de ello.

Volvamos al programa.

PILA SEGMENT STACK 'STACK'.

Esta línea lleva la directiva SEGMENT, que hace que el ensamblador incluya una entrada en el fichero objeto para este segmento.

PILA es el nombre con el que nos referimos a éste segmento dentro del listado fuente, mientras que 'STACK' incluida entre comillas simples es utilizada por el enlazador (LINKER). La palabra STACK sin comillas indica tanto al ensamblador como al enlazador (LINKER) que este segmento se utilizará como espacio de pila en ejecución.

Cada línea SEGMENT debe tener su una correspondiente ENDS (Fin de segmento).

DW 100h DUP (?) ; esta línea lleva la directiva DW (definir palabra).

Esta directiva permite fijar directamente los valores incluidos en el módulo al tamaño, en este caso, de una palabra, el uso de DUP nos permite fijar un número determinado de palabras (100h = 256 en este caso) a un mismo valor.

El valor se indica entre paréntesis; en esta línea es un interrogante, con lo que indicamos al ensamblador que no son importantes los contenidos iniciales de esta 256 palabras. El enlazador puede intentar reducir el tamaño del ejecutable gracias a que no nos interesan lon contenidos iniciales de esta posiciones de memoria.

La instrucción DUP se utiliza para definir arrays (vectores).

El formato sería:

contador DUP (valor_inicial [valor_inicial]...)

El valor de "contador" nos dice el número de veces que se repite lo que hay en "valor_inicial". El "valor_inicial" debe definirse siempre entre parétesis. El "valor_inicial" suele se o un "?" que nos indica que no nos interesa el contenido inicial del vector o un número, que nos indicará el contenido de cada posición del vector.

Ejemplos:

baray BYTE 5 DUP (1) ; En este cado tenemos un vector de 5 posiciones, y cada posición ocupa un BYTE y tiene un valor inicial de 1 (cada posición).

array DWORD 10 DUP (1) ; En este caso tenemos un vector de 10 posiciones, y cada posición ocupa una Doble Palabra (DWORD) y tiene un valor inicial de 1.

buffer BYTE 256 DUP (?) ; Vector de 256 posiciones, y cada posición ocupa un BYTE y tiene un valor inicial que no nos interesa.

También se puede definir un "array" de la siguiente forma:

```
warray WORD 2, 4, 6, 8, 10
```

La forma de acceder a alguno de estos elementos dentro del código de segmento sería:

```
DATOS SEGMENT 'DATA'
      btabla BYTE 12 DUP (?)
      warray WORD 2, 4, 6, 8, 10
DATOS ENDS
```

```
CODIGO SEGMENT 'CODE'
        ASSUME CS:CODIGO, DS:DATOS
UNO:
        MOV ax, DATOS
        MOV ds, ax
        XOR al, al      ; Ponemos lo que vamos a meter en "btabla" a 0
        MOV si,0        ; colocamos el puntero de "btabla" (SI) a 0 (el primer elemento de la matriz o el array empieza en la posición cero)
        MOV cx, 12              ; contador a 12.
DOS:
        MOV btabla[si], al    ; Movemos un valor de un BYTE (AL), (ya que hemos definido "btabla" de tipo BYTE) a "btabla". También podemos poner esta instrucción como:  MOV btabla + SI, AL
        INC si                  ; Al tratarse de elementos de un BYTE el puntero sólo se debe incrementar en una unidad.
        LOOP DOS
        XOR ax, ax
        MOV si; 0       ; Ponemos el índice del array a cero
        MOV xc, 5
TRES:
        MOV ax, warray[si]    ; Movemos el valor que indica el puntero (SI) dentro de "warray" a AX (Ya que hemos definido "warray" de tipo PALABRA).
        ; También podemos poner esta instrucción
        ; como:  MOV AX, warray + SI
        ADD si, 2               ; Incrementamos SI en 2 unidades debido a que "warray" esta definido con PALABRAS.
        LOOP TRES
  CODIGO ENDS
        END UNO
```

El algoritmo para acceder a un elemento dentro de un array, a partir de la dirección de comienzo (vector), puede ser (sería el valor que habría que moverle al puntero dentro del array para ac-

ceder al elemento a buscar):

(ELEMENTO - 1) * TIPO

Donde: ELEMENTO es el elemento a buscar y TIPO es el tamaño con el que hemos definido el array (1 - BYTE, 2 - WORD (2 bytes), 4 - DWORD (4 bytes), etc).

En el caso de matrices, ésta nos presenta un almacemaniento que ocupa posiciones consecutivas y que lo debemos tratar como un vector. Para tratar dicho vector se necesita conocer el número de filas y el número de columnas. El algoritmo para acceder a un elemento, a partir de la dirección de comienzo, sería:

(FILA - 1) * Nº COL * TIPO + (COL - 1) * TIPO

Donde: FILA es la posición de la fila a buscar, Nº COL es el número de columnas que tiene cada fila. TIPO es lo mismo que en los vectores. COL es la posición de la columna a buscar.

Por ejemplo: En una matriz de 4 filas por 3 columnas tipo BYTE, queremos ver el contenido de la posición (3, 2). (3 - 1) * 3 * 1 + (2 - 1) * 1

PILA ENDS

Cierra el segmento de pila. Esta directiva (ENDS) cierra el segmento abierto pos la directiva PILA SEGMENT. El identificador que antecede a ENDS debe ser el mismo que el que antecede a la directiva SEGMENT correspondiente, recordando que no son importantes las mayúsculas o minúsculas.

DATOS SEGMENT 'DATA'

Abre el segmento de datos. Esta directiva abre otro segmento. Este segmento lo utilizaremos para los datos del programa.

Mensaje DB "Primer Programa", '$'

Mensaje a escribir. (También lo podemos ver definido como: 'Primer Programa$,' o como: "Primer Programa","$", etc.).

Esta directiva sirve para reservar memoria para la variable mensaje. DB quiere decir "definir byte". En éste caso, se reservan 16 bytes (15 del mensaje y uno del $). El carácter $ se utiliza como delimitador del texto a escribir. Es decir, cuando vayamos a escribir mensaje por pantalla se escribirán todos los caracteres hasta encontrar el carácter $.

También podemos encontrar, por ejemplo, el mensaje:

```
"Pimer Programa", 13, 10, "$"
```

que nos indicaría que, después de escribir el texto, ejecutaría el carácter 13 (retorno de carro) y luego el carácter 10 (salto de línea).

En cuanto al tamaño de los datos, éste puede ser:

TAMAÑO	RANGO
BYTE, DB	Valor de 0 a 225 (1 Byte).
SBYTE	Valor entre: -128 y +127 (1 Byte).
WORD, DW	Valor de 0 a 65.535 (2 Bytes).
SWORD	Valor entre: -32.768 t +32.767 (2 Byte).
DWORD, DD	Valor de 0 a 4 Megabytes (4.294.967.259) (4 bytes).
SDWORD	Valor entre: -2.147.438.648 y +2.147.438.647 (4 bytes)
FWORD, DF	Tamaño de 6 Bytes. Utilizando sólo como variable de puntero para los procesadores 386/486.

```
QWORD, DQ    Tamaño de 8 Bytes.
TBYTE, DT    Tamaño de 10 Bytes.
```

En cuanto a número reales en "coma flotante":

TIPO DE DATO	BITS	DIGITOS SIGNIF.	RANGO APROXIMADO
REAL4	32	6 - 7	De: $\pm 1.18 \times 10^{-38}$ a $\pm 3.40 \times 10^{38}$
REAL8	64	15 - 16	De: $\pm 2.23 \times 10^{-308}$ a $\pm 1.79 \times 10^{308}$
REAL10	80	19	De: $\pm 3.37 \times 10^{-4932}$ a $\pm 1.18 \times 10^{4932}$

Para expresar un número real se utiliza este formato:

[+/-] entero. [fracción] [E] [[+/-] exponente]

Ejemplos:

```
corto     REAL4  25.23       ; 25,23
doble   REAL8  2.523E1       ; 2.523x10^1 = 25,23
diezbytes REAL10 -2523.0E-2  ; = -2523x10^-2 = -25,23
```

Seguimos con el programa.

DATOS ENDS

Cierra el segmento de datos.

CODIGO SEGMENT 'CODE'

Abre el segmento de código. Abre el segmento de código, donde incluímos el código del programa. Ni el nombre CODIGO, ni la clase 'CODE' son tratados de forma especial por el ensamblador o el enlazador.

Es en la última línea del programa fuente donde se indica el punto de entrada al programa, fijando tanto el valor de segmento

inicial para CS como el desplazamiento inicial dentro del segmento.

ASSUME CS:CODIGO, DS:DATOS, SS:PILA

Esta línea informa al ensamblador de los segmentos a los que apuntarán durante la ejecución los diferentes registros de segmento.

De este modo, si intentamos acceder a la etiqueta 'mensaje' definida en el segmento "DATOS", el ensamblador sabrá que puede acceder a ella por medio del registro DS. El registro CS se asocia con el segmento de CODIGO y el registro SS con el segmentos de PILA.

MOV AX, DATOS

Valor de segmento para "DATOS" (AX=dirección del segmento de DATOS). Almacena la componente segmento del segmento DATOS sobre el registro AX. Como el registro DS apunta al comienzo del PSP ***(Prefijo de Segmento de Programa; antes de que el procesador de comandos (COMAND.COM) del DOS pase el control al programa, construye un bloque de 256=100h bytes a partir de la primera posición de memoria disponible. El PSP contine campos como la dirección de retorno al DOS cuando acabe de ejecutarse el programa, la dirección del código si se pulsa Ctrl-Break, la dirección de la rutina del tratamiento de errores críticos, la cantidad de memoria disponible para el programa y los parámetros, etc.)***, es necesario cambiarlo para que se pueda acceder a los datos (en el segmento de datos) mediante el registro DS. La instrucción MOV no permite mover directamente a DS, por lo que se utiliza el registro AX como registro intermedio.

MOV DS, AX

Para acceder a 'mensaje'. Esta líneas inicializan el registro DS para que apunte al segmento donde reside el mensaje "mensaje". Como se ve, se puede utilizar el nombre de un segmento (DATOS en este caso) como valor inmediato, en cuyo caso el ensamblador, el

enlazador y el cargador del MS-DOS harán que el dato final cargado sea el valor de segmento adecuado (hemos direccionado el segmento DATOS mediante DS).

Al segmento de código no hace falta direccionarlo con CS, pues lo hace el sistema operativo MS-DOS.

Tampoco hace falta direccionar el registro SS para acceder a la pila, pues el DOS lo inicializa también.

En éste programa no existe segmento extra.

MOV DX, offset "mensaje"

Esta instrucción carga en el registro DX el desplazamiento de la etiqueta "mensaje", dentro del segmento donde ha sido definida. Podemos utilizar la expresión OFFSET etiqueta para obtener el desplazamiento dentro de un segmento de una etiqueta cualquiera (DS:DX = dirección del texto).

MOV AH, 09

Especifica la Función 09.

INT 21h

Invoca el servicio (interrupción) 21h. imprimir cadena.

Esta dos instrucciones invocan la función 09 de la interrupción 21h. correspondiente al MS-DOS. Este servicio envía a la salida estándar del programa (habitualmente la pantalla) la cadena de caracteres apuntada por DS:DX.

El MS-DOS reconoce como carácter de fin de cadena el código ASCII del carácter $ (dólar), que se puede encontrar después de la cadena mensaje. Si se olvida incluir el carácter $ el MS-DOS seguirá imprimiendo lo que encuentre hasta encontrar un $.

MOV AX, 4C00h

Servicio 4Ch; valor de retorno 0.

INT 21h

Invoca el servicio 4Ch de la interrupción 21h, que retorna al sistema operativo con el ERRORLEVEL, indicando en el registro AL, en éste cado cero.

CODIGO ENDS

Cierra el segmento "CODIGO".

END ENTRADA

Fin de programa; indica punto de entrada al programa.

La directiva END marca el ensamblador el final del código fuente. El símbolo a continuación de END indica al ensamblador en qué punto debe comenzar la ejecución del programa. El ensamblador parará la información al enlazador, que incluirá esta información en la cabecera del EXE.

Directivas Simplificadas de Segmento:

Si se quieren utilizar esta directivas, es necesario primero incluir una línea con la directiva .MODEL.

El formato de esta directiva es el siguiente:

.MODEL modelo de memoria [,opciones]

Los modelos de memoria pueden ser:

- TINY: Tipo de puntero del Segmento de Código: NEAR, tipo de puntero del Segmento de datos: NEAR; permite combunar el Segmento de Código y el Segmento de Datos.
- SMALL: Tipo de puntero del Segmento de Código: NEAR, tipo de puntero del Segmento de datos: NEAR.
- MEDIUM: Tipo de puntero del Segmento de Código: FAR, tipo de puntero del Segmento de datos: NEAR.
- COMPACT: Tipo de puntero del Segmento de Código: NEAR, tipo de puntero del Segmento de datos: FAR.
- LARGE: Tipo de puntero del Segmento de Código: FAR, tipo de puntero del Segmento de datos: FAR.
- HUGE: Tipo de puntero del Segmento de Código: FAR, tipo de puntero del Segmento de datos: FAR.
- FLAT: Tipo de puntero del Segmento de Código: NEAR, tipo de puntero del Segmento de datos: NEAR; permite combinar el Segmento de Código y el segmento de Datos, pero se utiliza, exclusivamente, con el sistema operativo OS/2 2.x.

Las opciones pueden ser:

- Lenguaje: La opción "lenguaje" facilita la compatibilidad del ensamblador con lenguajes de alto nivel (interfaz de lenguaje ensamblador con lenguaje de alto nivel), por determinar la codificación interna para nombre simbólicos públicos y

externos. Puede tomar los siguientes valores: *C, BASIC, FORTRAN, PASCAL, ¿TPASCAL?, SYSCALL y STDCALL*.

- Sistema Operativos: Pueden tomar los valores: *OS_OS2 o OS_DOS*.
- Tamaño de la Pila: Pueden tomar los valores: *NEARSTACK* (Cuando el Segmento de Pila y el Segmento de Datos comparten el mismo segmento físico (SS=DS)) y FARSTACK (Cuando el Segmento de Pila y el Segmento de datos no comparten el mismo segmento físico (SS≠DS)).

Por ahora utilizaremos el modelo SMALL para la generación de programas EXE que utilicen un sólo segmento para código y otro para datos, y el modelo TINY para programa de tipo COM.

Cuando el ensamblador procesa la directiva .MODEL, prepara la definición de algunos segmentos por defecto par el código, los datos y la pila. Los nombres, alineamiento, combinación y clase de estos segmentos vienen fijados apropiadamente por el modelo especificado, por lo que no necesitamos preocuparnos de ellos.

El modelo SMALL soporta un Segmento de Código y un Segmento de Datos.

El modelo MEDIUM soporta varios Segmentos de Código y un Segmento de Datos.

El modelo COMPACT soporta un Segmento de Código y varios Segmento de Datos.

Los modelos LARGE y HUGE son equivalentes, soportan varios Segmentos de Códigos y Varios Segmentos de Datos.

Para activar el tipo de procesador con el que vamos a trabajar y las instrucciones disponibles para ese tipo de procesador, se utilizan las sentencias:

.186, .286., .386 y .486

correspondientes al tipo de procesador del ordenador con el que

estemos trabajando.

Después de esto, el ensamblador reconoce algunas directivas especiales para abrir segmentos:

.CODE para abrir el Segmento de Código.
.DATA para abrir el Segmento de Datos.
.STACK para fijar el tamaño del Segmento de Pila.

Estas directivas, además de abrir el segmento asociado, cierran el último segmento abierto, por lo que no es necesaria la directiva ENDS.

La directiva .CODE, además de abrir el Segmento de Código, genera el ASSUME adecuado para el modelo especificado.

La directiva .STACK lleva un parámetro opcional que fija el tamaño de la pila en bytes. En su defecto, el ensamblador asume un tamaño de 1024 bytes. El ensamblador y el enlazador tratan siempre de forma especial el segmento de pila, de manera que los datos del segmento de pila no aparecen en el fichero ejecutable, reduciendo por tanto el tamaño de éste y haciendo el tamaño del EXE independiente del tamaño de la pila.

La directiva END, además de las funcione vistas, cierra el último segmento abierto. Es probable que se necesite inicializar registros de segmento del mismo modo que en la forma de codificación normal, como MOV ax, datos pero la directiva simplificada de segemto no permite ver el nombre de los segmentos generados.

Para solucionar este problema, el ensamblador le permite usar los símbolos : @CODE y @DATA en lugar del nombre de los segmento de Código y Datos, respectivamente.

Así quedaría, por tanto, el programa que codificamos anterior-

mente en versión EXE usando las directivas simplificadas de segmentos:

```
    .MODEL SMALL ; Modelo de memoria SMALL: Usamos un Segmento de Código y un de Datos.
    .STACK 200h    ; Tamaño de la pila 200h=512 bytes. En el otro programa eran: DW 100h, es decir, 256 palabras, es decir, 512 bytes.
    .DATA          ; Abre el Segmento de Datos
      mensaje DB "Primer Programa","$"      ; Mensaje a imprimir
    .CODE          ; Cierra el Segmento de Datos, abre el de Código y genera el
                   ; ASSUME
    ENTRADA:
      MOV ax, @DATA      ; Valor del segmento para ".DATA".
      MOV ds, ax   ; Para acceder a "mensaje".
      MOV dx, OFFSET mensaje    ; Todo lo de abajo es igual.
      MOV ah, 09
      INT 21h
      MOV ax, 4C00h
      INT 21h
    END ENTRADA    ; Fin del programa. Cierra el segmento de Código e indica el punto de entrada al programa.
```

MACROS.

Una macro consiste en una serie de lineas a la que se asocia un nombre y que se puede repetir en cualquier punto del listado sin más que dar su nombre.

Toda macro debe tener dos partes:

1) La cabezera de la macro: Aquí se especifica el nombre de identificación de la misma, es decir, al grupo de instrucciones que engloba esa macro :

   Nombre-Macro MACRO parámetro [, parámetro, ...]

2) Texto o cuerpo de la macro: Aquí se situan las instrucciones (sentencias).
3) Fin de macro (ENDM).

El formato a utilizar sería:

```
Nombre_Macro MACRO parámetro [, parámetro,...]
  sentencias
ENDM
```

La cabecera y el fin de la MACRO, siempre vienen dados por pseudoinstrucciones o directivas,.

El cuerpo de la MACRO, viene dado por instrucciones normales de ensamblador.

Cuando el ensamblador se encuentra con una cabecera de MACRO, lo que hace es almacenarla con su cuerpo correspondiente en una tabla. Posteriormente, cuando en el programa se utilice la cabecera o el nombre de esa MACRO, el ensamblador accederá a la tabla mencionada y sustituirá, en el programa, ese nombre de MACRO por el cuerpo de

la misma.

A la operación de búsqueda en la tabla de macros se le conoce como "Llamada a la MACRO", y a la sustitución del nombre de la MACRO por el cuerpo de la misma se denomina "Expansión de la MACRO".

Según lo comentado, si nos fijamos en un programa codificado en ensamblador, tenemos dos versiones de programa. Una de ellas escrita directamente y otra mediante macros, con lo que nos podíamos encontrar que después de la expansión de las macros las dos versiones de programa son iguales.

El empleo de macros no debemos confundirlo con el empleo de procedimientos, ya que una llamada a una macro sustituye su nombre por el cuerpo de la misma, en cambio, una llamada a procedimientos lo único que realiza es un enlace con otro programa. Las macros se suelen incluir todas dentro de ficheros específicos. En cada uno de los ficheros situaremos las macros que tengan alguna relación entre sí.

Nos podemos encontrar con macros que en la realización de una tarea puedan utilizar datos distintos.

Para no tener que definir una macro para cada dato distinto que tengamos que utilizar, se debe permitir la utilización de "Parámetro Formales", entendiéndose que estos datos serían los nombres que utilizaremos en el cuerpo de la MACRO cuando la definiésemos por primera vez.

Estos parámetros van situados en la cabecera de la definición de la macro.

En la llamada a la macro sería donde situaríamos los "Parámetros Reales", con los que va a trabajar la macro. Cuando se hace una expansión de la macro los parámetros formales se sustituyen por los parámetros reales.

INCLUDE. Esta directiva nos proporciona la posibilidad de introducir, dentro de nuestro programa un conjunto de instrucciones que están situadas en otro fichero.

Estas instrucciones se sustituyen justamente en la posición en la que se encuentra la directiva. Dicha directiva se usa dentro de cualquiera de los segmentos, pero su utilización corriente es dentro del Segmento de Código.

Se suele utilizar para poder incluir dentro de un programa los ficheros de MACROS.

Su formato es: INCLUDE Nombre_Fichero

Si el Fichero a incluir en el programa (Nombre_Fichero), no se encuentra en el mismo directorio del programa o en la misma unidad, habrá que indicarle en la instrucción, el camino de acceso para encontrar ese Fichero.

Ejemplo: En un disco tenemos un fichero con una serie de MACROS y que se llama:

MACROS.ASM:

```
Listado de MACROS.ASM
retorno MACRO
           MOV ah, 4ch
          INT 21h
ENDM
display  MACRO cadena
        MOV dx, OFFSET cadena
        MOV ah, 09h
        INT 21h
ENDM
  leer_teclado MACRO
           MOV ah, 08h
           INT 21h
```

```
ENDM
```

En nuestro ordenador estamos realizando nuestro programa:

```
PILA SEGMENT STACK 'STACK'
      dw 100h dup (?)
PILA ENDS
DATOS SEGMENT 'DATA'
      paso db "MIOOO"
      control db 5 dup (?)
      men1 db "Clave de acceso:".13,10,"$"
      men2 db "Bienvenido","$"
DATOS ENDS
CODIGO SEGMENT 'CODE'
      ASSUME CS:  CODIGO, DS: DATOS, SS: PILA
EMPEZAR:
MOV ax, DATOS
MOV ds, ax
INCLUDE A:\ MACROS,ASM   ; Aquí le decimos al programa que incluya las macros
del porgrama  que se encuentra en el fichero MACROS:ASM que se encuentra en un disco
en la unidad A de nuestro ordenador.
DOS:
 MOV si, 0
 display men1     ; "display" es una MACRO que tiene parámetro.
 MOV cx, 5
UNO:
     leer teclado  ; Otra MACRO. Esta macro no tiene parámetros.
     MOV control[SI], al
     INC si
     LOOP  UNO
     MOV si, 0
     MOV cx, 5
TRES:
```

```
        MOV al, control[SI]
        CMP al, paso[SI]
        JNE DOS
        INC SI
        LOOP TRES
        display men2   ; Otra MACRO
        retorno              ; Otra MACRO sin parámetros
CODIGO ENDS
        ENDS EMPEZAR
```

Con las directivas simplificadas las macros funcionan igual.

LOCAL: Directiva que le indica al ensamblador que dentro de la macro van a existir una serie de etiquetas que deben de ser identificadas de una forma especial al realizar la expansión de la macro y siempre considerando la relación que existe con la tabla de símbolos. Con ello se evita las definiciones múltiples de estas etiquetas.

El formato es el siguiente:

LOCAL etiquetas (separadas por comas)

Esta directiva debe situarse, en el caso de que exista después de la cabecera de la macro.

Ejemplo:

```
esperar MACRO numero
        LOCAL SEGUIR
        MOV cx, numero
seguir:
```

```
                    LOOP seguir
                ENDM
```

Otra cosa aconsejable que se puede hacer en referencia a las macros es guardar previamente en la pila el valor de los registros que vayamos a usar dentro de la macro y después, cuando acaben las instruciones de la macro, volver a retornárselos.

Ejemplo:

```
display MACRO cadena
        PUSH dx
        PUSH ax
        MOV dx, offset cadena
        MOV ax, 09h
        INT 21h
        POP ax
        POP dx
ENDM
```

Ejercicios:

1.- Disponemos de una matriz almacenada en una serie de bytes en memoria. Dicha matriz está estructurada mediante 4 filas y 3 columnas. Realizar la suma de los elementos de esta matriz recorriéndola fila a fila.

2.- Disponemos de una matriz de 4 filas y 3 columnas ocupando cada elemento de la matriz un byte. Calcular la suma de los elementos de esta matriz columna a columna,

3.- Disponemos de una matriz almacenada en palabras de memoria cuya estructura de 4 filas y 4 columnas. Se pide realizar la suma de sus dos diagonales.

4.- Disponemos de una matriz almacenada en palabras de memoria que posee una estructura de 4 filas y 3 columnas. Sumar los elementos que ocupen posiciones impares, a continuación, sumar los que ocupen posiciones pares, y la suma de elementos impares es mayor que la de los pares, efectuar la resta "impares - pares", y si no es así, efectuar la resta "pares - impares".

5.- Se trata de teclear un máximo de 40 caracteres que irán apareciendo en pantalla. Si durante el tecleo no deseamos seguir con el mismo, podemos romper la secuencia pulsando el ENTER. Una vez tecleados los 40 caracteres, aparecerá en pantalla un literal que nos indicará que introduzcamos una palabra a localizar dentro de la secuencia inicial tecleada. Una vez localizada la palabra se nos deberá indicar cuantas veces aparece esta palabra dentro del texto

inicial tecleado.

PROCEDIMIENTOS.

El uso de los procedimientos va a estar marcado por elementos principales:

· **CALL:** (Llamada a un procedimiento). Esta instrucción transfiere el control a otro punto de la memoria, cargado IP (cuando el procedimiento llamado está dentro del mismo segmento (NEAR)), y eventualmente CS (cuando el procedimiento llamado está en otro segmento (FAR)), con nuevos valores. La dirección actual (CS:IP o solamente IP, en función de si la instrucción altera CS o no) se introduce en la pila para permitir un posterior retorno. El destino del salto puede estar indicado tras el código de operación de la instrucción o puede obtenerse indirectamente de un registro o posición de memoria especificada en la instrucción. Hay varios tipos de llamadas (a etiquetas, a variables, etc), nosotros vamos a tratar las llamadas a etiquetas.

Formato:

CALL [tipo_salto] destino

"tipo_salto" indica si la llamada es a un procedimiento cercano (entoces su valor es NEAR) o lejano (su valor es FAR), y "destino" es el nombre del procedimiento al que vamos a llamar.

· **PROC:** Indica el comienzo de un procedimiento (un procedimiento es un bloque de instrucciones que sirven para realizar una tarea determinada y que pueden invocarse desde varios puntos del programa. Puede considerarse como una subrutina).

Formato:

```
Nombre_Procedimiento PROC atributo
                     sentencias
                     RET
Nombre_Procedimiento ENDP
```

El "atributo" puede ser NEAR o FAR dependiendo de si el salto es dentro del mismo segmento (NEAR) o no (FAR).

Los valores de IP (NEAR) o CS:IP (FAR) se guardan en la pila cuando se produce una llamada a un procedimiento. Para que estos valores puedan ser restaurados después de la ejecución del procedimiento y elñ programa sigua con su ejecución normal se ejecuta la instricción RET (retornar valores).

Ejemplo:

```
PILA SEGMENT STACK 'STACK'
     dw 100h dup (?)
PILA ENDS
DATOS SEGMENT 'DATA'
     mensaje db "Primer Programa:","$"
DATOS ENDS
CODIGO SEGMENT 'CODE'
     ASSUME CS:  CODIGO, DS: DATOS, SS: PILA
     INCLUDE MACROS.ASM
ENTRADA:
     MOV ax, DATOS
     MOV ds, ax
     display mensaje
     retorno
     ;
     ; **** Procedimientos ****
     ;
     retorno PROC
```

```
            PUSH ax
            MOV ax, 4C00h
            INT 21h
            POP ax
            RET
      retorno ENDP
CODIGO ENDS
      END ENTRADA
```

(Fichero MACROS.ASM)

```
      display MACRO mem
              PUSH dx
              PUSH ax
              MOV dx, offset mem
              MOV ah, 09h
              INT 21h
              POP ax
              POP dx
      ENDM
```

COMPILACION.

Para conseguir un programa ejecutable, el programa fuente escrito en lenguaje ensamblador, entre otras tareas, habrá que ensamblarlo y linkarlo (LINKER).

Para ello el MASM 6.0 nos suministra una instrucción que nos realiza estas dos funciones: ML.

ML es el ensamblador que nos suministra MASM 6.0 y se encuentra (normalmente) en el subdirectorio: \MASM\BIN. El ensamblaje de un programa fuente escrito en ensamblador se realiza de la siguiente forma:

```
ML [\opción] Nombre_Programa.ASM
```

Si el fichero no se encuentra en el mismo directorio que el ensamblador, hay que poner el PATH donde se encuentra el fichero fuente. Para visualizar las opciones con las que cuenta el ensamblador basta con ejecutar:

```
ML /?
```

Algunas de estas opciones son:

- /AT: Habilita el modelo TINY (para ficheros .COM).
- /c: Ensambla sin Linkar (LINK).
- /Zi: Añade información simbólica para el BEBUG.
- /Zm: Habilita la compatibilidad con MASM 5.10.

Si se quiere ejecutar el MASM 6.0 desde un entorno gráfico, se puede ejecutar un programa que hay en el directorio \MASM\BIN y que se llama PWB. Si fuera necesario la ejecución de LINK este se encuentra en el directorio \MASM\BINB.