

Índice

- 0. Introducción
- 1. Arquitectura Del 8086
 - 1.1. Registros del 8086. Segmentación
 - 1.1.1. Registros de propósito general
 - 1.1.2. Registros de segmento
 - 1.1.2.1. Segmentación y direccionamiento
 - 1.1.3. Registro apuntador de instrucciones (IP).
 - 1.1.4. Registros apuntadores (SP y BP).
 - 1.1.5. Registros índice (SI y DI).
 - 1.1.6. Registro de banderas, flags, o registro de estado (FL).
 - 1.2. Modos de direccionamiento
 - 1.2.1. Registros de segmento por defecto
 - 1.2.2. Direccionamiento de localidades de memoria
 - 1.3. Pila (stack).
- 2. Conjunto de instrucciones.
- 3. Programación del PC sobre dos
 - 3.1. Directivas del ensamblador
 - 3.1.1. Directivas para declaración y manejo de segmentos.
 - 3.1.2. Directivas para definición de datos.
 - 3.1.3. Etiquetas y variables
 - 3.1.4. Constantes numéricas
 - 3.1.5. La Directiva EQU.
 - 3.2. Esquema de un programa
 - 3.3. Ensamblado, *linkado* y montado
 - 3.4. Procedimientos
 - 3.4.1. Paso de parámetros a procedimientos
 - 3.4.1.1. Registros
 - 3.4.1.2. Memoria Intermedia
 - 3.4.1.3. Pila
 - 3.5. Macros
 - 3.5.1. Paso de parámetros a las macros
 - 3.6. Interrupciones en el PC
 - 3.6.1. Tratamiento de interrupciones
 - 3.6.2. Interrupciones sectorizadas
 - 3.6.3. Tipos de interrupciones.
 - 3.6.4. Capturar una interrupción
 - 3.7. La memoria de vídeo
 - 3.7.1. El sistema de visualización
 - 3.7.2. El modo texto.
 - 3.7.3. El modo gráfico
 - 3.8. Programas .COM
 - 3.9. Programas residentes

0.- INTRODUCCIÓN.

Este documento, más que un manual de programación en ensamblador del procesador de Intel 8086, es una manual de prácticas para los alumnos de Arquitectura e Computadores, ya que se han omitido intencionalmente muchos detalles y características de este lenguaje. Además sólo se hace mención a la programación en MS-DOS y con el ensamblador TASM de Borland.

1.- ARQUITECTURA DEL 8086.

En 1978 Intel presentó el procesador **8086**. Poco después, desarrolló una variación del 8086 para ofrecer un diseño ligeramente más sencillo y compatibilidad con los dispositivos de entrada/salida de ese momento. Este nuevo procesador, el **8088**, fue seleccionado por IBM para su **PC** en 1981. Ambos poseen una arquitectura interna de 16 bits y pueden trabajar con operandos de 8 y 16 bits; una capacidad de direccionamiento de 20 bits, es decir, existe la posibilidad de direccionar 2^{20} posiciones de memoria (bytes), lo que equivale a 1Mb de memoria principal y comparten el mismo conjunto de 92 instrucciones.

La estructura interna del 8086 puede verse en la Figura 2. El 8086 se divide en dos unidades lógicas: una **unidad de ejecución (EU)** y una **unidad de interfaz del bus (BIU)**. El papel de la EU es ejecutar instrucciones, mientras que la BIU envía instrucciones y datos a la EU. La EU posee una unidad aritmético-lógica, una unidad de control y 10 registros. Permite ejecutar las instrucciones, realizando todas las operaciones aritméticas, lógicas y de control necesarias.

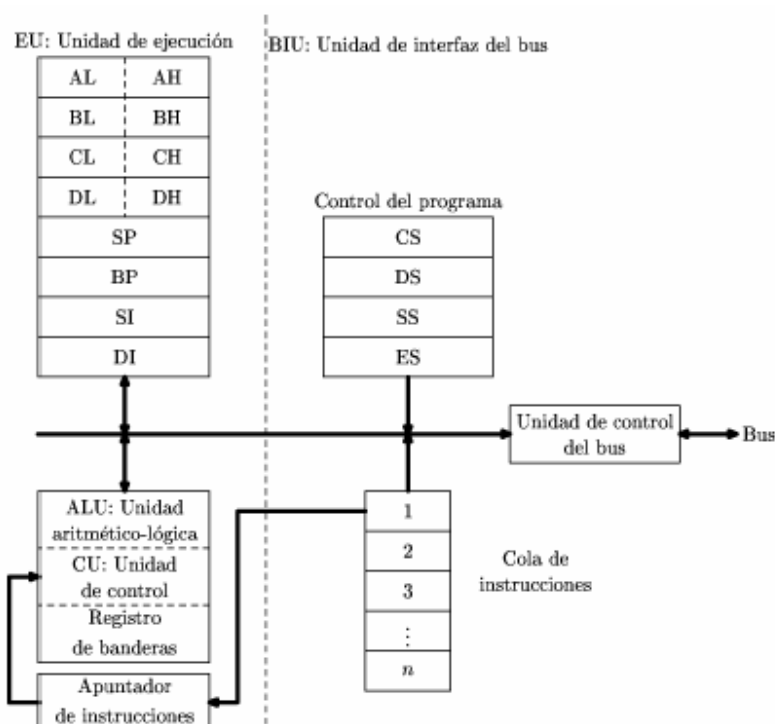


Figura 2. *Arquitectura interna del 8086.*

1.1.- Registros del 8086. Segmentación

Los *registros* del procesador tienen como misión fundamental almacenar las posiciones de memoria que van a sufrir repetidas manipulaciones, ya que los accesos a memoria son mucho más lentos que los accesos a los registros. El 8086 dispone de 14 registros de 16 bits que se emplean para controlar la ejecución de instrucciones, direccionar la memoria y proporcionar capacidad aritmética y lógica. Cada registro puede almacenar datos o direcciones de memoria. Los registros son direccionables por medio de un nombre. Por convención los bits de un registro se numeran de derecha a izquierda:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0	1	0	0	0	1	1	1	0

Los diferentes registros del 8086 se clasifican en: registros de propósito general o de datos, registros de segmento, registro apuntador de instrucciones (IP), registros apuntadores (SP y BP), registros índice (SI y DI) y registro de banderas, FLAGS o registro de estado (FL).

AX	SP	CS	IP
BX	BP	DS	FLAGS o FL
CX	SI	SS	Registro puntero de instrucciones;
DX	DI	ES	y Registro de banderas, FLAGS o de estado (FL)
Registros de propósito general o de datos	Registros punteros y Registros índice	Registros de segmento	

Figura 3. Registros del 8086/88 y del 80286.

1.1.1.- Registros de propósito general.

Se utilizan para cálculo y almacenamiento de propósito general. Los programas leen datos de memoria y los dejan en estos registros, ejecutan operaciones sobre ellos, y guardan los resultados en memoria. Hay cuatro registros de propósito general que, aparte de ser usados a voluntad por el programador, tienen fines específicos:

Registro AX	Este registro es el acumulador principal, implicado en gran parte de las operaciones de aritméticas y de E/S.
Registro BX	Recibe el nombre de registro base ya que es el único registro de propósito general que se usa como un índice en el direccionamiento indexado. Se suele utilizar para cálculos aritméticos.
Registro CX	El CX es conocido como registro contador ya que puede contener un valor para controlar el número de veces que se repite una cierta operación.
Registro DX	Se conoce como registro de datos. Algunas operaciones de E/S requieren su uso, y las operaciones de multiplicación y división con cifras grandes suponen que el DX y el AX trabajando juntos

Los registros de propósito general se pueden direccionar como una palabra o como un byte. El byte de la izquierda es la parte alta o byte más significativo y el byte de la derecha es la parte o byte menos significativo.

RX	
RH	RL

Siguiendo esta nomenclatura, es posible referirse a cada uno de los dos bytes, byte de orden alto o más significativo y byte de orden bajo o menos significativo, de cada uno de estos registros. Por ejemplo: AH es el byte más significativo del registro AX, mientras que AL es el byte menos significativo.

Así, por ejemplo, si AX contiene 00FFh, AH=00h (su parte alta) y AL=FFh (su parte baja); si lo incrementamos en 1, AX pasa a valer 0100h (lógicamente), y con ello AH=01h, AL=00h. Si en lugar de incrementar en 1 AX lo hacemos con AL (byte inferior), AH se quedará como estaba y AL será FFh+01h=00h, es decir, AX=0000h (vamos, que cuando se manipula una parte del registro la otra no se ve afectada en absoluto, pero, obviamente, sí el registro completo)

Uno puede mover los datos de unos registros a otros con prácticamente total libertad. También podremos realizar operaciones sobre ellos, como sumar el contenido de BX y DX para guardar el resultado en DX, y cosas así. La primera restricción al respecto (y bastante razonable) es que los operandos tendrán que ser del mismo tamaño (no podremos sumar BX con DH, por ejemplo).

1.1.2.- Registros de Segmento.

Los registros de segmento son registros de 16 bits que constituyen la implementación física de la arquitectura segmentada del 8086. Para entender para qué sirven estos registros primero debemos explicar brevemente en que consiste la segmentación.

1.1.2.1.- Segmentos y direccionamiento

Para acceder a una dirección de memoria lo lógico sería colocar dicha dirección completa en un registro y acceder mediante este. Sin embargo el 8086 tiene 20 líneas en el bus de direcciones (20 bits) mientras que sus registros son de 16 bits, de forma que siguiendo esta filosofía solo podríamos direccionar 64KBytes (2^{16}) del mega posible (2^{20}).

La solución a esto consistió en usar dos registros de 16 bits, pero no dispuestos consecutivamente, ya que obtendríamos una dirección de 32 bits pasándonos del rango de memoria disponible, sino que un registro especifica un bloque de memoria, el segmento, y el otro un desplazamiento u *offset* dentro de este bloque. Puesto que utilizamos registros de 16 bits tendremos 64K posibles segmentos de 64 KBytes cada uno.

Para direccionar cualquier posición de memoria, el procesador combina la dirección del segmento que se encuentra en un registro de segmento con un valor de desplazamiento, para calcular la dirección real de 20 bits.

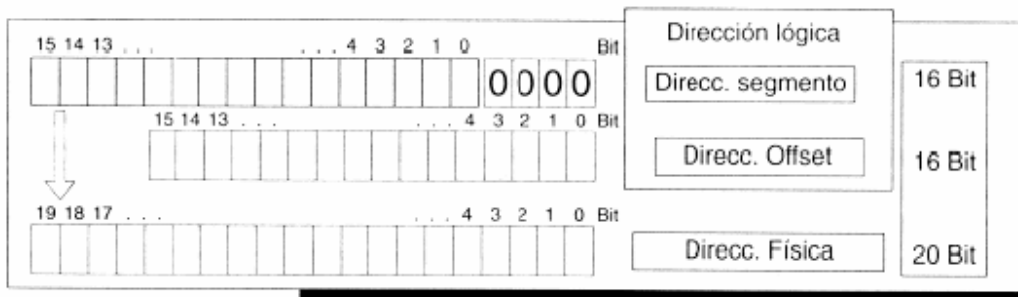


Figura 4. Construcción de una dirección de memoria en base a una dirección de segmento y una dirección de offset.

Como vemos en la figura anterior, lo primero que se hace es añadir 4 bits a 0 al valor que se encuentra en el registro de segmento, lo cual es equivalente a multiplicar dicho valor por 16. Es por esto que cualquier segmento empieza siempre en una dirección de memoria (de 20 bits) múltiplo de 16, esto es, el segmento 0 comenzaría en la dirección 00000h y acabaría en la 0FFFFh; el segmento 1 comenzaría en la dirección 00010h (16 en hexadecimal) y acabaría en la 1000Fh; etc

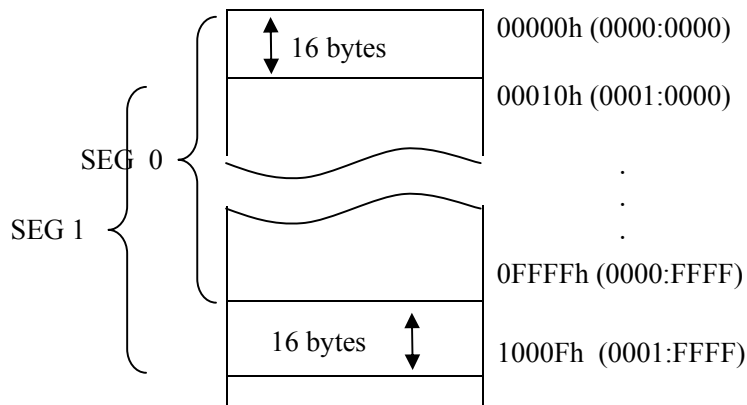


Figura 5. Segmentación de la memoria

Por tanto, como podemos ver en la figura, los distintos segmentos se solapan, esto es, hay pares segmento-*offset* que apuntan a la misma dirección de memoria.

Por ejemplo, dado el segmento 045EH y un desplazamiento de 0032H:

Dirección del segmento:	045E0H
Desplazamiento dentro del segmento:	+0032H
Dirección real:	<u>04612H</u>

y, dado el segmento 045FH y un desplazamiento de 0022H:

Dirección del segmento:	045F0H
Desplazamiento dentro del segmento:	+0022H
Dirección real:	<u>04612H</u>

Ahora bien, un programa puede tener uno o varios segmentos, los cuales pueden comenzar en casi cualquier lugar de la memoria, variar en tamaño y estar en cualquier orden, pero en general, tendrá un segmento para el código, otro para los datos y otro para la pila. Para apuntar a estos segmentos es para lo que sirven los registros de segmento:

Registro CS	Registro Segmento de Código. Establece el área de memoria dónde está el programa durante su ejecución.
Registro DS	Registro Segmento de Datos. Especifica la zona donde los programas leen y escriben sus datos.
Registro SS	Registro Segmento de Pila. Permite la colocación en memoria de una pila, para almacenamiento temporal de direcciones y datos.
Registro ES	Registro Segmento Extra. Para acceder a un segmento distinto de los anteriores sin necesidad de modificar los otros registros de segmento.

El segmento de código contiene las instrucciones que serán ejecutadas, mientras que el segmento de datos contiene los datos a los que las instrucciones hacen referencia.

1.1.3.- Registro Apuntador de Instrucciones (IP).

Se trata de un registro de 16 bits que contiene el desplazamiento de la dirección de la siguiente instrucción que se ejecutará. Está asociado con el registro CS en el sentido de que IP indica el desplazamiento de la siguiente instrucción a ejecutar dentro del segmento de código determinado por CS:

Dirección del segmento de código en CS:	25A40H
Desplazamiento dentro del segmento de código en IP:	+ 0412H
Dirección de la siguiente instrucción a ejecutar:	<u>25E52H</u>

1.1.4.- Registros Apuntadores (SP y BP).

Los registros apuntadores están asociados al registro de segmento de pila, SS, y permiten acceder a los datos almacenados en la pila. Representan un desplazamiento respecto a la dirección determinada por SS:

Registro SP	Proporciona un valor de desplazamiento que se refiere a la palabra actual que está siendo procesada en la pila.
Registro BP	Facilita la referencia a los parámetros de las rutinas, los cuales son datos y direcciones transmitidos vía la pila.

En general, el registro IP indica el desplazamiento de la instrucción actual que es ejecutada dentro del segmento de código; el registro SP indica el desplazamiento dentro del segmento de pila, apuntando al último elemento introducido en la pila; y un operando (que sea una referencia a memoria) de una instrucción indica un desplazamiento en el segmento de datos del programa.

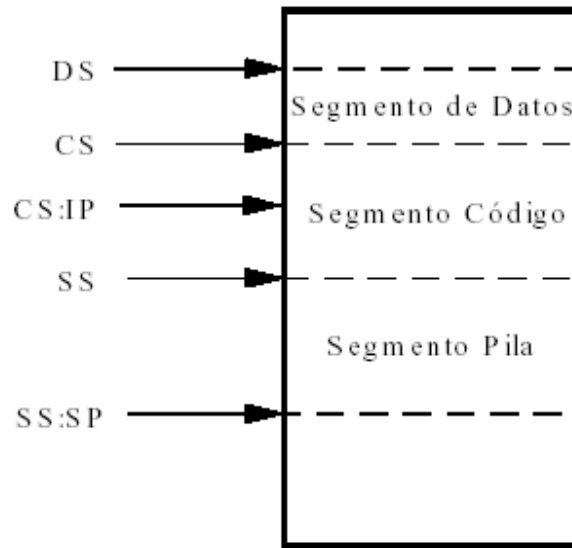


Figura 6 Esquema de direccionamiento para un programa.

1.1.5.- Registros Índice (SI y DI).

Los registros índice se utilizan fundamentalmente en operaciones con cadenas y para direccionamiento indexado:

Registro SI	Registro índice fuente requerido en algunas operaciones con cadenas de caracteres. Este registro está asociado con el registro DS.
Registro DI	Registro índice destino requerido también en determinadas operaciones con cadenas de caracteres. Está asociado al registro DS o ES.

Los registros de índice se pueden usar como registros de datos sin problemas para sumas, movimiento de datos... no así los de segmento, que tienen fuertes limitaciones. Cuando se diga "un registro" como operando de una instrucción, eso incluye en principio cualquier registro menos los de segmento.

1.1.6.- Registro de banderas, FLAGS, o registro de estado (FL).

Es un registro de 16 bits, pero sólo se utilizan nueve de ellos. Sirven para indicar el estado actual de la máquina y el resultado del procesamiento. La mayor parte de las instrucciones de comparación y aritméticas modifican este registro. Algunas instrucciones pueden realizar pruebas sobre este registro para determinar la acción siguiente (como las instrucciones de salto)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C

Los bits 0, 2, 4, 6, 7 y 11 son indicadores de condición que reflejan los resultados de las operaciones del programa; los bits 8 al 10 son indicadores de control que, modificados por el programador, sirven para controlar ciertos modos de procesamiento, y el resto no se utilizan. El significado de cada uno de los bits es el siguiente:

OF	Bit de Overflow o desbordamiento. Indica desbordamiento de un bit de orden alto (más a la izquierda), después de una operación aritmética.
DF	Bit de Dirección. Designa la dirección, creciente (0) o decreciente (1), en operaciones con cadenas de caracteres.
IF	Bit de Interrupción. Indica que una interrupción externa, como la entrada desde el teclado, sea procesada o ignorada.
TF	Bit de Trap o Desvío. Procesa o ignora la interrupción interna de <i>trace</i> (procesamiento paso a paso).
SF	Bit de Signo. Indica el valor del bit más significativo del registro después de una operación aritmética o de desplazamiento.
ZF	Bit Cero. Se pone a 1 si una operación produce 0 como resultado
AF	Bit de Carry Auxiliar. Se pone a 1 si una operación aritmética produce un acarreo del bit 3 al 4. Se usa para aritmética especializada (ajuste BCD).
PF	Bit de Paridad. Se activa si el resultado de una operación tiene paridad par.
CF	Bit de Acarreo. Contiene el acarreo de una operación aritmética o de desplazamiento de bits.

1.2.- Modos de direccionamiento.

Las operaciones se hacen entre registros o registros y memoria, pero **nunca** entre memoria y memoria (salvo algunas operaciones con cadenas de caracteres). Los modos de direccionamiento determinan el lugar en que reside un operando, un resultado o la siguiente instrucción a ejecutar, según el caso, en memoria.

Para ilustrar los distintos modos de direccionamiento, vamos a usar la instrucción

```
MOVE destino, fuente
```

que carga el contenido del operando fuente en el destino

1.- **INMEDIATO**. El operando aparece especificado directamente en la instrucción.

Ejemplo: El operando fuente en MOVE AX, 789AH => copia el valor 789AH en AX

2.- **MODO REGISTRO**. El operando es un registro.

Ejemplo: Los operandos en MOV AX, CX => se copia en AX el contenido de CX

3.- **DIRECTO ABSOLUTO A MEMORIA**. El operando es una dirección de memoria a la que se quiere acceder.

Ejemplo: El operando fuente en MOV AX, [078AH] => copia en AX el contenido de la dirección DS:078AH

4.- **DIRECTO RELATIVO A UN REGISTRO BASE**. El operando es una dirección de memoria a la que se desea acceder, y se calcula mediante un registro base. Este registro base será el BX o el BP según deseemos trabajar con el segmento DS o SS respectivamente.

Ejemplo: El operando fuente en MOV AX, [BX + 2] => copia en AX el contenido de la dirección DS:BX+2

5.- DIRECTO RELATIVO A UN REGISTRO INDICE. El operando es una dirección de memoria a la que se desea acceder, y se calcula en base a un registro índice. Este registro índice será el SI o el DI.

Ejemplo: El operando fuente en MOV AX, [SI + 10] => copia en AX el contenido de la dirección DS:SI+10

Si el desplazamiento no existe en los dos últimos modos, hablamos de direccionamiento indirecto por registro base o por registro índice.

6.- INDEXADO A PARTIR DE UNA BASE. El operando es una dirección de memoria a la que se desea acceder, y se calcula en base a un registro base y un registro índice. Hay cuatro configuraciones posibles: el registro base puede ser el BX o el BP, y el registro índice puede ser el SI o el DI.

Ejemplo: El operando destino en MOV [BP + DI + 2], AX => copia en la dirección SS:BP+DI+2 el contenido de AX

Estos tres últimos modos de direccionamiento pueden resumirse en:

[BX | BP] + [SI | DI] + [Desplazamiento]

Si aparece BP el registro de segmento por defecto es SS, en caso contrario es DS.

Nota: Los modos de direccionamiento del 3 al 6 pueden estar precedidos de un registro de segmento: MOV AX, ES: [BX + SI + 6].

1.2.1.- Registros de Segmento por defecto.

Existen unos segmentos asociados por defecto a los registros de desplazamiento (IP, SP, BP, BX, DI y SI). Sólo es necesario declarar el segmento cuando no coincide con el asignado por defecto.

La Figura 7 muestra los segmentos que se emplean por defecto para cada uno de los registros que se encuentran en la primera columna:

	CS	SS	DS	ES
IP	Sí	Nunca	Nunca	Nunca
SP	Nunca	Sí	Nunca	Nunca
BP	Prefijo	Por Defecto	Prefijo	Prefijo
BX	Prefijo	Prefijo	Por Defecto	Prefijo
SI	Prefijo	Prefijo	Por Defecto	Prefijo
DI	Prefijo	Prefijo	Por Defecto	Por Defecto Cadenas

Figura 7. Segmentos por defecto.

1.2.2.- Direccionamiento de Localidades de Memoria.

El 8086 posee un bus de datos de 16 bits y por tanto manipula cantidades de esta longitud (llamadas palabras). Cada palabra a la que se accede consta de dos bytes, un byte de orden alto o más significativo y un byte de orden bajo o menos significativo. El sistema almacena en memoria estos bytes de la siguiente manera: el byte menos significativo en la dirección baja de memoria y el byte más significativo en la dirección alta de memoria, tal y como se muestra en la Figura 8.

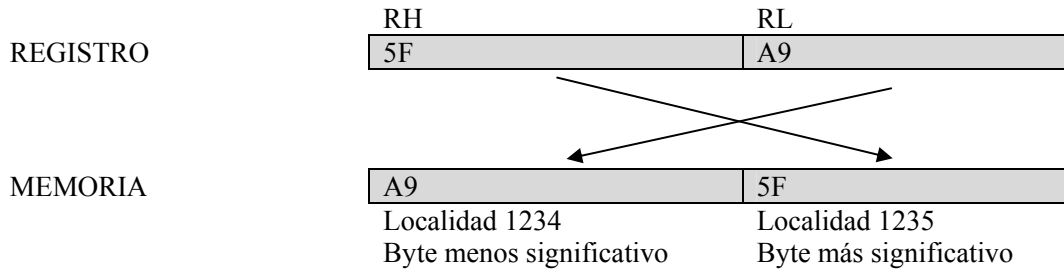
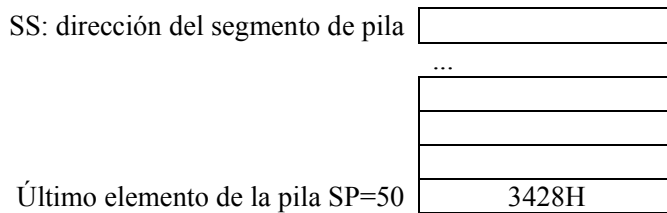


Figura 8. *Direccionamiento de Localidades de Memoria.*

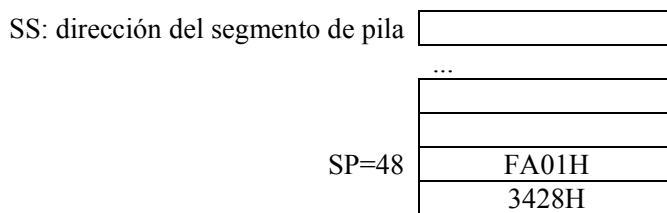
1.3.- PILA (STACK).

La pila no es más que un área para el almacenamiento temporal de direcciones y datos de tipo LIFO (Last In First Out), esto es, el último dato en entrar en la pila es el primero en salir. El registro SS indica la dirección de inicio de la pila, mientras que el registro SP apunta al byte siguiente al último que pertenece a la pila. La pila se caracteriza porque empieza a almacenar datos en la localidad más alta y avanza hacia abajo en la memoria, esto es, cada vez que se introduce un dato en la pila se decrementa SP, y cuando se saca, se incrementa. Existen instrucciones especiales para introducir (PUSH) y sacar (POP) datos de la pila que se encargan de modificar apropiadamente el registro SP. Los datos que se introduzcan en la pila **siempre son de dos bytes**.

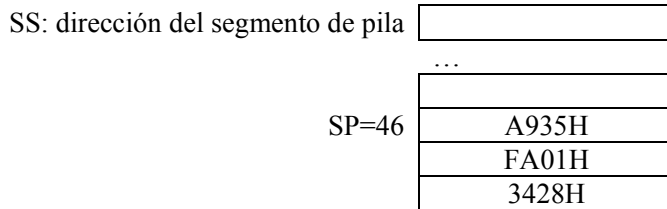
Ejemplo: Introducimos en la pila los registros AX y BX que contienen FA01H y 35A9H respectivamente. Suponemos que el registro SP contiene 50 bytes.



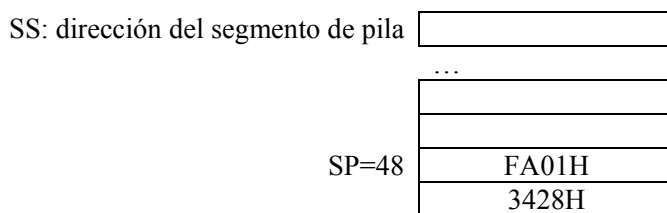
2. PUSH AX: Se introduce en la pila AX ⇒ se decrementa SP y se introduce el dato



3. PUSH BX: Se introduce en la pila BX \Rightarrow se decrementa SP y se introduce el dato



4. POP BX: Se extrae de la pila un dato y se introduce en BX \Rightarrow se extrae el dato y se incrementa SP.



2.- CONJUNTO DE INSTRUCCIONES.

En este apartado vamos a describir las operaciones más importantes del 8086 mediante la siguiente clasificación:

TRANSFERENCIA DE DATOS
 ARITMÉTICAS
 LÓGICAS Y MANEJO DE BITS
 TRANSFERENCIA DE CONTROL
 MANEJO DE CADENAS
 ENTRADA/SALIDA

El formato general de instrucción es el que sigue:

[etiqueta] operación [operando1 [,operando2]]

La operación puede tener cero, uno o dos operandos explícitos. Cuando tiene dos, el primero es el operando destino y el segundo el operando fuente. Por lo general, ambos operandos deben ser del mismo tamaño: no podemos sumar un registro de 16 bits con uno de 8 bits, por ejemplo. El contenido del operando fuente no es modificado como resultado de la ejecución de la instrucción. Los operandos pueden ser de tres tipos: registros, valores inmediatos y direcciones de memoria (entre corchetes si especificamos valores concretos, como en [3BFAh], o en direccionamiento relativo), aunque los operandos fuente y destino **no pueden ser dos direcciones de memoria** (salvo algunas operaciones con cadenas)

2.1. Instrucciones de Transferencia de Datos.

Su misión es intercambiar la información entre los registros y las posiciones de memoria. Las operaciones de este tipo más relevantes son:

Operación	MOV: Mover datos
<i>Descripción</i>	Transfiere datos entre dos registros o entre un registro y memoria, y permite llevar datos inmediatos a un registro o a memoria.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	MOV {registro/memoria},{registro/memoria/inmediato}
<i>Ejemplo</i>	MOV AX, 54AFH

Las operaciones MOV **no permitidas** son de memoria a memoria, inmediato a registro de segmento y de registro de segmento a registro de segmento. Para estas operaciones es necesario utilizar más de una instrucción.

Operación	XCHG: Intercambiar datos
<i>Descripción</i>	Intercambia datos entre dos registros o entre un registro y memoria.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	XCHG {registro/memoria},{registro/memoria}
<i>Ejemplo</i>	XCHG AL,AH

No pueden utilizarse registros de segmento como operandos, ni tampoco dos direcciones de memoria.

Operación	PUSH: Guardar en la pila
<i>Descripción</i>	Guarda en la pila una palabra para su uso posterior. SP apunta al tope de la pila; PUSH decrementa SP en 2 y transfiere la palabra a SS:SP
<i>Banderas</i>	No las afecta
<i>Formato</i>	PUSH {registro/memoria/inmediato(sólo 286 o posteriores)}
<i>Ejemplo</i>	PUSH DX

Operación	POP: Sacar una palabra de la pila
<i>Descripción</i>	Saca de la pila una palabra previamente guardada y la envía a un destino especificado. SP apunta al tope de la pila; POP la transfiere al destino especificado e incrementa SP en 2.
<i>Banderas</i>	No las afecta
<i>Formato</i>	POP {registro(excepto CS, se debe usar RET)/memoria}
<i>Ejemplo</i>	POP BX

Operación	LEA: Cargar dirección efectiva
<i>Descripción</i>	Carga una dirección cercana en un registro
<i>Banderas</i>	No las afecta
<i>Formato</i>	LEA registro, memoria
<i>Ejemplo</i>	LEA AX, MEN1

La instrucción LEA AX, MEN1 es equivalente a MOV AX, OFFSET MEN1. Sin embargo no siempre ocurre así, por ejemplo, LEA DX, [SI + datos] corresponde a MOV DX, OFFSET datos, mas la instrucción ADD DX, SI.

2.2.- Instrucciones Aritméticas.

Sirven para llevar a cabo operaciones aritméticas manipulando los registros y las posiciones de memoria:

Operación	ADD: Sumar números binarios
Descripción	Suma números binarios desde la memoria, registro o inmediato a un registro, o suma números en un registro o inmediato a memoria. Los valores pueden ser un byte o una palabra.
Banderas	AF, CF, OF, PF, SF y ZF.
Formato	ADD {registro/memoria},{registro/memoria/inmediato}
Ejemplo	ADD DL,AL

Operación	SUB: Restar números binarios
Descripción	Resta números binarios en un registro, memoria o inmediato de un registro, o resta valores en un registro o inmediato de memoria.
Banderas	AF, CF, OF, PF, SF y ZF.
Formato	SUB {registro/memoria},{registro/memoria/inmediato}
Ejemplo	Ejemplo SUB AL,CL

Operación	DEC: Disminuye en uno
Descripción	Disminuye 1 de un byte o una palabra en un registro o memoria.
Banderas	AF, OF, PF, SF y ZF.
Formato	DEC {registro/memoria}
Ejemplo	DEC DL

Operación	INC: Incrementa en uno
Descripción	Incrementa en uno un byte o una palabra en un registro o memoria.
Banderas	AF, OF, PF, SF y ZF.
Formato	INC {registro/memoria}
Ejemplo	INC [1B15h]

Operación	MUL: Multiplicar sin signo
Descripción	Multiplica dos operandos sin signo.
Banderas	CF y OF (AF, PF, SF y ZF quedan indefinidas). CF = OF = 1 => AH ≠ 0 ó DX ≠ 0.
Formato	MUL {registro/memoria} (Ver tabla)
Ejemplo	--

Operando 1	Operando 2	Producto	Ejemplo
AL	R/M 8 bits	AX	MUL BL
AX	R/M 16 bits	DX:AX	MUL BX

Operación	IMUL: Multiplicar con signo (enteros)
Descripción	Multiplica dos operandos con signo. IMUL trata el bit de más a la izquierda como

	el signo
<i>Banderas</i>	CF y OF (AF, PF, SF y ZF quedan indefinidas). CF = OF = 1 => AH ≠ 0 Ó DX ≠ 0.
<i>Formato</i>	IMUL {registro/memoria} (Ver tabla)
<i>Ejemplo</i>	--

Operando 1	Operando 2	Producto	Ejemplo
AL	R/M 8 bits	AX	IMUL BL
AX	R/M 16 bits	DX:AX	IMUL BX

Operación	DIV: Dividir sin signo
<i>Descripción</i>	Divide un dividendo sin signo entre un divisor sin signo. La división entre cero provoca una interrupción de división entre cero.
<i>Banderas</i>	(AF, CF, OF, PF, SF y ZF quedan indefinidas)
<i>Formato</i>	DIV {registro/memoria} (Ver tabla)
<i>Ejemplo</i>	--

Dividendo	Divisor	Cociente	Resto	Ejemplo
AX	R/M 8 bits	AL	AH	DIV BL
DX:AX	R/M 16 bits	AX	DX	DIV CX

Operación	IDIV: Dividir con signo
<i>Descripción</i>	Divide un dividendo con signo entre un divisor con signo. La división entre cero provoca una interrupción de división entre cero. IDIV trata el bit de la izquierda como el signo.
<i>Banderas</i>	(AF, CF, OF, PF, SF y ZF quedan indefinidas)
<i>Formato</i>	IDIV {registro/memoria}
<i>Ejemplo</i>	--

Dividendo	Divisor	Cociente	Resto	Ejemplo
AX	R/M 8 bits	AL	AH	IDIV BL
DX:AX	R/M 16 bits	AX	DX	IDIV CX

Operación	NEG: Niega
<i>Descripción</i>	Invierte un número binario de positivo a negativo y viceversa. NEG trabaja realizando el complemento a dos.
<i>Banderas</i>	AF, CF, OF, PF, SF y ZF.
<i>Formato</i>	NEG {registro/memoria}
<i>Ejemplo</i>	NEG AL

2.3.- Instrucciones Lógicas y de Manejo de Bits.

Se trata de instrucciones para realizar operaciones lógicas con los datos tales como AND, OR, XOR, etc., así como manipulación de los mismos a nivel de bits.

2.3.1.- Instrucciones Lógicas.

Operación	AND: Conjunción lógica
Descripción	Realiza la conjunción lógica de los bits de los dos operandos.
Banderas	CF (0), OF (0), PF, SF y ZF (AF queda indefinida).
Formato	AND {registro/memoria},{registro/memoria/inmediato}
Ejemplo	AND AL,BL

Operación	NOT: Negación lógica
Descripción	Complementa todos los bits del operando.
Banderas	No las afecta.
Formato	NOT {registro/memoria}
Ejemplo	NOT DX

Operación	OR: Disyunción lógica
Descripción	Realiza la disyunción lógica de los bits de los dos operandos.
Banderas	CF (0), OF (0), PF, SF y ZF (AF queda indefinida).
Formato	OR {registro/memoria},{registro/memoria/inmediato}
Ejemplo	OR CL,AH

Operación	XOR: Disyunción exclusiva
Descripción	Realiza la disyunción lógica exclusiva de los bits de los dos operandos.
Banderas	CF (0), OF (0), PF, SF y ZF (AF queda indefinida).
Formato	XOR {registro/memoria}
Ejemplo	XOR DL,AL

Operación	CMP: Comparar
Descripción	Se emplea para comparar el contenido de dos campos de datos. CMP resta el segundo operando del primero y actualiza las banderas, pero no guarda el resultado.
Banderas	AF, CF, OF, PF, SF y ZF.
Formato	CMP {registro/memoria},{registro/memoria/inmediato}
Ejemplo	CMP AL,12h

2.3.2.- Instrucciones de Manejo de Bits.

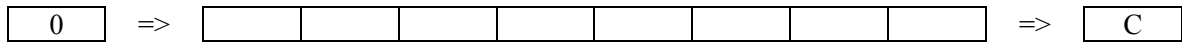
El formato general de este tipo de instrucciones es el siguiente:

Operación_Movimiento_Bits {registro/memoria},{CL, 1}
--

La operación indicada se aplicará al primer operando el número de veces que especifica el segundo operando.

SHR: Desplazamiento lógico a la derecha

Se mueven todos los bits a la derecha, en el bit más significativo se mete un cero, y el bit que sale por la derecha pasa a la bandera CF.



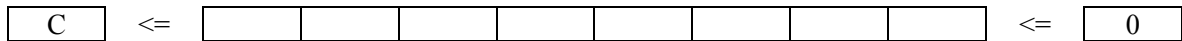
SAR: Desplazamiento aritmético a la derecha

Se mueven todos los bits a la derecha, en el bit más significativo se mete la bandera SF, y el bit que sale por la derecha pasa a la bandera CF.



SHL: Desplazamiento lógico a la izquierda

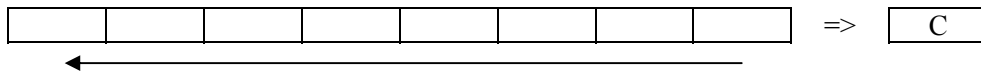
Se mueven todos los bits a la izquierda, en el bit menos significativo se mete un cero, y el bit que sale por la izquierda pasa a la bandera CF.



SAL: Desplazamiento aritmético a la izquierda (igual que SHL)

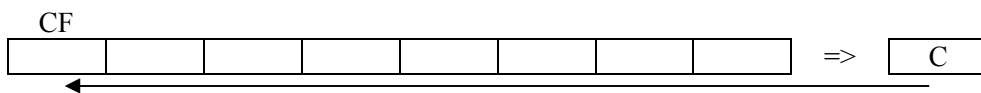
ROR: Rotación lógica a la derecha

Se mueven todos los bits a la derecha, en el bit más significativo se mete el bit que sale por la derecha, que se copia también en la bandera CF.



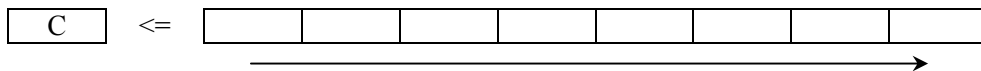
RCR: Rotación a la derecha con acarreo

Se mueven todos los bits a la derecha, en el bit más significativo se mete el valor de la bandera CF, y el bit que sale por la derecha pasa a ser el nuevo valor de la bandera CF.



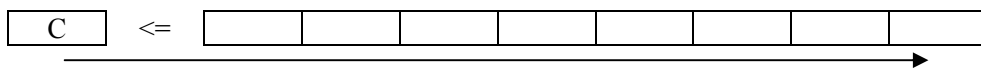
ROL: Rotación lógica a la izquierda

Se mueven todos los bits a la izquierda, en el bit menos significativo se mete el bit que sale por la izquierda, que se copia también en la bandera CF.



RCL: Rotación a la izquierda con acarreo

Se mueven todos los bits a la izquierda, en el bit menos significativo se mete el valor de la bandera CF, y el bit que sale por la izquierda pasa a ser el nuevo valor de la bandera CF.



2.4.- Instrucciones de Transferencia de Control.

Dentro de esta categoría de instrucciones, podemos distinguir entre instrucciones de transferencia de control condicionales e incondicionales, bucles, llamadas a procedimientos y subrutinas de atención a la interrupción. Las condicionales provocan una alteración en la secuencia normal de ejecución del programa, haciendo que el flujo de ejecución ‘salte’ de un punto del programa a otro sin que ambos sean consecutivos, dependiendo de que se cumpla o no una determinada condición relacionada normalmente con el registro de estado. Las incondicionales tienen el mismo efecto pero sin depender de los valores que en ese momento tengan las banderas del registro de estado. Los bucles permiten ejecutar una determinada secuencia de operaciones varias veces. Por último, los procedimientos o subrutinas aglutinan operaciones que se repiten reiteradamente a lo largo del programa, o bien, contienen instrucciones que realizan una acción muy específica.

2.4.1.- Instrucciones de Transferencia de Control Condicionales.

Transfieren el control dependiendo de las configuraciones en el registro de banderas. Por ejemplo, se pueden comparar dos datos y después saltar considerando los valores de las banderas que la comparación ha establecido. Cada instrucción posee un único operando que no es más que el desplazamiento que indica dónde ir si el test es positivo. Según consideremos los datos a comparar con o sin signo, distinguimos tres tipos de saltos condicionales.

Datos sin signo:

Mnemónico	Descripción	Flags
JE/JZ	Salto si igual	Z = 1
JNE/JNZ	Salto si no igual	Z = 0
JA/JNBE	Salto si superior	C = 0 y Z = 0
JAE/JNB	Salto si superior o igual	C = 0
JB/JNAE	Salto si inferior	C = 1
JBE/JNA	Salto si inferior o igual	C = 1 o Z = 1

Datos con signo:

Mnemónico	Descripción	Flags
JE/JZ	Salto si igual	Z = 1
JNE/JNZ	Salto si no igual	Z = 0
JG/JNLE	Salto si mayor	Z = 0 y S = 0
JGE/JNL	Salto si mayor o igual	S = 0
JL/JNGE	Salto si menor	S <> 0
JLE/JNG	Salto si menor o igual	Z = 1 o S <> 0

Tests aritméticos:

Mnemónico	Descripción	Flags
JS	Salto si signo negativo	S = 1
JNS	Salto si signo positivo	S = 0
JC	Salto si carry	C = 1
JNC	Salto si no carry	C = 0
JO	Salto si overflow	O = 1
JNO	Salto si no overflow	O = 0
JP/JPE	Salto si paridad par	P = 1

JNP/JPO	Salto si paridad impar	P = 0
JCXZ	Salto si CX = 0	CX = 0

2.4.2.- Instrucciones de Transferencia de Control Incondicionales.

Operación	JMP: Salto incondicional
<i>Descripción</i>	Salta a la dirección designada (operando). La dirección especificada puede ser corta (-128 a +127 bytes), cercana (dentro de 32K) o lejana (a otro segmento).
<i>Banderas</i>	No las afecta
<i>Formato</i>	JMP {registro/memoria}
<i>Ejemplo</i>	JMP FINAL

2.4.3.- Bucles.

Un bucle es un grupo de instrucciones que se ejecutan cíclicamente un número concreto de veces. Para construir bucles disponemos de las siguientes instrucciones:

Operación	LOOP: Repetir
<i>Descripción</i>	Controla la ejecución de un grupo de instrucciones un número específico de veces. Antes de iniciar el ciclo, CX debe contener el número de veces que ha de repetirse. LOOP aparece al final del conjunto de instrucciones que se repiten y decrementa CX. Al llegar a cero, permite que el flujo de ejecución pase a la siguiente instrucción. En caso contrario salta a la etiqueta que determina el comienzo del bucle.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	LOOP etiqueta
<i>Ejemplo</i>	<pre> MOV CX,num_veces Etiqueta: ;Instrucciones del bucle LOOP Etiqueta </pre>

Nota: LOOPE/LOOPZ tienen la misma función que LOOP, pero la condición para salir del bucle es que CX = 0 ó Z = 0. Para LOOPNZ/LOOPNE la condición es CX ≠ 0 ó Z = 1.

2.4.4.- Llamada a procedimientos.

Dentro del segmento de código es posible tener cualquier número de procedimientos. Un procedimiento (o subrutina) es una sección de código que realiza una tarea bien definida y clara (por ejemplo, situar el cursor en una posición concreta de la pantalla). La utilización de procedimientos en los programas es aconsejable porque:

- Reduce el número de líneas de código.
- Permite una mejor organización del programa.
- Facilita la localización de errores.
- Aumenta la legibilidad del programa.

La llamada a procedimientos se gestiona mediante dos instrucciones: CALL y RET.

Operación	CALL: Llamar a un procedimiento
------------------	--

<i>Descripción</i>	Llama a un procedimiento cercano o lejano. En una llamada a un procedimiento cercano se guarda en la pila IP. Después se carga el IP con el desplazamiento de la dirección de destino (donde se encuentra la primera instrucción del procedimiento). Si la llamada es a un procedimiento lejano, el proceso a seguir es el mismo pero ahora el tratamiento de las direcciones incluye también considerar el registro de segmento CS.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	CALL {registro/memoria}
<i>Ejemplo</i>	CALL PROCEDIMIENTO

Operación	RET: Regresar de un procedimiento
<i>Descripción</i>	Regresa de un procedimiento al que se entró previamente con un CALL cercano o lejano. Lo que hace esta instrucción es recuperar de la pila la dirección de la siguiente instrucción que se almacenó al hacer la llamada. Esto permitirá continuar la ejecución del programa en la siguiente instrucción al CALL.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	RET [VALOR POP]
<i>Ejemplo</i>	RET

2.6.- Instrucciones para Manejo de Cadenas.

Una cadena es una secuencia de bytes contiguos. Las operaciones que se pueden realizar sobre las cadenas son las siguientes:

Operación	MOVSB/MOVSX: Mover cadena
<i>Descripción</i>	Mueve cadenas entre localidades de memoria. El primer operando es apuntado por ES:DI (destino) y el segundo operando por DS:SI (fuente). Normalmente se utiliza el prefijo REP que hace que la operación se ejecute CX veces (se le resta uno a CX tras cada operación), de forma que según DF sea uno o cero, tras cada transferencia (byte o palabra) DI y SI disminuyen o aumentan.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	[REP] MOVSB/MOVSX
<i>Ejemplo</i>	MOV CX,3 REP MOVSB

Nota: CMPSB y CMPSX son similares a las dos instrucciones anteriores pero se utilizan para comparar bytes o palabras.

Operación	LODSB/LODSX: Cargar un byte/palabra
<i>Descripción</i>	Carga el registro acumulador (AX o AL) con el valor de la localidad de memoria determinada por DS:SI. SI se incrementa tras la transferencia.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	LODSB/LODSX
<i>Ejemplo</i>	LODSB/LODSX

Nota: STOSB y STOSX son similares a las dos instrucciones anteriores pero se utilizan para almacenar bytes o palabras en posiciones de memoria.

2.7.- Instrucciones de entrada/salida.

Los puertos de entrada y salida (E/S) permiten al procesador comunicarse con los periféricos. El 8086/88 utiliza buses de direcciones y datos ordinarios para acceder a los periféricos, pero habilitando una línea que distinga el acceso a los mismos de un acceso convencional a la memoria (si no existieran los puertos de entrada y salida, los periféricos deberían interceptar el acceso a la memoria y estar colocados en un área de la misma). Para acceder a los puertos de entrada y salida se utilizan las instrucciones IN y OUT.

Operación	IN: Entrada
<i>Descripción</i>	Transfiere desde un puerto de entrada un byte a AL o una palabra a AX
<i>Banderas</i>	No las afecta.
<i>Formato</i>	IN {AX/AL}, {número_puerto/DX} Número_puerto entre [0..255] DX entre [0..65535]
<i>Ejemplo</i>	IN AL, 0Bh

Operación	OUT: Salida
<i>Descripción</i>	Transfiere un byte desde AL o una palabra desde AX hasta el puerto.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	OUT {número_puerto/DX}, {AX/AL} Número_puerto entre [0..255] DX entre [0..65535]
<i>Ejemplo</i>	OUT AL, 0Bh

3.- PROGRAMACIÓN DEL PC SOBRE DOS

3.1.- Directivas del ensamblador.

El programa ensamblador permite usar diferentes enunciados que sirven para controlar la forma en que un programa se ensambla y lista. Estos enunciados reciben el nombre de *directivas*. Se caracterizan porque sólo tienen influencia durante el proceso de ensamblado, pero no generan código ejecutable alguno. Estas directivas pueden variar algo de un programa ensamblador a otro. Nosotros explicaremos algunas de las más usadas en el TASM.

3.1.1.- Directivas para declaración y manejo de segmentos.

Directiva	.MODEL (nótese “.”)
<i>Descripción</i>	Especifica el modelo de memoria utilizado (véase sección 4.2).
<i>Formato</i>	.MODEL modelo
<i>Ejemplo</i>	.MODEL Compact

La siguiente tabla resume los distintos tipos de **modelos de memoria**:

MODELO	CARACTERÍSTICAS
TINY	Datos y código cogen en un solo segmento de 64K.
SMALL	Un segmento de datos y otro de código.
COMPACT	Múltiples segmentos de datos y un único segmento de código. Este modelo y el anterior son los que normalmente se utilizan.
LARGE	Múltiples segmentos de datos y de código.

HUGE	Múltiples segmentos de datos y de código. Los segmentos pueden pasar 64K, pero haciendo operaciones especiales de normalización de direcciones.
------	---

Directiva	.DATA (nótese “.”)
Descripción	Las declaraciones siguientes se insertarán en el segmento de datos. Se continúa donde la anterior directiva .DATA terminó (véase sección 4.2).
Formato	.DATA

Directiva	.CODE (nótese “.”)
Descripción	Las declaraciones siguientes se insertarán en el segmento de código. Se continúa donde la anterior directiva .CODE terminó (véase sección 4.2)
Formato	.CODE

Directiva	.STACK (nótese “.”)
Descripción	Las declaraciones siguientes se insertarán en el segmento de pila. Se continúa donde la anterior directiva .STACK terminó. (véase sección 4.2)
Formato	.STACK

Directiva	PROC
Descripción	Para definir cada uno de los procedimientos incluidos en el segmento de código.
Formato	NombreProc PROC ... NombreProc ENDP
Ejemplo	

Directiva	END
Descripción	Finaliza todo el programa.
Formato	END [dir inicial] ; Generalmente etiqueta del PROC principal.

3.1.2.- Directivas para definición de datos.

El ensamblador permite definir elementos para datos de diferentes longitudes de acuerdo con un conjunto de directivas específicas para ello. El formato general es el siguiente:

[nombre]	Dn	Contador_Repeticiones DUP (expresión)
----------	----	---------------------------------------

Donde Dn es una de las directivas de la siguiente tabla:

DIRECTIVA	DESCRIPCIÓN
DB	Definir un byte. Sirve además para definir cadenas de caracteres.
DW	Definir una palabra (2 bytes).
DD	Definir una palabra doble (4 bytes).
DF	Definir una palabra larga (6 bytes).
DQ	Definir una palabra cuádruple (8 bytes).
DT	Definir diez bytes (10 bytes).

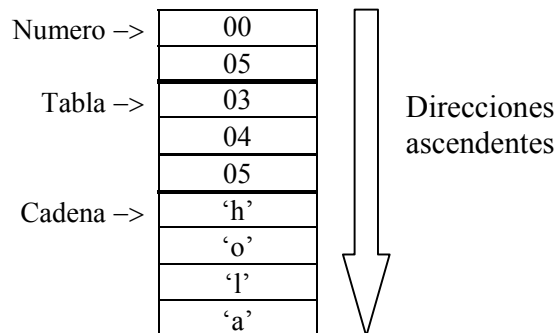
EJEMPLO	COMENTARIO
---------	------------

DATO1	DB	?	No se inicializa.
DATO2	DB	25	Constante decimal.
DATO3	DB	10101011B	Constante binaria.
DATO4	DB	1BH	Constante hexadecimal.
DATO5	DB	1,2,3,4,5,6,7,8,9,10	Diez bytes inicializados.
DATO6	DB	5 DUP(?)	Cinco bytes no inicializados.
DATO7	DB	5 DUP(14)	Cinco bytes inicializados a 14.
DATO8	DB	'Cadena de caracteres'	Cadena de caracteres. 1 byte por cada caracter
DATO9	DW	0FFF0H	Constante hexadecimal.
DATO10	DW	10,12,14,16,18,20	Seis palabras inicializadas.
DATO11	DD	?	No se inicializa.
DATO12	DD	14,49	Dos palabras dobles inicializadas.

Los datos se almacenan en memoria en el orden en que han sido declarados y podemos acceder a ellos usando el nombre dado en la declaración. Por ejemplo, si declaramos los siguientes datos:

```
Numero dw 5
Tabla 3, 4, 5
Cadena db 'hola'
```

El aspecto de la memoria sería el siguiente:



El nombre de cada dato es equivalente a una dirección de memoria, de forma que si sumamos o restamos un valor al nombre de una variable, accedemos a otra dirección de memoria. Por ejemplo:

```
MOV AX, Numero    => AX=0005
MOV AL, Tabla     => AL = 02
MOV AL, Tabla + 1 => AL = 03
MOV AL, Cadena    => AL = 'h' (código ASCII de la 'h')
MOV AL, Tabla + 3 => AL = 'h'
```

Asimismo, al igual que cuando operamos con dos registros, el “tamaño de la variable” debe ser el mismo que el del otro operando, esto es, operaciones como las siguientes darían un error al ensamblar:

```
MOV AL, Numero
MOV AX, Tabla
```

3.1.3.- Etiquetas y variables.

Un *identificador* es un nombre aplicado a elementos incluidos en el programa ensamblador. Hay dos tipos de identificadores que son:

- *Nombre o etiqueta de dato*: se refiere a la dirección de un dato almacenado en una determinada posición de memoria, esto es, una variable.
- *Etiqueta de instrucción*: se refiere a la dirección de una instrucción. En TASM la etiqueta se debe escribir al principio de la línea seguida de dos puntos (:)

```
etiqueta: MOV AX, BX
```

Las mismas reglas se aplican tanto para los nombres como para las etiquetas:

LETRAS DEL ALFABETO	De la A a la Z.
DÍGITOS	Del 0 al 9 (no puede ser el primer carácter).
CARACTERES ESPECIALES	Signo de interrogación (?) Subrayado (_) Signo del dólar (\$) Arroba (@) Punto (.)

Nota: ciertas palabras en lenguaje ensamblador están *reservadas* para propósitos específicos y no pueden ser usadas como identificadores (instrucciones, directivas, operadores y todos aquellos símbolos predefinidos usados por el ensamblador).

3.1.4.- Constantes Numéricas.

Las constantes numéricas se usan para definir valores aritméticos y direcciones de memoria. Las constantes numéricas no llevan comillas, como sucede con las cadenas de caracteres, y van seguidas de un indicador de la base utilizada. Si se omite éste, se supone el sistema decimal.

CONSTANTE	FORMATO	EJEMPLO
DECIMAL	[-] 0...9[D]	1234D
HEXADECIMAL	0...F[H]	0F0AH
BINARIO	0/1[B]	11101000B

Si el primer dígito de una constante hexadecimal es una letra (A..F), debe anteponerse un cero, para que el ensamblador pueda distinguir que se trata de una constante numérica y no una cadena de caracteres.

3.1.5.- La Directiva EQU.

La directiva EQU no define ningún tipo de dato, sino que define constantes, o sea, define valores que el ensamblador puede sustituir en otras instrucciones. Por ejemplo,

```
VALOR EQU 100
```

Cada vez que en el programa aparezca VALOR, el ensamblador lo sustituirá por 100. Así,

```
MOV AX, VALOR
```

Es equivalente a

```
MOV AX, 100
```

Es muy útil para evitar tener que actualizar en varias instrucciones una constante que se repite y que se modifica al depurar el programa.

3.2.- Esquema de un programa

Llegados a este punto ya podemos mostrar la estructura típica de un programa en ensamblador del 8086 con TASM:

```
.MODEL small ;obligatorio especificar el modelo de memoria

.STACK 100H ;reservamos memoria para la pila

.DATA
;declaración de datos o variables, si las hay

.CODE
    MOV AX, @DATA
    MOV DS, AX ;inicializamos DS

    ;aquí nuestro código

    MOV AH, 4Ch
    INT 21h
END
```

Todo el texto que haya en una línea después de un ‘;’ (punto y coma) el TASM lo trata como un comentario. Las dos primeras líneas del código sirven para que el registro DS tenga el valor del segmento donde el sistema operativo a introducido los datos al cargar el programa en memoria. Los registros CS y SS no es necesario inicializarlos, ya el Sistema Operativo se encarga de que apunten correctamente a los segmentos correspondientes, el de código y el de la pila, respectivamente.

Las dos últimas líneas indican al Sistema Operativo que el programa ha terminado y le devuelve el control del sistema.

3.3.- Ensamblado, linkado y montado.

El código descrito en el apartado anterior lo habremos escrito en un fichero de texto, con un editor cualquiera de texto, y en el caso de usar TASM, le habremos dado la extensión “.asm”. Pero obviamente este código aún no es ejecutable. Para que el procesador pueda interpretarlo debemos generar un fichero ejecutable (.EXE o .COM).

Ensamblado

Un ensamblador es el programa que convierte un listado de código fuente, escrito en un fichero de texto, en código objeto, es decir, lenguaje máquina (el que finalmente interpretará el procesador) en el que sólo le faltan las referencias a rutinas externas. El código fuente es un fichero que contiene la secuencia de instrucciones en lenguaje ensamblador que forma el programa, así como ciertas directivas o comandos para el propio ensamblador, que ha sido creado utilizando un editor de texto. El código objeto consta del código en lenguaje máquina y de la información necesaria para enlazarlo con otros módulos objeto. Con TASM el proceso de ensamblado se realiza escribiendo lo siguiente en la ventana de comandos:

```
tasm /zi /l nombre
```

donde nombre es el nombre del fichero .asm con el código fuente. La extensión no es necesaria ponerla y los parámetros /zi y /l sólo son útiles si queremos hacer un proceso de depuración (debug) con el Turbo Debugger. Tras la ejecución de esta línea se nos creará un fichero “nombre.obj

Linkado y montado

Al construir un programa algunos de sus módulos pueden colocarse en el mismo módulo fuente y ensamblarse juntos, otros pueden estar en módulos diferentes y ser ensamblados separadamente. En cualquier caso, los módulos objeto resultantes, algunos de los cuales pueden estar agrupados en librerías, deben ser enlazados para formar el módulo de carga, antes de que se pueda ejecutar el programa. Además de dar como salida el módulo de carga, el *linker* o enlazador imprime un mapa de memoria que indica donde serán cargados los módulos objeto en la memoria. Después de crearse el módulo de carga, éste es cargado por el cargador en la memoria del ordenador y comienza la ejecución. Para *linkar* con TASM escribimos:

```
tlink /v nombre
```

Tras esto se nos creará el fichero ejecutable (.exe o .com) que el sistema operativo se encargará de cargar en memoria cuando lo ejecutemos. De nuevo, la extensión no es necesaria ponerla y el parámetro /v sólo es útil para el Turbo Debugger

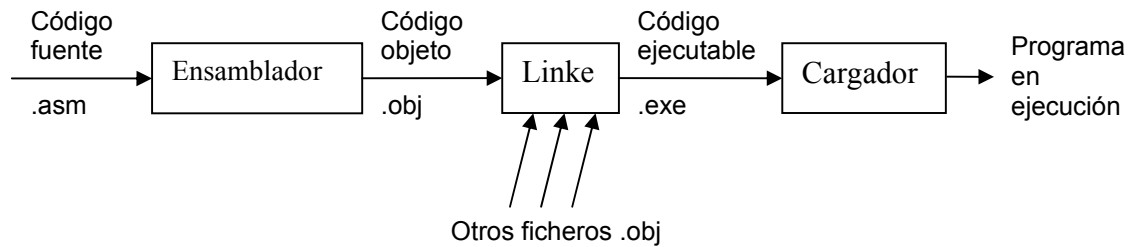


Figura 9: *Proceso de ensamblado, linkado y montado*

3.4.- Procedimientos

Una procedimiento no es más que una subrutina, o un fragmento de código al que se le llama, generalmente varias veces, desde el programa principal, o desde una procedimiento, jerárquicamente superior. Un procedimiento se declara mediante la directiva PROC, y se puede declarar en cualquier parte del segmento de código

```

.CODE
...
nombre_subrutina PROC

; código de la subrutina

RET
ENDP
...
END
  
```

Cuando se llama a un procedimiento simplemente se realiza un salto al trozo de código donde está definida la subrutina, y cuando ésta termina se vuelve al código desde el que se realizó la llamada. La llamada a la subrutina se realiza mediante la instrucción CALL y la vuelta al código la realizamos con RET, por lo que es importante acabar siempre los procedimientos con esta instrucción:

```

.CODE

CALL nombre_procedimiento
MOV AX, BX

RET
ENDP

END
  
```

El diagrama muestra flechas que conectan 'nombre_procedimiento' en la línea de la instrucción CALL con 'nombre_procedimiento PROC' y 'nombre_procedimiento PROC ; código de la subrutina' en el código de la subrutina, y una flecha que conecta 'RET' con la línea de la instrucción CALL.

Lo que ocurre realmente al hacer CALL es que, antes de realizar el salto, se almacena en la pila la dirección de retorno (la dirección de la siguiente instrucción a CALL), y la instrucción RET recupera esta dirección de la pila y salta a la misma.

Una práctica muy común y recomendable al escribir en una subrutina es guardar en la pila, al principio de su código, todos los registros que vaya a modificar y volverlos a sacar al final del código (antes de la instrucción RET). De esta forma el programa principal que realiza la llamada no

tiene que preocuparse de que la subrutina pueda modificarle el valor de algún registro que le fuese necesario.

3.4.1.- Paso de parámetros a procedimientos

3.4.1.1.- Registros.

Los parámetros para el procedimiento que se llama se pasan en los registros del procesador. De esta forma, se establece un compromiso entre el procedimiento que llama y el llamado, ya que éste último espera cada parámetro en un determinado registro del procesador. Este mecanismo es sumamente rápido, ya que no requiere realizar accesos a memoria. Por otro lado, adolece de dos inconvenientes importantes: el número de registros del procesador es limitado, es decir, el número de los parámetros está condicionado por los registros del procesador disponibles; tener los parámetros en registros supone no poder usar estos registros para realizar cualquier otra operación, a menos que se salvaguarden (por ejemplo, en la pila), lo que significa más accesos a memoria.

```
.MODEL SMALL
.DATA
    DATO1 DW 25
    DATO2 DW 33
.CODE
    MOV AX, SEG DATO1    ;equivalente a @DATA
    MOV DS, AX

MAIN:
    MOV BX, DATO1
    MOV CX, DATO2
    CALL SUMA
; ; ; LA SUMA ESTÁ EN AX!!!

    MOV AH, 4Ch
    INT 21h

SUMA PROC
    MOV AX, BX
    ADD AX, CX
    RET
SUMA ENDP

END MAIN
```

3.4.1.2.- Memoria Intermedia.

Otra forma de pasar los parámetros a un procedimiento, es utilizar posiciones de memoria específicamente dedicadas a ello. El procedimiento que llama almacena los parámetros en determinadas posiciones de memoria, donde el procedimiento llamado acude a recogerlos. Con este esquema, los parámetros no ocupan registros del procesador; como contrapartida se consume memoria, y el acceso a los parámetros es más lento que en el caso anterior. Otra desventaja adicional que escapa a primera vista es que este tipo de paso de parámetros no permite

procedimientos recursivos, ya que cada vez que se llama al procedimiento se modifican los parámetros.

```
PARAM1 DB ?
PARAM2 DW ?
. . .
MOV PARAM1,6AH ;Guarda en PARAM1 el primer parámetro
MOV PARAM2,12345D ;Guarda en PARAM2 el segundo parámetro
CALL PROCEDIMIENTO ;Llama al procedimiento
```

3.4.1.3.- Pila.

La tercera posibilidad para pasar los parámetros es hacer uso de la pila. Ésta es la que se usa formalmente, aunque por su “complejidad” no la explicaremos en este manual. Antes de llamar a un procedimiento, el llamante introduce en la pila los parámetros del procedimiento llamado. Este mecanismo supone igual que en el caso anterior un consumo de memoria, pero en este caso sólo temporalmente. Además, esta posibilidad permite el uso de procedimientos recursivos.

3.5.- MACROS

Una macro es un fragmento de código que el ensamblador copia, al ensamblar el código, en las líneas que contengan llamadas a la macro, esto es, cada vez que se realiza una llamada a una MACRO se sustituye esa llamada por el código de la MACRO. A diferencia del procedimiento no se realiza ningún salto ni la definición de la macro es código realmente (se convierte en código cuando lo copia en el lugar de la llamada), por lo que podemos declarar la MACRO en cualquier parte, dentro o fuera de la zona de código, aunque siempre la declaración debe estar antes que la llamada. Para poder utilizar etiquetas dentro de las macros, debemos declararlas como locales, ya que en caso contrario, al hacer distintas llamadas, tendríamos varias etiquetas con el mismo nombre.

```
nombre_macro  MACRO
LOCAL etiqueta_local
...
ENDM

.CODE
    ;programa principal
    ...
    nombre_macro          ;llamada a la MACRO. Esta línea se
                           sustituye por el código dentro de la
                           MACRO

END
```

3.5.1.- Paso de parámetros a las MACROS

A las macros se les puede pasar cualquier número de parámetros como se ve en el siguiente ejemplo:

```
Suma MACRO a, b
ADD a, b
```

ENDM

La llamada

Suma AX, BX

Es equivalente a

ADD AX, BX

3.6.- Interrupciones en el PC.

Una interrupción es una situación especial que suspende la ejecución de un programa de modo que el sistema pueda realizar una acción para tratarla. Tal se da, por ejemplo, cuando un periférico requiere la atención del procesador para realizar una operación de E/S.

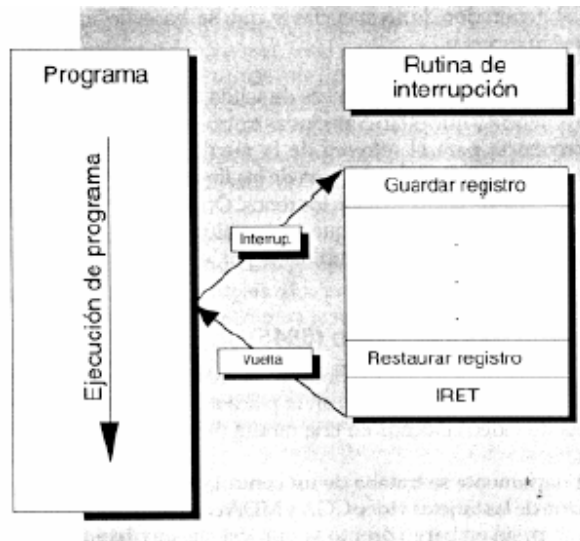


Figura 10. Esquema de una interrupción.

3.6.1.- Tratamiento de interrupciones.

Cuando se produce una petición de interrupción, se desencadena una secuencia de acciones:

- Finalizar la ejecución de la instrucción en curso.
- Almacenar en la pila el registro de estado.
- Almacenar en la pila la dirección de retorno: registros CS e IP.
- Inhibir las interrupciones.
- Colocar en CS:IP la dirección de comienzo de la rutina que tratará la interrupción.
- La rutina toma el control y almacenará todos los registros que utilice.
- Tratamiento de la interrupción.
- Se recuperan de la pila los registros previamente apilados.
- Con la instrucción IRET la rutina devuelve el control, ya que se restituye el registro de estado y la dirección de retorno CS:IP almacenada previamente.

- Se ejecuta la instrucción que sigue a aquella que estaba ejecutándose cuando se produjo la interrupción.

La rutina de tratamiento de la interrupción debe almacenar en la pila todos aquellos registros que vaya a utilizar antes de comenzar su tarea y restituirlos al finalizar, de modo que cuando se reanude la tarea interrumpida, se mantengan los valores que había en los registros.

3.6.2.- Interrupciones vectorizadas.

Toda interrupción aceptada por el procesador implica la ejecución de un programa específico para tratarla; ese programa recibe el nombre de *rutina de servicio de interrupción*. La dirección de comienzo, o puntero, se encuentra almacenada en cuatro posiciones de memoria consecutivas de una tabla. Las dos primeras posiciones contienen el offset o desplazamiento; las dos últimas, el segmento. La tabla contiene 256 punteros a memoria denominados **vectores de interrupción** y se encuentra entre las localidades de memoria 0000:0000H y 0000:03FFH. El vector número cero ocupa las direcciones 0, 1, 2 y 3, el vector número uno las cuatro siguientes y así sucesivamente. En total esta tabla ocupa $256 * 4 \text{ bytes} = 1\text{K}$.

Dirección		Número de interrupción
0000:0000	OFFSET	INT 0
0000:0001		
0000:0002	SEGMENTO	
0000:0003		
0000:0004	OFFSET	INT 1
0000:0005		
0000:0006	SEGMENTO	
0000:0007		
...		
0000:03FC	OFFSET	INT 255
0000:03FD		
0000:03FE	SEGMENTO	
0000:03FF		

Figura 11. Estructura de la tabla de vectores de interrupción.

3.6.3.- Tipos de Interrupciones.

Las **interrupciones hardware** hacen referencia a aquellas peticiones de interrupción, que llegan al procesador mediante una línea externa denominada INTR, procedentes de algún dispositivo periférico (por ejemplo, el teclado).

Las **interrupciones software** son una colección de procedimientos en código máquina que forman parte del sistema operativo y que pueden ser usados por el programador de aplicaciones. Para acceder a estos procedimientos contamos con las correspondientes entradas en la tabla de vectores de interrupción. Todos los procedimientos finalizan con la instrucción IRET, por lo que la llamada a estos procedimientos no se realiza con la instrucción CALL sino con la instrucción INT N donde N es el número del vector de interrupción. Se pueden clasificar en procedimientos BIOS

(10H a 1FH) y procedimientos DOS (20H a 3FH) (el BIOS contiene un conjunto de rutinas que se encuentran en un chip de memoria ROM, o memoria de sólo lectura, para dar soporte a los dispositivos).

Llamada a procedimientos DOS y BIOS.

Operación	INT: Interrupción
<i>Descripción</i>	Interrumpe el flujo normal de ejecución de un programa, transfiriendo el control a una de las 256 direcciones que se encuentran almacenadas en la tabla de vectores de interrupción.
<i>Banderas</i>	IF (0) y TF (0).
<i>Formato</i>	INT N° Vector
<i>Ejemplo</i>	INT 21H

Operación	IRET: Regresar de un procedimiento DOS o BIOS
<i>Descripción</i>	<i>Descripción</i> Regresa de un procedimiento al que se entró previamente con INT. Lo que hace esta instrucción es recuperar de la pila la dirección de la siguiente instrucción que se almacenó al hacer la llamada. Esto permitirá continuar la ejecución del programa en la siguiente instrucción a INT.
<i>Banderas</i>	No las afecta.
<i>Formato</i>	IRET [VALOR POP]
<i>Ejemplo</i>	IRET

Las **interrupciones internas o excepciones** son generadas por el propio procesador cuando se produce una situación anormal, como error de división o ejecución paso a paso.

3.6.4.- Capturar una interrupción.

La captura de una interrupción consiste básicamente en sustituir su entrada correspondiente de la tabla de vectores de interrupción por la dirección de una rutina propia, esto es, sustituiremos el segmento y el offset de la rutina original por nuestro segmento de código (registro CS) y el offset en el que se encuentre nuestra rutina dentro del segmento, que podemos conocer utilizando

```
OFFSET nombre_rutina
```

De esta forma, cuando se produce la interrupción, se ejecuta nuestra rutina y no la que había originalmente. Por ello, lo ideal es que nuestra rutina incluya al final un salto al código original de tratamiento de la interrupción. Por ejemplo, si queremos capturar la interrupción de pulsación de teclado (INT 09H) haríamos algo así.

```
CLI
MOV AX, 0000H
MOV ES, AX
MOV ES:[09h*4], OFFSET nuestra_rutina
MOV ES:[09h*4+2], CS
STI
```

La instrucción CLI inhabilita las interrupciones y STI las vuelve a habilitar. En otro punto del programa declararíamos la nueva rutina de servicio:

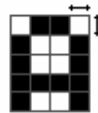
```
nuestra_rutina PROC
;guardar registros en la pila
...
;recuperar registros de la pila
IRET
ENDP
```

3.7.- La memoria de video.

3.7.1.- El sistema de visualización.

El sistema de visualización de un ordenador se compone básicamente de un monitor y una tarjeta gráfica insertada en la placa base. Los monitores se conectan a las tarjetas gráficas, estando éstas a su vez conectadas a un *bus* de expansión. La **tarjeta gráfica, tarjeta de vídeo o tarjeta controladora de vídeo**, es la encargada de controlar la información que se muestra en la pantalla del monitor. Esta tarjeta genera, por un lado, las señales de sincronización horizontal y vertical que controlan el monitor, y por otro, las señales que llevan el contenido de la memoria RAM de vídeo a la pantalla. Por su parte, el microprocesador es el encargado de colocar en la memoria de vídeo los datos procedentes de la aplicación que se ejecuta en cada momento. Los datos son convertidos en información, susceptible de ser representada en el monitor, gracias a la intervención de un programa residente llamado **controlador** o **driver**.

El sistema puede funcionar en **modo texto** y en **modo gráfico**. En cualquier caso, la información se representa mediante puntos luminosos llamados *pixels*.



En la imagen se muestra un trozo de la pantalla en la que aparece la letra “A” construida con *pixels*. La división entre *pixels* está dibujada en un color más suave. Las flechas al margen indican las medidas de un *pixel*.

3.7.2.- El modo texto.

En modo texto la pantalla se divide en celdas formadas por la intersección de 25 filas y 40 u 80 columnas. Cada celda es una palabra con dos bytes: un **byte atributo** y un **byte carácter**. El byte carácter simplemente contiene el código ASCII del carácter que se quiere visualizar en la celda de la pantalla. El byte atributo especifica cómo se va a visualizar dicho carácter (color, intensidad, parpadeo y color de fondo). Esta información se almacena en la RAM a partir, por lo general, aunque depende del tipo de tarjeta de video, de la posición B800:0000

Así pues, los dos primeros bytes de la memoria de vídeo, direcciones B800:0000 y B800:0001, contienen el carácter y el atributo de la posición (1,1) de la pantalla. En el offset 2, direcciones B800:0002 y B800:0003, tenemos el par correspondiente a la posición (1,2), etc. Así

hasta llegar a la posición (1,80), offset 158 (9EH). El offset 160 guardará el par correspondiente a la posición (2,1) de la pantalla, y así hasta la dirección B800:0F9Eh (offset 3998)

De una forma resumida, la fórmula a aplicar para acceder a una posición de la pantalla (f, c), donde f es la fila entre 1 y 25 y c es la columna entre 1 y 80, es:

$$offset(f,c) = (f-1)*80 + (c-1) * 2$$

Finalmente, ¿cómo se codifica el atributo de un carácter de la pantalla? Las siguientes tablas dicen cómo hacerlo:

Bit	Descripción
7	Parpadeo
6-4	Color de fondo
3	Color de fuente luminoso
2-0	Color de fuente

Bits de color	Normal	Luminoso
000b	Negro	Gris oscuro
001b	Azul	Azul claro
010b	Verde	Verde claro
011b	Cyan	Cyan claro
100b	Rojo	Rojo claro
101b	Magenta	Magenta claro
110b	Marrón	Amarillo
111b	Gris claro	Blanco

Con lo que el atributo 07H, en binario 00000111b, equivale a un carácter gris claro (111b) sobre un fondo negro (000b), y es el tipo de carácter por defecto en MS-DOS.

3.7.3.- El modo gráfico.

En el modo gráfico los bytes de la memoria de vídeo dan el color a uno o varios píxeles de la pantalla. Si usamos 256 colores, cada byte especifica el color de un píxel ($2^8 = 256$), si utilizamos modos de vídeo con más colores necesitaremos más de un byte para cada píxel, y si usamos menos colores, con un solo byte podemos especificar el color de varios píxeles.

Como ejemplo, estudiaremos el modo de vídeo estándar de las tarjetas VGA y SVGA de 256 colores y 320x200 *pixels*, tan famoso unos años atrás, en los que cualquier juego o presentación se realizaba en ese modo de vídeo.

En este modo la correspondencia entre direcciones de memoria y de pantalla es muy sencilla, ya que el byte en la dirección absoluta A0000h (el primer byte de la memoria de vídeo) da el color para el píxel de la esquina superior izquierda (0,0). La dirección siguiente corresponde al siguiente píxel a la derecha. Así hasta la posición A013Fh, que corresponde a la posición de la pantalla (0,319), es decir, la esquina superior derecha. La fórmula pues de acceso a un *píxel* queda

$$offset = (320 * y) + x$$

Pero antes de poder escribir en modo gráfico, debemos *cambiar* al modo gráfico. Esto se consigue con la función 00h de la interrupción 10h. Poniendo AH=00, AL=13h, y llamando a la

interrupción 10h, conseguimos introducirnos en este modo de pantalla. Poniendo AH=00h, AL=03h, conseguimos volver al modo texto 80x25.

3.8.- Programas .COM

Hasta ahora siempre hemos supuesto que estamos programando programas .EXE. Los programas .EXE y los .COM difieren básicamente en cuanto al tamaño del código ejecutable, la inicialización y los segmentos, la forma en que quedan almacenados en los ficheros y la velocidad de carga en memoria.

Tamaño del programa.

El tamaño de un programa .EXE sólo está limitado por la memoria de la que se disponga, mientras que un programa .COM está restringido a un segmento, y por tanto, a una longitud máxima de 64K, incluyendo el PSP. El PSP es un bloque de 256 bytes que el DOS inserta antes de los programas .COM y .EXE cuando los carga en memoria su ejecución. Un programa .COM es más pequeño que su correspondiente .EXE. Uno de los motivos es el encabezado, que ocupa espacio en disco y que precede a un programa .EXE, que no aparece en un .COM.

Inicialización.

Al cargar un programa .COM en memoria para su ejecución, todos los registros de segmento se inicializan de forma automática con la dirección del PSP. Los registros CS y DS contendrán la dirección de segmento inicial correcta, luego no hay que manipularlos.

Segmentos.

Segmento de Pila. En un programa .EXE hay que definir un segmento de pila, mientras que en un programa .COM la pila se genera automáticamente. De este modo, cuando se escribe un programa .COM la definición de la pila se omite.

Segmento de Datos. Un programa .EXE por lo común define un segmento de datos e inicializa el registro DS con la dirección de ese segmento. Como en un programa .COM los datos están definidos dentro del segmento de código, tampoco se tiene que definir el segmento de datos.

Segmento de Código. En un programa .COM el PSP, la pila, el segmento de datos y el segmento de código se encuentran dentro del mismo segmento, que como máximo tendrá 64K.

El aspecto, por tanto, de un programa .COM sería el siguiente:

```
.MODEL SMALL
.CODE
ORG 100h ;se salta los 255 bits porque no es codigo nuestro

main:
    JMP begin
    ;declaración de variables

begin:
```

```

        ;código
    ...
    MOV AH, 4Ch
    INT 21h
END main

```

Para generar el ejecutable .COM con el TASM, hay que *linkar* usando el parámetro /t

```
tlink /t nombre
```

3.9.- Programas residentes.

Existen determinados programas que están diseñados para permanecer en memoria mientras otros se ejecutan. Normalmente estos programas se activan mediante la pulsación de una secuencia especial de teclas. Estos programas se cargan en memoria después de que lo haga el DOS y antes de activar otros programas de procesamiento normal. Casi siempre son programas .COM y también son conocidos como *programas residentes en memoria* (TSR: termina pero permanece residente).

Para hacer que el programa quede residente, en lugar de una terminación normal, se utiliza la función INT 27H (también se podría utilizar la función 31H de la INT 21H, aunque tiene un funcionamiento ligeramente distinto):

```

MOV DX,direccion ;
INT 27H

```

La dirección que hay que pasar en DX es la dirección siguiente a la última línea de código que queremos que quede residente en memoria. Normalmente se coloca una etiqueta después de dicha línea y se pasa el offset de esta etiqueta a DX.

```

.MODEL SMALL
.CODE
ORG 100h ;normalmente el programa residente será .COM

main:
    JMP begin
    ;declaración de variables

begin:
    ;código residente
    ...

residente:
    ;código no residente
    MOV DX, offset residente
    INT 27h
END main

```