ENSAMBLADOR DEL 8086/88

Apuntes realizados por:
Juan Fernández Peinador
Revisados por:
Diego Sevilla Ruiz
Dpto. Ingeniería y Tecnología de Computadores
Facultad de Informática - Universidad de Murcia
Febrero de 1998

INDICE

0.- INTRODUCCIÓN. 4

1.- LA FAMILIA DEL 8086/88. 4

2.- ARQUITECTURA DEL 8086. 6

2.1.- REGISTROS DEL 8086/88 Y DEL 80286. 7
2.1.1.- Registros de propósito general. 7
2.1.2.- Registros de Segmento. 8
2.1.3.- Registro Apuntador de Instrucciones (IP). 8
2.1.4.- Registros Apuntadores (SP y BP). 8
2.1.5.- Registros Indice (SI y DI). 9
2.1.6.- Registro de banderas, FLAGS, o registro de estado (FL). 9

2.2.- SEGMENTOS Y DIRECCIONAMIENTO. 10
2.2.1.- Segmentos y Desplazamientos (*offsets*). 10
2.2.2.- Direccionamiento de Localidades de Memoria. 11
2.2.3.- Direccionamiento de Programas. 12

2.3.- PILA (STACK). 13

2.4.- MODOS DE DIRECCIONAMIENTO. 14
2.4.1.- Registros de Segmento por defecto. 15

2.5.- REGISTROS DEL 80386 Y SUPERIORES. 16

2.6.- EJEMPLO: CÓDIGO MÁQUINA VS. MNEMÓNICOS. 16

3.- CONJUNTO DE INSTRUCCIONES. 17

3.1.- Codificación de las instrucciones. 17

3.2.- Instrucciones de Transferencia de Datos. 19

3.3.- Instrucciones Aritméticas. 20

3.4.- Instrucciones Lógicas y de Manejo de Bits. 23
3.4.1.- Instrucciones Lógicas. 23
3.4.2.- Instrucciones de Manejo de Bits. 24

3.5.- Instrucciones de Transferencia de Control. 25
3.5.1.- Instrucciones de Transferencia de Control Condicionales. 25
3.5.2.- Instrucciones de Transferencia de Control Incondicionales. 26
3.5.3.- Bucles. 27
3.5.4.- Llamada a procedimientos. 27

3.6.- Instrucciones para Manejo de Cadenas. 28

3.7.- Instrucciones de Control de Flags. 29

3.8.- Instrucciones de entrada/salida. 29

4.- PROGRAMACIÓN DEL PC EN ENSAMBLADOR SOBRE DOS. 30

4.0.- INTRODUCCIÓN. MODELO DE TRES CAPAS. 30

4.1.- DIRECTIVAS DEL ENSAMBLADOR. 32
4.1.1.- Directivas para listar : PAGE y TITLE. 32
4.1.2.- Directivas para declaración y manejo de segmentos. 32
4.1.3.- Directivas para definición de datos. 34
4.1.4.- Etiquetas y variables. 35

4.1.5.- Constantes Numéricas. 35
4.1.6.- La Directiva EQU. 35
4.1.7.- El operador PTR. 36

4.2.- DIRECTIVAS, SEGMENTOS, PROCEDIMIENTOS Y PASO DE PARÁMETROS. *36*
4.2.1.- Segmentos y modelos de memoria 36
4.2.2.- Paso de parámetros a procedimientos 40

4.3.- PREFIJO DE SEGMENTO DE PROGRAMA (PSP). *46*
4.3.1.- Algunos Campos del PSP. 47

4.4.- INICIALIZAR UN PROGRAMA PARA SU EJECUCIÓN. *48*

4.5.- TERMINAR LA EJECUCIÓN DE UN PROGRAMA. *49*

4.6.- DISCOS. *49*
4.6.1.- Principio de Almacenamiento Inverso (*big-endian*). 50
4.6.2.- Estructura física de un disco. 51
4.6.3.- Estructura lógica de un disco. 52
4.6.4.- Particiones de un Disco Duro. 61
4.6.6- Parámetros absolutos de un Disco Duro. 62
4.6.7.- Leer y Escribir sectores en un disco. 62

4.7.- LA MEMORIA DE VIDEO. *63*
4.7.1.- El sistema de visualización. 63
4.7.2.- Tipos de Tarjetas Gráficas. 65
4.7.3.- La RAM de Vídeo. 65
4.7.4.- El modo texto. 66
4.7.5.- El modo gráfico. 68

4.8.- ASIGNACIÓN Y LIBERACIÓN DE MEMORIA. *70*

4.9.- DIFERENCIAS ENTRE PROGRAMAS .COM Y .EXE. *71*

4.10.- INTERRUPCIONES EN EL PC. *74*
4.10.1.- ¿ Qué es una interrupción ?. 74
4.10.2.- Tratamiento de interrupciones. 75
4.10.3.- Interrupciones vectorizadas. 75
4.10.4.- Tipos de Interrupciones. 76
4.10.5.- Circuito Controlador de Interrupciones: i8259. IRQ's. 78
4.10.6.- Capturar una interrupción. 79
4.10.7.- Circuito Temporizador: i8253. 79

4.11.- PROGRAMAS RESIDENTES. *81*
4.11.1.- Cómo hacer que el programa quede residente. 81
4.11.2.- Activación del programa residente. 81
4.11.3.- Obtener dirección de interrupción. 82
4.11.4.- Establecer dirección de interrupción. 82
4.11.5.- Ejemplo: BOCINA.ASM. 82

4.12.- INTERFAZ DE ENSAMBLADOR CON OTROS LENGUAJES DE PROGRAMACIÓN. *85*
4.12.1.- Interfaz con Pascal. 85
4.12.2.- Interfaz con C. 88

BIBLIOGRAFÍA **90**

DIRECCIONES WEB **90**

SOFTWARE **90**

0.- INTRODUCCIÓN.

El *cerebro* del ordenador es el **procesador**. Su función es ejecutar los programas almacenados en la memoria principal tomando las instrucciones, examinándolas y ejecutándolas una tras otra. Para ello, el procesador realiza todas las operaciones aritméticas, lógicas y de control que son necesarias.

1.- LA FAMILIA DEL 8086/88.

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 (1MB) y comparten el mismo conjunto de instrucciones.

La filosofía de diseño de la familia de 8086 se basa en la compatibilidad. De acuerdo con este principio, para permitir la compatibilidad con los anteriores sistemas de 8 bits, el 8088 se diseñó con un bus de datos de 8 bits, lo cual le hace ser más lento que su hermano el 8086; éste es capaz de cargar una palabra ubicada en una dirección par en una sola operación de lectura de memoria mientras el 8088 debe realizar dos lecturas, leyendo cada vez un byte. Ambos disponen de 92 tipos de instrucciones, que pueden ejecutar con diversos modos de direccionamiento, y pueden hacer referencia hasta a 64K puertos de entrada/salida (65536 puertos). Versiones mejoradas del 8086 son los procesadores 80186, 80286, 80386, 80486, Pentium (P5) y Pentium II (P6). Cada uno de ellos permite operaciones adicionales y mayor capacidad de procesamiento.

El procesador de Intel **80286** se caracteriza por poseer dos modos de funcionamiento completamente diferenciados: el **modo real** en el que se encuentra nada más ser conectado y el **modo protegido** en el que se facilita el **procesamiento multitarea** y permite el almacenamiento con sistema de **memoria virtual**. El procesamiento multitarea consiste en ejecutar varios procesos, de manera aparentemente simultánea, con la ayuda del sistema operativo para conmutar automáticamente de uno a otro, optimizando el uso del procesador. La memoria virtual permite al ordenador usar más memoria de la que realmente tiene, almacenando parte de ella en disco: de esta forma, los programas creen tener a su disposición más memoria de la que realmente existe; cuando acceden a una parte de la memoria lógica que no existe físicamente, se produce una excepción y el sistema operativo se encarga de acceder al disco y extraerla.

Cuando el procesador está en modo protegido, los programas de usuario tienen un acceso limitado al juego de instrucciones; sólo el proceso supervisor (normalmente el S.O.) está capacitado para realizar ciertas tareas. Esto es así para evitar que los programas de usuario puedan campar a sus anchas y entrar en conflicto unos con otros, en materia de recursos como memoria o periféricos. Además, de esta manera, aunque un error software provoque que se *cuelgue* un proceso, los demás pueden seguir funcionando normalmente, y el sistema operativo podría abortar ese proceso. Por desgracia, con el DOS no se trabaja en modo protegido, y el comportamiento anómalo de un único proceso provoca la caída de todo el sistema.

El 8086 no posee ningún mecanismo para apoyar la multitarea ni la memoria virtual desde el procesador, por lo que es difícil diseñar un sistema multitarea que sea realmente operativo.

Las características generales del 80286 son: un bus de datos de 16 bits, un bus de direcciones de 24 bits (16MB), 25 instrucciones más que el 8086 y algunos modos de direccionamiento adicionales.

Por su parte, el procesador **80386** dispone de una arquitectura de registros de 32 bits con un bus de direcciones también de 32 bits (4GB) y más modos de funcionamiento: el **modo real**, el **modo protegido** (relativamente compatible con el del 80286) y el **modo *virtual 86*** que permite emular el funcionamiento de varios 8086. Como ocurre con el 80286, los distintos modos de funcionamiento son incompatibles entre sí y requieren un sistema operativo específico capaz de aprovechar toda las posibilidades que ofrecen.

El procesador **80486** se diferencia del anterior en la integración dentro del chip del coprocesador matemático y la memoria caché, y en un aumento notable del rendimiento gracias a una técnica denominada segmentación que permite el solapamiento en la ejecución de las instrucciones mediante un *pipeline*.

El siguiente eslabón de la cadena es el **Pentium**. Se diferencia del 80486 en su bus de datos que es de 64 bits y en un elevado nivel de optimización y segmentación en la ejecución de las instrucciones, con dos *pipelines*, que le permiten, en muchos casos, si se emplean compiladores optimizadores, solapar la ejecución de varias instrucciones consecutivas. Además, el Pentium tiene la capacidad de predecir el destino de los saltos y la unidad de coma flotante ha sido considerablemente mejorada. Por último, el **Pentium II** tiene una estructura análoga a la del Pentium.

Una característica común a partir del 80386 es la disponibilidad de **memorias caché** de alta velocidad de acceso que almacenan una pequeña porción de la memoria principal. Cuando la CPU accede a una posición de memoria, ciertos circuitos se encargan de ir depositando el contenido de esa posición y el de las posiciones inmediatamente consecutivas en la memoria caché. Cuando sea necesario acceder a memoria, por ejemplo en busca de la siguiente instrucción del programa, la información requerida se encontrará en la memoria caché lo que hará que el acceso sea muy rápido.

Procesador	Registros	Bus Datos	Bus Direcciones
8088/80188	16 bits	8 bits	20 bits (1MB)
8086/80186	16 bits	16 bits	20 bits (1MB)
80286	16 bits	16 bits	24 bits (16MB)
80386	32 bits	32 bits	32 bits (4GB)
80486	32 bits	32 bits	32 bits (4GB)
Pentium	32 bits	64 bits	32 bits (4GB)
Pentium II	32 bits	64 bits	32 bits (4GB)

Figura 1. ***Características principales de la familia del 8086/88.***

2.- ARQUITECTURA DEL 8086.

El 8086 es un circuito integrado (CI) que posee unos 29.000 transistores NMOS, lo que supone unas 9.000 puertas lógicas. Está montado en una placa de silicio sobre una cápsula de cerámica con 40 pines.

De los 40 pines, 20 están dedicados a la tarea de especificar la dirección de memoria, por tanto, existe la posibilidad de direccionar 2^{20} posiciones de memoria (bytes), lo que equivale a 1Mb de memoria principal.

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.

La BIU tiene tres elementos fundamentales: la unidad de control del bus, la cola de instrucciones y los registros de segmento. La BIU controla el bus externo que comunica el procesador con la memoria y los distintos dispositivos de E/S. Los registros de segmento controlan el direccionamiento y permiten gestionar hasta 1 MB de memoria principal. La BIU accede a la memoria para recuperar las instrucciones que son almacenadas en la cola de instrucciones constituida por 6 bytes (4 bytes para el 8088). Mientras la BIU busca las instrucciones, la EU ejecuta las instrucciones que va recogiendo de la cola, es decir, la BIU y la EU trabajan en paralelo.

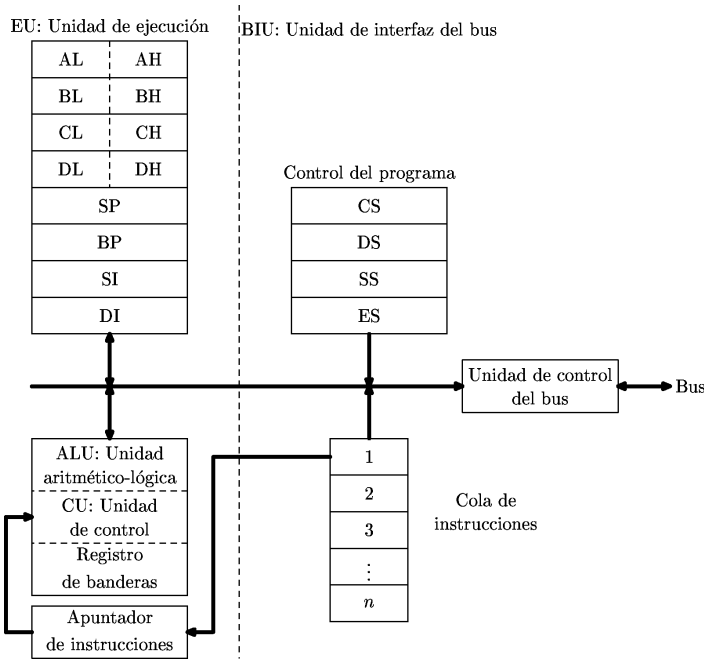


Figura 2. ***Arquitectura interna del 8086.***

2.1.- REGISTROS DEL 8086/88 Y DEL 80286.

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	1	1	0	1	0	1	0	1	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).

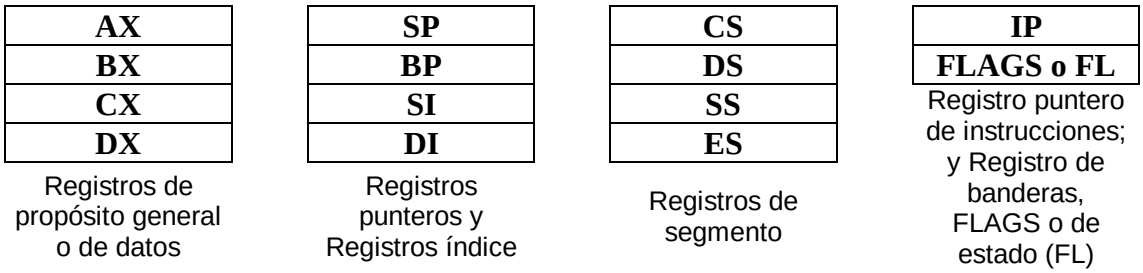


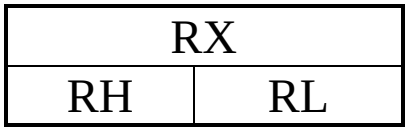
Figura 3. ***Registros del 8086/88 y del 80286.***

2.1.1.- 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 y el byte de la derecha es la parte baja:



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.

2.1.2.- Registros de Segmento.

Los registros de segmento son registros de 16 bits que constituyen la implementación física de la arquitectura segmentada del 8086.

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. Se suele utilizar en algunas operaciones con cadenas de caracteres para direccionar la memoria.

2.1.3.- 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:	25E52H

2.1.4.- Registros Apuntadores (SP y BP).

Los registros apuntadores están asociados al registro de segmento SS y permiten acceder a los datos almacenados en la pila:

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.

2.1.5.- Registros Indice (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.

2.1.6.- 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.

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 *trace* (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.

2.2.- SEGMENTOS Y DIRECCIONAMIENTO

Un *segmento* es un área especial de memoria en un programa que comienza en un *límite de párrafo*, es decir, en una posición de memoria divisible entre 16 (10H). Un segmento puede estar ubicado en casi cualquier lugar de la memoria y puede alcanzar hasta 64K de longitud. Por tanto, sólo necesitamos 16 bits para especificar la dirección de comienzo de un segmento (recordemos que el 8086 utiliza 20 líneas para direccionar la memoria), puesto que los cuatro bits menos significativos serían siempre cero.

En el 8086 se definen cuatro segmentos:

DS ⇒ Segmento de Datos
SS ⇒ Segmento de Pila
CS ⇒ Segmento de Código
ES ⇒ Segmento Extra

Para direccionar un segmento en particular basta con cambiar el contenido del registro de segmento apropiado.

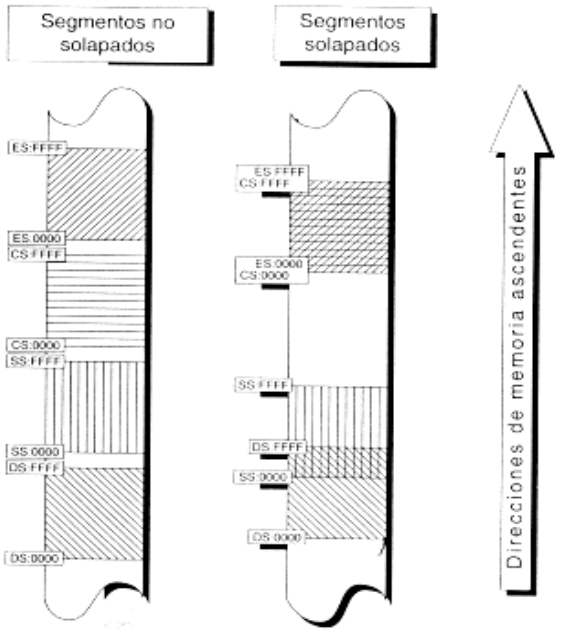


Figura 4. ***Segmentos solapados y no solapados.***

2.2.1.- Segmentos y Desplazamientos (*offsets*).

Todas las direcciones de memoria están referidas a la dirección de comienzo de algún segmento. La distancia en bytes desde la dirección de inicio del segmento se define como el *offset* o desplazamiento. El desplazamiento es una cantidad de dos bytes, que por tanto va de 0000H hasta FFFFH (0 a 65535) por lo que un segmento abarca como máximo 64K. 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.

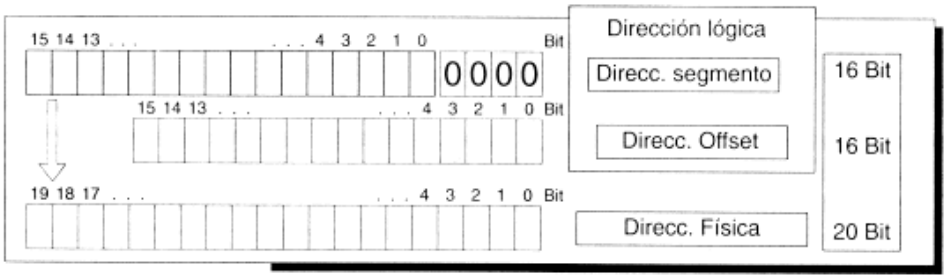


Figura 5. ***Construcción de una dirección de memoria en base a una dirección de segmento y una dirección de offset.***

Dado el registro DS con valor 045FH y un desplazamiento de 0032H:

Dirección del segmento de datos en el registro DS:	045F0H
Desplazamiento dentro del segmento de datos:	+ 0032H
Dirección real:	04622H

Nota: 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.

2.2.2.- 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 en secuencia inversa de bytes, es decir, 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 6.

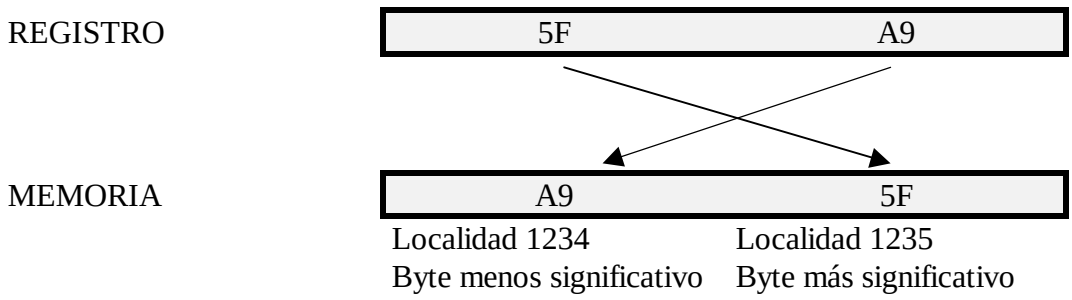


Figura 6. ***Direccionamiento de Localidades de Memoria.***

Es importante tener en cuenta esta peculidaridad ya que al almacenar palabras en una sola operación, la recuperación de esas palabras de memoria ha de hacerse en longitudes de 16 bits; en caso de recuperarse en 2 octetos, el primer octeto recuperado (dirección más baja de memoria) correspondería al octeto menos significativo y el siguiente al más significativo.

2.2.3.- Direccionamiento de Programas.

Cuando ejecutamos un programa, el DOS carga en la memoria el código máquina del programa. El registro CS contiene la dirección de inicio del segmento de código del programa y el registro DS la dirección de inicio del segmento de datos. 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. El registro IP indica el desplazamiento de la instrucción actual que es ejecutada dentro del segmento de código. Un operando (que sea una referencia a memoria) de una instrucción indica un desplazamiento en el segmento de datos del programa.

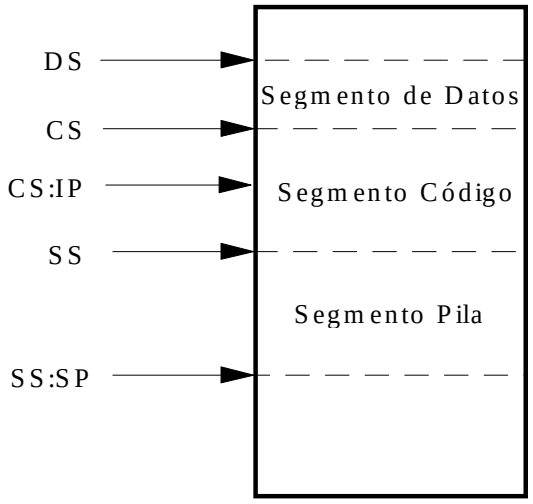


Figura 7. ***Esquema de direccionamiento para un programa.***

Supongamos que ejecutamos un programa .EXE y el DOS determina que el segmento de código comienza en la dirección 04AF0H y el segmento de datos en la dirección 04B10H. Por tanto, el registro CS contendrá el valor 04AFH y el registro DS el valor 04B1H. En un momento concreto de la ejecución del programa el IP contiene un desplazamiento de 0013H. La pareja CS:IP determina la dirección de la siguiente instrucción a ejecutar:

Dirección del segmento de código en el registro CS:	04AF0H
Desplazamiento dentro del segmento de código:	+ 0013H
Dirección real de la instrucción:	04B03H

Sea A01200 el código máquina de la instrucción que comienza en 04B03H. El código simbólico correspondiente es: MOV AL, [0012h]. La localidad de memoria 04B03 contiene el primer byte de la instrucción, los dos bytes siguientes contienen el desplazamiento, en secuencia invertida de bytes, dentro del segmento de datos:

Dirección del segmento de datos en el registro DS:	04B10H
Desplazamiento dentro del segmento de datos:	+ 0012H
Dirección real del dato:	04B22H

Si la dirección de memoria 04B22H contiene el valor 1BH, este se carga en la parte baja del registro AX (AL). Cuando el procesador está buscando cada uno de los

bytes que componen la instrucción, incrementa el registro IP, de manera que al finalizar la ejecución de la instrucción, contenga el desplazamiento de la siguiente instrucción.

Este ejemplo se representa gráficamente en la siguiente figura:

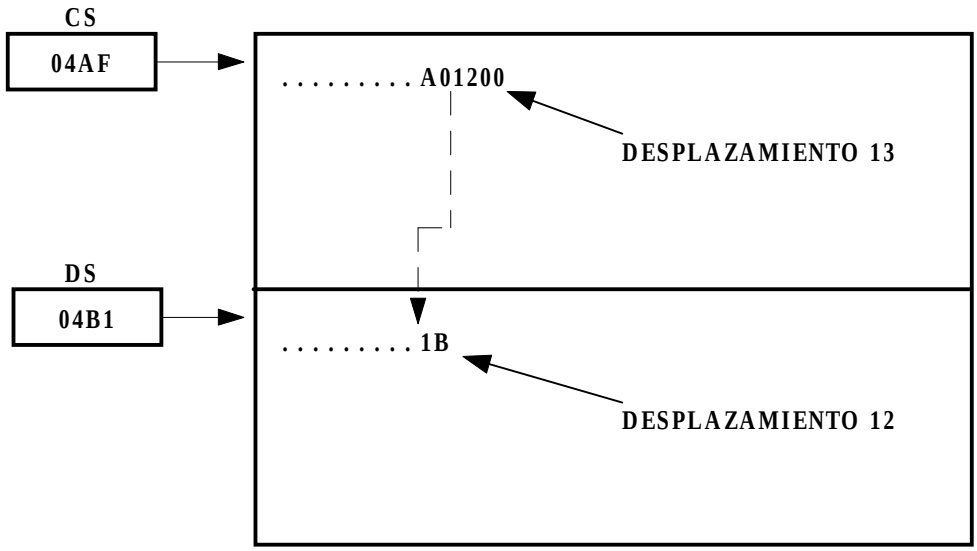
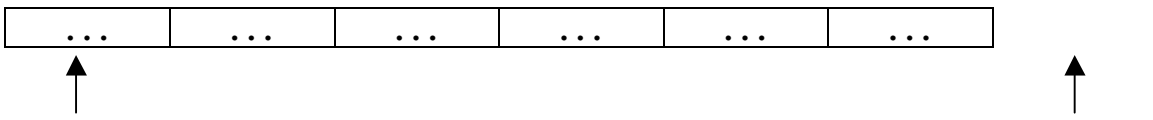


Figura 8. ***Ejemplo de direccionamiento para un programa.***

2.3.- PILA (STACK).

Los programas ejecutables (.COM y .EXE) requieren una zona de memoria reservada denominada *pila* (*stack*). La pila no es más que un área para el almacenamiento temporal de direcciones y datos.

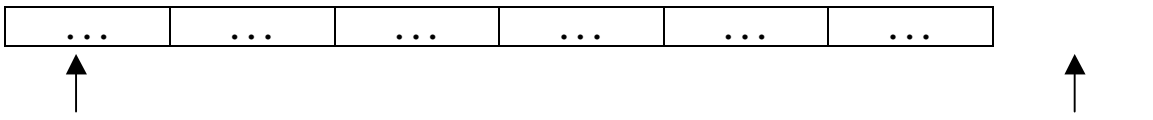
El registro SS es inicializado por el DOS con la dirección de inicio de la pila, mientras que el registro SP contiene el tamaño de la pila: apunta al byte siguiente al último que pertenece a la pila. La pila se caracteriza porque empieza a almacenar datos en la localidad más alta y avanza hacia abajo en la memoria:



SS: dirección del segmento de pila SP: tope de la pila

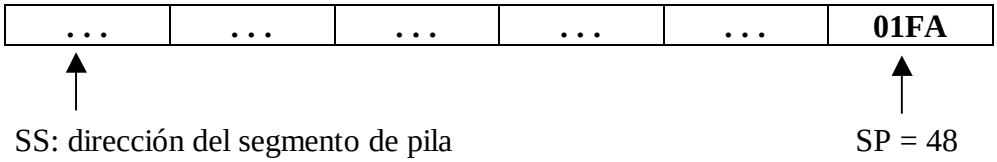
Ejemplo: Introducimos en la pila los registros AX y BX que contienen FA01H y 35A9H respectivamente. Suponemos que el registro SP contiene 50 bytes.

1. Estado inicial de la pila.

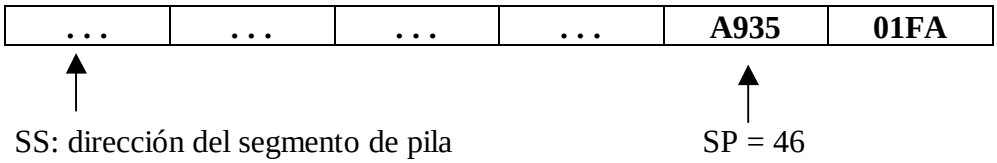


SS: dirección del segmento de pila SP = 50

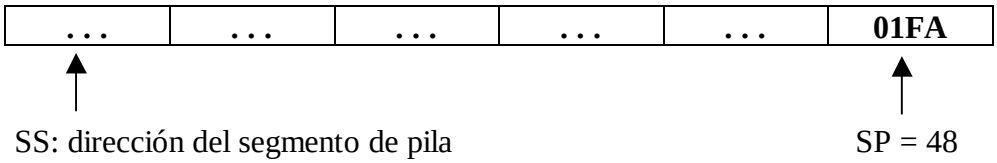
2. Se introduce en la pila AX ⇒ PUSH AX.



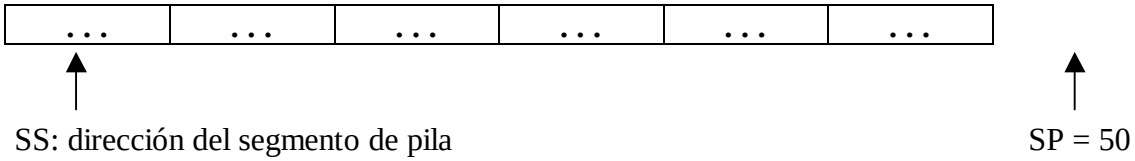
3. Se introduce en la pila BX ⇒ PUSH BX.



4. Se extrae de la pila BX ⇒ POP BX. El valor extraído de la pila se guarda en BX.



5. Se extrae de la pila AX ⇒ POP AX. El valor extraído de la pila se guarda en AX.



Es importante darse cuenta con este ejemplo de que las instrucciones POP deben ejecutarse en orden inverso a las órdenes PUSH: hay que sacar los valores en orden inverso al que se mandaron a la pila. Además, en un programa .EXE hay que definir una pila suficientemente grande para contener todos los valores que podrían ser guardados en ella durante la ejecución del programa.

2.4.- 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.

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

Ejemplo: El operando fuente en MOVE AX, 789AH

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

Ejemplo: Los operandos en MOV AX, 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]
El operando destino en MOV DIR1, AX

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]
El operando destino en MOV [BP + 2], AX

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]
El operando destino en MOV [DI + 2], AX

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 fuente en MOV AX, [BX + SI + 6]
El operando destino en MOV [BP + DI + 2], 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].

2.4.1.- 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. En este caso, la instrucción incluye un byte adicional (a modo de prefijo) para indicar cuál es el segmento referenciado.

La Figura 9 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 9. ***Segmentos por defecto.***

2.5.- REGISTROS DEL 80386 Y SUPERIORES.

Los 80386 y superiores disponen de muchos más registros de los que se muestran a continuación. Sin embargo, bajo el DOS sólo se suelen emplear los que veremos, que constituyen básicamente una extensión a 32 bits de los registros originales del 8086.

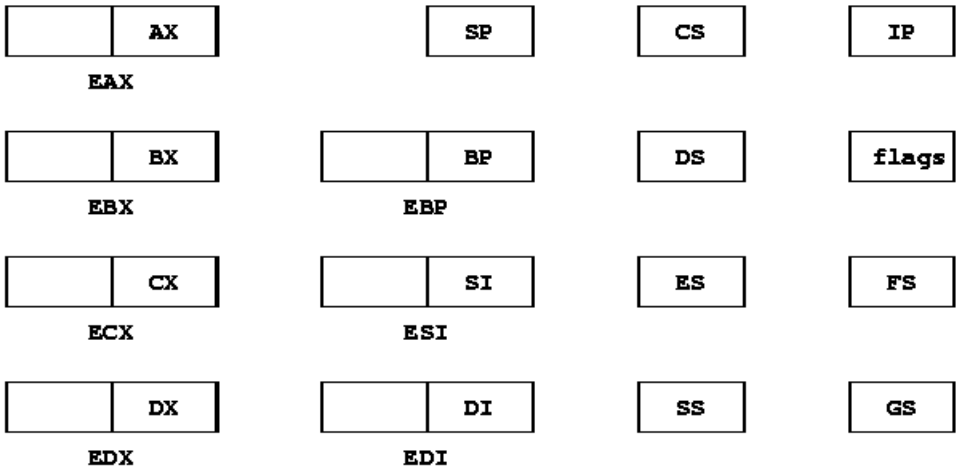


Figura 10. ***Registros del procesador 80386 y superiores.***

Como podemos observar, se amplía el tamaño de los registros de datos (que pueden ser accedidos en fragmentos de 8, 16 ó 32 bits) y se añaden dos nuevos registros de segmento de propósito general (FS y GS). Algunos de los registros mostrados son en realidad de 32 bits (EIP en lugar de IP, ESP en vez de SP y EFLAGS), pero bajo el DOS no pueden ser empleados de manera directa.

2.6.- EJEMPLO: CÓDIGO MÁQUINA VS. MNEMÓNICOS.

A pesar de que aún no hemos explicado el conjunto de instrucciones del procesador, exponemos el siguiente ejemplo que resulta meramente ilustrativo. La utilidad de este programa es dejar claro que lo único que maneja el procesador son números, aunque nosotros utilicemos símbolos (mnemónicos) que nos ayudan a entender dichos números.

```
14D3:7A10  B9 D0 07   MOV CX,7D0H          ; CX = 7D0H (2000 decimal = 7D0 hexadecimal)
14D3:7A13  B8 00 B0   MOV AX,0B800h        ; segmento de la memoria de pantalla
14D3:7A16  8E D8      MOV DS,AX            ; apuntar segmento de datos a la misma
14D3:7A18  BB 00 00   MOV BX,0             ; apuntar al primer carácter ASCII de la pant.
14D3:7A1B  C6 07 20   MOV BYTE PTR [BX],32; BYTE PTR para indicar que 32 es de 8 bits
14D3:7A1E  43         INC BX               ; BX=BX+1 -> apuntar al byte de color
14D3:7A1F  43         INC BX               ; BX=BX+1 -> apuntar al siguiente carácter
14D3:7A20  49         DEC CX               ; CX=CX-1 -> queda un carácter menos
14D3:7A21  75 F8      JNZ –8               ; si CX no es 0, ir 8 bytes atrás (14D3:7A1B)
```

El programa quedaría en memoria de esta manera: la primera columna indica la dirección de memoria donde está el programa que se ejecuta (CS = 14D3h e IP = 7A10h al inicio). La segunda columna es el código máquina que ejecuta el procesador. Como puede apreciarse la longitud en bytes de las instrucciones varía. La tercera columna contiene el mnemónico de las instrucciones. Todo lo que sigue a un punto y coma son comentarios.

3.- 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
CONTROL DE FLAGS
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. 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).

3.1.- Codificación de las instrucciones.

Algunas instrucciones tienen un objetivo muy específico por lo que un código de máquina de un byte es apropiado. Por ejemplo:

CÓDIGO DE MÁQUINA	INSTRUCCIÓN SIMBÓLICA
40	INC AX
FD	STD

Otras instrucciones que hacen referencia a memoria, incluyen operandos inmediatos o acceso a registros son más complicadas y requieren dos o más bytes de código de máquina. En general las instrucciones se codifican sobre 4 campos y tienen un tamaño de 1 a 6 bytes. Los cuatro campos son:

- CÓDIGO DE OPERACIÓN (Siempre aparece)
- MODO DE DIRECCIONAMIENTO (byte EA)
- DESPLAZAMIENTO DEL DATO
- VALOR INMEDIATO

C. OP. (6 bits)	D	W	MOD	R	E	G	R	/	M	DES.(8 ó 16 bits)	INM.(8 ó 16 bits)

El bit D de dirección vale 0 si la fuente está en el campo REG y 1 si es el destino el que figura en el campo REG. El bit W indica un acceso en palabra o en octeto. Los campos MOD y R/M especifican el modo de direccionamiento. Si MOD vale 11 indica un registro y si vale 00, 01 ó 10 un desplazamiento. R/M contiene el segundo registro o el tipo de direccionamiento a memoria. El campo REG determina el registro. Los registros se codifican de la siguiente forma:

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Las distintas combinaciones de los tres bits R/M y los dos bits de MOD se muestran a continuación:

R/M	MOD = 00	MOD = 01|10	MOD=11 W=0	MOD=11 W=1
000	DS:[BX+SI]	DS:[BX+SI+D]	AL	AX
001	DS:[BX+DI]	DS:[BX+DI+D]	CL	CX
010	SS:[BP+SI]	SS:[BP+SI+D]	DL	DX
011	SS:[BP+DI]	SS:[BP+DI+D]	BL	BX
100	DS:[SI]	DS:[SI+D]	AH	SP
101	DS:[DI]	DS:[DI+D]	CH	BP
110	Directo	SS:[BP+D]	DH	SI
111	DS:[BX]	DS:[BX+D]	BH	DI

Nota: D será un desplazamiento de uno o dos bytes según MOD sea 01 ó 10.

Sea la instrucción `ADD AX, BX` cuyo código máquina es:

```
00000011  11 000 011
```

- Los seis primeros bits corresponden al código de operación.
- D = 1 ⇒ El destino está en REG.
- W = 1 ⇒ El ancho es de una palabra.
- MOD = 11 ⇒ El segundo operando es un registro.
- REG = 000 ⇒ El primer operando es el registro AX (D = 1).
- R/M = 011 ⇒ El segundo operando es el registro BX.

3.2.- 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**
Descripción	Transfiere datos entre dos registros o entre un registro y memoria, y permite llevar datos inmediatos a un registro o a memoria.
Banderas	No las afecta.
Formato	MOV {registro/memoria},{registro/memoria/inmediato}
Ejemplo	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**
Descripción	Intercambia datos entre dos registros o entre un registro y memoria.
Banderas	No las afecta.
Formato	XCHG {registro/memoria},{registro/memoria}
Ejemplo	XCHG AL,AH

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

Operación	**PUSH: Guardar en la pila**
Descripción	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.
Banderas	No las afecta.
Formato	PUSH {registro/memoria/inmediato(sólo 286 o posteriores)}
Ejemplo	PUSH DX

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

Operación	**PUSHF: Guardar en la pila las banderas**
Descripción	Guarda en la pila el contenido del registro de banderas para su uso posterior.
Banderas	No las afecta.
Formato	PUSHF (sin operandos)
Ejemplo	PUSHF

Operación	**POPF: Sacar de la pila las banderas**
Descripción	Saca una palabra de la pila y la manda al registro de estado.
Banderas	Afecta a todas.
Formato	POPF (sin operandos)
Ejemplo	POPF

Operación	**LEA: Cargar dirección efectiva**
Descripción	Carga una dirección cercana en un registro.
Banderas	No las afecta.
Formato	LEA registro, memoria
Ejemplo	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.

3.3.- 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	**ADC: Sumar con acarreo**
Descripción	Por lo común es usado en suma de múltiples palabras binarias para acarrear un bit en el siguiente paso de la operación. ADC suma el contenido de la bandera CF al primer operando y después suma el segundo operando al primero, al igual que ADD.
Banderas	AF, CF, OF, PF, SF y ZF.
Formato	ADC {registro/memoria},{registro/memoria, inmediato}
Ejemplo	ADC AX,CX

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	SUB AL,CL

Operación	**SBB: Restar con acarreo**
Descripción	Normalmente, se usa esta operación en la resta binaria de múltiples palabras para acarrear el bit uno de desbordamiento al siguiente paso de la aritmética. SBB resta primero el contenido de la bandera CF del primer operando y después el segundo operando del primero, de manera similar a SUB.
Banderas	AF, CF, OF, PF, SF y ZF.
Formato	SBB {registro/memoria},{registro/memoria/inmediato}
Ejemplo	SBB AX,CX

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 $\Rightarrow$ AH $\neq$ 0 Ó DX $\neq$ 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
Banderas	CF y OF (AF, PF, SF y ZF quedan indefinidas). CF = OF = 1 $\Rightarrow$ AH $\neq$ 0 Ó DX $\neq$ 0.
Formato	IMUL {registro/memoria} (Ver tabla)
Ejemplo	--

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

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**
Descripción	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.
Banderas	(AF, CF, OF, PF, SF y ZF quedan indefinidas)
Formato	IDIV {registro/memoria}
Ejemplo	--

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

3.4.- 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.

3.4.1.- 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

Operación	**TEST: Examinar bits**
Descripción	Operación similar a CMP pero a nivel de bits. TEST realiza un AND lógico de los dos operandos para establecer el valor de las banderas pero sin almacenar el resultado.
Banderas	CF (0), OF (0), PF, SF y ZF (AF queda indefinida).
Formato	TEST {registro/memoria},{registro/memoria/inmediato}
Ejemplo	TEST AL,11110000b

3.4.2.- Instrucciones de Manejo de Bits.

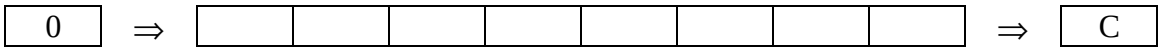
El formato general de este tipo de instrucciones es el siguiente:

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

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

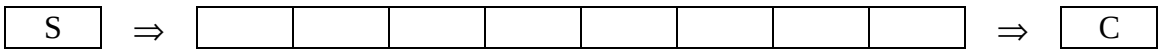
SHR: Desplazamiento lógico a la derecha

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.



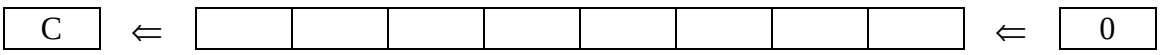
SAR: Desplazamiento aritmético a la derecha

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

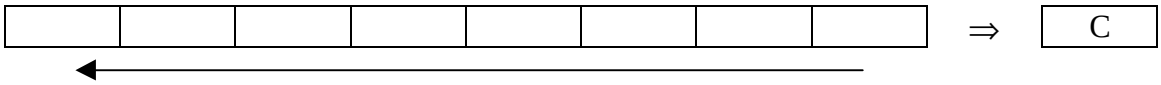
Se mueven todos los bits a la izquierda, en el bit menos significativo se mete un cero, y el bit que sale por la izquierda pasa a la bandera CF.



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

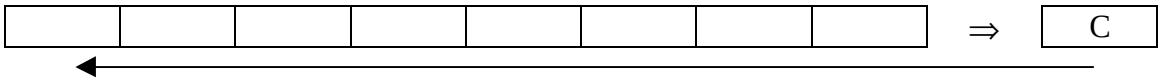
ROR: Rotación lógica a la derecha

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 .



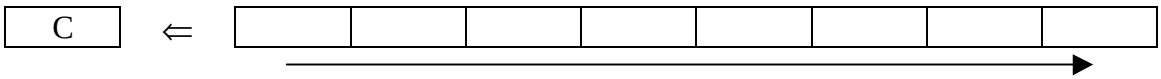
RCR: Rotación a la derecha con acarreo

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.



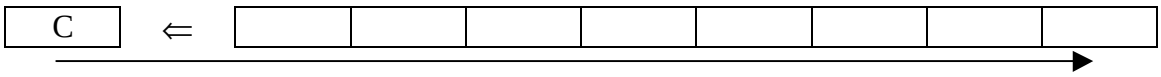
ROL: Rotación lógica a la izquierda

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.



3.5.- 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.

3.5.1.- 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 <> O
JLE/JNG	Salto si menor o igual	Z = 1 o S <> O

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

3.5.2.- Instrucciones de Transferencia de Control Incondicionales.

Operación	**JMP: Salto incondicional**
Descripción	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).
Banderas	No las afecta.
Formato	JMP {registro/memoria}
Ejemplo	JMP FINAL

3.5.3.- 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**
Descripción	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.
Banderas	No las afecta.
Formato	LOOP etiqueta
Ejemplo	MOV CX,num_veces Etiqueta: *Instrucciones del bucle* LOOP Etiqueta

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

3.5.4.- 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**
Descripción	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.
Banderas	No las afecta.
Formato	CALL {registro/memoria}
Ejemplo	CALL PROCEDIMIENTO

Operación	**RET: Regresar de un procedimiento**
Descripción	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.
Banderas	No las afecta.
Formato	RET [VALOR POP]
Ejemplo	RET

3.6.- 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/MOVSW: Mover cadena**
Descripción	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.
Banderas	No las afecta.
Formato	[REP] MOVSB/MOVSW
Ejemplo	MOV CX,3 REP MOVSB

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

Operación	**LODSB/LODSW: Cargar un byte/palabra**
Descripción	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.
Banderas	No las afecta.
Formato	LODSB/LODSW
Ejemplo	LODSB/LODSW

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

3.7.- Instrucciones de Control de Flags.

Estas instrucciones permiten manipular los bits del registro de estado:

Mnemónico	Descripción de la Operación
CLD	DF = 0
STD	DF = 1
CLI	IF = 0
STI	IF = 1
CLC	CF = 0
STC	CF = 1
CMC	Complementar CF

3.8.- 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**
Descripción	Transfiere desde un puerto de entrada un byte a AL o una palabra a AX
Banderas	No las afecta.
Formato	IN {AX/AL},{número_puerto/DX} Número_puerto ∈ [0..255] DX ∈ [0..65535]
Ejemplo	IN AL, 0Bh

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

4.- PROGRAMACIÓN DEL PC EN ENSAMBLADOR SOBRE DOS.

4.0.- INTRODUCCIÓN. MODELO DE TRES CAPAS.

¿Por qué es importante conocer el lenguaje ensamblador de una máquina?

- Un programa escrito en lenguaje ensamblador requiere considerablemente menos memoria y se ejecuta más rápidamente que un programa escrito en un lenguaje de alto nivel como Pascal o C.
- El lenguaje ensamblador ofrece al programador la posibilidad de realizar tareas muy específicas que sería muy difícil llevar a cabo en un lenguaje de alto nivel.
- El conocimiento del lenguaje ensamblador permite una comprensión de la arquitectura de la máquina que ningún lenguaje de alto nivel puede ofrecer.
- Desarrollar aplicaciones en lenguajes de alto nivel resulta mucho más productivo que hacerlo en ensamblador, pero este último resulta especialmente atractivo cuando hay que optimizar determinadas rutinas que suponen un cuello de botella para el rendimiento del sistema.
- Los programas residentes y las rutinas de servicio de interrupción casi siempre se escriben en ensamblador.
- La carga inicial de un S.O. debe realizarse en ensamblador, pues hacerlo con un lenguaje de alto nivel supondría usar instrucciones que en ese momento no pueden ser ejecutadas por la máquina.

Modelo de tres capas.

El DOS es un sistema operativo que proporciona un acceso general e independiente a los diferentes recursos que ofrece el sistema. Los dispositivos controlados son de muy diferente naturaleza. El DOS ofrece rutinas que permiten manejarlos sin acceder a ellos directamente. Las funciones principales del DOS son:

- Administración de la memoria.
- Administración de archivos y directorios.
- Gestión de E/S.
- Carga y ejecución de programas.
- Manejo de interrupciones del DOS.

Los tres elementos fundamentales del DOS son IO.SYS, MSDOS.SYS y COMMAND.COM (estos ficheros suelen encontrarse en el directorio raíz del disco duro; los dos primeros ficheros tienen activados los atributos de sistema y oculto):

- IO.SYS contiene algunas de las funciones de inicialización necesarias cuando se arranca la máquina, así como diversos controladores y rutinas para gestionar la E/S.
- MSDOS.SYS actúa como núcleo (*kernel*) del DOS y se ocupa de la administración de archivos, de memoria y de E/S.

- COMMAND.COM es un procesador de comandos (*shell*) que hace de intermediario entre el usuario y el sistema operativo.

El Sistema Básico de Entrada/Salida (**BIOS**) se encuentra en un chip de memoria de sólo lectura, y contiene un conjunto de rutinas para dar soporte a los diferentes dispositivos.

Cuando un programa de usuario solicita un servicio del DOS, éste podría transferir la solicitud al BIOS, el cual a su vez accederá al dispositivo solicitado haciendo uso de las rutinas que para ello están disponibles en el chip ROM. Algunas veces, el programa puede realizar la petición directamente al BIOS, frecuentemente para servicios de pantalla y teclado. En ocasiones el programa podría manipular el hardware directamente.

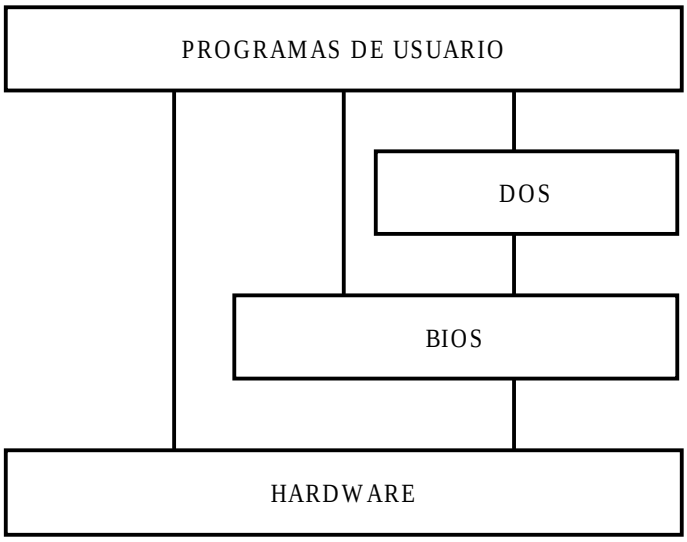


Figura 11. ***Modelo de tres capas.***

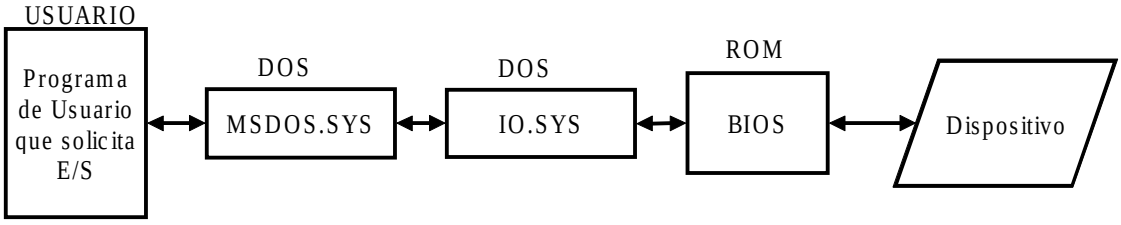


Figura 12. ***Interfaz DOS-BIOS.***

4.1.- DIRECTIVAS DEL ENSAMBLADOR.

El lenguaje 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. Las directivas más comunes se explican en este apartado.

4.1.1.- Directivas para listar : PAGE y TITLE.

Las directivas PAGE y TITLE controlan el formato de un listado de un programa en ensamblador. No tienen ningún efecto sobre la ejecución del programa.

Directiva	**PAGE**
Descripción	Determina al comienzo del programa el número máximo de líneas para listar en una página, así como el número de columnas. El valor por defecto es cincuenta líneas y ochenta columnas.
Formato	PAGE [longitud] [, ancho]
Ejemplo	PAGE 60, 100

Directiva	**TITLE**
Descripción	Se emplea para hacer que aparezca un título para el programa en la línea dos de cada página del listado.
Formato	TITLE texto
Ejemplo	TITLE PASM Programa en ensamblador

4.1.2.- Directivas para declaración y manejo de segmentos.

4.1.2.1.- Directiva ASSUME.

Directiva	**ASSUME**
Descripción	Cada programa utiliza el registro SS para direccionar la pila, el DS para el segmento de datos y el CS para el segmento de código. Esta directiva se usa para determinar el propósito de cada segmento del programa.
Formato	ASSUME SS:segpila, DS:segdatos, CS:segcodigo, ES:segextra
Ejemplo	ASSUME SS:STACKSG, DS:DATASG, CS:CODESG, ES:EXSG

4.1.2.2.- Directiva SEGMENT.

Directiva	**SEGMENT**			
Descripción	Esta directiva define un segmento para el programa (habrá que definir tantos como tengamos).			
Formato	NOMBRE	OPERACIÓN	OPERANDO	COMENTARIO
	Nombre	SEGMENT	[opciones]	;Inicio Segmento
	. . .			
	Nombre	ENDS		;Fin Segmento

[*opciones*]	ALINEACIÓN: indica el límite donde comienza el segmento. El valor por defecto es PARA que alinea el segmento con un límite de párrafo, es decir, de manera que la dirección inicial es divisible por 16. COMBINAR: necesaria si se combina el segmento con otros segmentos cuando son *enlazados* después de ensamblar. El segmento de pila suele incluir en su definición STACK. Otros valores admitidos son COMMON y PUBLIC. CLASE: determina si se agruparán los segmentos al enlazar. Por convención se emplea 'code' para el segmento de código, 'data' para el segmento de datos, y 'stack' para el segmento de pila.

4.1.2.3.- Directiva PROC.

Directiva	**PROC**			
Descripción	Para definir cada uno de los procedimientos incluidos en el segmento de código.			
Formato	NOMBRE	OPERACIÓN	OPERANDO	COMENTARIO
	Nombre Segmento	SEGMENT	[opciones]	;Inicio
	NombreProc	PROC	NEAR/FAR	
	. . .			
	NombreProc	ENDP		
	Nombre	ENDS		;Fin Segmento

4.1.2.4.- Directiva .MODEL.

Directiva	**.MODEL (nótese ".")**
Descripción	Especifica el modelo de memoria utilizado (véase sección 4.2). Lenguaje puede ser C, Pascal, Assembler, FORTRAN, etc.
Formato	.MODEL modelo[,lenguaje]
Ejemplo	.MODEL Compact

4.1.2.5.- Directiva .DATA.

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

4.1.1.6.- Directiva .CODE.

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

4.1.2.7.- Directiva .STACK.

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

4.1.2.8.- Directiva END.

Directiva	**END**
Descripción	Finaliza todo el programa.
Formato	OPERACIÓN OPERANDO END [dir_inicial] ; Generalmente etiqueta del PROC principal.

4.1.3.- 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 incializados a 14.
DATO8	DB	'Cadena de caracteres'	Cadena de caracteres.
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.

4.1.4.- 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.
- *Etiqueta de instrucción*: se refiere a la dirección de una instrucción.

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).

4.1.5.- 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.

4.1.6.- 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.

4.1.7.- El operador PTR.

La directiva PTR indica que el dato a continuación es un puntero (véase sección 4.2). Sólo se usa en casos en los que hay una ambigüedad en el tamaño de los datos que una instrucción debe manejar. Normalmente, con el tipo de las etiquetas se puede saber de qué longitud es una determinada variable. Sin embargo, hay veces en las que no es posible. Por ejemplo, consideremos que guardamos la dirección de un array en el área de datos. Nos interesa conocer por separado las dos partes de la dirección: el desplazamiento y el segmento. Para ello, podemos utilizar dos variables:

```
.DATA
OFS  DW   ?
SEG  DW   ?
```

La dirección completa está formada por las dos palabras en el orden en que están declaradas (recuérdese el principio de almacenamiento inverso), o sea, tomando 4 bytes (2 palabras) a partir de la dirección de OFS. La instrucción LES carga en el registro ES y en otro registro especificado una dirección *far* de memoria. Así:

```
LES  DI,OFS
```

carga en ES:DI la dirección almacenada en el par <OFS,SEG>. Sin embargo, OFS es un puntero a una única palabra, por lo que el ensamblador informará de ello. El operador PTR viene a ayudarnos, ya que nos permite cambiar el *significado* de la variable OFS y convertirla en un puntero a una *doble palabra* de la siguiente manera:

```
LES  DI,DWORD PTR OFS
```

Se indica que en la operación, OFS es un *puntero* a una doble palabra. Los modificadores disponibles para PTR son los siguientes:

- BYTE, como, por ejemplo, `MOV AL,BYTE PTR[BX]`.
- WORD
- DWORD

4.2.- DIRECTIVAS, SEGMENTOS, PROCEDIMIENTOS Y PASO DE PARÁMETROS.

4.2.1.- Segmentos y modelos de memoria

Un programa en ensamblador puede, a través de las directivas anteriores, definir varios segmentos de datos o de código. Como se vio, en la arquitectura 8086/88, un segmento puede albergar hasta 64 Kbytes, por lo cual, definir varios segmentos de

código no es normal para un programa escrito enteramente en ensamblador[1]. Aún así, un programador, ya sea por legibilidad, comodidad o necesidad, puede definir varios segmentos tanto de código como de datos.

Si el programa sólo posee un segmento de datos, podemos declararlo de la siguiente manera:

```
DATASG    SEGMENT   PARA PUBLIC 'DATA'
          DATO      DW ?
          ; MÁS DATOS DEFINIDOS AQUÍ ...
DATASG    ENDS
```

Este segmento de datos determinará dónde se pueden encontrar todos los datos del programa, así, si al inicio del programa asignamos a DS el valor del segmento de DATOS, haremos que todos los datos puedan ser accedidos a través del registro de segmento DS. A este tipo de programas se le llama programas con **datos locales** (datos NEAR):

```
CODESG    SEGMENT   PARA PUBLIC    'Code'
BEGIN     PROC      FAR
          ASSUME    SS: STACKSG, DS: DATASG, CS: CODESG
          MOV       AX, DATASG
          MOV       DS, AX
          ; INSTRUCCIONES DEL PROCEDIMIENTO PRINCIPAL BEGIN
          BEGIN     ENDP
CODESG    ENDS
          END       BEGIN
```

Nótese que no se puede asignar valores inmediatos directamente a DS. La directiva ASSUME indica al ensamblador que el registro DS tiene el mismo valor que el segmento DATOS.

Igualmente, un programa puede definir varios segmentos de código o sólo uno. En el último caso, se dirá que el programa es de **código local**, es decir, las llamadas a procedimientos ocurren siempre dentro del mismo segmento, por lo que la dirección de comienzo de cualquiera de ellos se puede especificar como un desplazamiento en el propio segmento de código. Estos son procedimientos NEAR. En el caso de que el programa defina varios segmentos de código, un procedimiento queda especificado no sólo por el desplazamiento en su segmento de código, sino también por la dirección de comienzo del segmento en el que está definido (procedimientos FAR). Declarando un procedimiento FAR, especificamos que ese procedimiento puede ser llamado por procedimientos que están definidos en otros segmentos de código. Declarándolo como NEAR sólo podrá ser llamado por procedimientos definidos en su mismo segmento de código.

¿Y para qué esta distinción entre programas con código local o no, y programas con datos locales o no? Esta distinción es necesaria para especificar la *dirección* de cada

[1] Suponiendo que cada instrucción ocupa una media de 4 bytes, harían falta 16384 (65535 / 4) instrucciones para llenar un segmento de código.

uno de los elementos del programa. En un programa con datos locales, una etiqueta de datos (como en el ejemplo anterior, DATO) corresponde a una palabra: el desplazamiento del dato en el segmento de datos:

```
MOV  BX,OFFSET DATO
MOV  [BX],3444
```

En BX se carga la *dirección* de DATO, y posteriormente se actualiza su *valor* a 3444 (decimal)[2]. Este ejemplo nos sirve para ver la diferencia entre la *dirección* y el *valor* de una variable en ensamblador.

El caso de un programa con datos no locales, es decir, que define más de un segmento de datos, es algo distinto. Supongamos que definimos dos segmentos de datos en nuestro programa. Éstos se llamarán DATOS1 y DATOS2:

```
DATOS1    SEGMENT PARA PUBLIC 'DATA'
          DATO1     DW ?
DATOS1    ENDS
DATOS2    SEGMENT PARA PUBLIC 'DATA'
          DATO2     DW ?
DATOS2    ENDS
```

Cada uno de ellos alberga un dato, donde DATO1 está en el segmento DATOS1 y DATO2 está en DATOS2. En este caso podemos elegir cualquiera de los dos segmentos como el segmento de datos apuntado por DS. Aún así, en este caso, la *dirección* de una variable ya no se puede especificar sólo con el desplazamiento en su segmento de datos, sino también con el segmento en el que está definido. Por tanto, para actualizar, por ejemplo, DATO2, debemos ejecutar una secuencia de instrucciones parecida a esta:

```
MOV  AX,DATOS2
MOV  ES,AX    ;SE PODRÍA UTILIZAR DS
MOV  BX,OFFSET DATO2[3]
MOV  ES:[BX],3444
```

Por lo tanto, la dirección de cualquier variable en un programa que no es de datos locales es un par (*Segmento*, *Desplazamiento*). En el caso de DATO2 es: (DATOS2,OFFSET DATO2).

Igual que ocurre con los datos, ocurre con los procedimientos. Los procedimientos que están declarados como FAR requieren que se especifique el segmento y el desplazamiento a la hora de invocarlos con una instrucción CALL. Además, está claro que la dirección de retorno (a la que el procedimiento retornará con una instrucción RET) también estará formada por un par (*Segmento*, *Desplazamiento*), que se extraerá adecuadamente de la pila de forma automática por la instrucción RET.

[2] El ejemplo anterior también se podría haber escrito simplemente como MOV `DATO,3444`.

[3] También se podría utilizar aquí LEA `BX,DATO2`, que es totalmente equivalente.

La siguiente tabla[4] resume los distintos tipos de programas según sus **modelos de memoria** (datos locales, código local, etc.):

MODELO	CARACTERÍSTICAS
TINY	Datos y código cogen en un solo segmento de 64K. Todas las direcciones tanto de datos como de procedimientos son NEAR (sólo especifican un desplazamiento dentro del segmento).
SMALL	Un segmento de datos y otro de código. Todas las direcciones son NEAR, aunque hay un segmento distinto para datos y otro para código.
COMPACT	Múltiples segmentos de datos y un único segmento de código. Las direcciones de datos son FAR -especifican un par (*Segmento,Desplazamiento*)- y las direcciones de código (procedimientos) son NEAR. Este modelo y el anterior son los que normalmente se utilizan.
LARGE	Múltiples segmentos de datos y de código. Tanto los datos como los procedimientos tienen direcciones FAR.
HUGE	Múltiples segmentos de datos y de código. Los segmentos pueden pasar 64K, pero haciendo operaciones especiales de normalización de direcciones. Estas normalizaciones son realizadas por compiladores de C.

Para facilitar el uso de segmentos y de modelos de memoria (que, más tarde, como veremos en la sección 4.12.- INTERFAZ DE ENSAMBLADOR CON OTROS LENGUAJES DE PROGRAMACIÓN., nos permitirán enlazar ensamblador con C), los programas ensambladores Macro-Assembler (MASM) a partir de la versión 5 y Turbo-Assembler (TASM) a partir de la versión 3, ofrecen un conjunto de directivas, llamadas **directivas simplificadas**, que sirven para especificar de una manera estándar el contenido de un segmento.

Por ejemplo, la directiva .CODE equivale a una declaración de segmento como _TEXT SEGMENT WORD PUBLIC 'CODE' es decir, se crea un segmento de nombre _TEXT alineado en una dirección de palabra y que contiene código. Igualmente existen las directivas .DATA, DATA?, .STACK, etc. La directiva .DATA? declara un segmento de variables no inicializadas (declaradas con DW ?, por ejemplo). La siguiente tabla muestra el conjunto de directivas disponibles para el modelo **COMPACT** de memoria:

Directiva	Segmento	Alineamiento	Exportación	Descripción
.CODE	_TEXT	WORD	PUBLIC	'CODE'
.FARDATA	FAR_DATA	PARA	PRIVATE	'FAR_DATA'
.FARDATA?	FAR_BSS	PARA	PRIVATE	'FAR_BSS'
.DATA	_DATA	WORD	PUBLIC	'DATA'
.CONST	CONST	WORD	PUBLIC	'CONST'
.DATA?	_BSS	WORD	PUBLIC	'BSS'
.STACK	STACK	PARA	STACK	'STACK'

Con estas directivas, un programa puede definir de forma estándar los segmentos que utiliza. Como ejemplo, veamos el siguiente programa:

[4] Tomada de HelpPC 2.10. Copyright 1991 David Jurgens.

```
.MODEL COMPACT

.DATA
     DATO      DB 0

.CODE
     ; CÓDIGO DEL PROGRAMA
```

Define un segmento de código y otro de datos ajustándose al modelo COMPACT de memoria. Las directivas se pueden utilizar tantas veces como se necesite, y continúan donde la anterior definición para ese segmento terminó.

La utilización de los modelos de memoria determina si un programa es de datos locales o de código local. El modelo de memoria establece de forma predefinida si las *direcciones* de los datos y del código son FAR o NEAR. Si los segmentos se definen sin utilizar las directivas simplificadas, es el programador el que se encarga de saber qué datos (y procedimientos) son accedidos a través de direcciones NEAR o FAR.

4.2.2.- Paso de parámetros a procedimientos

Los modelos de memoria también son importantes a la hora del paso de parámetros a los procedimientos, ya que definen el *tamaño* de las direcciones: una etiqueta de datos en un programa con datos locales ocupa sólo una palabra (el desplazamiento dentro del segmento); una etiqueta de datos en un programa con datos no locales corresponde a dos palabras (el segmento donde está definida y el desplazamiento dentro del mismo).

Es importante distinguir aquí que existen dos tipos de parámetros a procedimientos: parámetros que especifican una etiqueta de datos (***dirección***), y parámetros que especifican el contenido de una etiqueta de datos (***valor***). Los primeros también reciben el nombre de **punteros** o **apuntadores**.

Para ilustrar esta diferencia, podemos ver un ejemplo sencillo con un procedimiento que realiza la suma de dos palabras. El procedimiento devuelve en el registro AX el valor de la suma de los dos operandos dados como parámetros. En una primera versión, el procedimiento aceptará el ***valor*** de los operandos en los registros BX y CX. Este ejemplo también nos servirá para ilustrar el uso de las directivas simplificadas:

```
.MODEL SMALL
.DATA
     DATO1     DW 25
     DATO2     DW 33

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

```
SUMA PROC
     MOV        AX,BX
     ADD        AX,CX
     RET
SUMA ENDP
     END  MAIN
```

En esta primera versión, el procedimiento SUM acepta el ***valor*** de los datos a sumar. El siguiente programa, en cambio, implementa un procedimiento SUMA que acepta la ***dirección*** en la que debe buscar cada uno de los operandos. BX y CX se convierten entonces en ***punteros***. Nótese que sólo ocupan una palabra porque utilizamos el modelo SMALL de memoria. Un registro adicional para cada uno de los parámetros habría sido necesario en caso de utilizar un modelo de memoria que no sea de datos locales:

```
.MODEL SMALL

.DATA
     DATO1      DW 25
     DATO2      DW 33

.CODE

MAIN:
     MOV        BX, OFFSET DATO1
     MOV        CX, OFFSET DATO2
     CALL       SUMA

     ; ¡¡¡LA SUMA ESTÁ EN AX!!!

SUMA PROC
     MOV        AX,[BX]
     MOV        BX,CX
     ADD        AX,[BX]
     RET
SUMA ENDP
     END MAIN
```

Nótese el cambio en las zonas recuadradas. BX y CX se convierten en punteros que nos sirven para especificar la ***dirección*** en la que las variables a sumar se pueden encontrar.

Hay tres formas de pasar los parámetros a los procedimientos, y cada una de ellas tiene sus ventajas y sus inconvenientes. A continuación se explica cada una de ellas.

4.2.2.1.- 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 este 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.

Para realizar llamadas a las funciones del DOS y del BIOS, hay que pasar los parámetros en registros. Por ejemplo, para visualizar una cadena de caracteres, se usa la función 21H del DOS:

```
CADENA    DB   'Esto se verá en pantalla','$'
. . .
MOV  AH,09H          ; Petición de pintar en pantalla
LEA  DX, cadena      ; Carga la dirección de la cadena
                     ; en DX
INT  21H             ; Ejecuta la función
```

El procedimiento SUMA anterior también es un ejemplo de este tipo de paso de parámetros.

4.2.2.2.- 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
...
```

4.2.2.3.- Pila.

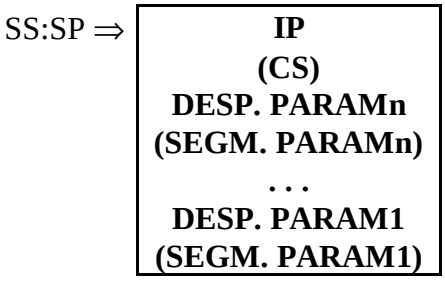
La tercera posibilidad para pasar los parámetros es hacer uso de la pila. Ésta es la que generalmente se usa. Antes de llamar a un procedimiento, el llamante apila (introduce en la pila) los parámetros del procedimiento llamado. A continuación, al ejecutar el correspondiente `CALL <procedimiento>`, la dirección de retorno también se guarda en la pila. 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.

Paso de parámetros

Como se comentó, tanto los procedimientos como los datos pueden tener direcciones FAR o NEAR. En el primer caso se deben también apilar los registros de segmento de cada parámetro junto con el desplazamiento. En el segundo caso sólo el desplazamiento.

```
(PUSH DS)           ; Guardar el segmento de PARAM1
LEA AX,PARAM1       ; En AX el desplazamiento de PARAM1
PUSH AX             ; Guardar el desplazamiento de PARAM1

(PUSH DS)           ; Guardar el segmento de PARAM2
LEA AX,PARAM2       ; En AX el desplazamiento de PARAM2
PUSH AX             ; Guardar el desplazamiento de PARAM2

. . .

(PUSH DS)           ; Guardar el segmento de PARAMn
LEA AX,PARAMn       ; En AX el desplazamiento de PARAMn
PUSH AX             ; Guardar el desplazamiento de PARAMn

CALL      PROCEDIMIENTO  ; Llama al procedimiento (FAR)
. . .
```

La llamada se generará FAR de forma automática si el procedimiento está definido en otro segmento de código. En otro caso será NEAR, y la parte PUSH CS no aparecerá. La pila quedaría como sigue. Los elementos encerrados entre paréntesis identifican a los que no estarían presentes caso de que el programa fuese de datos locales y/o de código local:



Recogida de parámetros

Durante el desarrollo del procedimiento, la pila puede crecer más debido a que se vayan apilando más datos temporales o a que se generen variables locales. Entonces, ¿cómo conseguimos acceder a los parámetros a través de una dirección fija incluso si el principio de la pila cambia durante la ejecución del procedimiento? La respuesta está en el registro BP. El registro BP apuntará a un punto fijo en la pila desde el que, a través de direcciones relativas a BP (esto es, BP+xxx ó BP-xxx) se podrán acceder tanto a los parámetros como a las variables locales definidas en el procedimiento. Esto se consigue haciendo lo siguiente:

- Se apila el registro BP: `PUSH BP`
- Se iguala BP a SP para poder acceder mediante él a los elementos que hay en la pila: `MOV BP,SP`
- Se accederá a los parámetros de la pila mediante BP.

Así, BP apunta al valor almacenado en la pila de BP; BP + 2 es la dirección de retorno del procedimiento; etc. El procedimiento se esquematiza con dos figuras a continuación. La primera se refiere a un procedimiento que se llamó con una llamada FAR y que posee parámetros que son ***punteros FAR***. La segunda es igual, solo que todo es NEAR.

A) FAR

BP ⇒	**BP**
BP + 2 ⇒	**IP**
BP + 4 ⇒	**CS**
BP + 6 ⇒	**DESP. PARAMn**
BP + 8 ⇒	**SEGM. PARAMn**
	. . .
BP + 6 + 4 * (n –1) ⇒	**DESP. PARAM1**
BP + 8 + 4 * (n –1) ⇒	**SEGM. PARAM1**

B) NEAR

BP ⇒	**BP**
BP + 2 ⇒	**IP**
BP + 4 ⇒	**DESP. PARAMn**
	. . .
BP + 4 + 2 * (n –1) ⇒	**DESP. PARAM1**

En este ejemplo, los parámetros se pasan al procedimiento en forma de posiciones de memoria o ***punteros***, pero también sería posible pasar directamente en la pila los valores de los parámetros.

Variables locales

También es posible declarar variables locales al procedimiento en la pila (como de hecho lo realizan la mayoría de los compiladores de C o Pascal). Esto se consigue restando una cantidad suficiente al registro SP. Por ejemplo, imaginemos que necesitamos reservar espacio para tres *word*s. Sólo tenemos que restar la cantidad de bytes a SP: `SUB SP,6`

¿Cómo se accederá entonces a cada una de estas tres variables? Pues con desplazamientos *negativos* sobre BP: la primera variable está en BP-2, la segunda en BP-4 y la tercera en BP-6. Por ejemplo, para inicializar la primera variable al valor 3222, podemos escribir: `MOV [BP-2],3222`

Retorno de un procedimiento

Para retornar de un procedimiento se debe restaurar la pila al estado de antes de la llamada. Para ello, basta con asignar a SP el valor de BP (que, como vimos, apunta al anterior valor de BP en la pila). Después, hay que *desapilar* BP para devolverlo a su valor anterior. Finalmente, debemos regresar con RET indicándole además, cuántos bytes debe eliminar de la pila, correspondientes a los parámetros. Nótese que ambos ejemplos presentados poseen un distinto número de bytes en los parámetros. Así, el código podría ser:

```
MOV SP,BP
POP BP
RET <NNN>
```

Ejemplo

Finalmente, el mismo procedimiento SUMA se implementará con paso por pila:

```
.MODEL SMALL

.DATA
     DATO1     DW 25
     DATO2     DW 33

.CODE

MAIN:
     MOV       AX, OFFSET DATO1
     PUSH      AX
     MOV       AX, OFFSET DATO2
     PUSH      AX
     CALL      SUMA

     ; ¡¡¡LA SUMA ESTÁ EN AX!!!

SUMA PROC
     PUSH      BP              ; PRÓLOGO
     MOV       BP,SP

     MOV       BX,[BP + 4]     ; OBTENER EL PRIMER PARÁMETRO
     MOV       AX,[BX]

     MOV       BX,[BP + 6]     ; SEGUNDO PARÁMETRO
     ADD       AX,[BX]

     MOV       SP,BP           ; EPÍLOGO: RESTAURAR EL
     POP       BP              ; ESTADO DE LA PILA

     RET       4
SUMA ENDP
     END MAIN
```

Aquí también, los parámetros son **punteros** a las variables reales. No se han añadido variables locales al no ser necesarias. Esta codificación puede parecer tediosa, pero es la que mejor integra el uso de parámetros (incluso un número variable de ellos), recursividad y variables locales.

4.3.- PREFIJO DE SEGMENTO DE PROGRAMA (PSP).

Al teclear el nombre de un fichero ejecutable, el DOS lo busca en el directorio activo (o en los definidos con PATH) y lleva a cabo las siguientes operaciones:

- Creación del **Segmento de programa**. Se determina la dirección de memoria libre más baja del TPA (Área de Programas Transitorios).

- Creación del **Segmento del entorno**. Se localiza un bloque de memoria libre y se copia en él las variables de entorno activas en COMMAND.COM. Este bloque de memoria se convierte en el entorno del proceso.

- Creación del **PSP**. El PSP está formado por los primeros 256 bytes del segmento de programa.

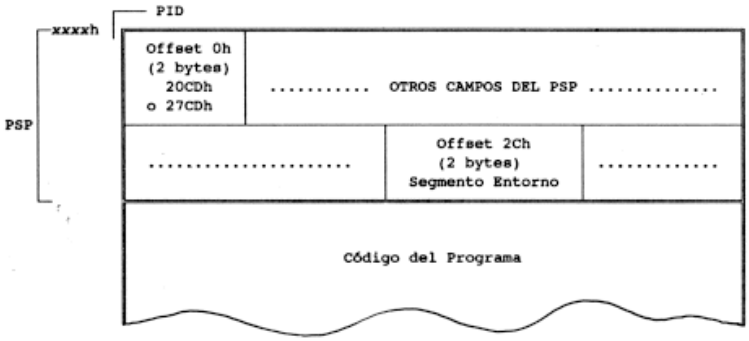


Figura 13. ***Estructura del Segmento de Programa.***

Nota: El código del programa se carga a partir del offset 100H, detrás del PSP.

4.3.1.- Algunos Campos del PSP.

PREFIJO DE SEGMENTO DE PROGRAMA

Offset	Longitud	Descripción
(1) 00H	2 bytes	Instrucción de la INT 20H (20CDh)
02H	2 bytes	Dirección del segmento siguiente a la memoria del programa.
04H	1 byte	(reservado; suele ser 0)
(1) 05H	5 bytes	Función despachadora del DOS.
(2) 06H	2 bytes	Número de bytes disponibles en el segmento de código del programa.
0AH	4 bytes	Dirección de la INT 22H (Lugar de Terminación).
0EH	4 bytes	Dirección de la INT 23H (Gestión de Ctrl–C).
12H	4 bytes	Dirección de la INT 24H (Gestión de Errores Críticos).
(2) 16H	2 bytes	PID del proceso padre.
(2) 18H	20 bytes	Tabla para 20 ficheros handle.
2CH	2 bytes	Dirección del segmento donde comienza el entorno
(2) 2EH	4 bytes	Dirección de la pila del proceso (SS:SP).
(2) 32H	2 bytes	Contador de handles. Número máximo de entradas en la tabla de handles.
(2) 34H	4 bytes	Dirección de la tabla de handles.
38H	24 bytes	(reservados)
(1) 50H	3 bytes	Instrucciones INT 21H (CD21) y RETF (CB).
53H	2 bytes	(reservados)
(2) 55H	7 bytes	Campos de extensión para el primer FCB por defecto.
(1) 5CH	16 bytes	Primer FCB por defecto.
(1) 6CH	20 bytes	Segundo FCB por defecto.
80H	128 bytes	Area de Transferencia de Disco (DTA) por defecto.
80H	1 byte	Longitud de los parámetros en la línea de comandos.
81H	127 bytes	Parámetros introducidos en la línea de comandos.

(1) : Campo obsoleto
(2) : Campo indocumentado

Figura 14. ***Estructura del PSP.***

A continuación se describirán aquellos campos del PSP más interesantes.

- **Campo 00H (2 bytes).** El primer campo del PSP contiene la instrucción que se encarga de finalizar la ejecución de un programa, aunque es aconsejable emplear la función 4CH de la Int 21H para el fin de ejecución. En programas .COM, en el tope de la pila se coloca la palabra 0H. Así, si un programa termina con RET, se retornará a la dirección de memoria CS:0, que es precisamente el primer campo del PSP, donde se guarda esta instrucción de terminación.
- **Campo 02H (2 bytes).** Este campo almacena la dirección del siguiente segmento de memoria libre[5], lo cual sirve para calcular el tamaño del bloque de memoria en el que se encuentra el PSP, es decir, el tamaño del bloque de memoria localizado para el programa.
- **Campo 0AH (4 bytes).** Almacena la dirección asociada a la interrupción 22H. Esta dirección indica el lugar donde se desvía el programa cuando finaliza su ejecución.
- **Campo 0EH (4 bytes).** Contiene la dirección de la rutina que toma el control cuando el usuario pulsa la tecla Ctrl-C.
- **Campo 12H (4 bytes).** Guarda la dirección de la rutina que asume el control del sistema cuando se produce un error crítico.

[5] Recuérdese la aritmética de segmentos de la sección 2.2. Si el programa ocupa 500H bytes (PSP + código), el siguiente **segmento** disponible es el CS+50H. Así, un salto de segmento significa un salto en offset de 16 (o 10H) bytes.

- **Campo 2CH (4 bytes).** Dirección del segmento donde comienza el entorno. El entorno es una zona de memoria que guarda las variables del entorno y su valor asociado. Para separar cada variable se utiliza un byte nulo (00H). El formato es el siguiente:

Nombre_Variable=Valor,00H

. . .

Nombre_Variable=Valor,00H,00H

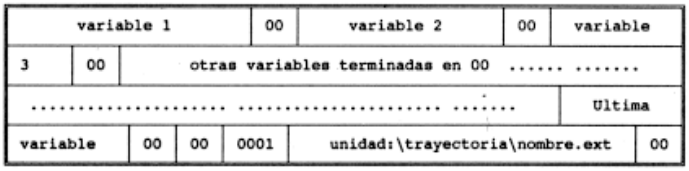


Figura 15. ***Estructura del Segmento de Entorno.***

A continuación hay una palabra (2 bytes) cuyo valor es 0001H si el propietario del PSP no es el COMMAND.COM y cualquier otro valor si lo es. Por último, hay una cadena ASCII que almacena la trayectoria y el nombre del fichero ejecutable dueño del PSP.

- **Campo 80H (128 bytes).** Aquí se guardan los parámetros introducidos en la línea de comandos tras el nombre del fichero ejecutable. En el offset 80H se guarda la longitud de los parámetros en la línea de comandos, y a partir del offset 81H los parámetros en sí. En esta cadena no se incluyen los caracteres de redirección '<' y '>'. La redirección de entrada y salida es transparente para las aplicaciones.

4.4.- INICIALIZAR UN PROGRAMA PARA SU EJECUCIÓN.

Veamos cómo hacerlo mediante el siguiente ejemplo:

```
1   PAGE 60,132
2   TITLE     PRIMERO Estructura de un programa .EXE
3   ;--------------------------------------------------
4   STACKSG   SEGMENT   PARA STACK 'Stack'
5   ...
6   STACKSG   ENDS
7   ;--------------------------------------------------
8   DATASG    SEGMENT   PARA 'Data'
9   ...
10  DATASG    ENDS
11  ;--------------------------------------------------
12  CODESG    SEGMENT   PARA 'Code'
13  BEGIN     PROC      FAR
14            ASSUME    SS:STACKSG,DS:DATASG,CS:CODESG
15            MOV       AX,DATASG      ;Se asigna DATASG
16            MOV       DS,AX          ;a DS
17            ...
18            MOV       AX,4C00H  ; Salida al DOS
19            INT       21H
20  BEGIN     ENDP
21  CODESG    ENDS
22            END       BEGIN
```

LÍNEA	EXPLICACIÓN
1	La directiva PAGE establece el formato del listado (60 líneas y 132 columnas)
2	La directiva TITLE identifica el nombre del programa PRIMERO.
3, 7 y 11	Comentarios.
4-6	Se define el segmento de pila STACKSG (no se muestra su contenido).
8-10	Se define el segmento de datos DATASG (no se muestra su contenido).
12-21	Se define el segmento de código CODESG.
13-20	El segmento de código consta de un único procedimiento, llamado BEGIN. Tareas: 1. Indicar al ensamblador qué segmentos asocia con los registros de segmentos. 2. Cargar el DS con la dirección del segmento de datos. 3. Finalizar la ejecución mediante la función 4C de la Int 21H.
22	Final del programa.

En caso de no finalizar con `MOV AX,4C00h` e `INT 21h`, se hará con la instrucción `RET` es responsabilidad del programador apilar DS y el valor 0 al principio del código (con lo cual se ejecutaría la instrucción contenida en el primer campo del PSP).

4.5.- TERMINAR LA EJECUCIÓN DE UN PROGRAMA.

Para finalizar la ejecución de los programas se utiliza la función 4CH de la interrupción 21H, cuyo objetivo es realizar una petición de terminación de la ejecución de un programa. También es posible utilizar esta operación para pasar un código de regreso en el registro AL, el cual podría ser utilizado para comprobaciones posteriores dentro de un archivo por lotes (ERRORLEVEL).

```
MOV  AH,4CH          ; Solicitud de terminación
MOV  AL,ret_prog     ; Código de retorno(opcional)
INT  21H             ; Salida al DOS
```

4.6.- DISCOS.

En un ordenador sólo es posible trabajar *directamente* con los datos de la memoria RAM. Pero la memoria RAM es volátil, es decir, pierde su contenido cuando se apaga el ordenador. Por este motivo, para conservar datos de manera permanente se utilizan dispositivos de almacenamiento tales como disquetes o discos duros. En cualquier caso, cabe distinguir tres elementos:

1. **Unidad de Disco.** Dispositivo mecánico formado por una pila de uno o más discos (plato), que rotan sobre un eje, y dos o más cabezas de lectura/escritura cuya misión es leer y escribir la información en el disco. En las unidades de disquetes el plato comienza a girar cada vez que se accede al disquete, mientras que en las unidades de disco duro el movimiento giratorio es constante durante todo el tiempo que está encendido el ordenador. El plato de un disco duro gira a una velocidad de entre 3.600 y 5.400 r.p.m.

2. **Controlador de Disco.** Dispositivo electrónico que establece la conexión entre el procesador y el disco. Por una parte, enlaza físicamente la unidad de disco con el bus de datos (camino por el que fluyen los datos dentro del ordenador). Y por otro lado, transforma las peticiones del sistema operativo en instrucciones especiales que actúan directamente sobre la unidad de disco.

3. **Disco.** Dispositivo magnético que almacena la información. Normalmente, los discos se componen de una superficie circular plana de plástico (disquete) o metal (disco duro) recubierta de algún óxido magnetizable. La información se graba sobre el disco mediante alteraciones de su superficie con un campo magnético.

A grandes rasgos, el funcionamiento de un disco es como sigue: el controlador envía a la unidad las órdenes necesarias para que desplace la cabeza y lea o escriba datos sobre el disco.

En todo disco, ya sea un disquete o un disco integrado en una unidad de disco duro, hay que distinguir una estructura física y una estructura lógica.

La **estructura física** divide el disco según sus elementos físicos:

- Caras
- Pistas o cilindros
- Sectores

La **estructura lógica** divide el disco según los elementos que gestionan el almacenamiento y organización de los datos:

- Sector de arranque
- FAT (*File Allocation Table* o Tabla de Localización de Ficheros)
- Directorio raíz
- Área de datos

La estructura física es inherente al disco. Por el contrario, la estructura lógica la crea el usuario al formatear el disco (FORMAT en DOS). El S.O. almacena los datos según la estructura lógica, por eso hay que formatear un disco antes de usarlo.

4.6.1.- Principio de Almacenamiento Inverso (*big-endian*).

Para poder comprender la forma en la que el DOS trabaja con el disco es necesario conocer el **principio de almacenamiento inverso**, más conocido como orden *big-endian* (el byte más a la derecha es el más significativo). Según este principio, cuando el DOS trata algún campo por palabras (y no como bytes independientes) el almacenamiento y la lectura del contenido de ese campo es inverso. Lo mejor para poder comprender este concepto es ilustrarlo con un ejemplo; supongamos que existe un campo formado por 3 bytes en el que se almacena el valor de la extensión de un fichero, si la extensión es TXT, en el campo se almacenará el valor 54 58 54. Supongamos ahora que existe un segundo campo de una palabra, entonces el almacenamiento del valor 4587 (11EBH) quedaría EB 11 (en vez de 11 EB que sería el almacenamiento directo). Si nos encontramos con un campo de dos palabras, por ejemplo, el valor 3167485950

(BCCBFFFEH) el tratamiento de este valor se hará de forma inversa, esto es, FE FF CB BC. Nótese que cada dos dígitos hexadecimales componen un byte.

Queremos almacenar				***Se almacena***			
Palabra 1		Palabra 2		Palabra 2		Palabra 1	
byte 1	byte 2	byte1	byte2	byte 2	byte 1	byte 2	byte 1
BC	**CB**	**FF**	**FE**	**FE**	**FF**	**CB**	**BC**

4.6.2.- Estructura física de un disco.

La estructura física de un disco es común para discos duros y disquetes, y está compuesta de caras, pistas o cilindros y sectores.

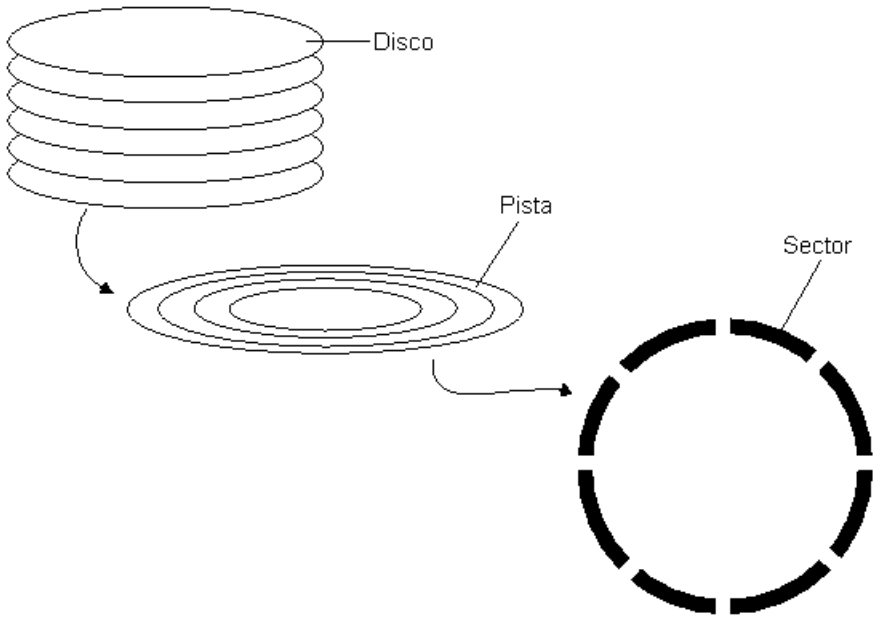


Ilustración 1. ***Estructura física de un disco.***

Cada disco puede tener una o dos **caras** (*heads*), según almacene información en una o las dos superficies del disco. Cada cara se divide en círculos concéntricos llamados **pistas** (*tracks*) en los disquetes y **cilindros** (*cylinders*) en los discos duros. Cada pista se divide en segmentos llamados **sectores**. El sector es la unidad mínima de información para los discos. En general, todos los sectores de los disquetes y discos duros tienen un tamaño de **512 bytes**.

En la Tabla 1 se muestran los parámetros usuales de la estructura física de los disquetes estándares.

	360KB (5 ¼)	**720KB (3 ½)**	**1.2MB (5 1/4)**	**1.44MB (3 1/2)**
Nº Caras	2	2	2	2
Nº Pistas/Cara	40	80	80	80
Nº Sectores/Pista	9	9	15	18
Tamaño Sector	512	512	512	512
Densidad	Doble	Doble	Alta	Alta

Tabla 1. ***Parámetros físicos de los disquetes estándares.***

Generalmente, la *densidad* de un disquete hace referencia al número de pistas por cara del disquete, siendo de doble densidad los que tiene 40 y de alta densidad los que tiene 80. Sin embargo, los fabricantes venden los disquetes de 360KB y 720KB como de doble densidad, y los de 1.2MB y 1.44MB como de alta densidad, atendiendo al criterio de número de sectores por pista como característica diferenciadora.

4.6.3.- Estructura lógica de un disco.

La estructura lógica de un disco no tiene por qué coincidir con la estructura física, pero está condicionada por ella. La estructura lógica se crea cuando se formatea un disco.

Cuando se formatea un disco, ya sea disquete o disco duro, se realizan dos cosas: se define el número de pistas y el número sectores por pista, y se divide el disco en cuatro zonas diferentes.

- Número de pistas y número de sectores por pista. El número de pistas de un disco está determinado por su estructura física, sin embargo, es posible crear una estructura lógica con menos pistas por cara o menos sectores por pista (nunca con más).

- División en cuatro zonas. Al formatear el disco, éste queda dividido en cuatro zonas:

Sectores Reservados (Sector de Arranque)
FAT (y sus posibles copias)
Directorio raíz
Área de Datos

Cada una de estas áreas tiene una misión diferente y un tamaño variable. El objetivo de esta división es que los datos puedan ser leídos y grabados. Los próximos apartados se dedican a explicar más detalladamente estas cuatro zonas, haciendo especial hincapié en su localización en el disco.

4.6.3.1.- El Sector de Arranque.

Generalmente, el sector de arranque (*boot sector*) se localiza siempre en el sector físico 0 de cualquier disco. Cada vez que se inicializa el ordenador, se busca el sector de arranque en la unidad A o, si no hay disquete en la unidad A y hay disco duro, en el sector de arranque de la unidad C. El sector de arranque contiene un pequeño programa (*bootstrap-loader*) que carga el DOS en memoria durante la inicialización del sistema. El programa en primer lugar comprueba si existen los ficheros del sistema IO.SYS y MSDOS.SYS. Si no existen se muestra en pantalla un mensaje de error. Si existen los carga en memoria y les cede el control. Además de la rutina de carga, el sector de arranque contiene también otra información útil relativa a la estructura física y lógica del disco. Los valores del sector de arranque dependen del formateo del disco.

La Tabla 2 muestra el formato general del sector de arranque para un disco duro o un disquete.

Desplazamiento	Longitud	Descripción
00H	3 bytes	Instrucción de salto al programa de arranque
03H	8 bytes	Identificación del sistema (fabricante,utilidad, ...)
0BH	1 palabra	Número de bytes por sector
0DH	1 byte	Número de sectores por *cluster*
0EH	1 palabra	Número de sectores reservados (con *boot sector*)
10H	1 byte	Número de copias de la FAT
11H	1 palabra	Número máximo de entradas del directorio raíz
13H	1 palabra	Número total de sectores
15H	1 byte	Identificador del disco (F0 para 3 ½ y 1.44MB)
16H	1 palabra	Número de sectores para cada FAT
18H	1 palabra	Número de sectores por pista
1AH	1 palabra	Número de cabezas (caras)
1CH	1 palabra	Número de sectores ocultos (0)
20H	4 bytes	Número total de sectores si 13H es 0
24H	1 byte	Número de unidad física (sólo discos duros)
25H	1 byte	Reservado
26H	1 byte	Byte de marca con el valor 29H
27H	4 bytes	Número de serie del disco
2BH	11 bytes	Etiqueta del disco (NO NAME si no tiene)
36H	5 bytes	Tipo de FAT: FAT12 o FAT16
. . .	. . .	. . . . .
. . .	. . .	Rutina de Arranque (*Bootstrap-Loader*)

Tabla 2. ***Estructura del sector de arranque (Bloque de parámetros del BIOS).***

Sector de Comienzo y Longitud del Sector de Arranque.

El sector de arranque se sitúa siempre en el sector físico 0 de todo disco. Los sectores reservados se utilizan para almacenar el código de la rutina de arranque cuando no se puede acoplar completa en el sector de arranque. Generalmente, el único sector reservado es el sector de arranque. El número de sectores reservados, incluyendo el sector de arranque, se especifica en el offset 0EH del sector de arranque (normalmente suele ser 1).

Definición
Sector_A = Primer sector del sector de arranque
Longitud_A = Longitud del sector de arranque
Valor
Sector_A = Sector 0
Longitud_A = Número de sectores reservados (offset 0EH)

4.6.3.2.- Directorios y entradas de directorio.

El directorio raíz comienza en el primer sector libre tras la última copia de la FAT. El directorio raíz (y cualquier otro directorio) está formado por elementos de **32** bytes que se llaman **entradas de directorio**. Habrá una entrada de directorio por cada

fichero almacenado, por cada subdirectorio, y sólo en el raíz una entrada para la etiqueta o volumen del disco.

Desplazamiento	Descripción	Tamaño
00H	Nombre del fichero	8 bytes
08H	Extensión del fichero	3 bytes
0BH	Atributo	1 byte
0CH	Reservado	10 bytes
16H	Hora	1 palabra
18H	Fecha	1 palabra
1AH	Número del *cluster* inicial	1 palabra
1CH	Tamaño del fichero	2 palabras

Tabla 3. ***Organización de una entrada de directorio.***

Nombre del fichero.

Los primeros 8 bytes de una entrada de directorio contienen el nombre del fichero, en formato ASCII. Si el nombre tiene menos de 8 caracteres completamos con blancos hacia la derecha. Las letras deben ser mayúsculas. No intercalar blancos en el nombre, ya que la mayor parte de los comandos del DOS no lo admitirían como parámetro (podemos crear ficheros de forma que no puedan borrarse fácilmente).

Cuando se borra un fichero, el DOS pone el primer byte del nombre, en la entrada de directorio, a E5H para indicar que esa entrada puede utilizarse para otro nombre de fichero. Así mismo, las entradas de la FAT correspondientes al fichero se pondrán a 0 para indicar que los *clusters* que ocupaba se encuentran ahora disponibles.

Extensión del fichero.

A continuación de los 8 bytes para el nombre aparecen 3 más para la extensión que también será almacenada en formato ASCII. Al igual que con el nombre se rellena con caracteres blancos (código ASCII 32) si ocupa menos de los 3 bytes. Cuando el directorio contiene una entrada de etiqueta de volumen, el nombre y la extensión son tratados como un campo combinado de 11 bytes.

Atributos del fichero.

Corresponden al byte siguiente a los 11 bytes anteriores. En este byte cada bit tiene un significado:

7	6	5	4	3	2	1	0	Significado
.	.	.	.	.	.	.	1	Sólo lectura
.	.	.	.	.	.	1	.	Oculto
.	.	.	.	.	1	.	.	Sistema
.	.	.	.	1	.	.	.	Etiqueta de volumen
.	.	.	1	.	.	.	.	Subdirectorio
.	.	1	.	.	.	.	.	Fichero
.	0	.	.	.	.	.	.	No se utiliza
0	.	.	.	.	.	.	.	No se utiliza

Tabla 4. ***Significado del byte de atributo.***

Nota: U**na entrada de etiqueta de volumen sólo es reconocida en el directorio raíz.**

Hora del fichero.

Este campo consta de una palabra para especificar la hora en la que el fichero fue creado o modificado por última vez. El valor para este campo se obtiene como:

Hora = (Horas*2048) + (Minutos*32) + (Segundos/2)

Horas	Minutos	Segundos
5 bits	6 bits	5 bits

Fecha del fichero.

Este campo, al igual que el anterior, consta de una palabra, y establece la fecha en la que se creó o se modificó el archivo por última vez. El valor para este campo se obtiene como:

Fecha = ((Año-1980)*512)+(Mes*32)+Día

Año	Mes	Día
7 bits	4 bits	5 bits

Número del *cluster* de comienzo.

Es un campo de una palabra de longitud en el que se especifica el número de *cluster* de comienzo del fichero en el área de datos. Para los ficheros que no tienen asignado espacio y para las entradas de etiqueta de volumen, el número de *cluster* de comienzo es 0.

Tamaño del fichero.

Este campo de 2 palabras de longitud contiene el tamaño en bytes del fichero (ficheros de hasta 4GB). Gracias a este campo es posible conocer el tamaño real de un archivo, ya que la asignación de espacio que hace el DOS para los archivos es en unidades de *cluster* (es decir, de 512 en 512 bytes) por lo que es probable que el último *cluster* no esté ocupado totalmente.

Subdirectorios.

El directorio raíz tiene un tamaño fijo y está almacenado en una posición fija del disco. Un subdirectorio no tiene tamaño fijo, y puede almacenarse en cualquier lugar del área de ficheros como ocurre con cualquier otro fichero. El formato para las entradas de directorio en un subdirectorio es idéntico al de las entradas de directorio para el directorio raíz.

Un directorio padre tiene una entrada por cada uno de sus subdirectorios. La entrada para un subdirectorio es igual que la entrada para un fichero, excepto que en el byte del atributo marca la entrada como de un subdirectorio y en que el campo de tamaño del fichero se pone a 0.

Cuando el DOS crea un subdirectorio, pone dos entradas especiales en él, con . y .. como nombres. El . se refiere al subdirectorio presente (el primer byte es 2EH) y .. se refiere al directorio padre (los dos primeros bytes son 2E2EH). El *cluster* de comienzo de . es el *cluster* de comienzo del propio directorio, y el de .. es el *cluster* de comienzo del directorio padre (que si es el raíz será el número 0).

Sector de Comienzo y Longitud del Directorio Raíz.

El directorio raíz se localiza tras la FAT y presenta una diferencia importante con el resto de directorios: su tamaño es limitado. El directorio raíz tiene un número máximo de entradas de directorio especificado en el sector de arranque (offset 11H).

Definición			
	Sector_A	=	Primer sector del sector de arranque
	Longitud_A	=	Longitud del sector de arranque
	Sector_F	=	Primer sector de la FAT
	Longitud_F	=	Longitud de la FAT
	Sector_D	=	Primer sector del directorio raíz
	Longitud_D	=	Longitud del directorio raíz
Valor			
	Sector_A	=	Sector 0
	Longitud_A	=	Número de sectores reservados (offset 0EH)
	Sector_F	=	Sector_A + Longitud_A
	Longitud_F	=	Nº de FAT's (offset 10H) * Sectores por FAT (offset 16H)
	Sector_D	=	Sector_F + Longitud_F
	Longitud_D	=	(Nº de Entradas (offset 11H) * 32)/Bytes por sector (0BH)

4.6.3.4.- Espacio de Datos de Usuario.

Todos los ficheros de datos y los subdirectorios se almacenan en el área de ficheros. Cuando se crea un fichero, o cuando se extiende un fichero ya existente, el espacio asignado al fichero crece. Conforme el fichero va creciendo, el DOS le va asignando nuevos *clusters*. Puesto que esta asignación de espacio es dinámica los *clusters* que conforman un fichero no tienen por qué ser contiguos, la FAT se va a encargar de mantener enlazados todos los *clusters* para cada fichero.

Sector de Comienzo y Longitud del Espacio de Datos de Usuario.

Definición			
	Sector_A	=	Primer sector del sector de arranque
	Longitud_A	=	Longitud del sector de arranque
	Sector_F	=	Primer sector de la FAT
	Longitud_F	=	Longitud de la FAT
	Sector_D	=	Primer sector del directorio raíz
	Longitud_D	=	Longitud del directorio raíz
	Sector_U	=	Primer sector del Espacio de Datos de Usuario
	Longitud_U	=	Longitud del Espacio de Datos de Usuario

Valor		
Sector_A	=	Sector 0
Longitud_A	=	Número de sectores reservados (offset 0EH)
Sector_F	=	Sector_A + Longitud_A
Longitud_F	=	Nº de FAT's (10H) * Sectores por FAT (offset 16H)
Sector_D	=	Sector_F + Longitud_F
Longitud_D	=	(Nº de Entradas(offset 11H) * 32) /Bytes por sector (0BH)
Sector_U	=	Sector_D + Longitud_D
Longitud_U	=	TotSectores(13H)–Longitud_A–Longitud_D–Longitud_F

4.6.3.5.- La FAT (Tabla de Localización de Ficheros).

Guardar la información en los discos en sectores contiguos, es decir, secuencialmente, no es la forma más eficiente de utilizar un disco debido a la fragmentación que se produce cuando almacenamos ficheros y posteriormente los borramos. Los huecos que quedan no serían aprovechados si guardásemos la información de forma secuencial.

Para evitar este problema, se utilizan los **clusters** o **unidades de asignación**. Un *cluster* es un conjunto de uno o más sectores contiguos. El sector es la unidad mínima de información para el disco, pero el *cluster* es la unidad mínima de información para el DOS. Cuando se copia un fichero, no se hace sector a sector sino *cluster* a *cluster*. Por tanto, aunque un fichero ocupe un solo byte, el fichero consume un *cluster* del disco. El número de sectores que posee un *cluster* ha de ser potencia de dos y su valor es uno de los parámetros que encontramos en el sector de arranque (offset 0DH).

Los *clusters* están formados por sectores contiguos, y sin embargo, los ficheros se almacenan en *clusters* que no necesitan ser contiguos. ¿Cómo se sabe el *cluster* inicial que ocupa un fichero y la secuencia subsiguiente? Mediante la **FAT: Tabla de Asignación** o **Localización de Ficheros**. La FAT es una tabla formada por elementos de 12 o 16 bits que corresponden a cada uno de los *clusters* del disco. Si un disco tiene más de 4096 (2^{12}) *clusters*, entonces los elementos de la FAT son de 16 bits, en caso contrario, son de 12 bits (1,5 bytes). En el offset 36H aparece la cadena FAT12 o FAT16 según el número de bits de cada elemento de la FAT. Cada *cluster* del disco tiene asociado un elemento de la tabla que coincide con su posición, o sea, el *cluster* 5 del disco tiene asociado el elemento de la FAT de la posición 5. Este elemento nos dice cual es el siguiente cluster que forma parte del fichero. Cuando se alcanza el último *cluster* del fichero, la entrada correspondiente de la FAT poseerá una marca especial para indicarlo. Los valores que puede contener la FAT son:

Valor de 12 bits	Valor de 16 bits	Significado
000H	0000H	*Cluster* no utilizado
FF7H	FFF7H	*Cluster* dañado
FF8-FFFH	FFF8-FFFFH	Último *cluster* en un fichero
XXXH	XXXXH	Siguiente *cluster* de un fichero.

Tabla 5. ***Valores posibles para las entradas de la FAT:*** **(para una FAT con entradas de 12 bits y para una FAT con entradas de 16 bits).**

El espacio que pertenece a un fichero viene dado por una cadena de entradas de FAT, cada una de las cuales apunta a la siguiente entrada de la cadena. El número del primer *cluster* (comienzo de la cadena para un fichero) aparece en la entrada de directorio del fichero. Cuando se necesita espacio para un fichero, el DOS busca en la FAT *clusters* que no estén en uso (el valor para la entrada de FAT de ese cluster ha de ser 0) y los añade a la cadena de clusters del fichero. Cuando se libera espacio ocupado de un fichero, el DOS marca como libres las entradas de la FAT de los *clusters* liberados. Ejemplo:

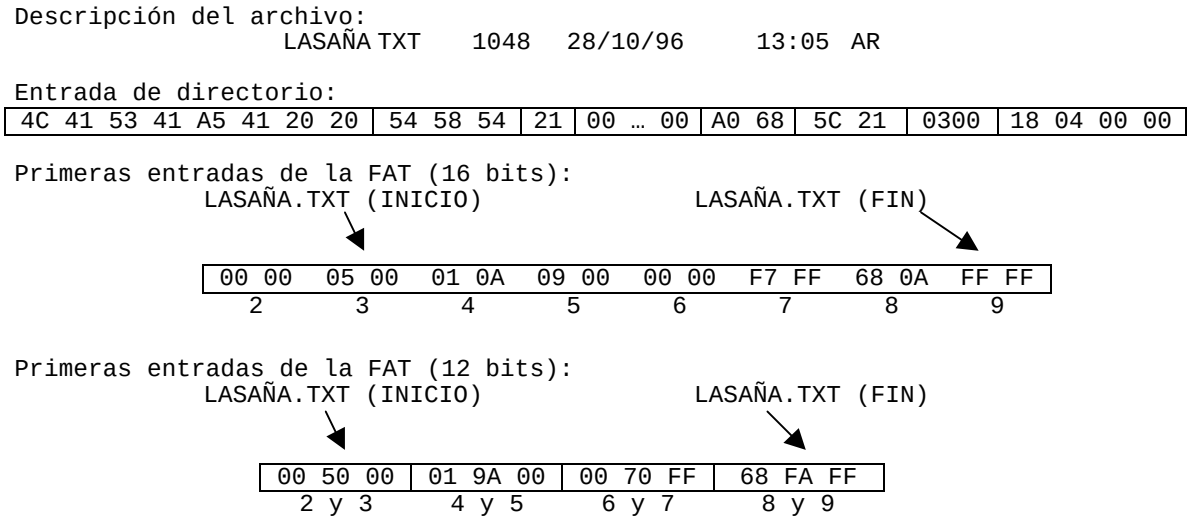


Ilustración 2. ***Entrada de directorio y FAT para LASAÑA.TXT***

Las dos primeras entradas de la FAT están reservadas para su utilización por el DOS, y dado que cada entrada se corresponde con un *cluster*, el *cluster* número 2 será el primer *cluster* de espacio disponible en el área de ficheros.

Con lo visto hasta ahora, se puede comprender que, para el DOS, un fichero no es más que un conjunto de *clusters* que se gestionan mediante la FAT. A continuación se explica cómo se realizan las operaciones más comunes en relación a la FAT.

NAVEGAR POR LA FAT

Leer el Sector de Arranque.

En el sector de arranque se encuentran todos los datos que permiten averiguar las características físicas y lógicas del disco.

Averiguar el Número de Bits de la FAT.

Para averiguar el número de bits de la FAT se puede consultar el offset 36H del sector de arranque. Allí aparecerá una cadena con cinco caracteres con el valor “FAT12” o “FAT16”. La principal diferencia entre la FAT de 12 bits y la FAT de 16 bits es que las FAT de 12 bits tiene un rango de 0 a 4095 *clusters*, mientras que las FAT de 16 bits tienen un rango de 0 a 65535 *clusters*.

Obtener el primer cluster de un fichero.

El primer *cluster* de un fichero aparece en la entrada de directorio del fichero, en los bytes 1AH y 1BH. En primer lugar, hay que averiguar el sector donde se encuentra la entrada de directorio del fichero, lo cual implica leer los directorios de su trayectoria absoluta. Ejemplo: Fichero C:\DOS\SYS.COM

- Calcular el número de sector donde comienza el directorio raíz.
- Leer los sectores del directorio raíz buscando la entrada del DOS.
- Obtener de la entrada del directorio su *cluster* de inicio.
- Convertir el *cluster* de inicio a número de sector lógico.
- Leer el sector donde se almacenan las entradas del directorio y buscar una con nombre SYS.COM.
- Buscar el *cluster* de inicio del fichero en la entrada correspondiente.

Desplazarse por la FAT.

Una vez hallado el *cluster* de inicio del fichero, se sigue la cadena de *clusters* del mismo. La localización de la entrada de FAT correspondiente a un *cluster* es sencilla cuando las entradas son de 16 bits, ya que únicamente hemos de multiplicar el número de *cluster* por 2. Pero para una FAT de 12 bits la cosa es más complicada. Aquí cada par de entradas de FAT ocupan 3 bytes (12 + 12 bits, la entrada 0 y la 1 ocupan los primeros tres bytes, y así sucesivamente). Para calcular la posición en la FAT para la entrada correspondiente a un *cluster* dado habremos de multiplicar el número de *cluster* por 1,5 y quedarnos con el resultado en la forma siguiente:

- **Si el número de *cluster* es par**, el resultado será un **número entero positivo** que nos dará la posición de la entrada en la FAT. Si leemos en palabras de 16 bits cogeremos 4 dígitos hexadecimales, y de estos 4 sólo 3 son válidos, para ello aplicaremos el principio de almacenamiento inverso y desecharemos el dígito más a la izquierda.
- **Si el número de *cluster* es impar**, al multiplicar por 1,5 nos quedará un **número con cifras decimales**, debemos truncar el número, leer la palabra (16 bits) desde esa posición, y de los cuatro dígitos hexadecimales, aplicando también el principio de almacenamiento inverso, desechamos el dígito más a la derecha.

Nota: Los dos primeros elementos de la FAT (0 y 1) están reservados. El primer byte contiene el identificador del disco (F0 para un disquete de 3 ½ con 1.44MB y F8 para un disco duro) y el resto el valor FFH.

Sectores de un cluster.

Algunas veces necesitaremos saber los sectores que ocupa un determinado *cluster*. Para hallar el primer sector ocupado por un *cluster*:

- Se resta 2 al número del *cluster* (recordemos que los *clusters* 0 y 1 están reservados).
- Multiplicamos el resultado por el número de sectores por *cluster*.
- Sumamos el número de sector de comienzo del área de datos.

Sector de Comienzo y Longitud de la FAT.

La FAT se encuentra en sectores contiguos a continuación del sector de arranque. El número de sectores ocupados por la FAT se define en los parámetros del sector de arranque (offset 16H). Normalmente suele haber dos copias de la FAT por motivos de seguridad.

Definición		
Sector_A	=	Primer sector del sector de arranque
Longitud_A	=	Longitud del sector de arranque
Sector_F	=	Primer sector de la FAT
Longitud_F	=	Longitud de la FAT
Valor		
Sector_A	=	Sector 0
Longitud_A	=	Número de sectores reservados (offset 0EH)
Sector_F	=	Sector_A + Longitud_A
Longitud_F	=	Nº de FAT's (offset 10H) * Sectores por FAT (offset 16H)

OPERACIONES COMUNES

Copiar un fichero.

Cuando se va a copiar un fichero, es necesario reservar espacio en la FAT para el nuevo fichero. Se busca el primer *cluster* libre no defectuoso de la tabla (000H si la FAT es de 12 bits y 0000H si es de 16 bits) y se emplea. A continuación se busca el siguiente *cluster* libre no defectuoso de la tabla y se utiliza también. Así sucesivamente hasta que se copia todo el fichero. De este modo el fichero puede estar almacenado en *clusters* dispersos.

Manipular un fichero.

Si se quiere realizar alguna tarea con un fichero, se busca en el directorio dónde esté el fichero y se obtiene su *cluster* de inicio examinando la entrada de directorio correspondiente. Con el valor obtenido se indexa la FAT y se consigue el valor del siguiente *cluster* del fichero. Repetimos la operación hasta que se alcance la marca de fin del fichero.

Borrar un fichero.

Cada vez que se borra un fichero, se escribe el valor (0)000H en las entradas de la FAT de los *clusters* que ocupaba el fichero, para que se puedan utilizar a la hora de copiar nuevos ficheros en el disco. Además, se marca la entrada de directorio correspondiente al fichero poniendo E5H en el primer carácter del nombre.

Dado que cada fichero ocupa como mínimo un *cluster*, el sistema FAT desaprovecha espacio en el disco. Este espacio desaprovechado es la diferencia entre el número de bytes ocupados por los *clusters* asignados a un fichero, y el número de bytes reales del fichero (*slack*).

4.6.4.- Particiones de un Disco Duro.

Los discos duros pueden dividirse en *particiones*. Las particiones responden a una necesidad importante: instalar diferentes sistemas operativos en el mismo disco duro (cada sistema operativo utiliza un esquema de organización de los datos en disco diferente). Por ejemplo, la organización de los datos que hace el DOS mediante la FAT no es válida para UNIX.

Una partición es un conjunto de cilindros contiguos cuyo tamaño es definido por el usuario. Las particiones están separadas lógicamente entre sí y los datos de una no se mezclan con los de las demás particiones. Por ello, cada partición puede soportar un sistema operativo diferente; también es posible usar todas ellas para DOS en cuyo caso cada partición corresponde a una unidad lógica. Un disco duro puede tener varias particiones pero, sólo una partición *activa*. La partición activa es la que toma el control del ordenador cuando se arranca. El primer sector absoluto de un disco duro tiene un pequeño programa de inicialización y una tabla de particiones. Cuando arrancamos el ordenador el programa lee la tabla de particiones y cede el control a la partición activa.

4.6.4.1.- Tabla de Particiones.

La tabla de particiones se compone de cuatro entradas de 16 bytes que almacenan la información relevante sobre cada una de las cuatro particiones permitidas. En la Tabla 6 se presenta la estructura general de la tabla de particiones.

Cada entrada de 16 bytes almacena el tamaño, la localización y el tipo de partición: cabeza/cilindro/sector de inicio, tipo de la partición, cabeza/cilindro/sector de finalización, número de sector absoluto dónde comienza la partición y número total de sectores de la partición. Su formato aparece en la Tabla 7.

Desplazamiento	Tamaño	Descripción
000H	445 bytes	(Reservados)
1BEH	16 bytes	Entrada partición 1
1CEH	16 bytes	Entrada partición 2
1DEH	16 bytes	Entrada partición 3
1EEH	16 bytes	Entrada partición 4
1FEH	2 bytes	Marca igual a AA55H

Tabla 6. ***Estructura de la tabla de particiones de un disco duro.***

Desplazamiento	Tamaño	Descripción
0H	1 bytes	Indica si la partición es activa (80H)
1H	1 bytes	Cabeza del primer sector
2H	2 bytes	Cilindro/sector del primer sector
4H	1 bytes	Tipo de partición
5H	1 bytes	Cabeza del último sector
6H	2 bytes	Cilindro/sector del último sector
8H	4 bytes	Número de sector relativo al inicio
CH	4 bytes	Longitud de la partición en sectores

Tabla 7. Estructura de cada entrada de una partición.

4.6.5.- Particiones del DOS.

Para usar un disco duro primero hay que particionarlo (FDISK) y a continuación dar formato a cada una de las particiones (FORMAT). Bajo DOS, un disco duro puede tener dos particiones, una partición *primaria* y una partición *extendida*.

La **partición primaria** es imprescindible y designa la partición en la cual se almacenan los ficheros de arranque del sistema. La partición primaria es la partición activa.

La **partición extendida** puede dividirse a su vez en varias unidades lógicas. Una partición extendida sólo puede crearse cuando ya existe una partición primaria.

4.6.6- Parámetros absolutos de un Disco Duro.

Para acceder a los parámetros del disco duro, independientemente del número de las particiones, hay que obtener el contenido de la palabra doble de memoria situada en la dirección 0000:0104 que contiene la dirección donde se guarda una tabla de 16 bytes con los parámetros absolutos de la primera unidad de disco duro.

Desplazamiento	Tamaño	Descripción
00H	2 bytes	Número de cilindros
02H	1 bytes	Número de cabezas
03H	11 bytes	(Reservados)
0EH	1 byte	Sectores por pista
0FH	1 byte	(Reservado)

Tabla 8. ***Estructura de la tabla de parámetros absolutos de un disco duro.***

Si se trata de una partición extendida, los datos de la entrada corresponden al total de las unidades lógicas que almacena. La partición extendida posee su propia tabla de particiones, pero sólo se usan las dos primeras entradas. La primera entrada corresponde a la primera unidad lógica de la partición extendida, y la segunda indica si hay otra unidad lógica. En caso afirmativo, ésta contiene de nuevo una tabla de particiones. Con ello se crea una especie de lista enlazada que permite acceder a la información de las diferentes particiones.

4.6.7.- Leer y Escribir sectores en un disco.

Para leer uno o más sectores se utiliza la función Int 13H, subfunción 02H del BIOS. Son necesarios los siguientes parámetros:

- AH: 02H
- AL: número de sectores, hasta el máximo por pista.
- CH: número de pista (la primera pista es la cero).
- CL: bits 7-6 número de pista (bits superiores).
- CL: bits 5-0 número de sector inicial (el primer sector es el uno).
- DH: número de cabeza.
- DL: número de unidad (0 = A, 1 = B, 2 = C, 3 = D, etc.).
- ES:BX: dirección del área de datos en que se depositan los sectores leídos.

A continuación ilustraremos la lectura de un sector mediante un ejemplo:

```
SLEIDO    DB   512 DUP(?)     ; Área de Almacenamiento
. . .
          MOV  AH,02H    ; Petición de lectura
          MOV  AL,01H    ; 1 Sector
          LEA  BX,SLEIDO ; Área en ES:BX
          MOV  CH,05     ; Pista 5
          MOV  CL,03     ; Sector 3
          MOV  DH,00     ; Cabeza 0
          MOV  DL,03     ; Unidad 3(D)
          INT  13H       ; Llamada al BIOS
```

Si la operación se efectúa con normalidad, la bandera de acarreo se pone a cero y AL contiene el número de sectores que la operación ha leído realmente. En caso contrario, se pone a uno la bandera de acarreo y se devuelve en AH el código de error.

La operación contraria, es decir, la escritura, corresponde a la función Int 13H, subfunción 03H del BIOS. Los parámetros que deben incluir los registros son los mismos que en el caso anterior, siendo en este caso la zona de memoria definida, la que contiene la información que es escrita en el disco. Los valores que se devuelven son similares a los de la lectura.

4.7.- LA MEMORIA DE VIDEO.

4.7.1.- 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. El **monitor** es una de las partes más importantes del ordenador ya que es el primer medio por el que el usuario conoce los resultados de su trabajo. 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**.

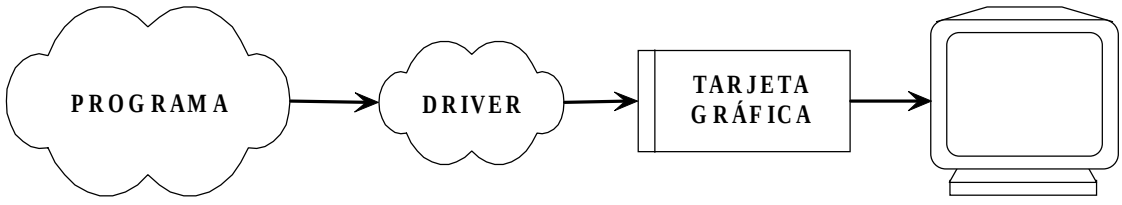
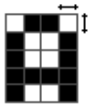


Figura 16. ***Sistema de visualización en un ordenador.***

La tarjeta gráfica es un dispositivo hardware que sirve de intermediario entre el ordenador y el monitor. Esta tarjeta posee:

- Memoria RAM para almacenar la información a representar.
- Memoria ROM o generador de caracteres.
- Puertos programables.

El sistema puede funcionar en **modo texto** y en **modo gráfico**. En cualquier caso, la información se representa mediante puntos luminosos llamados *pixels*. Varias veces por segundo, para evitar que la imagen desaparezca, se produce un barrido de la pantalla por un haz de electrones para iluminar los *pixels* correspondientes a la información que se mostrará.



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*.

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).

A continuación se muestran varios ejemplos de ubicaciones del cursor:

	Formato decimal		Formato hexadecimal	
Ubicación en pantalla	Fila	Columna	Fila	Columna
Esquina superior izquierda	00	00	00H	00H
Esquina superior derecha	00	79	00H	4FH
Centro de la pantalla	12	39/40	0CH	27H/28H
Esquina inferior izquierda	24	00	18H	00H
Esquina inferior derecha	24	79	18H	4FH

En modo gráfico la pantalla se divide en puntos o *pixels* individuales cuyo número depende de la tarjeta gráfica instalada en el ordenador y de la resolución máxima admitida por el monitor.

La diferencia entre los modos gráfico y texto estriba en cómo la tarjeta interpreta la memoria que comparte con el procesador. En cada refresco del monitor (lo cual ocurre desde 50 hasta 100 veces por segundo dependiendo del hardware disponible) se lee e interpreta la memoria de vídeo. El haz de electrones es guiado para ofrecer una imagen fiable de la memoria de vídeo. Sin embargo, en modo texto, la tarjeta interpreta cada byte en la memoria como un carácter a imprimir en la pantalla (obteniendo la matriz de puntos de la memoria ROM de la tarjeta) junto con su atributo, que indica el color del

carácter y del fondo. En modo gráfico, por el contrario, cada byte de la memoria de vídeo es interpretado como el color de un *pixel* individual o un conjunto de *pixels*[6].

4.7.2.- Tipos de Tarjetas Gráficas.

En la siguiente tabla se resume los principales tipos de tarjetas gráficas con sus características principales. En cualquier caso los valores que aparecen son meramente orientativos; las características de cada tarjeta en particular dependen del fabricante porque el mercado de este tipo de dispositivos es bastante heterogéneo.

TARJETAS GRÁFICAS							
CARACTERÍSTICAS	TIPO DE TARJETA						
	MDA	CGA	HGC	EGA	VGA	SVGA	XGA
Año Aparición	1981	1981	1982	1984	1987		1991
Res. Modo Gráfico	NO	320X200	720X348	640X350	640X480	1024X768	1024X768
Matriz por carácter	8X8	8X8	14X9	14X8	16X9	16X9	16X9
Nº Colores Máxima Res.	2	4	2	26	16	256	256
Paleta Colores	2	16	2	64	262.144	262.144	262.144
Memoria RAM	4K	16K	64K	256K	>256K	>512K	>512K
Dirección Memoria	B0000-10000	B8000-BC000		A0000-BFFFF	A0000-BFFFF	A0000-BFFFF	A0000-BFFFF
Memoria ROM	NO	NO		C0000-B3FFF	C0000-C5FFF	C0000-C7FFF	
Tipo monitor	TTL	RGB	TTL	RGB TTL	Analógico	Analógico	Analógico

Tabla 9. ***Tipos de Tarjetas Gráficas.***

4.7.3.- La RAM de Vídeo.

Cada dirección de la memoria de vídeo corresponde a una determinada posición de la pantalla. Esta memoria de vídeo forma parte del espacio de direcciones de la memoria central. La dirección de arranque de esa memoria y su tamaño dependen del tipo de tarjeta y monitor instalados. La memoria de vídeo es una memoria RAM situada en la tarjeta gráfica y es totalmente independiente de la propia RAM del ordenador. Esta memoria de vídeo es leída cada cierto tiempo (dependiendo de la frecuencia de refresco), enviándose su contenido a la pantalla del monitor. Cuanto mayor sea la memoria de vídeo, más información se puede reproducir en el monitor, o lo que es lo mismo, se pueden conseguir mayores resoluciones o mayor número de colores, pero en ningún caso se aumenta la velocidad de construcción de la imagen.

Por ejemplo, una tarjeta SVGA visualizando con 1024x768 puntos de resolución tiene un total de 786.432 puntos. Si vamos a emplear 256 colores, necesitamos al menos 8 bits por *pixel*, y por tanto, se necesitan 8·786.432 = 6.291.456 bits de memoria de vídeo, es decir, 786432 bytes. Con estos datos, sabemos que nuestra tarjeta SVGA deberá tener al menos 1Mb de memoria de vídeo, ya que 512K no serían suficientes.

[6] Hay que tener en cuenta que un byte da la posibilidad de distinguir 256 colores distintos. En modos de menos colores, como en las tarjetas EGA, un byte puede almacenar colores para más de un *pixel* (por ejemplo, 2 pixels de 16 colores), sin contar las restricciones históricas de la memoria de vídeo.

La siguiente tabla muestra la memoria RAM de vídeo necesaria para algunas tarjetas y modos gráficos:

TARJETAS GRÁFICAS					
TARJETA	COLORES				
	16 4 bits	256 8 bits	32.768 15 bits	65.536 16 bits	16.777.216 24 bits
VGA (640X480)	256K	512K	1M	1M	1M
SVGA (800x600)	256K	512K	1M	1M	1,5M
SVGA (1024x768)	512K	1M	2M	2M	2,5M
SVGA (1280x1024)	1M	1,5M	3M	3M	4M
SVGA (1600x1200)	1M	2M	4M	4M	6M

Tabla 10. ***Memoria de Vídeo para cada resolución y modo gráfico.***

4.7.4.- El modo texto.

Como se dijo, en el modo texto la tarjeta interpreta la memoria de vídeo como pares <*carácter, atributo*>. En la Tabla 9 se indica dónde comienza la memoria de vídeo para cada una de las tarjetas listadas. Así, en el offset 0 del comienzo de esta memoria está el carácter que se muestra en la esquina superior izquierda. En el offset 1 está su atributo, en el 2 está el carácter situado justo a su derecha, etc. Como ejemplo, valga la siguiente secuencia extraída de *Turbo Debugger*:

Esta secuencia es la que aparece en la línea superior de este programa. Si analizamos la memoria de vídeo, obtenemos algo así:

```
B800:0000 20 70 20 70 F0 74 20 70 20 70 46 74 69 70 6C 70  p p-t p pFtiplp
B800:0010 65 70 20 70 20 70 45 74 64 70 69 70 74 70 20 70 ep p pEtdpiptp p
B800:0020 20 70 56 74 69 70 65 70 77 70 20 70 20 70 52 74  pVtipepwp p pRt
B800:0030 75 70 6E 70 20 70 20 70 42 74 72 70 65 70 61 70 upnp p pBtrpepap
B800:0040 6B 70 70 70 6F 70 69 70 6E 70 74 70 73 70 20 70 kpppopipnptpsp p
B800:0050 20 70 44 74 61 70 74 70 61 70 20 70 20 70 4F 74  pDtaptpap p pOt
B800:0060 70 70 74 70 69 70 6F 70 6E 70 73 70 20 70 20 70 pptpipopnpsp p p
B800:0070 57 74 69 70 6E 70 64 70 6F 70 77 70 20 70 20 70 Wtipnpdpopwp p p
B800:0080 48 74 65 70 6C 70 70 70 20 70 20 70 20 70 20 70 Hteplppp p p p p
B800:0090 20 70 20 70 20 70 52 2F 45 2F 41 2F 44 2F 59 2F  p p pR/E/A/D/Y/
```

Obsérvese primero que la dirección de memoria es la B8000H (dirección absoluta), por lo que estamos en un adaptador a color. El primer byte de la memoria de vídeo es un 20H (32 decimal), es decir, un espacio. Como vemos, un espacio es lo que guarda la posición más a la izquierda de la imagen anterior. El siguiente byte guarda el valor 70H. Según lo dicho, este valor corresponde al atributo del carácter, que indica su color y el color del fondo. La codificación de estos valores la estudiaremos después. Así pues, los dos primeros bytes de la memoria de vídeo contienen el carácter y el atributo de la posición (1,1) de la pantalla[7]. En el offset 2, 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. Debido al principio de

[7] Suponemos una numeración (fila, columna). En la pantalla, los rangos serán (1,1) para la esquina superior izquierda y (25,80) para la esquina inferior derecha.

almacenamiento inverso, si leemos una palabra completa de la memoria de vídeo, esto nos dejará al atributo en la parte alta de la palabra y al carácter en la parte baja. Por ejemplo, el código:

```
MOV AX,0B800H
MOV ES,AX
MOV AX,WORD PTR ES:[0000]
```

dejará en AL (la parte baja del registro AX) el carácter en la posición (1,1) de la pantalla. En AH (la parte alta de AX) dejará el atributo para ese carácter.

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 que antes vimos (70H, en binario 01110000b) equivale a un carácter negro (000b) sobre un fondo gris claro (111b).

Para saber dónde hay que mirar, es decir, la dirección absoluta B0000H ó B8000H (si tenemos un adaptador monocromo o uno color, según la Tabla 9), basta con mirar en la palabra de memoria almacenada en 40H:63H. Si el valor es 3B4H, el adaptador es mono, y debemos mirar en la dirección B0000H. Por el contrario, si guarda 3D4H, el adaptador es color y debemos mirar en B8000H. El siguiente código ejemplifica la comprobación:

```
MOV AX,40H                ; XOR AX,AX
MOV ES,AX                 ; MOV ES,AX
MOV AX,WORD PTR ES:[63H] ; MOV AX,WORD PTR ES:[463H]
CMP AX,03D4H
JE COLOR
```

```
MONO:
     MOV AX,0B000H
     JMP CONTINUE
COLOR:
     MOV AX,0B800H
CONTINUE:
     MOV ES,AX
```

(el código a la derecha es equivalente, pero un byte más corto). Sin embargo, este test ha quedado ya casi obsoleto, debido a la no utilización de tarjetas monocromas.

Como un ejemplo final, aquí hay una rutina que acepta una posición en la pantalla y un carácter (con su atributo) y lo escribe en la pantalla:

```
;IMPRIMIR EN LA POSICIÓN <X,Y> UN CARÁCTER
;AX = POSICIÓN Y, BX = POSICIÓN X, CARÁCTER=CL, CH=ATRIBUTO
PRINTAT PROC
     PUSH DX
     PUSH ES

     DEC  AX             ; (F-1)
     DEC  BX             ; (C-1)

     MOV  DX,80          ; (F-1)*80
     MUL  DX
     ADD  BX,AX          ; (F-1)*80+(C-1)
     SHL  BX,1           ; ^^^^^^^^^^^^^^*2
     MOV  AX,0B800H      ; SUPONEMOS TARJETA COLOR
     MOV  ES,AX
     MOV  ES:[BX],CX

     POP  ES
     POP  DX
     RET
PRINTAT ENDP
```

4.7.5.- El modo gráfico.

En el modo gráfico, cada byte de la memoria de vídeo da el color a un conjunto de *pixels* de la pantalla. Antes de la aparición de las tarjetas SuperVGA, la correspondencia entre direcciones en la memoria de datos y puntos de la pantalla era algo truculenta. Esto era debido a que se utilizaban modos gráficos en los que se mostraban menos de 256 colores (número máximo de colores que un byte puede identificar). Con estos modos, un byte de la memoria de vídeo podía (debido a las restricciones de memoria existentes) guardar información para varios *pixels* de la pantalla. La mejora con las tarjetas SVGA (e incluso VGA en alguno de sus modos) es que se utilizan 256 colores *como mínimo*. Esto hace que ahora un byte de la memoria de vídeo del color para un solo *pixel*, aunque lo normal es que *varios pixels* den el color de un solo punto, como en los modos de 32K colores, 64K colores, 16M colores, etc. Estos últimos modos de memoria de vídeo se conocen como **memoria lineal**, ya que la correspondencia de *offsets* de memoria crecientes es hacia *pixels* crecientes hacia la derecha y hacia abajo.

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.

Este modo es el más cómodo que ha existido nunca. Se adapta bien a la estructura segmentada del 80x86, ya que toda la memoria de vídeo cabe en 64K (320*200 = 64000, lo que es menor que 65536 o 64K). Además 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 *pixel* de la esquina superior izquierda (0,0). La dirección siguiente corresponde al siguiente *pixel* a la derecha. Así hasta la posición A013Fh, que corresponde a la posición de la pantalla (0,319)[8], es decir, la esquina superior derecha. La fórmula pues de acceso a un *pixel* queda parecida a la que utilizábamos en modo texto, salvo que ahora consideramos coordenadas comenzadas en 0, por lo que la fórmula queda:

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

Da la casualidad que esta multiplicación se puede realizar de forma muy rápida. 320 en binario es 101000000, por lo que sólo se necesitan dos sumas y dos desplazamientos para realizar la multiplicación. Si suponemos que la coordenada *y* se guarda en AX y la coordenada *x* se guarda en BX, la posición de memoria viene calculada por el siguiente código ensamblador:

```
MOV CX,6
SHL AX,CL
ADD BX,AX
SHL AX,1
SHL AX,1
ADD BX,AX
```

y ahora BX guarda el *offset* de la memoria de vídeo. Si suponemos que ES guarda el valor necesario (A000h), podemos escribir MOV BYTE PTR ES:[BX],<*color*>. El código anterior es más rápido que una multiplicación y una suma. Una rutina, pues que escriba un *pixel* en la pantalla en cualquier posición podría ser la siguiente:

```
;PINTAR EN LA POSICIÓN <X,Y> DE UN COLOR
;AX = POSICIÓN Y, BX = POSICIÓN X, CH=COLOR
PIXAT PROC
     PUSH ES

     MOV  CL,6           ; 320 = 64 + 256
     SHL  AX,CL          ; (Y*64)
     ADD  BX,AX          ; (Y*64) + X
     SHL  AX,1           ; (Y*128)
     SHL  AX,1           ; (Y*256)
     ADD  BX,AX          ; (Y*64) + X + (Y*256) = Y*320 + X

     MOV  AX,0A000H      ; SEGMENTO DE MEMORIA DE VÍDEO
     MOV  ES,AX
```

[8] Suponemos numeración (fila,columna).

```
     MOV  ES:[BX],CH

     POP  ES
     RET
PIXAT ENDP
```

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.

Finalmente, la manipulación directa de la memoria de vídeo es posible. Por ejemplo, el siguiente código borra la pantalla (estableciendo el color 0 a todos los *pixels*):

```
MOV  AX,0A000H
MOV  ES,AX
MOV  CX,320*200 ; ESTA OPERACIÓN LA REALIZA EL ENSAMBLADOR
XOR  DI,DI
CLD
REP  STOSB
```

o podemos escribir cada línea de su propio color:

```
     CLD
     XOR  DI,DI
     MOV  AX,0A000H
     MOV  ES,AX
     MOV  AX,0FF00H
BUCLE:
     MOV  CX,320
     REP  STOSB
     INC  AX
     JNZ  BUCLE
```

Nótese que asignando a AX el valor ff00h nos ahorramos una comparación en cada bucle, ya que el propio INC actualiza los *flags*.

4.8.- ASIGNACIÓN Y LIBERACIÓN DE MEMORIA.

Los servicios proporcionados por el DOS permiten *asignar, liberar y modificar el tamaño* de un área de memoria.

Asignación de memoria.

Para asignar un bloque de memoria, se emplea la función 48H, colocando en BX el número requerido de párrafos.

```
MOV   AH,48H                ; Petición de asignación de memoria
MOV   BX,num_parrafos       ; Número de párrafos que se reservan
INT   21H
```

Si el DOS puede llevar a cabo la operación con éxito, se pone a cero el flag de acarreo y en AX se devuelve la dirección del segmento con el bloque de memoria asignado. La operación busca en la memoria un bloque lo suficientemente grande como para satisfacer la petición.

En caso de que la operación no se realice satisfactoriamente, se pone a uno la bandera de acarreo y se devuelve en AX un código de error y en BX el tamaño, en párrafos, del bloque más grande disponible.

Liberación de memoria.

Para liberar un bloque de memoria, se emplea la función 49H, colocando en ES la dirección del segmento del bloque que será liberado.

```
MOV   AH,49H                  ; Petición de liberación de memoria
LEA   ES,dir_segmento         ; Dirección del bloque por párrafos
INT   21H
```

Si el DOS puede llevar a cabo la operación con éxito, se pone a cero el flag de acarreo y se almacena 00H en el segundo y tercer bytes del bloque, lo que significa que queda libre. En caso contrario, se pone un uno en la bandera de acarreo y se devuelve en AX un código de error.

Modificación de un bloque de memoria asignada.

Mediante la función 4AH se puede aumentar o disminuir el tamaño de un bloque de memoria. Se debe inicializar BX con el número de párrafos conservados y ES con la dirección del PSP del programa.

```
MOV   AH,4AH                  ; Petición para modificar la memoria
MOV   BX,num_parrafos         ; Número de párrafos
LEA   ES,dir_PSP              ; Dirección del PSP
INT   21H
```

Si la operación se lleva a cabo con éxito se pone a cero la bandera de acarreo. En caso contrario, se pone a uno, se devuelve en AX un código de error y en BX el tamaño máximo posible.

4.9.- DIFERENCIAS ENTRE PROGRAMAS .COM Y .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 para

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.

Ejemplo de un programa .COM y su equivalente .EXE.

```
            Page 60,132
TITLE PCOM  Programa .COM para mover y sumar
CODESG      SEGMENT     PARA 'Code'
            ASSUME      CS:CODESG,DS:CODESG,SS:CODESG,ES:CODESG
            ORG         100h        ; Inicio al final del PSP
BEGIN:      JMP MAIN                ; Salto pasando los datos
; -------------------------------------------------------
FLDA        DW          250
FLDB        DW          125
FLDC        DW          ?
; -------------------------------------------------------
MAIN        PROC        NEAR
            MOV         AX,FLDA
            ADD         AX,FLDB
            MOV         FLDC,AX

            MOV         AX,4C00H    ; Salida al DOS
            INT         21H
MAIN        ENDP
CODESG      ENDS
            END         MAIN
```

```
            Page 60,132
TITLE PEXE  Programa .EXE para mover y sumar
; --------------------------------------------------------
STACKSG     SEGMENT     PARA STACK 'Stack'
            DW          32 DUP(0)
STACKSG     ENDS
; --------------------------------------------------------
DATASG      SEGMENT     PARA 'Data'
FLDA        DW          250
FLDB        DW          125
FLDC        DW          ?
DATASG      ENDS
; --------------------------------------------------------
CODESG      SEGMENT     PARA 'Code'
BEGIN       PROC        FAR
            ASSUME      SS:STACKSG,DS:DATASG,CS:CODESG
            MOV         AX,DATASG   ; Se asigna dirección DATASG
            MOV         DS,AX       ; a DS
            MOV         AX,FLDA
            ADD         AX,FLDB
            MOV         FLDC,AX

            MOV         AX,4C00H    ; Salida al DOS
            INT         21H
BEGIN       ENDP
CODESG      ENDS
            END         BEGIN
```

Carga y ejecución de un programa .COM.

Al cargar un programa .COM, el DOS:

- Establece los cuatro registros de segmento con la dirección del primer byte del PSP.
- Establece el apuntador de la pila (SP) al final del segmento de 64K, desplazamiento FFFEH (o al final de la memoria si el segmento no es lo bastante grande) y guarda en la pila una palabra con ceros.
- Establece el apuntador de instrucciones en 100H.

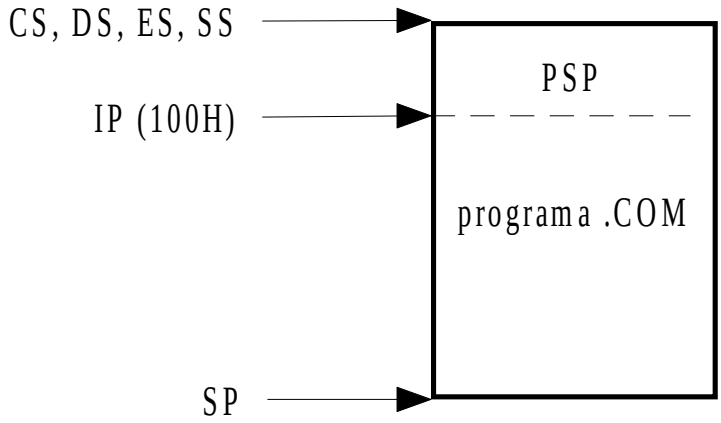


Figura 17. ***Estructura de un programa .COM.***

Carga y ejecución de un programa .EXE.

Un programa .EXE almacenado en disco consta de dos partes: un *registro de encabezado* que contiene información de *reubicación* y control, y el *módulo ejecutable real*.

El encabezado es de longitud variable, y puede llegar a ser grande si hay muchos elementos reubicables. El encabezado contiene información acerca del tamaño del módulo ejecutable, dónde será cargado en memoria, la dirección de la pila y los desplazamientos de reubicación que serán insertados para direcciones incompletas de máquina.

Al cargar un programa .EXE, el DOS:

- Lee el encabezado y lo envía a la memoria.
- Calcula el tamaño del módulo ejecutable (tamaño total del archivo menos el tamaño del encabezado) y lo carga en memoria en el segmento inicial.
- Lee los elementos a reubicar y suma al valor de cada elemento el valor del segmento inicial.
- Establece los registros DS y ES con la dirección del segmento del PSP.
- Establece el registro SS con la dirección del PSP, más 100H, más el desplazamiento SS. Además, coloca en el registro SP el tamaño de la pila.
- Establece CS con la dirección del PSP, más 100H, más el desplazamiento para el CS. Además, coloca en IP el desplazamiento correspondiente (directiva `END <punto_de_inicio>`).

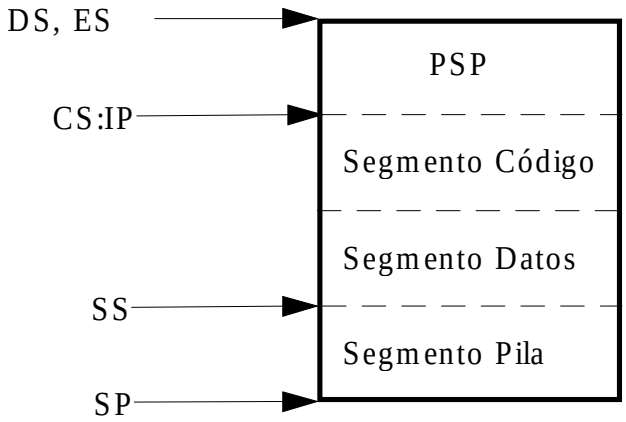


Figura 18. ***Estructura de un programa .EXE.***

4.10.- INTERRUPCIONES EN EL PC.

4.10.1.- ¿ Qué es una interrupción ?.

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 situación

se da, por ejemplo, cuando un periférico requiere la atención del procesador para realizar una operación de E/S.

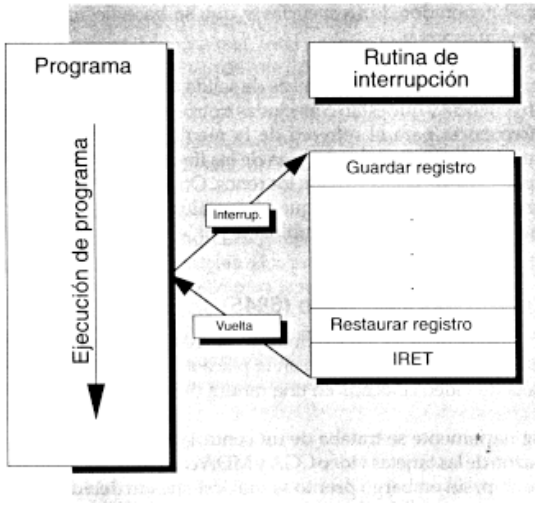


Figura 19. ***Esquema de una interrupción.***

4.10.2.- 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.

4.10.3.- 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 bytes = 1K.

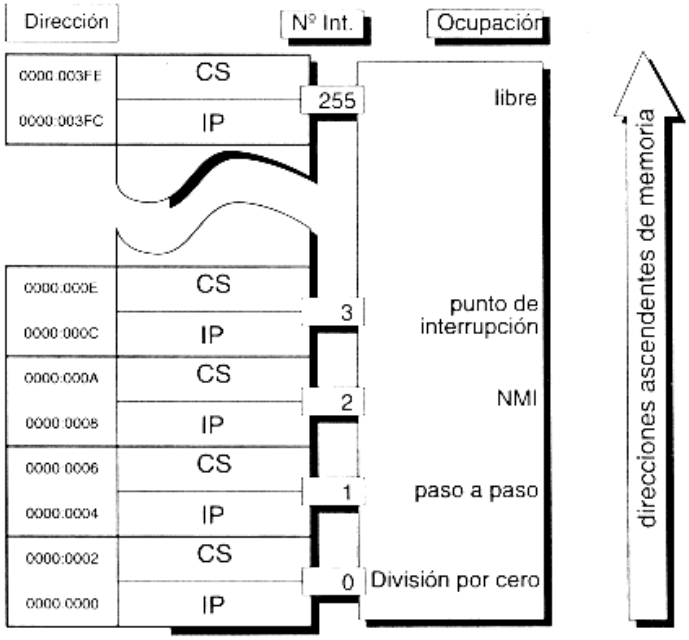


Figura 20. ***Estructura de la tabla de vectores de interrupción.***

4.10.4.- 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). Una petición de interrupción sobre este pin es enmascarable mediante el bit IF del registro de estado, es decir, si se pone a cero IF no se reconocerán las peticiones de interrupción, mientras que si está a uno sí se admiten tales peticiones. En ese momento la CPU termina la ejecución de la instrucción en curso y realiza las siguientes operaciones:

- Finaliza la ejecución de la instrucción en curso.
- Almacena en la pila el registro de estado.
- Almacena en la pila la dirección de retorno: registros CS e IP.
- Inhibe las interrupciones.
- Activa el pin INTA (a nivel bajo). El dispositivo al comprobar la activación de INTA sabe que su petición ha sido reconocida.
- El dispositivo periférico pone en el bus de datos el número de vector de interrupción, y éste es leído por la CPU.
- Multiplica el número de vector leído por cuatro para obtener la dirección de la tabla donde se encuentra el vector de interrupción.
- Coloca en CS:IP la dirección de comienzo de la rutina que tratará la interrupción.
- Se ejecuta la rutina de servicio de interrupción que finaliza en IRET, restituyéndose el registro de estado, y los registros CS e IP.

Nota: Esta restitución supone la autorización automática de las interrupciones.

Existe 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. Estos procedimientos llamados mediante INT reciben el nombre de **interrupciones software** (similares a los servicios ofrecidos por MIPS ejecutados con `syscall`). 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***
Descripción	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.
Banderas	IF (0) y TF (0).
Formato	INT Nº_Vector
Ejemplo	INT 21H

Operación	**IRET: Regresar de un procedimiento DOS o BIOS**
Descripción	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.
Banderas	No las afecta.
Formato	IRET [VALOR POP]
Ejemplo	IRET

Nota: Es posible ejecutar dentro de un programa una rutina de servicio de interrupción correspondiente a una interrupción hardware conociendo su número de vector de interrupción.

Las **interrupciones internas o excepciones** son generadas por el propio procesador cuando se produce una situación anormal:

- INT 0: error de división, generada automáticamente cuando el cociente no cabe en el registro o el divisor es cero. Sólo se puede producir cuando se ejecutan DIV o IDIV.
- INT 1: paso a paso, se produce tras cada instrucción cuando el procesador está en modo traza (utilizada en la depuración de programas).

La INT 2 o interrupción no enmascarable (**NMI**) es una **interrupción externa** que tiene prioridad absoluta y se produce incluso aunque estén inhibidas las interrupciones (con CLI) para indicar un hecho muy urgente (fallo de alimentación o error de paridad en la memoria).

4.10.5.- Circuito Controlador de Interrupciones: i8259. IRQ's.

En el PC la activación del pin INTR no la realizan directamente los dispositivos periféricos sino un circuito integrado denominado i8259. Este circuito se conecta con la CPU mediante el bus de datos y los pines INTR e INTA. Además, ofrece ocho pines o líneas IRQ0 a IRQ7 para que ocho dispositivos (15 en el AT) puedan realizar peticiones de interrupción. Cada línea tiene asociada un número de vector de interrupción y una prioridad (como regla general el número de vector de interrupción correspondiente a cada línea IRQn es el n + 8). El 8259 actúa de árbitro y una vez seleccionada la línea aceptada, gestiona la demanda de interrupción con la CPU.

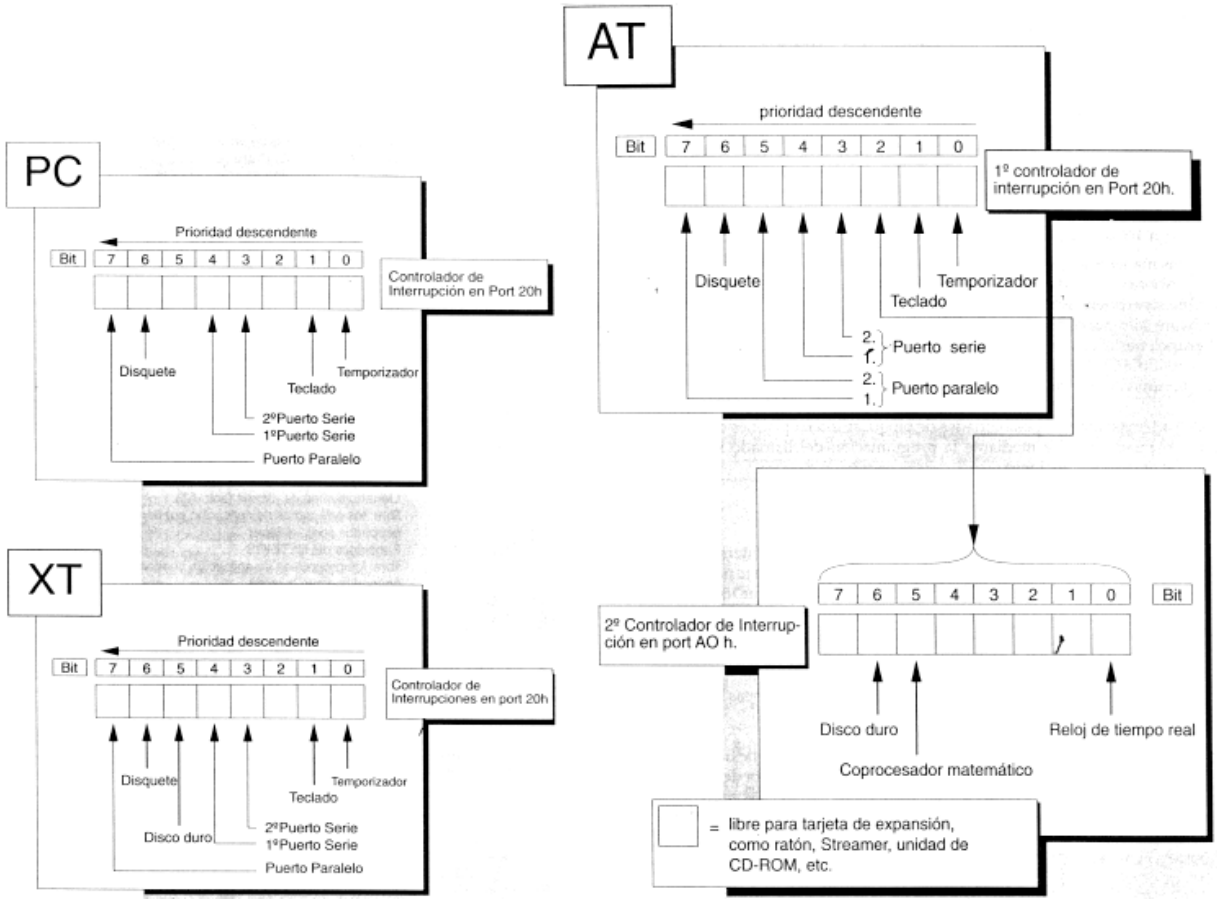


Figura 21. ***Requerimiento de interrupción y prioridades en un XT y en un AT.***

4.10.6.- 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. De esta forma, cuando se produce la interrupción, se ejecuta nuestra rutina y no la que había originalmente. Por ello, lo normal es que nuestra rutina incluya al final un salto al código original de tratamiento de la interrupción. Gráficamente:

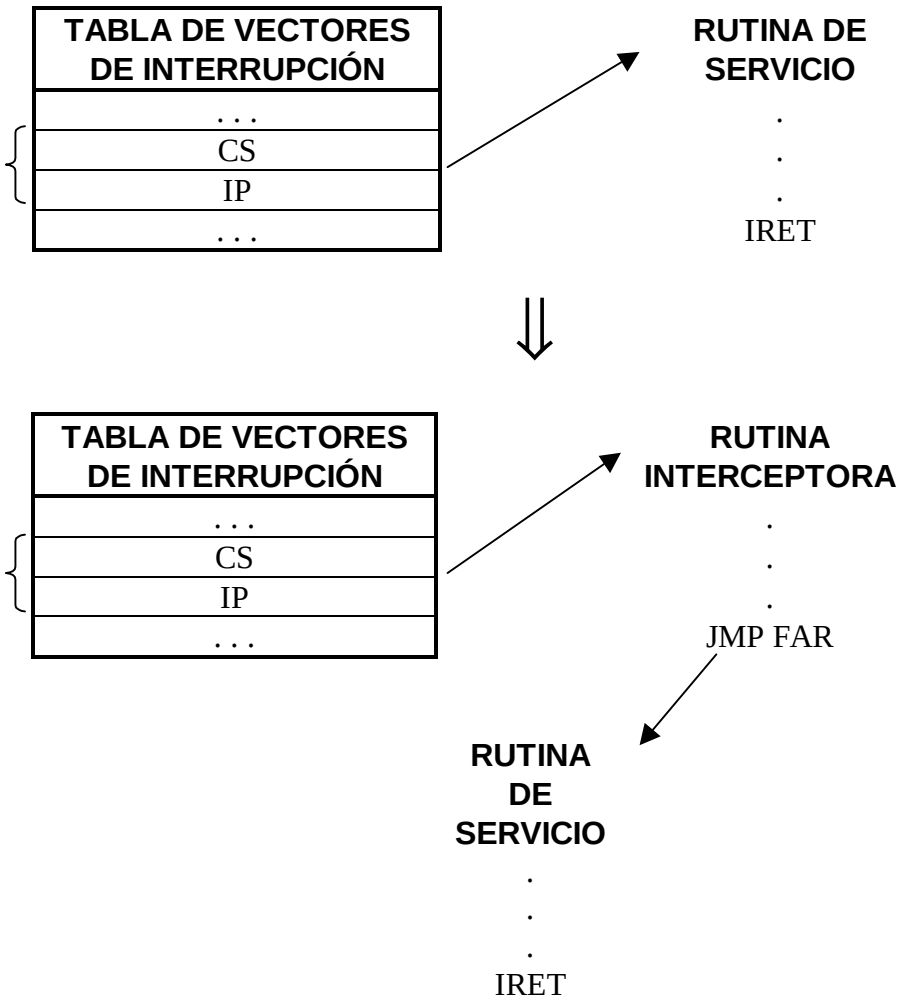


Figura 22. ***Interceptar una interrupción.***

4.10.7.- Circuito Temporizador: i8253.

El circuito temporizador o *timer* es como el corazón del reloj del sistema, y se utiliza además de generador de señales para el refresco de memoria y de pulsos para el reloj, para emitir un sonido a una cierta frecuencia a través del altavoz. El altavoz está gobernado por el *chip* PPI (i8255 *Programmable Peripheral Interface*). A través del puerto B del PPI (puerto 97 o 61H), al altavoz le llegan dos señales procedentes de los bits 0 y 1 de dicho puerto (ver Figura 23):

- BIT 0 del puerto B del PPI determina si el temporizador actúa o no sobre el altavoz según sea 1 ó 0.
- BIT 1 del puerto B del PPI especifica si el altavoz permanece activo o inactivo según valga 1 ó 0.

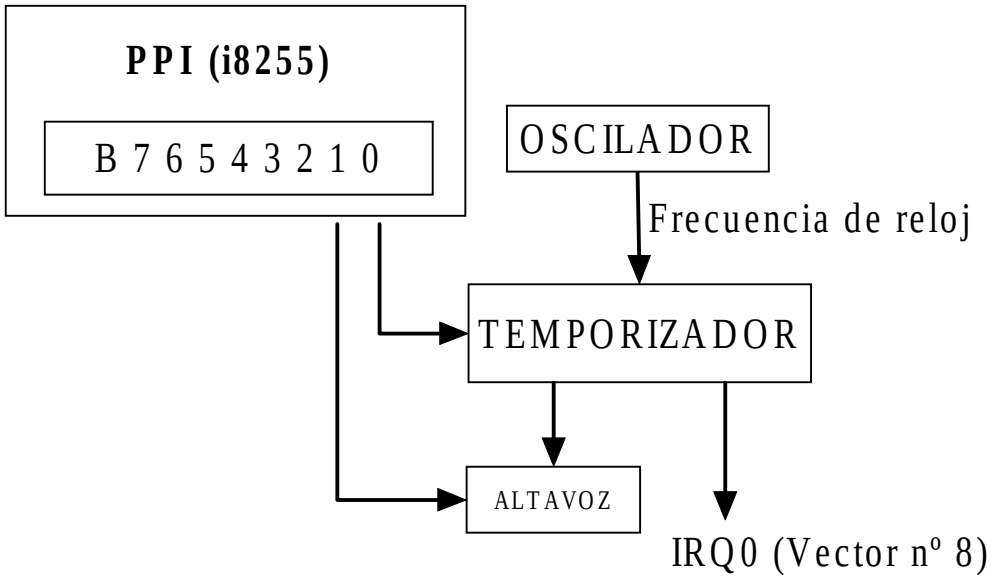


Figura 23. ***El temporizador o timer (i8253).***

Veamos como controlar el altavoz usando el timer. El reloj del sistema (i8284A) oscila a una frecuencia de 1.193.180 ciclos por segundo (aproximadamente 1.193 MHzs) que se toma como referencia. El temporizador genera una interrupción (IRQ0, vector de interrupción 08H) cada 65.536 ciclos de reloj, es decir 18,2 veces por segundo (*tic* de reloj que se usa para calcular la hora). Para producir un sonido a una determinada frecuencia hay que seguir los siguientes pasos:

- Obtener el período del temporizador:

PERÍODO = 1.193.180/FRECUENCIA_DESEADA

- Preparar el temporizador para recibir el período:

ENVIAR 182 (B6H) AL PUERTO 67 (43H)

- Cargar el período en el temporizador:

ENVIAR EL PERÍODO AL PUERTO 66 (42H)

- Activar el temporizador y el altavoz:

PONER A 1 LOS BITS 0 Y 1 DEL PUERTO 97 (61H)

- Controlar la duración:

INSTRUCCIÓN LOOP

- Desactivar el temporizador y el altavoz:

PONER A 0 LOS BITS 0 Y 1 DEL PUERTO 97 (61H)

4.11.- 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).

4.11.1.- Cómo hacer que el programa quede residente.

Para hacer que el programa quede residente, en lugar de una terminación normal, se utiliza la función 31H de la Int 21H (mantener el programa):

```
MOV  AH,31H              ; Petición para TSR
MOV  DX,tamaño_programa  ; Tamaño del programa en párrafos
INT  21H
```

El tamaño del programa se debe dar en párrafos. Un párrafo son 16 bytes, por lo que el tamaño del programa habrá que dividirlo entre 16.

Cuando se ejecuta la rutina de inicialización, el DOS reserva el bloque de memoria en donde el programa reside y carga los programas subsecuentes en una parte superior de la memoria.

Nota: la función 27H consigue un efecto similar a la 31H, pero tiene la limitación de 64K.

4.11.2.- Activación del programa residente.

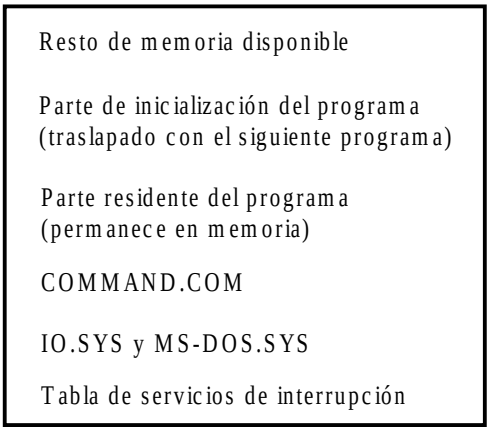
La idea es que los programas residentes intercepten cualquier pulsación de teclas, realizando las acciones oportunas cuando la secuencia de teclas pulsada coincida con su patrón de activación, y pasando por alto las demás pulsaciones. Un programa residente por lo común, consta de las siguientes partes:

- Procedimiento de inicialización que ejecuta sólo la primera vez el programa y que realiza las siguientes tareas:

    1. Reemplaza la dirección en las tablas de servicios de interrupción con su propia dirección.
    2. Establece el tamaño de la parte del programa que permanece residente.
    3. Utiliza una instrucción Int que indica al DOS que termine la ejecución del programa actual, pero que deje residente parte de él.

- Un procedimiento que permanezca residente y que es activado, por ejemplo, por una entrada desde el teclado, o por el reloj.

En realidad, el procedimiento de inicialización realiza todas las tareas necesarias para que el programa permanezca residente y después se borra él mismo. La organización de la memoria sería como sigue:



4.11.3.- Obtener dirección de interrupción.

```
MOV   AH,35H                ; Petición de interrupción
MOV   AL,int#               ; Número de interrupción
INT   21H
```

La operación devuelve la dirección de la interrupción en ES:BX como (*Segmento:Offset*).

4.11.4.- Establecer dirección de interrupción.

```
MOV   AH,25H            ;Petición de dirección de interrupción
MOV   AL,int#           ;Número de interrupción
LEA   DX, newaddr       ;Nueva dirección para la interrupción
INT   21H
```

La operación reemplaza la dirección actual de la interrupción con la nueva dirección. Entonces, en realidad, cuando la interrupción especificada ocurre, el proceso enlaza a su programa (residente) en lugar de a la dirección normal de interrupción.

4.11.5.- Ejemplo: BOCINA.ASM.

El programa residente BOCINA hace sonar el altavoz cada vez que se utiliza el panel numérico y la tecla NUMLOCK (BLOQNUM) está activada. Su objetivo es advertir al usuario de que está tecleando números en lugar de moverse con las teclas de dirección, Inicio, Fin, AvPág o RePág. Este programa intercepta la interrupción del teclado, INT 09H, para examinar la tecla presionada y actuar en consecuencia.

```
TITLE   BOCINA1 TSR: verifica NUMLOCK en el área de datos del BIOS.
;-----------------------------------------------------------------
CODESG      SEGMENT     PARA 'Code'
            ASSUME CS:CODESG
            ORG   100H
BEGIN:      JMP   INICIO      ; Salto a la inicializacion.
DIRINT9     DD    ?           ; Direccion de la (INT 09H).
TESTNUM:
      PUSH  AX                ; Guardar registros.
      PUSH  CX
      PUSH  DS
      MOV   AH,02H
      INT   16H               ; Obtiene byte de estado del teclado (AL)
      TEST  AL,00100000B      ; ¿NumLock?
      JZ    EXIT
      IN    AL,60H            ; Obtiene tecla pulsada desde el puerto.
      CMP   AL,71             ; ¿Codigo de rastreo < 71?
      JL    EXIT
      CMP   AL,83             ; ¿Codigo de rastreo > 83?
      JG    EXIT
      MOV   AL,0B6H           ; Preparar el temporizador para recibir
      OUT   43H,AL            ; el periodo.
      MOV   AX,5000
      OUT   42H,AX            ; Enviar el periodo al temporizador.
      IN    AL,61H
      MOV   AH,AL
      OR    AL,00000011B      ; Poner a 1 los bits 0,1 del puerto 61H.
      OUT   61H,AL            ; Activacion temporizador y altavoz.
      MOV   CX,10000          ; Fijar duracion.
PAUSE:
      LOOP  PAUSE
      MOV   AL,AH
      OUT   61H,AL            ; Desactivar altavoz.
EXIT:
      POP   DS                ; Restaurar registros.
      POP   CX
      POP   AX
      JMP   DIRINT9           ; Ejecutar antigua INT 09H.
INICIO:
      CLI                     ; Inhibir interrupciones.
      MOV   AH,35H            ; Obtener direccion de antigua INT 09H.
      MOV   AL,09H
      INT   21H
      MOV   WORD PTR DIRINT9,BX ; Guardar direccion de antigua INT 09H.
      MOV   WORD PTR DIRINT9+2,ES
      MOV   AH,25H            ;Establecer nueva direccion para INT 09H.
      MOV   AL,09
      MOV   DX,OFFSET TESTNUM
      INT   21H
      MOV   AH,31H            ; Peticion para permanecer residente.
      MOV   DX,OFFSET INICIO / 16 + 1
      STI                     ; Habilitar interrupciones.
      INT   21H
CODESG      ENDS
      END   BEGIN
```

Los puntos siguientes del programa residente son de interés:

- CODESG inicia el segmento de código de BOCINA.ASM. La primera instrucción ejecutable, JMP INICIO, transfiere la ejecución, pasando la parte residente, al procedimiento de inicialización. Después usa la función 35H del DOS para conseguir la dirección de la rutina de servicio de la INT 09H. La operación devuelve tal dirección en ES:BX, y se almacena en DIRINT9. A continuación, con la función 25H, modificamos la tabla de vectores de interrupción para que el vector de interrupción del teclado referencie el punto de entrada al programa residente, es decir, TESTNUM. En el último paso, se establece el tamaño de la parte residente (todo el código hasta INICIO) en DX, y se usa la función 31H del DOS para salir haciendo que el programa quede residente. Observemos como el código de nuestro programa a partir de INICIO, será sustituido por el del próximo programa que sea cargado para su ejecución.
- TESTNUM es el nombre del procedimiento residente que se activará cuando el usuario presione una tecla. El sistema transfiere la ejecución a la dirección indicada por el vector de interrupción del teclado que ha sido cambiado para que contenga la dirección de TESTNUM. Como esta interrupción puede ocurrir cuando el usuario esté ejecutando otro programa, como un editor, es necesario guardar en la pila todos los registros que se usan. El programa accede al byte de estado del teclado, mediante la función 02H de la INT 16H del BIOS, y determina si la tecla NumLock está activada. En caso afirmativo, si la tecla presionada pertenece al teclado numérico, se hace sonar el altavoz, tal y como se explicó en el apartado anterior. Las instrucciones finales restauran los registros y saltan al código de la antigua INT 09H.

El siguiente ejemplo ayudará a aclarar el proceso de funcionamiento. Primero explicamos como se opera sin un TSR que intercepte la interrupción:

1. El usuario pulsa una tecla, y el teclado produce una interrupción INT 09H.
2. Se busca en la tabla de vectores de interrupción la dirección de la rutina de servicio para la INT 09H.
3. Se transfiere el control a la rutina correspondiente.
4. La rutina obtiene el carácter y (si es un carácter estándar) lo envía al buffer del teclado.

Ahora veremos qué sucede cuando está activo el programa residente:

1. El usuario pulsa una tecla, y el teclado produce una interrupción INT 09H.
2. Se busca en la tabla de vectores de interrupción la dirección de la rutina de servicio para la INT 09H, que ahora es la dirección de TESTNUM.
3. Se transfiere el control a la rutina correspondiente.
4. Si NumLock está activada y el carácter es un número del teclado numérico, TESTNUM hace sonar la bocina.
5. TESTNUM salta a la dirección original de la INT 09H guardada previamente.
6. La rutina obtiene el carácter y (si es un carácter estándar) lo envía al buffer del teclado.

4.12.- INTERFAZ DE ENSAMBLADOR CON OTROS LENGUAJES DE PROGRAMACIÓN.

En esta sección estudiaremos la interfaz que ofrece el ensamblador hacia otros lenguajes de programación como Pascal o C. Por *interfaz* nos referimos a los mecanismos que hacen posible tanto que un procedimiento en C o Pascal llame a procedimientos ensamblador, como que uno en ensamblador llame a otros definidos en aquellos lenguajes.

El ejemplo que utilizaremos será un procedimiento `Suma` que aceptará tres argumentos: dos palabras como operandos y un puntero a una palabra como resultado. En definitiva, su signatura desde Pascal sería:

```
procedure Suma(a:integer;b:integer;var result:integer);
```

y desde C:

```
void Suma(int a,int b,int* result);
```

Este procedimiento se implementará en ensamblador y se mostrará cómo se pueden crear procedimientos C y Pascal que llaman a este procedimiento para beneficiarse de su funcionalidad.

Por último, resulta conveniente que el lector esté familiarizado con la sintaxis de segmentos expuesta en la sección 4.2. Para la realización de los ejemplos se ha utilizado Borland Turbo Assembler 4.0, Borland Pascal 7.0 y Borland C++ 5.01.

4.12.1.- Interfaz con Pascal.

Desde Pascal podemos definir cualquier procedimiento o función como `external` (externo). Esto significa que el cuerpo del procedimiento o función están implementadas en otro módulo independiente, y que la resolución de su posición se deberá hacer en la etapa de *linkado*. El siguiente programa Pascal declara el procedimiento `Suma` como externo, y en su cuerpo, ejecuta una llamada al mismo para realizar una prueba:

```
program PruSuma;

{***Procedimiento externo, definido en ensamblador***}
procedure Suma(a:Word;b:Word;var result: Word); external;
{$L SUMA.OBJ}

var a:Word;
    b:Word;
    result:Word;

begin
     a := 2;
     b := 5;
```

```
      Suma(a,b,result);

      Writeln('El resultado es: ',result);
end.
```

Nótese cómo en negrita aparece el nombre del módulo objeto que implementa el procedimiento `Suma`. Así pues: ¿cómo se declarará el procedimiento en ensamblador?

Vimos que la directiva .MODEL aceptaba un especificador de lenguaje. En este caso utilizaremos Pascal. Esto servirá para que el procedimiento sepa cómo tratar sus argumentos. Esto lo veremos a continuación. El programa que implementa la función de suma es bien sencillo. Nótese cómo al final se declara al procedimiento `Suma` como PUBLIC. Esto hace que pueda ser accedido desde Pascal:

```
.model compact,pascal

.code

Suma proc
    ARG a:word,b:word,c:ptr word

    ; Salvar registros iniciales
    push es
    push di

    ; Realizar la suma
    mov ax,a
    add ax,b

    ; Y guardarla a donde apunta “c”
    les di,c
    mov es:word ptr [di],ax

    pop di
    pop es
    ret

Suma endp
    public Suma
    end
```

Nótese cómo ahora no aparece la convención de salvado de registro BP, establecimiento de la pila, etc., vistos en la sección 4.2. ¿Qué ha pasado?

La línea en negrita, incluye una línea de argumentos. Estos argumentos son tratados como corresponde dependiendo del lenguaje especificado en .MODEL. Para hacernos una idea de qué añade el compilador, veamos qué código genera finalmente. Esto lo podemos ver ensamblando el programa con la opción `-la` (producir un listado completo):

```
Turbo Assembler    Version 4.0          13/12/98 21:36:18       Page 1
suma.asm

      1     0000                    .model compact,pascal
      2
      3     0000                    .code
      4
```

```
      5     0000                  Suma proc
      6                             ARG a:word,b:word,c:ptr word
      7
      8                             ; Salvar registros iniciales
1     9     0000  55                PUSH    BP
1    10     0001  8B EC             MOV     BP,SP
1    11     0003  06                push es
     12     0004  57                push di
     13
     14     0005  8B 46 0A          mov ax,[bp+0Ah]
     15     0008  03 46 08          add ax,[bp+08h]
     16
     17     000B  C4 7E 04          les di,[bp+04h]
     18     000E  26: 89 05         mov es:word ptr [di],ax
     19
     20     0011  5F                pop di
     21     0012  07                pop es
1    22     0013  5D                POP     BP
1    23     0014  C2 0008           RET     00008h
     24
     25     0017                  Suma endp
     26                             public     Suma
     27                             end
```

Los componentes en negrita son los que ha introducido el ensamblador por nosotros. Véase cómo ha incluido la secuencia inicial de salvaguarda de BP y su posterior restauración al terminar (incluyendo la limpieza de la pila en RET). También ha sustituido el acceso a los argumentos por su correspondiente desplazamiento sobre BP.

El estudio de los accesos relativos a BP nos dice cómo el Pascal almacena los parámetros en la pila para la llamada. A la entrada del procedimiento, la pila queda así:

SP = BP	ANT. BP
BP + 2	DIR. RET.
BP + 4	c (segmento)
BP + 6	c (*offset*)
BP + 8	b
BP + 10	a

(la pila crece hacia arriba). Observe como “c”, al estar declarado como PTR WORD (*puntero a word*), y al estar en el modelo de memoria COMPACT, ocupa **dos** palabras: es una dirección de datos. Lo primero que se apiló fue “a”, es decir, el parámetro más a la izquierda del procedimiento. Esta es la convención que utiliza Pascal: apila los argumentos en la pila de izquierda a derecha. Es fácil ver que de esta manera no se permite el paso de un número variable de argumentos.

Al ensamblar el anterior programa, ya hemos obtenido el SUMA.OBJ, por lo que ya podemos compilar y ejecutar nuestro programa en Pascal original PRUSUMA.PAS, obteniendo como resultado:

```
El resultado es: 7
```

como esperábamos.

4.12.2.- Interfaz con C.

El interfaz con C es parecido al de Pascal. De hecho, al abstraer el tipo de lenguaje de interfaz con la directiva .MODEL, **todo el código** escrito en ensamblador lo podemos utilizar para C. Así, el listado para C queda de la siguiente manera:

```
Turbo Assembler    Version 4.0          13/12/98 22:03:19       Page 1
suma.asm

      1     0000                    .model compact,c
      2
      3     0000                    .code
      4
      5     0000                    Suma proc
      6                               ARG a:word,b:word,c:ptr word
      7
      8                               ; Salvar registros iniciales
1     9     0000  55                  PUSH    BP
1    10     0001  8B EC               MOV     BP,SP
1    11     0003  06                  push es
     12     0004  57                  push di
     13
     14     0005  8B 46 04            mov ax,[bp + 04h]
     15     0008  03 46 06            add ax,[bp + 06h]
     16
     17     000B  C4 7E 08            les di,[bp + 08h]
     18     000E  26: 89 05           mov es:word ptr [di],ax
     19
     20     0011  5F                  pop di
     21     0012  07                  pop es
1    22     0013  5D                  POP     BP
1    23     0014  C3                  RET     00000h
     24
     25     0015                    Suma endp
     26                              public    Suma
     27                              end
```

Aparte de la declaración .MODEL y del nombre del procedimiento, algo más ha cambiado. Ahora los desplazamientos con respecto a BP son distintos. Además, la última instrucción RET no especifica ningún valor. Esto diferencia a la convención C. La pila ahora queda así:

SP = BP	ANT. BP
BP + 2	DIR. RET.
BP + 4	a
BP + 6	b
BP + 8	c (offset)
BP + 10	c (segmento)

Los argumentos están apilados justo al revés. Es decir, el primero que se apiló fue el de más a la derecha de los parámetros (de derecha a izquierda). Esto tiene dos consecuencias:

- Por un lado, permite que existan un número variable de argumentos, ya que los parámetros de izquierda a derecha permanecen en posiciones fijas con respecto a BP[9];
- Por otro, es necesario que el que realiza la llamada, que es el que conoce cuántos argumentos ha enviado, es el que limpie la pila de argumentos. En la convención Pascal es el procedimiento llamado el que la limpia. En C, es el procedimiento llamador el que la limpia, añadiéndole la cantidad correspondiente a SP.

El programa C correspondiente es el siguiente:

```
#include <stdio.h>

extern void Suma(int a, int b, int* result);

int a,b;
int result;

void
main(void)
{
    a = 5;
    b = 2;

    Suma(a,b,&result);

    printf("El resultado es: %d",result);
}
```

Utilizamos la semántica de punteros con C, y al igua que antes se declara Suma como `extern`.

Una salvedad final. Al ensamblar con Turbo Assembler, hay que incluir la opción `/ml` para que conserve las mayúsculas y minúsculas tal y como son introducidas en el programa. El C es sensitivo a mayúsculas y minúsculas, al contrario que Pascal. Finalmente, la línea que consigue obtener el programa final es:

```
bcc –mc prusuma.c suma.obj
```

donde la opción `-mc` indica el modelo COMPACT de memoria. El resultado obtenido es el mismo que el del correspondiente programa Pascal.

[9] Conviene que el lector lo compruebe.

BIBLIOGRAFÍA

[1] Peter Abel. **Lenguaje Ensamblador y Programación para PC IBM y compatibles**. Editorial Prentice Hall, 3ª edición [1996].

[2] Miguel Ángel Rodríguez-Roselló. **8086/8088 Programación en Ensamblador en entorno MS-DOS**. Editorial Anaya Multimedia [1988].

[3] Michael Tischer. **PC Interno**. Editorial Marcombo [1993].

DIRECCIONES WEB

[1] http://www.gui.uva.es/udigital/

El Universo Digital del IBM PC, AT y PS/2.

[2] ftp://download.intel.nl/design/

FTP de Intel en Holanda.

[3] http://www.sandpile.org/index.shtml

Technical 80x86 processor information.

[4] http://infopad.eecs.berkeley.edu/CIC/

CPU Info Center.

[5] http://www.cpu-central.com/

CPU Central.

SOFTWARE

[1] The Visible Computer (TVC).
[2] HelpPC 2.10 por David Jurgens.
[3] Turbo Assembler 4.0.
[4] Turbo Debugger 5.0.