LECCION 1:

ASM POR AESOFT. (lección 1).

- CONOCIENDO LA MAQUINA.

Hola a todos los seguidores del curso de ensamblador de AESOFT.
Esta primera lección va a ser sobre todo teórica.
No se puede programar en ensamblador sin tener un conocimiento mas o menos profundo acerca de las peculiaridades de la maquina que vamos a programar. Esto es obvio, ya que en ensamblador tenemos un control total del sistema, por tanto, cuanto mas sepamos acerca de el, mas provecho podremos sacar del mismo, y mejores resultados obtendremos.
En primer lugar debemos saber con que estamos trabajando, y no me refiero al lenguaje en sí, sino al ordenador en cuestión. Tendremos que conocer de que partes consta, como se comunican entre sí, etc.
Esta parte teórica es casi imprescindible para poder entender ciertas técnicas de programación.
Deciros también que vamos a estudiar el Pc en general, no distinguiremos entre distintas familias de procesadores (8086,286,386,etc) ya que vamos a utilizar ensamblador del 8086. En realidad es el ensamblador que yo utilizo. Por tanto, aunque estemos programando sobre un 80386 o un 80486, desarrollaremos código ejecutable para la familia 80x86 en general.
Cualquier exposición que haga ser v lida para toda la familia Pc.
Bueno... Despues de este preambulo, pasamos al meollo de la cuestión:
Un Pc es un ordenador o computador, como mas os guste, compuesto principalmente por procesador, chips de memoria, varios chips inteligentes o programables, y el bus de datos y direcciones. Junto con todo esto, nos encontramos los periféricos como son monitor, disqueteras, teclado, etc, que se comunican con el procesador. Esto (la comunicación del procesador con periféricos) se ver mas adelante.

El procesador:

Es el chip que ejecuta los programas. El procesador o CPU, lleva a cabo una gran variedad de cálculos, comparaciones numéricas y transferencias de datos como respuesta a las peticiones de los programas que están siendo ejecutados en memoria.
La CPU controla las operaciones b sicas del ordenador enviando y recibiendo señales de control, direcciones de memoria y datos de un lugar a otro del ordenador a través de un grupo de 'sendas electrónicas' llamadas BUS. Localizadas a lo largo de este bus están las puertas de entrada y salida (E/S en castellano o I/O en inglés), las cuales conectan a la memoria y a los chips de apoyo al bus. Los datos pasan a través de estas puertas de E/S mientras viajan desde y hasta la CPU y otras partes del ordenador.
Como os decía antes, trataremos el procesador 8086, siendo válido todo lo que digamos para el resto de procesadores de la familia PC.
El procesador 8086 es un procesador de 16 bits. Esto quiere decir entre otras cosas, que el procesador va a manipular en una sóla operación datos de hasta 16 bits. Es decir, cuando transfiera datos a la memoria o los traiga desde ella, lo podrá hacer de 16 bits en 16 bits. Aquí juega un papel decisivo el BUS de datos, ya que es por el por donde circulan los datos en las transferencias. Mas detalles unas cuantas líneas mas abajo.
El procesador cuenta con una serie de registros usados para realizar las operaciones de cálculo, y como almacenamiento de datos. Para que os hagais una idea, un registro del procesador es algo así como una zona de memoria dentro del procesador donde se puede almacenar información, de forma que el acceso a esta información es instantáneo,

ya que no hay que utilizar el bus de datos que conecta el procesador con la memoria para obtener dichos datos.
Estos registros se dividen en 5 grupos, según sus funciones:
1.- Registros de datos: AX, BX, CX y DX.
Se usan para cálculo y almacenamiento de propósito general. Son utilizados por los programas para realizar cálculos, así como para transferir datos de una posición de memoria a otra, ya que no se puede hacer de forma directa. Es decir, que no podemos transferir un dato de la posición de memoria X a la posición Y sin antes depositar ese dato temporalmente en un registro del procesador.
Estos registros tienen una longitud de 16 bits, pero podemos descomponerlos cuando nos interese en un par de registros de 8 bits. Quedando de la forma siguiente:
AX = AH + AL
Siendo AX el registro de 16 bits, compuesto por la conjunción (que no la suma) de el registro AH de 8 bits (los 8 bits mas signiicativos o de mas a la izquierda) y el registro AL de 8 bits (los 8 bits menos significativos o de mas a la derecha).
BX = BH + BL
CX = CH + CL
DX = DH + DL
Para estos tres registros se aplica lo mismo que para el registro AX.
Cada uno de estos registros tiene funciones especiales que es interesante conocer. Por ejemplo el registro AX es el llamado acumulador, hace que muchas operaciones tengan una forma mas corta, ya que lo especifican implícitamente.
Es decir, que hay operaciones que actúan sobre el registro AX en particular.
BX se suele utilizar en muchas instrucciones como registro base, sobre todo en transeferencias de datos entre memoria y procesador.
CX es el registro contador, muchas instrucciones lo utilizan para hacer incrementos o decrementos automáticos, para realizar bucles, etc. DX es el registro de datos, se suele utilizar para operaciones de 32 bits, para almacenar los 16 bits (o palabra) mas significativos.

2.- Registros Indice: SI, DI, BP y SP. Se utilizan para acceder a memoria cuando se establece el modo de direccionamiento mediante indexación o con punteros. SI y DI indican el índice fuente y destino respectivamente. BP y SP indican el puntero base y el puntero de la pila respectivamenete. Mas adelante hablaremos de la pila (que es un tipo de datos estructurado).
Estos 4 registros son de 16 bits, y no pueden ser utilizados como registros dobles de 8 bits.
3.- Registros de Segmento: CS, DS, SS y ES. Estos registros apuntan al principio de un bloque de 64 ks de memoria o segmento, de ahí lo de la 'S' con la que finalizan todos los registros: CS: registro segmento de código. Establece al área donde se halla el programa en ejecución. DS: registro segmento de datos. Especifica la zona donde el programa lee y escribe los datos por defecto. SS: registro segmento de pila. Especifica el área donde se encuentra la pila del sistema. ES: registro segmento extra. Se utiliza como una extensión del segmento de datos. Es decir, indica otro área de datos aparte del especificado por DS.
4.- Puntero de instrucción: IP. Su longitud es de 16 bits como el resto de registros. Indica la dirección de la siguiente instrucción a ejecutar, y su valor es ajustado durante la ejecución de la instrucción en curso. Esta dirección est en el área de 64 ks de direcciones especificado por CS.
CS e IP en conjunción conforman la dirección física real de la siguiente instrucción a ejecutar. Mas adelante detallaremos este asunto.
5.- Registro FLAGS. O banderas de estado.

Su longitud es de 16 bits. Cada uno de estos bits contiene cierta información booleano (verdadero o falso). Según el valor de cada uno de estos bits sea 1(verdadero) ó 0(falso), informar del estado de alguna situación en particular.
Dentro del registro de FLAGS hay 7 bits que no se utilizan. Los nombres de los utilizados son: Of, Df, If, Tf, Sf, Zf, Af, Pf y Cf.
Estos bits se clasifican en dos grupos: - Flags de estado (Cf, Af, Of, Zf, Pf y Sf): muestran el estado del procesador. - Flags de control (Df, If, Tf): determinan como el procesador responderá a determinadas situaciones. El programador manipulará estos bits para controlar el modo de ejecución de algunas instrucciones.
A continuación se muestra el significado de cada uno de los flags:
Cf: Bit de Carry (acarreo), se activa (se pone a 1) si se produce acarreo en una operación aritmética. Pf: Bit de paridad, se activa si el resultado de una operación tiene paridad par, es decir, si el resultado tiene un nº par de unos. Af: Bit de carry auxiliar, se activa si una operación aritmética produce acarreo de peso 16. Zf: Bit de cero, se activa si una operación produce 0 como resultado. Suele ser el mas utilizado(mas consultado) por los programadores, por lo menos por mí (AESOFT). Se utiliza para comparaciones de datos y para otras muchas cosas como ciertos bucles,etc. Sf: Bit de signo, se activa si el bit mas significativo de un resultado es 1. Por convención cuando se opera con números negativos, se utiliza el bit de mayor peso para indicar el signo: si el bit es cero, entonces se trata de un número positivo, si es 1, se trata de número negativo. Ya veremos todo esto mas adelante. Tf: Bit trap o desvío. Si Tf=1, el procesador ejecuta las instrucciones una a una bajo control del usuario. Se pone a 1 este bit para realizar depuraciones del código que se está ejecutando. De esta forma se puede seguir el flujo del programa. If: Bit de interrupción, si vale 1, las interrupciones están permitidas, y si vale 0, no. Df: Se usa en las instrucciones que manipulan cadenas de bytes. Según coloque el programador este bit, a '0' o a '1', las cadenas de bytes serán tratadas en sentido de direcciones crecientes o decrecientes. Of: Bit de overflow, indica desbordamiento en una operación aritmética.

La Memoria: Los chips de memoria se dedican meramente a almacenar la información hasta que se necesita. El número y la capacidad de almacenamiento de estos chips que hay dentro del ordenador determinan la cantidad de memoria que podremos utilizar para los programas y los datos. Normalmente la memoria siempre se ha dividido en dos tipos principales, como son la RAM y la ROM. De todos será conocida esta distinción y sus caraterísticas, así que no entro en el tema. Si alguien tiene alguna duda, que lo diga. Pues bien, en principio, nosotros vamos a trabajar con el rango de memoria 0 - 1048576, es decir, vamos a trabajar con el primer Mb(megabyte). Existen técnicas para tratar la memoria que hay en direcciones superiores a éstas, pero eso ya se ver en otro momento.
Pues bien, en ese Megabyte inicial, tenemos tanto RAM como ROM. La memoria RAM que es la que usamos para almacenar tanto programas como datos empieza en las direcciones bajas y llega hasta el inicio de la ROM. La ROM, evidentemente está situada en las posiciones altas de memoria, y es ahí donde se almacenan las rutinas mas básicas del ordenador, como las rutinas de acceso a discos, pantalla, etc.
Un tema muy importante en el mundo del PC es la denominada barrera de los 640 Ks que seguro habreis oido hablar. Esto quiere decir que aunque tengamos instalados 8 Megabytes en nuestro ordenador, sólo podremos ejecutar nuestro programa en las primeras 640 ks.
Para poder acceder a los 7 Megabytes restantes debemos utilizar funciones especiales de manejo de memoria extendida, expandida, etc. Pero bueno, eso no nos interesa ahora en absoluto. Os remito al fichero MEMO.ZIP que podreis encontrar en el BBS, en el cual, un usuario (Pc-adicto) nos ofrece una exposición mas amplia acerca de la memoria del Pc.

Chips inteligentes: Se trata de chips de apoyo, de los que se sirve el procesador o el usuario. Estos chips existen debido a que el procesador no puede controlar todo el ordenador sin ayuda. Al delegar ciertas funciones de control a otros chips, le queda mas tiempo libre para atender a su propio trabajo. Estos chips de apoyo pueden ser responsables de procesos tales como el flujo de información a través de la circuitería interna(como el controlador de interrupciones y el controlador DMA) y controlar el flujo de información de uno a otro dispositivo(como un monitor o una unidad de disco) concectado al ordenador. En resumen, estos chips están ahí para librar de trabajo al procesador.

Bus de direcciones: El bus de direcciones del Pc utiliza 20 líneas de señal para transmitir las direcciones de memoria y de los dispositivos conectados al bus. Como a través de cada una de esas 20 lineas pueden viajar dos posibles valores(0 ó 1, es decir tensíon alta o tensión baja), el ordenador puede especificar 2^20 (2 elevado a 20) direcciones. Es decir puede direccionar 1 Megabyte. Estamos hablando del 8086 con un bus de direcciones de 20 bits. Modelos superiores como el 80386 o el pentium direccionan muchísimo mas, tanto como les permite su bus: 32 bits en el caso del 386 y 64 bits en el caso del pentium. Es decir, 2^32 y 2^64 direcciones respectivamente.

Bus de datos: El bus de datos trabaja con el bus de direcciones para transportar los datos a través del ordenador. Este bus de datos es de 16 bits, al igual que el tama través o de registro. Aunque no tiene que coincidir, como sucede con el procesador 8088, el cual tiene un tamaño de registro de 16 bits y un bus de datos de 8 bits. Esto es así por simple economía. Por ahorrar costes. Claro que tiene su parte mala, y es que es mas lento al realizar transferencias de datos.
Bueno, bueno, bueno... Todo esto que parece muy rollero es muy importante. No es lo mismo programar sabiendo con que se trabaja que programar a ciegas. Muchas veces es imprescindible recrear gráficamente ciertas instrucciones, para darse cuenta de un fallo en el programa.
Venga, ahora quiero que me conteis dudas que teneis, aclaraciones, etc. Que la próxima lección ya es mas práctica. Pero hasta que no se tenga claro cómo funciona el ordenador, no se puede hacer que funcione correctamente.
Lo dicho, espero mensajes vuestros antes de la siguiente lección. Un saludo de AESOFT.

LECCION 2:

- DIRECCIONAMIENTO DE MEMORIA EN EL 8086. - SEGMENTACION.

Hola a todos los seguidores del curso de ensamblador de AESOFT. En esta lección vamos a ver cómo direcciona la memoria el 8086, es decir, cómo el 8086 accede a cada una de las posiciones de memoria.
La forma en que la CPU construye las direcciones de memoria es muy importante para la programación del sistema, debido a que constantemente utilizamos instrucciones de transferencias de datos, de acceso a funciones de la BIOS, del DOS, etc.
Mas adelante estudiaremos la BIOS. Valga por ahora que es un conjunto de utilidades y procedimientos grabados en la ROM (memoria de sólo lectura), los cuales se encargan de acceder al nivel mas bajo en cuanto a programación se refiere. Es decir, estas rutinas se encargan de manipular el hardware por nosotros. BIOS son las siglas de Basic Input Output System (Sistema básico de entrada/salida).
En cuanto al DOS (sistema operativo de disco), decir que aquí nos referi- mos no a las utilidades o comandos que trae consigo, que es lo típico que se enseña en academias e

institutos, sino a la estructura interna del mismo: interrupción 21h, 24h, etc. Ya veremos también que es una interrupción.
Bien, antes de entrar de lleno en el tema, conviene saber un poco del por que del mismo. Es decir, que llevó a que fuera de esta forma y no de otra.
A principio de los años 80, Intel (fabricante de la familia de procesadores 80x86) se propuso dar un gran paso adelante con respecto a la competencia. En aquel tiempo los microprocesadores que imperaban entre los ordenadores domésticos eran de 8 bits, es decir, tenían un ancho de bus de datos de 8 bits, el tamaño de palabra de memoria era de 8 bits, y los registros del procesador eran de 8 bits. Un claro ejemplo de esto fue el microprocesador Z80 (de la empresa Zilog), el cual estaba incorporado en máquinas tan famosas como los spectrum, amstrad, msx, etc.
Como he dicho, el ancho del bus de datos era de 8 bits. Esto quiere decir que todas las transferencias de datos que se hicieran se harían de 8 en 8 bits, es decir, byte a byte.
Pues bien, aunque el microprocesador era de 8 bits, y la mayoría de registros también lo fuera, había alguno mayor (16 bits). Me estoy refiriendo sobre todo al registro de direcciones que era de 16 bits. De esta forma, un amstrad cpc-464 podía acceder a 64 ks de memoria. 64 Ks es la máximo que podía direccionar el z80 original.
En ese momento Intel se planteó superar esa barrera de las 64 Ks, pero tenía un problema. El z80 por ejemplo, había consguido tener registros de 16 bits cuando el microprocesador es de 8. Pero pasar de 16 bits de capacidad en registros en aquellos momentos no era posible para los microprocesadores. Es decir, no había suficientes avances tecnológicos como para conseguir tamaños de registros mayores en un microprocesador. De tal manera que había que buscar una fórmula diferente... Y ahí es cuando surgió el tema de los segmentos que tantos quebraderos de cabeza a dado hasta ahora y sigue dando.
A Intel se le ocurrió la idea de construir una dirección de 20 bits de ancho y colocarla en el bus de direcciones para poder dirigirse a la memoria. Pero al ser los registros de 16 bits, sólo había una solución posible para crear este ancho de 20 bits: Usar 2 registros de 16 bits!!!
El 8086 divide el espacio de direcciones (1 Mbyte) en segmentos, cada uno de los cuales contiene 64 Ks de memoria (la máxima direccionable por un sólo registro). Entonces, para direccionar una posición de memoria nos valemos de dos registros: Registro de segmento y de offset. Ya vimos en la lección anterior que había varios registros de segmento: cs (registro de segmento de código), ds (de datos), etc.
Pues bien, este primer registro (de segmento), indica dónde comienza el trozo de 64 Ks que buscamos. Y el segundo registro (el de offset), contiene el desplazamiento dentro de ese segmento.
Bien. Hemos visto que son necesarios 2 registros para direccionar ese Mbyte de memoria, y tenemos un bus de direcciones de 20 bits. Esto nos conduce a que el microprocesador debe realizar unas operaciones sobre estos dos registros para obtener la dirección física de 20 bits. Esto se logra de la siguiente manera: El 8086 mueve el valor del segmento 4 bits a la izquierda y le suma el valor del desplazamiento para crear una dirección de 20 bits.
Gráficamente: Tenemos dos registros de 16 bits. DS: XXXXXXXXXXXXXXXX BX: XXXXXXXXXXXXXXXX 15 87 0 15 87 0 Byte alto Byte bajo Byte alto Byte bajo
(mas significativo)(menos significativo)
El primer registro, es el de segmento (en este caso, segmento DS, de datos). El segundo registro es el de offset o desplazamiento. En este caso utilizamos el registro BX para direccionar dentro de el segmento. Podíamos haber utilizado también el registro SI, el DI, etc. A partir de estos dos registros, debemos acceder a una posición de memoria física dentro del Mbyte de que disponemos para el 8086.

Pongamos que el registro DS tiene el valor 0B800h (en hexadecimal) (podeis utilizar SB-CALCU de SAN BIT para hacer los cambios de base, y trabajar con bases diferentes a la decimal. También para la decimal, por supuesto). Y el registro BX contiene el valor 0037h.
Tenemos pues (en binario): DS: 1011100000000000 BX: 0000000000110111 Para obtener la dirección física de memoria, y teniendo en cuenta todo lo dicho relativo a segmentos, el microprocesador acturaría así: (Gráficamente)
Haría una suma de la siguiente forma:
DS: 1011100000000000 BX: + 0000000000110111 ---------- 10111000000000110111
Obteniendo así la dirección de 20 bits necesaria para cubrir todo el Mbyte.
Si ese número (10111000000000110111) que está en binario, lo pasamos a hexadecimal, tenemos que la dirección física correspondiente a la anterior segmentada es: 0B8037h.
De todo lo anterior, se desprende que los segmentos empiezan siempre en direcciones divisibles por 16. Mas técnicamente: cada segmento comienza en una dirección de párrafo. Un párrafo son 16 bytes. Por supuesto nunca habrá un segmento que empiece en una dirección impar, por ejemplo. Como ejemplo: El primer segmento posible empieza en la dirección física 0. El segundo empieza en la dirección
Esto es mas complejo de lo que parece. Si tienes alguna duda, ya sabes...
Si le das vueltas a la idea, te darás cuenta que diferentes combinaciones de direcciones segmentadas dan una misma dirección física.
También se puede apreciar que los segmentos se pueden superponer unos a otros, pueden ser idénticos, o pueden encontrarse en partes totalmente lejanas en la memoria.
Si llegados a este punto no comprendes el tema de los segmentos, no sigas, ya que te perderías. Dime que no entiendes y lo explicaré mas detalladamente.
Es todo por ahora. Ah, y no sé si recibísteis la primera lección. Parece que todo estaba claro. Si en éste no os surge ninguna duda, ya me mosqueo }:-)))
Saludos. AESOFT..

LECCION 3:

ASM POR AESOFT. (lección 3). ----------
- CHIPS DE APOYO (Ampliación de la lección 1). ----------

Hola a todos los seguidores del curso de ensamblador de AESOFT.
A petición de algunos lectores de este curso, incluir, en esta lección una ampliación de la primera. Mas concretamente, desarrollar, un poco mas el tema de los chips de apoyo (inteligentes, porgramables, etc.) que toque tan ligeramente. Aunque aún no es el momento de estudiarlos por separado, y por tanto en profundidad, daré una relación de ellos, y que función realizan.
Bueno, menos rollo y al grano:

- CHIPS DE APOYO (Ampliación de la lección 1): ---------- Ya vimos en la primera lección que se entendía por chips de apoyo, soporte, etc. También llamados controladores, ya que controlan una parte del hardware para ir aligerando el trabajo de la CPU. De esta forma la CPU tiene mas tiempo para la ejecución del programa correspondiente. En muchos casos, estos chips son programables. Por supuesto, estos chips pueden ser programados por el programador en ensamblador (valga la redundancia), con lo cual no trabajan por su cuenta, sino que aceptan las instrucciones que les hacen funcionar a través de la CPU.

A continuación se da una relación de los diferentes chips de apoyo o controladores del Pc:
- El controlador programable de interrupciones (chip 8259) En un Pc, una de las tareas esenciales de la CPU consiste en responder a las interrupciones del hardware. Una interrupción del hardware es una señal generada por un componente del ordenador que indica que ese componente requiere la atención del procesador. Por ejemplo el reloj del sistema, el teclado, y los controladores de disco, generan interrupciones de hardware en un momento dado para que se lleve a cabo su tarea. En ese momento, la CPU responde a cada interrupción, llevando a cabo la actividad de hardware apropiada, ejecutando lo que se llama rutina de atención a la interrupción, que es una porción de código que se ejecuta como respuesta a una petición de interrupción.
--- Tomemos como ejemplo el teclado. (Puede ser conveniente leer antes el apartado 'Interrupciones', que viene desarrollado mas abajo).
El usuario pulsa una tecla. Inmediatamente, la circuitería digital del periférico detecta la pulsación de la tecla y almacena su "código de rastreo" (toda tecla tiene asociado un código de 8 bits denominado scan code) en un registro reservado para tal fin, llamado puerto de teclado. (Mas adelante, al hablar de puertos, se amplía la información). Entonces, el teclado activa una línea de petición de interrupción, mas concretamente, la línea IR1 del 8259. (IR son las siglas de Interrupt Request, o petición de interrupción. También se puede decir IRQ, que es a lo que estamos mas acostumbrados, sobre todo cuando instalamos una tarjeta de sonido o algo por el estilo). A continuación, el 8259 activa el pin INTR de la CPU. (El pin INTR se activa cada vez que se produce una petición de interrupción, es una línea externa que comunica al Procesador con el exterior). Por último, y resumiendo mucho, la CPU termina la instrucción en curso, y ejecuta la rutina de atención a la interrupción. Al terminar de ejecutar esta rutina, el control vuelve a la siguiente instrucción por donde se había quedado en el programa en curso. Todos los registros deben tener el valor que tenían antes de ejecutar dicha rutina.
--- El controlador programable de interrupciones se llama a menudo por sus siglas: PIC.
- El controlador DMA (chip 8237). Algunas partes del ordenador son capaces de transferir datos hacia y desde la memoria, sin pasar a través de los registros de la CPU. Esta operación se denomina acceso directo a memoria o DMA (Direct Memory Access), y la lleva a cabo un controlador conocido como controlador DMA. El propósito principal de dicho controlador, es el de permitir a las unidades de disco leer y escribir datos prescindiendo de pasar por los registros del microprocesador. De esta forma, las transferencias de datos se hacen mas rápidas. Pero esto es sólo en teoría, ya que con los modernos procesadores que cuentan con una frecuencia de proceso varias veces mas rápida que la del bus, el controlador DMA, apenas ofrece ninguna ventaja.
- El Interface de periferia (chip 8255). El interface de periferia crea una conexión entre la CPU y los dispositivos periféricos como el teclado y el altavoz. Actúa como una especie de intermediario utilizado por la CPU para comunicar determinadas señales al dispositivo deseado.
- El generador de reloj (chip 8248). Este generador siministra las señales de reloj que coordinan el microprocesador y los periféricos. Produce una señal oscilante de alta frecuencia. Por ejemplo, en el IBM PC original esta frecuencia era de 14,31818 megahercios o millones de ciclos por segundo. No hay que confundir esta frecuencia con la frecuencia del procesador. Otros chips que necesitan una señal de tiempo regular, la obtienen del generador de reloj, dividiendo la frecuencia base por una constante para obtener la frecuencia que necesitan para realizar sus tareas. Por ejemplo, el 8088 del IBM PC, funcionaba a 4,77 MHz, una tercera parte de la frecuencia base. El bus interno del IBM PC y el temporizador utilizan una frecuencia de 1,193 MHz, es decir, un cuarto del ratio del 8088 y una doceava parte del ratio base.

- El temporizador o timer (chip 8253). Este chip genera señales de tiempo a intervalos regulares controlados por software. Esto es, que podemos cambiar la frecuencia de estos intervalos por medio de un programa. El timer dispone de varias líneas de salida, funcionando cada una con una frecuencia independiente a las otras, y conectadas cada una a otros componentes del sistema. Una función esencial del contador es la de generar un tic-tac de reloj que mantenga actualizada la hora del día. Otra de las señales producidas por el contador puede ser utilizada para controlar la frecuencia de los tonos producidos por el altavoz del ordenador.
- El controlador de vídeo (chip 6845). El controlador de vídeo, al contrario del resto de chips de apoyo presentados hasta ahora, no se encuentra en la placa madre del PC, sino que está depositado en una tarjeta de video colocada en una ranura de ampliación. Es el corazón de las tarjetas de video CGA, EGA, VGA, etc.
- Controladores de entrada/salida. Los PCs tienen varios subsistemas de entrada/salida con circuitería de control especializada que proporciona un interfaz entre la CPU y el hardware de E/S. Por ejemplo, el teclado tiene un chip controlador propio que transforma las señales eléctricas producidas por las pulsaciones de teclas en un código de 8 bits que representa la tecla pulsada. Todas las unidades de disco disponen de circuitería independiente que controla directamente la unidad. La CPU se comunica con el controlador a través de un interfaz. Los puertos serie y paralelo también disponen de sus propios controladores de entrada/salida.
- Los coprocesadores matematicos (8087/80287/80387). Son utilizados en caso de estar disponibles en el ordenador, para trabajar con números en coma flotante y coma real, cosa que el 8086 no puede hacer.
Todos estos chips, se conectan entre sí, a través del BUS, que ya sabemos en que consiste.

Esto es todo por ahora. Un saludo. AESOFT....

LECCION 4:

ASM POR AESOFT. (lección 4). ----------
- ENTRADA/SALIDA (COMUNICACION CON EL HARDWARE I). - INTERRUPCIONES (COMUNICACION CON EL HARDWARE II). - LA PILA DEL 8086. ----------
Hola a todos. En esta lección vamos a tratar aspectos muy interesantes de la programación del Pc, interrupciones, puertos, etc. Espero que tengais claro lo del error que os comentaba en la lección 1, acerca de los buses. Si no es así, a que esperais para preguntar... Bueno, seguimos con lo nuevo:
- ENTRADA/SALIDA (COMUNICACION CON EL HARDWARE I). ---------- La comunicación entre un programa y el hardware, es decir los chips de apoyo y las tarjetas de ampliación, se efectúa mediante los llamados Ports (puertos, en castellano). Estos puertos son zonas de memoria de 1 o 2 bytes de tamaño, en las cuales se depositan los datos que van a ser utilizados por los chips o tarjetas, y también se depositan los datos que devuelven estos chips o tarjetas al procesador.
En el Pc, existe una zona de memoria de 64 Ks, ajena a la memoria principal, dedicada a los puertos. Es decir, estos 64 Ks de memoria no tienen nada que ver con la memoria disponible para los programas.
Para realizar los movimientos de datos entre puertos y procesador existen instrucciones especiales: IN y OUT. Tanto IN como OUT tienen dos formas de uso, las cuales utilizan el

registro AL o AX para almacenar los datos a leer o escribir. La forma difiere en función de que se quiera acceder a un puerto menor de 256 o mayor.
+ M,todo directo o estático: corresponde cuando se quiere acceder a un puerto menor de 256.
IN AL,10H ---> lee un byte del puerto 10h IN AX,10H ---> lee una palabra del puerto 10h OUT 0FFH,AL --> escribe el valor de AL en el puerto 0FFH
+ M,todo inndirecto o dinamico: corresponde al caso de querer acceder a un puerto mayor de 256, para lo cual se utiliza el registro DX indicando el número de puerto.
IN AL,DX ----> lee un byte del puerto indicado por DX. (Antes hemos tenido que introducir en DX la dirección del puerto). OUT DX,AX ----> escribe la palabra contenida en AX en el puerto DX.
Algunos ejemplos de puerto son: 60H ----------> acepta entradas de teclado. 61h ----------> controla el altavoz. 3F0H-3F7H ----> Opera sobre la controladora de discos.
En el PC cualquier subsistema, exceptuando la memoria, está controlado por el procesador a través de los puertos.
- INTERRUPCIONES (COMUNICACION CON EL HARDWARE II). ---------- Las interrupciones constituyen la forma en que la circuitería externa informa al microprocesador de que algo ha sucedido (como que se ha pulsado una tecla, por ejemplo) y solicita que se emprenda alguna acción. Pero no acaba ahí su utilidad, ya que las interrupciones ademas son el medio principal de comunicción entre las funciones de la BIOS y el DOS. En este segundo caso, son mal llamadas interrupciones. Mas bien habría que decir funciones, ya que nos sirven para hacer una llamada a una función BIOS o DOS, como por ejemplo la acción de cambiar de modo de video, para la cual se utiliza la interrupción 10h (Driver o controlador de vídeo), con el número adecuado de función. Mas adelante veremos cómo llamar a una función.
Al primer tipo de interrupciones se les denomina interrupciones de hardware, y son las interrupciones reales. Esto es, que estando un programa en ejecución, se interrumpe éste para ejecutar un trozo de código necesario para atender a la petición de un dispositivo, como puede ser el teclado. Acto seguido, se reanuda la ejecución del programa en cuestión. Son las interrupciones que vimos en la lección 3, al hablar del PIC.
Al segundo tipo de interrupciones se les denomina interrupciones de software, y son las 'ficticias', ya que no hay ningún dispositivo pidiendo atención del procesador, sino que es el programa del usuario el que ejecuta una función BIOS o DOS, mediante una interrupción. En este caso, no se interrumpe el programa de forma súbita, sino que es dicho programa el que lanza una interrupción, la cual tiene su rutina de atención a la interrupción (como vimos en la lección 3) 'conectada' a un grupo de funciones o rutinas.
Veamos las interrupciones con mas detalle:
+ La tabla de vectores: ---------- Toda interrupción aceptada conduce a la ejecución de un subprograma específico, como hemos visto. Pero cómo sabe el procesador dónde empieza este subprograma, una vez que atiende a la interrupción... La respuesta nos la da la tabla de vectores.
Esta tabla de vectores contiene las direcciones de comienzo o punteros al subprograma de atención a la interrupción. La tabla está compuesta de 256 entradas. Es decir, son posibles 256 interrupciones diferentes en el PC.
Cada una de estas entradas, contiene la dirección de inicio del código de atención a una interrupción, en la siguiente forma: 2 primeros bytes (una palabra) que contienen la dirección base del segmento, y los 2 últimos bytes que contienen el desplazamiento. En total 4 bytes para indicar el comienzo de una interrupción, en la forma segmento:desplazamiento.
Ya vimos en la segunda lección cómo transformar una dirección segmentada (segmento:desplazamiento) en una dirección física o real.

Durante la aceptación de una interrupción, el 8086 carga la dirección base del segmento en el registro CS y el desplazamiento en el contador de programa IP. De esta forma, la siguiente instrucción a ejecutar, que viene dada por los registros CS:IP, ser la primera del subprograma de atención a la interrupción.
+ Pines (líneas de bus) para demandar interrupción desde el exterior. ---------- Existen 3 líneas externas jerarquizadas que son, por orden de prioridades decrecientes: RESET, NMI e INTR. Sólo INTR es enmascarable (cuando un pin de demanda de interrupción está enmascarado -inhabilitado- la activación del pin, no produce ninguna interrupción).
Es decir, que si se activan los pines RESET o NMI, siempre van a conducir a la ejecución de una interrupción. Pero si se activa el pin INTR, tenemos dos opciones (dependiendo de si está enmascarado o no), que son hacer caso omiso de la petición de interrupción, o atender dicha interrupción, respectivamente.
Pin INTR: Una petición de interrupción sobre este pin es enmascarable mediante el bit IF (bandera de interrupción) del registro FLAGS o registro de estado. Este bit IF, es la mascara de INTR. Para saber si está enmascarada o no la línea INTR, se mira este flag. El cual puede tener (obviamente) dos valores: 0 y 1. Enmascarado es 0.
Para manipular este bit, disponemos de dos instrucciones en ensamblador: CLI (Clear IF, o borrar flag IF) que lo pone con valor 0. STI (Set IF, o activar flag IF) que lo pone con valor 1.
La petición de interrupción se realiza activando el pin INTR con nivel alto (1) y debe mantenerse así hasta que por el pin INTA (pin asociado al pin INTR. Es activo a nivel bajo (0), indicando que se ha aceptado la interrupción solicitada por medio del pin INTR) el 8086 indique que ha sido aceptada.
Entonces... Contamos con el pin INTR para pedir al procesador atención a
una interrupción, y con el pin asociado INTA que estr con valor (0)
cuando la interrupción haya sido aceptada.
INTR ---> Interrupt Request (petición de interrupción). INTA ---> Interrupt Accepted (interrupción aceptada).
Veamos cómo actúa la CPU desde que se activa el pin INTR hasta que se retorna del subprograma de atención a la interrupción:
Debido a que la interrupción interrumpir al programa en ejecución en cualquiera de sus instrucciones, es necesario resguardar el contenido del registro de estado (FLAGS), para que al volver de la interrupción, tengan las banderas el mismo valor. Y sobre todo, hay que guardar la dirección de la siguiente instrucción a ejecutar en el programa actual.
Pero dónde se guardan todos estos datos... En una zona de memoria denominada PILA, la pila del procesador. (Explicación en el último apartado de esta lección). Al acto de introducir un dato en la pila se le denomina apilar, y a sacarlo de la misma se le denomina desapilar.
Pues bien, el procesador hará lo siguiente: - Apila el contenido del registro de estado (flags) - Apila la dirección de retorno (contenido de los registros CS e IP). - Inhibe las interrupciones (IF=0 y TF=0, mas adelante se comenta la utilidad del flag TF o TRACE). Esto se hace para que no se produzca otra interrupción durante la secuencia de aceptación de la interrupción. Esto es muy importante. - Activa el pin INTA (lo pone a nivel bajo). El dispositivo que ha solicitado la interrupción, al notar el cambio en el pin INTA, queda enterado de la aceptación de la interrupción. - Lee el número del vector de interrupción del bus de datos. Previamente, el dispositivo lo ha depositado en respuesta a la activación del pin INTA. - Obtiene la dirección del subprograma de atención a la interrupción. Dicha dirección se encuentra (como hemos visto antes) almacenada en la tabla de vectores. - El 8086 ejecuta la subrutina que finaliza con la instrucción IRET, o

Retorno de Interrupción, cuya ejecución restituye en CS e IP la dirección de retorno salvada en la pila, y en el registro de estado el valor de los flags.
Al restaurar los flags, se anula la inhibición anterior de IF y TF, con lo cual, otra vez se aceptan interrupciones. Pudiendo así tener interrupciones en cascada.
Repasar el ejemplo de la pulsación de tecla de la lección 3, a ver si ahora se ve con mas claridad.
Pin NMI: Este pin está reservado a acontecimientos graves, como puede ser un corte de corriente, un error de memoria, del bus, etc.
La activación de NMI no conlleva ninguna lectura en el bus de datos del nº de vector de interrupción, sino que la CPU directamente busca el vector de interrupción número 2.
Pin RESET: Inicializa el sistema. En la petición de RESET no se almacena nada en la pila ni se accede a la tabla de vectores para conseguir la dirección de comienzo. Al activar el pin RESET, el registro de estado queda borrado (0). CS = 0FFFFh. IP = 00000h. De esta manera, la siguiente instrucción a ejecutar por el procesador es la contenida a partir de FFFF:0, código de reinicialización y carga del sistema operativo. Son los últimos bytes de la ROM. El resto de registro de segmentos quedan con valor 0. DS = 0000 ES = 0000 SS = 0000
+ Interrupciones internas o desvíos. ---------- El microprocesador 8086 tiene 2 interrupciones internas: 'División imposible' y 'funcionamiento paso a paso (TRACE)'.
División imposible: Se produce cuando se divide por 0, o cuando el cociente resultante de la división no cabe en el registro preparado para contenerlo. En ambos casos, se ejecuta la interrupción 0.
Funcionamiento paso a paso: Si el programador coloca a (1) el bit TF (TRACE) del registro de estado, al final de cada instrucción, la CPU bifurcará a la posición de memoria indicada por el vector de interrupción número 1. Esto es lo que utilizan los debuggers o depuradores de código para hacer un seguimiento del programa, instrucción por instrucción.
Mas adelante, cuando hablemos acerca de la programación de utilidades residentes, entraremos en la programación práctica de las interrupciones. Valga lo dicho hasta ahora como base teórica.
- La pila del procesador: ---------- La pila es una característica interna del 8086. Es una estructura de datos situada en la RAM. Proporciona a los programas un lugar donde almacenar datos de forma segura, pudiendo compartirlos con otros procedimientos o programas de forma cómoda y práctica.
La función mas importante de la pila es la de mantener las direcciones de retorno en las llamadas a procedimientos e interrupciones, así como guardar los parámetros pasados a estos procedimientos. La pila también se utiliza para almacenamiento temporal de datos dentro de un programa, y para muchas cosas mas que se aprenden con la práctica.
La pila tiene su nombre por analogía con los montones de platos apilados (pilas de platos). Cuando un dato nuevo es introducido en la pila, se dice que es apilado (push) debido a que se sitúa por encima de los demas, es decir se sitúa en la CIMA de la pila.
Una pila opera en el orden último-en-entrar - primero-en-salir: LIFO (LAST IN FIRST OUT) o lo que es lo mismo, el último en entrar es el primero en salir.
Esto significa que cuando la pila se utiliza para seguir la pista de los retornos de las subrutinas, la primera llamada a subrutina que se hizo, es la última que se devuelve. De esta manera, la pila mantiene ordenado el funcionamiento del programa, las subrutinas y rutinas de tratamiento de interrupción, sin importar la complejidad de la operación.
La pila crece en orden inverso. Es decir, a medida que se añaden nuevos datos, la cima de la pila se acerca mas a posiciones mas bajas de memoria.

Existen 3 registros destinados a gestionar la pila. Registro de segmento de pila (SS): que indica la dirección base del segmento de pila Puntero de pila (SP): que apunta a la cima de la pila. Puntero base de pila (BP): que se usa para moverse a través de la pila sin cambiar la cima. Se suele utilizar para acceder a los distintos parámetros al llamar a una función.
Los elementos que se almacenan en la pila son del tipo palabra (2 bytes). Esto quiere decir, entre otras cosas, que el puntero de pila (SP), así como el puntero base de pila (BP), incrementan/decrementan en 2 su valor para apuntar a un nuevo elemento dentro de la pila, fruto de apilar o desapilar un elemento.
También conlleva el que si queremos almacenar un byte en la pila, primero lo debemos convertir en palabra (2 bytes), y luego almacenar esa palabra. Esto es muy sencillo, sólo hay que meter ese byte o registro de 8 bits en un registro de 16 bits y almacenar este registro.
Las instrucciones para manejar la pila son: PUSH ---> Guarda un dato en la pila. Decrementando SP en 2 unidades, para que apunte al nuevo elemento a introducir. Ejemplo: PUSH AX --> Apila el contenido de AX en la cima de la pila.
POP ----> Obtiene un dato de la pila. Incrementando SP en 2 unidades, para que apunte al nuevo elemento a introducir. Ejemplo: POP AX --> Desapila el contenido de la cima de la pila en el registro AX. Es decir, AX contendrá el valor que hubiera en la cima de la pila, y el puntero de pila se actualiza incrementándolo en 2.
PUSHF --> Guarda el contenido del registro de estado (FLAGS) en la pila. Decrementando SP en 2 unidades, para que apunte al nuevo elemento a introducir. No es necesario indicar sobre que actúa esta instrucción, lo lleva implícito en su nombre PUSHF (PUSH FLAGS).
POPF ---> Introduce en el registro FLAGS el contenido de la cima de la pila. Incrementando SP en 2 unidades, para que apunte al nuevo elemento a introducir. Al igual que con la instrucción anterior, no es necesario indicar sobre que actúa esta instrucción POPF (POP FLAGS).

Conviene recordar el hecho de que la pila crece en orden inverso al normal, es decir de direcciones de memoria altas a direcciones bajas. Por lo tanto es necesario tener en cuenta el uso que se va a hacer de la pila en el programa, debido a que si reservamos espacio en nuestro programa para una pila pequeña, en caso de sobrepasarla haciendo muchos push seguidos, machacaría nuestro programa.
Hay que tener en cuenta que no sólo es nuestro programa el que utiliza la pila mediante la instrucción PUSH y mediante llamadas a procedimientos, interrupciones, etc. Sino que mientras nuestro programa corre se están sucediendo numerosas interrupciones que conllevan muchos PUSH. Por ejemplo, 18'2 veces por segundo se produce la interrupción de reloj, con lo cual, todas estas veces se está apilando y posteriormente quitando información de la pila.
Por regla general, basta con tener una pila de unos 2 KS, es decir, espacio para almacenar 1024 elementos. Es muy difícil que se sobrepase este tamaño.

Bueno... Aquí seguro que hay dudas. Venga, decidme que quereis que explique mas detenidamente, que dentro de un par de lecciones empezamos a programar, y hay que tenerlo todo claro.
Un saludo. AESOFT....

LECCION 5:

ASM POR AESOFT. (lección 5). ----------

- CODIFICACION DE LAS INSTRUCCIONES EN EL 8086. ----------

Hola de nuevo, aplicados alumnos :-) En esta lección vamos a tratar conceptos muy t,cnicos acerca del formato de las instrucciones en código m quina. Veremos cómo se codifican las instrucciones en el 8086.

- CODIFICACION DE LAS INSTRUCCIONES EN EL 8086. (Este apartado es muy t,cnico. Aunque no es imprescindible comprender lo que se expone a continuación para programar en ensamblador, es muy útil conocer cómo el procesador interpreta lo que le 'pedimos'. Esto nos da un mayor conocimiento acerca de la maquina en cuestión. Y de esta forma entendemos el porque de ciertas sintaxis de instrucciones. Y resolveremos mas f cilmente los errores una vez que se nos presenten). ---------- Cada procesador tiene un conjunto de instrucciones para manejarlo, así como para manejar la maquina por medio de el. Indistintamente del lenguaje de programación que estemos utilizando, cuando obtenemos el ejecutable, ,ste est compuesto únicamente por ese tipo de instrucciones b sicas (instrucciones de código m quina). Dependiendo de la calidad y prestaciones de ese lenguaje de programación, el código resultante, necesitar mas instrucciones del procesador o menos. De todos es conocido, que hay lenguajes de alto o medio nivel (como C, pascal, basic, etc.) en los que para una misma tarea, uno dar un ejecutable mas grande que otro. Velocidad, aparte. Esto no sucede así con ensamblador, en el que para cada instrucción, existe una y sólo una instrucción en código m quina.

Pues bien, ahora vamos a ver la estructura de esas instrucciones b sicas o de código m quina.

Las instrucciones del 8086 se codifican sobre 4 campos como m ximo, y tienen un tamaño de 1 a 6 bytes. Es decir, dependiendo de la instrucción de que se trate, necesitar mas o menos bytes para su codificación, así como mas o menos campos.

Los cuatro campos en una instrucción código maquina son: 1.- Código de operación: Este campo siempre aparece (obviamente). Una vez que el procesador descifra el significado de este campo, sabe si la instrucción consta de mas campos o si se trata de una instrucción de un sólo campo. 2.- Modo de direccionamiento (byte EA): Le indica al procesador el número de operandos que acompañan al código de operación, así como el tipo de estos operandos(registros, memoria, valor inmediato). 3.- Desplazamiento del dato (sobre 8 o 16 bits): En caso de existir este campo, supone un desplazamiento sobre la dirección dada por un registro índice o base (especificado este registro mediante el byte EA). 4.- Valor inmediato (sobre 8 o 16 bits): Almacena un valor num,rico de 8 o 16 bits, que va a ser utilizado para una transferencia, una operación aritm,tica, etc.

Ahora entramos un poco mas en detalle:

Primero veremos un esquema de una instrucción código m quina:

Ú----------¿ ³ 8 bits 2 3 3 8 ó 16 bits 8 ó 16 bits ³ ³ ÉÍÍÍÍÍÍÍÍÍÍÍ» ÉÍÍÍÑÍÍÍÑÍÍÍ» ÉÍÍÍÍÍÍÍÍÍÍÍÍÍ» ÉÍÍÍÍÍÍÍÍÍÍÍÍÍ» ³ ³ º código de º º ³ ³ º º º º Valor º ³ ³ º operación º ºMOD³REG³R/Mº ºDesplazamientoº º Inmediato º ³ ³ ÈÍÍÍÍÍÍÍÍÍÍÍ¼ ÈÍÍÍÏÍÍÍÏÍÍÍ¼ ÈÍÍÍÍÍÍÍÍÍÍÍÍ¼ ÈÍÍÍÍÍÍÍÍÍÍÍÍÍ¼ ³ ³ _-- 1 byte -_ _-- 1 byte -_ _ 1 ó 2 bytes -_ _ 1 ó 2 bytes -_ ³ À----------Ù

- El código de operación est codificado sobre 8 bits. Por medio de este campo se sabe si va a ser necesario cualquier otro de los tres restantes. También el código de operación contiene información acerca de si se va a trabajar con palabras o con bytes.

- Byte EA ó Modo de direccionamiento: Contiene 3 campos. Los campos MOD y R/M especifican el modo de direccionamiento, y el campo REG especifica el registro de que se trata en la instrucción.
El campo MOD que es de 2 bits puede tener 4 valores diferentes: Los 3 primeros seleccionan el desplazamiento en los modos de direccionamiento de memoria. El cuarto selecciona un registro. Detallemos la función de estos bits en cada una de las 4 posibilidades: 00 ---> No hay desplazamiento. 01 ---> Se usa un byte para codificar el desplazamiento. 10 ---> Se usan 2 bytes (una palabra) para codificar el desplazamiento. 11 ---> Hace que R/Maseleccione un registro usando la misma codificación de los registros que para REG (ver mas abajo), en lugar de un modo de direccionamiento de la memoria. Es decir, que se produce una transferencia de un registro a otro.
El campo REG que es de 3 bits codifica el registro empleado. Por tanto es posible especificar hasta 8 registros diferentes por medio de este campo. Dependiendo de que se trate de acceso a palabras o a octetos, se seleccionar un registro de entre un grupo de 8, o de un segundo grupo de 8 registros. Para cuando se accede a registros de 16 bits, el campo REG codifica los registros de palabra de la siguiente manera:
AX (000), CX (001), DX (010), BX (011) SP (100), BP (101), SI (110), DI (111)
Cuando se accede a registros de 8 bits, la codificación de los registros de tamaño byte queda como sigue:
AL (000), CL (001), DL (010), BL (011) AH (100), CH (101), DH (110), BH (111)
El campo R/M indica el segundo registro (si lo hay) o el tipo de direccionamiento a memoria.
En caso de que haya segundo registro, ,ste se codifica de la misma forma que para el campo REG.
En caso de que se trate de un modo de direccionamiento de memoria, estos tres bits seleccionan uno de los modos de direccionamiento posibles de acuerdo con la siguiente tabla:
000 desplazamiento final = [BX] + [SI] + desplazamiento 001 desplazamiento final = [BX] + [DI] + desplazamiento 010 desplazamiento final = [BP] + [SI] + desplazamiento 011 desplazamiento final = [BP] + [DI] + desplazamiento 100 desplazamiento final = [SI] + desplazamiento 101 desplazamiento final = [DI] + desplazamiento 110 desplazamiento final = [BP] + desplazamiento 111 desplazamiento final = [BX] + desplazamiento

- El desplazamiento en caso de existir, supone un incremento en la dirección dada por un registro índice o base, dando lugar así a un desplazamiento final, dentro de un segmento dado. Es decir, como se ve en la tabla superior, podemos acceder a memoria a través de un registro base (BX) o un registro índice (SI, DI), etc, o bien hacerlo a través de uno de esos registros, pero ayud ndonos de un desplazamiento que se suma a la dirección que tienen establecida esos registros. Veremos mas adelante la utilidad de utilizar desplazamientos sobre un registro base o índice.
Como ejemplo: Tenemos el registro DI apuntando a (con valor igual a) la dirección 3000h (direcciones siempre en hexadecimal). En esa dirección tenemos el comienzo de una cadena de caracteres que queremos convertir a myúsculas. Y una vez que los hemos convertido, los queremos copiar a la memoria de pantalla.
Pues bien, podemos ir incrementando DI para tratar cada uno de estos caracteres, o bien podemos utilizar DI junto con un desplazamiento para acceder a cada uno de los caracteres. Es decir, para acceder al primer elemento sería DI+0, para el segundo, sería DI+1, etc. De esta forma, al terminar la tarea, DI seguir apuntando al principio de la cadena, y podremos copiar la cadena desde el principio a donde corresponda.
Si no utiliz ramos desplazamiento, tendríamos que tener una variable apuntando al inicio de la cadena, para tenerlo luego localizable. Bueno... Esto es un simple ejemplo. Las

posibilidades que nos ofrece el utilizar desplazamientos acompañando al registro base o índice son mucho mas interesantes que lo que acabamos de ver en el ejemplo.
- El valor inmediato se utiliza cuando hacemos movimientos de datos a registros o a memoria. Por ejemplo queremos introducir en el registro AX la cantidad 37867 (93EBH), pues ese 37867 sería el valor inmediato.
En ensamblador la instrucción sería: MOV AX,37867 Simple, ¨no? Mover (MOV) la cantidad 37867 al registro AX. Próximamente se ver el resto de instrucciones en ensamblador, mientras tanto, y por ser necesario ahora, aprenderemos el uso de la instrucción MOV.
La instrucción como hemos podido ver, se utiliza para movimientos o transferencias de datos: de registro a registro, de registro a memoria, y de memoria a registro. Pero nunca de memoria a memoria, ya que la arquitectura del procesador y bus no lo permiten.
La sintaxis b sica de la instrucción es la siguiente: MOV destino,fuente. El destino siempre a la izquierda, y la fuente a la derecha.
Ejemplos: * MOV ax,5 ---> mueve el valor inmediato (o dato) 5 al registro AX. Examinemos esta instrucción. Alguien podría pensar que como el valor 5 cabe en un sólo registro de 8 bits (AL en este caso), el registro AH quedaría como estaba antes de la instrucción. Pues no es así. Si le decimos al procesador que introduzca un 5 en AX, así se har . Poniendo a cero el registro AH, para que AX tenga el valor 5.
Veamos cómo se codifica esta instrucción: MOV AX,5 ---> B8 05 00 (Código m quina, siempre en hexadecimal). En primer lugar tenemos el primer byte que contiene el código de operación (B8). Debido a que este código de operación(B8) tiene implícita la utilización del registro AX como destino, no es necesario el byte EA ó byte de direccionamiento, que sí sería necesario para transferencias con otros registros. Como vimos en la primera lección al hablar de registros, el registros AX (AH, AL) se utiliza normalmente como acumulador, de tal manera que existen operaciones especiales para trabajar con el, como la instrucción B8 y otras muchas de movimiento de datos, en las que no se especifica el registro mediante el byte EA, ya que est implícito en el código de operaicón. De esta manera se gana velocidad en la ejecución del programa utilizando los registros para lo que han sido creados. AX acumulador, CX contador, etc.
Despues del código de operación tenemos dos bytes (1 palabra). Estos dos bytes forman el campo Valor Inmediato, que como vemos aquí es de 16 bits. Como os habreis dado cuenta, de los 4 campos que puede tener una instrucción código m quina, ,sta sólo tiene dos: El primero (código de operación), y el último (valor inmediato).
Y volviendo de nuevo al campo Valor inmediato y a su tamaño en esta instrucción (2 bytes): El orden de estos bytes es muy significativo. Veamos... Tenemos el valor 5 para introducir en una palabra. Lo normal sería que en el código se almacenara este cinco como (00 05), pues en el 8086 esto no es así. Como siempre, para acelerar el programa cuando se manejan transferencias de datos, se llegó a la conclusión de que si se almacenan los bytes que componen una palabra en orden inverso al normal, luego es mucho mas r pido recuperarlos. Y es así como se hace en la pr ctica. Cada vez que almacenamos una palabra en memoria, el byte de mayor peso queda a la derecha del byte de menor peso. De lo anterior se desprende que el número 5 al introducirlo en una palabra de memoria, quedaría como (05 00).
Otro ejemplo: Una vez que almacenamos el número 8BC3H en memoria, si hacemos un volcado de memoria para ver que tenemos, veremos que en memoria no est el número como 8BC3H, sino que nos encontramos con C38BH.

* MOV al,5 ---> Introduce el valor 5 en el registro AL. En este caso, sí que AH queda como estaba antes de la instrucción, ya que en la misma no interviene tal registro de ninguna forma (ni implícita al referirse a AX, ni explícita al referirnos a el en concreto). La

instrucción se codifica como: MOV AL,5 ---> B0 05 Este ejemplo es pr cticamente como el anterior, excepto que el código de operación en vez de ser B8 es B0, y ademas ya no hay 2 bytes en el campo valor inmediato, sino que hay uno sólo, ya que vamos a introducir el dato en un registro de tamaño byte.
Ejemplo cuando se trata de transferencias entre registros:
* MOV CX,SI ---> Introduce el valor del registro SI en el registro CX. La instrucción se codifica como: MOV CX,SI ---> 8B CE En esta instrucción tenemos un código de operando y el byte EA. Mediante este byte EA el procesador sabe que registros intervienen en la transferencia. Descomponiendo el byte EA en sus dígitos binarios, tenemos: CE ---> 11001110 El campo MOD con valor 11, hace que R/Maseleccione un registro como fuente. El campo REG con valor 001, indica que el registro destino es CX. El campo R/M con valor 110, indica que el registro fuente es SI.

--- Hemos visto la manera de introducir un dato en un registro. ¨Pero cómo hacemos para introducir un dato en memoria? Bien, para esto se utilizan las variables (que también existen en ensamblador) o bien, se indica una posición de memoria concreta, pasando de variables. Hay una tercera manera que es utilizar registros índice o base.
+ En el primer caso, es muy simple. Si queremos introducir el valor 70h en la variable X, basta con escribir MOV X,70h. Previamente la variable X la hemos definido y hemos definido también su tamaño: byte, palabra, doble palabra. Una vez que el compilador d, el código ejecutable, lo que antes era la variable X, ahora ser la posición de memoria ocupada por la variable. Es decir, que el usar variables es para darnos una gran comodidad a los programadores. Podríamos hacer un programa sin usar variables, indicando posiciones de memoria directamente, pero eso es ya mas parecido a código maquina puro que a ensamblador.
+ En el segundo caso, el de indicar la posición de memoria concreta, hay que tener en cuenta si esa posición de memoria la utilizamos como un byte o como una palabra. Esto es así ya que si por medio del programa queremos guardar un 5 en la posición de memoria 7654h (por ejemplo), el procesador no sabe si queremos guardar un byte o una palabra.
³ Para que no surja ningún tipo de lios, el lenguaje ensamblador cuenta ³ con ciertos convencionalismos para tratar estas transferencias a memoria. ³ Cuando queremos introducir un byte en una posición dada de memoria lo ³ hacemos con el siguiente formato: MOV BYTE PTR DS:[7654H],5 ³ ^^^^^^^^ ^^ ^ ^ ³ BYTE PTR indica que vamos a acceder a una posición de memoria de tipo BYTE. ³ ³ Caundo queremos introducir una palabra a partir de una posición de memoria ³ el formato queda como sigue: MOV WORD PTR DS:[7654H],5 ³ ^^^^^^^^ ^^ ^ ^ ³ WORD PTR indica que vamos a acceder a una posición de memoria de tipo WORD.
Tened en cuenta también que cuando se quiere acceder a una posición concreta de memoria sin pasar por una variable, se debe indicar entre corchetes, como en los ejemplos de arriba. Pero eso no es todo, se debe indicar un segmento, para que el procesador sepa a que zona de 64 ks de la memoria pertenece la posición dada entre los corchetes. En este caso indicamos el segmento DS (segmento de datos), que es lo usual. Aunque también podríamos haber seleccionado el segmento ES(segmento extra de datos) para así poder transferir algo fuera de nuestra zona de datos.
Obs,rvese la manera de indicar una dirección en dirección segmentada, no real. Primero se indica el segmento, luego dos puntos para separar, y luego entre corchetes el offset o desplazamiento dentro de ese segmento. Segmento:[desplazamiento] DS:[2626h], ES:[FFFFh], etc.
+ En el tercer caso nos valemos de un registro índice o base, el cual contiene la dirección de la posición de memoria que nos interesa, para acceder a dicha posición de memoria.

Un ejemplo: MOV BYTE PTR [DI],5 Obs,rvese que aquí no es necesario indicar el segmento al que nos referimos. Se coge por defecto el segmento DS. En definitiva, cuando accedemos a memoria a través de registros indice o base, no es necesario indicar el segmento. Mientras que si lo hacemos en forma directa, indicando la posición de memoria tal que [2635h], debemos indicar el segmento con el que vamos a tratar.
---------- Que lioooooooooooooooooooooooo... ¨verdad? He intentado ponerlo lo mas claro posible, con muchos ejemplos, y como se lo explicaría a una persona que tuviera a mi lado. Pasando de rollos teóricos de libros y demas parafernalia, pero si aún así os resulta lioso o complicado, no os preocupeis. Estoy aquí para re-explicar lo que haga falta. Y ademas cuando empecemos a hacer programillas, todo esto se ver muy claro en la pr ctica.
---------- Sigamos:
Veamos ahora cómo se codifica una instrucción en la que se hace acceso a memoria.
* MOV WORD PTR DS:[7654H],5 ---> Esta instrucción introduce el valor 5 a partir de la posición de memoria 7654h. Y digo a partir, ya que necesita dos posiciones de memoria para almacenarlo, ya que se trata de un valor inmediato de 16 bits (esto se determina al poner lo del WORD PTR). Con lo cual, la palabra con valor 5, queda almacenada en dos posiciones de memoria, la indicada [7654h] y la contigua [7655h]. Si tenemos en cuenta lo que hemos comentado antes acerca de cómo el 8086 almacena las datos de tipo palabra en memoria, sabremos de antemano que la posición [7654h] contendr el valor 05, y la posición [7655h] contendr el valor 00.
Veamos cómo se codifica esta instrucción:
MOV WORD PTR [7654H],5 ---> C7 06 54 76 05 00 Vemos que esta instrucción ha ocupado el m ximo posible (6 bytes). De tal forma que los 4 campos de instrucción están presentes. Vamos a estudiarla detenidamente: Lo primero que tenemos es el código de operación: C7. Este código indica una operación MOV sobre una dirección concreta ó desplazamiento, y con un valor num,rico de tipo palabra.
El 3§ y 4§ byte juntos forman el desplazamiento (tener en cuenta lo del tema del orden inverso en los bytes), y los bytes 5§ y 6§ juntos forman el valor inemdiato a introducir (tener en cuenta de nuevo lo del orden inverso).
Y nos queda el 2§ byte, que es el byte EA o de direccionamiento. ¨Que por que lo he dejado para el final? je. Porque llevo 2 o 3 horas intentando descubrir el por que de que sea 06. No me cuadra por ningún sitio, ya que este 6 indica que no hay desplazamiento, cuando sí lo hay.
A ver si para la próxima lección, consigo descifrar el misterio.
Un saludo. AESOFT....

Para cualquier duda o consulta, deja un mensaje a:
- Francisco Jesus Riquelme
- AESOFT (pseudónimo dentro del BBS Club).
Podr s encontrarme en:
BBS Club: (968) 201819/201262 14k4 x2
FidoNet 2:341/43.9 MasterNet 17:3468/301.3

þþþ BBS Club þþþ (968) 201819/201262 28k8 Apartado de Correos 2064, 30080 Murcia FidoNet 2:346/401 SubNet 93:3468/101 RANet MasterNet VirNet Internet ClubNet
Un saludo. AESOFT....

Para cualquier duda o consulta, deja un mensaje a:

- Francisco Jesus Riquelme
- AESOFT (pseudónimo dentro del BBS Club).
Podr s encontrarme en:
BBS Club: (968) 201819/201262 14k4 x2
FidoNet 2:341/43.9 MasterNet 17:3468/301.3

ASM POR AESOFT. (lección 6). ----------
- CODIFICACION DE LAS INSTRUCCIONES EN EL 8086 (continuación lección 5). - MODOS DE DIRECCIONAMIENTO EN EL 8086. ----------

- CODIFICACION DE LAS INSTRUCCIONES EN EL 8086 (continuación lección 5). ----------
En la lección 5 nos quedamos por descifrar el valor del byte EA en la siguiente instrucción: MOV WORD PTR DS:[7654H],5 ---> C7 06 54 76 05 00
Teníamos claro el primer byte (código de operación) que nunca ofrece problemas de interpretación. Y era obvio también el campo 'desplazamiento' (formado por los bytes 3 y 4), así como el campo 'valor inmediato' (formado por los bytes 5 y 6).
Pero ante el segundo byte (byte EA o de direccionamiento) surgía la duda. He estado probando con instrucciones parecidas para ver si se aclaraba el tema. Y como conclusión puedo decir que cuando se direcciona una posición de memoria (como destino) sin ayuda de ningún registro (SI, DI,BX+SI, etc), el byte EA tendr valor 06. Esto lo digo de modo empírico. No lo he leído en ningún sitio, ni nada por el estilo. Me baso en las pruebas que he estado haciendo para encontrar algún sentido a ese 06.
Sinceramente, no s, por el porque de esta 'peculiaridad'. Por que se me ocurriría poner este ejemplo! :-) Por ejemplo, si hubiera usado un registro junto con el desplazamiento en esa instrucción habría problema. Veamos cómo quedaría la cosa:
* MOV WORD PTR DS:[DI+7654H],5 ---> Introduce el valor 5 (tipo palabra) en la posición de memoria direccionada mediante [DI] + 7654H. Es pr cticamente igual que la instrucción anterior, excepto que en ,sta, hemos incluido el registro DI para acceder a la posición de memoria oportuna.
La codificación de la instrucción quedaría como sigue: MOV WORD PTR DS:[DI+7654H],5 ---> C7 85 54 76 05 00
Estudiemos el 'byte EA' de esta instrucción tan parecida a la anterior: 85h en binario queda como 10000101. El campo MOD con valor 10 indica que se usan 2 bytes para codificar el desplazamiento. Hasta ahora perfecto: 2 bytes para almacenar el desplazamiento 7654h. El campo REG con valor 000, ya que el valor a introducir no est en ningún registro, sino que es un dato inmediato (05) El campo R/M con valor 101, indicando que la dirección final viene dada por [DI] + desplazamiento.
Como podemos ver, la codificación de esta instrucción se ajusta a las tablas de codificación que vimos en la lección 5.
Veamos algún ejemplo mas, para que nos quede mas claro: * MOV BYTE PTR [DI+7],0AEH Esta instrucción deposita el valor 0AEH en la posición de memoria apuntada por DI+7, dentro del segmento direccionado por DS. Veamoslo mas detalladamente estudiando su codificación:
MOV BYTE PTR [DI+7],0AEH ---> C6 45 07 AE En primer lugar tenemos el código de operación C6, que le indica al procesador que se va a transferir un dato inmediato de tipo byte a una posición de memoria.
El tercer byte (campo 'desplazamiento') tiene un valor de 7, que es el que hemos indicado en la instrucción. Este valor es el que se sumar a DI para direccionar la posición de memoria deseada.

El cuarto byte (campo 'valor inmediato') tiene un valor de AE. Este valor es el que se depositar en la posición de memoria direccionada mediante DI+7.
Sólo nos queda estudiar el segundo byte (byte EA o modo de direccionamiento), el cual tiene un valor de 45 (en hexadecimal). Este 45, queda como 01000101 en binario.
Tenemos así que el campo MOD tiene un valor de 01. Si miramos en las tablas de codificación (ver lección 5), tenemos que 01 indica que se utiliza un byte para indicar desplazamiento. El campo REG tiene valor 000. Como no hay ningún registro involucrado en la instrucción, este campo no se tiene en cuenta. Por último, el campo R/M tiene como valor, 101. Esto indica que el desplazamiento final estar dado mediante [DI] + desplazamiento.
Como vemos, en esta instrucción est totalmente claro cada valor en cada uno de los campos del byte EA o 'modo de direccionamiento'.
Un último ejemplo:
* MOV WORD PTR [BX+SI+37H],AX Esta instrucción, introduce el valor del registro AX en la dirección de memoria indicada mediante el valor de la suma BX+SI+37H. Dentro del segmento DS, por supuesto. Siempre que no se especifique otro segmento, las transferencias a/desde memoria se har n sobre el segmento DS.
Veamos su codificación: MOV WORD PTR [BX+SI+37H],AX ---> 89 40 37 Simplemente 3 bytes para codificar esta instrucción que parece tan complicada. El primer byte (código de operación) tiene el valor 89 (hexadecimal). Este código 89 indica que se va a realizar una transferencia de datos de tipo palabra desde un registro (tamaño palabra) a un destino que puede ser (dependiendo del valor del byte EA): un registro de tamaño palabra o una dirección de memoria. Ya sabemos que el destino se trata de una dirección de memoria porque lo hemos indicado así al teclear la instrucción, pero el procesador no lo sabr hasta que descifre el valor del byte EA.
Este byte EA tiene un valor de 40h, que en binario queda como 01000000 El campo MOD tiene un valor de 01, que indica que se usa 1 byte para codificar el desplazamiento. Sólo es necesario 1 byte para codificar el valor 37h. El campo REG, que aquí tiene valor 000, indica que el registro fuente empleado es AX. El campo R/M con valor 000, indica que la dirección final se consigue mediante la suma: [BX] + [SI] + desplazamiento. Ver tablas en la lección 5. Como vemos aquí también est totalmente claro el valor del byte EA.
El tercer byte (campo 'desplazamiento') contiene el valor 37h como hemos indicado en la instrucción.
Espero que estos nuevos ejemplos hayan sido ilustrativos. En caso de que aún exista alguna duda acerca de la codificación de las instrucciones, levantad la mano :-)))

- MODOS DE DIRECCIONAMIENTO EN EL 8086. ----------

Como ya hemos visto, las instrucciones están codificadas con 0, 1 ó 2 operandos. Estos operandos son cada uno de los campos que puede componer una instrucción ('byte EA', 'desplazamiento' y 'valor inmediato'). Pues bien, las operaciones se realizan entre registros o registros y memoria. Nunca entre memoria y memoria. Si echamos un vistazo al esquema de codificación, nos daremos cuenta del porque: No hay dos campos para almacenar 2 posiciones diferentes de memoria, mientras que sí que se pueden indicar 2 registros diferentes (uno fuente y otro destino) en una misma instrucción.
Existen varios modos de direccionamiento. Esta variedad se ofrece para una mayor comodidad en la programación. Así por ejemplo se puede utilizar el modo directo, cuando se conoce la dirección de la posición de memoria que nos interese. El resto de modos se utilizar n cuando no vayamos a acceder a una dirección de memoria conocida, o cuando nos sea mas cómodo por cualquier motivo.
* Modos de direccionamiento:

Inmediato --------- El dato aparece directamente en la instrucción. Es decir, se indica explícitamente en la instrucción.
Ejemplo: MOV AL,0AEH ---> Almacena el valor AE en el registro de tipo palabra (AL).
Modo Registro ---------- Cuando en la instrucción se realizan transferencias entre registros del procesador.
Ejemplo: MOV AX,BX ---> Introduce el valor de BX en AX.
Directo absoluto a memoria ---------- Aparece en la instrucción de forma explícita la dirección a la cual se quiere acceder. Esto es, en la codificación de la instrucción podemos encontrar 2 bytes indicando la posición de memoria que se quiere direccionar.
Ejemplo: MOV CX,WORD PTR DS:[7777] ---> Introduce el valor almacenado en la posición de memoria 7777 (dentro del registro DS) en el registro de tipo palabra (CX).
Directo relativo a un registro base ---------- Accede a una posición de memoria con la ayuda de un registro base. Estos registros base como ya vimos son BX y BP. La dirección final de la posición de memoria se indicar de la forma: Dirección final = [Registro_Base] + desplazamiento. Donde Registro_Base es o bien BX o bien BP.
Ejemplo: MOV AX,BYTE PTR [BX+7] ---> Introduce en AX el valor contenido en la posición de memoria direccionada mediante BX+7.
Directo relativo a un registro índice (Indexado) ---------- La dirección se obtiene sumando un desplazamiento (que se indicar en la instrucción) al contenido de un registro índice. Registros índice como ya vimos son: SI y DI. Donde SI se suele utilizar como registro índice fuente. 'Indice fuente' en ingl,s = 'Source Index' = SI. Obviamente, DI se utilizar como registro índice destino. 'Indice destino' en ingl,s = 'Destine Index' = DI.
La dirección final de la posición de memoria se indicar de la forma: Dirección final = [Registro_Indice] + desplazamiento. Donde Registro_Indice es o bien SI o bien DI.
Ejemplo: MOV AX,WORD PTR [SI+7] ---> Introduce en AX el valor contenido en la posición de memoria direccionada mediante SI+7.
En realidad, excepto cuando se utilizan los dos registros fuentes en una misma operación (al copiar cadenas de caracteres p.e.), en cuyo caso hay que asignar a SI la dirección de la cadena origen, y a DI hay que asignarle la dirección de la cadena destino... Excepto en este tipo de operaciones (como digo), podemos hacer uso de los registros SI y DI indistintamente al trabajar con direccionamientos. Es decir, que las dos instrucciones siguientes son correctas, y realizan exactamente la misma tarea (siempre que SI y DI tengan el mismo valor): MOV AX,WORD PTR [SI+7] MOV AX,WORD PTR [DI+7]
Modos indirectos de direccionamiento ---------- Si en los dos últimos modos de direccionamiento no se especifica desplazamiento, entonces hablamos de: Modo de direccionamiento indirecto por registro base, Modo de direccionamiento indirecto por registro índice, respectivamente.
Ejemplo: MOV AX,BYTE PTR [SI] ---> Introduce en AX el contenido de la posición de memoria direccionada mediante SI.
Indexado mediante una base ---------- La posición de memoria seleccionada se direcciona aquí mediante cuatro configuraciones posibles. Un registro BX o BP contiene la base y un registro SI o DI contiene el desplazamiento. Ademas, puede existir un desplazamiento opcional indicado mediante un valor num,rico.
La dirección final de la posición de memoria se indicar de la forma: Dirección final = [Registro_Base] + [Registro_Indice] + desplazamiento. Donde Registro_Base es o bien BX o bien BP. Donde Registro_Indice es o bien SI o bien DI. Tenemos así las cuatro configuraciones posibles mencionadas:
- [BX] + [SI] + desplazamiento - [BX] + [DI] + desplazamiento - [BP] + [SI] + desplazamiento - [BP] + [DI] + desplazamiento

Ejemplo: MOV BYTE PTR [BX+SI],AL ---> Introduce el valor del registro AL en la posición de memoria direccionada mediante [BX] + [SI] + desplaz.

En todos los modos de direccionamiento, excepto en los dos primeros, se puede indicar un segmento diferente a DS para realizar las operaciones con la memoria. De esta forma, podemos manejar todo el Megabyte de memoria sin necesidad de modificar de valor el registro DS (registro de segmento de datos). Así, podemos utilizar el registro ES cuando queramos acceder a posiciones de memoria que est,n fuera de nuestro segmento de datos.
Ejemplo: MOV AX,WORD PTR ES:[BX+7] ---> Introduce en AX el valor contenido en la posición de memoria direccionada mediante BX+7, dentro del segmento indicado por ES.

ASM POR AESOFT. (lección 7). ---------- - DUDAS DE LECCIONES ANTERIORES - MAS INFORMACION Y EJEMPLOS ACERCA DEL REGISTRO DE ESTADO (FLAGS) (ampliación de la lección 1). - REPASO AL TEMA DE LOS SEGMENTOS (ampliación de la lección 2). - CONJUNTO DE INSTRUCCIONES DEL 8086 (I). ----------
Hola de nuevo, aplicados alumnos. :-) En esta lección, aparte de repasar algunos temas anteriores, a petición de algunos lectores, empezaremos a estudiar las instrucciones con que contamos en el 8086. Las dividiremos en grupos, y veremos sus características y funcionamiento.
Que disfruteis la lección de hoy, que mi trabajo me ha costado. :-)

- DUDAS DE LECCIONES ANTERIORES ---------- En este apartado os dejo un mensaje que me parece de utilidad para el curso. En este mensaje se habla sobre la codificación de ciertas instrucciones, y como os decía me parece útil para todos los seguidores del curso. Ahí va:
--- Inicio del mensaje
FJR> Veamos cómo se codifica esta instrucción:
FJR> MOV AX,5 ---> B8 05 00 (Código m quina, siempre
FJR> en hexadecimal).
FJR> En primer lugar tenemos el primer byte que contiene
FJR> el código de
FJR> operación (B8).
FJR> Debido a que este código de operación(B8) tiene
FJR> implícita la utilización
FJR> del registro AX como destino, no es necesario el
FJR> byte EA ó byte de
FJR> direccionamiento, que sí sería necesario para
FJR> transferencias con otros
FJR> registros.
DT> Osea que cada mov que hagas tiene un 'código' distinto si se hace a
DT> ax, a cx, etc... ? y se ha seguido algún orden lógico a la hora de
DT> asignarle números a las intrucciones?, osea, ¨por que b8 -> mov ax,?
En efecto, ese B8 tiene su razón de ser. En primer lugar, todas las operaciones del tipo MOV registro,dato_inmediato tienen un código de operación cuyo primer dígito hexadecimal es B. Hay 16 códigos de operación diferentes (uno para cada registro, como tú muy bien observabas) para el tipo de operación MOV registro,dato_inmediato.
Por supuesto estos código siguen un orden:

B0 ---> MOV AL,dato_inmediato_tamaño_byte B1 ---> MOV CL,dato_inmediato_tamaño_byte B2 ---> MOV DL,dato_inmediato_tamaño_byte B3 ---> MOV BL,dato_inmediato_tamaño_byte
B4 ---> MOV AH,dato_inmediato_tamaño_byte B5 ---> MOV CH,dato_inmediato_tamaño_byte B6 ---> MOV DH,dato_inmediato_tamaño_byte B7 ---> MOV BH,dato_inmediato_tamaño_byte
B8 ---> MOV AX,dato_inmediato_tamaño_word B9 ---> MOV CX,dato_inmediato_tamaño_word BA ---> MOV DX,dato_inmediato_tamaño_word BB ---> MOV BX,dato_inmediato_tamaño_word
BC ---> MOV SP,dato_inmediato_tamaño_word BD ---> MOV BP,dato_inmediato_tamaño_word BE ---> MOV SI,dato_inmediato_tamaño_word BF ---> MOV DI,dato_inmediato_tamaño_word
Podr s observar que el orden de los registros no es AL, BL, CL, DL... Sino AL, CL, DL, BL. Lo mismo para los registros de 8 bits de mayor peso (AH, CH, DH, BH), Y para los registros de 16 bits (AX, CX, DX, BX, SP, BP, SI, DI).
Un saludo ---------- Francisco Jesus Riquelme ---------- FidoNet 2:341/43.9 MasterNet 17:3468/301.3
--- Fin del mensaje
Espero que os haya parecido interesante.

- MAS INFORMACION Y EJEMPLOS ACERCA DEL REGISTRO DE ESTADO (FLAGS). (ampliación de la lección 1). ---------- Ya vimos algo acerca del registro de estado (FLAGS) en la lección 1. En esta lección, tratar, de desarrollar un poco mas para los que no lo entendieron del todo en esa primera lección.
El registro de flags ó palabra de estado est compuesto por una serie de banderas (flags en ingl,s) que no son mas que simples bits o dígitos binarios, los cuales pueden tener un valor de uno (bit activo) o un valor de cero (bit no activo). Cada uno de estos bits mantiene una información determinada. Ya dimos en la primera lección una relación de estos bits de estado ó flags, agrupados en categorías según su función.
Veamos mas detenidamente uno de estos grupos de flags ó banderas: * Flags de estado * Estos flags están íntimamente ligados a las operaciones aritm,ticas, que son enumeradas y detalladas mas abajo. Estos flags nos ofrecen información acerca del resultado de la última operación efectuada.
Es decir, si tras realizar una multiplicación se ha producido desbordamiento, el flag ó bit de estado Of (flag de overflow) se pondr con valor 1.
Si fruto de otra operación, como una resta obtenemos un número negativo, el flag Sf (flag de signo), se pondr a 1, indicando que el resultado de la operación ha dado un número negativo.
Si tras una operación, como puede ser una resta, el resultado obtenido es cero, se activar el flag Zf (flag Cero. Zero en ingl,s).
Una operación puede afectar a un sólo flag, a ninguno, o a varios. Es decir, dependiendo del tipo de instrucción de que se trate, el procesador tendr que actualizar un número determinado de flags. Por ejemplo, las instrucciones de salto, tanto condicionales como incondicionales, no actualizan ningún flag. La instrucción MOV tampoco actualiza ningún flag. Mientras que las instrucciones aritm,ticas actualizan muchos de los flags, para así indicar el estado de la operación. Tomemos las instrucciones SUB, ADD, ADC, etc. Todas estas instrucciones afectan a los siguientes flags: Of, Sf, Zf, Af, Pf, Cf. En realidad, la mayoría de las instrucciones aritm,ticas afectan a esos flags.
De esta forma, tras realizar cada operación, mediante estos flags sabremos si el resultado es negativo, si es cero, si se ha producido overflow, etc.

Como hemos visto, las operaciones modifican los flags para indicar el estado de tal operación. Pero esa no es la única forma de que los flags cambi,n su valor. En ensamblador disponemos de instrucciones para modificar el valor de un flag determinado.
- CLC (Clear Cf. Borrar ó poner a cero el flag de acarreo). Sintaxis: CLC.
- STC (Set Cf. Activar ó poner a uno el flag de acarreo). Sintaxis: STC.
- CLI ((Clear If. Borrar ó poner a cero el flag de interrupción). Sintaxis: CLI. Esta instrucción la usamos cuando queremos que ninguna interrupción enmascarable nos interrumpa el proceso en el que estamos. (Esto ya lo vimos en la lección 4).
- STI (Set If. Activar ó poner a uno el flag de interrupción). Sintaxis: STI. Mediante esta instrucción habilitamos de nuevo las interrupciones. (Visto en la lección 4).
- CLD (Clear Df. Borrar ó poner a cero el flag Df). Sintaxis: CLD. Esta instrucción se usa cuando se est trabajando con hileras ó cadenas de caracteres. Ya la estudiaremos entonces.
- STD (Set Df. Activar ó poner a uno el flag Df). Sintaxis: STD. Esta instrucción se usa cuando se est trabajando con hileras ó cadenas de caracteres. Ya la estudiaremos entonces.
El resto de los flags no puede modificarlos el programador mediante las instrucciones CLx y STx. Pero siempre hay otros m,todos.
~~~ A ver si alguien me dice cómo podemos modificar el flag Tf, por ejemplo. Os dar, una pista: ¨Recordais las instrucciones PUSHF y POPF? Espero vuestros mensajes. Si a nadie se le ocurre, ya dejar, yo la solución en una próxima lección.
- REPASO AL TEMA DE LOS SEGMENTOS (ampliación de la lección 2). ---------- Debido a que no quedó claro para todos el tema de los segmentos, intentar, complementar la información que ya dí acerca del tema de la segmentación con una exposición mas coloquial de dicho tema.
Tenemos 1 Mbyte de memoria para nuestro uso. 1 Mbyte son 1024 Ks. Y 1024 Ks son a su vez, 1048576 bytes. O sea, que podemos manejar 1048576 bytes de memoria desde nuestro programa. Ahora debemos tener en cuenta que los registros del 8086 son de 16 bits, es decir, tienen capacidad para albergar 16 bits diferentes. Cada uno de estos bits puede tener un valor de 1 o de 0, independientemente del valor que tengan los bits contiguos. Por tanto, tenemos 2^16 combinaciones diferentes para ese registro, es decir, el registro puede tener 2^16 valores diferentes, o lo que es lo mismo, el registro puede representar 65536 valores diferentes.
Hemos dicho que los registros en el 8086 son de tamaño de 16 bits (como mucho). Entonces, en teoría, sólo podríamos indicar 65536 posiciones de memoria. Pero sólo en teoría, ya que como vismos en la lección 2, se puede acceder a todas las posiciones de ese Mbyte usando registros de 16 bits.
Usamos entonces 2 registros de 16 bits. Por medio del primero, seleccionamos el trozo (segmento) de ese Mbyte donde se encuentra la dirección que nos interesa. Por medio del segundo registro, indicamos cu l es la dirección que nos interesa dentro de ese trozo ó segmento.
El primer registro se llamar registro de segmento, y puede ser uno de los que ya conocemos: CS, DS, ES, SS.
El segundo registro es lo que se llama offset ó desplazamiento dentro de ese segmento ó trozo de Mbyte.
Ya vimos en la lección 2 como se formaba la dirección final a partir de estos dos registros ó direccionamiento segmentado.
El valor depositado en el registro de segmento, se multiplica por 16 a la hora de buscar el segmento (trozo de Mbyte actual), de esta forma se puede acceder a todo el Mbyte, ya que 65536*16 = 1048576 (1 Mbyte). Esto es algo que hace internamente el procesador con registros especiales para este propósito. ¨Pero que pasa con los 15 bytes que quedan
~~~

entre una dirección y otra? Para eso tenemos el segundo registro: Una vez que ya se sabe dónde comienza el segmento, es decir, una vez que ya sabe el procesador con que trozo de Mbyte va a trabajar a continuación, lo que hace es sumar al principo de éste, el valor depositado en el segundo registro (offset ó desplazamiento). De esta forma, se produce el acceso a la dirección deseada.
Si a pesar de esta explicación alguno no lo entiende, que sea mas concreto, y me diga exactamente que es lo que no entiende.

- CONJUNTO DE INSTRUCCIONES DEL 8086 (I). ---------- En este apartado vamos a estudiar las operaciones fundamentales para empezar a programar en ensamblador. Una lista completa del conjunto de instrucciones del 8086 se dar mas adelante, en otra lección. Por ahora, tendremos suficiente con estudiar las instrucciones mas representativas dentro de cada grupo:
--- Movimiento de datos. Las instrucciones pertenecientes a este grupo, tienen como objetivo: - Actualizar un registro con un valor. - Copiar la información de un registro a una posición de memoria. - Copiar la información de una posición de memoria a un registro. - Mover la información de un registro a otro. - Intercambiar la información entre dos registros. En este grupo (Movimiento de datos) podíamos incluir varias de las instrucciones que vamos a ver en grupos sucesivos, como por ejemplo cuando hablemos de las instrucciones para el manejo de hileras (cadenas de caracteres), entre otras, estudiaremos las instrucciones para transferir hileras, que bien se podían incluir en este grupo debido a su naturaleza de movimiento de datos. De cualquier modo, se enmarquen en un grupo o en otro, quedará suficientemente claro durante su exposición sus características y finalidad.
Como vimos en lecciones anteriores, la instrucción principal usada en movimientos de datos es la instrucción MOV. Con la instrucción MOV, podemos: - Mover el contenido de un registro fuente o una posición de memoria a un registro destino. O bien, mover el contenido de un registro a una posición de memoria.
Su sintaxis es como sabemos: MOV destino,fuente. Ejemplo: MOV BX,SI ---> Mueve el contenido del registro SI al registro BX.
- Mover un dato (valor inmediato) a un registro o posición de memoria. Sintaxis: MOV destino,valor. Ejemplo: MOV BYTE PTR [SI],7 ---> Introduce el número 7 en la posición de memoria direccionada por SI. Ejemplo: MOV AX,25 ---> Mueve el número 25 al registro AX.
Aparte de la instrucción tenemos varias mas para realizar movimientos de datos, como pueden ser:
- XCHG Intercambia el contenido de dos registros, o bien el contenido de un registro y el de una posición de memoria. Sintaxis: XCHG registro,registro/memoria XCHG viene del inglés EXCHANGE (Cambio). Por tanto es un cambio entre los dos valores dados tras el código de operación de la instrucción. Ejemplo: XCHG AX,WORD PTR [BX] ---> Tras ejecutarse esta instrucción, AX contendrá el valor que hubiera en la posición de memoria direccionada por BX, y viceversa. Ejemplo: CX,DX ---> Intercambia los contenidos de CX y DX.
- Todas las relacionadas con la pila: PUSH, POP, PUSHF, POPF. Las cuales las estudiamos en la lección 4. Ejemplo: PUSH AX ---> Introduce en la cima de la pila, el valor contenido en AX. - Ademas de las instrucciones enumeradas, y como ya hemos dicho arriba, en este grupo existen varias instrucciones mas que veremos mas adelante, conforme sea necesaria su utilización.
--- Transferencia de control. Son un conjunto de instrucciones que permiten al programador romper el flujo secuencial en un programa. Su función consiste en añadir un valor de desplazamiento al puntero de instrucción (IP), y en algunos casos variar también

el valor de CS. La finalidad está en permitir ejecutar trozos de código si se cumple una condición, ejecutar trozos de código repetidas veces (bucle), ejecutar trozos de códigos desde diferentes puntos del programa (procedimientos), etc.
Son 5 los tipos de instrucciones de transferencia de control.
Podemos clasificar las instrucciones de transferencia de control en los siguientes subgrupos: - Saltos incondicionales (JMP). - Bucles (LOOP). - Saltos condicionales (Jnnn). Donde nnn indica la condición. - Llamadas a procedimientos (CALL). - Llamadas a interrupciones o funciones (INT).
Vamos a desarrollar cada uno de estos grupos:
... - JMP (salto incondicional). Provoca la transferencia de control a la dirección que se especifica a continuación del código de operación. Su sintaxis es: JMP dirección Donde dirección puede ser una etiqueta (La etiqueta es un nombre que asociamos a una línea de instrucción en ensamblador. Es como una especie de apuntador a esa línea), o una dirección contenida en un registro o variable. Los saltos pueden ser directos o indirectos. Así como también pueden realizarse saltos dentro del mismo segmento (NEAR), y saltos intersegmento (FAR).
Directo y NEAR: JMP etiqueta ---> Salto a la dirección etiqueta. Etiqueta puede encontrarse antes o despues de la instrucción de salto. Es decir, los saltos se pueden realizar hacia adelante o hacia detrás de la instrucción en curso. Incluso es posible tener una instrucción de salto a esa misma instrucción. Es decir:
;*** porción de código. Etiqueta: JMP etiqueta
;*** fin de la porción de código.
Lo cual nos daría como resultado un bloqueo del ordenador, ya que el control no saldría de esa línea. Sería algo así como un bucle sin fin.
El salto directo y NEAR, es el salto mas común. Raramente se utilizan los que aparecen a continuación.
Indirecto y NEAR: JMP [BX] ---> Salto a la dirección indicada por la variable direccionada mediante BX. Es un salto dentro del mismo segmento.
Indirecto y FAR: JMP FAR PTR [BX] ---> Salto tipo FAR (a otro segmento) donde BX contiene la dirección de comienzo de una doble palabra con los nuevos valores de IP y CS.
Ejemplo de salto directo y NEAR:
;*** Inicio: JMP Sanbit MOV cx,7 Sanbit: MOV cx,6 ;***
Al ejecutar este trozo de código desde la etiqueta Inicio, la instrucción (MOV cx,7) nunca se ejecutar . Por tanto, al final de este trozo de código, la variable CX tendr valor 6.
Obsérvese que las etiquetas pueden tomar cualquier nombre, siempre que éste no pertenezca al lenguaje ensamblador. Al final de la etiqueta debe aparecer el carécter ':' (los dos puntos), el cual le indica al ensamblador que es una etiqueta de tipo NEAR, es decir, que va a ser utilizada para saltos dentro del mismo segmento. Si no apareciesen los dos puntos ':', se consideraría de tipo FAR, utilizada para saltos entre segmentos. Lo mas común es utilizar etiquetas tipo NEAR. Para saltos intersegmentos se suelen utilizar otro método diferente al salto a una etiqueta tipo FAR. Yo nunca he usado una etiqueta tipo FAR en los años que llevo con el ensamblador, y seguramente vosotros tampoco la useis nunca.
... - LOOP (Bucle) Esta instrucción sirve para ejecutar un trozo de código un número de veces dado (indicado mediante el registro CX). En cada iteración del bucle se decrementa el valor del registro CX. El bucle finalizar cuando CX tenga valor 0, es decir, cuando se hayan producido tantas iteraciones como indicaba CX antes de entrar al bucle.
Veamos un ejemplo:
;*** MOV CX,7 INICIO_BUCLE: ADD WORD PTR [BX],CX INC BX LOOP INICIO_BUCLE MOV SI,BX ;***

En el ejemplo que estamos tratando, tenemos un bucle que se va a repetir 7 veces. En cada una de estas iteraciones se van a realizar dos operaciones aritméticas (echar un vistazo al apartado de operaciones aritméticas, para saber que hace el cuerpo del bucle). Tras realizar las dos operaciones, llegamos a la instrucción LOOP inicio_bucle. Esta instrucción primero comprueba si CX vale 0, en caso afirmativo, no hace nada y sigue el flujo de control por la siguiente instrucción (en este caso: MOV SI,BX). En caso de que CX tenga un valor distinto de 0, se decrementa su valor, y se bifurca a la dirección inicio_bucle. O sea, que se realiza la siguiente iteración.
Del mismo modo que el utilizar variables nos evita tener que indicar posiciones de memoria concretas del modo: [2346h], [7283h], etc, siendo infinítamente mas cómodo usar nombres como: coordenada_superior, valor_total, posicion_cursor, modo, etc... Del mismo modo, como os decía, usar etiquetas es la solución que nos ofrece el ensamblador para poder dirigirnos a posiciones de memoria en nuestros saltos, bucles, etc. También se pueden usar las etiquetas para indicar dónde empiezan determinadas estructuras de datos.
... - Saltos condicionales (Jnnn). Los saltos condicionales se usan en ensamblador para ejecutar trozos de código dependiendo del valor de determinado registro o variable. Llegamos a este punto que para realizar un salto condicional, antes hemos de hacer una comparación. Aunque se pueden realizar saltos condicionales sin antes haber hecho una comparación correspondiente, lo usual es hacer la comparación.
Por tanto, antes de seguir con la los saltos condicionales, tenemos que saber cómo se realizan las comparaciones en ensamblador, y que finalidad tiene el que tras cada comparación haya un salto condicional.
* COMPARACIONES * Las comparaciones están íntimamente relacionadas con los saltos condicionales. Es mas, es raro el programa ensamblador en el que se encuentre una comparación y acto seguido no haya un salto condicional. La sintaxis de la instrucción de comparación es:
CMP registro,registro CMP registro,memoria CMP memoria,registro CMP registro,valor CMP valor,registro
El orden de los operandos a comparar es muy importante: No es lo mismo la instrucción CMP AX,7 que CMP 7,AX. No es lo mismo, debido a que en la comparación obetenemos mas información que un simple 'son iguales' o 'son diferentes'. Fruto de una comparación sabemos que operando es el mayor.
Usaremos una de las 5 sintaxis de arriba dependiendo de lo que vamos a comparar. Si queremos comparar 2 registros, por ejemplo AX con CX, la instrucción apropiada ser CMP AX,CX.
Los datos a comparar deben ser del mismo tamaño. Es decir, se comparar un dato de tipo byte con otro de tipo byte; Un dato de tipo palabra con otro dato de tipo palabra. Pero nunca se comparar un dato de tipo byte con otro de tipo palabra. Ejemplo de mala utilización de CMP: CMP AX,CL ---> No podemos comparar un dato de tipo palabra (AX) con un dato de tipo byte (CL).
Hemos visto que íntimamente ligado a los saltos condicionales están las instrucciones de comparación. Pues bien, el 'medio de comunicación' (por decirlo de alguna manera) entre una comparación y el salto condicional asociado, son las banderas de estado (FLAGS). Para aclarar esto, veamos cómo actúa una instrucción de comparación: Lo que hace la instrucción de comparación es restar al primer operando el segundo, pero eso lo hace mediante unos registros internos del procesador, a los que no tiene acceso el programador. De esta forma, los operandos usados por el programador quedan inalterados. Al realizar esta resta, se actualiza el registro de estado (FLAGS). Es decir, si fruto de la comparación, los dos datos eran iguales, la bandera o flag Zf tendrá valor activo, indicando que fruto de esa resta interna que ha hecho el procesador el resultado

es un cero. Es decir, los datos son iguales. Cuando un dato es menor que otro, son otros flags los que se activan, como el flag Cf (flag de acarreo o Carry). Al principio de la lección aparece mas desarrollado todo lo relacionado con los FLAGS.
Estudiemos mas profundamente el tema de los saltos condicionales: Todos los saltos condicionales deben estar dentro del rango (+127, -128) bytes. Es decir, que sólo se pueden saltar 127 bytes hacia adelante y 128 bytes hacia detrás dentro del código del programa. Si sumamos esos 127 bytes y los otros 128, tenemos un valor de 255. Para los que no les suene ese valor, deciros que es el mayor número que puede contener un dato de tipo byte.
Es decir, que se reserva un byte para indicar lo grande que va a ser el salto. Como el salto puede ser hacia adelante o hacia detrás, hay que dividir ese 255 en la mitad (mas o menos) para los valores positivos (saltos hacia adelante) y otra mitad para los negativos (saltos hacia detrás).
¨Que hacer cuando se quiere realizar un salto condicional mayor que esos 127/128 bytes? Muy sencillo: Un salto condicional a un salto incondicional.
También es útil conocer que existen saltos condicionales empleados cuando se comparan datos con signo, y los saltos condicionales empleados en comparaciones de datos sin signo.
Veamos los posibles saltos condicionales que podemos encontrar en el 8086:
* Saltos basados en datos sin signo:
Instrucción Efecto Flag comprobados ----------
JE/JZ (salta si igual) Zf=1 JNE/JNZ (salta si no igual) Zf=0 JA/JNBE (salta si superior) Cf=0 y Zf=0 JAE/JNB (salta si superior o igual) Cf=0 JB/JNAE (salta si inferior) CF=1 JBE/JNA (salta si inferior o igual) CF=1 ó Zf=1
* Saltos basados en datos con signo:
Instrucción Efecto Flags comprobados ----------
JE/JZ (salta si igual) Zf=1 JNE/JNZ (salta si no igual) Zf=0 JG/JNLE (salta si mayor) Zf=0 y Sf=Of JGE/JNL (salta si mayor o igual) Sf=Of JL/JNGE (salta si menor) Sf<>Of JLE/JNG (salta si menor o igual) ZF=1 ó Zf<>Of
Ademas de estos saltos encontramos una serie de saltos condicionales basados en comprobaciones aritméticas especiales:
Instrucción Efecto Flags comprobados ----------
JS (salta si negativo) Sf=1 JNS (salta si no negativo) Sf=0 JC (salta si se ha producido acarreo) Cf=1 JNC (salta si no se ha producido acarreo) Cf=0 JO (salta si se ha producido *overflow*) Of=1 JNO (salta si no se ha producido overflow) Of=0 JP/JPE (salta si *paridad par*) Pf=1 JNP/JPO (salta si *paridad impar*) Pf=0 JCX (salta si CX=0) CX=0 (registro CX=0)
overflow Overflow es lo mismo que desbordamiento, y se produce cuando tras una operación aritmética, el resultado es demasiado grande para que quepa en su destino. Al producirse overflow, se activa el flag Of.
paridad par , *paridad impar* La paridad indica el número de unos (1) en un registro o variable. Paridad par indica que ese registro tiene un número par de unos. Paridad impar indica que el registro tiene un número impar de unos.
Al realizar cada operación aritmética, el procesador comprueba el número de unos del resultado. Si ese número de unos es par (paridad par), activa el flag Pf. Si es impar, lo pone a 0.
Veamos la equivalencia entre las sentencias if..then de los lenguajes de alto nivel, y las construcciones CMP..Jnnn.
El equivalente a la sentencia: 'If modo=5 then fondo=7', vendría dado en ensamblador por la construcción:
;***

CMP modo,5 jnz no_fon mov fondo,7
no_fon:
;***
Veamos otro ejemplo: El equivalente a: 'If modo=5 then fondo=7 else fondo=6', vendría dado en ensamblador por:
;***
CMP modo,5 jnz no_fon mov fondo,7 jmp short fin_fon ;** a continuación se explica lo de 'jmp short'
no_fon: mov fondo,6
fin_fon:
;***
jmp short se utiliza cuando se quiere hacer un salto incondicional a una posición de memoria que está dentro del rango (-127 , +128). Es decir, que sobra con un byte para indicar el desplazamiento. de esta forma, nos ahorramos uno de los dos bytes que serían necesarios en caso del salto incondicional normal.
El salto incondicional normal (JMP) necesita dos bytes para poder especificar cualquier dirección dentro del segmento actual. Añadiéndole la palabra 'short', como hemos visto, hacemos que sólo necesite un byte para especificar la nueva dirección donde pasar el control.
Otro ejemplo: El equivalente de 'If modo <> 24 then fondo=modo' quedaría en ensamblador de la siguiente manera:
;*** suponemos las variables (fondo y modo) de tipo byte.
CMP modo,24 jz fin_fon mov al,modo mov fondo,al
fin_fon:
;***
Un último ejemplo: El equivalente de 'If modo < 23 then modo=23' quedaría en ensamblador de la siguiente manera:
;***
CMP modo,23 jnb fin_fon mov modo,23
fin_fon:
;***

... - Llamadas a procedimientos (CALL). Al igual que en los lenguajes de alto nivel, en ensamblador tenemos los llamdos procedimientos, trozos de código que van a ser usados en distintas partes del programa. Los cuales nos permiten tener un código mas legible, mas estructurado. El formato de un procedimiento en ensamblador es tal como sigue:
Tomemos como ejemplo un procedimiento llamado inicializacion.
inicializacion PROC
. . Cuerpo del procedimiento. . .
RET inicializacion ENDP
Cuando el procedimiento va a ser llamado desde otro segmento, se dice que es un procedimiento tipo FAR. Y se declara así:
inicializacion PROC FAR
;. ;. Cuerpo del procedimiento. ;. ;.
RET inicializacion ENDP
Cuando el procedimiento se usa sólo en el segmento donde se ha declarado, se denomina procedimiento NEAR. En este caso no es necesario indicar que se trata de NEAR. Es decir, que si no se especifica que es FAR, se supone que es NEAR.
O sea, que los dos formatos siguientes, son equivalentes:
inicializacion PROC ; Cuerpo del procedimiento. RET inicializacion ENDP

;****
inicializacion PROC NEAR ; Cuerpo del procedimiento. RET inicializacion ENDP
Para llamar a un procedimiento y transferirle de este modo el control, usamos la instrucción:
CALL nombre_procedimiento. En caso del ejemplo anterior, sería: CALL inicializacion.
Se retorna del procedimiento mediante la instrucción RET (Retorno de procedimiento).
Existen dos tipos de llamadas a procedimientos: * Llamadas directas: Mediante la instrucción CALL nombre_procedimiento. Donde nombre_procedimiento es el nombre que se le ha dado al procedimiento en cuestión. * Llamadas indirectas: Aquí no se especifica el nombre del procedimiento en cuestión, sino la dirección que contiene la dirección de comienzo del procedimiento que se quiere llamar. Este método se suele usar mucho en programación de utilidades residentes, cuando se llama a una interrupción parcheada (ya veremos todo esto próximamente). En este tipo de llamada, en función de que la llamada sea de tipo NEAR o FAR, las posiciones de memoria donde tengamos almacenada la dirección a la que queremos llamar serán de tipo WORD (palabra) ó DWORD (doble palabra).
Pero bueno, por ahora tenemos suficiente con las llamadas directas a procedimientos.
... - Llamadas a Interrupciones o funciones (INT). Ya vimos en lecciones anteriores el funcionamiento de las interrupciones. Vimos que podían ser de tipo hardware, y de tipo software. Pues bien, aquí las que nos interesan son las de tipo software. Que son ni mas ni menos que llamadas a procedimientos o subrutinas que se encuentran en la ROM del ordenador, y por otra parte también están las funciones del DOS (sistema operativo) entre otras. Es decir, hay ciertas funciones de muy bajo nivel, como acceso a discos, teclado, etc, que vienen ya programadas en la ROM del ordenador, para así mantener compatibilidad con el resto de PC's, y por otra parte, ayudar al usuario en la programación. También el sistema operativo ofrece al programador una serie de funciones para manejo de ficheros, memoria, etc.
Pues bien, la manera de utilizar estas funciones (pasarles el control), es a través de la instrucciín INT. Su sintaxis es la siguiente: INT numero_interrupcion. Donde numero_interrupcion es un número del 0 al 255.
Por ejemplo, para acceder al driver de vídeo, se usa la interrupción 10h. INT 10H ---> Provocaría una llamada a la interrupción 10h (16 en decimal).
Para acceder a las funciones del DOS, tenemos la interrupción 21h INT 21H ---> Provocaría una llamada a la interrupción 10h (16 en decimal).
Estas interrupciones software se dividen en funciones, y éstas a su vez en subfunciones. Para acceder a cada función/subfunción de una interrupción software, existe una convención de llamada. Es decir, para acceder a una determinada función/subfunción, hay que introducir en unos registros determinados un valor adecuado.
Por ejemplo, para crear un fichero, accedemos a la función 3Ch de la interrupción 21h. La llamada se realiza así en ensamblador:
;******
MOV AH,3Ch ;Seleccionamos función INT 21H ;pasamos el control a la función.
;*****
Otro ejemplo: para leer un carácter desde el teclado, llamamos a la función 00h de la interrupción 16h.
La llamada se realiza así en ensamblador:
;******
MOV AH,00h ;Seleccionamos función INT 16H ;pasamos el control a la función.
;*****
Hay dos manuales de bolsillo que son prácticamente imprescindibles para un programador en ensamblador. Estos libros son:

- Funciones del Ms-Dos (RAY DUNCAN / ANAYA MULTIMEDIA). - La Rom Bios de IBM (RAY DUNCAN / ANAYA MULTIMEDIA).
Contienen una gran información acerca de las funciones del DOS y de la ROM.
De todas formas, para el que no los pueda o quiera comprar (1000 pelas cada uno, mas o menos), próximamente daré una relación de las interrupciones del 8086, junto con información similar a la que viene en estos dos manuales.

--- Instrucciones aritméticas.
(En un principio sólo trabajaremos con números codificados en binario puro. Es decir, números sin signo.)
A diferencia de los lenguajes de alto nivel, en los que existen multitud de instrucciones aritméticas, en ensamblador del 8086 contamos sólo con unas pocas instrucciones básicas fundamentales, como son la suma, la resta, el producto, la división, y poco mas.
- ADD (Suma en el 8086). Realiza la suma entre dos operandos dados. Estos operandos deben ser del mismo tamaño. Sintaxis: ADD operando1,operando2. Se realiza la suma de los dos operandos, y se deposita en operando1. Tened en cuenta que puede producirse desbordamiento. Tomemos el caso (ADD AX,BX) cuando AX=0F000H y BX=3333H. Al realizarse la suma, se produce overflow (desbordamiento), quedando en AX tras la ejecución, el siguiente valor: 2333H. Con la correspondiente pérdida del dígito mas significativo. Esta situación se indica en el registro de estado (FLAGS) activando el flag de overflow (Of).
Otro ejemplo: ADD CX,WORD PTR [BX] ---> Suma a CX el valor contenido en la posición de memoria direccionada mediante BX.
Otro mas: ADD BYTE PTR [SI],7 ---> Introduce el valor 7 en la posición de memoria direccionada por SI.
Otro: ADD variable1,2345h ---> Suma a la variable1 (que hemos tenido que definir de tipo palabra) el valor 2345h (tipo palabra).
- SUB (Resta en el 8086). Realiza la resta entre dos operandos dados. Estos operandos deben ser del mismo tamaño. Sintaxis: SUB operando1,operando2. Resta del primer operando el segundo. Aquí también se nos pueden plantear situaciones especiales, como cuando restemos a un operando pequeño uno mas grande (Recordemos que por ahora sólo trabajamos en binario puro. Es decir, números sin signo). Tomemos el caso (SUB CX,DX) cuando CX vale 0077h y DX vale 8273h. Tras realizarse la operación, CX tendría el valor 7E74h. Esto se debe a que la resta se realiza de derecha a izquierda, y bit a bit, como vamos a ver ahora. Cómo se realiza realmente la resta (bas,monos en el ejemplo): El procesador tiene los dos valores en binario: CX = 0000000001110111 DX = 1000001001110011 Acto seguido, procede a realizar la resta, bit a bit (y de derecha a izquierda).
CX = 0000000001110111 - DX = 1000001001110011 ---------- CX = 0111111001110100 = 7E74H en base hexadecimal.
Por tanto, CX=7E74H tras realizar la operación.
Otro ejemplo: SUB AX,37h ---> Resta a AX el valor 37h
Otro mas: SUB BYTE PTR ES:[SI],AL ---> Resta el valor contenido en AL, a la posición direccionada mediante SI, dentro del segmento de datos apuntado por ES.
Otro: SUB variable1,word ptr [di] ---> Este ejemlo como podreis deducir por vosotros mismos, es un ejemplo de instrucción no permitida. Como ya vimos en lecciones anteriores, no podemos direccionar dos posiciones de memoria diferentes dentro de la misma instrucción. De tal manera, que esta instrucción habrá que descomponerla en 2 diferentes: MOV AX,WORD PTR [DI] ---> Deposito en AX el valor contenido en la posición de memoria direccionada por DI. De esta manera, en la siguiente instrucción usar, AX y no una dirección de memoria.

SUB variable1,AX ---> Ahora sí. Restamos a variable1 (que al fin y al cabo, es una posición de memoria. Tipo palabra en este caso) el contenido del registro AX.
- INC (Incremento en una unidad). Se utiliza cuando lo que se quiere hacer es una suma de una unidad. Entonces se utiliza esta instrucción. La sintaxis es: INC operando. Ejempo: INC AX ---> Incrementa el valor de AX en una unidad. Si antes de la instrucción, AX tenía el valor 3656h, ahora tendr el valor 3657h. Muy importante: Si antes de la instrucción, AX tenía el valor 0FFFFH, ahora tendr el valor 0000h. Al sumar bit a bit y de derecha a izquierda, queda todo Cero, y al final quedaría un 1, que se pierde porque no cabe en el registro. Aquí pues también se produciría overflow.
Otro ejemplo: INC BYTE PTR [BX] ---> Incrementa en una unidad el valor contenido en la posición de memoria direccionada por BX.
- DEC (Decremento en una unidad). Se utiliza cuando se quiere restar una unidad a un valor dado. La sintaxis de la instrucción es: DEC operando. Ejemplo: INC AX ---> Decrementa el valor de AX en una unidad. Si antes de la instrucción, AX tenía el valor 3656h, ahora tendrá el valor 3655h. Muy importante: Si antes de la instrucción, AX tenía el valor 0000H, ahora tendrá el valor 0FFFFh. Al restar bit a bit y de derecha a izquierda, queda todo con valor 1, quedando al final 0FFFFH fruto de este DEC.
Otro ejemplo: DEC BYTE PTR [BX] ---> Decrementa en una unidad el valor contenido en la posición de memoria direccionada por BX.
- ADC (Suma teniendo en cuenta el acarreo anterior). Se utiliza para operaciones cuyos operandos tienen mas de un registro de longitud. A la hora de hacer la suma, se tiene en cuenta el posible acarreo de una operación anterior. Esto es posible, gracias al flag Cf ó flag de acarreo. Tanto ésta como la siguiente son instrucciones poco usadas. Yo nunca las uso.
- SBB (Resta teniendo en cuenta 'lo que me llevo' de la operación anterior:-)) Se utiliza para operaciones cuyos operandos tienen mas de un registro de longitud. A la hora de hacer la resta, se tiene en cuenta 'lo que me llevo' de una operación anterior. Esto es posible, gracias al flag Cf ó flag de acarreo.
* MULTIPLICACION Y DIVISION * Estas operaciones aceptan sólo un operando, de forma que según sea su tamaño byte o palabra, asumen que el otro operando está en AL ó AX respectivamente. Esta es una de las instrucciones que os decía (en la lección 1) que tienen el registro acumulador (AX/AH/AL) implícito en la instrucción. De tal manera que no hace falta especificarlo, y sólo es necesario indicar el otro operando involucrado en la operación.
- MUL (multiplicación de datos sin signo). Sintaxis: MUL operando. Realiza la multiplicación del operando dado, con el acumulador. Dependiendo del tamaño del operando introducido en la operación, el procesador tomar AL o AX como segundo operando.
* Operando de tipo byte: El procesador asume que el otro operando se encuentra almacenado en el registro AL, y el resultado de la operación lo deposita en el registro AX.
* Operando de tipo palabra: El procesador asume que el otro operando está almacenado en el registro AX, y el resultado de la operación lo depositar en el par de registros DX,AX. Teniendo DX la parte mas significativa ó de mayor peso del resultado.
-IMUL (multiplicación de datos con signo). Igual que arriba, pero teniendo en cuenta que se trabaja con números con signo.
- DIV (División de datos sin signo). Sintaxis: DIV divisor. Divide el operando almacenado en el registro acumulador por el divisor. Es decir, acumulador/divisor. Dependiendo del tamaño del divisor introducido, el procesador asume que el dividendo se encuentra en AX ó en el par de registros DX,AX.
* Divisor de tipo byte: El procesador asume que el dividendo se encuentra almacenado en el registro AX. El resultado de la operación se desompone en AH (resto) y AL (cociente).

* Divisor de tipo palabra: El procesador asume que el dividendo se encuentra almacenado en el par de registros DX,AX. Teniendo DX la parte mas significativa. El resultado de la operación se descompone en DX (resto) y AX (cociente).
- IDIV (División de datos con signo). Igual que arriba, pero teniendo en cuenta que se trabaja con números con signo.
Hay que tener muy en cuenta al utilizar estas instrucciones de división, que la ejecución de la operación no desemboque en error. Esto sucede con la famosa división por Cero, entre otras situaciones. También sucede cuando el cociente obtenido en una división no cabe en el registro utilizado para almacenarlo. En estos casos, se produce una INT 0, que origina la terminación del programa en curso.

--- Instrucciones de manejo de bits.
* Instrucciones de desplazamiento de bits * Son instrucciones que nos permiten desplazar los bits dentro de un regitro o una posición de memoria. Estas instrucciones actúan sobre datos de tipo byte (8 bits) y de tipo palabra (16 bits).
- SHL (desplazamiento a la izquierda). Mediante esta instrucción podemos desplazar a la izquierda los bits de un registro o posición de memoria. Esto que puede parecer poco pr ctico, es muy útil en determinadas situaciones. Por ejemplo, es la manera mas rápida y cómoda de multiplicar por 2. Sintaxis: SHL registro,1 SHL registro,CL SHL memoria,1 SHL registro,CL
Los desplazamientos pueden ser de una sóla posición o de varias. Cuando queremos realizar un sólo desplazamiento usamos los formatos: SHL registro,1 SHL memoria,1
Pero cuando queremos realizar desplazamientos de mas de 1 posición, debemos usar el registro CL para indicar el número de desplazamientos deseados.
Veamos algunos ejemplos para aclararlo. Ejemplo: Queremos desplazar a la izquierda una posición los bits del registro AL. La instrucción necesaria sería: SHL AL,1. Veamos el efecto de la instrucción: Supongamos que en un principio, AL = B7h. Tenemos pues, antes de realizar la operación el registro AL de 8 bits, con el siguiente valor en cada uno de estos 8 bits: 10110111. Tras realizar el desplazamiento, el registro quedaría como: 01101110. Hemos desplazado todos los bits una posición a la izquierda. El bit de mayor peso (bit 7), el de mas a la izquierda, se pierde. Y el bit de mas a la derecha (bit 0) ó de menor peso, toma el valor 0. El registro AL (tras la instrucción) tiene un valor de 6EH. Si volvemos a ejecutar la instrucción (SHL AL,1) con el nuevo valor de AL, tras la ejecución, tendremos los bits del registro de la siguiente manera: AL = 11011100. Si pasamos este número binario a hexadecimal, tenemos que AL = 0DCH. Si seguimos realizando desplazamientos a la izquierda, terminaremos por quedarnos con el registro con todos los bits a Cero, debido a que el valor que entra por la derecha en cada desplazamiento es un cero (0).
Otro Ejemplo: Queremos desplazar a la izquierda los bits del registro AL 3 posiciones. Para llevar a cabo el desplazamiento, primero tenemos que introducir en CL el número de 'movimientos' a la izquierda que se van a realizar sobre cada bit. Y luego, ejecutar la instrucción de desplazamiento en sí.
MOV CL,3 ---> Indicamos 3 'desplazamientos'. SHL AL,CL ---> Realiza el desplazamiento hacia la izquierda (3 veces).
Supongamos que antes de ejecutar la instrucción, AL = 83h. En binario: AL = 10000011. Tras la instrucción, los bits quedarían así: AL = 00011000. En hexadecimal: AL = 18H.
Un último ejemplo: Veamos ahora el caso especial en el que se utiliza la instrucción SHL para realizar multiplicaciones por 2. Supongamos que queremos multipicar el contenido del registro AL por 2. Pues bien, sólo podremos multiplicarlo mediante (SHL AL,1) cuando estemos seguros que el bit de mayor peso (de mas a la izquierda) valga cero. Es decir, el bit 7 ó de mayor peso no puede ser 1, ya que se perdería al realizar el desplazamiento,

con lo cual la multiplicación sería errónea. Siempre que tengamos la certeza que el bit de mayor peso vale cero podremos utilizar (SHL reg/mem,1) para duplicar (multiplicar por 2). Evidentemente, si hacemos 2 desplazamientos, estamos multiplicando por 4, y así sucesivamente: 3 desplazamientos = multiplicar por 8, etc.
Veamos el ejemplo: Queremos multiplicar por 8 el registor AL. Previamente en AL hemos depositado un número del 1 al 10. Por lo tanto, sabemos con certeza que el bit 7 vale 0, con lo cual podemos ahorrar tiempo utilizando la multiplicación mediante desplazamientos. La cosa quedaría como:
MOV CL,3 SHL AL,CL
- SHR (desplazamiento a la derecha). Mediante esta instrucción podemos desplazar a la derecha los bits de un registro o posición de memoria. Es la instrucción opuesta y complementaria a SHL. En este caso, la instrucción puede utilizarse para realizar divisiones por 2.
Sintaxis: SHR registro,1 SHR registro,CL SHR memoria,1 SHR registro,CL
Los desplazamientos pueden ser de una sóla posición o de varias. Cuando queremos realizar un sólo desplazamiento usamos los formatos: SHR registro,1 SHR memoria,1
Pero cuando queremos realizar desplazamientos de mas de 1 posición, debemos usar el registro CL para indicar el número de desplazamientos deseados.
Veamos algunos ejemplos. Ejemplo: Queremos desplazar a la derecha una posición los bits del registro DX. La instrucción necesaria sería: SHR DX,1. Veamos el efecto de la instrucción: Supongamos que en un principio, DX = 4251h. Tenemos pues, antes de realizar la operación el registro DX de 16 bits, con el siguiente valor en cada uno de estos 16 bits: 0100001001010001 Tras realizar el desplazamiento, el registro quedaría como: DX = 0010000100101000. Hemos desplzado todos los bits una posición a la derecha. El bit de menor peso (bit 0), el de mas a la derecha, se pierde. Y el bit de mas a la izquierda (bit 15) ó de mayor peso, toma el valor 0. El registro DX (tras la instrucción) tiene un valor de 2128H, vemos que es pr cticamente la mitad de su anterior valor. Si volvemos a ejecutar la instrucción (SHR DX,1) con el nuevo valor de DX, tras la ejecución, tendremos los bits del registro de la siguiente manera: DX = 0001000010010100. Si pasamos este número binario a hexadecimal, tenemos que DX = 1094H, que vuelve a ser la mitad del valor anterior. Si seguimos realizando desplazamientos a la derecha, terminaremos por quedarnos con el registro con todos los bits a Cero, debido a que el valor que entra por la izquierda en cada desplazamiento es un cero (0).
En ambas instrucciones SHL y SHR, el valor que entra nuevo en los desplazamientos es un cero. Seguidamente veremos instrucciones similares a ,stas que permiten que entre un número distinto de cero al realizar los desplazamientos. Son las instrucciones SAL y SAR.

Cuando realizamos divisiones mediante SHR, como el bit que se pierde es el de menor peso (el de mas a la derecha), no tenemos el problema que se nos planteaba con la multiplicación mediante SHL. Es decir, aquí como mucho el resultado final pierde el valor media unidad (0.5).
- SAL y SAR. Estas instrucciones se diferencian de las anteriores (SHL y SHR) en que el nuevo valor binario que entra al realizar el desplazamiento es igual al bit de mayor peso. De cualquier modo, no pongais demasiada atención en estas instrucciones. Rara vez (por no decir nunca) las tendreis que utilizar.
* Instrucciones de rotación de bits * Son instrucciones an logas a las anteriores (de desplazamiento). La diferencia es que aquí no se producen desplazamientos, sino rotaciones en los bits. Es decir, no se pierde ningún bit, sino que entra por el lado opuesto a por donde sale.
Estas instrucciones al igual que las anteriores, actúan sobre datos de tipo byte (8 bits) y de tipo palabra (16 bits).

- ROL (Rotación a la izquierda). Rota a la izquierda los bits de un registro o posición de memoria. El bit mas significativo no se pierde, sino que al rotar, entra por el otro extremo del operando, pasando a ser ahora el bit menos significativo. Sintaxis: ROL registro,1 ROL registro,CL ROL memoria,1 ROL registro,CL
Veamos un ejemplo: Tenemos el registro AL con el valor 78h, que en binario es: 01111000. Si ejecutamos la instrucción (ROL AL,1), tendremos acto seguido que AL tiene el valor binario 11110000. Si volvemos a ejecutar esa instrucción con el nuevo valor de AL, tendremos 11100001. Si lo volvemos a hacer repetidas veces, tendremos: 11000011 10000111 00001111 00011110 00111100 01111000 ---> vuelta al valor original.
- ROR (Rotación a la derecha). Rota a la derecha los bits de un registro o posición de memoria. El bit menos significativo no se pierde, sino que al rotar, entra por el otro extremo del operando, pasando a ser ahora el bit mas significativo. Sintaxis: ROR registro,1 ROR registro,CL ROR memoria,1 ROR registro,CL
Ejemplo: Tenemos el registro AL con el valor 78h (igual que en el ejemplo anterior). En binario, AL = 01111000. Si ejecutamos la instrucción (ROR AL,1), tendremos acto seguido que AL tiene el valor binario 00111100. Si volvemos a ejecutar esa instrucción con el nuevo valor de AL, tendremos 00011110. Si lo volvemos a hacer repetidas veces, tendremos: 00001111 10000111 11000011 11100001 11110000 01111000 ---> vuelta al valor original.
- RCL y RCR (Rotar a izquierda y derecha con carry ó acarreo). Estas instrucciones son variantes de las anteriores. La diferencia estriba en que la acción de rotar se va a hacer en dos pasos: 1.- El bit que se encuentra en el flag Cf es el utilizado para introducir en el extremo del operando. 2.- El bit que sale por el otro extremo (bit rotado) pasa a la bandera Cf.
Ejemplo: Tenemos el registro AL con el valor 78h (igual que en el ejemplo anterior). En binario, AL = 01111000. Tenemos también el falg Cf (flag de Carry ó acarreo) con valor 1. AL = 01111000. Cf=1.
Si ejecutamos la instrucción (RCR AL,1), tendremos acto seguido que AL tiene el valor binario 10111100. El valor que ha entrado por la izquierda es el que tenía la bandera Cf. Pero a la vez, la bandera Cf despues de ejecutar la instrucción, tendr valor cero (0), por el bit rotado (el que ha salido por la derecha).
No os preocupeis si os parece muy lioso. Este tipo de instrucciones casi nunca se utilizan. Yo nunca las he utilizado en ningún programa. De cualquier manera, cuando hagamos unos cuantos programas, ya tendreis soltura suficiente como para probarlas.

--- No hemos visto todos los grupos de instrucciones: Nos queda por ver, principalmente, las operaciones lógicas (AND, OR, etc.) y las operaciones con hileras ó cadenas de caracteres. Esto lo veremos en una próxima lección. Por hoy ya hay demasiadas cosas nuevas.

Esto es todo por hoy. El próximo día mas. Practicad un poco con las instrucciones que hemos visto hoy. Probad a hacer algún programilla con ellas, aunque no lo ensambleis luego. Lo importante es saber para que sirven las instrucciones que hemos visto.

ASM POR AESOFT. (lección 8). ---------- - DUDAS DE LECCIONES ANTERIORES - SOLUCION AL EJERCICIO DE LA LECCION ANTERIOR - CONJUNTO DE INSTRUCCIONES DEL 8086(II): * Operaciones lógicas ó booleanas (Continuación del apartado Operaciones de manejo de bits, lección 7). *

Operaciones de manejo de hileras o cadenas de caracteres. ---------- ----------

Saludos, mis queridos programadores. :-))

En la lección de hoy, vamos a seguir con la relación de las instrucciones del 8086.

También veremos en primer lugar la respuesta que le doy a un usuario acerca de unas dudas que me plantea. Dicho mensaje me parece de inter,s general, por eso lo incluyo en la lección de hoy.
Por último, aunque no en último lugar, dar, la solución al sencillo ejercicio que os propuse en la lección anterior, ya que parece que nadie me ha dicho cómo resolverlo.

- DUDAS DE LECCIONES ANTERIORES ---------- A continuación os muestro un mensaje que considero de inter,s para todas las personas que siguen el curso:

--- ---------- inicio del mensaje.

> Cada uno de estos registros tiene funciones
> especiales que es interesante
> conocer. Por ejemplo el registro AX es el
> llamado acumulador, hace que
> muchas operaciones tengan una forma mas corta,
> ya que lo especifican
> implícitamente. Es decir, que hay operaciones
> que actúan sobre el
> registro AX en particular.

AJ> Con esto te refieres por ejemplo a cuando se llama a una
AJ> interrupcion con un valor en AX sin que tengamos que indicarle para
AJ> nada donde tiene que encontrar ese valor, puesto que ya sabe que lo va
AJ> a encontrar en AX :-?

Me refería mas bien a ciertas instrucciones aritm,ticas, que presuponen que un operando se encuentra almacenado en AX, y el otro operando puede estar en cualquiera de los otros registros. Cuando leas la lección 7 (que debes tener ya en tus manos) comprender s esto mejor.
El registro AX (como acumulador que es), se utiliza en otras muchas instrucciones de forma implícita (esto es, que no es necesario indicarlo expresamente). Entre estas otras instrucciones podemos encontrar ciertas instrucciones de transferencia de cadenas de caracteres: LODS y STOS, que utilizan el registro AL (en el caso de las instrucción LODSB y STOSB) ó el registro AX (en el caso de las instrucción LODSW y STOSW). La diferencia entre estas instrucciones estriba en el hecho de trabajar con bytes (registros de 8 bits) ó trabajar con palabras (registros de 16 bits). Pero bueno, no me enrollo aquí mas, ya que estas instrucciones se desarrollan en la lección 8.
En cuanto a lo que me dices acerca de las interrupciones, no sería el ejemplo mas acertado, pero es v lido, ya que cuando llamas a una interrupción (DOS, BIOS, etc.), ,sta sabe que función de la interrupción ejecutar gracias al registro AX, que contiene el número de dicha función.
Entonces podemos decir que la instrucción INT (llamada a interrupción), utiliza el registro AX implícitamente al no aludir en la sintaxis de dicha instrucción a tal registro, y utilizar la instrucción dicho registro para conocer el número de función que ejecutar. Obviamente antes de la instrucción INT, hemos tenido que cargar en AX (mediante la instrucción MOV, por ejemplo) el valor adecuado (número de la función a ejecutar).

> Af: Bit de carry auxiliar, se activa si una
> operación aritm,tica produce
> acarreo de peso 16.
AJ> ¨que es eso de acarreo de peso 16?
El peso (al hablar de los bits de un registro) es la posición que ocupa un bit determinado dentro de un registro. Por decirlo de alguna manera, es la importancia de ese bit dentro del registro. Al igual que ocurre en la base decimal, en la que el dígito de mas a la derecha de un número es el menos importante (de menor peso), así ocurre en la base binaria (bits) y en el resto de las bases, por supuesto.
Entonces cuando hablamos de acarreo de peso 16, nos referimos al acarreo que surge fruto de trabajar con los bits de mayor peso (peso 16), los de mas a la izquierda.
--- Una cosa que no dije en su momento: El flag Af (bit de carry auxiliar) se utiliza en operaciones decimales. No debe tenerse en cuenta en operaciones aritm,ticas con enteros. De cualquier modo, no es necesario prestarle demasiada atención a este flag. Cuando uno empieza, nunca lo utiliza. Y cuando ya lleva mucho tiempo programando, tiene experiencia y hace cosas muy técnicas que requieran de dicho flag... Entonces, evidentemente ya sabreis todo lo necesario acerca de el. :-))) O sea, que pasando de el.
> Venga, ahora quiero que me conteis dudas que
> teneis, aclaraciones, etc.
AJ> Como ves te he hecho caso y aqui tienes un par de dudillas ;-)
Pues ahí queda mi respuesta. Espero haberte ayudado.
--- ---------- fin del mensaje
Espero que os haya parecido interesante.

- SOLUCION AL EJERCICIO DE LA LECCION ANTERIOR ----------
----- Esto es lo que os proponía en la lección anterior:
--- A ver si alguien me dice cómo podemos modificar el flag Tf, por ejemplo. Os dar, una pista: ¨Recordais las instrucciones PUSHF y POPF? Espero vuestros mensajes. Si a nadie se le ocurre, ya dejar, yo la solución en una próxima lección.

Y aquí est la solución:
Debido a que no podemos acceder directamente a determinados bits del registro de estado (FLAGS), debemos realizar una serie de operaciones para que de esta forma nos sea posible la modificación de los bits a los que no podemos acceder directamente.
Si lo que queremos es poner el flag Tf con valor (1), entonces basta con realizar la siguiente operación:
MOV AX,0000000100000000b PUSH AX POPF
La primera instrucción prepara un nuevo registro de estado (FLAGS) en el que como sólo nos interesa el flag Tf, lo ponemos a 1 (que es lo que queremos), y los otros bits (flags) los dejamos a Cero, los podriamos haber dejado a 1 también. Para el caso que nos ocupa, da igual.
La segunda instrucción deja este nuevo registro de estado en la pila. Esto se hace así, ya que tenemos una instrucción que sacar ese nuevo registro de estado de la pila, y lo pondr como nuevo registro de estado o FLAGS. La tercera línea hace que el nuevo registro de estado sea la palabra introducida en la pila por la orden (PUSH AX).
Es decir, se trata de utilizar la instrucción POPF para poder coger de la pila un nuevo registro de estado a gusto del programador, que previamente ha sido depositado en la pila (mediante PUSH registro).
Si queremos poner el flag Tf con valor (0), basta con:
MOV AX,0 PUSH AX POPF

Con lo expuesto hasta ahora se resolvía el ejercicio que os pedí, pero esto en la pr ctica no es nada útil, ya que estamos 'machacando' el valor del resto de los flags, cuando sólo queremos modificar uno en concreto. para evitar eso, nos valemos de las operaciones lógicas que vamos a ver a continuación. Estas operaciones lógicas las utilizaremos (en este caso) para aislar el resto de los bits, y así mantener su valor original.
~~~ Una vez que hayais estudiado las operaciones lógicas que se desarrollan mas abajo, estareis en condiciones de solucionar el siguiente ejercicio:
... Se trata de modificar el flag Tf, pero (y esto es muy importante) sin cambiar el valor del resto de flags. Espero que alguien me d, la solución (si lo haceis todos, mejor) :-)).

- CONJUNTO DE INSTRUCCIONES DEL 8086 (II) ----------
Continuamos en este apartado con la relación y explicación de las instrucciones del 8086:
* Operaciones lógicas ó booleanas * (Continuación de Operaciones de manejo de bits). Todos habreis visto que en las buenas calculadoras (os recuerdo que la mejor es SB-CALCU del programa SANBIT :-)) aparecen una serie de operaciones como son: NOT, AND, OR, etc... Pues bien, esas son las llamadas operaciones lógicas. En serio, os recomiendo que utiliceis el programa SANBIT para realizar todo tipo de operaciones booleanas (Un poco de publicidad :-)))
Estas operaciones trabajan a nivel de bits, con un tamaño de operando dado. Esto es, no es lo mismo un NOT (7) con un tipo de datos byte, que originaría como resultado 248, que hacer un NOT (7) con un tipo de datos word, que originaría como resultado 65528. Vemos pues, que el tipo de datos sobre el que se realiza una operación lógica, condiciona el resultado de la operación.
La finalidad de estas instrucciones es modificar uno o varios bits concretos de un registro o una posición de memoria. Son útiles para aislar ciertos bits, y trabajar con ellos como si fueran variables independientes. Es decir, usando adecuadamente estas operaciones lógicas, podemos tener hasta 16 variables de tipo lógico (valor 0 ó 1) en un registro de tipo word. Como veremos a continuación, estas operaciones se utilizan muchas veces para realizar con mayor rapidez y menor código de ejecución ciertas acciones, que por m,todos mas comunes acarrearían mas tiempo y código.

- NOT lógico. La operación lógica NOT, consiste en sustituir los unos por ceros y viceversa en un operando dado. Sintaxis: NOT registro/posición_de_memoria. Ejemplo: NOT AL. Supongamos que AL tiene el valor 99h. En binario tendríamos: AL = 10011001. La instrucción lo que hace es invertir el valor de cada bit. Si antes de la instrucción valía 1, ahora valdr 0, y viceversa. Siguiendo este criterio, despues de realizar la operación, AL valdr 01100110, que en base 16 (hexadecimal) es: AL = 66H. Si ejecutamos de nuevo la instrucción NOT AL, con el nuevo valor de AL, parece evidente lo que vamos a obtener, ¨no? Por supuesto obtendremos el mismo valor que al principio: AL = 99h.
Ejemplo: Veamos ahora que sucede con la operación NOT AX, suponiendo que AX = 99H. Ahora estamos trbajando sobre un operando de tamaño word ó palabra, por lo tanto, tenemos 16 bits a los que cambiar su valor. Antes de la instrucción, AX tenía el valor 0000000010011001. Despues de la instrucción, AX = 1111111101100110. En base 16, AX = 0FF66H.
--- El primer 0 de 0FF66H se pone para que el ensamblador sepa que estamos
--- refiri,ndonos a un número, y no a una variable llamada FF66H.
- AND lógico. La operación lógica AND, al igual que las restantes -y al contrario que la operación NOT- opera sobre dos operandos. El resultado de la operación se almacena en el primer operando, que puede ser un registro o una posición de memoria.
Este resultado se obtiene de la siguiente manera: Se compara cada uno de los bits del primer operando con sus correspondientes bits del segundo operando. Si ambos tienen el
~~~

valor (1), el bit correspondiente del operando resultado se pone a valor (1). Por el contrario, si alguno de los dos bits (o los dos bits) tiene valor (0), el bit correspondiente del operando resultado valdr (0). De ahí viene el nombre de la instrucción: AND ... Uno y (AND) otro, los dos bits deben tener valor (1) para que el bit correspondiente del resultado tenga valor (1). Esta operación se utiliza para poner a (0) determinados bits.
Sintaxis: AND registro,registro AND registro,posición_de_memoria AND registro,valor_inmediato AND posición_de_memoria,registro AND posición_de_memoria,valor_inmediato
Ejemplo: Supongamos que AX = 1717H y VAR1 (Variable con la que accedemos a una posición de memoria de tipo Word) tiene el valor 9876H. En binario tendríamos:
AX = 0001011100010111 VAR1 = 1001100001110110
Veamos cómo se realizaría la operación lógica (AND AX,VAR1)... Se trata de operandos de tipo word (16 bits), por tanto hay que realizar 16 operaciones (1 para cada posición de bit) con los bits. Ya hemos visto mas arriba la forma en que opera esta instrucción. Veamos que resultado nos daría:
AX = 0001011100010111 VAR1 = 1001100001110110 ---------- Tras la instrucción: AX = 0001000000010110, que pasado a base 16 nos queda AX = 1016h.
- OR lógico. La operación lógica OR se utiliza al contrario que la operación AND, para poner a (1) determinados bits de un registro o posición de memoria. La operación lógica OR, opera sobre dos operandos, almacenando el resultado de dicha operación en el primer operando, que puede ser un registro o una posición de memoria.
Este resultado se obtiene de la siguiente manera: Se compara cada uno de los bits del primer operando con sus correspondientes bits del segundo operando. Si alguno tiene el valor (1), el bit correspondiente del operando resultado se pone a valor (1). Por el contrario, si los dos bits tiene valor (0), el bit correspondiente del operando resultado valdr (0). De ahí viene el nombre de la instrucción: OR ... Uno u (OR) otro, con que uno sólo de los dos bits tenga valor (1), el bit correspondiente del resultado tendr valor (1).
Sintaxis: OR registro,registro OR registro,posición_de_memoria OR registro,valor_inmediato OR posición_de_memoria,registro OR posición_de_memoria,valor_inmediato
Ejemplo: Supongamos que CL = 25H y COLUM (Variable con la que accedemos a una posición de memoria de tipo Byte) tiene el valor 0AEH. En binario tendríamos:
COLUM = 10101110 CL = 00100101
Veamos cómo se realizaría la operación lógica (OR COLUM,CL)... Se trata de operandos de tipo byte (8 bits), por tanto hay que realizar 8 operaciones (1 para cada posición de bit) con los bits. Veamos que resultado nos daría:
COLUM = 10101110 CL = 00100101 ---------- Tras la instrucción: COLUM = 10101111, que pasado a base 16, nos queda la variable COLUM = 0AFH.
- XOR (OR Exclusivo). La instrucción XOR opera en modo parecido a OR, pero con una diferencia muy importante, que le hace tener el sobrenombre de OR Exclusivo:
Se compara cada uno de los bits del primer operando con sus correspondientes bits del segundo operando. Si uno y sólo uno de ambos bits comparados tiene valor (1) -obviamente el otro bit debe ser (0), es decir, ambos bits tienen diferente valor-, entonces el bit correspondiente del resultado tendr valor (0)
He aquí la diferencia con la instrucción OR: Mientras que la operación OR admitía que uno o los dos bits fuera (1) para poner a (1) el bit resultante, la instrucción XOR exige que sólo uno de esos bits tenga valor (1), es decir, que ambos bits tengan diferente valor.
Sintaxis: XOR registro,registro XOR registro,posición_de_memoria XOR registro,valor_inmediato XOR posición_de_memoria,registro XOR posición_de_memoria,valor_inmediato

Esta instrucción se utiliza normalmente para poner a Cero un registro. Es la manera mas r pida de poner a cero un registro. Ve moslo con un ejemplo:
Supongamos que queremos poner a cero el registro AX. Da igual el valor que tenga AX para obtener el resultado final de la siguiente operación, pero le damos por ejemplo el valor AX = 2637h. En binario tendríamos: AX = 0010011000110111
Si realizamos la operación (XOR AX,AX), cu l podría ser el resultado a almacenar en AX? Parece evidente, no? Hemos quedado en que el bit resultante tiene valor (1) si los dos bits comparados son diferentes, entonces llegamos a la conclusión que todos los bits del resultado van a tener valor (0), ya que al comparar un registro consigo mismo, todos y cada uno de los bits de ambos registros (que en realidad es el mismo registro) son iguales de uno a otro registro.
Veamos cómo se realizaría la operación lógica (XOR AX,AX)... Se trata de operandos de tipo word (16 bits), por tanto hay que realizar 16 operaciones (1 para cada posición de bit) con los bits. Veamos que resultado nos daría:
AX = 0010011000110111 AX = 0010011000110111 ---------- Tras la instrucción: AX = 0000000000000000, ya que todas las parejas de bits comparados tienen el mismo valor.
Esta forma de borrar un registro es muy r pida, y gener muy poco código ejecutable. Vamos a comparar este m,todo con el 'tradicional' (MOV AX,0):
La instrucción (MOV AX,0) tiene un código ejecutable de 3 bytes, y tarda en realizarse 4 pulsos de reloj. Mientras que la instrucción (XOR AX,AX) tiene un código de 2 bytes, y tarda en realizarse 3 pulsos de reloj.
Puede parecer que la diferencia no es muy grande, pero cuando se ejecutan miles o millones de estas instrucciones en un programa, se nota la diferencia.
- TEST. Esta operación es similar a AND, pero con una diferencia muy importante: La instrucción TEstáno modifica el valor de los operandos, sino sólo el registro de estado (FLAGS). Para realizar la operación utiliza registros internos del procesador, de esta manera, los operandos pasados a la instrucción TEST, no se ven alterados. Esta instrucción se realiza para comprobar el valor de un cierto bit ó ciertos bits dentro de un registro.
Sintaxis: TEST registro,registro TEST registro,posición_de_memoria TEST registro,valor_inmediato TEST posición_de_memoria,registro TEST posición_de_memoria,valor_inmediato
Veamos un ejemplo:
Existen dos posiciones de memoria de tipo byte consecutivas que utiliza la ROM BIOS para mantener y actualizar el estado de ciertas teclas especiales. Para este ejemplo, nos interesa sólo la primera posición 0000:0417H. Es decir, la posición 0417h dentro del segmento 0000h.
Esta posición de memoria contiene las siguientes informaciones lógicas en cada uno de sus bits:
³bit 7 6 5 4 3 2 1 0
³ Ú---Â---Â---Â---Â---Â---Â---Â---¿ Estado de teclas cuando el bit
³ ³ ³ ³ ³ ³ ³ ³ correspondiente tiene valor (1).
³ À---Á---Á---Á---Á---Á---Á---Á---Ù ---
³ ³ ³ ³ ³ ³ ³ ³
³ ³ ³ ³ ³ ³ ³ ³ À----> Tecla Mays. derecha pulsada.
³ ³ ³ ³ ³ ³ ³ À--------> Tecla Mays. izquierda pulsada.
³ ³ ³ ³ ³ ³ À------------> Tecla Control pulsada.
³ ³ ³ ³ ³ À----------------> Tecla Alt pulsada.
³ ³ ³ ³ À-------------------> Scroll Lock activado.
³ ³ ³ À------------------------> Num Lock activado.

³ ³ À----------> Caps Lock (Bloq Mays) activado.
³ À----------> Insert activado.
³
³
³ Evidentemente, cuando el bit correspondiente a una información lógica
³ en particular (Tecla ALT, p.e.) tiene valor (0), esto indica todo lo
³ contrario a lo mostrado arriba. En el caso del bit 3 (tecla ALT),
³ si estuviera con valor (0), querría decir que no est pulsada en estos
³ momentos.

El bit 7 (bit de mayor peso, o de mas a la izquierda) del byte direccionado por esa posición de memoria, contiene información booleana ó lógica (valor verdadero ó falso) acerca del modo de inserción de teclado. Es decir, mediante este bit podremos saber si el modo de inserción est activado o no.

Debido a que hay otra serie de variables lógicas ó booleanas dentro de este byte, no podemos realizar una comparación a nivel de byte para conocer el estado de insertar. Esto es, no podemos escribir algo así como: CMP BYTE PTR ES:[DI],10000000b, ya que el resto de bits tienen un valor variable dependiente de ciertas circunstancias (mayúscula izquierda pulsada, tecla ALT pulsada, etc).

Debemos entonces usar la instrucción TEST, con la que podemos 'TESTar' ó examinar el valor de un determinado bit concreto. En el caso que nos ocupa (examinar el estado del modo de inserción), debemos examinar el valor del bit 7 de la posición de memoria 0000:0417h.

Veamos todo el proceso:

```
;******** trozo de programa.
XOR AX,AX ; MOV ES,AX ; Mediante estas dos instrucciones, hago que la base del ; segmento direccionado mediante el registro ES, se ; encuentre en la posición 0000h (Al principio de la memoria ; del PC). ; Observaciones: ; - No es posible introducir un valor_inmediato en un ; registro de segmento. Es decir, no podemos escribir algo ; como: MOV ES,8394h. El procesador no lo permite. ; Debemos, pues, valernos de otro registro, como AX, DX, ; etc, para introducir el valor deseado en el registro de ; segmento. ; - Utilización de la operación lógica (XOR AX,AX) para poner ; a Cero el registro AX, ganando en rapidez, y menor tamaño ; del programa ejecutable, en comparación con (MOV AX,0). ; - Tenemos el registro ES apuntando ya al segmento adecuado. ; Sigamos:
TEST BYTE PTR ES:[0417H],10000000b ; Examinemos esta instrucción tan ; compleja en profundidad... ; - Vemos que su sintaxis es del tipo: ; TEST posición_de_memoria,valor_inmediato. ; - Debemos indicar que vamos a comparar una posición de tipo ; byte, para evitar errores. Eso lo hacemos mediante: ; BYTE PTR. ; - Debemos indicar a continuación el segmento y desplazamiento ; donde se encuentra ese byte al que vamos a acceder: ; ES:[0417H] ; - Utilizamos el número 10000000b como valor_inmediato. De ; esta forma es como vamos a examinar el bit 7 ó de estado ; de insertar. ; Hemos visto que la instrucción TEST es un AND que no ; modifica el valor de los operandos, sino sólo los flags. ; Por tanto, sólo podemos saber el valor de ese bit 7, ; comprobando el valor de los flags tras la operación. ; El flag que hay que 'mirar' es el flag Zf (flag Cero). ; Si tras la operación el flag Zf est con valor (1), esto ; quiere decir que el bit 7 de ES:[0417H] est desactivado, ; es decir, que estamos en estado de NO_INSERCION.
; Veamos Gráficamente el proceso de esta instrucción TEST para comprenderlo
; mejor:
;
; Vamos a suponer que el byte ES:[417H] tiene el siguiente valor: 10101110b.
; Es un valor que he puesto al azar para poder operar con algo concreto.
;
```

; Tenemos entonces los dos valores siguientes para hacer el TEST:
;
; ES:[0417H] = 10101110
; valor_inmediato = 10000000
; ----------
; Tras la ejecución, un registro interno del procesador (no accesible por
; el programador) tendr el siguiente valor: 10000000b
; Y el flag Zf que indica si el resultado tiene un valor Cero, estar puesto
; a (0), ya que el resultado tiene un valor distinto de cero.
; Por tanto, comprobando el valor del flag Zf tras la operación, sabremos
; que insertar est en modo activo.
JZ no_insertando ; Si el flag Zf tiene valor (1): ; Hacemos un salto condicional a una posición del ; programa, en la que se realizan las acciones ; pertinentes en el caso de que no est, el teclado en ; modo inserción.
[....] ; se realizan las instrucciones adecuadas para el caso ; en que el teclado est en modo de inserción.
JMP fin_insercion ; salto el trozo siguiente, reservado para cuando el ; teclado est en modo de NO_INSERCION.
no_insertando: ; indica el inicio del código preparado para utilizar ; en caso de NO_INSERCION.
[....] ; se realizan las instrucciones adecuadas para el caso ; en que el teclado NO est en modo de inserción.

fin_insercion: ; Hemos terminado de trabajar por ahora con el tema de ; la inserción, pasamos a otra cosa dentro del programa.
[....] ; sigue el programa.

;****** fin del trozo de programa.
Veamos cómo quedaría sin tantas explicaciones:
;********* trozo de programa:
XOR AX,AX MOV ES,AX TEST BYTE PTR ES:[0417H],10000000b JZ no_insertando [....] ;acciones para cuando INSERTANDO. JMP fin_insercion
no_insertando: [....] ;acciones para cuando NO_INSERTANDO.
fin_insercion: [....] ;sigue el programa.
;**********fin del trozo de programa.
Supongamos ahora que el byte ES:[417H] tiene el siguiente valor: 00100010b. Con este nuevo valor, el estado de insertar es falso (NO_INSERTAR).
Tenemos entonces los dos valores siguientes para hacer el TEST: ES:[0417H] = 00100010 valor_inmediato = 10000000 ---------- Tras la ejecución, un registro interno del procesador tendr el siguiente valor: 00000000b Y el flag Zf que indica si el resultado tiene un valor Cero, estar puesto a (1), ya que el resultado tiene un valor de cero. En este caso, al comprobar el valor del flag Zf, sabremos que insertar est en modo inactivo (es decir, el teclado est en modo SOBREESCRIBIR).
- NEG. Esta operación, por su forma de trabajar(equivale a un NOT seguido de un INC) podemos estudiarla aquí, pero la vamos a dejar para cuando tratemos los números negativos, ya que se utiliza para eso, para convertir un número en negativo.
NEG AX
º (Equivalentes)
NOT AX INC AX

--- Operaciones para el manejo de hileras o cadenas de caracteres Este tipo de instrucciones nos permiten (a grandes rasgos) realizar movimientos de bloques de memoria de una posición de la memoria a otra.
Veamos cada una de ellas:
- MOVS (MOV String, mover cadena de caracteres). Se utiliza para mover un byte (MOVSB) o una palabra (MOVSW) desde la posición de memoria direccionada por DS:SI a la dirección ES:DI. Antes de introducir esta instrucción en el programa, debemos haber cargado debidamente los registros con sus valores apropiados. En caso de mover un byte, utilizamos la sintaxis: MOVSB. En caso de mover una palabra, utilizamos la sintaxis: MOVSW.
En estos momentos puede parecer de poca utilidad esta instrucción, ya que el mismo resultado lo podemos obtener con la instrucción MOV que vimos en lecciones anteriores. Veremos la utilidad de esta instrucción y las siguientes, cuando veamos las partículas: REP (REPetir), REPZ ó REPE, y REPNZ ó REPNE.
MOVSB ---> Mueve el byte direccionado por DS:SI a ES:DI. MOVSW ---> Mueve la palabra direccionada por DS:SI a ES:DI.
- LODS (LOaD String, cargar cadena de caracteres en el acumulador). Se utiliza para introducir en el registro acumulador (AX si trabajamos con palabras; AL si trabajamos con bytes) la palabra o byte direccionado mediante DS:SI.
LODSB ---> Introduce en el registro AL (tamaño byte) el byte direccionado mediante DS:SI. LODSW ---> Introduce en el registro AX (tamaño palabra ó word) el byte direccionado mediante DS:SI.
- STOS (STOre String, almacenar cadena de caracteres). Almacena el contenido del acumulador (AX si trabajamos con palabras; AL si trabajamos con bytes) en la posición de memoria direccionada mediante ES:DI.
STOSB ---> Almacena el contenido del registro AL (tamaño byte) en la posición de memoria ES:DI. STOSW ---> Almacena el contenido del registro AX (tamaño palabra) en la posición de memoria ES:DI.
- CMPS (CoMPare String, comparar cadenas de caracteres). Se utiliza para comparar cadenas de caracteres. Compara las cadenas que empiezan en las direcciones DS:SI y ES:DI. Podemos comparar un byte (byte a byte, cuando usemos la partícula REP) o una palabra (palabra a palabra, cuando usemos la partícula REP).
CMPSB ---> Compara el byte situado en DS:SI con el byte situado en ES:DI. CMPSW ---> Compara la palabra situada en DS:SI con la palabra situada en ES:DI.
-SCAS (No_se_que String :-) Compara el contenido del acumulador (AX si trabajamos con palabras; AL si trabajamos con bytes) con la palabra o byte situado en la posición de memoria ES:DI.
SCASB ---> Compara el contenido del registro AL (tamaño byte) con el byte situado en la posición ES:DI. SCASW ---> Compara el contenido del registro AX (tamaño word) con la palabra situada en la posición ES:DI.
Bien... Hasta ahora hemos visto que estas instrucciones trabajan sólo con un byte o palabra. Son raras las ocasiones en las que utilizamos estas instrucciones para mover sólo un byte o palabra. Lo mas normal es utilizar estas instrucciones antecedidas de una de las siguientes partículas de repetición:
+ REP (REPetir CX veces) Repite una de las operaciones de movimiento/comparación de cadenas tantas veces como indique el registro CX. Tras cada una de estas repeticiones se decrementa el valor del registro CX, para saber cuando debe parar. También se incrementa/decrementa el valor de los punteros usados (SI y/o DI). Dependiendo de la instrucción de que se trate, habr que actualizar uno sólo de los punteros (STOS, LODS, SCAS) ó los dos (MOVS, CMPS).

Y por que digo: incrementa/decrementa? Pues porque los movimientos o comparaciones se pueden hacer hacia atr s o hacia delante. Cuando hacemos un movimiento/comparación hacia delante, a cada paso del bucle, se incrementan los registros SI y/o DI. Cuando hacemos un movimiento/comparación hacia atr s, a cada paso del bucle, se decrementan los registros SI y/o DI.
Dependiendo del tamaño usado (byte o palabra), los incrementos o decrementos en los puneteros ser n de 1 ó de 2 unidades, respectivamente.
La forma en que el programador indica que los movimientos/comparaciones se realizar n hacia delante o hacia atr s, viene dada por la manipulación del flag Df. Para indicar que queremos que los movimientos/comparaciones se realicen hacia delante, debemos poner el flag Df con valor (0). Para indicar que queremos que los movimientos/comparaciones se realicen hacia atr s, debemos poner el flag Df con valor (1). Ya vimos la manera de modificar el valor del flag Df:
STD ---> Pone el flag Df con valor (1). CLD ---> Pone el flag Df con valor (0).
Ejemplo: Queremos mover 77 palabras (hacia adelante) desde la posición de memoria 7384h del segmento 8273h a la posición de memoria 7263h:8293h. La cosa quedaría así:
MOV AX,7263H MOV ES,AX ;Registro ES con valor adecuado (7263h). MOV DI,8293H ;Puntero destino (DI) con valor adecuado (8293h). ;Dirección de destino en ES:DI (7263H:8293H). MOV AX,8273H MOV DS,AX ;Registro DS con valor adecuado (8273h). MOV SI,7384H ;Puntero (SI) con valor adecuado (7384h). ;Direcciones fuente y destino con su valor adecuado.
MOV CX,77 ;Indicamos que queremos hacer 77 movimientos. CLD ;Los movimientos los vamos a hacer hacia delante.
REP MOVSW ;Realiza 77 veces la instrucción MOVSW, actualizando ;debidamente los punteros fuente (SI) y destino (DI), ;tras cada iteración. Así mismo, decrementa el registro ;CX para saber cuando debe dejar de realizar repeticiones ;de la instrucción MOVSW. ;Cuando CX tenga valor Cero, dejar de realizar ;movimientos. ;En cada una de las 77 iteraciones, se coge la palabra ;contenida en DS:SI y la copia en ES:DI. ;Acto seguido, añade dos unidades a SI y a DI, para ;procesar el resto de las 77 palabras que componen el ;bloque que hemos indicado.

+ REPZ ó REPE (Repetir mientras CX <> 0, y Zf = 1). Repite una de las operaciones de comparación (operaciones de movimiento de cadenas no tienen sentido con REPZ) de cadenas tantas veces como indique el registro CX, siempre que el flag Zf sea 1. Es decir, se realizar la operación de comparación mientras CX sea distinto de Cero (aún queden elementos por comparar), y los dos elementos (bytes o palabras) comparados sean iguales. O sea, mientras que los elementos comparados sean iguales, y queden elementos por comparar, se proceder a comparar los siguientes.
Ejemplo: Queremos comparar la cadena situada en DS:SI con la cadena situada en ES:DI. La longitud de la cadena ser de 837 bytes. Supongamos que todos los registros de dirección tienen su valor adecuado.
;registros de dirección DS, SI, ES, DI con su valor adecuado.
MOV CX,837 ;837 comparaciones de elementos de tipo byte. REPZ CMPSB ;Realiza la instrucción CMPSB mientras CX <> 0 y el ;flag Zf tenga valor (1), es decir: ;cada vez que se realiza una comparación, comprueba si ;los elementos comparados son iguales ('mirando' Zf), ;si no son iguales, deja de realizar comparaciones. ;Si son iguales entonces comprueba si quedan elementos ;por comparar (CX <> 0), en caso de que no queden, ;deja de realizar comparaciones. ;Tras cada comparación, actualiza debidamente los ;punteros fuente (SI) y destino (DI). ;En este caso se añade una unidad a cada uno de estos ;dos registros (SI y DI).

JNZ diferentes ;Si tras la instrucción (REP CMPSB), el flag Zf ;tiene valor (0), eso quiere decir, que algún ;byte de la cadena fuente (DS:SI) no coincide con ;su correspondiente en la cadena destino (ES:DI). ;Es decir, las cadenas no son iguales. ;Entonces, realizamos un salto condicional a un ;trozo de código utilizado para el caso de que las ;cadenas sean diferentes.
[...] ;grupo de instrucciones que se ejecutan cuando ;las cadenas son iguales.
JMP fin_comparaciones ; salto el trozo de instrucciones que se ;ejecutan cuando las cadenas son diferentes.
diferentes: ;aquí empieza el grupo de instrucciones que se ;ejecutan cuando las cadenas son diferentes.
[...] ;grupo de instrucciones que se ejecutan cuando ;las cadenas son diferentes.
fin_comparaciones: ;sigue el programa...

La instrucción (REPZ CMPSB) ya veremos que es muy importante al tratar la programación de utilidades residentes. Mediante esta instrucción sabremos si ya ha sido instalado en memoria el programa. Simplemente hay que buscar un trozo del programa desde el principio de la memoria hasta la posición donde se encuentra el trozo a buscar. Si no se produce ninguna coincidencia, es porque el programa residente no est instalado. Si se produce una coincidencia, es porque el programa residente ya est instalado, con lo cual damos un mensaje al usuario (Programa ya instalado en memoria), y salimos a la linea de comandos otra vez. Pero bueno, ya veremos esto en profundidad al tratar los RESIDENTES.
+ REPNZ ó REPNE (Repetir mientras CX <> 0, y Zf <> 1). Repite una de las operaciones de comparación de cadenas tantas veces como indique el registro CX, siempre que el flag Zf sea 0. Es decir, se realizar la operación de comparación mientras CX sea distinto de Cero (aún queden elementos por comparar), y los dos elementos (bytes o palabras) comparados sean DIFERENTES. O sea, mientras que los elementos comparados sean diferentes, y queden elementos por comparar, se proceder a comparar los siguientes.

Nota: Cuando realizamos comparaciones/movimientos de cadenas de longitud par, lo lógico sería hacerlo de palabra en palabra, mientras que si la longitud es impar, es imprescindible trabajar con bytes. ¨Alguna duda al respecto?
Un último ejemplo de todo el tema de cadenas de caracteres: Queremos copiar la cadena origen (cadena_origen) al principio de Cadena_destino.
;******datos
[...]
Cadena_origen db 'SanBit V6.0 (Super Utilidades)' Cadena_destino db 'SanBit V5.6 (Utilidades residentes)'
[...] ;******fin de datos.
;*****código de programa PUSH DS POP ES ;Mediante estas dos instrucciones, lo que hago es darle al ;registro ES el mismo valor que tiene DS. Esto se hace ya que ;las dos cadenas están dentro del mismo segmento. ;Suponemos que el registro DS estaba desde un principio apuntando ;al principio de los datos, como es normal.
MOV SI,OFFSET Cadena_origen
--- ;Mediante OFFSET, lo que hacemos es
--- ;introducir en el registro SI, el desplazamiento (offset en
--- ;ingl,s) de la variable Cadena_origen. Es decir, hacemos que SI ;contenga la dirección de Cadena_origen. Utilizamos SI como ;puntero a Cadena_origen. ;Por el contrario con la instrucción (MOV SI,Cadena_origen), lo ;que haríamos sería introducir la primera palabra contenida ;en Cadena_origen al registro SI. MOV DI,OFFSET Cadena_destino ;Hacemos

que DI apunte a la variable ;Cadena_destino. Es decir, DI tendr la dirección de la variable ;Cadena_destino.
CLD ;Movimiento de datos hacia delante.
MOV CX,15 ;15 es la mitad de 30 (longitud de Cadena_origen).
REP MOVSW ;Realiza 15 movimientos de tipo palabra. Es decir, mueve 30 ;bytes desde Cadena_origen a Cadena_destino. ;Al trabajar con palabras en lugar de bytes, se gana mucho ;en velocidad, ya que el procesador tiene que utilizar el ;BUS la mitad de veces.
;******fin de código de programa.
Tras la ejecución de este trozo de programa, la variable Cadena_Destino tendr la siguiente cadena: 'SanBit V6.0 (Super Utilidades)entes)' Podemos observar que los últimos 6 caracteres permanecen intactos, mientras que los primeros 30 han sido 'machacados' por la instrucción, introduciendo en su lugar el contenido de Cadena_origen. Esto es todo por ahora.

ASM POR AESOFT. (lección 9). * Dedicada a Xavi * ---------- - BASES NUMERICAS (DECIMAL, BINARIA, ETC...) - CAMBIO DE BASE - REPRESENTACION NUMERICA: Rango, Resolución, etc... * Coma fija sin signo (Binario Puro). * Complemento a 2. * BCD. ----------
Hola de nuevo a todos los seguidores del CURSO DE ASM.
En esta lección vamos a tratar un tema muy importante en programación, como es el empleo de determinadas bases numéricas (binaria y hexadecimal) para la representación de la información.
Al finalizar la lección tendremos claro (eso espero :-) cómo se almacena un dato en la memoria, entenderemos por que determinados tipos de datos (tipo byte, tipo palabra, etc..) admiten un rango de representación determinado, así como una resolución determinada, sabremos operar en bases diferentes a la decimal (base 10), etc, etc...
--- El Rango de representación es el intervalo comprendido entre el menor número representable y el mayor. Así por ejemplo, el rango del tipo de dato byte es 0..255. Es decir, se pueden representar números desde el 0 al 255. El rango del tipo de dato palabra (Word) es 0..65535. Lo que es lo mismo, se pueden representar números comprendidos entre el 0 y el 65535, ambos inclusive.
--- La resolución de la representación es la diferencia num,rica que existe entre un número representable y el inmediatamente siguiente.
- BASES NUMERICAS ---------- Antes de entrar de lleno en las bases 2 y 16 que son las bases con las que trabaja el ordenador (en realidad el ordenador sólo trabaja en base 2, la base 16 se utiliza de cara al programador para compactar el número resultante de utilizar la base 2, que sería muy largo y engorroso para utilizar constantemente en los programas)... ... antes de meternos de lleno con ,stas bases, como os decía, nos sería muy útil para su entendimiento el saber del porque de la base decimal.
* Base Decimal (Base 10). Es la base a la que estamos acostumbrados desde siempre, la base num,rica mas utilizada. En esta base 10, contamos con 10 dígitos: 0,1,2,3,4,5,6,7,8 y 9. Mediante estos 10 dígitos podemos expresar cualquier número que deseemos.
El sistema de numeración decimal (base decimal) es un sistema de numeración posicional, al igual que los restantes sistemas que vamos a ver (binario, hexadecimal,etc), y a diferencia del sistema de numeración romano, por ejemplo.
Un sistema posicional es aquel en el que un número viene dado por una cadena de dígitos, estando afectado cada uno de estos dígitos por un factor de escala que depende de la posición que ocupa el dígito dentro de la cadena dada.
Es decir, que el dígito 9, valdr 9 si est al final de la cadena, en la posición reservada para las unidades; valdr 90 si el dígito se encuentra en la posición reservada para las decenas

(2¦ posición de derecha a izquierda); valdr 900 si el dígito se encuentra en la posición reservada para las centenas; etc, etc...
A esto es a lo que se le llama posicional, dependiendo de la posición que ocupe un dígito dentro de la cadena num,rica, tendr un valor o tendr otro.
Así por ejemplo, el número 8346 se podría descomponer como sigue: 8346 = (8 * 10^3) + (3 * 10^2) + (4 * 10^1) + (6 * 10^0)
El factor de escala de que habl bamos arriba, son las diferentes potencias de 10 que multiplican a un dígito dependiendo de su posición dentro de la cadena num,rica.
Ahora nos podríamos preguntar por que tenemos como sistema de numeración usual al sistema decimal, por que es el mas usado por todo tipo de gente, a que se debe que en todo el mundo sea el sistema utilizado por las personas (ya veremos que las m quinas no usan el sistema decimal, sino el binario).
Pues es bien sencillo: Porque tenemos 10 dedos. :-) Aún recordaremos eso que nos decían (a qui,n no?) en clase cuando empez bamos a contar, sumar, etc.. : No vale contar con los dedos! Intuitivamente, utiliz bamos nuestra elemental calculadora: las manos, para contar, realizar sumas y restas sencillas, etc.
* Base Binaria (Base 2). En esta base sólo contamos con 2 dígitos: 0 y 1. Al igual que la base decimal tiene su razón de ser, la base 2 o binaria tampoco ha surgido debido a un mero convencionalismo, sino que se basa en algo concreto: Electricidad.
Toda la información que se manipula dentro de un ordenador se hace de acuerdo a señales el,ctricas. Es lo único que entiende el ordenador. Mediante una señal el,ctrica alta, se representa el valor 1; mediante una señal el,ctrica baja se representa el 0.
. (1) : Tensión el,ctrica alta. . (0) : Tensión el,ctrica baja.
Todo el trabajo del procesador, buses, etc... se realiza de acuerdo a este sistema binario. Cuando se recibe una señal el,ctrica alta, se interpreta como que ha llegado un dato de valor (1). Cuando la señal es baja, el dato es un (0).
Todo el flujo de datos en el interior del ordenador, y del ordenador con los periféricos, se realiza mediante estas informaciones el,ctricas.
Para representar cadenas numéricas, se emplean cadenas de señales el,ctricas. Así por ejemplo, para representar el número 10001101 (base 2), el ordenador utilizaría la cadena de señales el,ctricas: Tensión alta, Tensión baja, Tensión baja, Tensión baja, Tensión alta, Tensión alta, Tensión baja, Tensión alta.
El factor de escala en esta base, son las potencias de 2 que afectan a un dígito dado dependiendo de su posición en la cadena num,rica.
Obs,rvese que al decir potencias de 2, me estoy refiriendo a potencias de 2 (en base 10). Es decir, para obtener la traducción de ese número en base 2 a su valor correspondiente en base 10, utilizamos las potencias de 2 mencionadas. Estas potencias de 2 en base 10, serían potencias de 10 en base 2. Es decir, el número 10 en base 2 equivale al número 2 en base 10.
Ve moslo mas claro. El número 10100101 se puede traducir a base 10 como: 10100101 = (1*2^7)+(0*2^6)+(1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0). O lo que es lo mismo: 10100101 (base 2) = 128+0+32+0+0+4+0+1 (base 10) = 165 (base 10)
* Base hexadecimal (Base 16). Como hemos mencionado al principio de la lección, la base hexadecimal surgió para compactar la información binaria. Se utiliza un dígito hexadecimal para representar una cadena de 4 dígitos binarios. Teniendo en cuenta que con 4 dígitos binarios podemos representar 16 números diferentes: 0,1,10,11,100,101,110,111,1000,1001,1010, etc... ...Teniendo en cuenta esto, un dígito hexadecimal tiene que poder tomar 16 valores diferentes. Para la base 10, tenemos 10 dígitos diferentes: del 0 al 9; para la base 2, nos servimos de dos de esos dígitos que ya teníamos para la base 10: el 0 y el 1. Pero en la base 16, que tenemos 16 dígitos diferentes, no podemos valernos sólo de los dígitos de la base decimal, ya que sólo hay

10 diferentes, y necesitamos 16. La solución es utilizar letras para representar los 6 dígitos que nos faltan.
Tenemos entonces que los dígitos hexadecimales son: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E y F.
A equivale a 10 en base 10. B equivale a 11 en base 10. C equivale a 12 en base 10. D equivale a 13 en base 10. E equivale a 14 en base 10. F equivale a 15 en base 10.
Del mismo modo que en la base 10, el último dígito es el 9; en la base 2, el último dígito es el 1; en la base 16, el último dígito ser F. Si sumamos a F una unidad, obtendremos el número 10 (base hexadecimal). Este número 10h (se utiliza el sufijo 'h' para indicar que se trabaja con base hexadecimal, al igual que el sufijo 'b' indica que se est trabajando con base binaria) equivale a 16 en base 10.
El factor de escala en esta base, son las potencias de 16 que afectan a un dígito dado dependiendo de su posición en la cadena num,rica.
De forma similar que al hablar de la base binaria, al decir potencias de 16, me estoy refiriendo a potencias de 16 (en base 10). Es decir, para obtener la traducción de ese número en base 16 a su valor en base 10, utilizamos las potencias de 16 mencionadas. Estas potencias de 16 en base 10, serían potencias de 10 en base 16. Es decir, el número 10 en base 16 equivale al número 16 en base 10.
¨Difícil de entender? Vamos a ver si esto lo aclara: El número AF34h se puede traducir a base 10 como: AF34 (base 16) = (10*16^3)+(15*16^2)+(3*16^1)+(4*16^0) (base 10). O lo que es lo mismo: AF34 = (10*4096)+(15*256)+(3*16)+4 = 40960+3840+48+4 = 44852
Hemos dicho que la base hexadecimal tiene como finalidad compactar la información binaria. 4 dígitos binarios se puden compactar en un sólo dígito hexadecimal. Tomemos por ejemplo el número 1010000101010101b. Nos es mas f cil indicar este número mediante su correspondiente número en base hexadecimal. Compactamos entonces toda esa cadena de información binaria en sólo 4 dígitos de información en base hexadecimal.
El proceso para llevar a cabo este cambio es sencillo. De derecha a izquierda de la cadena num,rica, se van cogiendo cadenas de 4 dígitos binarios, y se transforman a su correspondiente dígito hexadecimal. Cojamos los primeros 4 dígitos binarios: 0101. 0101 (base 2) = (0*8) + (1*4) + (0*2) + (1*1) = 5 (base 16). Siguiendo el proceso con el resto de la cadena, tenemos que el número resultante en base 16 es A155h, que es mucho mas f cil de recordar y almacenar (en cuanto a código fuente se refiere) que el correspondiente en base 2.
En caso de que el número en binario tenga menos de 4 dígitos, se rellenan las posiciones que faltan hacia la izquierda con ceros. Es decir, si tenemos el número 100101b, al pasarlo a base hexadecimal, tenemos el dígito de las unidades 5 (0101b), y para el dígito de las decenas tenemos que encontrar el correspondiente hexadecimal a la cadena 10b, que es lo mismo que 0010b, O sea 2 en hexadecimal. Tenemos entonces que 100101b = 25h.
Veamos una muestra de números en las tres bases mencionadas para ver sus equivalencias:

³ Decimal Binario Hexadecimal
³ ------- ------- -----------
³ 0 0 0
³ 1 1 1
³ 2 10 2
³ 3 11 3
³ 4 100 4
³ 5 101 5
³ 6 110 6
³ 7 111 7

³ 8 1000 8
³ 9 1001 9
³ 10 1010 A
³ 11 1011 B
³ 12 1100 C
³ 13 1101 D
³ 14 1110 E
³ 15 1111 F
³ 16 10000 10
³ 32 100000 20
³ 40 101000 28
³ 64 1000000 40
³
³ [...] [.....] [...]
* Base octal (Base 8). Al igual que la base hexadecimal, se utiliza para compactar información binaria, pero en este caso, la compactación es menor, de tal manera que casi no se usa. Mientras que en la base hexadecimal con un sólo dígito se puede representar una cadena de 4 dígitos binarios, en la base octal un dígito sólo puede representar 3 dígitos binarios. Los dígitos posibles para la base octal, evidentemente, son los que van del 0 al 7.
No profundizaremos mas en esta base, ya que es totalmente similar a la base 16, y no se suele utilizar.
- CAMBIO DE BASE ---------- A continuación se detalla el procedimiento para obtener el equivalente de un número en cualquiera de las bases expuestas.
Voy a prescindir de dar la teoría del m,todo, para verlo directamente en la pr ctica. Tomemos un número dado en cada una de las diferentes bases. Por ejemplo el número 18732 en base decimal, que es 492C en base hexadecimal, y 100100100101100 en base 2. Veamos cómo se llega de uno de esos números a otro, al cambiar de base.
* Cambio de base 2 a base 10.
100100100101100b = (1*2^14)+(1*2^11)+(1*2^8)+(1*2^5)+(1*2^3)+(1*2^2)= = 16384+2048+256+32+8+4 = 18732 (base 10).
* Cambio de base 10 a base 2.
³ 18732:2
³ ³0³ 9366:2
³ À-Ù ³0³ 4683:2
³ _ À-Ù ³1³ 2341:2
³ _ À-Ù ³1³ 1170:2
³ _ À-Ù ³0³ 585:2
³ _ À-Ù ³1³ 292:2
³ _ À-Ù ³0³ 146:2
³ _ À-Ù ³0³ 73 :2
³ _ À-Ù ³1³ 36 :2
³ _ À-Ù ³0³ 18 :2
³ _ À-Ù ³0³ 9 :2
³ _ À-Ù³1³ 4 :2
³ _ À-Ù³0³ 2 :2
³ _ À-Ù³0³ 1 :2
³ _ À-Ù³1³ 0
³ _ À-Ù

Partiendo del último resto de las sucesivas divisiones, y hasta llegar al primero, obtenemos: 100100100101100b, que es el equivalente en base 2 del número 18732 en base 10.
* Cambio de base 2 a base 16.
100100100101100b = 100 1001 0010 1100 = 492C en base 16. (4) (9) (2) (C)
* Cambio de base 16 a base 2.
492Ch = 0100 1001 0010 1100 = 100100100101100 en base 2.
* Cambio de base 16 a base 10. 492Ch = (4*16^3)+(9*16^2)+(2*16^1)+(12*16^0)= = (4*4096)+(9*256)+(2*16)+(12) = = 16384+2304+32+12 = 18732 en base 10.
* Cambio de base 10 a base 16.
³ 18732 :16
³ ³12 ³ 1170 :16
³ ³(C)³ ³2³ 73 :16
³ À---Ù_ À-Ù ³9³ 4 :16
³ _ À-Ù ³4³ 0
³ _ À-Ù
Partiendo del último resto de las sucesivas divisiones, y hasta llegar al primero, obtenemos: 492Ch, que es el equivalente en base 16 del número 18732 en base 10.
Por supuesto, para automatizar el proceso de cambio de bases, existen calculadoras especiales, que permiten trabajar con diferentes bases, permiten representar en cada una de esas bases, realizar operaciones lógicas con los números, etc. En la lección anterior se ha hablado de este tipo de programas, así que no har, mas incapi, en lo sucesivo.
- REPRESENTACION NUMERICA. ---------- A continuación vamos a ver las dos formas b sicas de trabajar con valores números en ensamblador. Dependiendo de que trabajemos sólo con números naturales (enteros positivos, incluido el Cero), utilizaremos el sistema de representación binario puro. Si trabajamos con números enteros en general (positivos y negativos), se utiliza el complemento a 2.
Ademas de estos dos sistemas de representación, veremos un tercero: El famoso BCD, que sirve para codificar y decodificar r pidamente información decimal (con la que estamos familiarizados) en una base binaria a la que no estamos tan acostumbrados.
Aparte de los tres sistemas de representación num,rica mencionados, hay varios mas, como son: signo-magnitud, complemento a 1, etc, etc... que no vamos a tratar ahora, ya que o bien unos no son usados en el Pc, o bien son tan complejos de usar que se salen de la finalidad de este curso, al menos por el momento.
* Coma fija sin signo (Binario Puro). Es el sistema usado para representar números enteros positivos. Este es un sistema de representación posicional con base 2 (binario), sin parte fraccionaria, y que sólo admite números positivos.
Mediante este sistema se pueden representar (para un tipo de dato de longitud N) todos los enteros positivos desde 0 hasta (2^N)-1. Tenemos entonces que su rango es [0..(2^N)-1] y su resolución es la unidad (1), ya que trabajamos con enteros.
Antes de estudiar este sistema con los tipos de datos propios del ensamblador, y por tanto del ordenador, vamos a hacer un estudio del mismo con un tipo de dato general, para así comprender perfectamente y sin lugar a dudas la base de este sistema, sus características, etc.
Tomemos una cadena num,rica de longitud 2, es decir, dos dígitos. Con este sistema vamos a poder representar todos los enteros positivos en el rango [0..(2^2)-1] = [0..3]. Es decir, mediante esta cadena num,rica de 2 bits de longitud, podremos representar 4 números enteros positivos diferentes:
³ Cadena num,rica: XX ³ -- ³ Números posibles: 00 ³ : 01 ³ : 10 ³ : 11
Si la cadena num,rica fuese de 1 sólo dígito, sólo podríamos representar 2 números distintos: 0 y 1. En definitiva se trata de un sólo bit.

Tomemos ahora una cadena de 3 bits o dígitos de longitud (N=3). En este caso podremos representar todos los enteros positivos en el rango [0..(2^3)-1] = [0..7]. Es decir, todos los números enteros comprendidos entre 0 y 7.
Veamos ahora este sistema de representación con los 3 tamaños de datos b sicos en el Pc:
- Tamaño Byte (8 bits). La longitud de este tipo de dato es de 8 bits, es decir, N=8. Por tanto en este tipo de datos (y con el sistema de representación Binario Puro), vamos a poder representar enteros comprendidos en el Rango [0..(2^8)-1] = [0..255]. Esto quiere decir que en un registro o posición de memoria de tamaño byte (8 bits), vamos a poder tener 256 valores diferentes.
- Tamaño Word ó palabra (16 bits). La longitud de este tipo de dato es de 16 bits. Por tanto en este tipo de datos (y con el sistema de representación Binario Puro), vamos a poder representar enteros comprendidos en el Rango [0..(2^16)-1] = [0..65535]. Esto quiere decir que en un registro o posición de memoria de tamaño palabra (16 bits), vamos a poder tener 65536 valores diferentes. O sea, que el mayor número que se podr representar en este tamaño de dato, con este sistema de representación es el 65535; y el menor número representable ser el 0.
- Tamaño DWord, Double Word, ó Doble palabra (32 bits). La longitud de este tipo de dato es de 32 bits. Por tanto en este tipo de datos (y con el sistema de representación Binario Puro), vamos a poder representar enteros comprendidos en el Rango [0..(2^32)-1] = [0..4294967295].
* Complemento a 2. El sistema de representación Complemento a 2, es el usado por el Pc (entre otras cosas) para poder realizar sumas y restas con números enteros sin tener que hacer comprobaciones del signo de los operandos. Es decir, la instrucción ADD AX,BX se ejecutar igual si los números son positivos, que si son negativos, que si uno es positivo y otro negativo.
Veamos las características de este sistema de representación desde un punto de vista pr ctico, para entenderlo mejor y mas r pido:
Supongamos que estamos trabajando con el tipo de dato Byte (8 bits), luego N (longitud de la cadena num,rica) es igual a 8. Si estuvi,ramos trabajando sólo con números enteros positivos (binario puro), el registro ó posición de memoria admitiría 256 valores positivos. Pero estamos utilizando el sistema de Complemento a 2, porque vamos a trabajar con enteros en general, positivos y negativos. Luego esos 256 valores han de dividirse en 2 grupos: uno para los positivos, y otro para los negativos.
Para los números positivos se reservan los códigos que tengan el bit 7 (bit mas significativo, ya que estamos trabajando con datos de 8 bits) con valor 0. Es decir, los códigos 00000000 hasta 01111111.
El resto de códigos posibles se utilizan para representar los números negativos. Es decir, los códigos 10000000 hasta 11111111.
Es f cil determinar de esta forma si un número es positivo o negativo. Si es positivo, su bit mas significativo valdr 0. Si es negativo, su bit mas significativo valdr 1.
Hemos visto los códigos reservados para cada grupo de números, los positivos y los negativos. Veremos a continuación las diferencias en la representación entre los positivos y los negativos:
Los números positivos se representan en binario puro, como hemos visto en el apartado anterior. Es decir, si queremos representar el número 37 en un registro de tipo byte, quedaría de la forma 00100101. Como podemos observar, al utilizar números positivos en Complemento a 2, estamos utilizando la representación en Binario Puro.
Hay que tener en cuenta el mayor número positivo representable en este tipo de dato (8 bits) y con este sistema de representación. Es decir, no se pude representar números

positivos mas all del 01111111b, por tanto el mayor número positivo representable es el 127 para este tamaño de dato byte.
Todo lo expuesto hasta ahora (y a continuación) para el tipo de dato byte, es extensivo para el resto de tipos de datos (palabra y doble palabra), teniendo en cuenta su diferente longitud (N), obviamente.
En general, el rango de representación de los números positivos en el sistema de complemento a 2 es: [0..(2^(N-1))-1]
³ En el caso de dato de tipo byte: [0..(2^(8-1))-1]
³ [0..(2^7)-1]
³ [0..128-1]
³ [0..127]
³
³ Para el tipo de dato word: [0..(2^(16-1))-1]
³ [0..(2^15)-1]
³ [0..32768-1]
³ [0..32767]
³
³ Etc....
Con los números negativos es cuando sí se utiliza el Complemento a 2. Para representar un número negativo hay que realizar el complemento del número de la siguiente manera: Hay que restar el módulo del número negativo a representar de 2^N.
Por ejemplo: Para ver cómo quedaría el número -108 en un registro de tipo byte, haríamos lo siguiente: Tenemos una longitud de dato de 8 bits, luego N=8 y por tanto, 2^N=256. Ahora restamos a 256 el módulo de (-108). Es decir, le quitamos a 108 su signo, y lo restamos de 256.
³ 256 100000000
³ -108 - 1101100
³ ---- ---------
³ 148 10010100
³ ³
³ À- Observamos que el bit de mas a la
³ izquierda tiene valor 1, indicando
³ que se trata de un número negativo.
A estas alturas, podremos apreciar que para representar un número negativo en complemento a 2, no introducimos ese número en el registro ó posición de memoria, sino su complemento con respecto a 2^N.
De esta manera, todos los números negativos tendr n su bit mas significativo con valor 1, indicando su condición de número negativo.
Para obtener el rango de los números negativos, tenemos que calcular el mínimo y el m ximo representable. Tomemos el tipo de dato byte, para concretar: El mas pequeño número negativo es -1, y su complemento es 11111111. Como en los números negativos, el bit mas significativo (bit 7 en este caso) debe ser 1, el m ximo negativo representable ser el m ximo al que se le pueda hacer el complemento sin que el bit mas significativo sea 0, ya que entonces sería positivo. Llegamos entonces a la conclusión de que tal m ximo es -128, con el complemento 10000000b.
En general, el rango de representación de los números negativos en el sistema de complemento a 2 es: [-(2^(N-1))..-1]
³ En el caso de dato de tipo byte: [-(2^(8-1))..-1]
³ [-(2^7)..-1]
³ [-128..-1]
³

³ Para el tipo de dato word: [-(2^(16-1))..-1]
³ [-(2^15)..-1]
³ [-32768..-1]
³
³ Etc....
El rango completo de representación (negativos y positivos) en el sistema de Complemento a 2 es: [-(2^(N-1))..(2^(N-1))-1]. Siendo N, la longitud del dato.
Veamos algunos ejemplos de representación para comprenderlo definitivamente...
Ejemplo: Cómo se representar el número -7 en un registro de tamaño word? Tamaño word = 16 bits de longitud, luego N=16, y por tanto 2^N=65536.
³ 65536 10000000000000000
³ - 7 - 111
³ ----- -----------------
³ 65529 1111111111111001
³ ³
³ À- Bit mas significativo con valor 1, indicando
³ número negativo.
Ejemplo curioso: Supongamos que estamos trabajando con un tipo de dato byte. El número 10000000b representado en binario puro es 128, y representado en Complemento a 2, sería -128. Probad a realizar el complemento y obervar,is que el complemento de 10000000b es el mismo 10000000b (con N=8, claro).
Como creo que para alguien que ha visto esto por primera vez, le resultar difícil de entender, y es posible que con lo expuesto hasta ahora no lo tenga del todo claro, me extender, un poco mas:
Lo que viene a continuación es el quid de la cuestión de la representación en Complemento a 2.
Tomemos el registro AL, por ejemplo. Si AL = 10010101, tenemos dos números diferentes dependiendo de que estemos teniendo en cuenta los números negativos o no. Esto es, dependiendo de si estamos trabajando en binario puro o en complemento a 2.
Es decir, en binario puro, el número 10010101 es 149 en base decimal. Mientras que si estamos bas ndonos en el Complemento a 2, el número que tendríamos ya no sería un 149, sino que sería un número negativo por tener su bit mas significativo con valor 1. Mas concretamente se trataría del número -107. Para obtener el número negativo del que se trataba, hacemos el complemento a 2. Como el número ya estaba en complemento a 2, complemento con complemento se anulan, y obtenemos el original número negativo que es -107.
Es decir, 256-149=107. Ahora le ponemos el signo, y tenemos el -107.
Hemos visto, pues, que dependiendo de que tengamos en cuenta (Complemento a 2) ó No (Binario Puro) el signo de un número, ,ste podr tener 2 valores diferentes, el número positivo, y su complementario negativo.
Pero esto sólo sucede, obviamente, con los números que tienen el bit mas significativo con valor 1. Si tomamos por ejemplo el número 3, dar igual que estemos trabajando con binario puro o complemento a 2, tendremos en ambos casos el número positivo 3, ya que el bit mas significativo al ser 0, no deja la posibilidad de ser un número negativo.
Hemos dicho arriba que el ordenador utiliza el sistema de Complemento a 2 por la comodidad que le supone para realizar las sumas y las restas. Veamos cómo lleva a cabo el ordenador estas operaciones, y así entenderemos mejor el porque de este sistema de representación.
Supongamos que queremos sumar los registros AL y CL. Estamos trabajando en nuestro programa con números positivos y negativos, con lo cual el sistema de representación que tenemos que tener presente es el de Complemento a 2.

D,mosle valores a AL y CL: AL=-37. CL=120. Tenemos pues un número negativo y otro positivo. Veamos cómo se representan ambos números en sus respectivos registros, teniendo en cuenta las características de la representación en Complemento a 2.
AL=11011011 CL=01111000
Como podemos observar, en AL tenemos el número en Complemento a 2, mientras que en CL, el número quedaría como binario puro.
Tras realizar la suma (ADD AL,CL), en AL deber quedar el valor 83, positivo, por supuesto. Veamos si es así. Realicemos la suma:
³ AL=11011011
³ + CL=01111000
³ ----------
³ 101010011
³ ³
³ À--Este bit se desprecia, ya que se sale del registro de
³ 8 bits. Ocuparía una novena posición que no hay.
³ Tenemos entonces AL=01010011.
Veamos el equivalente de 01010011b en base decimal. Para que la operación sea correcta, deberíamos obtener el valor 83 = ((-37)+120). Haced ese cambio de base que os digo, y obtendreis el resultado que esper bamos (83).
Supongamos ahora que queremos sumar los registros AL y CL de nuevo, pero esta vez estamos trabajando con dos números positivos, con lo cual, el sistema de representación que nos interesa es el de Binario Puro.
En esta ocasión, CL tiene el mismo valor que antes, CL=120. Pero AL que ahora es un número positivo tiene un valor al azar, por ejemplo el valor 219.
AL=219 CL=120
Tenemos pues, dos números positivos. Veamos cómo se quedarían ambos números en sus respectivos registros.
AL=11011011 CL=01111000
---Que casualidad!!! :-) ---Pero si tienen los mismos valores que en el ejemplo anterior!!!
Tengamos en cuenta en este caso, que al realizar la suma vamos a obtener desbordamiento, ya que 120+219=339 que sobrepasa el valor 255 (m ximo número representable en binario puro para un tipo de dato de 8 bits). Ese 339 en binario sería 101010011. Los bits que caben en el registro son los 8 de la derecha: 01010011, que en base decimal es 83!!!
Veamos si es así. Realicemos la suma:
³ AL=11011011
³ + CL=01111000
³ ----------
³ 101010011
³ ³
³ À--Este bit se desprecia, ya que se sale del registro de
³ 8 bits. Ocuparía una novena posición que no hay.
³ Esto lo indica el procesador mediante el flag de Overflow
³ Of.
³ Y el registro nos queda con el contenido AL=01010011.
01010011b en base decimal es 83 como hemos indicado arriba.
Mediante estos dos casos pr cticos, llegamos a la conclusión del valor que tiene el sistema de representación Complemento a 2. Os sugiero que si aún queda algún pice de duda en cuanto a este sistema de representación usado por el ordenador para representar los números negativos, os pongais vuestros propios ejemplos: Suma de

número negativo y número negativo. Resta de número negativo y número positivo. Resta de número negativo y número negativo. Etc...
De esta forma, se comprender de forma pr ctica el sistema de representación num,rica que usa el Pc para estos números enteros, y el porque del mismo.
Para finalizar con el apartado de Complemento a 2, veamos unos ejemplos de representación para unos valores dados, en los que se comparan las representaciones Binario Puro y Complemento a 2. En este caso, estos valores se van a encontrar almacenados en un registro de 8 bits (tamaño byte), por ejemplo AL.

Contenido de AL Binario Puro Complemento a 2
---------- ---------- ---------- 00000000 0 0 00000001 1 1 00000010 2 2 00000101 5 5 00100000 32 32 00111101 61 61 01111111 127 127 10000000 128 -128 10000001 129 -127 10000111 135 -121 10111111 191 -65 11111110 254 -2 11111111 255 -1

* BCD (Sistema Decimal codificado en Binario). Este sistema BCD, sirve (como hemos mencionado antes) para codificar y decodificar r pidamente información decimal en una base binaria.
Se utiliza un byte para almacenar cada dígito decimal. Es decir, mediante este m,todo no se realiza la traducción de una cadena num,rica en base decimal a la base binaria propia del ordenador, sino que se almacena como una cadena de bytes, uno de ellos para cada dígito de la cadena decimal introducida.
Mientras que el número 133 en base decimal se almacena en un sólo byte utilizando el sistema de representación Binario Puro, necesitar 3 bytes para ser representado en BCD. La representación en BCD sería la cadena de bytes '1','3','3'. Es decir, los bytes con código ASCII 31h,32h,33h.
El número 23849 necesitaría sólamente 2 bytes (una palabra) para ser representado en Binario Puro, mientras que en BCD necesitaría 5 bytes, uno para cada dígito decimal. 32h,33h,38h,34h,49h.
Lo expuesto hasta ahora sobre el sistema BCD, tiene un car cter orientativo. En esta lección no abordaremos en detalle este sistema de representación, ya que no me parece interesante. De cualquier modo, en una futura lección le prestaremos mas atención a este sistema, simplemente para que lo conozcais mejor. Eso os lo dejo a vuestra elección.

ASM POR AEsoft. (lección 10). ---------- - funciones DOS: - INT 21H (DOS-API)

- funciones BIOS: - INT 10H (FUNCIONES DEL DRIVER DE VIDEO) - INT 16H (FUNCIONES DEL DRIVER DE TECLADO) ----------
Hola de nuevo a todos los seguidores del CURSO DE ASM.
En las lecciones 7 y 8 hemos visto las mas importantes instrucciones con que contamos en Ensamblador del 8086. En próximas lecciones iremos viendo el resto de instrucciones, según las vayamos necesitando.
En esta lección vamos a ver algo tan importante como ese conjunto de instrucciones. vamos a ver las principales interrupciones software (funciones) que tenemos disponibles para usar en nuestros programas.
Mediante la llamada a una de estas funciones podemos leer un fichero, cambiar de modo de vídeo, aceptar un car cter desde teclado, etc, etc.

Estas interrupciones software nos permiten trabajar con los distintos subsistemas (teclado, vídeo, discos duros y disqueteras, etc..) de una forma relativamente cómoda. Nos ofrecen las rutinas b sicas para trabajar con ellos.
Para cada uno de los tres servicios mas importantes de cara al programador (funciones DOS, funciones de pantalla y funciones de teclado), vamos a enumerar las funciones fundamentales para empezar a trabajar.
Repito, la lista de funciones que se expone a lo largo de la lección, no es completa. Para eso existen libros y manuales especiales. Al final de la lección se ofrece bibliografía suficiente.
Conforme vaya avanzando el nivel del curso iremos viendo nuevos servicios como la INT 33H (controlador del ratón), la INT 13H (Controlador de disco), etc...

- funciones DOS ---------- Las funciones DOS son todas aquellas interrupciones software de las que est provisto el Sistema Operativo. Estas funciones, entre las que se encuentra la INT 21H (la mas importante de todas), son utilizadas por el programador, y por el propio sistema operativo para acceder a los distintos subsistemas del procesador como son discos, teclado, etc...
Usaremos estas funciones cuando queramos: crear ficheros, borrar ficheros, leer ficheros, solicitar memoria libre para trabajar con los datos de nuestros programas, dejar programas residentes, etc, etc...
En definitiva, las funciones del DOS nos proporcionan un vínculo de comunicación cómodo y seguro entre nuestro programa y los diferentes subsistemas con los que podemos trabajar.
Veremos que esto también nos lo proporcionan las funciones BIOS, pero a otro nivel mas bajo. Por ejemplo, mientras que las funciones DOS nos permiten trabajar con ficheros, las funciones BIOS sólo nos permiten trabajar con discos en función a pistas, sectores, etc. Es decir, mas a bajo nivel.
En una próxima lección veremos la estructura interna de los discos: Tanto la estructura física: cabezas (caras), cilindros (pistas), sectores.... Como la estructura lógica que usa el MS-DOS: BOOT RECORD, FAT, DIRECTORIO, CONTENIDO DE FICHEROS. ³ À----------¿ (sector de arranque) (tabla de localización de ficheros)
Y volviendo a lo que nos ocupa: las funciones BIOS nos facilitan el trabajo con la parte física: pistas, sectores, etc... Mientras que las funciones DOS, nos permiten trabajar con la parte lógica: ficheros.
Por supuesto, al trabajar con la parte lógica (ficheros), el DOS debe hacer uso de la parte física ó sectores (que es donde están almacenados los ficheros). Para este trabajo, las mismas funciones DOS usan las funciones BIOS.
Hemos dicho que la mas importante de las funciones DOS es la INT 21H. Pues bien, aparte de ,sta, hay algunas mas como son: - INT 20H (Terminación de proceso). - INT 22H (Dirección del gestor de terminación del programa en curso). - INT 23H (Dirección del gestor de CTRL+C). - INT 24H (Dirección del gestor de errores críticos). - INT 25H (Lectura de sectores de disco). - INT 26H (Escritura de sectores a disco). - INT 27H (Dejar programa residente y salir). - INT 28H a INT 2EH (Reservadas. No est permitido su uso al programador). - INT 2FH (Interrupción múltiple ó Interrupción del Multiplexor).
La INT 20H es una de las muchas maneras de finalizar un programa.
Las interrupciones 22H, 23H y 24H las estudiaremos en la próxima lección, cuando estudiemos la construcción de programas en ASM, y la gestión de los mismos por parte del DOS.
Las interrupciones 25H y 26H se utilizan para leer y escribir sectores de disco. Hemos visto antes que era la BIOS la que tenía este cometido. La razón de ser de estas dos funciones es que utilizan un diferente formato de referencia a los sectores. La BIOS trata

a los sectores del disco de forma física: Sector X del Cilindro Y de la Cabeza Z... Mientras que el DOS enumera secuencialmente todos los sectores del disco, desde (el sector 1 del cilindro 0 de la cabeza 0), hasta el último sector del disco. Referenciando así a cada sector mediante un sólo número de sector lógico.
La INT 27H es un m,todo antiguo de dejar programas residentes en memoria. No se suele utilizar. En su lugar, disponemos de la función 31h de la INT 21H, que veremos a continuación.
Las funciones reservadas han sido pensadas por los programadores del DOS, para uso exclusivo del mismo DOS, o son funciones vacías (sin contenido) en previsión para un aumento de funciones en una próxima versión del DOS.
La interrupción 2FH ó interrupción del Multiplexor, proporciona información acerca del estado de ciertos programas residentes del DOS, como son DOSKEY, PRINT, APPEND, SHARE, etc. Podremos saber si están instalados o no, y en caso de que est,n instalados, la INT 2FH nos sirve como un interfaz ó vínculo de comunicación con estos programas residentes.
- INT 21H (DOS-API) ---------- La mayoría de servicios ó funciones del sistema operativo MS-DOS se obtienen a través de la interrupción software 21H. Es por esto que se le denomina DOS-API: DOS-APPLICATION-PROGRAM-INTERFACE, ó Interfaz entre programas de aplicaciones y el MS-DOS.
La INT 21H est compuesta por un grupo de funciones. Cuando se accede a la INT 21H, hay que indicar el número de función que queremos ejecutar. En determinadas casos, una función es tan compleja que necesita de varias subfunciones para poder desempeñar todo su trabajo. En este caso, deberemos indicar el número de esa función, y también el número de la subfunción a la que queremos acceder dentro de esa función.
La llamada a la INT 21H se realizar como sigue:
+ Introducimos en (AH) el número de función a la que deseamos acceder. + En caso de que deseemos acceder a una subfunción dentro de una función, debemos indicarlo introduciendo en (AL) el número de esa subfunción. + Llamar a la INT 21H.
Aparte de los registros mencionados (AH y AL), la mayoría de funciones requieren de otros registros para realizar su trabajo. En estos registros se puede almacenar el comienzo de un buffer de disco al trabajar con ficheros; el comienzo de una cadena de texto para sacar por pantalla; etc..
Un ejemplo: Queremos usar la función 9h de la INT 21H para sacar por pantalla una cadena de texto. Dicha cadena de texto se encuentra almacenada a continuación de la etiqueta Cadena_De_Texto.
Procederemos como sigue:
Mov AH,9 ;indicamos número de función a ejecutar. Mov DX,offset cadena_de_texto ;introducimos en DX la dirección donde ;se encuentra la cadena en cuestión. INT 21H ;llamamos a la INT 21H, la cual ejecutar la función 9h.
A continuación vamos a ver las funciones b sicas dentro de cada grupo para empezar a programar en ensamblador. No las vamos a ver todas, porque muchas de ellas ya no se usan, como son las funciones de acceso a ficheros por el m,todo FCB (que veremos cuando hablemos de la estructura de los discos). Tampoco veremos en un principio aquellas funciones que no vamos a utilizar hasta pasado un buen tiempo, como son funciones relacionadas con la información sobre el país, p gina de códigos, y unas pocas mas que sólo rara vez se utilizan.
De cualquier modo, siempre es interesante poder conocerlas todas, así que si teneis oportunidad, hay por ahí manuales, libros, etc, en los que vienen todas las funciones detalladas.
ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ Funciones de la INT 21H ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
* Funciones de Entrada/Salida de caracteres *

Ú----------Â----------¿ ³ INT 21H Función 01H ³ ³ Ã----------´ ³ ³ Entrada de Car cter con Eco (ó salida) ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 01H ³ ³ ³ Ã DEVUELVE: ³ ³ AL = Código ASCII del Car cter leído. ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Se lee un car cter del dispositivo de entrada estándar, y se ³ ³ envía al dispositivo estándar de salida. Si al llamar a la ³ ³ función no había ningún car cter disponible, se esperar a que ³ ³ lo haya. ³ ³ ³ ³ Normalmente el dispositivo estándar de entrada es el teclado, ³ ³ y el dispositivo estándar de salida es la pantalla. Es lo que ³ ³ se conoce en MS-DOS como CON (de consola: teclado y pantalla). ³ ³ ³ ³ Tanto la Entrada como la Salida pueden ser redireccionadas. ³ ³ Ya veremos lo que eso significa con mas detalle. Sirva ahora que ³ ³ es posible que los caracteres se puedan tomar de un dispositivo ³ ³ diferente al teclado, y se puedan enviar a un dispositivo ³ ³ diferente de la pantalla. ³ ³ ³ ³ Es decir, que como entrada podemos tener los caracteres de un ³ ³ fichero, y como salida podríamos tener la impresora. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0, se toma el car cter desde teclado, y se ³ ³ envía hacia la pantalla. En la versión 1.0 del DOS, esta ³ ³ función no admitía ninguna redirección de Entrada/Salida. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 02H ³ ³ Ã----------´ ³ ³ Salida de Car cter ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 02H ³ ³ DL = Código ASCII a enviar al dispositivo de salida. ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Se envía el car cter depositado en el registro DL al dispositivo ³ ³ estándar de salida. ³ ³ ³ ³ La salida puede ser redireccionada hacia un fichero, impresora, etc ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0, se envía el car cter hacia la pantalla. ³ ³ En la versión 1.0 del DOS, esta función no admitía ³ ³ redirección de Salida. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 05H ³ ³ Ã----------´ ³ ³ Envío de Car cter a la Impresora ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 05H ³ ³ DL = Código ASCII a enviar al dispositivo de salida. ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ Se envía el car cter depositado en DL al puerto paralelo. ³ ³ Si no se ha redireccionado la salida, el dispositivo por defecto ³ ³ en el puerto paralelo de salida (LPT1 ó PRN) es la impresora. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0, se envía el car cter hacia el primer ³ ³ dispositivo de listado (PRN ó LPT1). ³ ³ En versiones posteriores de MS-DOS, se puede redireccionar ³ ³ la salida. ³ ³ ³ À----------Ù
Ú----------Â----------¿ ³ INT 21H Función 09H ³ ³ Ã----------´ ³ ³ Visualización de una cadena de caracteres ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 09H ³ ³ DS:DX = Segemento:Desplazamiento de la cadena a visualizar. ³ ³ DS debe apuntar al segmento donde se encuentra la cadena. ³ ³ DX debe contener el desplazamiento de la cadena dentro de ³ ³ ese segmento. ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ Se envía una cadena de caracteres al dispositivo estándar de salida.³ ³ Si no se ha redireccionado la salida, la cadena se enviar a la ³ ³ pantalla. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: La cadena debe finalizar con un car cter $ (24H), para que ³ ³ el DOS pueda reconocer el fin de la cadena. ³ ³ ³ À----------Ù
Ú----------Â----------¿ ³ INT 21H Función 0BH ³ ³ Ã----------´ ³ ³ Comprobación del estado de la entrada ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 0BH ³ Ã DEVUELVE: ³ ³ AL = 00H si no hay car cter disponible. ³ ³ AL = FFH si hay algún car cter disponible. ³ ³ ³ Ã EFECTO: ³ ³ Se

comprueba si hay algún car cter procedente del dispositivo ³ ³ estándar de entrada. La entrada puede ser redireccionada. ³ ³ Si no hay tal redirección, se comprueba el buffer de teclado. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: En caso de que haya un car cter disponible, sucesivas ³ ³ llamadas a esta función seguir n dando un valor verdadero ³ ³ hasta que se recoja el car cter con alguna de las funciones ³ ³ de entrada de car cter, como la función 01h. ³ ³ ³ À----------Ù

* Funciones de manejo de disco *

Ú----------Â----------¿ ³ INT 21H Función 0EH ³ ³ Ã----------´ ³ ³ Selección de disco ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 0EH ³ ³ DL = Código de la unidad (0=A , 1=B , etc...) ³ Ã DEVUELVE: ³ ³ AL = Número de unidades lógicas del sistema. ³ ³ ³ Ã EFECTO: ³ ³ Selecciona una unidad de disco para que se convierta en la unidad ³ ³ por defecto o actual. Ofrece ademas información acerca del número ³ ³ total de unidades lógicas del sistema. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: Una unidad física como un disco duro puede estar particionada³ ³ en varias unidades lógicas designadas por C , D, E ,etc. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 19H ³ ³ Ã----------´ ³ ³ Obtener disco actual ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 19H ³ Ã DEVUELVE: ³ ³ AL = Código de la unidad actual (0=A , 1=B , etc...) ³ ³ ³ Ã EFECTO: ³ ³ Devuelve el código de la unidad de disco activa o por defecto. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: Las mismas que para la función 0EH ³ ³ ³ À----------Ù

* Funciones de gestión de directorios *

Ú----------Â----------¿ ³ INT 21H Función 39H ³ ³ Ã----------´ ³ ³ Crear directorio ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 39H ³ ³ DS:DX = Segemento:Desplazamiento de una cadena ASCIIZ ³ ³ con el nombre del directorio. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Se crea un nuevo directorio usando la unidad de disco y la vía de ³ ³ acceso especificada en la cadena ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0 del MS-DOS no existían subdirectorios. ³ ³ ASCIIZ es una cadena de códigos ASCII que termina con el ³ ³ código ASCII 0h. ³ ³ Así, si queremos crear un subdirectorio llamado PROGS dentro ³ ³ del directorio DOS, la cadena ASCIIZ se definiría así: ³ ³ Nombre_del_directorio db 'C:\DOS\PROGS',0 ³ ³ Si en el momento de llamar a la función, la unidad activa es ³ ³ la unidad C, no es necesario indicarlo. De igual forma, si ³ ³ nos encontramos en el directorio DOS, tampoco necesitamos ³ ³ indicarlo. ³ ³ ³ ³ Esta función se abortar (indic ndolo mediante Cf y AX) si: ³ ³ -la vía de acceso es incorrecta. ³ ³ -el directorio a crear ya existe. ³ ³ -el directorio a crear es un subdirectorio del Raiz, ³ ³ y ,ste ya est lleno. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 3AH ³ ³ Ã----------´ ³ ³ Borrar directorio ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 3AH ³ ³ DS:DX = Segemento:Desplazamiento de una cadena ASCIIZ ³ ³ con el nombre del directorio a borrar. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Se elimina el directorio indicado de la unidad de disco y la vía ³ ³ de acceso especificada en la cadena ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0 del MS-DOS

no existían subdirectorios. ³ ³ ³ ³ Esta función se abortar (indic ndolo mediante Cf y AX) si: ³ ³ -la vía de acceso es incorrecta. ³ ³ -el directorio especificado es el directorio activo. ³ ³ -el directorio especificado no est vacío, es decir, ³ ³ contiene algún fichero. ³ ³ ³ À----------
----------Ù

Ú----------Â----------¿ ³ INT 21H Función 3BH ³ ³ Ã----------´ ³ ³ Establecer directorio actual ³ ³ Ã----------
------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 3BH ³ ³ DS:DX = Segemento:Desplazamiento de una cadena ASCIIZ ³ ³ con el nombre del directorio. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Establece como directorio actual el indicado mediante la cadena ³ ³ ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0 del MS-DOS no existían subdirectorios. ³ ³ ³ ³ Esta función se abortar (indic ndolo mediante Cf y AX) si ³ ³ la vía de acceso especificada en la cadena ASCIIZ es ³ ³ incorrecta. ³ ³ ³ À----------
-------Ù

Ú----------Â----------¿ ³ INT 21H Función 47H ³ ³ Ã----------´ ³ ³ Obtener directorio actual ³ ³ Ã----------
---Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 47H ³ ³ DL = Código de la unidad (0 = unidad por defecto, actual; ³ ³ 1 = A ; 2 = B ; etc...) ³ ³ DS:SI = Segmento:Desplazamiento de un buffer de 64 bytes. ³ ³ Este buffer contendr el nombre del directorio, con ³ ³ toda la vía de acceso al mismo, en forma de cadena ³ ³ ASCIIZ. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Establece como directorio actual el indicado mediante la cadena ³ ³ ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: En la versión 1.0 del MS-DOS no existían subdirectorios. ³ ³ ³ ³ Esta función se abortar (indic ndolo mediante Cf y AX) si ³ ³ el código de unidad no es v lido. Es decir, no se especifica ³ ³ una unidad v lida. ³ ³ ³ ³ La vía de acceso que antecede al nombre del directorio no ³ ³ incluye el código '\' (directorio raiz), ni el identificador ³ ³ de la unidad. ³ ³ El nombre de directorio acaba con el car cter 00h que cierra ³ ³ la cadena ASCIIZ. ³ ³ ³ À-------
----------Ù
* Funciones de manejo de Ficheros *
Ú----------Â----------¿ ³ INT 21H Función 3CH ³ ³ Ã----------´ ³ ³ Crear Fichero ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 3CH ³ ³ CX = Atributos del fichero: ³ ³ 00H Fichero Normal. ³ ³ 01H Fichero de Sólo Lectura. ³ ³ 02H Fichero Oculto. ³ ³ 03H Fichero de Sistema. ³ ³ DS:DX = Segmento:Desplazamiento de una cadena ASCIIZ con ³ ³ el nombre de fichero. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ AX = Handle o manejador de fichero. ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Si el fichero indicado mediante la cadena ASCIIZ ya existía, ³ ³ entonces se vacía su contenido, quedando con longitud 0. ³ ³ Si el fichero no existía, entonces se crea. ³ ³ En ambos casos, el fichero se abre, y se devuelve un Handle ó ³ ³ manejador de fichero para los restantes accesos a dicho fichero. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Esta función se abortar (indic ndolo mediante Cf y AX) si: ³ ³ -La vía de acceso indicada en la cadena ASCIIZ es errónea. ³ ³ -Se va a crear el fichero en el directorio raiz y ,ste ya ³ ³ est lleno. ³ ³ ³ ³ El Handle es un número mediante el cual se referencia a un ³ ³ fichero dado. De esta forma es mas cómodo trabajar con ³ ³ ficheros. Para cada uno de los ficheros con que trabajemos, ³ ³ tendremos un número (Handle) que lo representar . ³ ³ ³ À----------
----------Ù

Ú----------Â----------¿ ³ INT 21H Función 3DH ³ ³ Ã----------´ ³ ³ Abrir Fichero ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 3DH ³ ³ AL = Modo de acceso. ³ ³ ³ ³ Bits 0-2: Permiso de lectura/escritura. ³ ³ 000b Acceso de sólo lectura. ³ ³ 001b Acceso de sólo escritura. ³ ³ 010b Acceso de lectura/escritura. ³ ³ ³ ³ Bit 3: 0b (Reservado). ³ ³ ³ ³ Bits 4-6: Modo de compartición de Fichero. ³ ³ 000b Sólo el programa actual puede acceder ³ ³ al archivo. ³ ³ 001b Sólo el programa actual puede acceder ³ ³ al archivo. ³ ³ 010b Otro programa puede leer el archivo, ³ ³ pero no escribir en el. ³ ³ 011b Otro programa puede escribir en el ³ ³ archivo, pero no leerlo. ³ ³ 100b Otro programa puede leer y escribir en³ ³ el archivo. ³ ³ ³ ³ Bit 7: Bit de 'herencia' ó Handle-flag. ³ ³ 0b Si el handle se hereda por un programa ³ ³ hijo. ³ ³ 1b Si el handle no se hereda. ³ ³ ³ ³ DS:DX = Segmento:Desplazamiento de una cadena ASCIIZ con ³ ³ el nombre de fichero. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ AX = Handle o manejador de fichero. ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función se habre un fichero ya existente, y se ³ ³ devuelve un Handle para acceder al fichero en lo sucesivo. ³ ³ ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ El puntero de fichero se coloca sobre el primer byte del ³ ³ fichero. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 3EH ³ ³ Ã----------´ ³ ³ Cerrar Fichero ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 3EH ³ ³ BX = Handle. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función se cierra un fichero que estuviera abierto. ³ ³ Se utiliza el Handle para indicar el fichero a cerrar. ³ ³ Tras cerrar el fichero, dicho Handle se libera para nuevos ficheros.³ ³ Se vuelvan al disco todos los buffers internos asociados al fichero.³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Si por error se llamara a esta función con el valor 0, se ³ ³ cerraría el dispositivo de entrada estándar (teclado), que ³ ³ tiene asociado ese handle 0. En cuyo caso no se aceptarían ³ ³ datos del teclado. ³ ³ Si el valor del handle fuera 1, se cerraría la pantalla, y ³ ³ no se enviarían caracteres a la pantalla. ³ ³ Hay en total 5 handles reservados para referenciar a ³ ³ diferentes dispositivos: ³ ³ ³ ³ Handle 0 ---> Dispositivo estándar de entrada. (CON). ³ ³ Handle 1 ---> Dispositivo estándar de salida. (CON). ³ ³ Handle 2 ---> Dispositivo estándar de error. (CON). ³ ³ Handle 3 ---> Dispositivo auxiliar estándar. (AUX). ³ ³ Handle 4 ---> Dispositivo estándar de listado. (PRN). ³ ³ ³ ³ Normalmente, el handle 0 referencia al teclado. ³ ³ El handle 1, a la pantalla. ³ ³ El handle 2, a la pantalla. Se utiliza a la hora de mostrar ³ ³ errores. Por eso lo de dispositivo de error. ³ ³ El handle 4, a la impresora. ³ ³ El handle 3 referencia a un dispositivo auxiliar. ³ ³ ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 3FH ³ ³ Ã----------´ ³ ³ Lectura de Fichero o dispositivo. ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 3FH ³ ³ BX = Handle. ³ ³ CX = Número de bytes a leer. ³ ³ DS:DX = Segmento:Desplazamiento del buffer donde se ³ ³ depositar n los carateres leidos. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ AX = Bytes transferidos. ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Dado un handle v lido, se realiza una transferencia desde el ³ ³ fichero referenciado por ese handle hacia el buffer de memoria ³ ³ especificado mediante DS:DX. Se transferir n tantos caracteres ³ ³ como se especifique en CX. Acto seguido, se actualiza el puntero ³ ³ de fichero hasta el car cter que sigue al bloque leido. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³

Si se devuelve el flag Cf con valor (0), pero AX=0, esto ³ ³ quiere decir que el puntero de fichero estaba apuntando ³ ³ al final de fichero, y por eso no se ha podido leer ningún ³ ³ car cter. ³ ³ Si se devuelve el flag Cf con valor (0), pero el contenido ³ ³ del registro AX es menor que la cantidad de bytes a leer, ³ ³ (indicado mediante CX antes de llamar a la función), esto ³ ³ significa que se produjo algún error, o que no se pudo leer ³ ³ todos los caracteres solicitados, porque se llegó al final ³ ³ de fichero. ³ ³ ³ ³ Mediante esta función es posible leer caracteres del teclado,³ ³ usando el handle 0. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 40H ³ ³ Ã----------´ ³ ³ Escritura en Fichero o dispositivo. ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 40H ³ ³ BX = Handle. ³ ³ CX = Número de bytes a escribir. ³ ³ DS:DX = Segmento:Desplazamiento del buffer desde donde se ³ ³ van a tomar los caracteres a escribir. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ AX = Bytes transferidos. ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Dado un handle v lido, se realiza una transferencia desde el ³ ³ buffer de memoria indicado mediante DS:DX hacia el fichero o ³ ³ dispositivo referenciado por el Handle. ³ ³ Se transferir n tantos caracteres como se especifique en CX. ³ ³ Acto seguido, se actualiza el puntero de fichero una posición por ³ ³ delante del bloque escrito, para que futuras escrituras no ³ ³ 'machaquen' los datos que ya hubiera. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Si se devuelve el flag Cf con valor (0), pero AX=0, esto ³ ³ quiere decir que el dispositivo en el que se encuentra ³ ³ el fichero ya estaba lleno antes de la llamada a esta ³ ³ función. ³ ³ Si se devuelve el flag Cf con valor (0), pero el contenido ³ ³ del registro AX es menor que la cantidad de bytes a escribir,³ ³ (indicado mediante CX antes de llamar a la función), esto ³ ³ significa que se produjo algún error, o que no se pudo ³ ³ escribir todos los caracteres solicitados, porque se ha dado ³ ³ una condición de disco lleno. ³ ³ ³ ³ Mediante esta función es posible escribir caracteres en la ³ ³ pantalla, usando el handle 1. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 41H ³ ³ Ã----------´ ³ ³ Borrar Fichero. ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 41H ³ ³ DS:DX = Segmento:Desplazamiento de la cadena ASCIIZ ³ ³ con el nombre del fichero a borrar. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Se borra el fichero indicado mediante la cadena ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ La función se abortar si: ³ ³ - La vía de acceso contenida en la cadena ASCIIZ es errónea. ³ ³ - Si el fichero a borrar es de sólo lectura. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 42H ³ ³ Ã----------´ ³ ³ Establecer puntero de fichero. ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 42H ³ ³ AL = Código de desplazamiento: ³ ³ 00h Desplazamiento desde el inicio del fichero. ³ ³ 01h Desplazamiento desde la posición actual del ³ ³ puntero. ³ ³ 02h Desplazamiento desde el final del fichero. ³ ³ ³ ³ BX = Handle del fichero. ³ ³ CX = Mitad mas significativa del desplazamiento. ³ ³ DX = Mitad menos significativa del desplazamiento. ³ ³ ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ DX = Mitad mas significativa del puntero actualizado. ³ ³ AX = Mitad menos significativa del puntero actualizado.³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Cambia el valor del puntero de fichero, permitiendo así un acceso ³ ³ aleatorio al fichero. Podremos, mediante

esta función, escribir y ³ ³ leer caracteres en cualquier posición del fichero, sin pasar por ³ ³ las anteriores. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Independientemente del tipo de llamada, el valor del puntero ³ ³ de fichero devuelto por la función en DX,AX se corresponde ³ ³ con un desplazamiento desde el inicio del fichero. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 43H Subfunción 00h ³ ³ Ã----------´ ³ ³ Obtener atributos de fichero ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 43H ³ ³ AL = 00h ³ ³ DS:DX = Segmento:Desplazamiento de la cadena ASCIIZ con el ³ ³ nombre del fichero. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ CX = Atributos del fichero. ³ ³ Si Bit 0 = 1, Fichero de sólo lectura. ³ ³ Si Bit 1 = 1, Fichero oculto. ³ ³ Si Bit 2 = 1, Fichero de sistema. ³ ³ Si Bit 5 = 1, El archivo ha sido modificado desde ³ ³ el último bakcup. ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función se obtiene información acerca de los ³ ³ atributos de un determinado fichero, indicado mediante la cadena ³ ³ ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Se producir un error si la cadena ASCIIZ indicada es errónea.³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 43H Subfunción 01h ³ ³ Ã----------´ ³ ³ Establecer atributos de fichero ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 43H ³ ³ AL = 01h ³ ³ CX = Nuevos atributos para fichero. ³ ³ Si ponemos Bit 0 = 1, atributo de sólo lectura. ³ ³ Si ponemos Bit 1 = 1, atributo de oculto. ³ ³ Si ponemos Bit 2 = 1, atributo de sistmea. ³ ³ Debemos establecer Bit 3 = 0. ³ ³ Debemos establecer Bit 4 = 0. ³ ³ Si ponemos Bit 5 = 1, indicar que el fichero ha ³ ³ sido modificado desde el último backup. ³ ³ ³ ³ DS:DX = Segmento:Desplazamiento de la cadena ASCIIZ con el ³ ³ nombre del fichero. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función se establecen nuevos atributos a un fichero ³ ³ dado. Este fichero se indica mediante la cadena ASCIIZ. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Se producir un error si la cadena ASCIIZ indicada es errónea.³ ³ ³ ³ No puede usarse esta función para establecer atributo de ³ ³ etiqueta de volumen (bit 3), ni atributo de directorio(bit 4).³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 56H ³ ³ Ã----------´ ³ ³ Renombrar Fichero ó Mover Fichero ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 56H ³ ³ DS:DX = Segmento:Desplazamiento de la cadena ASCIIZ con el ³ ³ nombre actual del fichero. ³ ³ ES:DI = Segmento:Desplazamiento de la cadena ASCIIZ con el ³ ³ nuevo nombre para el fichero. ³ Ã DEVUELVE: ³ ³ Si se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 0 ³ ³ ³ ³ Si NO se ejecutó correctamente: ³ ³ Flag de acarreo (Cf) = 1 ³ ³ AX = Código de error. ³ ³ ³ Ã EFECTO: ³ ³ Esta función permite cambiar el nombre a un fichero dado. ³ ³ También permite mover el fichero a otro directorio dentro del mismo ³ ³ dispositivo. ³
³ En el segundo caso, lo que debemos hacer es cambiar el camino que ³ ³ nos lleva al fichero, en vez del nombre de fichero en sí. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Se producir un error si: ³ ³ - la cadena ASCIIZ indicada es errónea. ³ ³ - Ya existe un fichero con el mismo nombre que el indicado. ³ ³ - Se intenta mover el fichero a otro dispositivo o unidad. ³ ³ - El fichero se mueve al directorio raíz, y ,ste est lleno. ³ ³ ³ À----------Ù

* Funciones de terminación de procesos *

Ú----------Â----------¿ ³ INT 21H Función 00H ³ ³ Ã----------´ ³ ³ Terminar ejecución del Programa ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 00H ³ ³ CS = Dirección del segmento del PSP. ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ Se finaliza la ejecución del programa en curso. Esto conlleva: ³ ³ - Liberación de toda la memoria asignada al programa. ³ ³ - Todos los buffers de fichero son vaciados. ³ ³ - Se cierra cualquier fichero abierto por el programa. ³ ³ - Se restauran los tres vectores de interrupción (INT 22H, INT 23H, ³ ³ INT 24H) cuyo contenido original fue almacenado en la pila. ³ ³ ³ Ã VERSION DE MS-DOS: 1.0 ó superior. ³ ³ ³ Ã NOTAS: ³ ³ Esta función est pensada para programas COM, en los cuales, ³ ³ el PSP est en el mismo segmento que el código del programa. ³ ³ En los programas EXE esto no sucede así, estando el PSP en ³ ³ diferente segmento al código del programa, por tanto, no se ³ ³ debe llamar a esta función desde un programa .EXE. ³ ³ ³ ³ En ambos casos (COM y EXE), es preferible utilizar la función ³ ³ 4CH, ya que devuelve un código de retorno la programa padre. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 4CH ³ ³ Ã----------´ ³ ³ Terminación de Programa con Código de Retorno ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 4CH ³ ³ AL = Código de retorno para el programa padre. ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ Se finaliza la ejecución del programa en curso, y se devuelve un ³ ³ código de retorno al programa padre. Mediante este código de ³ ³ retorno, se puede ofrecer información al programa padre acerca de ³ ³ la ejecución del programa (si se ha producido error, etc...) ³ ³ ³ ³ La terminación del programa conlleva: ³ ³ - Liberación de toda la memoria asignada al programa. ³ ³ - Todos los buffers de fichero son vaciados. ³ ³ - Se cierra cualquier fichero abierto por el programa. ³ ³ - Se restauran los tres vectores de interrupción (INT 22H, INT 23H, ³ ³ INT 24H) cuyo contenido original fue almacenado en la pila. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: Este es el m,todo idóneo de terminación de programas, ya que ³ ³ no necesita que el registro CS tenga ningún contenido ³ ³ especial. Y aparte, devuelve información al programa padre. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 21H Función 31H ³ ³ Ã----------´ ³ ³ Finalizar el Programa y Dejar Residente ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 31H ³ ³ AL = Código de retorno para el programa padre. ³ ³ DX = Cantidad de memoria a dejar residente (en p rrafos). ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ Se finaliza la ejecución del programa en curso, y se devuelve un ³ ³ código de retorno al programa padre. Mediante este código de ³ ³ retorno, se puede ofrecer información al programa padre acerca de ³ ³ la ejecución del programa (si se ha producido error, etc...) ³ ³ Ademas de esto, y lo mas importante: Se deja residente el programa ³ ³ o parte del mismo, de forma que las siguientes ejecuciones de ³ ³ programas no lo 'machaquen'. ³ ³ ³ ³ La terminación del programa conlleva: ³ ³ - Liberación de toda la memoria asignada al programa. ³ ³ - Todos los buffers de fichero son vaciados. ³ ³ - Se cierra cualquier fichero abierto por el programa. ³ ³ - Se restauran los tres vectores de interrupción (INT 22H, INT 23H, ³ ³ INT 24H) cuyo contenido original fue almacenado en la pila. ³ ³ ³ Ã VERSION DE MS-DOS: 2.0 ó superior. ³ ³ ³ Ã NOTAS: Un p rrafo equivale a 16 bytes. Por tanto, en DX habr que ³ ³ indicar el número_total_de_bytes_a_dejar_residentes dividido ³ ³ por 16. ³ ³ ³ ³ Esta función se utiliza en programación de utilidades ³ ³ residentes (como SideKick o SanBit). ³ ³ Una vez que el programa queda residente en memoria, la ³ ³ activación del mismo se realiza de acuerdo a algún criterio ³ ³ del propio programa (como la pulsación de una combinación de ³ ³ teclas). En este caso, el programa residente toma el control,³ ³ y al acabar su tarea le

devuelve el control al programa ³ ³ que se estuviera ejecutando antes de la activación del ³ ³ residente. ³ ³ ³ À----------Ù

- funciones BIOS ---------- Antes de meternos de lleno con las funciones BIOS, y mas concretamente con los 2 Servicios que mas nos interesan en un principio: INT 10H (Driver ó controlador de vídeo) e INT 16H (Driver ó controlador de teclado)... Vamos a ver que se entiende por BIOS.
En primer lugar, decir que BIOS son las siglas de Basic Input/Output System. O lo que es lo mismo: Sistema b sico de Entrada/Salida.
Es decir, permite una comunicación (Entrada/Salida de información) entre el ordenador en sí (circuitos, dispositivos, componentes) y los programas que lo utilizan.
La BIOS es un conjunto de funciones contenidas en memoria ROM, mediante el cual, los programas se comunican con el hardware y los dispositivos conectados.
Las funciones de la BIOS permiten al programador acceder a los diferentes subsistemas del ordenador sin necesidad de saber el modelo ni marca de tal subsistema o perif,rico. Es decir, estas funciones facilitan la compatibilidad entre los PC's.
Si queremos acceder al disco duro, simplemente usamos la función de la BIOS que nos permite acceder al disco duro. No necesitamos conocer los cientos de modelos de discos duros que existen, y atendiendo a que se trate de un modelo o de otro distinto, actuar de diferente manera.
Esto nos permite la portabilidad de nuestros programas a cualquier PC.
Aparte de los dos servicios principales que vamos a tratar en esta lección (INT 10H e INT 16H) existen varios mas que permiten controlar el resto de subsistemas del PC:
- INT 11H (Buscar la configuración del equipo). - INT 12H (Determinar el tamaño de memoria RAM). - INT 13H (Driver o Controlador de disco). La INT 13H la estudiaremos en profundidad al tratar el tema de los discos en el PC. - INT 14H (Driver del Puerto Serie ó Puerto de Comunicaciones). - INT 15H (Driver del Puerto Paralelo [Impresora]). - INT 17H (Escritura de sectores a disco). - INT 18H (ROM del BASIC). - INT 19H (Reinicialización del Sistema ó Reset ó Reboot). - INT 1AH (Driver del reloj de Tiempo Real).
Todos estos servicios los estudiaremos a su debido tiempo. En esta lección vamos a tratar los mas usuales para empezar a programar, como son los encargados del teclado y la pantalla, que en definitiva es con lo que primero se empieza a probar.
- INT 10H (Driver de Vídeo) ---------- Dedicaremos varias lecciones a estudiar la programación gr fica, y por consiguiente el Driver de Vídeo. Si os encontrais con algo nuevo en las siguientes funciones (p gina de vídeo, tarjetas gr ficas, etc..) no os preocupeis que lo veremos mas adelante. Vamos por partes, que en programación en ensamblador todo est relacionado de alguna manera, y no podemos verlo todo el mismo día.
Se utiliza el mismo formato de llamada que para la INT 21h.
ÍÍÍÍÍÍÍÍÍÍÍ Funciones Fundamentales de la INT 10H ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
Ú----------Â----------¿ ³ INT 10H Función 00H ³ ³ Ã----------´ ³ ³ Establecer modo de Vídeo ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 00H ³ ³ AL = Modo de vídeo (Ver notas). ³ ³ ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Selecciona y activa el modo de vídeo especificado. ³ ³ A no ser que se utilice el truco que se indica a continuación, ³ ³ al llamar a esta función, se borra la pantalla. ³ ³ Por una parte, esto nos quita el trabajo de borrar la pantalla por ³ ³ nosotros mismos. Pero por otra parte, cuando deseamos que el ³ ³ contenido de la pantalla en el modo seleccionado no se pierda, ³ ³ como puede ser el caso de programación de utilidades residentes, ³ ³ esa función de borrado autom tico nos puede acarrear demasiadas ³ ³ molestias. ³ ³ Por suerte hay una especie de 'truco' para evitar este borrado ³ ³ autom tico de la pantalla. Consiste en poner con valor 1 el bit 7 ³ ³ del

registro AL (que contiene el modo de vídeo) en la llamada a la ³ ³ función. ³ ³ Así por ejemplo, si queremos cambiar a modo 13h, y queremos que no ³ ³ se pierda el contenido que hubiera en la pantalla en este modo, ³ ³ en vez de introducir en AL el número 13h (00010011b), ³ ³ introduciríamos el número 93h (10010011b). ³ ³ ³ Ã TARJETA GRAFICA: TODAS (MDA, CGA, EGA, MCGA, VGA...) ³ ³ ³ Ã NOTAS: ³ ³ Modos de vídeo y características principales: ³ ³ ³ ³ ÉÍÍÍÍÍÑÍÍÍÍÍÍÍÍÍÍÍÍÍÑÍÍÍÍÍÍÍÍÍÑÍÍÍÍÍÍÍÍÍÍÍÍÍÍÑÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ» ³ ³ º ³ ³ ³ Modo Texto ³ Tarjetas que º ³ ³ º Modo ³ Resolución ³ Colores ³ ó ³ Soportan este º ³ ³ º ³ ³ ³ Modo Gr fico ³ Modo º ³ ³ ÌÍÍÍÍÍØÍÍÍÍÍÍÍÍÍÍÍÍÍØÍÍÍÍÍÍÍÍÍØÍÍÍÍÍÍÍÍÍÍÍÍÍÍØÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¹ ³ ³ º 00H ³ 40 por 25 ³ 16 ³ Texto ³ (2) (3) (4) (5)º ³ ³ º 01h ³ 40 por 25 ³ 16 ³ Texto ³ (2) (3) (4) (5)º ³ ³ º 02h ³ 80 por 25 ³ 16 ³ Texto ³ (2) (3) (4) (5)º ³ ³ º 03h ³ 80 por 25 ³ 16 ³ Texto ³ (2) (3) (4) (5)º ³ ³ º 04h ³ 320 por 200 ³ 4 ³ Gr fico ³ (2) (3) (4) (5)º ³ ³ º 05h ³ 320 por 200 ³ 4 ³ Gr fico ³ (2) (3) (4) (5)º ³ ³ º 06h ³ 640 por 200 ³ 2 ³ Gr fico ³ (2) (3) (4) (5)º ³ ³ º 07h ³ 80 por 25 ³ 2 ³ Texto ³(1) (3) (5)º ³ ³ º 0Dh ³ 320 por 200 ³ 16 ³ Gr fico ³ (3) (5)º ³ ³ º 0Eh ³ 640 por 200 ³ 16 ³ Gr fico ³ (3) (5)º ³ ³ º 0Fh ³ 640 por 350 ³ 2 ³ Gr fico ³ (3) (5)º ³ ³ º 10h ³ 640 por 350 ³ 4 ³ Gr fico ³EGA de 64 Ks de RAMº ³ ³ º 10h ³ 640 por 350 ³ 16 ³ Gr fico ³EGA>=64 Ks y VGA º ³ ³ º 11h ³ 640 por 480 ³ 2 ³ Gr fico ³ (4) (5)º ³ ³ º 12h ³ 640 por 480 ³ 16 ³ Gr fico ³ (5)º ³ ³ º 13h ³ 320 por 200 ³ 256 ³ Gr fico ³ (4) (5)º ³ ³ ÈÍÍÍÍÍÏÍÍÍÍÍÍÍÍÍÍÍÍÍÏÍÍÍÍÍÍÍÍÍÏÍÍÍÍÍÍÍÍÍÍÍÍÍÍÏÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼ ³ ³ ³ ³ Modos superiores al 13h pertenecen a tarjetas Super-Vga ó superior. ³ ³ ³ ³ Código de las tarjetas gr ficas: MDA = (1) ³ ³ CGA = (2) ³ ³ EGA = (3) ³ ³ MCGA = (4) ³ ³ VGA = (5) ³ ³ ³ À---Ù Ú--Â--------------------------¿ ³ INT 10H Función 01H ³ ³ Ã--´ ³ ³ Establecer tamaño del Cursor ³ ³ Ã--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 01H ³ ³ Bits 0-4 de CH = Línea inicial del Cursor. ³ ³ Bits 0-4 de CL = Línea final del Cursor. ³ ³ ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Se selecciona un nuevo tamaño de Cursor en modo texto. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Ver función 02h en caso de querer hacer desaparecer el ³ ³ cursor de la pantalla. ³ ³ Dependiendo del tipo de tarjeta y modo de vídeo de que se ³ ³ trate, el cursor dispondr de mas o menos líneas de pixels ³ ³ para dibujarlo. Así por ejemplo, en varios modos de vídeo ³ ³ de la VGA, se disponen de 14 líneas para crear el cursor, ³ ³ mientras que en la CGA se disponen sólo de 8 líneas. ³ À---Ù Ú--Â--------------------------¿ ³ INT 10H Función 02H ³ ³ Ã--´ ³ ³ Posicionar el Cursor ³ ³ Ã--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 02H ³ ³ BH = P gina de vídeo. ³ ³ DH = Línea donde situar el cursor. ³ ³ DL = Columna donde situar el cursor. ³ ³ ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Posiciona el cursor en pantalla, de acuerdo a las coordenadas ³ ³ indicadas en los registros DH y DL. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Sólo se desplazar el cursor si la p gina de vídeo indicada ³ ³ mediante BH es la p gina de vídeo activa. Esto es así ya ³ ³ que existe un cursor independiente para cada una de las ³ ³ p ginas de vídeo con las que contamos en el modo actual. ³ ³ ³ ³ Las coordenadas para la columna empiezan a partir de 0. ³ ³ Las coordenadas para la fila empiezan a partir de 0. ³ ³ Esto quiere decir que la esquina superior izquierda de la ³ ³ pantalla tendr las coordenadas (línea=0,columna=0). ³ ³ ³ ³ La columna m xima es la 39 (si estamos en un modo de vídeo ³ ³ de 40 columnas) ó la 79 (si estamos en un modo de vídeo de ³ ³ 80 columnas). ³ ³ La línea m xima es la 24 (si estamos en un modo de vídeo de ³ ³ 25 líneas) ó la 49 (si estamos en un modo de vídeo de 50 ³ ³ líneas). ³ ³ ³ ³ Un 'truco' para hacer desaparecer el cursor de la pantalla ³ ³ consiste en dar valores no v lidos para la columna o la ³ ³ fila. Por ejemplo, si damos a la columna el valor 100, el ³ ³ cursor desaparecer de la pantalla. ³ ³ ³ À---Ù

Ú--Â--------------------------¿ ³ INT 10H Función 03H ³ ³ Ã---------------------------------------´ ³ ³ Buscar posición del Cursor y Tamaño ³ ³ Ã--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 03H ³ ³ BH = P gina de vídeo. ³ ³ ³ Ã DEVUELVE: ³ ³ CH = Línea inicial del cursor ³ ³ CL = Línea final del cursor ³ ³ DH = Fila donde se encuentra el cursor ³ ³ CH = Columna donde se encuentra el cursor ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Obtiene las coordenadas del cursor en una p gina de vídeo dada. ³ ³ También se obtiene el tamaño del cursor. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Cada p gina de vídeo tiene su propio cursor. ³ ³ Se puede acceder al cursor de cada p gina independientemente³ ³ de cu l sea la p gina activa. ³ ³ ³ À---Ù

Ú--Â--------------------------¿ ³ INT 10H Función 05H ³ ³ Ã---------------------------------------´ ³ ³ Establecer nueva p gina de vídeo ³ ³ Ã--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 05H ³ ³ AL = P gina de vídeo (Ver Notas). ³ ³ ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ ³ ³ Establece como activa la nueva p gina de vídeo indicada mediante ³ ³ el registro AL, mostr ndose por pantalla el contenido de esa ³ ³ p gina de vídeo, y situando el cursor en la posición que le ³ ³ corresponda para esa p gina de vídeo. ³ ³ ³ Ã TARJETA GRAFICA: CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Las p ginas posibles para cada modo de vídeo son: ³ ³ ³ ³ ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÑÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÑÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ» ³ ³ º Número de P ginas ³ Modos de Vídeo ³ Tarjetas que lo soportan º ³ ³ º ³ ³ º ³ ³ º 8 (0 - 7) ³ 00h y 01h ³ CGA, EGA, MCGA, VGA. º ³ ³ º 4 (0 - 3) ³ 02h y 03h ³ CGA. º ³ ³ º 8 (0 - 7) ³ 02h y 03h ³ ECGA, MCGA, VGA. º ³ ³ º 8 (0 - 7) ³ 07h ³ EGA, VGA. º ³ ³ º 8 (0 - 7) ³ 0Dh ³ EGA, VGA. º ³ ³ º 4 (0 - 3) ³ 0Eh ³ EGA, VGA. º ³ ³ º 2 (0 - 1) ³ 0Fh ³ EGA, VGA. º ³ ³ º 2 (0 - 1) ³ 10h ³ EGA, VGA. º ³ ³ º ³ ³ º ³ ³ ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÏÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÏÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼ ³ ³ Los modos de vídeo 11h en adelante cuentan sólo con una p gina ³ ³ de vídeo. ³ ³ ³ ³ El cambio de una p gina a otra dentro del mismo modo de vídeo, no ³ ³ afecta a su contenido, es decir, no se borra la pantalla, como sí ³ ³ ocurre al cambiar de modo de vídeo. Siempre que no utilicemos el ³ ³ 'truco', claro. ³ ³ ³ À---Ù

Ú--Â--------------------------¿ ³ INT 10H Función 06H ³ ³ Ã---------------------------------------´ ³ ³ Desplazar líneas de texto hacia arriba ³ ³ Ã--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 06H ³ ³ AL = Número de líneas a desplazar. Si AL=0, se borra ³ ³ toda la ventana seleccionada mediante los registros³ ³ CX y DX. ³ ³ BH = Atributo a usar en las líneas borradas. ³ ³ CH = Línea donde comienza la ventana de texto. ³ ³ CL = Columna donde comienza la ventana de texto. ³ ³ DH = Línea donde acaba la ventana de texto. ³ ³ DL = Columna donde acaba la ventana de texto. ³ ³ ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ ³ ³ Desplaza hacia arriba un número determinado de líneas en la ventana ³ ³ especificada mediante los registros CX y DX. ³ ³ Las líneas desplazadas, quedan vacías, rellen ndose con blancos. ³ ³ El color utilizado en estas líneas vacías se indica mediante el ³ ³ registro BH. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Mediante la llamada a esta función obtenemos un m,todo ³ ³ cómodo, aunque lento de borrar la pantalla. ³ ³ ³ ³ El Atributo indicado mediante BH es el color que se va a ³ ³ utilizar en el rea borrada ó desplazada. ³ ³ ³ À---Ù

Ú--Â--------------------------¿ ³ INT 10H Función 07H ³ ³ Ã---------------------------------------´ ³ ³ Desplazar líneas de texto hacia abajo ³ ³ Ã--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 07H ³ ³ AL = Número de líneas a desplazar. Si AL=0, se borra ³ ³ toda la ventana seleccionada mediante los registros³ ³ CX y DX. ³ ³ BH = Atributo a usar en las líneas borradas. ³ ³ CH = Línea donde comienza la ventana de texto. ³ ³ CL = Columna donde comienza la ventana de texto. ³ ³ DH = Línea donde acaba la ventana de texto. ³ ³ DL = Columna donde acaba la ventana de texto. ³ ³ ³ Ã

DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ ³ ³ Desplaza hacia abajo un número determinado de líneas en la ventana ³ ³ especificada mediante los registros CX y DX. ³ ³ Las líneas desplazadas, quedan vacías, rellen ndose con blancos. ³ ³ El color utilizado en estas líneas vacías se indica mediante el ³ ³ registro BH. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Mediante la llamada a esta función obtenemos un m,todo ³ ³ cómodo, aunque lento de borrar la pantalla. ³ ³ ³ ³ El Atributo indicado mediante BH es el color que se va a ³ ³ utilizar en el rea borrada ó desplazada. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 10H Función 08H ³ ³ Ã----------´ ³ ³ Leer car cter y atributo ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 08H ³ ³ BH = P gina de vídeo. ³ Ã DEVUELVE: ³ ³ AH = Atributo (color del car cter). ³ ³ AL = Código ASCII del car cter leido. ³ ³ ³ Ã EFECTO: ³ ³ ³ ³ Mediante la llamada a esta función, se devuelve en AL el código ³ ³ del car cter situado en la posición del cursor. ³ ³ Asímismo, obtenemos en AH el color de este car cter. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Esta función es usada por la utilidad SB-ASCII del programa ³ ³ SanBit para obtener el código de los caracteres que hay en ³ ³ pantalla. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 10H Función 09H ³ ³ Ã----------´ ³ ³ Escribir car cter y atributo ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 09H ³ ³ AL = Código del car cter a escribir. ³ ³ BH = P gina de vídeo donde escribir el car cter. ³ ³ BL = Atributo ó color que va a tener el car cter. ³ ³ CX = Cantidad de veces que se debe escribir el car cter,³ ³ uno a continuación de otro. ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ ³ ³ Se escribe un car cter en la posición actual del cursor, en la ³ ³ p gina de vídeo deseada. ³ ³ El car cter tendr el color indicado mediante BL. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: En caso de querer escribir un car cter sin modificar el ³ ³ color que afectara a esa posición, debe usarse la función ³ ³ 0AH. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 10H Función 0AH ³ ³ Ã----------´ ³ ³ Escribir car cter ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 0AH ³ ³ AL = Código del car cter a escribir. ³ ³ BH = P gina de vídeo donde escribir el car cter. ³ ³ CX = Cantidad de veces que se debe escribir el car cter,³ ³ uno a continuación de otro. ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ ³ ³ Se escribe un car cter en la posición actual del cursor, en la ³ ³ p gina de vídeo deseada. ³ ³ El car cter tendr el mismo color que tuviera el car cter antiguo ³ ³ en esa posición. Es decir, se modifica el car cter, pero no el ³ ³ color. ³ ³ ³ Ã TARJETA GRAFICA: MDA, CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: En caso de querer modificar el color del car cter, usar ³ ³ la función 09h. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 10H Función 0CH ³ ³ Ã----------´ ³ ³ Escribir un punto ó pixel gr fico ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 0CH ³ ³ AL = Valor del color a usar. ³ ³ BH = P gina de vídeo donde escribir el car cter. ³ ³ CX = Columna donde escribir el pixel (coordenada ³ ³ gr fica x). ³ ³ CX = Fila donde escribir el pixel (coordenada ³ ³ gr fica y). ³ Ã DEVUELVE: ³ ³ NADA. ³ Ã EFECTO: ³ ³ ³ ³ Da un nuevo color a un pixel gr fico. ³ ³ ³ Ã TARJETA GRAFICA: CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Función v lida sólo para modos gr ficos. ³ ³ ³ ³ Ver función 00h para obtener información acerca del valor ³ ³ m ximo de fila,

columna y color en cada modo de vídeo. ³ ³ ³ À----------Ù
Ú----------Â----------¿ ³ INT 10H Función 0DH ³ ³ Ã----------´ ³ ³ Obtener el color de un pixel gr fico ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 0DH ³ ³ BH = P gina de vídeo. ³ ³ CX = Columna del pixel que nos interesa (coordenada ³ ³ gr fica x). ³ ³ DX = Fila del pixel que nos interesa (coordenada ³ ³ gr fica y). ³ Ã DEVUELVE: ³ ³ AL = Valor del color del pixel. ³ Ã EFECTO: ³ ³ ³ ³ Obtiene el color de un punto gr fico de la pantalla. ³ ³ Este punto se referenciar mediante las coordenadas gr ficas (x,y): ³ ³ (CX,DX). ³ ³ ³ Ã TARJETA GRAFICA: CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Función v lida sólo para modos gr ficos. ³ ³ ³ ³ Ver función 00h para obtener información acerca del valor ³ ³ m ximo de fila, columna y color en cada modo de vídeo. ³ ³ ³ À----------Ù
Ú----------Â----------¿ ³ INT 10H Función 0FH ³ ³ Ã----------´ ³ ³ Obtener el Modo de vídeo actual ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 0FH ³ Ã DEVUELVE: ³ ³ AL = Modo de vídeo actual. ³ ³ AH = Cantidad de caracteres que caben en una línea en ³ ³ el modo de vídeo actual. ³ ³ BH = Número de la p gina activa. ³ Ã EFECTO: ³ ³ ³ ³ Mediante esta función podemos saber en todo momento, en que modo ³ ³ de vídeo estamos trabajando. ³ ³ En un principio esta función puede no tener mucho sentido, ya que ³ ³ en nuestro programa, en todo momento sabemos con que modo de vídeo ³ ³ estamos trabajando, ya que lo establecemos nosotros. ³ ³ ³ ³ La utilidad de esta función reside sobre todo en la construcción de ³ ³ programas residentes. El programa residente, antes de activarse ³ ³ definitivamente y mostrar sus datos y rótulos por pantalla, debe ³ ³ saber el modo de vídeo anterior, en el que estaba trabajando el ³ ³ usuario, para que al salir del programa residente, se pueda ³ ³ restituir, y así no se pierda la información que había anteriormente³ ³ en pantalla. ³ ³ ³ Ã TARJETA GRAFICA: CGA, EGA, MCGA, VGA. ³ ³ ³ Ã NOTAS: Esta función se puede emplear también para conocer el ³ ³ número m ximo de caracteres que caben en una línea del ³ ³ modo actual de vídeo. ³ ³ ³ À----------Ù

- INT 16H (Driver de Teclado) ---------- Los servicios o funciones de la INT 16H o driver de teclado, nos permiten de una manera cómoda, gestionar todo el trabajo relativo a la entrada de caracteres por medio del teclado.
Con la ayuda de estas funciones, no necesitamos trabajar directamente sobre los registros y estructuras de datos necesarios para el buen funcionamiento de la entrada de teclado.
De cualquier modo, en una próxima lección estudiaremos a fondo el teclado y todo lo relacionado con el, como buffer de teclado (que es donde se almacenan las pulsaciones de teclas hasta que se procesan por el programa) y demas...

Se utiliza la misma convención de llamada que para la INT 21H y la INT 10H.
ÍÍÍÍÍÍÍÍÍÍÍ Funciones Fundamentales de la INT 16H ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
Ú----------Â----------¿ ³ INT 16H Función 00H ³ ³ Ã----------´ ³ ³ Leer car cter del teclado ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 00H ³ ³ ³ Ã DEVUELVE: ³ ³ AH = Código extendido de la tecla, ó Scan-Code. ³ ³ AL = Código ASCII de la tecla. ³ ³ ³ Ã EFECTO: ³ ³ Acepta un car cter del teclado y devuelve el código ASCII de dicho ³ ³ car cter, así como el código de rastreo ó el código extendido de ³ ³ la tecla. ³ ³ Si no había ningún car cter disponible en el buffer de teclado, la ³ ³ función esperar a que se introduzca algún car cter desde el ³ ³ teclado. ³ ³ Una vez se ha leido el car cter mediante esta función, se elimina ³ ³ del buffer, para que siguientes llamadas a esta función no lo ³ ³ vuelvan a coger. ³ ³ ³ Ã MODELO DE

ORDENADOR: Todos (PC, XT, AT...) ³ ³ ³ Ã NOTAS: ³ ³ El código de rastreo (Scan Code) es el número que identifica ³ ³ a una tecla en particular. ³ ³ Los códigos extendidos se asignan a teclas o combinaciones ³ ³ de teclas que no disponen de un símbolo ASCII para ³ ³ representarlas. Como ejemplo tendríamos las teclas de ³ ³ función, las teclas Home, End, etc... ³ ³ ³ ³ Cuando a la salida de la función AL = 0, esto quiere decir ³ ³ que nos encontramos ante un código de tecla extendido, como ³ ³ Home, Insert, F1, (Alt + F2), etc... En este caso, lo que ³ ³ tenemos en AH es el código extendido de esa tecla o ³ ³ combinación de teclas. ³ ³ ³ ³ Cuando nos encontramos ante la pulsación de una tecla ³ ³ 'normal' como 'a', '1', etc, en este caso, el registro AL ³ ³ contiene el código ASCII de la tecla, y el registro AH ³ ³ contiene el código de rastreo de la tecla. ³ ³ ³ À----------
----------Ù
Ú----------Â----------¿ ³ INT 16H Función 01H ³ ³ Ã----------
----------´ ³ ³ Obtener estado del buffer de teclado ³ ³ Ã----------
----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 01H ³ ³ ³ Ã DEVUELVE: ³ ³ Si había alguna tecla disponible en el buffer: ³ ³ Flag de cero (Zf) = 0 ³ ³ AH = Código de rastreo ó Código extendido ³ ³ AL = Código ASCII de la tecla. ³ ³ ³ ³ Si no había ninguna tecla disponible en el buffer: ³ ³ Flag de cero (Zf) = 1 ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función se puede saber cuando hay algún car cter ³ ³ esperando en el buffer de teclado. En caso de haber algún car cter ³ ³ en el buffer, se muestra su código ASCII y su (código de rastreo ³ ³ ó código extendido, según proceda). Esta función no elimina ningún ³ ³ car cter del buffer de teclado. Por tanto, sucesivas llamadas a esta³ ³ función, mostrar n siempre el mismo resultado. ³ ³ ³ Ã MODELO DE ORDENADOR: Todos (PC, XT, AT...) ³ ³ ³ Ã NOTAS: ³ ³ Para borrar el car cter del buffer, utilizar la función 00H. ³ ³ ³ À----------
----------Ù
Ú----------Â----------¿ ³ INT 16H Función 02H ³ ³ Ã----------
----------´ ³ ³ Obtener estado del teclado ³ ³ Ã----------
--Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 02H ³ ³ ³ Ã DEVUELVE: ³ ³ AL = Valor del Byte de estado del teclado. ³ ³ ³ ³ ³ ³ Byte de Estado de teclado: ³ ³ ³ ³ bit bit bit bit bit bit bit bit ³ ³ 7 6 5 4 3 2 1 0 ³ ³ Ú---Â---Â---Â---Â---Â---Â---Â---¿ Estado de teclas cuando el bit ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ correspondiente tiene valor (1) ³ ³ À---Á---Á---Á---Á---Á---Á---Á---Ù ----------
³ À----> Tecla Mays. derecha pulsada. ³ ³ ³ ³ ³ ³ ³ ³ À--------> Tecla Mays. izquierda pulsada.³ ³ ³ ³ ³ ³ ³ À------------> Tecla Control pulsada. ³ ³ ³ ³ ³ ³ À----------
--> Tecla Alt pulsada. ³ ³ ³ ³ ³ À--------------------> Scroll Lock activado. ³ ³ ³ ³ À----------
-----> Num Lock activado. ³ ³ ³ À----------> Caps Lock (Bloq Mays) activado³ ³ À----------> Insert activado. ³ ³ ³ ³ ³ ³ ³ Ã EFECTO: ³ ³ Mediante la llamada a esta función, obtenemos información acerca ³ ³ del estado de ciertas teclas especiales (CTRL, ALT, MAYS, etc). ³ ³ Asímismo, nos muestra el estado de Num Lock (Bloque num,rico), etc. ³ ³ ³ Ã MODELO DE ORDENADOR: Todos (PC, XT, AT...) ³ ³ ³ Ã NOTAS: ³ ³ El byte de estado del teclado se encuentra en la dirección ³ ³ 0000:0417h, dentro de la zona de datos de la BIOS. ³ ³ ³ À----------Ù
Ú----------Â----------¿ ³ INT 16H Función 03H ³ ³ Ã----------
----------´ ³ ³ Establecer factor de Repetición ³ ³ Ã----------
--------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 03H ³ ³ AL = 05H ³ ³ BH = Retardo antes de la primera repetición de tecla. ³ ³ BL = Velocidad de Repetición. ³ ³ ³ Ã DEVUELVE: ³ ³ NADA. ³ ³ ³ Ã EFECTO: ³ ³ Esta función nos permite modificar la velocidad de repetición de ³ ³ teclado, así como el tiempo que se necesita tener pulsada una tecla ³ ³ para que se genere la primera repetición. ³ ³ Ambas cosas son totalmente diferentes. ³ ³ ³ Ã MODELO DE ORDENADOR: Todos (PC, XT, AT...) ³ ³ ³ Ã NOTAS: ³ ³ Esta función es la base de la utilidad SB-TECLA del programa ³ ³ SanBit. Para ver los diferentes valores posibles que pueden ³ ³ tomar los registros BH y BL, echar un vistazo a dicha ³ ³ utilidad. ³ ³ ³ À----------
----------Ù

Ú----------Â----------¿ ³ INT 16H Función 05H ³ ³ Ã----------´ ³ ³ Simular pulsación de tecla ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 05H ³ ³ CH = Código extendido ó Código de rastreo. ³ ³ CL = Código ASCII de la tecla en cuestión. ³ ³ ³ Ã DEVUELVE: ³ ³ Si el Buffer de Teclado no est lleno (cabe otro car cter): ³ ³ Bit de acarreo (Cf) = 0 ³ ³ AL = 00h ³ ³ Si el Buffer de Teclado est lleno (no caben mas caracteres): ³ ³ Bit de acarreo (Cf) = 1 ³ ³ AL = 01h ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función podemos simular la pulsación de una tecla. ³ ³ ³ Ã MODELO DE ORDENADOR: AT y superiores. ³ ³ ³ Ã NOTAS: ³ ³ Esta función la utilizan ciertos programas residentes para ³ ³ realizar Macros de Teclado. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 16H Función 10H ³ ³ Ã----------´ ³ ³ Leer car cter del teclado Expandido ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 10H ³ ³ ³ Ã DEVUELVE: ³ ³ AL = Código de Rastreo ó Código Extendido. ³ ³ AL = Código ASCII del car cter introducido. ³ ³ ³ Ã EFECTO: ³ ³ Acepta un car cter del teclado y devuelve el código ASCII de dicho ³ ³ car cter, así como el código de rastreo ó el código extendido de ³ ³ la tecla. ³ ³ Si no había ningún car cter disponible en el buffer de teclado, la ³ ³ función esperar a que se introduzca algún car cter desde el ³ ³ teclado. ³ ³ Una vez se ha leido el car cter mediante esta función, se elimina ³ ³ del buffer, para que siguientes llamadas a esta función no lo ³ ³ vuelvan a coger. ³ ³ ³ Ã MODELO DE ORDENADOR: AT y superiores. ³ ³ ³ Ã NOTAS: ³ ³ Sólo para teclados Expandidos. ³ ³ ³ ³ Esta función es similar a la función 00h. La diferencia ³ ³ reside en que esta función est preparada para trabajar con ³ ³ teclados expandidos (de los de 101 y 102 teclas), mientras ³ ³ que la función 00h (que es la función original de entrada de ³ ³ caracteres desde teclado) trabaja con los códigos originales ³ ³ del teclado de 84 teclas. ³ ³ Es decir, entre otras, no acepta las teclas de función F11 ³ ³ ni F12. ³ ³ ³ ³ Los códigos de rastreo (Scan Codes) son diferentes para cada ³ ³ una de estas funciones, ya que cada uno de los teclados (el ³ ³ original y el expandido) generan un código de rastreo ³ ³ diferente al del otro modelo de teclado para sus teclas. ³ ³ ³ ³ Para mas información, ver función 00h. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 16H Función 11H ³ ³ Ã----------´ ³ ³ Obtener estado del buffer de teclado ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 11H ³ ³ ³ Ã DEVUELVE: ³ ³ Si había alguna tecla disponible en el buffer: ³ ³ Flag de cero (Zf) = 0 ³ ³ AH = Código de rastreo ó Código extendido ³ ³ AL = Código ASCII de la tecla. ³ ³ ³ ³ Si no había ninguna tecla disponible en el buffer: ³ ³ Flag de cero (Zf) = 1 ³ ³ ³ Ã EFECTO: ³ ³ Mediante esta función se puede saber cuando hay algún car cter ³ ³ esperando en el buffer de teclado. En caso de haber algún car cter ³ ³ en el buffer, se muestra su código ASCII y su (código de rastreo ³ ³ ó código extendido, según proceda). Esta función no elimina ningún ³ ³ car cter del buffer de teclado. Por tanto, sucesivas llamadas a esta³ ³ función, mostrar n siempre el mismo resultado. ³ ³ ³ Ã MODELO DE ORDENADOR: AT y superiores. ³ ³ ³ Ã NOTAS: ³ ³ Sólo para teclados Expandidos. ³ ³ ³ ³ Esta función es similar a la función 01h. La diferencia ³ ³ reside en que esta función tiene en cuenta los códigos de ³ ³ tecla especiales de los teclados expandidos, como por ejemplo³ ³ las teclas de función F11 y F12. ³ ³ ³ ³ Para mas información, ver función 01h. ³ ³ ³ À----------Ù

Ú----------Â----------¿ ³ INT 16H Función 12H ³ ³ Ã----------´ ³ ³ Obtener estado del Teclado Expandido ³ ³ Ã----------Ù ³ ³ ³ Ã LLAMADA: ³ ³ AH = 12H ³ ³ ³ Ã DEVUELVE: ³ ³ AX = Valor de la Palabra de estado del teclado expandido. ³ ³ ³ ³ Palabra de Estado del teclado expandido: ³ ³ ³ ³ 15 13 11 9 8 7 6 5 4 3 2 1 0 ³ ³ Ú-Â-Â-Â-Â-Â-Â-Â-Â-Â-Â-Â-Â-Â-Â-Â-¿ Estado de teclas cuando el bit ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ correspondiente tiene valor (1) ³ ³ À-Á-Á-Á-

Á-Á-Á-Á-Á-Á-Á-Á-Á-Á-Á-Á-Á-Ù ------------------------------- ³ ³ 14 12 10 ³ À----> Tecla Mays. derecha pulsada. ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À------> Tecla Mays. izquierda pulsada.³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À--------> Alguna tecla CONTROL pulsada. ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À----------> Alguna tecla ALT pulsada ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À------------> Scroll Lock activado. ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À--------------> Num Lock activado. ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À----------------> Caps Lock (Bloq Mays) activado³ ³ ³ ³ ³ ³ ³ ³ ³ ³ À------------------> Insert activado. ³ ³ ³ ³ ³ ³ ³ ³ ³ À--------------------> CONTROL izquierda pulsada. ³ ³ ³ ³ ³ ³ ³ ³ À----------------------> ALT izquierda pulsada. ³ ³ ³ ³ ³ ³ ³ À------------------------> CONTROL derecha pulsada. ³ ³ ³ ³ ³ ³ À--------------------------> ALT derecha pulsada. ³ ³ ³ ³ ³ À----------------------------> Tecla Scroll Lock pulsada. ³ ³ ³ ³ À------------------------------> Tecla Num Lock pulsada. ³ ³ ³ À--------------------------------> Tecla Caps Lock pulsada. ³ ³ À----------------------------------> Tecla SysRep pulsada. ³ ³ ³ ³ EFECTO: ³ ³ Mediante la llamada a esta función, obtenemos información acerca ³ ³ del estado de ciertas teclas especiales del teclado expandido. ³ ³ ³ Ã MODELO DE ORDENADOR: AT y superiores. ³ ³ ³ Ã NOTAS: ³ ³ Sólo para teclados Expandidos. ³ ³ ³ ³ Esta función es similar a la función 02h. La diferencia ³ ³ reside en que esta función ofrece un byte de información ³ ³ adicional acerca del estado de los conmutadores y teclas ³ ³ especiales. Por ejemplo: Mientras que con la función 02h ³ ³ podíamos saber si se había pulsado CTRL, con la función ³ ³ 12h podemos saber cu l de las teclas de CTRL ha sido pulsada.³ ³ ³ ³ Para mas información, ver función 02h. ³ ³ ³ À--Ù
Las funciones de teclado que acabamos de ver deben utilizarse preferentemente a las suministradas por el DOS, ya que ,stas ofrecen mayor rapidez de ejecución y mas prestaciones que las que facilita el DOS.

Para mas información acerca de los servicios y funciones vistos en esta lección, así como otros servicios no tratados por el momento:
- La ROM BIOS de IBM. (Ray Duncan... Microsoft Press/Anaya Multimedia).
- Funciones del MS-DOS. (Ray Duncan... Microsoft Press/Anaya Multimedia).
- Extensiones del MS-DOS. (Ray Duncan... Microsoft Press/Anaya Multimedia).

Los tres libros de arriba son manuales de bolsillo, pero también hay libros que traen un ap,ndice con los diversos servicios del sistema.
Tenemos por ejemplo el libro PC Interno, y otros muchos.
Espero que a alguien le haya venido bien esta lección.
* Fin de la lección 10 *

Un saludo. Francisco Jesus Riquelme.

ASM por AeSoft. (lección 11). -- - PROGRAMAS EN MS-DOS: - PROGRAMAS (COM) - PROGRAMAS (EXE) - PSP (Prefijo de Segmento de Programa) - BLOQUE DE ENTORNO - COLA DE ORDENES - PSEUDO_INSTRUCCIONES - MODELOS Y EJEMPLOS - CREAR EL PROGRAMA EJECUTABLE (ENSAMBLAR-LINKAR) ---

Hola de nuevo a todos los seguidores del curso de asm (ASM por AEsoft). En las 10 primeras lecciones hemos tratado lo que he considerado imprescindible para poder empezar a programar en ensamblador.
Hemos estudiado el ordenador sobre el que vamos a trabajar, las principales instrucciones ASM para programarlo, las funciones mas importantes (a este nivel del curso) que nos ofrece el DOS, la BIOS y el Driver de Teclado... y algunas cosas mas que he considerado necesarias antes de meternos a programar.
Pues bien, ha llegado el momento. Vamos a empezar a programar. En esta lección estudiaremos los dos tipos de programas con los que contamos en MS-DOS: programas COM y programas EXE.
Ademas, trataremos las estructuras de datos asociadas: Prefijo de Segmento de Programa, Bloque de Entorno, Cola de Ordenes, etc...
... Y muchas mas cosas interesantes.
- PROGRAMAS EN MS-DOS ---------- Bajo el sistema operativo MS-DOS y todos los compatibles como DR-DOS, PC-DOS, etc.. tenemos 2 modelos diferentes de programas ejecutables: El modelo COM y el modelo EXE. En siguientes apartados veremos sus diferencias, ventajas y desventajas. En este apartado vamos a ver lo que tienen en común.
Como hemos dicho, estos dos modelos de programas son los únicos que reconoce el DOS. Dejaremos a un lado los Ficheros De Proceso Por Lotes o ficheros BATCH (Extensión BAT), ya que aunque son ejecutables, no hay código ejecutable directamente por el procesador dentro de ellos, sino llamadas a otros programas y comandos propios pertenecientes a un pseudo-lenguaje de programación BATCH. No les daremos por tanto la condición de Programa, sino de Fichero de Proceso Por Lotes.
Ambos programas (COM y EXE) se cargan y ejecutan en el rea de programas transitorios (TPA) (siempre que haya memoria suficiente para hacerlo), llam ndose por tanto 'Programas transitorios'. Todos los programas se cargan para su ejecución en el TPA, pero hay programas especiales que se quedan residentes antes de terminar su ejecución. Estos programas se llaman 'Programas Residentes', y la zona de memoria donde se encuentran se denomina 'Area de Programas Residentes'.
Como podemos ver, cuando dejamos un programa residente, estamos robando memoria al TPA para agrandar el Area de Programas Residentes. De forma similar, cuando desinstalamos un Programa Residente de la memoria, el TPA crece de acuerdo al tamaño de dicho Programa. (Este tema lo veremos en profundidad al tratar los Programas Residentes).
Sigamos con las generalidades:
Al programa que se va a ejecutar se le puede indicar que ejecute una determinada tarea de las que ofrece al usuario mediante lo que se llama Cola De Ordenes, que no es ni mas ni menos que los par metros que se le pasan a un programa a continuación del nombre de programa.
Ejemplo: ARJ A FILE *.* En este ejemplo, la Cola De Ordenes est formada por todo lo que hay a la derecha de 'ARJ' (nombre del programa), esto es, 'A FILE *.*'. Atendiendo a la Cola De Ordenes pasada al programa, ,ste realizar una de las tareas posibles u otra. En este caso, la Cola De Ordenes le indicaría al programa que comprima (A) en el fichero FILE todos los archivos del directorio actual (*.*).
La Cola De Ordenes es el conjunto de par metros que se le pasan al programa para que realice una determinada tarea. Mediante estos par metros, un programa puede realizar la tarea solicitada por el usuario de entre toda una serie de tareas diferentes soportadas por el programa. Esto ofrece una gran versatilidad a los programas.
(V,ase el apartado dedicado a la Cola de Ordenes).

Por supuesto, también podemos hacer programas que no acepten ningún par metro, por lo tanto, se ignorar cualquier información pasada al programa mediante la Cola De Ordenes, y se ejecutar el programa sin mas.
De hecho es así como vamos a trabajar en un principio, ya que los programas que empecemos a hacer ser n tan concisos que sólo realizar n una determinada tarea.
Conforme avancemos en el Curso, ya tendremos tiempo de realizar programas suficientemente complejos que necesiten de par metros para seleccionar una de sus múltiples tareas ofrecidas.
Pero bueno, eso ser mas adelante. Volvamos ahora a lo que nos toca... Hemos dicho que la Cola de Ordenes es pasada al programa para que ,ste sepa la tarea que debe realizar. Ahora la cuestión es: ¨Dónde se almacena dicha Cola de Ordenes para que el programa pueda acceder a ella? La respuesta a este interrogante nos lleva a otro de los apartados de esta lección: El Prefijo de Segmento de Programa (PSP). Como adelanto, decir que el PSP es una zona de datos de 256 bytes (100H bytes) utilizada para diferentes propósitos: Almacenar la Cola de Ordenes, servir de Buffer de Fichero por defecto, resguardar ciertos vectores de interrupción, etc.. Para cada programa en memoria (ya sea transitorio ó residente) existe un PSP asociado.
(V,ase el apartado dedicado al PSP).
Aparte del PSP, el DOS ofrece a cada programa otra estructura de datos llamada Bloque de Entorno (Environment Block). Este Bloque de Entorno contiene información acerca de distintas Variables de Entorno, como son el Path, Prompt y otras tantas. Ademas de estas variables, el Bloque de Entorno ofrece el nombre del programa dueño del PSP y de dicho Bloque. Aunque no lo parezca en un principio, esta última información nos puede ser muy útil en determinados programas.
(V,ase el apartado dedicado al Bloque de Entorno).

- PROGRAMAS .COM ---------- El nombre de COM viene de 'Copy Of Memory', y quiere decir algo así como que el contenido del fichero COM formado por las instrucciones y datos que componen el programa, es una copia exacta del programa una vez cargado en memoria.
Los programas COMase cargan en memoria a partir de la dirección 100h, justo a continuación del PSP. Por tanto, cuando creemos nuestro programa COM debemos indicarle al ensamblador que utilicemos (MASM, TASM, etc) que nuestro programa empezar a partir de dicha dirección 100H. Esto lo hacemos mediante la pseudo_instrucción ORG 100H. Esta instrucción no genera ningún código ejecutable, simplemente le indica al ensamblador con el que estamos trabajando que el código que genere debe empezar (ORiGen) en la dirección 100h.
Todos los accesos a variables, saltos en el programa, etc.. se har n teniendo en cuenta que el programa empieza en la dirección 100h, y no en la 00h. Si no utiliz ramos la instrucción ORG 100h, el código ejecutable resultante estaría construido en base a una dirección de comienzo 00h. Al cargar el programa en memoria para su ejecución (a partir de la dirección 100h), habría 100h bytes de diferencia en todos los accesos a memoria, saltos, llamadas a procedimientos, etc.
Otra cosa importante a tener en cuenta es la pila. Cuando el DOS carga un programa COM en memoria para su ejecución, sitúa la pila justo al final del segmento en el que se carga el programa. Vamos a ver cómo quedaría el programa en memoria mediante un gr fico:

CS:0000 -Â->Ú----------¿
DS:0000 -´ ³ ³
ES:0000 -´ ³ Prefijo de Segmento de Programa (PSP) ³

SS:0000 -Ù ³ 256 (100h) Bytes ³ Ú->Ã----------´
CS:0100 -Ù ³ ³
(CS:IP) ³ Código y datos del Programa ³ ³ ³
CS:FFFF -¿ ³ ³
DS:FFFF -´ ³ ³
ES:FFFF -´ ³ Pila (Stack) ³ ³ ³ ³ Ã->À----------Ù ³
SS:FFFF -Ù
(SS:SP)
Todos los registros de Segmento, incluido SS (registro de Segmento de Pila) se inicializan con valor 0, apuntando así al principio del segmento donde se carga el programa, en definitiva, apuntando al principio del PSP, ya que dicho PSP se sitúa justo al principio del segmento.
El registro IP (Puntero de Instrucción) se inicializa con valor 100h para que apunte a la primera instrucción del programa. La primera instrucción del programa se encuentra justo despues del PSP y normalmente suele ser un salto (JMP). Esto es así ya que los datos suelen estar antes que el código del programa.
É»º PSP º Lo enmarcado entre
CS:0100 -->Ì¹ -¿ esta llave es el
(CS:IP) º Salto hacia el CODIGO DEL PROGRAMA º ³ contenido del programa Ç----------¶ ³ antes de cargarlo en º DATOS DEL PROGRAMA º ³ memoria para su Ç----------¶ Ã- ejecución. º º ³ Una vez que se carga º CODIGO DEL PROGRAMA º ³ un programa en memoria º º ³ para su ejecución, º º ³ el PSP y la PILA se Ì¹ -Ù consideran parte de º PILA º dicho programa. È¼

Hemos dicho que los datos suelen estar antes que el código en el programa. Hay varios motivos para que esto sea así, por una parte es mas cómodo para el programador tener los datos al principio del programa, es obvio, ¨no? El compilador también agradecer que se definan los datos antes de referirse a ellos en el código.
Ahora nos surge un problema: Si situamos los datos al principio del programa (Offset 100h) el procesador tomar estos datos como instrucciones y las ejecutar , es decir, -se ejecutarían los datos! Para remediarlo, al principio del programa incluimos una instrucción de salto (JMP) hacia el código del programa, saltando así los datos.
;***** Ú----------¿ JMP Codigo_Programa ³ Salto hacia el CODIGO DEL PROGRAMA ³ ;Inicio_Datos Ã----------´ ;[Datos del programa] ³ DATOS DEL PROGRAMA ³ ;Fin_Datos ³ ³ Ã----------´
Codigo_Programa: ³ ³ ;Inicio_Codigo_Programa ³ ³ ;[Codigo del Programa] ³ CODIGO DEL PROGRAMA ³ ;Fin_Codigo_Programa ³ ³ À----------Ù
;*****
Unos p rrafos mas arriba hemos dicho que la pila se sitúa justo al final del segmento (El registro SP apunta al Offset 0FFFFh). Ya sabemos de otras lecciones que la pila crece (mediante los sucesivos PUSH's) hacia direcciones mas bajas de memoria. Tenemos entonces que la pila crece en dirección al final del programa, el cual se encuentra al principio del segmento. Es importante tener esto presente, ya que puede ser motivo de graves errores. Aunque no es normal, se puede dar el caso de que al crecer la pila debido a múltiples Apilamientos (PSUH's), ,sta machaque el código del programa. Esto puede suceder en determinados casos como:
- El código del programa COM es muy grande, ocupa casi todo el segmento. Entonces, por muy poco que crezca la pila, acabar machacando dicho código del programa.
- Aunque el código del programa no sea demasiado extenso, el uso que se hace de la pila es excesivo (mediante apilamientos). Por ejemplo, en funciones recurrentes(*) que pasan

los par metros a través de la pila. En estos casos, la pila puede crecer tanto que acabe machacando al programa, por pequeño que ,ste sea.
(*) Una función recurrente es aquella que puede invocarse a sí misma. Si no se depura bien dicha función, puede derivar en infinitas llamadas a sí misma. En cada una de estas llamadas, la pila crece, de tal manera que al cabo de unas cuantos cientos o miles de estas llamadas, la pila acaba machacando al programa.
- Etc...

Recapitulemos.. Tenemos un sólo segmento para el programa COM. Los primeros 256 (100h) bytes de dicho segmento están ocupados por el PSP. A continuación nos encontramos con el programa, y al final del segmento tenemos la Pila, la cual crece en dirección al programa. En un primer momento, todos los registros de segmento (DS, CS, ES y SS) apuntan al principio del segmento (Offset 0000h). El registro IP apunta al Offset 100h, primera instrucción del programa, justo a continuación del PSP. Como los datos del programa se suelen depositar al principio del mismo, dicha instrucción situada en el Offset 100h suele ser un salto hacia el principio del código del programa. El registro SP (Puntero de Pila) apunta al Offset 0FFFFh (último byte del segmento). La pila crece de direcciones mas altas 0FFFFh hacia direcciones mas bajas.
Una característica importante relacionada con la forma en que el DOS le cede el control a los programas COM es la siguiente:
Una vez que un programa COM toma el control, el DOS reserva toda la memoria libre para este programa. Es decir, por muy pequeño que sea el programa COM, el DOS le d toda la memoria libre del sistema.
En la próxima lección ampliaremos información relacionada con este punto, inconvenientes que conlleva y soluciones.

- PROGRAMAS .EXE ---------- A diferencia de los programas COM (los cuales cuentan como m ximo con un segmento (64 Ks) para código, datos y pila, es decir, para todo el programa), los programas EXE disponen de toda la memoria del Area de Programas Transitorios (TPA) para su uso.
En un programa EXE, los datos, pila y código se definen en segmentos independientes. Se utiliza un segmento distinto para cada una de esas tres principales estructuras. Aunque, en realidad, podemos tener varios segmentos de datos, cada uno accesible de forma independiente. (Ver modelo de programa EXE).
El fichero EXE cuenta con una cabecera que le indica al DOS como ubicar cada uno de los diferentes segmentos definidos en memoria. Esta cabecera la proporciona el programa LINK, nosotros no debemos preocuparnos por ella.
Una vez que el DOS ha cargado el programa EXE en memoria para su ejecución, ,ste quedaría de la siguiente manera:

ES:0000 -¿ Ã-> Ú----------¿
DS:0000 -Ù ³ Prefijo de Segmento de Programa (PSP)³ Ã----------´ ³ ³
CS:IP ---> ³ Segmento de Código del Programa ³ ³ ³ Ã----------´ ³ ³ ³ Segmento de Datos ³ ³ ³
SS:0000 ---> Ã----------´ ³ ³ ³ Segmento de Pila ³ ³ ³
SS:SP ---> À----------Ù
Como podemos ver, el PSP se sitúa al principio de todo segmento de programa, como ocurría con los programas COM.
En un principio ES y DS apuntan al PSP, mas tarde deberemos hacer que DS apunte a nuestro segmento de datos para poder acceder a ,stos.

El par de registros CS:IP apuntan a la primera instrucción de nuestro programa. Esta primera instrucción a ejecutar viene dada por la pseudo_instrucción END (Fin de Programa). (Ver el apartado dedicado a las pseudo_instrucciones).
El par de registros SS:SP apuntan a la base de la pila (ya que aún no hay ninguno), y a la vez apuntan a la cima de la pila (ya que el primer elemento que se introduzca en la pila se har según la dirección de SS:SP).
En el gr fico vemos que los tres segmentos (código, datos y pila) siguen este orden, pero eso no tiene por que ser así. Dependiendo de la memoria libre que haya en el sistema, y la distribución de esa memoria libre, estos tres segmentos pueden estar en cualquier posición de la memoria, y en cualquier orden.
Una vez que nuestro programa toma el control, har accesible su segmento de datos (como podemos ver en el modelo y en el ejemplo de progs EXE), obteniendo la siguiente representación gr fica:

ES:0000 ---> Ú----------¿ ³ Prefijo de Segmento de Programa (PSP)³ Ã----------´ ³ ³
CS:IP ---> ³ Segmento de Código del Programa ³ ³ ³
DS:0000 ---> Ã----------´ ³ ³ ³ Segmento de Datos ³ ³ ³
SS:0000 ---> Ã----------´ ³ ³ ³ Segmento de Pila ³ ³ ³
SS:SP(**)--> À----------Ù

(**) En caso de haber utilizado algún PUSH en las instrucciones de inicialización, SS:SP no apuntarían al final del segmento de pila como muestra el dibujo, sino unas posiciones mas hacia el inicio de la pila:
SS:0000 ---> Ã----------´ ³ ³ ³ Segmento de Pila ³ SS:SP ---> ³ ³ À----------Ù

Cabe resaltar que con los programas EXE no tenemos el inconveniente que se nos plantea con los programas COM (relativo al consumo total de la memoria al cargarse para su ejecución). Gracias al uso de la cabecera con que cuentan los programas EXE, el DOS sólo reserva para dichos programas la cantidad justa de memoria que se necesita para ubicar cada uno de sus segmentos.

En la próxima lección ampliaremos información relativa al apartado de programas COM y EXE. Por ahora ya es bastante extensa la lección.

- Prefijo de Segmento de Programa (PSP) ----------
Estructura del PSP:
0000H Ú----------¿ ³ Int 20h (Terminar Programa) ³ (2 Bytes) 0002H Ã----------´ ³ Dirección de Inicio del Segmento ³ (2 Bytes) ³ que hay a continuación del Programa ³ 0004H Ã----------´ ³ (Reservado) ³ (1 Byte) 0005H Ã----------´ ³ Llamada lejana (Far-Call) al ³ (5 Bytes) ³ Distribuidor de funciones del DOS ³ 000AH Ã----------´ ³ Resguardo del vector de interrupción ³ (4 Bytes) ³ Int 22h (Gestor de Terminación) ³ 000EH Ã----------´ ³ Resguardo del vector de interrupción ³ (4 Bytes) ³ Int 23h (Gestor de CTRL+C) ³ 0012H Ã----------´ ³ Resguardo del vector de interrupción ³ (4 Bytes) ³ Int 24h (Gestor de Errores Críticos) ³ 0016H Ã----------´ ³ (Reservado) ³ (22 Bytes) 002CH Ã----------´ ³ Dirección del Segmento de Entorno ³ (2 Bytes) 002EH Ã----------´ ³

(Reservado) ³ (46 Bytes) 005CH Ã----------´ ³ Primer FCB (Bloque de Control de ³ (16 Bytes) ³ Fichero) por defecto ³ 006CH Ã----------´ ³ Segundo FCB (Bloque de Control de ³ (20 Bytes) ³ Fichero) por defecto ³ 0080H Ã----------´ ³ Par metros pasados al programa ³ (128 Bytes) ³ y ³ ³ DTA (Area de Transferencia de Disco) ³ ³ por defecto ³ À----------Ù Total: (256 Bytes)

* Offset 0000H * El primer campo del PSP contiene una instrucción código maquina (20CD). Se trata de la instrucción (Int 20h). Esta instrucción genera la Interrupción 20h, utilizada para terminar la ejecución del programa. Esta instrucción (como ya vimos en la lección 10) ha quedado obsoleta, sustituy,ndose su uso por la función 4Ch de la INT 21h. Por tanto no le daremos mayor importancia a este primer campo del PSP.

* Offset 0002H * En este campo se almacena la dirección del siguiente segmento de memoria a continuación de nuestro programa.

Ú----------¿ -¿ ³ PSP ³ ³ ³ Código, datos ³ Ã- Programa ³ Pila ³ ³ À----------Ù -Ù Ú----------¿ -¿ --> Inicio del siguiente segmento. ³ Segmento ajeno a ³ ³ ³ Nuestro Programa ³ ³ ³ ³ Ã- Siguiente segmento al programa. ³ ³ ³ À----------Ù -Ù

Mediante este campo podemos saber el tamaño del bloque de memoria en el que se ha cargado el programa. Restando a la dirección de segmento almacenada en el offset 0002h la dirección de inicio del segmento donde se ha cargado el PSP, tenemos el tamaño del bloque (en p rrafos) que contiene a nuestro programa.

Si multiplicamos ese valor por 16 (un p rrafo=16 bytes) obtendremos el tamaño (en bytes) de memoria que ha suministrado el DOS para nuestro programa.

* Offset 0004H * Campo Reservado.

* Offset 0005H * Aquí nos encontramos con una curiosa forma de acceder a las funciones de la INT 21h. Este m,todo de acceso que vamos a ver ha quedado obsoleto, pero se sigue manteniendo en el PSP por motivos de compatibilidad.

Se trata de una llamada lejana (FAR-CALL) al distribuidor de funciones del DOS. Este distribuidor de funciones es una rutina que ejecuta una de las funciones(*) de la INT 21H. La función a ejecutar en este caso se indica mediante el registro CL, y no AH (como es costumbre).

(*)Mediante este tipo de llamadas sólo se puede acceder a las funciones numeradas de la 00h a la 24h. Es decir, CL sólo debe contener un número comprendido entre el 00h y el 24h al realizar este tipo de llamadas a la INT 21H.

* Offset 000AH * En estos 4 bytes se almacena el contenido del vector de interrupción 22h, es decir, la dirección donde comienza la rutina de atención a la INT 22H. De esta manera, aunque durante la ejecución del programa se modifique el valor de este vector de interrupción, este campo (000AH) sirve para resguardar el valor original.

El vector INT 22h contiene la dirección de la rutina que recibe el control cuando se finaliza el programa mediante uno de los siguientes m,todos: - INT 20H - INT 27H - INT 21H (funciones 00H, 31H, 4CH)

* Offset 000EH * En estos 4 bytes se almacena el contenido del vector de interrupción 23h, es decir, la dirección donde comienza la rutina de atención a la INT 23H. La INT 23h se ejecuta cada vez que el DOS detecta la pulsación de la combinación de teclas CTRL+C, y provoca la interrupción del programa en curso.

Si la variable de sistema BREAK est con valor OFF (BREAK=OFF), la detección de CTRL+C sólo se produce en las funciones de Entrada/Salida de caracteres. Si (BREAK=ON), ademas de en dichas funciones de E/S, se comprobar la pulsación de CTRL+C en la mayoría de las restantes funciones del DOS.

En muchos programas se deshabilita el efecto de la INT 23H (CTRL+C) para que el usuario no pueda interrumpir dicho programa. Mediante el campo 000EH, el DOS se

asegura que al salir del programa en curso se mantenga el antiguo valor del vector INT 23H.
* Offset 0012H * En estos 4 bytes se almacena el contenido del vector de interrupción 24h, es decir, la dirección donde comienza la rutina de atención a la INT 24H. La INT 24h contiene la dirección del Gestor de Errores Críticos. El Gestor de Errores Críticos recibe el control (mediante la INT 24H) cada vez que se produce un Error Crítico. Ejemplos de errores críticos son: - Intentar escribir en una disquetera vacía (sin disquete), - Intentar escribir en un disquete protegido contra escritura, - Error de CRC (Código de Redundancia Cíclica) en los datos leidos/escritos. - Que la impresora se quede sin papel cuando se le manda imprimir, - etc.
(Cuando estudiemos Programación de Residentes, trataremos en profundidad esta INT 24h, la cual nos ser muy útil y necesaria).
* Offset 002CH * En este campo se almacena la dirección de inicio del segmento de memoria que contiene el Bloque de Entorno. (Ver el apartado - Bloque de Entorno - para mas información).
* Offset 005CH * Este campo contiene al primer Bloque de Control de Fichero (FCB) por defecto. Este FCB est compuesto por varios campos: - Unos que ofrecen variada información acerca de un determinado fichero, - Y los restantes que se utilizan para el Control del Fichero.
El m,todo de acceso a ficheros mediante FCB dejó de utilizarse a partir de la versión 2.0 del MS-DOS, en favor del m,todo Handle (mucho mas cómodo y vers til). Todas las funciones de manejo de ficheros que vimos en la lección 10 se basan en el m,todo Handle. No merece la pena (al menos en un principio) siquiera enumerar las funciones FCB.
Si se sigue incluyendo soporte FCB en el DOS es simplemente por motivos de compatibilidad con programas antiguos. Veamos la estructura de un FCB (por curiosidad):
0000H Ú----------¿ ³ Identificador de la Unidad ³ (1 Byte) ³ (A, B, C, etc) ³ 0001H Ã----------´ ³ Nombre del fichero ³ (8 Bytes) 0009H Ã----------´ ³ Extensión del fichero ³ (3 Bytes) 000CH Ã----------´ ³ Número de este FCB ³ (2 Bytes) 000EH Ã----------´ ³ Tamaño de Registro ³ (2 Bytes) 0010H Ã----------´ ³ Tamaño de Fichero ³ (4 Bytes) 0014H Ã----------´ ³ Fecha de Creación o ³ (2 Bytes) ³ Actualización del Fichero ³ 0016H Ã----------´ ³ Hora de Creación o ³ (2 Bytes) ³ Actualización del Fichero ³ 0018H Ã----------´ ³ (Reservado) ³ (8 Bytes) 0020H Ã----------´ ³ Número de Registro Actual ³ (1 Byte) 0021H Ã----------´ ³ Número de Registro Relativo³ (4 Bytes) À----------Ù
Todos los campos donde aparece la palabra 'Registro' se emplean para leer/escribir de forma aleatoria (no secuencial) en el fichero.
En el m,todo Handle (el que se utiliza a partir de la v 2.0 del MS-DOS) se emplea la función 42h de la INT 21h para desplazarse por el fichero, y luego las funciones de lectura/escritura con el tamaño de bloque (Registro en FCB) a leer/escribir.
* Offset 006CH * Este campo contiene al segundo Bloque de Control de Fichero (FCB) por defecto. (Lo indicado para el campo anterior es aplicable a ,ste).
* Offset 0080H * Este campo cumple dos cometidos: - Almacena la Cola de Ordenes que el usuario le ha pasado al programa. - Sirve como buffer de fichero por defecto (DTA por defecto).
El problema es que los 128 bytes de este campo no se reparten entre la Cola de Ordenes y el DTA por defecto, sino que ambas informaciones se solapan. Ambas estructuras de datos usan estos 128 bytes por separado:
Este campo contendr el contenido de la Cola de Ordenes hasta que se produzca la primera transferencia de datos a/desde un fichero (usando el m,todo FCB). Es decir, en

primer lugar este campo almacena la Cola de Ordenes. El programador lo que debe hacer es salvar el contenido de este campo (Cola de Ordenes) a la zona de datos del programa antes de realizar el primer acceso a ficheros mediante el m,todo FCB. De esto se desprende que este campo es provisional para almacenar la Cola de Ordenes, quedando destinado a realizar las veces de buffer de fichero por defecto (DTA por defecto).
Teniendo en cuenta que las funciones FCB han quedado obsoletas y no se deben utilizar salvo casos excepcionales, el problema de solapamiento expuesto no se debe tener en cuenta, ya que al no ser invocada ninguna función FCB, la Cola de Ordenes no ser 'machacada'.
De cualquier manera (y sobre todo, para los casos en que se utilice alguna función FCB) suele ser una buena pr ctica salvar la Cola de Ordenes a un 'lugar seguro' dentro de la zona de datos del programa.
(V,ase el apartado dedicado a la Cola de Ordenes).

- Bloque de Entorno (Environment Block) ---------- El Bloque de Entorno es una zona de memoria que cumple dos cometidos bien diferenciados. Por una parte, almacena las distintas Variables de Entorno, así como el valor de dichas variables. Por otra parte, ofrece una cadena ASCIIZ con la vía de acceso al programa dueño de este Bloque de Entorno.
Las Variables de Entorno se sitúan al principio del Bloque de Entorno, separadas unas de otras por el byte 00h. Al final de la última Variable de Entorno se sitúan dos bytes con valor 00h que indican el fin de las Variables de Entorno y el comienzo del nombre de programa.
A continuación de los dos bytes con valor 00h que indican el final de las Variables de Entorno, nos encontramos con otros dos bytes antes de poder acceder al nombre del programa. Estos dos bytes (1 palabra) contendr n el valor 0001h si el programa no se trata del COMMAND.COM, por tanto para acceder al nombre del programa, simplemente tenemos que buscar el valor 01h desde el principio del Bloque de Entorno.
Vamos a tratar el tema desde un punto de vista pr ctico. Lo que aparece a continuación es el contenido del Bloque de Entorno de un programa de prueba que he hecho para tal efecto.

```
;***Ejemplo de Bloque de Entorno.
COMSPEC=C:\DOS\COMMAND.COM PATH=C:\;C:\WINDOWS;C:\UTIL;
C:\DOS;C:\SANBIT60;C:\RATON;C:\ENSAMBLA;C:\SCAN PROMPT= $P$G
TEMP=C:\WINDOWS\TEMP CLIPPER=f:50 _ C:\CURSOASM\BE.COM
;***Fin del Ejemplo de Bloque de Entorno.
```

En el bloque de Entorno pasado al programa de prueba (BE.COM) podemos ver en primer lugar la cadena de variables de Entorno con sus respectivos valores.
1.- COMSPEC=C:\DOS\COMMAND.COM 2.- PATH=C:\;C:\WINDOWS;C:\UTIL;C:\DOS;C:\SANBIT60;
C:\RATON;C:\ENSAMBLA;C:\SCAN 3.- PROMPT=PG 4.- TEMP=C:\WINDOWS\TEMP 5.- CLIPPER=f:50
Las tres primeras variables son variables de Sistema. La primera de ellas COMSPEC indica la vía de acceso al COMMAND.COM o cualquier otro Int,rprete de Comandos que indiquemos, como por ejemplo el 4DOS. La segunda y tercera variable son de sobra conocidas por todos, ¨no?
Las variables 4 y 5 se definen mediante la orden SET dentro del AUTOEXEC.BAT. Mediante esta orden SET podemos crear variables de Entorno a nuestro gusto. Es otra forma de indicarle una determinada configuración a un programa.

Por ejemplo: Hemos creado un programa llamado HEXA.COM, el cual puede ejecutarse en idioma Ingl,s o Español. Ademas, el programa necesita un buffer de disco para su trabajo. Este buffer tendr una longitud a gusto del usuario. Pues bien, hay varias maneras de hacerle llegar toda esta información al programa. Principalmente, podemos:
- Utilizar la Cola de Ordenes para pasarle los par metros oportunos al programa. Esta sería la opción mas cómoda y 'Elegante'.
- Utilizar un fichero de configuración desde donde el programa tomaría esta información. Esta opción se suele utilizar cuando hay demasiados par metros a tener en cuenta en el programa. Como por ejemplo: Tarjeta gr fica utilizada, Utilización o no del ratón, Utilización o no de Joystick, datos personales del usuario, etc.. En programas que requieren de tantos datos, es imprescindible la utilización de un fichero de configuración.
- Finalmente, podemos hacerle llegar los par metros a través de una variable de Entorno. Esto lo haríamos incluyendo una orden SET dentro del AUTOEXEC.BAT con el formato: SET VARIABLE_ENTORNO=CONTENIDO_DE_LA_VARIABLE. En el ejemplo que estamos tratando (HEXA.COM), la orden SET quedaría de la forma: SET HEXA=ESP;64 Dicha orden indicaría que va a haber una nueva variable de entorno que se va a llamar HEXA, cuyo contenido son dos informaciones separadas por el car cter (;). La primera información indica el idioma (ESP). La segunda información indica que el usuario quiere un buffer de 64 Ks.

La ventaja que tiene utilizar una variable de Entorno para pasar información a los programas es que esa información va a ser accesible por cualquier programa que lo desee. De esta forma si tenemos una aplicación con varios programas que necesitan de una información para su buen funcionamiento, ,sta sería una buena opción. No lo es tanto crear una variable de entorno para ser utilizada por un sólo programa. Por tanto, en el ejemplo del programa HEXA.COM la opción de la Variable de Entorno no sería la mas acertada. Pero bueno, ahí est la información, y que cada uno la utilice a su gusto.

Volvamos a generalizar... Cada una de las variables de Entorno se separa de las contiguas por medio de un byte separador con valor 00h. En el ejemplo puede parecer que ese byte separador sea un simple car cter de espacio, pero no es así, lo que ocurre es que el código 00h se representa en pantalla como un car cter en blanco, al igual que ocurre con el código 32 (Espacio). Bueno, ya que estamos... :-) Hay un tercer car cter que se representa en pantalla como blanco: el código 255.

Un viejo truco para crear directorios 'Ocultos' consiste en insertar códigos 255 en el nombre de directorio. Así, un usario ajeno a nuestro sistema, creer que estos caracteres son blancos y no conseguir introducir correctamente el nombre de directorio.

Era gracioso ver cómo los profesores intentaban entrar en directorios sin poder conseguirlo. Ponían cada cara... :-)))

Bien... El grupo de variables de Entorno se cierra con dos bytes con valor 00h. Tras estos dos bytes nos encontramos con una palabra con valor 0001h. Al tratarse de una palabra, se almacena en memoria de la forma 0100h. Ya vimos en lecciones anteriores el porque de esta peculiar manera de almacenar las palabras en memoria.

Si echamos un vistazo al bloque de Entorno del ejemplo, veremos el monigote sonriente (código 01h), luego un car cter en blanco, que es en realidad un código 00h. Y a partir de ahí tenemos la cadena ASCIIZ con la vía de acceso al nombre del programa. En este caso, tal cadena es C:\CURSOASM\BE.COM

A continuación se amplía información pr ctica relacionada con el Bloque de Entorno mediante dos programas ampliamente comentados. Estos dos programas son en realidad uno sólo escrito dos veces, una vez en formato de programa COM; y otra vez escrito en formato EXE. Sirvan estas dos versiones para ver diferencias y similitudes entre la forma de crear programas COM y programas EXE.

- Cola de Ordenes ---------- Como hemos visto a lo largo de las referencias que hemos hecho a la Cola de Ordenes en el resto de la lección, dicha estructura de datos no es mas que el conjunto de par metros que se le pasan al programa al ejecutarlo. Pues bien, lo dicho anteriormente hay que ampliarlo:
Cuando el DOS le pasa los par metros al programa a través del Offset 80h del PSP, no sólo le pasa todos y cada uno de los par metros tal y como los ha introducido el usuario a continuación del nombre del programa, sino que incluye un byte que precede a toda la cadena de par metros, el cual indica el tamaño (en bytes) de dicha cadena de par metros.
Es decir, en el Offset 80h del PSP encontramos un byte que nos indica el tamaño de la cadena de par metros que vamos a encontrar a continuación, a partir del Offset 81h.
Luego la Cola de Ordenes est formada por un byte contador, en el Offset 80h; y la cadena de par metros que ha introducido el usuario, a partir del Offset 81h.
Retomemos el ejemplo del programa HEXA.COM... Lo queríamos ejecutar en Español (Castellano para los ling istas :-) y con un buffer de 64 Ks. Ya habíamos visto cómo se le podía pasar esta información (par metros) al programa mediante una Variable de Entorno. Ahora veremos cómo se hace mediante par metros en la línea de Comandos (COMMAND.COM), que es lo normal:
HEXA ESP;64
La línea anterior ejecuta el programa HEXA con dos par metros separados por el símbolo ';'. En realidad, sólo el usuario y el programa saben que se trata de dos par metros separados por tal símbolo, ya que el DOS sólo reconoce una cadena de par metros de 7 bytes de longitud ' ESP;64'. Es decir, el DOS no reconoce varios par metros independientes, sino una cadena de par metros, por tanto los símbolos de separación como ';', '/', etc... forman parte de la sintaxis que imponga el programa en sí. Para el DOS cada uno de estos símbolos son simplemente un byte mas en la cadena de par metros.
Una vez ejecutado el programa HEXA con los par metros anteriores, el campo 80h del PSP contendría la Cola de Ordenes tal como sigue:
_ ESP;64
Podemos ver en primer lugar el símbolo '_'. Este símbolo tiene el código ASCII 07h. Esto quiere decir que la cadena de par metros que vamos a encontrar a continuación (Offset 81h) tiene una longitud de 7 bytes.
A continuación del byte contador '_' encontramos la cadena de par metros introducidos por el usuario: ' ESP;64'.
Podemos observar que el espacio que separa el nombre del programa de la cadena de par metros se considera parte de la cadena de par metros. En definitiva, cualquier car cter que se introduzca a continuación del nombre del programa se considera par metro. Esto no parece que tenga mucho sentido en un principio, ya que si despues del nombre del programa introducimos un car cter distinto de espacio, este nombre de programa cambia, ya no es el mismo, por tanto no estamos ejecutando el mismo programa. Es decir, si despues del nombre de programa HEXA introducimos un car cter distinto de espacio (por ejemplo, el car cter 'C'), ya no estamos intentando ejecutar el programa HEXA, sino el programa HEXAC. Por tanto, el único car cter que en un principio debería introducirse a continuación del nombre del programa sería el espacio. Siendo esto así, no se debería considerar parte de la cadena de par metros, ya que sería información inútil. Mejor dicho, no sería información, ya que de antemano sabríamos que el primer car cter iba a ser un espacio en blanco...
Como decía, esto no parece tener mucho sentido si no fuera porque hay ciertos caracteres especiales como '/' que el DOS acepta que se escriban justo a continuación del nombre del programa, y se consideran inicio de la cadena de par metros De tal forma que HEXA/ESP;64 ejecuta el programa HEXA con el siguiente valor en la Cola de Ordenes (Offset 80h): '_/ESP;64'

- Pseudo_Instrucciones ---------- Pseudo_Instrucciones son aquellas instrucciones que aparecen en el código fuente del programa y no generan código ejecutable alguno. Tienen como misión ofrecer cierta información al ensamblador necesaria durante el proceso de ensamblaje. Vamos a enumerar y estudiar las mas comunes:
--- Pseudo_Instrucciones de cara al listado del código fuente (PAGE y TITLE): Se utilizan para indicar el formato del listado del código fuente, en caso de que se solicite.
PAGE: La pseudo_instrucción PAGE le indica al ensamblador el número de líneas que tendr cada p gina del listado, así como el número de columnas de que constar cada línea. Sintaxis: PAGE X,Y Donde X es el número de líneas por p gina. Y es el número de caracteres por línea.
Los valores por defecto para PAGE son X=60, Y=132. O sea: PAGE 60,132
TITLE: Mediante esta pseudo_instrucción ofrecemos un 'título' o nombre de programa que ser utilizado como cabecera de p gina en caso de que solicite un listado del código fuente. Podemos utilizar cualquier texto que queramos, pero lo recomendable es utilizar el nombre del programa en cuestión, y a continuación (si lo deseamos) un comentario acerca del programa. Es decir, si nuestro programa se llama CAPTU y se trata de un capturador de pantallas, utilizaremos la pseudo_instrucción: TITLE CAPTU_Capturador de pantallas.
Disponemos de 60 caracteres de longitud para indicar el nombre del programa y cualquier comentario adicional.
Sintaxis: TITLE *Texto* Donde *Texto* es una cadena de 60 caracteres como m ximo, que por regla general constar del nombre del programa en primer lugar, y adicionalmente (si se desea), un comentario acerca del programa.

--- Pseudo_Instrucciones de definición de segmentos (SEGMENT y ENDS): Mediante estas dos pseudo_instrucciones definimos el principio y final de cada uno de los diferentes segmentos de nuestro programa:
- Segmento de código (siempre debemos definirlo, ya trabajemos con programas COM o programas EXE). Si estamos desarrollando un programa COM, ,ste ser el único segmento que debamos definir, ya que los restantes segmentos (Datos, Pila y Extra) coincidir n con el segmento de código, al disponer sólo de un segmento de tamaño para todo el programa.
- Si estamos construyendo un programa EXE, debemos definir también otro de los 3 segmentos restantes: el Segmento de Pila. El Segmento de Datos también ser imprescindible definirlo, a no ser que estemos construyendo un programa en el que no utilicemos variables propias. En cuyo caso no es necesario definir el Segmento de Datos, ya que no tenemos datos que incluir en el.
El segmento Extra no se suele definir, ya que normalmente se utiliza para acceder a variables y estructuras de datos ajenas al programa, de forma que su dirección de inicio se actualiza durante la ejecución del programa, según ,ste lo vaya requiriendo. Como decía, no se suele definir, pero podemos hacerlo. Podemos definir un segmento de datos Extra y asign rselo al registro ES. De esta forma podemos tener dos segmentos de datos en nuestro programa: Uno direccionado mediante DS y el otro (el Extra) direccionado mediante ES.
En realidad podemos tener incluso cientos de segmentos de datos en nuestro programa. Para acceder a cada uno de ellos utilizaremos la pseudo_instrucción ASSUME que veremos mas adelante.
Ú----------¿ ³ ³ ³ Nota: Al hablar de segmentos en esta ocasión, no me estoy refiriendo ³ ³ a 64 ks de memoria, sino a una

porción de memoria de un tamaño ³ ³ que puede ir de pocos bytes a 64 ks. ³ ³ En realidad no estamos hablando de segmentos propiamente dichos, ³ ³ sino de porciones de segmento, a las que tratamos como ³ ³ segmentos. El segmento de datos (por ejemplo) no tiene por que ³ ³ tener un tamaño de 64 ks. En realidad nos dar igual su tamaño ³ ³ a la hora de acceder a el. Sólo nos es necesario saber su ³ ³ comienzo (De eso se encarga el registro DS) y conocer la ³ ³ dirección dentro de ese segmento de la variable a la que queremos³ ³ acceder (basta con indicar el nombre de la variable para que ³ ³ el ensamblador genere su dirección real durante el proceso de ³ ³ ensamblado-linkado). ³ ³ ³
À----------Ù
Cada uno de los segmentos se definen según el siguiente formato:
Ú----------¿ ³ ³ ³ Nombre_seg SEGMENT [Opciones] ³ ³ ³ ³ . ³ ³ . ³ ³ . ³ ³ ³ ³ ³
Nombre_seg ENDS ³ ³ ³ À----------Ù
Nombre_seg es el nombre con el que vamos a referirnos al Segmento. Como vemos en el modelo, dicho nombre de segmento debe utilizarse como encabezamiento y final del segmento. Para indicar el inicio de segmento se utiliza la pseudo_instrucción SEGMENT, de la forma Nombre_seg SEGMENT [Opciones]. Para señalar el final del segmento utilizamos la pseudo_instrucción ENDS, de la forma Nombre_seg ENDS.
Mediante [Opciones] se engloban 3 tipos de informaciones adicionales que se le pueden pasar al ensamblador al realizar la definición de segmentos. Veamos cada uno de estos tipos:
- Tipo Alineamiento (Alignment type) Mediante esta opción le indicamos al ensamblador el m,todo que debe emplear para situar el principio del segmento en la memoria. Hay cuatro m,todos posibles:
PARA (La dirección de inicio del segmento ser múltiplo de 16. Este es el valor por omisión. Recordemos que un p rrafo es igual a 16 bytes, y que la dirección de inicio de un segmento se suele referenciar mediante un número de p rrafo, ya que se da por supuesto que esta dirección de inicio ser múltiplo de 16).
BYTE (Se tomar el primer byte libre como dirección de inicio del segmento).
WORD (La dirección de inicio del segmento ser múltiplo de 2).
PAGE (La dirección de inicio del segmento ser múltiplo de 256).
- Tipo Combinación (Combine type) Esta opción indica si se combinar el segmento que estamos definiendo con otro y otros durante el 'linkado'. Los valores posibles para esta opción son: STACK, COMMON, PUBLIC, AT expresión, MEMORY.
El valor STACK lo vamos a utilizar siempre que definamos el segmento de Pila.
Los siguientes valores se utilizan cuando se van a 'linkar' (fusionar) diferentes programas en uno sólo. En estos casos ser necesario utilizar los valores COMMON, PUBLIC, etc.. a la hora de compartir variables, procedimientos, etc.
- Tipo Clase (Class type) La opción Clase se indica encerrando entre apóstrofes (comillas simples) una entrada. Mediante esta entrada se pueden agrupar diferentes segmentos durante el proceso de 'linkado'.

--- Pseudo_Instrucción ASSUME: Mediante esta pseudo_operación relacionamos cada uno de los segmentos definidos con su correspondiente registro de segmento. Así, al segmento de datos le asignaremos el registro DS; Al segmento de código le asignaremos el registro CS; Al segmento de pila le asignaremos el registro SS; Aunque no es normal asignar inicialmente un segmento al registro ES, cabe la posibilidad de hacerlo por diversas razones: - Que queramos tenerlo apuntando a alguno de los tres segmentos principales: código, datos o pila. - Que hayamos definido un cuarto segmento y queramos direccionarlo mediante el registro ES. - Cualquier otra razón no incluida en las dos anteriores.

En caso de no asignar un segmento a un registro de segmento, caben dos posibilidades: - Se omite la referencia a dicho registro de segmento. - Se utiliza la partícula NOTHING para indicar que dicho segmento no ha sido asignado a ningún segmento.
Ú----------¿ ³ ³ ³ Pongamos un caso pr ctico: ³ ³ Definimos 3 segmentos: Uno de datos llamado DatoSeg; otro de pila ³ ³ llamado PilaSeg y otro de código llamado CodeSeg. ³ ³ Tras la definición de estos segmentos debemos asignarles el registro ³ ³ de segmento correspondiente de una de las siguientes formas: ³ ³ ³ ³ - ASSUME CS:CodeSeg, DS:DatoSeg, SS:PilaSeg ³ ³ ³ ³ - ASSUME CS:CodeSeg, DS:DatoSeg, SS:PilaSeg, ES:NOTHING ³ ³ ³ À----------Ù
Una vez mas recalcar que ASSUME es una pseudo_operación, no genera código ejecutable. Su función es sólo la de ofrecer información al lenguaje ensamblador.
Por tanto ASSUME debe utilizarse en combinación del par de instrucciones que aparecen enmarcadas mas abajo. En el ejemplo anterior, una vez que empezara el código del programa deberíamos incluir el siguiente par de instrucciones para que en realidad DS apuntara al segmento de datos que hemos indicado:
Ú----------¿ ³ ³ ³ MOV AX,DatoSeg ³ ³ MOV DS,AX ³ ³ ³ À----------Ù
No es necesario hacer lo mismo con CS ni SS, ya que el DOS lo hace por sí sólo al ejecutar el programa. Es decir, debe preparar CS para que apunte al segmento de código, sino no se ejecutaría el programa. Y debe preparar también los registros de pila.
En realidad el DOS prepara también los registros DS y ES para que apunten al PSP. Por tanto si queremos que DS apunte a nuestro segmento de datos tendremos que indicarlo como hemos visto arriba.
El ejemplo anterior pertenece a un programa EXE. Si fuera COM no haría falta definir segmento de pila, ya que el DOS sitúa la pila al final del segmento donde se carga el programa.
Tampoco deberíamos haber utilizado las dos instrucciones de arriba para asignar a DS su segmento, ya que al disponer sólo de un segmento para nuestro programa COM, todos los registros de segmento (CS, DS, ES y SS) apuntan al principio del segmento donde se carga el programa.

Pongamos ahora que tenemos dos segmentos de datos, uno llamado Dato_1_Seg y otro llamado Dato_2_Seg. En un principio queremos que DS apunte a Dato_1_Seg. Esto lo hacemos mediante la pseudo_instrucción ASSUME y luego el par de instrucciones que hemos visto antes:
ASSUME DS:Dato_1_Seg
MOV AX,SEG Dato_1_Seg MOV DS,AX
Tenemos ya los datos incluidos en el segmento Dato_1_Seg direccionables mediante DS. Pero en un procedimiento dado de nuestro programa debemos acceder a unas variables incluidas en el segundo segmento de datos: Dato_2_Seg. Debemos entonces hacer que DS apunte a este otro segmento de datos. En este caso, ya no debemos utilizar la Pseudo_instrucción ASSUME, ya que causaría error al Linkar. Simplemente debemos utilizar el par de instrucciones MOV tal como sigue:
MOV AX,SEG Dato_2_Seg MOV DS,AX
Si pasado un tiempo (unas instrucciones, mejor dicho :-) queremos que DS apunte a Dato_1_Seg... ... ya sabemos cómo hacerlo, ¨no?
En caso de que tuvi,ramos mas segmentos de datos (3, 6, incluso 100) para acceder a cada uno de esos segmentos de datos, simplemente debemos emplear el par de instrucciones MOV para que DS apunte al nuevo segmento. Es un poco pesado y lioso, pero así es el tema de los registros.

--- Pseudo_Instrucciones de definición de Procedimientos (PROC y ENDP): Mediante estas dos pseudo_instrucciones definimos el principio y final de cada uno de los diferentes procedimientos de que disponga nuestro programa.
Ya vimos en lecciones anteriores cómo se definían los procedimientos, cómo se llamaban, etc. Así que no insistiremos mas en este punto. Sólo decir que la pseudo_instrucción PROC le indica al ensamblador que a continuación comienza un procedimiento; y que la pseudo_instrucción ENDP le indica que ha finalizado el procedimiento. Sólo informan al ensamblador, no generan código ejecutable. Tomemos el siguiente modelo de procedimiento. Así como est , sin mas código dentro que el RET, sólo generar el código ejecutable C3H (RET).
Nombre_Proc PROC . . .
RET Nombre_Proc ENDP

--- Pseudo_Instrucción END: La pseudo_instrucción END se utiliza para indicarle al ensamblador donde acaba nuestro programa, de la misma forma que ENDS le indica donde acaba un segmento y ENDP le indica donde acaba un procedimiento.
Sintaxis: END NombreProg Donde NombreProg es el nombre del procedimiento principal de nuestro programa si hemos definido un procedimiento principal; ó es el nombre de la etiqueta que marca el inicio de nuestro programa. (Ver los modelos de programas COM y EXE).

--- Pseudo_Instrucción ORG: (Tratada en el apartado de programas COM de esta misma lección)

--- Pseudo_Instrucciones de definición de datos: Nos valdremos de estas pseudo_instrucciones para definir todas las variables y constantes propias de nuestro programa.
* Definición de variables * Para definir variables utilizaremos la inicial de la palabra Define (D), seguida de la inicial del tipo de dato que queramos definir: B(Byte), W(Word), D(Double_Word ó Doble palabra), Q(Cuadruple palabra), T(Diez bytes).
Tenemos entonces que: Para definir un BYTE utilizamos la pseudo_instrucción DB. Para definir una PALABRA utilizamos la pseudo_instrucción DW. Para definir una DOBLE_PALABRA utilizamos la pseudo_instrucción DD. Para definir una CUADRUPLE_PALABRA utilizamos la pseudo_instrucción DQ. Para definir una VARIABLE_DE_10_BYTES_DE_TAMA¥O utilizamos la pseudo_instrucción DT.
Hemos visto cómo definir el tipo de dato de nuestra variable. Veamos ahora cómo le ponemos el nombre a cada variable. En caso de ponerle nombre a la variable (lo mas normal, aunque no es obligatorio), dicho nombre estaría a la izquierda de la definición del tipo de dato. Es decir: Nombre_Variable Dx Donde 'x' es el tipo de dato de la variable (B, W, etc..). Así, si queremos definir una variable llamada CONTADOR de tamaño Byte, lo haremos de la siguiente forma: CONTADOR DB
Hasta aquí sabemos ya cómo ponerle nombre a una variable, y sabemos indicar el tipo de dato de que se trata, vamos a ver ahora cómo podemos asignarle un valor inicial a esa variable.
Es obligatorio darle algún valor a la variable que estemos definiendo. No podemos decir simplemente el tipo de dato de que se trata, tenemos que darle un valor inicial. Por tanto el ejemplo de arriba habría que completarlo d ndole un valor a la variable CONTADOR. Aún en el caso en que no nos importe el valor que tenga la variable en un principio, tendremos que indic rselo al ensamblador mediante el símbolo (?).
Retomamos entonces el ejemplo anterior, y vamos a darle a la variable CONTADOR un valor inicial de 210. Quedaría así la definición de la variable: CONTADOR DB 210

En el caso de que nos sea indiferente el valor inicial de la variable, lo indicaremos de la siguiente manera: CONTADOR DB ?
En mi caso concreto, yo nunca uso el símbolo (?). En los casos en que defino variables cuyo valor inicial me es indiferente, les doy siempre valor 0. Es decir, yo habría definido la variable CONTADOR de la siguiente manera: CONTADOR DB 0
Veamos por ejemplo que tipo de dato utilizaríamos para almacenar el valor num,rico 237654 (base 10) en una variable llamada ACUMULADOR. Este valor es demasiado grande para que quepa en un byte (valor m ximo= 255), también es demasiado grande para el tipo Word ó palabra (valor m ximo=65535), sin embargo en el tipo DobleWord ó Doble_Palabra sí que cabe perfectamente. O sea que la definición (ACUMULADOR DD 237654) sería correcta, mientras que las definiciones (ACUMULADOR DB 237654) y (ACUMULADOR DW 237654) son erróneas.
Bien, hasta aquí ya sabemos cómo definir variables simples, vamos a ver ahora cómo definir variables compuestas (cadenas de caracteres, vectores, tablas, etc).
Mediante una sóla definición de variable podemos crear varios elementos del mismo tipo de dato, consiguiendo así un vector o tabla de elementos. En definitiva, una cadena de caracteres es una tabla de una dimensión ó vector de elementos de tipo byte ó caracter.
Veamos en primer lugar la sintaxis completa para la definición de datos:
[Nombre_Variable] Dx Expresión
Nombre_Variable es el nombre de la variable que estamos definiendo. Los corchetes indican que es optativo dicho nombre. Si no queremos ponerle nombre a una variable no se lo ponemos, como ya hemos dicho antes.
A continuación aparece la Pseudo_instrucción Dx para la definición del tipo de dato, donde x indica el tipo de dato como ya hemos visto.
Y vamos a lo que queda: Expresión: Mediante Expresión englobamos el dato ó cadena de datos (vector ó tabla) asignado/a a una variable.
Expresión puede ser un sólo valor, por ejemplo 2367 (VAR DW 2367); Puede ser el símbolo (?) si el valor inicial del dato nos es indiferente (VAR DW ?);
Y puede ser, por otra parte, una cadena de caracteres ó una tabla de valores num,ricos o alfanum,ricos, la cual se puede definir de varias formas:
- Encerrar entre comillas la cadena de caracteres (si se trata de datos alfanum,ricos): VAR DB 'Esto es un ejemplo' Obs,rvese que el tipo de datos es BYTE (DB), ya que estamos definiendo elementos del tipo BYTE (caracteres). Tenemos una cadena de caracteres ó tabla de una dimensión llamada VAR de 18 elementos (la longitud total de la cadena 'Esto es un ejemplo'). Para acceder a cada uno de los elementos de la tabla tendremos que utilizar un índice a continuación del nombre de la tabla.
Nota: En ensamblador el primer elemento de una tabla ó cadena de caracteres es el elemento 0.

Veamos unos ejemplos: + Deseamos introducir en el registro AL el primer elemento de la tabla.
MOV AL,VAR Mediante esta simple instrucción se introduce el primer elemento de la cadena de caracteres en el registro AL, quedando AL = 'E', o lo que es lo mismo AL = 69. Como podemos ver, para el primer elemento no es necesario utilizar un índice, ya que en caso de omisión del mismo, el ensamblador entiende que se quiere acceder al primer elemento (elemento 0) de la tabla. La instrucción de arriba sería equivalente a MOV AL,VAR+0
+ En este segundo ejemplo queremos introducir en AL el tercer elemento de la cadena de caracteres.
MOV AL,VAR+2 Tras esta instrucción, AL quedaría con el valor que tuviera el tercer elemento de la tabla, dicho valor es 't', luego AL='t'.

Nota: El ensamblador reconoce el inicio de una tabla, pero nunca su final. Es decir, que si intentamos acceder al elemento número 300 de una tabla ó cadena de caracteres que sólo tiene 20 elementos, el ensamblador asume que esa tabla tiene por lo menos esos 300 elementos y accede a ese elemento 300, que por supuesto no pertenece a la tabla.
Veamos un ejemplo: Tenemos en nuestro segmento de datos las siguientes cadenas de caracteres:
;**** VAR DB 'Esto es un ejemplo' FILENAME DB 'C:\CURSOASM\MODECOM.ASM' ;****
Como podemos ver tenemos dos cadenas de caracteres ó tablas de una dimensión. La primera, llamada VAR, de 18 elementos. La segunda, llamada FILENAME, de 23 elementos.
En nuestro segmento de código nos encontramos con la siguiente instrucción:
MOV AL,VAR+33
En principio, esta instrucción parece errónea, ya que la cadena VAR tiene sólo 18 elementos, pero como decía antes, al ensamblador eso no le importa. Es el programador el que tiene que preocuparse de utilizar los índices correctos. Por lo tanto, tras la ejecución de esa instrucción, AL = 'D'.

- Separar cada uno de los elementos de la cadena mediante comas: (V lido para datos num,ricos y alfanum,ricos).
Así, la cadena del ejemplo anterior VAR DB 'Esto es un ejemplo' quedaría de esta forma como: VAR DB 69,115,116,111,32,101,115,32,117,110,32,101,106,101,109,112 DB 108,111
Como podemos ver, hemos descompuesto la cadena inicial en dos cadenas mas pequeñas para que cupieran en la pantalla. Vemos que a la segunda cadena no le hemos dado nombre. En realidad, es parte de la primera cadena, pero es necesario volver a definir el tipo de datos para que el ensamblador sepa lo que hay a continuación. Una vez que obtenemos el ejecutable, tendremos una tira de bytes ó cadena de caracteres con los valores 'Esto es un ejemplo'.
Otras formas de definir esta cadena podrían ser:
VAR DB 69,115,116,111 DB 32,101,115,32 DB 117,110,32,101,106 DB 101,109,112 DB 108,111
O también:
VAR DB 69,'s',116,'o' DB 32,'e','s',32 DB 117,110,' ',101,'j' DB 101,109,112,108,'o'
También esta otra:
VAR DB 'Esto e' DB 's',' ','un' DB 101,'j,101,109,112,108,'o'
Y así miles de formas de definir la misma cadena.
Veamos ahora un ejemplo en el que deseamos crear un vector de 7 elementos num,ricos con los siguientes valores iniciales: 278,8176,736,3874,7857,22338,76
El vector, al que vamos a llamar Tabla_num, lo definiremos de la siguiente manera:
Tabla_num dw 278,8176,736,3874,7857,22338,76
Al igual que en los ejemplos anteriores, podemos definirlo de miles de formas diferentes, como:
Tabla_num dw 278,8176,736 dw 3874,7857,22338,76
O también:
Tabla_num dw 278 dw 8176 dw 736 dw 3874 dw 7857 dw 22338 dw 76
Etc...
Veamos ahora cómo hay que utilizar los índices en una tabla de elementos de tipo Word ó palabra. Tenemos que tener en cuenta que una palabra equivale a dos bytes, luego para acceder al siguiente elemento de la tabla deberemos incrementar en 2 unidades el índice.
Supongamos que queremos introducir en AX el primer elemento de la tabla. Esto se haría con la siguiente instrucción:

MOV AX,TABLA_NUM ;AX=278
Si ahora queremos introducir el segundo elemento, usaríamos esta otra instrucción:
MOV AX,TABLA_NUM+2 ;AX=8176
Para introducir el quinto elemento, usaríamos esta otra instrucción:
MOV AX,TABLA_NUM+8 ;AX=7857
Etc...

- Utilizar la partícula DUP para crear repeticiones de un mismo valor ó de un conjunto de valores. (V lido para datos num,ricos y alfanum,ricos).
Mediante esta partícula DUP podremos crear repeticiones de un mismo valor de forma cómoda. A la hora de definir tablas o grandes estructuras de datos con el mismo valor o valor indefinido, esta es la mejor opción.
Supongamos que queremos definir un vector de 100 elementos de tipo BYTE con valor inicial (para cada uno de estos elementos) 37... Si lo hici,ramos según hemos visto antes (elemento por elemento, y separando por comas cada uno de los elementos) podríamos pasar mas tiempo definiendo las variables que creando el programa, ademas, obtendríamos un código fuente demasiado grande debido a la definición poco inteligente de las variables. Sin embargo utilizando la partícula DUP la definición propuesta en este supuesto quedaría así de sencilla:
VECTOR DB 100 DUP (37)
Si nos atenemos a la sintaxis establecida al principio del apartado, el campo Expresión estaría compuesto en este ejemplo por: 100 DUP (37) Lo cual quiere decir que reserve espacio para 100 datos del tipo definido anteriormente (Byte en este caso: DB), cada uno de estos datos tendr n el valor inicial 37.
El equivalente a la definición VECTOR DB 100 DUP (37), prescindiendo del DUP, sería algo así como:
VECTOR DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37 DB 37,37,37,37,37,37,37,37,37,37
Como podemos ver, es mucho mas cómodo utilizar la partícula DUP. Ademas, reduce sensiblemente el tamaño del código fuente de un programa.
Una característica muy importante de la partícula DUP es que acepta la recursividad. Es decir, que podemos utilizar un DUP dentro de otro DUP mas externo.
Veamos un ejemplo para aclararlo... Supongamos que queremos definir una cadena de caracteres, la cual estar formada por 30 subcadenas consecutivas, cada una de las cuales a su vez estar compuesta por 10 caracteres con valor 'A', seguidos de 30 caracteres con valor 'e', seguidos de 300 caracteres con valor 'S'.
La definición de esta cadena, sin utilizar la partícula DUP sería algo pesadísimo, y ocuparía demasiado código fuente. Mientras que utilizando la partícula DUP quedaría algo así como:
CADENA DB 30 DUP (10 DUP ('A'),30 DUP ('e'),300 DUP ('S'))

* Definición de constantes (Pseudo_instrucción EQU) * Mediante la Pseudo_instrucción EQU definiremos las constantes de nuestro programa, en caso de utilizar alguna. En primer lugar, decir que la definición de una constante no genera ningún dato en el programa ejecutable. Las constantes se utilizan por motivos de comodidad y parametrización en un programa.
Supongamos que hacemos un programa para la gestión de los presupuestos, dividendos, etc.. de un bufete de abogados. Este bufete de abogados en principio est formado por 3

personas. En nuestro programa utilizamos miles de veces instrucciones que operan con el número de personas del bufete, como son división de dividendos entre los miembros del bufete, etc. Si en nuestro programa utilizamos siempre el número 3 para indicar el número de miembros, que pasaría cuando entrara un cuarto miembro al bufete... Tendríamos que buscar por todo el programa las instrucciones que operan con el número de miembros y cambiar el 3 por el 4 para así actualizar el programa a la nueva realidad. Y de nuevo se nos plantearía el problema si entrara un nuevo miembro, o si se fuera uno de los que ya estaban.
La solución a esto es utilizar una constante en lugar de un número concreto. Así en caso de cambios, sólo es necesario cambiar el valor de la constante en su definición, y volver a ensamblar-linkar el programa. Ahorrandonos así buscar por todo el programa cualquier referencia al número de miembros.
Num_Miembros EQU 3
Mediante la línea de arriba definimos una constante llamada Num_Miembros, la cual en un principio tendr valor 3. En el resto del programa, cada vez que tengamos que utilizar el número_de_miembros en cualquier operación, no introduciremos el número 3, sino la constante Num_Miembros. De esta forma, en caso de variación en el número_de_miembros sólo ser necesario modificar el valor de la constante en su definición.
Como hemos dicho anteriormente, la definición de la constante no genera ningún dato en el código ejecutable. Lo que hace el ensamblador es sustituir este nombre de constante que utilizamos (en este caso Num_Miembros) por su valor asociado.

Var1 db 'E' Var2 db 38,73 Const EQU 219 Var3 db 87
Las definiciones de variables y constante anteriores generarían los siguientes datos en el código ejecutable:
E&IW
Como podemos ver, entre el código 73 (I) y el código 87 (W) no encontramos el valor 219 (Û), ya que al tratarse de una constante no genera código ejecutable.

- MODELOS DE PROGRAMAS ----------
* MODELO DE PROGRAMA COM *
;---------- inicio del programa ----------
PAGE 60,132 TITLE Modelo_COM ;CopyRight (C) 1995. Francisco Jesus Riquelme. (AeSoft)

CSEG SEGMENT PARA PUBLIC 'CODIGO' ASSUME CS:CSEG, DS:CSEG, SS:CSEG ORG 100H
AEsoft_Prog:
JMP AEsoft_Code ;Salto al código del Programa.
;****** INICIO DE LOS DATOS
;Aquí se definen los datos del programa.
;****** FIN DE LOS DATOS

;****** INICIO DEL PROGRAMA
AEsoft_Code:
; --¿ ; ³ ; Ã-- Aquí estar el programa principal. ; ³ ; --Ù
MOV AH,4CH ;Función de Terminación de Programa. MOV AL,00 ;Ejecución del programa exitosa. Para indicar error, dar a ;AL un valor distinto de 00h. INT 21H ;Ejecución de la función (Salir del programa actual).

```
;****** FIN DEL PROGRAMA

;****** INICIO DE LOS PROCEDIMIENTOS

;Aquí se sitúan cada uno de los procedimientos de que conste el programa.
Proc_1 PROC
;--¿ ; ³ ; Ã- Código del Procedimiento Proc_1 ; ³ ;--Ù
RET
Proc_1 ENDP
;***
Proc_2 PROC
;--¿ ; ³ ; Ã- Código del Procedimiento Proc_2 ; ³ ;--Ù
RET
Proc_2 ENDP

;***
Proc_n PROC
;--¿ ; ³ ; Ã- Código del Procedimiento Proc_n ; ³ ;--Ù
RET
Proc_n ENDP

;****** FIN DE LOS PROCEDIMIENTOS

CSEG ENDS END AEsoft_Prog
;------------------------------ fin del programa ------------------------
```

* MODELO DE PROGRAMA EXE *

```
;----------------------------- Inicio del Programa -------------------
PAGE 60,132 TITLE Modelo_EXE
;***** Inicio de Segmento de Pila
STACK SEGMENT PARA STACK 'PILA' DW 64 DUP (0) ;Reservado espacio para 64 palabras. Los lenguajes de ;alto nivel suelen reservar espacio para 1000.
STACK ENDS
;***** Fin de Segmento de Pila

;***** Inicio de Segmento de Datos
DSEG SEGMENT PARA PUBLIC 'DATOS'
;--¿ ; ³ ; Ã-- Aquí se definen los datos propios del programa. ; ³ ;--Ù
DSEG ENDS
;***** Fin de Segmento de Datos

;***** Inicio de Segmento de Código
CSEG SEGMENT PARA PUBLIC 'CODIGO' ASSUME CS:CSEG, DS:DSEG, SS:STACK
AEsoft_Prg PROC FAR
;**** Comienzo del Procedimiento PRINCIPAL
;A continuación se actualiza el Registro DS con el valor adecuado. MOV AX,DSEG ;Mediante este par de instrucciones hacemos accesibles MOV DS,AX ;nuestros datos.
;--¿ ; ³ ; Ã-- Aquí se incluye el código del procedimiento principal. ; ³ ;--Ù
```

```
MOV AH,4CH ;Función de Terminación de Programa. MOV AL,00 ;Ejecución del programa exitosa. Para indicar error, dar ;a AL un valor distinto de 00h. INT 21H ;Ejecuto la función (Salgo del programa actual).

;**** Fin del Procedimiento Principal

;************************** Inicio de los Procedimientos
;Aquí se sitúan cada uno de los procedimientos de que consta el programa.
Proc_1 PROC
;--¿ ; ³ ; Ã- Código del Procedimiento Proc_1 ; ³ ;--Ù
RET
Proc_1 ENDP
;***
Proc_2 PROC
;--¿ ; ³ ; Ã- Código del Procedimiento Proc_2 ; ³ ;--Ù
RET
Proc_2 ENDP

;***
Proc_n PROC
;--¿ ; ³ ; Ã- Código del Procedimiento Proc_n ; ³ ;--Ù
RET
Proc_n ENDP

;************************** Fin de los procedimientos
AEsoft_prg ENDP
;***** Fin del Procedimiento principal.

CSEG ENDS
;***** Fin de Segmento de Código
END AEsoft_prg
;---------------------------- Fin del Programa --------------------

- EJEMPLOS DE PROGRAMAS ----------------------
* EJEMPLO DE PROGRAMA COM *
;---------------------------- inicio del programa ----------------------
PAGE 60,132
TITLE Bloque_Entorno ;CopyRight (C) 1995. Francisco Jesus Riquelme. (AeSoft)

CSEG SEGMENT PARA PUBLIC 'CODIGO' ASSUME CS:CSEG, DS:CSEG, SS:CSEG ORG 100H
AEsoft_Prog:
JMP AEsoft_Code ;Salto al código del Programa.
;****** INICIO DE LOS DATOS
FILE_HANDLE DW 0 ;Handle del fichero que voy a usar para ;almacenar el Bloque. FILE_NAME db 'FileBlk.inf',0 ;Cadena ASCIIZ con el Nombre del Fichero ;que almacenar el Bloque.
MENSAJE_DE_ERROR DB 'Se ha producido un error de Fichero. Programa ' DB 'Abortado.$'
;MENSAJE_DE_ERROR contiene el mensaje ;que se mostrar por pantalla si se ;produce un error de fichero.
```

;NOTAS: ; - El mensaje puede ocupar varias líneas. ; - El mensaje acaba cuando el DOS lee el ; car cter $, el cual no es enviado a ; la pantalla.

;****** FIN DE LOS DATOS

;****** INICIO DEL PROGRAMA
AEsoft_Code:
;Primero creo el fichero que va a contener el Bloque de Entorno. ;Al crearlo, queda abierto para las siguientes operaciones.
MOV AH,3CH MOV CX,00H ;Atributo de Fichero Normal. MOV DX,Offset FILE_NAME ;DS:DX apuntando al nombre del Fichero. INT 21H ;Ejecuto la función. ;A continuación compruebo si se ha producido ;error en la ejecución de la función.
JC ERROR_FILE ;Si a la vuelta de la ejecución de la INT 21H el flag Cf ;(Flag de Carry o Acarreo) tiene valor 1, esto quiere ;decir que se ha producido error. Por tanto salto a ;la rutina que trata dicho error, que simplemente dar ;un mensaje de Error al usuario y acto seguido finaliza ;el programa.
;Si el control del programa llega hasta aquí, es porque no hay error. ;Entonces en AX se devuelve el Handle con el que manejaremos al ;fichero recientemente creado.
MOV FILE_HANDLE,AX ;Almaceno el Handle que asigna el DOS a mi fichero.
;Ya tengo el fichero abierto, listo para almacenar el Bloque ;de Entorno.
;Ahora lo que voy a hacer es 'situarme' en el inicio del Bloque y ;copiarlo al fichero.
MOV AX,WORD PTR CS:[2CH] ;En AX, dirección del Bloque de Entorno.
;A continuación preparo los registros adecuados para ;invocar a la función 40h de la Int 21h (Escritura en ;Fichero). MOV DS,AX ;Registro de Segmento DS apuntando al principio del Bloque. MOV DX,0 ;El Bloque empieza en el Offset 00h.
;Ya tengo los registros DS:DX apuntando al inicio del Bloque. ;Ahora lo que debo saber es el tamaño de dicho Bloque. ;Hemos dicho que el nombre del programa aparece despues del valor ;0001h, y que el nombre del programa aparece como una cadena ASCIIZ. ;Esto quiere decir que el último byte ser un 00h. ;O sea que primero buscamos un byte 01h, saltamos el byte siguiente ;que sería el byte alto de la palabra 0001h, y a continuación buscamos ;un byte con valor 00h que nos indica el final del Bloque de Entorno. ;Usamos el registro CX (Contador) para llevar la cuenta del total de ;bytes de que est compuesto el Bloque de Entorno.
MOV CX,0 ;Inicializo el registro contador. MOV SI,DX ;Resguardo el contenido del registro DX (que voy a necesitar ;luego) utilizando el registro SI con el valor que tenía DX.
CALL LONGITUD_BLOQUE ;Llamo al procedimiento LONGITUD_BLOQUE para ;obtener en CX la longitud del Bloque de Entorno.
;Tras la ejecución del procedimiento LONGITUD_BLOQUE, ;ya tengo en CX el total de bytes que componen el Bloque de Entorno. ;Tengo también el par de registros DS:DX apuntando al inicio del ;Bloque. MOV AH,40H ;Número de la función (Escribir en fichero). MOV BX,CS:FILE_HANDLE ;(Handle de fichero). ;Debo indicar el segmento donde se encuentra ;la variable FILE_HANDLE, ya que el Registro DS ;no apunta a los datos, sino al Bloque de ;Entorno. INT 21H ;Ejecuto la función.
;Ya tengo copiado a fichero el Bloque de Entorno.
;Ahora cierro el fichero. Esto es muy importante. ;Todo fichero abierto debe ser cerrado antes de salir del programa.
MOV AH,3EH ;Número de la función (Cerrar fichero). MOV BX,CS:FILE_HANDLE ;(Handle de fichero). INT 21H ;Ejecuto la función.
JC ERROR_FILE ;Si se ha producido error al intentar cerrar el fichero, ;mostrar mensaje y salir del programa.

;Si llega hasta aquí el control del programa es porque no se ha ;producido ningún error. ;A continuación salgo del programa con código de retorno 0, indicando ;que no se ha producido error.
MOV AH,4CH ;Función de Terminación de Programa. MOV AL,00 ;Ejecución del programa exitosa. INT 21H ;Ejecuto la función (Salgo del programa actual).

ERROR_FILE:
;Si el control del programa llega hasta aquí, es porque se ha producido ;un error al trabajar con el fichero. A continuación muestro por ;pantalla un mensaje al usuario comunic ndolo, y finalizo el programa.
MOV AH,9 MOV DX,OFFSET CS:MENSAJE_DE_ERROR INT 21H ;Mostrado el mensaje de Error por la pantalla.
MOV AH,4CH ;Función de Terminación de Programa. MOV AL,01 ;Código de Retorno 01 (distinto de 0) que indica Error ;en la ejecución del programa. INT 21H ;Ejecuto la función (Salgo del programa actual).

;****** FIN DEL PROGRAMA

;****** INICIO DE LOS PROCEDIMIENTOS
LONGITUD_BLOQUE PROC
Bucle_Busca_01h:
CMP BYTE PTR [SI],01h JZ Encontrado_01h
INC SI ;Incremento el contenido del Registro Indice. INC CX ;Incremento el número de bytes totales del Bloque.
JMP SHORT Bucle_Busca_01h ;Salto corto (SHORT) hacia el inicio del ;bucle en busca del siguiente byte para ;comparar.
Encontrado_01h: ;Al llegar aquí, ya tenemos el byte 01h que indica que ;a continuación encontraremos el nombre del programa. ;Pero antes de ese nombre de Programa est el byte alto ;de la palabra 0001h. Es decir, tenemos que saltar un ;byte 00h antes de llegar al nombre del programa. ;Recordad la curiosa forma que tiene el procesador de ;almacenar las palabras en la memoria: ;El byte bajo (de menor peso), al principio. ;El byte alto, a continuación. ;Así, la palabra 0001h se almacena en memoria como 0100h. ;Si accedemos a este valor a nivel de palabra, no hay ;problema, ya que usaremos un registro de tipo palabra para ;almacenar el valor, y no nos enteraremos de esta peculiaridad. ;Pero si accedemos en modo byte, nos encontramos con que el ;primer byte ser el que por lógica debería ser el segundo, ;y viceversa. Por tanto, en el programa que nos toca, vamos ;a encontrar primero el byte 01h, y luego simplemente saltamos ;el siguiente byte, ya que sabemos que va a ser el byte con ;valor 00h.
ADD SI,2 ADD CX,2 ;Mediante las dos instrucciones de arriba he saltado el byte ;01h que acabo de encontrar al salir del bucle, y he saltado ;también el byte 00h (byte alto de la palabra 0001h). ;Ambos bytes los contabilizo como bytes del Bloque de Entorno ;mediante el incremento del registro CX.
;A continuación busco el byte 00h que cierra el nombre de programa ;y por tanto el Bloque de Entorno.

Bucle_busca_00h:
CMP BYTE PTR [SI],00h JZ Encontrado_00h
INC SI ;Incremento el contenido del Registro Indice. INC CX ;Incremento el número de bytes totales del Bloque.

JMP SHORT Bucle_Busca_00h ;Salto corto (SHORT) hacia el inicio del ;bucle en busca del siguiente byte para ;comparar.

Encontrado_00h:
INC CX ;Para añadir a la cuenta el byte 00h que cierra el Bloque ;de Entorno.
RET
LONGITUD_BLOQUE ENDP

;****** FIN DE LOS PROCEDIMIENTOS

CSEG ENDS END AEsoft_Prog

;----------------------------- fin del programa ------------------------

* EJEMPLO DE PROGRAMA EXE *

;----------------------------- Inicio del Programa -------------------
PAGE 80,132 TITLE Bloque_Entorno ;CopyRight (C) 1995. Francisco Jesus Riquelme. (AeSoft)
;***** Inicio de Segmento de Pila
STACK SEGMENT PARA STACK 'PILA' DW 64 DUP (0)
STACK ENDS
;***** Fin de Segmento de Pila

;***** Inicio de Segmento de Datos
DSEG SEGMENT PARA PUBLIC 'DATOS'
FILE_HANDLE DW 0 ;Handle del fichero que voy a usar para ;almacenar el Bloque.
FILE_NAME db 'FileBlk.inf',0 ;Cadena ASCIIZ con el Nombre del Fichero ;que almacenar el Bloque.
MENSAJE_DE_ERROR DB 'Se ha producido un error de Fichero. Programa ' DB 'Abortado.$'
;MENSAJE_DE_ERROR contiene el mensaje ;que se mostrar por pantalla si se ;produce un error de fichero.
;NOTAS: ; - El mensaje puede ocupar varias líneas. ; - El mensaje acaba cuando el DOS lee el ; car cter $, el cual no es enviado a ; la pantalla.

DSEG ENDS
;***** Fin de Segmento de Datos

;***** Inicio de Segmento de Código
CSEG SEGMENT PARA PUBLIC 'CODIGO' ASSUME CS:CSEG, DS:DSEG, SS:STACK
AEsoft_Prg PROC FAR
;Comienzo del Programa PRINCIPAL.
;Actualiza el Registro DS (Segmento de Datos) con el valor adecuado. MOV AX,DSEG MOV DS,AX
;Primero creo el fichero que va a contener el Bloque de Entorno. ;Al crearlo, queda abierto para las siguientes operaciones.
MOV AH,3CH MOV CX,00H ;Atributo de Fichero Normal. MOV DX,Offset FILE_NAME ;DS:DX apuntando al nombre del Fichero. INT 21H ;Ejecuto la función. ;A continuación compruebo si se ha producido ;error en la ejecución de la función.

JC ERROR_FILE ;Si a la vuelta de la ejecución de la INT 21H el flag Cf ;(Flag de Carry o Acarreo) tiene valor 1, esto quiere ;decir que se ha producido error. Por tanto salto a ;la rutina que trata dicho error, que simplemente dar ;un mensaje de Error al usuario y acto seguido finaliza ;el programa.
;Si el control del programa llega hasta aquí, es porque no hay error. ;Entonces en AX se devuelve el Handle con el que manejaremos al ;fichero recientemente creado.
MOV FILE_HANDLE,AX ;Almaceno el Handle que asigna el DOS a mi fichero.
;Ya tengo el fichero abierto, listo para almacenar el Bloque ;de Entorno.
;Ahora lo que voy a hacer es 'situarme' en el inicio del Bloque y ;copiarlo al fichero.
MOV AX,WORD PTR ES:[2CH] ;En AX, dirección del Bloque de Entorno. ;ES apunta desde el principio al PSP, como ;no hemos modificado su valor, ahora nos ;valemos de el.
;DS también apuntaba al PSP, pero unas líneas ;mas arriba hemos modificado su valor para ;que apunte a nuestro segmento de datos.
;A continuación preparo los registros adecuados para ;invocar a la función 40h de la Int 21h (Escritura en ;Fichero).
PUSH DS POP ES ;Resguardo la dirección del segmento de datos en el registro ;ES, ya que DS va a perder su valor original. ;De esta forma no perder, la dirección de mi segmento de datos.
MOV DS,AX ;Registro de Segmento DS apuntando al principio del Bloque. MOV DX,0 ;El Bloque empieza en el Offset 00h.
;Ya tengo los registros DS:DX apuntando al inicio del Bloque. ;Ahora lo que debo saber es el tamaño de dicho Bloque. ;Hemos dicho que el nombre del programa aparece despues del valor ;0001h, y que el nombre del programa aparece como una cadena ASCIIZ. ;Esto quiere decir que el último byte ser un 00h. ;O sea que primero buscamos un byte 01h, saltamos el byte siguiente ;que sería el byte alto de la palabra 0001h, y a continuación buscamos ;un byte con valor 00h que nos indica el final del Bloque de Entorno. ;Usamos el registro CX (Contador) para llevar la cuenta del total de ;bytes de que est compuesto el Bloque de Entorno.
MOV CX,0 ;Inicializo el registro contador. MOV SI,DX ;Resguardo el contenido del registro DX (que voy a necesitar ;luego) utilizando el registro SI con el valor que tenía DX.
CALL LONGITUD_BLOQUE ;Llamo al procedimiento LONGITUD_BLOQUE para ;obtener en CX la longitud del Bloque de Entorno.
;Tras la ejecución del procedimiento LONGITUD_BLOQUE, ;ya tengo en CX el total de bytes que componen el Bloque de Entorno. ;Tengo también el par de registros DS:DX apuntando al inicio del ;Bloque. MOV AH,40H ;Número de la función (Escribir en fichero). MOV BX,ES:FILE_HANDLE ;(Handle de fichero). ;Debo indicar el segmento donde se encuentra ;la variable FILE_HANDLE, ya que el Registro DS ;no apunta a los datos, sino al Bloque de ;Entorno. INT 21H ;Ejecuto la función.
;Ya tengo copiado a fichero el Bloque de Entorno.
;Ahora cierro el fichero. Esto es muy importante. ;Todo fichero abierto debe ser cerrado antes de salir del programa.
MOV AH,3EH ;Número de la función (Cerrar fichero). MOV BX,ES:FILE_HANDLE ;(Handle de fichero). INT 21H ;Ejecuto la función.
JC ERROR_FILE ;Si se ha producido error al intentar cerrar el fichero, ;mostrar mensaje y salir del programa.
;Si llega hasta aquí el control del programa es porque no se ha ;producido ningún error. ;A continuación salgo del programa con código de retorno 0, indicando ;que no se ha producido error.
MOV AH,4CH ;Función de Terminación de Programa. MOV AL,00 ;Ejecución del programa exitosa. INT 21H ;Ejecuto la función (Salgo del programa actual).

```
ERROR_FILE:
;Si el control del programa llega hasta aquí, es porque se ha producido ;un error al trabajar con el fichero. A continuación muestro por ;pantalla un mensaje al usuario comunic ndolo, y finalizo el programa.
MOV AH,9 MOV DX,OFFSET ES:MENSAJE_DE_ERROR INT 21H ;Mostrado el mensaje de Error por la pantalla.

MOV AH,4CH MOV AL,1 ;Código de Retorno que indica Error en la ejecución del Programa. INT 21H ;Finaliza el programa y vuelve al proceso padre con código de ;Error.
;**** FIN DEL PROGRAMA PRINCIPAL

;************************** Procedimientos:
LONGITUD_BLOQUE PROC
Bucle_Busca_01h:
CMP BYTE PTR [SI],01h JZ Encontrado_01h
INC SI ;Incremento el contenido del Registro Indice. INC CX ;Incremento el número de bytes totales del Bloque.
JMP SHORT Bucle_Busca_01h ;Salto corto (SHORT) hacia el inicio del ;bucle en busca del siguiente byte para ;comparar.
Encontrado_01h: ;Al llegar aquí, ya tenemos el byte 01h que indica que ;a continuación encontraremos el nombre del programa. ;Pero antes de ese nombre de Programa est el byte alto ;de la palabra 0001h. Es decir, tenemos que saltar un ;byte 00h antes de llegar al nombre del programa. ;Recordad la curiosa forma que tiene el procesador de ;almacenar las palabras en la memoria: ;El byte bajo (de menor peso), al principio. ;El byte alto, a continuación. ;Así, la palabra 0001h se almacena en memoria como 0100h. ;Si accedemos a este valor a nivel de palabra, no hay ;problema, ya que usaremos un registro de tipo palabra para ;almacenar el valor, y no nos enteraremos de esta peculiaridad. ;Pero si accedemos en modo byte, nos encontramos con que el ;primer byte ser el que por lógica debería ser el segundo, ;y viceversa. Por tanto, en el programa que nos toca, vamos ;a encontrar primero el byte 01h, y luego simplemente saltamos ;el siguiente byte, ya que sabemos que va a ser el byte con ;valor 00h.
ADD SI,2 ADD CX,2 ;Mediante las dos instrucciones de arriba he saltado el byte ;01h que acabo de encontrar al salir del bucle, y he saltado ;también el byte 00h (byte alto de la palabra 0001h). ;Ambos bytes los contabilizo como bytes del Bloque de Entorno ;mediante el incremento del registro CX.
;A continuación busco el byte 00h que cierra el nombre de programa ;y por tanto el Bloque de Entorno.

Bucle_busca_00h:
CMP BYTE PTR [SI],00h JZ Encontrado_00h
INC SI ;Incremento el contenido del Registro Indice. INC CX ;Incremento el número de bytes totales del Bloque.
JMP SHORT Bucle_Busca_00h ;Salto corto (SHORT) hacia el inicio del ;bucle en busca del siguiente byte para ;comparar.

Encontrado_00h:
INC CX ;Para añadir a la cuenta el byte 00h que cierra el Bloque ;de Entorno.
RET
LONGITUD_BLOQUE ENDP

;************************** Fin de los procedimientos
```

```
AEsoft_prg ENDP
CSEG ENDS END AEsoft_prg
;***** Fin de Segmento de Código

;---------------------------- Fin del Programa -------------------
```

- CREAR EL PROGRAMA EJECUTABLE (ENSAMBLAR-LINKAR) ---------- En este apartado vamos a ver cómo convertir nuestro código fuente en ejecutable.

Según hayamos creado el código fuente podremos obtener dos modelos diferentes de ejecutable: COM y EXE. Si hemos seguido ciertas reglas necesarias para poder conseguir un programa COM, podremos obtener los dos formatos mediante el mismo código fuente. Si no hemos seguido esas reglas, sólo podremos obtener un programa EXE.

A la hora de ensamblar-linkar necesitaremos de un paquete Ensamblador, con su Programa Ensamblador, su Linkador, etc. Los mas potentes según mi criterio son MASM (de MicroSoft) y TASM (de Borland). Aparte de estos dos paquetes, en el mercado existen varios mas, algunos son shareware.

En caso de utilizar un Ensamblador diferente a MASM y TASM, ,chale un vistazo a la documentación que acompaña al programa, ya que hay ciertos Ensambladores muy peculiares. Los hay, por ejemplo, que obtienen el programa COMasin necesidad de crear OBJ ni EXE intermedios.

Lo mencionado a continuación es v lido para MASM y TASM.

* CREAR EXE * Para obtener un programa EXE a partir de un código fuente (ASM) deberemos seguir los siguientes pasos: Supongamos que nuestro código fuente tiene por nombre PROG.ASM

1. Ensamblar el código fuente.

MASM PROG.ASM; (TASM en caso de utilizar Turbo Assembler)

El programa MASM es el 'Ensamblador'. Mediante este paso conseguimos el fichero OBJ (Objeto). Este fichero OBJ aún no est listo para ser ejecutado, ya que en el pueden existir referencias a datos, procedimientos, etc.. que se dejan en blanco para ser completadas por el programa LINK, ya que a MASM no se le d toda la información necesaria para poder completar estas referencias.

La sintaxis completa a emplear con el programa MASMase indica a continuación:

```
-------------------------------------------------------------------------------
Usage: masm /options source(.asm),[out(.obj)],[list(.lst)],[cref(.crf)][;]
/a Alphabetize segments
/b<number> Set I/O buffer size, 1-63 (in 1K blocks)
/c Generate cross-reference
/d Generate pass 1 listing
/D<sym>[=<val>] Define symbol
/e Emulate floating point instructions and IEEE format
/I<path> Search directory for include files
/l[a] Generate listing, a-list all
/M{lxu} Preserve case of labels: l-All, x-Globals, u-Uppercase Globals
/n Suppress symbol tables in listing
/p Check for pure code
/s Order segments sequentially
/t Suppress messages for successful assembly
/v Display extra source statistics
/w{012} Set warning level: 0-None, 1-Serious, 2-Advisory
```

/X List false conditionals
/z Display source line for each error message
/Zi Generate symbolic information for CodeView
/Zd Generate line-number information

Esta pantalla de ayuda se consigue mediante la orden MASM /H

2. Linkar (fusionar) los ficheros Objeto (OBJ). Convertir el fichero OBJ (Objeto) en EXE (Ejecutable).
Mediante este paso convertimos nuestro fichero OBJ en un fichero ejecutable (EXE). LINK completa las direcciones que MASM dejó pendientes en el módulo Objeto (Fichero OBJ), asign ndoles su dirección real. También fusiona ('linka') varios módulos OBJ en un mismo programa final (EXE) si así se había requerido. Por último, crea la cabecera del programa EXE necesaria para la posterior carga y ejecución de los distintos segmentos del programa por parte del DOS.
LINK PROG.OBJ; (TLINK en caso de utilizar Turbo Assembler)
Mediante estos dos pasos ya tenemos creado nuestro programa EXE a partir del código fuente escrito en ASM.
Los par metros (options) v lidos a emplear con el programa LINK se indican a continuación:

Microsoft (R) Overlay Linker Version 3.61
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.
Valid options are:
/BATCH /CODEVIEW
/CPARMAXALLOC /DOSSEG
/DSALLOCATE /EXEPACK
/FARCALLTRANSLATION /HELP
/HIGH /INFORMATION
/LINENUMBERS /MAP
/NODEFAULTLIBRARYSEARCH /NOEXTDICTIONARY
/NOFARCALLTRANSLATION /NOGROUPASSOCIATION
/NOIGNORECASE /NOPACKCODE
/OVERLAYINTERRUPT /PACKCODE
/PAUSE /QUICKLIBRARY
/SEGMENTS /STACK

Esta pantalla de ayuda se consigue mediante la orden LINK /HELP

* CREAR COM * Para poder obtener un programa COM a partir de un código fuente (ASM) debemos proceder como sigue:
1. Obtener el programa EXE mediante los dos pasos del apartado anterior.
2. Convertir el programa EXE en COM. Para llevar a cabo esta conversión existe la utilidad EXE2BIN, que como su nombre indica (hay que echarle imaginación :-) convierte los EXE a (*) BIN. *El 2 ese lo utilizan para abreviar la palabra TO, la cual se pronuncia igual que TWO (2).
EXE2BIN PROG.EXE
Mediante la línea anterior obtenemos el fichero BIN. Este fichero BIN debemos convertirlo en fichero COM, por fin. Para hacer esta tarea existe una orden de sistema operativo

llamada REN. :-) El contenido del fichero BIN est listo ya para ser ejecutado como si de un COMase tratara, en realidad su contenido es una COpia_de_Memoria (COM). Pero como el DOS no ejecuta BIN, tendremos que cambiarle su extensión a COM (que sí ejecuta).
REN PROG.BIN PROG.COM
Ya tenemos el programa listo para ser ejecutado como COM.
Nota: En lugar de utilizar la orden REN PROG.BIN PROG.COM es preferible la orden COPY PROG.BIN PROG.COM, ya que en cuanto hagamos la primera revisión o mejora del programa, la orden REN PROG.BIN PROG.COM no funcionar al existir ya el PROG.COM.
Nota2: Hay utilidades EXE2BIN que no devuelven un fichero BIN, sino un fichero COM. En este caso, un trabajo que nos ahorramos de cambiarle el nombre.
Nota3: Para poder obtener un programa COM, el código fuente debe cumplir los siguientes requisitos: - Incluir la pseudo_instrucción (ORG 100H) como ya se indicó en el apartado de programas COM. - No hacer ninguna referencia a segmentos definidos en el programa. Es decir, un programa COM no puede tener instrucciones como la siguiente (MOV AX,CSEG).
(Ver modelo y ejemplo de programa COM).
Nota4: Al linkar un fichero OBJ preparado para funcionar como COM, el Linkador nos dar un mensaje de advertencia (Warning) indic ndonos que falta por definir el segmento de pila. Eso lo hace porque no sabe si vamos a convertir el programa EXE resultante a formato COM.
No debemos hacer caso a este mensaje.

Bueno, pues eso es todo por esta lección.
saluDOS. Francisco Jesus Riquelme.