INTRODUCCIÓN

El documento que está visualizando tiene la función primordial de introducirlo a la programación en lenguaje Ensamblador. Su contenido se enfoca completamente hacia las computadoras que operan con procesadores de la familia x86 de Intel y, considerando que el ensamblador basa su funcionamiento en los recursos internos del procesador, los ejemplos descriptos no son compatibles con ninguna otra arquitectura.

GENERALIDADES

Unidades de información

Para que la PC pueda procesar la información es necesario que ésta se encuentre en celdas especiales llamadas registros.
Los registros son conjuntos de 8 o 16 flip-flops (basculadores o biestables).
Un flip-flop es un dispositivo capaz de almacenar dos niveles de voltaje, uno bajo, regularmente, alrededor, de 0.5 volts y otro alto comúnmente de 5 volts. El nivel bajo de energía en el flip-flop se interpreta como apagado o 0, y el nivel alto como prendido o 1. A estos estados se les conoce usualmente como bits, que son la unidad más pequeña de información en una computadora.
A un grupo de 16 bits se le conoce como palabra, una palabra puede ser dividida en grupos de 8 bits llamados bytes, y a los grupos de 4 bits les llamamos nibbles.

Sistemas numéricos

El sistema numérico que utilizamos a diario es el sistema decimal, pero este sistema no es conveniente para las máquinas debido a que la información se maneja codificada en forma de bits prendidos o apagados; esta forma de codificación nos lleva a la necesidad de conocer el cálculo posicional que nos permita expresar un número en cualquier base que lo necesitemos.

Proceso de creación de un programa

Para la creación de un programa es necesario seguir cinco pasos:

- Diseño del algoritmo.
- Codificación del mismo.
- Su traducción a lenguaje máquina.
- La prueba del programa.
- La depuración del programa.

En la etapa de diseño se plantea el problema a resolver y se propone la mejor solución, creando diagramas esquemáticos utilizados para el mejor planteamiento de la solución.
La codificación del programa consiste en escribir el programa en algún lenguaje de programación; en este caso específico en ensamblador, tomando como base la solución propuesta en el paso anterior.
La traducción al lenguaje máquina es la creación del programa objeto, esto es, el programa escrito como una secuencia de ceros y unos que pueda ser interpretado por el procesador.
La prueba del programa consiste en verificar que el programa funcione sin errores, o sea, que haga lo que tiene que hacer.
La última etapa es la eliminación de las fallas detectadas en el programa durante la fase de prueba. La corrección de una falla normalmente requiere la repetición de los pasos comenzando desde el primero o el segundo.
Para crear un programa en ensamblador existen dos opciones, la primera es utilizar algún software específico, por ejemplo el MASM (Macro Assembler, de Microsoft), y la segunda es utilizar el debugger del DOS; en esta primera sección se utilizará este último ya que se encuentra en cualquier PC con el sistema operativo MS-DOS, lo cual lo pone al alcance de cualquier usuario que tenga acceso a una máquina con estas características.
Debug solo puede crear archivos con extensión .COM, y por las características de este tipo de programas no pueden ser mayores de 64 Kb, además deben comenzar en el desplazamiento, offset, o dirección de memoria 0100H dentro del segmento específico.

Componentes Básicos de un Sistema MS-DOS

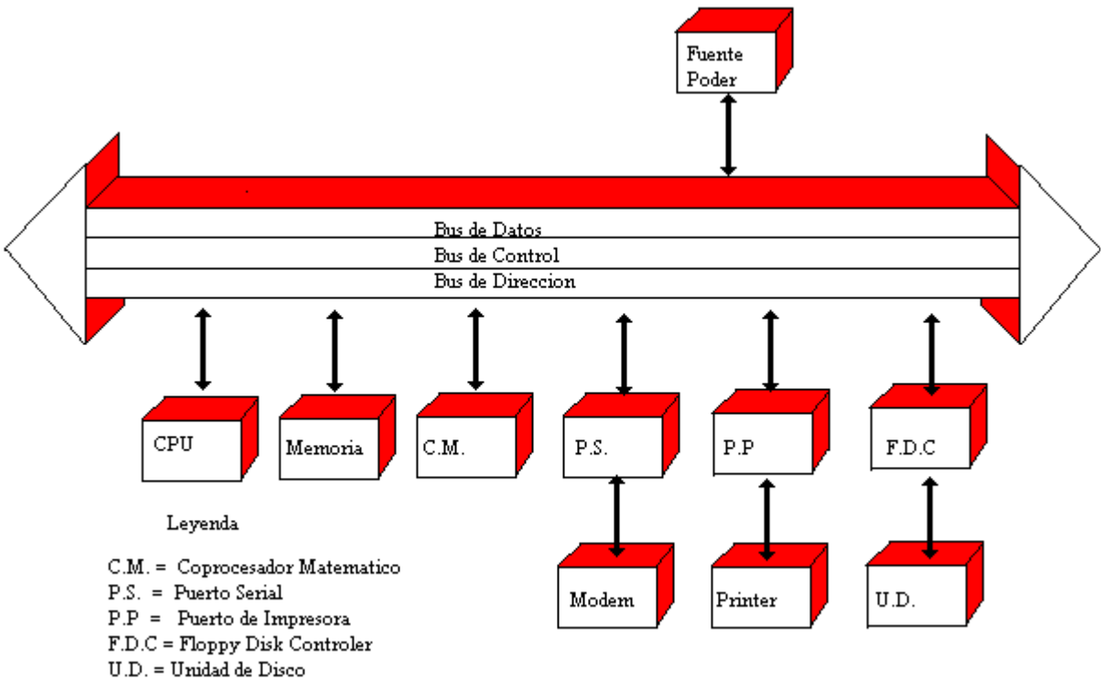


Fig. 1: Componentes de un sistema MS-DOS.

Las operaciones de un sistema de computación incluyendo un IBM PC's y compatibles están basadas en un concepto simple. Ellas guardan instrucciones y datos en la memoria y usan la CPU para repetir instrucciones y datos recibidos desde la memoria y ejecutan las instrucciones para manipular los datos (Computadoras basadas en la Arquitectura de Von Newmann), por lo tanto la CPU y la memoria son los dos componentes básicos de cualquier sistema de computación. La memoria se encuentra definida en dos tipos: Random Access Memory (**RAM)** la que permite la escritura y la lectura de cualquier localidad de memoria y la Read Only Memory (**ROM)**, que es la que contiene valores que pueden ser leídos pero no alterados. La ROM es usada para almacenar pequeños primitivos programas para ejecutar instrucciones de entrada y salida y control de periféricos. La RAM es usada por el Sistema Operativo y los programas para los usuarios. El Sistema Operativo es un componente fundamental en un sistema. Este programa de computadoras se toma la tarea de cargar otros programas y ejecutarlos, provee acceso a los archivos del sistema, maneja la E/S, y hace interfaces interactivas con el usuario. El sistema operativo es el que provee al sistema su personalidad. MS-DOS, OS/2, UNIX son ejemplo de algunos Sistema Operativos para PC, similarmente CP/M es un Sistema Operativos para antiguos microprocesadores de INTEL de 8 Bits como el 8080. El hardware de toda computadora, incluyendo las computadoras que usan el MS-DOS, está interconectado.
La CPU, memoria, y periféricos de entrada (teclado, escáner, lápiz óptico, lector de código de barras, micrófono, mouse, etc.) y periféricos de salida (monitor, impresora, parlantes, etc.) están todos interconectados por una serie de cables llamados Buses y cada bus esta claramente definido. Un Bus es un hardware que especifica una señal y tiempo estándar que son seguidos y entendidos por la CPU y su circuito de soporte (incluyendo periféricos aún no instalados). Los buses a su vez se clasifican en Bus de Datos, Bus de Direcciones y Bus de Control.

ARQUITECTURA INTERNA DEL INTEL 80x86

Fue el primer microprocesador de 16 bits que INTEL fabricó a principios del año 1978. Los objetivos de la arquitectura de dicho procesador fueron los de ampliar la capacidad del INTEL 80x80 de forma simétrica, añadiendo una potencia de proceso no disponible en los micros de 8 bits. Algunas de estas características son: aritmética en 16 bits, multiplicación y división con o sin signo, manipulación de cadena de caracteres y operación sobre bits. También se han realizado mecanismos de software para la construcción de códigos reentrante y reubicable. Su estructura interna está representada por la Fig. 2.
Consta de 2 unidades claramente diferenciadas denominadas EU (Unidad de Ejecución) y BIU (Interfaces del Bus).
La EU ejecuta las operaciones requeridas por las instrucciones sobre una UAL de 16 bits. No tiene conexión con el exterior y solamente se comunica con la BIU que es la parte que realiza todas las operaciones en el bus solicitadas por la EU. Un mecanismo, tal vez único dentro de

los microprocesadores aunque muy empleado dentro de los mínimos y grandes computadores, es el denominado de búsqueda anticipada de instrucciones (prefetch). En el INTEL 8086 existe una estructura FIFO en RAM de 6 octetos de capacidad que es llenada por la BIU con los contenidos de las instrucciones siguientes a la que la EU está ejecutando en ese momento.

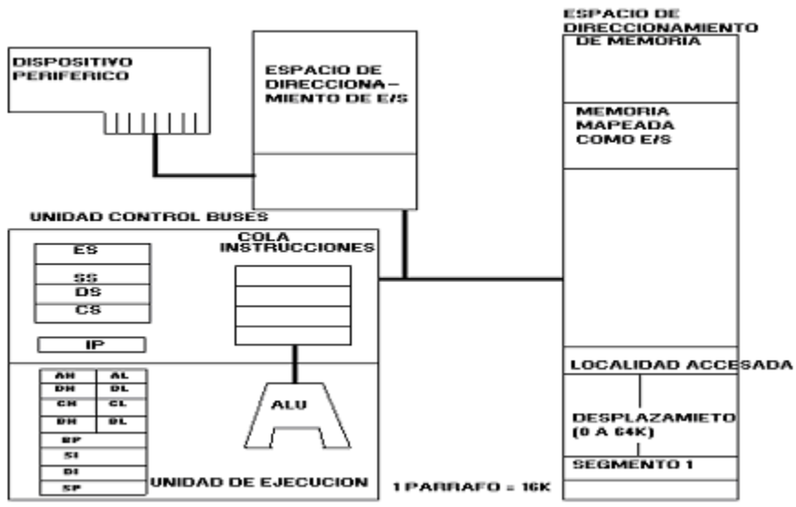


Fig. 2: Las unidades EU y BIU

El 8086 representa la arquitectura base para todos los microprocesadores de 16 bits de Intel: 8088, 8086, 80188, 80186 y 80286. Aunque han aparecido nuevas características a medida que estos microprocesadores han ido evolucionando; todos los procesadores Intel, usados en la actualidad en los PC's y compatibles son miembros de la familia 8086. El conjunto de instrucciones, registros y otras características son similares, a excepción de algunos detalles. Toda la familia 80x86 en adelante posee dos características, en común:

a) **Arquitectura Segmentada:** esto significa que la memoria es divida en segmentos con un tamaño máximo de 64k (información importante para el direccionamiento de la memoria en la futura programación segmentada en el lenguaje ensamblador).
b) **Compatibilidad:** las instrucciones y registros de las anteriores versiones son soportados por las nuevas versiones, y estas versiones son soportadas por versiones anteriores.

La familia de microprocesadores 80x86 consta de los siguientes microprocesadores:

- El ***8088*** es un microprocesador de 16 bits, usado en las primeras PC´S (XT compatibles). Soporta solamente el modo real. Es capaz de direccionar un megabyte de memoria y posee un bus de datos de 8 bits.
- El ***8086*** es Similar al 8088, con la excepción de que el bus de datos es de 16 bits.
- El ***80188*** es similar al 8088, pero con un conjunto de instrucciones extendido y ciertas mejoras en la velocidad de ejecución. Se incorporan dentro del microprocesador algunos chips que anteriormente eran externos, consiguiéndose unas mejoras en el rendimiento del mismo.
- El ***80186*** es igual al 80188 pero con un bus de datos de 16 bits.
- El ***80286*** Incluye un conjunto de instrucciones extendidos del 80186, pero además soporta memoria virtual, modo protegido y multitarea.
- El ***80386*** soporta procesamientos de 16 y 32 bits. El 80386 es capaz de manejar memoria real y protegida, memoria virtual y multitarea. Es más rápido que el 80286 y contiene un conjunto de instrucciones ampliables.
- El ***80386SX*** es similar al 80386 por un bus de datos de solo 16 bits.
- El ***80486*** incorpora un caché interno de 8 kb y ciertas mejoras de velocidad con respecto al 80386. Incluye un coprocesador matemático dentro del mismo chip.
- El ***80486SX*** es similar a los 80486 con la diferencia que no posee coprocesador matemático.
- El ***80486DX2*** es similar al 80486, pero con la diferencia de que internamente, trabaja al doble de la frecuencia externa del reloj.

Modelo	Bus de Datos	Coprocesador matemático
80386DX	32	Si
80386SL	16	No
80386SX	16	No
80486SX	32	No
80486DX	32	Si

El 80x86 tiene dos procesadores en el mismo chip. Estos son la Unidad de Ejecución y la Unidad de Interface de Bus, como ya se mencionó antes. Cada uno de ellos contiene su propio registro, su propia sección aritmética, sus propias unidades de control y trabajan de manera asincrónica el uno con el otro para proveer la potencia total de cómputo. La Unidad de Interface de Bus se encarga de buscar las instrucciones para adelantar su ejecución y proporciona facilidades en el manejo de las direcciones. Luego, la Unidad de Interface se responsabiliza del control de la adaptación con los elementos externos del CPU central. Dicha Unidad de Interface proporciona una dirección de 20 Bits o un dato de 16 para la unidad de memoria o para la unidad de E/S en la estructura externa de la computadora.

Terminales del microprocesador 8086

El microprocesador 8086 puede trabajar en dos modos diferentes: el modo mínimo y el modo máximo. En el modo máximo el microprocesador puede trabajar en forma conjunta con un microprocesador de datos numérico 8087 y algunos otros circuitos periféricos. En el modo mínimo el microprocesador trabaja de forma más autónoma al no depender de circuitos auxiliares pero esto, a su vez, le resta flexibilidad.
En cualquiera de los dos modos, las terminales del microprocesador se pueden agrupar de la siguiente forma:

- Alimentación.
- Reloj.
- Control y estado.
- Direcciones.
- Datos.

El 8086 cuenta con tres terminales de alimentación: tierra (GND) en las terminales 1 y 20 y Vcc = 5V en la terminal 40.
En la terminal 19 se conecta la señal de reloj, la cual debe provenir de un generador de reloj externo al microprocesador.
El 8086 cuenta con 20 líneas de direcciones (al igual que el 8088). Estas líneas son llamadas A_0 a A_{19} y proporcionan un rango de direccionamiento de 1MB.
Para los datos, el 8086 comparte las 16 líneas más bajas de sus líneas de direcciones, las cuales son llamadas AD_0 a AD_{15}. Esto se logra gracias a un canal de datos y direcciones multiplexado.
En cuanto a las señales de control y estado tenemos las siguientes:

- La terminal MX/MN controla el cambio de modo del microprocesador.
- Las señales S_0 a S_7 son señales de estado, éstas indican diferentes situaciones acerca del estado del microprocesador.
- La señal RD en la terminal 32 indica una operación de lectura.
- En la terminal 22 se encuentra la señal READY. Esta señal es utilizada por los diferentes dispositivos de E/S para indicarle al microprocesador si se encuentran listos para una transferencia.
- La señal RESET en la terminal 21 es utilizada para reinicializar el microprocesador.
- La señal NMI en la terminal 17 es una señal de interrupción no enmascarable, lo cual significa que no puede ser manipulada por medio de software.
- La señal INTR en la terminal 18 es también una señal de interrupción, la diferencia radica en que esta señal si puede ser controlada por software. Las interrupciones se estudian más adelante.
- La terminal TEST se utiliza para sincronizar al 8086 con otros microprocesadores en una configuración en paralelo.

- Las terminales RQ/GT y LOCK se utilizan para controlar el trabajo en paralelo de dos o más microprocesadores.
- La señal WR es utilizada por el microprocesador cuando éste requiere realizar alguna operación de escritura con la memoria o los dispositivos de E/S.
- Las señales HOLD y HLDA son utilizadas para controlar el acceso al bus del sistema.

Unidad aritmético-lógica

Conocida también como ALU, este componente del microprocesador es el que realmente realiza las operaciones aritméticas (suma, resta, multiplicación y división) y lógicas (and, or, xor, etc.) que se obtienen como instrucciones de los programas.

Buses internos (datos y direcciones)

Los buses internos son un conjunto de líneas paralelas (conductores) que interconectan las diferentes partes del microprocesador.
Existen dos tipos principales: el bus de datos y el bus de direcciones. El bus de datos es el encargado de transportar los datos entre las distintas partes del microprocesador; por otro lado, el bus de direcciones se encarga de transportar las direcciones para que los datos puedan ser introducidos o extraídos de la memoria o dispositivos de entrada y salida.

Cola de instrucciones

La cola de instrucciones es una pila de tipo FIFO (primero en entrar, primero en salir) donde las instrucciones son almacenadas antes de que la unidad de ejecución las ejecute.

TIPOS DE PROGRAMAS EJECUTABLES

ESTRUCTURA DEL PROGRAMA CON EXTENSIÓN .COM

Un programa con extensión .COM está almacenado en un archivo que contiene una copia fiel del código a ser ejecutado. Ya que no contienen información para la reasignación de localidades, son más compactos y son cargados más rápidamente que sus equivalentes EXE.
El MS-DOS no tiene manera de saber si un archivo con extensión .COM es un programa ejecutable válido. Este simplemente lo carga en memoria y le transfiere el control. Debido al hecho de que los programas COM son siempre cargados inmediatamente después del PSP y no contienen encabezado que especifique el punto de entrada al mismo, siempre debe comenzar en la dirección 0100h. Esta dirección deberá contener la primera instrucción ejecutable. La longitud máxima de un programa COM es de 65536 bytes, menos la longitud de PSP (256 bytes) y la longitud de la pila (mínimo 2 bytes).
Cuando el sistema operativo transfiere el control a un programa COM, todos los registros de segmento apuntan al PSP. El registro apuntador de pila (SP) contiene el valor en la memoria de OFFFEh si la memoria lo permite. En otro caso adopta el mínimo valor posible menos dos bytes (el MS-DOS introduce un cero en la pila antes de transferir el control al programa). Aún cuando la longitud de un programa COM no puede exceder de los 64 kb; las versiones actuales del MS-DOS reservan toda la memoria disponible. Si un programa .COM debe ejecutar otro proceso, es necesario que el mismo libere la memoria no usada de tal manera que pueda ser empleada por otra aplicación. Cuando un programa .COM termina; puede retornar al control del sistema operativo por varios medios. El método preferido es el uso de la función **4Ch** de la **Int 21h**, la cual permite que el programa devuelva un código de retorno al proceso que invocó. Sin embargo, si el programa está ejecutándose bajo la versión 1.00 del MS.DOS, el control debe ser retornado mediante el uso de la **Int 20h**. Un programa .COM puede ser ensamblado a partir de varios módulos objeto, con la condición de todos ellos empleen los mismos nombres y clases de segmentos y asegurando que el módulo inicial, con el punto de entrada en 0100h sea enlazado primero. Adicionalmente todos los procedimientos y funciones deben tener el atributo NEAR, ya que todo el código ejecutable estará dentro del mismo segmento.
Al enlazar un programa .COM, el enlazador mostrará el siguiente mensaje; "***Warning: no stack segment***". Este mensaje puede ser ignorado, ya que el mismo se debe a que se ha instruido al enlazador para que genere un programa con extensión .EXE donde el segmento de pila debe ser indicado de manera explícita, y no así en los .COM donde ésta es asumida por defecto. En la zona desde 000Ah hasta 0015h dentro del PSP se encuentran las

direcciones de las rutinas manejadoras de los eventos Ctrl-C y Error crítico. Si el programa de aplicación altera estos valores para sus propios propósitos, el MS-DOS los restaura al finalizar la ejecución del mismo.

ESTRUCTURA DEL PREFIJO DE PROGRAMA (PSP)

0000h	***INT 20 para el regreso a DOS.***
0002h	***Segmento final del bloque de asignación.***
0004h	***Reservado.***
0005h	***Invocación FAR a la función despachadora del MS-DOS.***
000Ah	***Vector de interrupción de terminación (Int 22h).***
000Eh	***Vector de interrupción Ctrl-C (Int 23h).***
0012h	***Vector de interrupción de error crítico (Int 24h).***
0016h	***Reservado.***
002Ch	***Segmento de bloque de variables de ambiente.***
002Eh	***Reservado.***
005Ch	***Bloque de control de archivo por defecto (#1).***
006Ch	***Bloque de control de archivo por defecto (#2).***
0080h	***Líneas de comandos y área de transferencia de disco.***
00FFh	***Final del PSP.***

La palabra de datos en desplazamiento 002Ch contiene la dirección del segmento de bloque de variables de ambiente (Environment block), el cual contiene una serie de cadenas ASCII. Este bloque es heredado del proceso que causó la ejecución del programa de aplicación. Entre la información que contiene tenemos el paso usado por el COMAND.COM para encontrar el archivo ejecutable, el lugar del disco donde se encuentra el propio COMAND.COM y el formato del prompt empleado por éste. La cola de comandos, la cual está constituida por los caracteres restantes en la línea de comandos, después del nombre del programa, es copiada a partir de la localidad 0081h en el PSP. La longitud de la cola, sin incluir el carácter de retorno al final, está ubicada en la posición 0080h. Los parámetros relacionados con redireccionamiento o piping no aparecen en esta posición de la línea de comandos, ya que estos procesos son transparentes a los programas de aplicación. Para proporcionar compatibilidad con CP/M, el MS-DOS coloca los dos primeros comandos en la cola, dentro de los bloques de control del archivo (FCB) por defecto en las direcciones PSP:005Ch y PSP:006Ch asumiendo que pueden ser nombres de archivos. Sin embargo, si alguno de estos comandos son nombres de archivos que incluyen especificaciones del paso, la información colocada en los FCB no será de utilidad ya que estas estructuras no soportan el manejo de estructuras jerárquicas de archivos y subdirectorios. Los FCB son de muy escaso uso en los programas de aplicación modernos. El área de 128 bytes ubicado entre las direcciones 0080h y 00FFh en el PSP puede también servir como área de transferencia de disco por defecto (DTA), la cual es establecida por el MS-DOS antes de transferir el control al programa de aplicación. A menos que el programa establezca de manera explícita otra DTA, este será usado como buffer de datos para cualquier intercambio con disco que este efectué. Los programas de aplicación no deben alterar la información contenida en el PSP a partir de la dirección 005Ch.

ESTRUCTURA DE UN PROGRAMA DE EXTENSIÓN .EXE

Los programas .EXE son ilimitados en tamaño (el límite lo dictamina la memoria disponible del equipo). Además, los programas .EXE pueden colocar el código, los datos y la pila en distintos segmentos de la memoria. La oportunidad de colocar las diversas partes de un programa en fragmentos diferentes de memoria y la de establecer segmentos de memoria con solamente códigos que pudieran ser compartidos por varias tareas, es muy significativo para ambientes multitareas tales como el Microsoft Windows. El cargador del MS-DOS sitúa al programa .EXE, inmediatamente después del PSP, aunque el orden de los segmentos que

lo constituyen pueden variar. El archivo .EXE contiene un encabezado, bloque de información de control, con un formato característico. El tamaño de dicho encabezado puede variar dependiendo del número de instrucciones que deben ser localizadas al momento de carga del programa, pero siempre será múltiplo de 512. Antes de que el MS-DOS transfiera el control al programa, se calculan los valores iniciales del registro del segmento de código (CS) y el apuntador de instrucciones (IP) basados en la información sobre el punto de entrada al programa, contenida en el encabezado del archivo .EXE. Esta información es generada a partir de la instrucción END en el módulo principal del programa fuente. Los registros de segmentos de datos y segmentos extras son inicializados de manera que apunten al PSP de modo que el programa pueda tener acceso a la información contenida.

IMAGEN DE MEMORIA DE UN PROGRAMA .EXE TÍPICO

SS:SP	***Segmento de Pila***
SS:0000h	***Datos del Programa***
CS:0000h	***Código del Programa***
DS:0000h	***Prefijo del Segmento del Programa***
ES: 0000h	

FORMATO DE UN ARCHIVO DE CARGA EXE

0000h	***Primera parte del identificador del archivo EXE (4Dh)***
0001h	***Segunda parte del identificador de archivo EXE (5Ah)***
0002h	***Longitud del archivo MOD 512***
0004h	***Tamaño del archivo, en páginas de 512 bytes, incluyendo encabezado***
0008h	***Número de ítems en la tabla de relocalizaciones***
000Ah	***Tamaño del encabezado en párrafos (16 bytes)***
000Ch	***Número mínimo de párrafos requeridos para el programa***
000Eh	***Máximo número de párrafos deseables para el programa***
0010h	***Desplazamiento del segmento del módulo de pila***
0012h	***Suma de chequeo***
0016h	***Contenido del apuntador de instrucciones al comenzar el programa***
0018h	***Desplazamiento del segmento del módulo de código***
001Ah	***Desplazamiento del primer ítem en la tabla de relocalizaciones***
001Bh	***Número de overplay (0 para la parte residente del programa)***
	Tabla de relocalizaciones
	Espacio reservado (longitud variable)
	Segmento de programas y datos
	Segmento de pila

El contenido inicial del segmento de pila y del apuntador de pila proviene también del encabezado del archivo. Esta información es derivada de la declaración del segmento de pila efectuada mediante la sentencia STACK. El espacio reservado para la pila puede ser inicializado o no dependiendo de la manera como este haya sido declarado. Puede ser conveniente en muchos casos inicializar el segmento de pila con un patrón de caracteres predeterminados que permitan su posterior inspección. Cuando el programa .EXE finaliza su ejecución debe retornar el control al sistema operativo mediante la función 4Ch de la Int 21h. Existen otros métodos, pero no ofrecen ninguna otra ventaja y son considerablemente menos convenientes "Generalmente requieren que el registro CS apunte al segmento de PSP".

Un programa .EXE puede ser construido a partir de varios módulos independientes. Cada módulo puede tener nombres diferentes para el segmento de código y los procedimientos pueden llevar el atributo NEAR o FAR, dependiendo del tamaño del programa ejecutable. El programador debe asegurarse de que los módulos a ser enlazados solo tengan una declaración de segmento de pila y que haya sido definido un único punto de entrada (por medio de la directiva END). La salida del enlazador es un archivo con extensión .EXE el cual puede ser ejecutado inmediatamente.

REGISTROS DE LA CPU

Los registros del procesador se emplean para controlar instrucciones en ejecución, manejar direccionamiento de memoria y proporcionar capacidad aritmética. Los registros son espacios físicos dentro del microprocesador con capacidad de 4 bits hasta 64 bits dependiendo del microprocesador que se emplee. Los registros son direccionables por medio de una viñeta, que es una dirección de memoria. Los bits, por conveniencia, se numeran de derecha a izquierda (15, 14, 13,…., 3, 2, 1, 0), los registros están divididos en seis grupos los cuales tienen un fin especifico. Los registros se dividen en:

- Registros de segmento.
- Registro apuntador de instrucciones.
- Registros apuntadores.
- Registros de propósitos generales.
- Registros índices.
- Registro de banderas.

REGISTROS DE SEGMENTO

Un registro de segmento se utiliza para alinear en un limite de párrafo o dicho de otra forma codifica la dirección de inicio de cada segmento y su dirección en un registro de segmento supone cuatro bits 0 a su derecha.
Un registro de segmento tiene 16 bits de longitud y facilita un área de memoria para direccionamientos conocidos como el segmento actual. Los registros de segmento son:

- Registro CS.
- Registro DS.
- Registro SS.
- Registro ES.
- Registro FS y GS.

Registro CS

El DOS almacena la dirección inicial del segmento de código de un programa en el registro CS. Esta dirección de segmento, más un valor de desplazamiento en el registro apuntador de instrucciones (IP), indica la dirección de una instrucción que es buscada para su ejecución. Para propósitos de programación normal, no se necesita referenciar el registro CS.

Registro DS

La dirección inicial de un segmento de datos de un programa es almacenada en el registro DS. En términos sencillos, esta dirección más un valor de desplazamiento en una instrucción, genera una referencia a la localidad de un byte especifico en el segmento de datos.

Registro SS

El registro SS permite la colocación en memoria de una pila, para almacenamiento temporal de direcciones y datos. El DOS almacena la dirección de inicio del segmento de pila de un programa en el registro SS. Esta dirección de segmento, más un valor de desplazamiento en el registro apuntador de la pila (SP), indica la palabra actual en la pila que está siendo direccionada. Para propósitos de programación normal, no se necesita referenciar el registro SS.

Registro ES

Algunas operaciones con cadenas de caracteres (datos de caracteres) utilizan el registro de segmento extra para manejar el direccionamiento de memoria. En este contexto, el registro ES esta asociado con el registro DI (índice). Un programa que requiere el uso del registro ES puede inicializarlo con una dirección apropiada.

Registros FS y GS
Son registros extra de segmento en los procesadores 80386 y posteriores a estos procesadores.

Registro Apuntador de instrucciones (IP)
El registro apuntador de instrucciones (IP) de 16 bits contiene el desplazamiento de dirección de la siguiente instrucción que se ejecuta.
El registro IP esta asociado con el registro CS en el sentido de que el IP indica la instrucción actual dentro del segmento de código que se está ejecutando actualmente.
En el ejemplo siguiente, el registro CS contiene 25A4[0]H y el IP contiene 412H. Para encontrar la siguiente instrucción que será ejecutada el procesador combina las direcciones en el CS y el IP así:

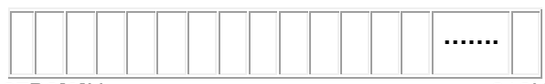
Segmento de dirección en el registro CS: 25A40H
Desplazamiento de dirección en el registro IP: + 412H
Dirección de la siguiente instrucción: 25E52H

Registros apuntadores
Los registros apuntadores están asociados con el registro SS y permiten al procesador accesar datos en el segmento de pila; los registros apuntadores son dos:

- El registro SP.
- El registro BP.

Registro SP
El apuntador de pila SP de 16 bits está asociado con el registro SS y proporciona un valor de desplazamiento que se refiere a la palabra actual que está siendo procesada en la pila. El ejemplo siguiente el registro SS contiene la dirección de segmento 27B3[0]H y el SP el desplazamiento 312H Para encontrar la palabra actual que esta siendo procesada en la pila el microprocesador combina las direcciones en el SS y el SP:

Dirección de segmento en el registro SS: 27B30H
Desplazamiento en el registro SP: + 312H
Dirección en la Pila: 27E42H



27B3[0]H 312H
Dirección del segmento SS Desplazamiento del SP

Registro BP
El registro BP de 16 bits facilita la referencia de parámetros, los cuales son datos y direcciones transmitidos vía la pila.

Registros de propósitos generales
Los registros de propósitos generales AX, BX, CX y DX son los caballos de batalla o las herramientas del sistema. Son únicos en el sentido de que se puede direccionarlos como una palabra o como una parte de un byte. El byte de la izquierda es la parte "alta", y el byte de la derecha es la parte "baja". Por ejemplo, el registro CX consta de una parte CH (alta) y una parte CL (baja), y usted puede referirse a cualquier parte por su nombre. Las instrucciones siguientes mueven ceros a los registros CX, CH y CL respectivamente:

```
Mov CX, 00
Mov CH, 00
Mov CL, 00
```

Los procesadores 80386 y posteriores permiten el uso de todos estos registros de propósito general, más las versiones de 32 bits: EAX, EBX, ECX y EDX.

Registros AX
El registro AX, el acumulador principal, es utilizado para operaciones que implican entrada/salida y la mayor parte de la aritmética. Por ejemplo, las instrucciones para

multiplicar, dividir y traducir suponen el uso del AX. También, algunas operaciones generan código más eficientes si se refiere al AX en lugar de los otros registros**.**

Registro BX
El BX es conocido como el registro base ya que es el único registro de propósito general que puede ser índice para el direccionamiento indexado. También es común emplear al BX para cálculos.

Registro CX
El CX es conocido como el registro contador. Puede contener un valor para controlar el número de veces que un ciclo se repite o un valor para corrimiento de bits, hacia la derecha o hacia la izquierda. El CX también es usado para muchos cálculos.

Registro DX
El DX es conocido como el registro de datos. Algunas operaciones de entrada/salida requieren su uso, y las operaciones de multiplicación y división con cifras grandes suponen al DX y al AX trabajando juntos.
Puede usar los registros de propósito general para suma y resta de cifras de 8, 16 o 32 bits.

Registros índices
Los registros SI y DI están disponibles para direccionamientos indexados y para sumas y restas.

Registro SI
El registro índice fuente SI de 16 bits es requerido por algunas operaciones con cadenas (de caracteres). En este contexto, el SI está asociado con el registro DS. Los procesadores 80386 y posteriores permiten el uso de un registro ampliado a 32 bits: el ESI.

Registro DI
El registro índice destino DI también es requerido por algunas operaciones con cadenas de caracteres. En este contexto, el DI está asociado con el registro ES. Los procesadores 80386 y posteriores permiten el uso de un registro ampliado a 32 bits: el EDI.

Registro de banderas
Los registros de banderas sirven parar indicar el estado actual de la máquina y el resultado del procesamiento. Cuando algunas instrucciones piden comparaciones o cálculos aritméticos cambian el estado de las banderas.
Las banderas están en el registro de banderas en las siguientes posiciones:

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

Banderas
Las banderas más comunes son las siguientes:

OF (Overflow flag, desbordamiento)
Indica el desbordamiento de un bit de orden alto (más a la izquierda) después de una operación aritmética.

DF (Direction flag, Dirección)
Designa la dirección hacia la izquierda o hacia la derecha para mover o comparar cadenas de caracteres.

IF (Interruption flag, Interrupción)
Indica que una interrupción externa, como la entrada desde el teclado sea procesada o ignorada.

TF (Trap flag, Trampa)
Examina el efecto de una instrucción sobre los registros y la memoria. Los programas depuradores como DEBUG, activan esta bandera de manera que pueda avanzar en la ejecución de una sola interrupción a un tiempo.

SF (Sign flag, Signo)

Contiene el signo resultante de una operación aritmética (0 = positivo y 1 = negativo).

ZF (Zero flag, Cero)
Indica el resultado de una operación aritmética o de comparación (0 = resultado diferente de cero y 1 = resultado igual a cero)

AF (Auxiliary carry flag, Acarreo auxiliar)
Contiene un acarreo externo del bit 3 en un dato de 8 bits, para aritmética especializada

PF (Parity flag, Paridad)
Indica paridad par o impar de una operación en datos de ocho bits de bajo orden (más a la derecha)

CF (Carry flag, Acarreo)
Contiene el acarreo de orden más alto (más a la izquierda) después de una operación aritmética; también lleva el contenido del ultimo bit en una operación de corrimiento o rotación.

CUADRO COMPARATIVO

TIPOS DE REGISTROS	FUNCIÓN
Registros de Segmento	Un registro de segmento tiene 16 bits de longitud y facilita un área de memoria para el direccionamiento conocida como el segmento actual
Registros de Apuntador de Instrucciones	Este registro esta compuesto por 16 bits y contiene el desplazamiento de la siguiente instrucción que se va a ejecutar. Los procesadores 80386 y posteriores tienen un IP ampliado de 32 bits llamado EIP.
Registros Apuntadores	Permiten al sistema accesar datos al segmento de la pila. Los procesadores 80386 tienen un apuntador de pila de 32 bits llamado ESP. El sistema maneja de manera automática estos registros.
Registros de Propósito General	Son los caballos de batalla del sistema y pueden ser direccionados como una palabra o como una parte de un bytes. Los procesadores 80386 y posteriores permiten el uso de todos los registros de propósitos general más sus versiones ampliadas de 32 bits llamados EAX, EBX, ECX y EDX.
Registros Índices	Sirven para el direccionamiento de indexado y para las operaciones de sumas y restas.
Registros de Banderas	Sirven para indicar el estado actual de la máquina y el resultado del procesamiento. De los 16 bits de registro de bandera 9 son comunes a toda la familia de los procesadores 8086.

Ejemplo de Representación de los Registros

Después de haber conceptualizado e interpretado los diferente tipos de registro es necesario plantear un ejemplo no muy practico pero si muy significativo, en el cual se representa la forma estructurada de un programa en el lenguaje ensamblador y como se utilizan los diferentes términos investigados, se verá que en el programa o en una pequeña porción de él se muestra como se colocan dentro, los diferentes tipos registros.

```
TITLE P17HANRD(EXE) Lectura secuencial de registros.
.MODEL SMALL
.STACK 64
/-------------------------------------------------------------------------------------
.DATA
ENDCDE DB 00              ;FIN DEL INDICADOR DE PROCESO.
HANDLE DW ?
IOAREA DB 32 DUP(´ ´)
OPENMSG DB ´*** Open error ***´ 0DH, 0AH
PATHNAM DB ´D:\NAMEFILE.SRT´, 0
READMSD DB ´*** Read error ***´ 0DH, 0AH
ROW DB 00
/-------------------------------------------------------------------------------------
.CODE
BEGIN PROC FAR
MOV AX,@data              ;inicializa
MOV DS,AX                 ;registro de
MOV ES,AX          ;segmento
MOV AX,0600H
```

Es posible visualizar los valores de los registros internos de la UCP utilizando el programa Debug.

MODOS DE DIRECCIONAMIENTO

Se les llama modos de direccionamiento a las distintas formas de combinar los operadores según el acceso que se hace a memoria.
Dicho de otra manera, un modo de direccionamiento será una forma de parámetro para las instrucciones. Una instrucción que lleve un parámetro, por lo tanto, usará un modo de direccionamiento, que dependerá de cómo direccionará (accesará) al parámetro; una instrucción de dos parámetros, combinará dos modos de direccionamiento.

Estos pueden clasificarse en 5 grupos:

1. Direccionamientos accesando dato inmediato y registro de datos (modos inmediato y de registro)
2. Direccionamiento accesando datos en memoria (modo memoria)
3. Direccionamiento accesando puertos E/S. (modo E/S)
4. Direccionamiento relativo
5. Direccionamiento implícito.

1. DIRECCIONAMIENTO ACCESANDO DATO Y REGISTRO INMEDIATO

1.1 Direccionamiento de registro.

Especifica el operando fuente y el operando destino. Los registros deben ser del mismo tamaño.
Ej. MOV DX, CX
MOV CL, DL.

1.2 Direccionamiento inmediato.

Un dato de 8 o 16 bits se especifica como parte de la instrucción. p.ej. MOV CL, 03H. Aquí el operando fuente está en modo inmediato y el destino en modo registro.

2. DIRECCIONAMIENTO ACCESANDO DATOS EN MEMORIA

2.1 Direccionamiento directo.

La dirección efectiva (EA) de 16 bits se toma directamente del campo de desplazamiento de la instrucción. El desplazamiento se coloca en la localidad siguiente al código de operación. Esta EA o desplazamiento es la distancia de la localidad de memoria al valor actual en el segmento de datos (DS) en el cual el dato está colocado.
Ej. MOV CX, START.
START puede definirse como una localidad de memoria usando las pseudoinstrucciones DB o DW.

2.2 Direccionamiento de registro indirecto.

La dirección efectiva EA está especificada en un registro apuntador o un registro índice. El apuntador puede ser el registro base BX o el apuntador base BP; el registro índice puede ser el Índice Fuente (SI) o el Índice Destino (DI).

Ej. MOV (DI),BX

2.3 Direccionamiento base (Relativo a la base).

La EA se obtiene sumando un desplazamiento (8 bits con signo o 16 bits sin signo) a los contenidos de BX o BP. Los segmentos usados son DS y SS. Cuando la memoria es accesada, la dirección física de 20 bits es calculada de BX y DS, por otra parte, cuando la pila es accesada, la dirección es calculada de BP y SS.
Ej. MOV AL, START (BX), el operando fuente está en modo base, y la EA se obtiene sumando los valores de START y BX.

2.4 Direccionamiento indexado (Indexado Directo).

EA se calcula sumando un desplazamiento (8 o 16 bits) a los contenidos de SI o DI.
Ej. MOV BH,START (SI).

2.5 Direccionamiento base indexado.

EA se calcula sumando un registro base (BX o BP), un registro índice (DI o SI), y un desplazamiento (8 o 16 bits).
Ej. MOV ALPHA (SI)(BX),CL.
Este direccionamiento proporciona una forma conveniente para direccionar un arreglo localizado en la pila.

2.6 Direccionamiento (cadena)

Este modo usa registros índice. La cadena de instrucciones automáticamente asume que SI apunta al primer byte o palabra del operando destino. Los contenidos de SI y DI son incrementados automáticamente (poniendo a 0 DF mediante la instrucción CLD) o decrementados (poniendo a 1 DF mediante la instrucción STD) para apuntar al siguiente byte o palabra. El segmento del operando fuente es DS y puede ser encimado.
El segmento del operando destino debe ser ES y no puede ser encimado. Ej. MOVS BYTE.

3. DIRECCIONAMIENTO ACCESANDO PUERTOS (E/S)

Hay dos tipos de direccionamiento usando puertos: directo e indirecto.
En el modo directo, el número de puerto es el operando inmediato de 8 bits, lo cual permite accesar puertos numerados del 0 al 255.
Ej. OUT 05H,AL.
En el modo indirecto, el número de puerto se toma de DX, permitiendo así 64K puertos de 8 bits o 32K puertos de 16 bits.
Las transferencias E/S de 8 y 16 bits deben hacerse vía AL y AX, respectivamente.

4. DIRECCIONAMIENTO RELATIVO.

En este modo el operando se especifica como un desplazamiento de 8 bits con signo, relativo al PC.
Ej. JNC START. Si C=0, entonces el PC se carga con PC+el valor de START.

5. DIRECCIONAMIENTO IMPLICITO.

Las instrucciones que usan este modo no tienen operandos.
Ej. CLC.

TRABAJANDO CON EL DEBUG

Para empezar a trabajar con Debug digite en el prompt de la computadora:

C:\> Debug [Enter]

En la siguiente línea aparecerá un guión, éste es el indicador del Debug, en este momento se pueden introducir las instrucciones del Debug. Utilizando el comando:

- r [Enter]

Se desplegaran todos los contenidos de los registros internos de la UCP; una forma alternativa de mostrarlos es usar el comando "r" utilizando como parámetro el nombre del registro cuyo valor se quiera visualizar. Por ejemplo:

- rbx [Enter]

Esta instrucción desplegará únicamente el contenido del registro BX y cambia el indicador del Debug de " - " a " : "

Estando así el prompt es posible cambiar el valor del registro que se visualiza tecleando el nuevo valor y a continuación [Enter], o se puede dejar el valor anterior presionando [Enter] sin teclear ningún valor.

Estructura de una instrucción en ensamblador

En el lenguaje ensamblador las líneas de código están constituidas de dos partes: la primera es el nombre de la instrucción que se va a ejecutar y la segunda son los parámetros del comando u operandos. Por ejemplo:

add ah, bh

Aquí "add" es el comando a ejecutar (en este caso una adición) y tanto "ah" como "bh" son los parámetros.

El nombre de las instrucciones en este lenguaje esta formado por dos, tres o cuatro letras. A estas instrucciones también se les llama nombres mnemónicos o códigos de operación, ya que representan alguna función que habrá de realizar el procesador.

Existen algunos comandos que no requieren parámetros para su operación, así como otros que requieren solo un parámetro.

Algunas veces se utilizarán las instrucciones como sigue:

add al, [170]

Los corchetes en el segundo parámetro nos indican que vamos a trabajar con el contenido de la casilla de memoria número 170 y no con el valor 170, a esto se le conoce como direccionamiento directo.

Primer programa

Se creará un programa que sirva para ilustrar lo analizado. Lo que se hará es una suma de dos valores que introduciremos directamente en el programa.

El primer paso es iniciar el Debug, este paso consiste únicamente en teclear debug [Enter] en el prompt del sistema operativo.

Para ensamblar un programa en el Debug se utiliza el comando "a" (Assembler); cuando se utiliza este comando se le puede dar como parámetro la dirección donde se desea que se inicie el ensamblado.

Si se omite el parámetro el ensamblado se iniciará en la localidad especificada por CS: IP, usualmente 0100H, que es la localidad donde deben iniciar los programas con extensión .COM, y será la localidad que utilizaremos debido a que Debug solo puede crear este tipo específico de programas.

Aunque en este momento no es necesario darle un parámetro al comando "a" es recomendable hacerlo para evitar problemas una vez que se haga uso de los registros CS: IP, por lo tanto tecleamos:

- a0100 [Enter]

Al hacer esto aparecerá en la pantalla algo como: 0C1B:0100 y el cursor se posiciona a la derecha de estos números. Nótese que los primeros cuatro dígitos (en sistema hexadecimal) pueden ser diferentes, pero los últimos cuatro deben ser 0100, ya que es la dirección que indicamos como inicio.
Ahora podemos introducir las instrucciones:

```
0C1B:0100 mov ax,0002   ;coloca el valor 0002 en el registro ax
0C1B:0103 mov bx,0004   ;coloca el valor 0004 en el registro bx
0C1B:0106 add ax,bx     ;le adiciona al contenido de ax el contenido de bx
0C1B:0108 int 20        ;provoca la terminación del programa.
0C1B:010A
```

No es necesario escribir los comentarios que van después del ";". Una vez digitado el último comando, int 20, se le da [Enter] sin escribir nada más, para volver al prompt del debugger.
La última línea escrita no es propiamente una instrucción de ensamblador, es una llamada a una interrupción del sistema operativo, estas interrupciones serán tratadas más a fondo posteriormente, por el momento solo es necesario saber que nos ahorran un gran número de líneas y son muy útiles para accesar a funciones del sistema operativo.
Para ejecutar el programa que escribimos se utiliza el comando "g", al utilizarlo veremos que aparece un mensaje que dice: "Program terminated normally". Naturalmente con un mensaje como éste no podemos estar seguros que el programa haya hecho la suma, pero existe una forma sencilla de verificarlo, utilizando el comando "r" del Debug podemos ver los contenidos de todos los registros del procesador, simplemente teclee:

- r [Enter]

Aparecerá en pantalla cada registro con su respectivo valor actual:

```
AX=0006 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0C1B ES=0C1B SS=0C1B CS=0C1B IP=010A NV UP EI PL NZ NA PO NC
0C1B:010A 0F DB oF
```

Existe la posibilidad de que los registros contengan valores diferentes, pero AX y BX deben ser los mismos, ya que son los que acabamos de modificar.
Otra forma de ver los valores, mientras se ejecuta el programa es utilizando como parámetro para "g" la dirección donde queremos que termine la ejecución y muestre los valores de los registros, en este caso sería: g108, esta instrucción ejecuta el programa, se detiene en la dirección 108 y muestra los contenidos de los registros.
También se puede llevar un seguimiento de lo que pasa en los registros utilizando el comando "t" (trace), la función de este comando es ejecutar línea por línea lo que se ensambla mostrando cada vez los contenidos de los registros.
Para salir del Debug se utiliza el comando "q" (quit).

Guardar y cargar los programas

No sería práctico tener que digitar todo un programa cada vez que se necesite, para evitar eso es posible guardar un programa en el disco, con la enorme ventaja de que ya ensamblado no será necesario correr de nuevo Debug para ejecutarlo.
Los pasos a seguir para guardar un programa ya almacenado en la memoria son:

- Obtener la longitud del programa restando la dirección final de la dirección inicial, naturalmente en sistema hexadecimal.
- Darle un nombre al programa y extensión.
- Poner la longitud del programa en el registro CX.
- Ordenar a Debug que escriba el programa en el disco.

Utilizando como ejemplo el programa anterior se tendrá una idea más clara de como llevar estos pasos.

Al terminar de ensamblar el programa se vería así:

```
0C1B:0100 mov ax,0002
0C1B:0103 mov bx,0004
0C1B:0106 add ax,bx
0C1B:0108 int 20
0C1B:010A
- h 10a 100
020a 000a
- n prueba.com
- rcx
CX 0000
:000a
-w
Writing 000A bytes
```

Para obtener la longitud de un programa se utiliza el comando "h", el cual muestra la suma y resta de dos números en hexadecimal. Para obtener la longitud de nuestro programa le proporcionamos como parámetros el valor de la dirección final de nuestro programa (10A) y el valor de la dirección inicial (100). El primer resultado que nos muestra el comando es la suma de los parámetros y el segundo es la resta.
El comando "n" nos permite poner un nombre al programa.
El comando "rcx" nos permite cambiar el contenido del registro CX al valor que obtuvimos del tamaño del archivo con "h", en este caso 000a, ya que nos interesa el resultado de la resta de la dirección inicial a la dirección final.
Por último el comando “w” escribe nuestro programa en el disco, indicándonos cuantos bytes escribió.
Para cargar un archivo ya guardado son necesarios dos pasos:

- Proporcionar el nombre del archivo que se cargará.
- Cargarlo utilizando el comando "l" (load).

Para obtener el resultado correcto de los siguientes pasos es necesario que previamente se haya creado el programa anterior.
Dentro del Debug escribimos lo siguiente:

```
- n prueba.com
- l
- u 100 109
0C3D:0100 B80200 MOV AX,0002
0C3D:0103 BB0400 MOV BX,0004
0C3D:0106 01D8 ADD AX,BX
0C3D:0108 CD20 INT 20
```

El último comando, "u", se utiliza para verificar que el programa se cargó en memoria, lo que hace es desensamblar el código y mostrarlo ya desensamblado. Los parámetros le indican a Debug desde donde y hasta donde desensamblar.
Debug siempre carga los programas en memoria en la dirección 100H, a menos que se le indique alguna otra dirección.

Condiciones, ciclos y bifurcaciones

Estas estructuras, o formas de control le dan a la máquina un cierto grado de decisión basado en la información que recibe.
La forma más sencilla de comprender este tema es por medio de ejemplos.
Se crearán tres programas que hagan lo mismo: desplegar un número determinado de veces una cadena de caracteres en la pantalla.

```
- a100
0C1B:0100 jmp 125 ; brinca a la dirección 125H
0C1B:0102 [Enter]
- e 102 'Cadena a visualizar 15 veces' 0d 0a '$'
- a125
0C1B:0125 MOV CX,000F ; veces que se desplegará la cadena
```

```
0C1B:0128 MOV DX,0102 ; copia cadena al registro DX
0C1B:012B MOV AH,09 ; copia valor 09 al registro AH
0C1B:012D INT 21 ; despliega cadena
0C1B:012F LOOP 012D ; si CX>0 brinca a 012D
0C1B:0131 INT 20 ; termina el programa.
```

Por medio del comando "e" es posible introducir una cadena de caracteres en una determinada localidad de memoria, dada como parámetro, la cadena se introduce entre comillas, le sigue un espacio, luego el valor hexadecimal del retorno de carro, un espacio, el valor de línea nueva y por último el símbolo '$' que el ensamblador interpreta como final de la cadena. La interrupción 21 utiliza el valor almacenado en el registro AH para ejecutar una determinada función, en este caso mostrar la cadena en pantalla, la cadena que muestra es la que está almacenada en el registro DX. La instrucción LOOP decrementa automáticamente el registro CX en uno y si no ha llegado el valor de este registro a cero brinca a la casilla indicada como parámetro, lo cual crea un ciclo que se repite el número de veces especificado por el valor de CX. La interrupción 20 termina la ejecución del programa.

Otra forma de realizar la misma función pero sin utilizar el comando LOOP es la siguiente:

```
- a100
0C1B:0100 jmp 125 ; brinca a la dirección 125H
0C1B:0102 [Enter]
- e 102 'Cadena a visualizar 15 veces' 0d 0a '$'
- a125
0C1B:0125 MOV BX,000F ; veces que se desplegará la cadena
0C1B:0128 MOV DX,0102 ; copia cadena al registro DX
0C1B:012B MOV AH,09 ; copia valor 09 al registro AH
0C1B:012D INT 21 ; despliega cadena
0C1B:012F DEC BX ; decrementa en 1 a BX
0C1B:0130 JNZ 012D ; si BX es diferente a 0 brinca a 012D
0C1B:0132 INT 20 ; termina el programa.
```

En este caso se utiliza el registro BX como contador para el programa, y por medio de la instrucción "DEC" se disminuye su valor en 1. La instrucción "JNZ" verifica si el valor de B es diferente a 0, esto con base en la bandera NZ, en caso afirmativo brinca hacia la dirección 012D. En caso contrario continúa la ejecución normal del programa y por lo tanto se termina.

Una última variante del programa es utilizando de nuevo a CX como contador, pero en lugar de utilizar LOOP utilizaremos decrementos a CX y comparación de CX a 0.

```
- a100
0C1B:0100 jmp 125 ; brinca a la dirección 125H
0C1B:0102 [Enter]
- e 102 'Cadena a visualizar 15 veces' 0d 0a '$'
- a125
0C1B:0125 MOV DX,0102 ; copia cadena al registro DX
0C1B:0128 MOV CX,000F ; veces que se desplegará la cadena
0C1B:012B MOV AH,09 ; copia valor 09 al registro AH
0C1B:012D INT 21 ; despliega cadena
0C1B:012F DEC CX ; decrementa en 1 a CX
0C1B:0130 JCXZ 0134 ; si CX es igual a 0 brinca a 0134
0C1B:0132 JMP 012D ; brinca a la dirección 012D
0C1B:0134 INT 20 ; termina el programa
```

En este ejemplo se usó la instrucción JCXZ para controlar la condición de salto, el significado de tal función es: brinca si CX=0
El tipo de control a utilizar dependerá de las necesidades de programación en determinado momento.

PROGRAMACIÓN EN ENSAMBLADOR

IMPORTANCIA DEL LENGUAJE ENSAMBLADOR

El lenguaje ensamblador es la forma más básica de programar un microprocesador para que éste sea capaz de realizar las tareas o los cálculos que se le requieran.
El lenguaje ensamblador es conocido como un lenguaje de bajo nivel, esto significa que nos permite controlar el 100 % de las funciones de un microprocesador, así como los periféricos asociados a éste.
A diferencia de los lenguajes de alto nivel, por ejemplo "Pascal", el lenguaje ensamblador no requiere de un compilador, esto es debido a que las instrucciones en lenguaje ensamblador son traducidas directamente a código binario y después son colocadas en memoria para que el microprocesador las tome directamente.
Aprender a programar en lenguaje ensamblador no es fácil, se requiere un cierto nivel de conocimiento de la arquitectura y organización de las computadoras, además del conocimiento de programación en algún otro lenguaje
La primera razón para trabajar con ensamblador es que proporciona la oportunidad de conocer más a fondo la operación de su PC, lo que permite el desarrollo de software de una manera más consistente.
La segunda razón es el control total de la PC que se tiene con el uso del mismo.
Otra razón es que los programas de ensamblador son más rápidos, más compactos y tienen mayor capacidad que los creados en otros lenguajes.
Por último el ensamblador permite una optimización ideal en los programas tanto en su tamaño como en su ejecución.
El Lenguaje Ensamblador es importante como se puede ver; es directamente traducible al Lenguaje de Máquina, y viceversa; simplemente, es una abstracción que facilita su uso para los seres humanos. Por otro lado, la computadora no entiende directamente al Lenguaje Ensamblador; es necesario traducirle a Lenguaje de Máquina. Pero, al ser tan directa la traducción, pronto aparecieron los programas Ensambladores, que son traductores que convierten el código fuente (en Lenguaje Ensamblador) a código objeto (es decir, a Lenguaje de Máquina). Surge como una necesidad de facilitar al programador la tarea de trabajar con lenguaje máquina sin perder el control directo con el hardware.

VENTAJAS Y DESVENTAJAS DEL LENGUAJE ENSAMBLADOR

Conocida es la evolución de los lenguajes de programación, cabe preguntarse: ¿En estos tiempos "modernos", para qué quiero el Lenguaje Ensamblador?
El proceso de evolución trajo consigo algunas desventajas, que se verán luego así como las ventajas de usar el Lenguaje Ensamblador, respecto a un lenguaje de alto nivel:
Ventajas del lenguaje ensamblador:

- Velocidad de ejecución de los programas
- Tamaño
- Flexibilidad

VELOCIDAD

El proceso de traducción que realizan los intérpretes, implica un proceso de cómputo adicional al que el programador quiere realizar. Por ello, nos encontraremos con que un intérprete es siempre más lento que realizar la misma acción en Lenguaje Ensamblador, simplemente porque tiene el costo adicional de estar traduciendo el programa, cada vez que lo ejecutamos.
De ahí nacieron los compiladores, que son mucho más rápidos que los intérpretes, pues hacen la traducción una vez y dejan el código objeto, que ya es Lenguaje de Máquina, y se puede ejecutar muy rápidamente. Aunque el proceso de traducción es más complejo y costoso que el de ensamblar un programa, normalmente podemos despreciarlo, contra las ventajas de codificar el programa más rápidamente.
Sin embargo, la mayor parte de las veces, el código generado por un compilador es menos eficiente que el código equivalente que un programador escribiría. La razón es que el compilador no tiene tanta inteligencia, y requiere ser capaz de crear código genérico, que sirva tanto para un programa como para otro; en cambio, un programador humano puede aprovechar las características específicas del problema, reduciendo la generalidad pero al mismo tiempo, no desperdicia ninguna instrucción, no hace ningún proceso que no sea necesario.

Para darnos una idea, en una PC, y suponiendo que todos son buenos programadores, un programa para ordenar una lista tardará cerca de 20 veces más en Visual Basic (un intérprete), y 2 veces más en C (un compilador), que el equivalente en Ensamblador.
Por ello, cuando es crítica la velocidad del programa, Ensamblador se vuelve un candidato lógico como lenguaje.
Ahora bien, esto no es un absoluto; un programa bien hecho en C puede ser muchas veces más rápido que un programa mal hecho en Ensamblador; sigue siendo sumamente importante la elección apropiada de algoritmos y estructuras de datos. Por ello, se recomienda buscar optimizar primero estos aspectos, en el lenguaje que se desee, y solamente usar Ensamblador cuando se requiere más optimización y no se puede lograr por estos medios.

TAMAÑO

Por las mismas razones que vimos en el aspecto de velocidad, los compiladores e intérpretes generan más código máquina del necesario; por ello, el programa ejecutable crece. Así, cuando es importante reducir el tamaño del ejecutable, mejorando el uso de la memoria y teniendo también beneficios en velocidad, puede convenir usar el lenguaje Ensamblador. Entre los programas que es crítico el uso mínimo de memoria, tenemos a los virus y manejadores de dispositivos (drivers). Muchos de ellos, por supuesto, están escritos en lenguaje Ensamblador.

FLEXIBILIDAD

Las razones anteriores son cuestión de grado: podemos hacer las cosas en otro lenguaje, pero queremos hacerlas más eficientemente. Pero todos los lenguajes de alto nivel tienen limitantes en el control; al hacer abstracciones, limitan su propia capacidad. Es decir, existen tareas que la máquina puede hacer, pero que un lenguaje de alto nivel no permite. Por ejemplo, en Visual Basic no es posible cambiar la resolución del monitor a medio programa; es una limitante, impuesta por la abstracción del GUI Windows. En cambio, en ensamblador es sumamente sencillo, pues tenemos el acceso directo al hardware del monitor.
Por otro lado, al ser un lenguaje más primitivo, el Ensamblador tiene ciertas desventajas respecto a los lenguajes de alto nivel:

Desventajas del lenguaje ensamblador:

- Tiempo de programación
- Programas fuentes grandes
- Peligro de afectar recursos inesperadamente
- Falta de portabilidad

TIEMPO DE PROGRAMACIÓN

Al ser de bajo nivel, el Lenguaje Ensamblador requiere más instrucciones para realizar el mismo proceso, en comparación con un lenguaje de alto nivel. Por otro lado, requiere de más cuidado por parte del programador, pues es propenso a que los errores de lógica se reflejen más fuertemente en la ejecución.
Por todo esto, es más lento el desarrollo de programas comparables en Lenguaje Ensamblador que en un lenguaje de alto nivel, pues el programador goza de una menor abstracción.

PROGRAMAS FUENTE GRANDES

Por las mismas razones que aumenta el tiempo, crecen los programas fuentes; simplemente, requerimos más instrucciones primitivas para describir procesos equivalentes. Esto es una desventaja porque dificulta el mantenimiento de los programas, y nuevamente reduce la productividad de los programadores.

PELIGRO DE AFECTAR RECURSOS INESPERADAMENTE

Tenemos la ventaja de que todo lo que se puede hacer en la máquina, se puede hacer con el Lenguaje Ensamblador (flexibilidad). El problema es que todo error que podamos cometer, o

todo riesgo que podamos tener, podemos tenerlo también en este Lenguaje. Dicho de otra forma, tener mucho poder es útil pero también es peligroso.
En la vida práctica, afortunadamente no ocurre mucho; sin embargo, al programar en este lenguaje verán que es mucho más común que la máquina se "cuelgue", "bloquee" o "se le vaya el avión"; y que se reinicialice. ¿Por qué?, porque con este lenguaje es perfectamente posible (y sencillo) realizar secuencias de instrucciones inválidas, que normalmente no aparecen al usar un lenguaje de alto nivel.
En ciertos casos extremos, puede llegarse a sobrescribir información del CMOS de la máquina; pero, si no la conservamos, esto puede causar que dejemos de "ver" el disco duro, junto con toda su información.

FALTA DE PORTABILIDAD

Como ya se mencionó, existe un lenguaje ensamblador para cada máquina; por ello, evidentemente no es una selección apropiada de lenguaje cuando deseamos codificar en una máquina y luego llevar los programas a otros sistemas operativos o modelos de computadoras. Si bien esto es un problema general a todos los lenguajes, es mucho más notorio en ensamblador: yo puedo reutilizar un 90% o más del código que desarrollo en "C", en una PC, al llevarlo a una RS/6000 con UNIX, y lo mismo si después lo llevo a una Macintosh, siempre y cuando esté bien hecho y siga los estándares de "C", y los principios de la programación estructurada. En cambio, si escribimos el programa en Ensamblador de la PC, por bien que lo desarrollemos y muchos estándares que sigamos, tendremos prácticamente que reescribir el 100 % del código al llevarlo a UNIX, y otra vez lo mismo al llevarlo a Mac.

SINTAXIS DE UNA LÍNEA EN ENSAMBLADOR

Un programa fuente en ensamblador contiene dos tipos de **sentencias**: las *instrucciones* y las *directivas*. Las instrucciones se aplican en tiempo de ejecución, pero las directivas sólo son utilizadas durante el ensamblaje.
El formato de una sentencia de instrucción es el siguiente:

[etiqueta] nombre_instrucción [operandos] [comentario]

Los corchetes, como es normal al explicar instrucciones en informática, indican que lo especificado entre ellos es opcional, dependiendo de la situación que se trate.

Campo de etiqueta: Es el nombre simbólico de la primera posición de una instrucción, puntero o dato. Consta de hasta 31 caracteres que pueden ser las letras de la A a la Z, los números del 0 al 9 y algunos caracteres especiales como «@», «_», «.» y «$».
Reglas:

- Si se utiliza el punto «.», éste debe colocarse como primer carácter de la etiqueta.
- El primer carácter no puede ser un dígito.
- No se pueden utilizar los nombres de instrucciones o registros como nombres de etiquetas.

Las etiquetas son de tipo NEAR cuando el campo de etiqueta finaliza con dos puntos (:); esto es, se considera cercana: quiere esto decir que cuando realizamos una llamada sobre dicha etiqueta el ensamblador considera que está dentro del mismo segmento de código (llamadas intrasegmento) y el procesador sólo carga el puntero de instrucciones IP. Téngase en cuenta que hablamos de instrucciones; las etiquetas empleadas antes de las directivas, como las directivas de definición de datos por ejemplo, no llevan los dos puntos y sin embargo son cercanas.

Las etiquetas son de tipo FAR si el campo de etiqueta no termina con los dos puntos: en estas etiquetas la instrucción a la que apunta no se encuentra en el mismo segmento de código sino en otro. Cuando es referenciada en una transferencia de control se carga el puntero de instrucciones IP y el segmento de código CS (llamadas intersegmento).

Campo de nombre: Contiene el mnemónico de las instrucciones o bien una directiva.

Campo de operandos: Indica cuales son los datos implicados en la operación. Puede haber 0, 1 ó 2; en el caso de que sean dos al 1º se le llama destino y al 2º -separado por una coma- fuente.

```
mov ax, es:[di]    -->    ax        destino

                   -->    es:[di]   origen
```

Campo de comentarios: Cuando en una línea hay un punto y coma (;) todo lo que sigue en la línea es un comentario que realiza aclaraciones sobre lo que se está haciendo en ese Programa; resulta de gran utilidad de cara a realizar futuras modificaciones al mismo.

CONSTANTES Y OPERADORES

Las sentencias fuente -tanto instrucciones como directivas- pueden contener constantes y operadores.

CONSTANTES
Pueden ser binarias (ej. 10010b), decimales (ej. 34d), hexadecimales (ej. 0E0h) u octales (ej. 21o ó 21q); también las hay de tipo cadena (ej. 'pepe', "juan") e incluso con comillas dentro de comillas de distinto tipo (como 'hola,"amigo"'). En las constantes hexadecimales, si el primer dígito no es numérico hay que poner un 0. Sólo se puede poner el signo (-) en las decimales (en las demás, calcúlese el complemento a dos) Por defecto, las numéricas están en base 10 si no se indica lo contrario con una directiva (poco recomendable como se verá).

OPERADORES ARITMÉTICOS
Pueden emplearse libremente (+), (-), (*) y (/) - en este último caso la división es siempre entera-. Es válida, por ejemplo, la siguiente línea en ensamblador (que se apoya en la directiva DW, que se verá más adelante, para reservar memoria para una palabra de 16 bits):

```
Dato    DW      12*(numero+65)/7
```

También se admiten los operadores MOD (resto de la división) y SHL/SHR (desplazar a la izquierda/derecha cierto número de bits). Obviamente, el ensamblador no codifica las instrucciones de desplazamiento (al aplicarse sobre datos constantes el resultado se calcula en tiempo de ensamblaje):

```
Dato    DW      (12 SHR 2) + 5
```

OPERADORES LÓGICOS

Pueden ser el AND, OR, XOR y NOT. Realizan las operaciones lógicas en las expresiones. Ej.:

```
MOV  BL,(255 AND 128) XOR 128   ; BL = 0
```

OPERADORES RELACIONALES

Devuelven condiciones de Verdadero (0FFFFh ó 0FFh) o Falso (0) evaluando una expresión. Pueden ser: EQ (igual), NE (no igual), LT (menor que), GT (mayor que), LE (menor o igual que), GE (mayor o igual que).
Ejemplo:

```
dato     EQU   100          ; «dato» vale 100
MOV  AL,dato GE 10  ; AL = 0FFh (Verdadero)
MOV  AH,dato EQ 99  ; AH = 0 (Falso)
```

OPERADORES DE RETORNO DE VALORES

* Operador SEG: devuelve el valor del segmento de la variable o etiqueta, sólo se puede emplear en programas de tipo EXE:

```
MOV  AX,SEG tabla_datos
```

* Operador OFFSET: devuelve el desplazamiento de la variable o etiqueta en su segmento:

```
MOV  AX,OFFSET variable
```

Si se desea obtener el offset de una variable respecto al grupo (directiva GROUP) de segmentos en que está definida y no respecto al segmento concreto en que está definida:

```
MOV  AX,OFFSET nombre_grupo:variable
```

También es válido:

```
MOV  AX,OFFSET DS:variable
```

* Operador .TYPE: devuelve el modo de la expresión indicada en un byte. El bit 0 indica modo «relativo al código» y el 1 modo «relativo a datos»; si ambos bits están inactivos significa modo absoluto. El bit 5 indica si la expresión es local (0 si está definida externamente o indefinida); el bit 7 indica si la expresión contiene una referencia externa. Este operador es útil sobre todo en las macros para determinar el tipo de los parámetros:

```
info     .TYPE  variable
```

* Operador TYPE: devuelve el tamaño (bytes) de la variable indicada. No válido en variables DUP:

```
kilos    DW   76
MOV  AX,TYPE kilos    ; AX = 2
```

Tratándose de etiquetas -en lugar de variables- indica si es lejana o FAR (0FFFEh) o cercana o NEAR (0FFFFh).

* Operadores SIZE y LENGTH: devuelven el tamaño (en bytes) o el nº de elementos, respectivamente, de la variable indicada (definida obligatoriamente con DUP):

```
matriz   DW    100 DUP (12345)
         MOV   AX,SIZE matriz     ; AX = 200
         MOV   BX,LENGTH matriz   ; BX = 100
```

* Operadores MASK y WIDTH: informan de los campos de un registro de bits (véase RECORD).

OPERADORES DE ATRIBUTOS

* Operador PTR: redefine el atributo de tipo (BYTE, WORD, DWORD, QWORD, TBYTE) o el de distancia (NEAR o FAR) de un operando de memoria. Por ejemplo, si se tiene una tabla definida de la siguiente manera:

```
Tabla    DW    10 DUP (0)        ; 10 palabras a 0
```

Para colocar en AL el primer byte de la misma, la instrucción MOV AL,tabla es incorrecta, ya que tabla (una cadena 10 **palabras**) no cabe en el registro AL. Lo que desea el programador debe indicárselo en este caso explícitamente al ensamblador de la siguiente manera:

```
MOV   AL,BYTE PTR tabla
```

Trabajando con varios segmentos, PTR puede redefinir una etiqueta NEAR de uno de ellos para convertirla en FAR desde el otro, con objeto de poder llamarla.

* Operadores CS:; DS:; ES: y SS: el ensamblador genera un prefijo de un byte que indica al microprocesador el segmento que debe emplear para acceder a los datos en memoria. Por defecto, se supone DS para los registros BX, DI o SI (o sin registros de base o índice) y SS para SP y BP. Si al acceder a un dato éste no se encuentra en el segmento por defecto, el ensamblador añadirá el byte adicional de manera automática. Sin embargo, el programador puede forzar también esta circunstancia:

```
MOV   AL,ES:variable
```

En el ejemplo, *variable* se supone ubicada en el Segmento Extra. Cuando se referencia una dirección fija hay que indicar el segmento, ya que el ensamblador no conoce en qué segmento está la variable; es uno de los pocos casos en que debe indicarse. Por ejemplo, la siguiente línea dará un error al ensamblar:

```
MOV   AL,[0]
```

Para solucionarlo hay que indicar en qué segmento está el dato (incluso aunque éste sea DS):

```
MOV   AL,DS:[0]
```

En este último ejemplo el ensamblador no generará el byte adicional ya que las instrucciones MOV operan por defecto sobre DS (como casi todas), pero ha sido necesario indicar DS para que el ensamblador nos entienda. Sin embargo, en el siguiente ejemplo no es necesario, ya que *midato* está declarado en el segmento de datos y el ensamblador lo sabe:

```
MOV   AL,midato
```

Por lo general no es muy frecuente la necesidad de indicar explícitamente el segmento: al acceder a una variable el ensamblador mira en qué segmento está declarada (véase la directiva SEGMENT) y según como estén asignados los ASSUME, pondrá o no el prefijo adecuado según sea conveniente. Es responsabilidad exclusiva del programador inicializar los registros de segmento al principio de los procedimientos para que el ASSUME no se quede en tinta mojada... sí se emplean con bastante frecuencia, sin embargo, los prefijos CS en las rutinas que gestionan interrupciones (ya que CS es el único registro de segmento que apunta en principio a las mismas, hasta que se cargue DS u otro).

* Operador SHORT: indica que la etiqueta referenciada, de tipo NEAR, puede alcanzarse con un salto corto (-128 a +127 posiciones) desde la actual situación del contador de programa. El ensamblador TASM, si se solicitan dos pasadas, coloca automáticamente instrucciones SHORT allí donde es posible, para economizar memoria (el MASM no)

* Operador '$': indica la posición del contador de posiciones («Location Counter») utilizado por el ensamblador dentro del segmento para llevar la cuenta de por dónde *se llega* ensamblando. Muy útil:

```
frase      DB     "simpático"
longitud   EQU    $-OFFSET frase
```

En el ejemplo, longitud tomará el valor 9.

* Operadores HIGH y LOW: devuelven la parte alta o baja, respectivamente (8 bits) de la expresión:

```
dato       EQU    1025
MOV    AL,LOW dato       ; AL = 1
MOV    AH,HIGH dato      ; AH = 4
```

PRINCIPALES DIRECTIVAS

La sintaxis de una sentencia directiva es muy similar a la de una sentencia de instrucción:

[nombre] nombre_directiva [operandos] [comentario]

Sólo es obligatorio el campo «nombre_directiva»; los campos han de estar separados por al menos un espacio en blanco. La sintaxis de «nombre» es análoga a la de la «etiqueta» de las líneas de instrucciones, aunque nunca se pone el sufijo «:». El campo de comentario cumple también las mismas normas. A continuación se explican las directivas más empleadas; aunque falta alguna que otra y las que son explicadas, no lo están en todos los casos, con gran profundidad.

DIRECTIVAS DE DEFINICIÓN DE DATOS

* DB (definir byte), DW (definir palabra), DD (definir doble palabra), DQ (definir cuádruple palabra), DT (definir 10 bytes): sirven para declarar las variables, asignándolas un valor inicial:

```
anno       DW    1991
mes        DB    12
numerazo   DD    12345678h
texto      DB    "Hola",13,10
```

Se pueden definir números reales de simple precisión (4 bytes) con DD, de doble precisión (8 bytes) con DQ y «reales temporales» (10 bytes) con DT; todos ellos con el formato empleado por el coprocesador. Para que el ensamblador interprete el número como real ha de llevar el punto decimal:

```
temperatura  DD    29.72
espanoles91  DQ    38.9E6
```

Con el operando DUP pueden definirse estructuras repetitivas. Por ejemplo, para asignar 100 bytes a cero y 25 palabras de contenido indefinido (no importa lo que el ensamblador asigne):

```
ceros      DB    100 DUP (0)
basura     DW    25 DUP (?)
```

Se admiten también los anidamientos. El siguiente ejemplo crea una tabla de bytes donde se repite 50 veces la secuencia 1,2,3,7,7:

```
tabla      DB    50 DUP (1, 2, 3, 2 DUP (7))
```

DIRECTIVAS DE DEFINICIÓN DE SÍMBOLOS

* EQU (EQUivalence): Asigna el valor de una expresión a un nombre simbólico fijo:

```
olimpiadas  EQU   1992
```

Donde *olimpiadas* ya no podrá cambiar de valor en todo el programa. Se trata de un operador muy flexible. Es válido hacer:

```
edad       EQU   [BX+DI+8]
MOV   AX,edad
```

* = (signo '='): asigna el valor de la expresión a un nombre simbólico variable: Análogo al anterior pero con posibilidad de cambiar en el futuro. Muy usada en macros (sobre todo con REPT)

```
num = 19
num = pepe + 1
dato = [BX+3]
dato = ES:[BP+1]
```

DIRECTIVAS DE CONTROL DEL ENSAMBLADOR

* ORG (ORiGin): pone el contador de posiciones del ensamblador, que indica el offset donde se deposita la instrucción o dato, donde se indique. En los programas COM (que se cargan en memoria con un OFFSET 100h) es necesario colocar al principio un ORG 100h, y un ORG 0 en los controladores de dispositivo (aunque si se omite se asume de hecho un ORG 0).

* END [expresión]: indica el final del fichero fuente. Si se incluye, *expresión* indica el punto donde arranca el programa. Puede omitirse en los programas EXE si éstos constan de un sólo módulo. En los .COM es preciso indicarla y, además, la expresión -realmente una etiqueta- debe estar inmediatamente después del ORG 100h.

* .286, .386 y .8087 obligan al ensamblador a reconocer instrucciones específicas del 286, el 386 y del 8087. También debe ponerse el «.» inicial. Con .8086 se fuerza a que de nuevo sólo se reconozcan instrucciones del 8086 (modo por defecto). La directiva .386 puede ser colocada dentro de un segmento (entre las directivas SEGMENT/ENDS) con el ensamblador TASM, lo que permite emplear instrucciones de 386 con segmentos de 16 bits; alternativamente se puede ubicar fuera de los segmentos (obligatorio en MASM) y definir éstos explícitamente como de 16 bits con USE16.

* EVEN: fuerza el contador de posiciones a una posición par, intercalando un byte con la instrucción NOP si es preciso. En buses de 16 ó más bits (8086 y superiores, no en 8088) es dos veces más rápido el acceso a palabras en posición par:

```
EVEN
dato_rapido DW     0
```

* .RADIX n: cambia la base de numeración por defecto. Bastante desaconsejable dada la notación elegida para indicar las bases por parte de IBM/Microsoft (si se cambia la base por defecto a 16, ¡los números no pueden acabar en 'd' ya que se confundirían con el sufijo de decimal!: lo ideal sería emplear un prefijo y no un sufijo, que a menudo obliga además a iniciar los números por 0 para distinguirlos de las etiquetas).

DIRECTIVAS DE DEFINICIÓN DE SEGMENTOS Y PROCEDIMIENTOS

* SEGMENT-ENDS: SEGMENT indica el comienzo de un segmento (código, datos, pila, etc.) y ENDS su final. El programa más simple, de tipo COM, necesita la declaración de un segmento (común para datos, código y pila). Junto a SEGMENT puede aparecer, opcionalmente, el tipo de *alineamiento*, la *combinación*, el uso y la clase:

```
nombre SEGMENT [alineamiento] [combinación] [uso] ['clase']
. . . . .
nombre ENDS
```

Se pueden definir unos segmentos dentro de otros (el ensamblador los ubicará unos tras otros). El alineamiento puede ser BYTE (ninguno), WORD (el segmento comienza en posición par), DWORD (comienza en posición múltiplo de 4), PARA (comienza en una dirección múltiplo de 16, opción por defecto) y PAGE (comienza en dirección múltiplo de 256). La combinación puede ser:

- (No indicada): los segmentos se colocan unos tras otros físicamente, pero son lógicamente independientes: cada uno tiene su propia base y sus propios offsets relativos.
- **PUBLIC**: usado especialmente cuando se trabaja con segmentos definidos en varios ficheros que se ensamblan por separado o se compilan con otros lenguajes, por ello debe declararse un nombre entre comillas simples -'clase'- para ayudar al linkador. Todos los segmentos PUBLIC de igual nombre y clase tienen una base común y son colocados adyacentemente unos tras otros, siendo el offset relativo al primer segmento cargado.
- **COMMON**: similar, aunque ahora los segmentos de igual nombre y clase se solapan. por ello, las variables declaradas han de serlo en el mismo orden y tamaño.
- **AT**: asocia un segmento a una posición de memoria fija, no para ensamblar sino para declarar variables (inicializadas siempre con '?') de cara a acceder con comodidad a zonas de ROM, vectores de interrupción, etc.

Ejemplo:

```
vars_bios   SEGMENT AT 40h
p_serie0    DW ?
vars_bios   ENDS
```

De esta manera, la dirección del primer puerto serie puede obtenerse de esta manera (por ejemplo):

```
MOV    AX,variables_bios    ; segmento
MOV    ES,AX                ; inicializar ES
MOV    AX,ES:p_serie0
```

- **STACK**: segmento de pila, debe existir uno en los programas de tipo EXE; además el Linkador de Borland (TLINK 4.0) exige obligatoriamente que la clase de éste sea también 'STACK', con el TLINK de Microsoft no siempre es necesario indicar la clase del segmento de pila. Similar, por lo demás, a PUBLIC.

- **MEMORY**: segmento que el linkeador ubicará al final de todos los demás, lo que permitiría saber dónde acaba el programa. Si se definen varios segmentos de este tipo el ensamblador acepta el primero y trata a los demás como COMMON. Téngase en cuenta que el linkeador no soporta esta característica, por lo que emplear MEMORY es equivalente a todos los efectos a utilizar COMMON. Olvídese de MEMORY.

El '*uso'* indica si el segmento es de 16 bits o de 32; al emplear la directiva .386 se asumen por defecto segmentos de 32 bits por lo que es necesario declarar USE16 para conseguir que los segmentos sean interpretados como de 16 bits por el linkeador, lo que permite emplear algunas instrucciones del 386 en el modo real del microprocesador y bajo el sistema operativo DOS.
Por último, 'clase' es un nombre opcional que empleará el linkeador para encadenar los módulos, siendo conveniente nombrar la clase del segmento de pila con 'STACK'.

* ASSUME (Suponer): Indica al ensamblador el registro de segmento que se va a utilizar para direccionar cada segmento dentro del módulo. Esta instrucción va normalmente inmediatamente después del SEGMENT. El programa más sencillo necesita que se «suponga» CS como mínimo para el segmento de código, de lo contrario el ensamblador empezará a protestar un montón al no saber que registro de segmento asociar al código generado. También conviene hacer un Assume del registro de segmento DS hacia el segmento de datos, incluso en el caso de que éste sea el mismo que el de código: si no, el ensamblador colocará un byte de prefijo adicional en todos los accesos a memoria para forzar que éstos sean sobre CS. Se puede indicar ASSUME NOTHING para cancelar un ASSUME anterior. También se puede indicar el nombre de un grupo o emplear «SEG variable» o «SEG etiqueta» en vez de nombre_segmento:

```
ASSUME reg_segmento:nombre_segmento[,...]
```

* PROC-ENDP permite dar nombre a una subrutina, marcando con claridad su inicio y su fin.

Aunque es redundante, es muy recomendable para estructurar los programas.

```
Cls     PROC
.....
Cls     ENDP
```

El atributo FAR que aparece en ocasiones junto a PROC indica que es un procedimiento lejano y las instrucciones RET en su interior se ensamblan como RETF (los CALL hacia él serán, además, de 32 bits). Observar que la etiqueta nunca termina con dos puntos.

DIRECTIVAS DE REFERENCIAS EXTERNAS

* PUBLIC: permite hacer visibles al exterior (otros ficheros objeto resultantes de otros listados en ensamblador u otro lenguaje) los símbolos -variables y procedimientos- indicados. Necesario para programación modular e interfaces con lenguajes de alto nivel. Por ejemplo:

```
PUBLIC proc1, var_x
proc1     PROC   FAR
          ...
proc1     ENDP
var_x     DW     0
```

Declara la variable var_x y el procedimiento proc1 como accesibles desde el exterior por medio de la directiva EXTRN.

EXTRN: Permite acceder a símbolos definidos en otro fichero objeto (resultante de otro ensamblaje o de una compilación de un lenguaje de alto nivel); es necesario también indicar el tipo del dato o procedimiento (BYTE, WORD o DWORD; NEAR o FAR; se emplea además ABS para las constantes numéricas):

```
EXTRN proc1:FAR, var_x:WORD
```

En el ejemplo se accede a los símbolos externos proc1 y var_x (ver ejemplos de PUBLIC) y a continuación sería posible hacer un CALL proc1 o un MOV CX,var_x. Si la directiva EXTRN se coloca dentro de un segmento, se supone el símbolo dentro del mismo. Si el símbolo está en otro segmento, debe colocarse EXTRN fuera de todos los segmentos indicando explícitamente el prefijo del registro de segmento (o bien hacer el ASSUME apropiado) al referenciarlo. Evidentemente, al final, al linkear habrá que enlazar este módulo con el que define los elementos externos.

INCLUDE nombre_fichero: Añade al fichero fuente en proceso de ensamblaje el fichero indicado, en el punto en que aparece el INCLUDE. Es exactamente lo mismo que mezclar ambos ficheros con un editor de texto. Ahorra trabajo en fragmentos de código que se repiten en varios programas (como quizá una librería de macros). No se recomiendan INCLUDE's anidados.

DIRECTIVAS DE DEFINICIÓN DE BLOQUES

NAME nombre_modulo_objeto: indica el nombre del módulo objeto. Si no se incluye NAME, se tomará de la directiva TITLE o, en su defecto, del nombre del propio fichero fuente.

GROUP segmento1, segmento2,... permite agrupar dos o más segmentos lógicos en uno sólo de no más de 64 Kb totales (ojo: el ensamblador no comprueba este extremo, aunque sí el enlazador). Ejemplo:

```
superseg   GROUP datos, codigo, pila

codigo     SEGMENT
           ...
codigo     ENDS

datos      SEGMENT
dato       DW 1234
datos      ENDS

pila       SEGMENT STACK 'STACK'
           DB 128 DUP (?)
pila       ENDS
```

Cuando se accede a un dato definido en algún segmento de un grupo y se emplea el operador OFFSET es preciso indicar el nombre del grupo como prefijo, de lo contrario el ensamblador no generará el desplazamiento correcto ¡ni emitirá errores!:

```
MOV   AX,dato              ; ¡incorrecto!
MOV   AX,supersegmento:dato  ; correcto
```

La ventaja de agrupar segmentos es poder crear programas COM y SYS que contengan varios segmentos. En todo caso, téngase en cuenta aún en ese caso que no pueden emplearse todas las características de la programación con segmentos (por ejemplo, no se puede utilizar la directiva SEG ni debe existir segmento de pila).

LABEL: Permite referenciar un símbolo con otro nombre, siendo factible redefinir el tipo. La sintaxis es: nombre LABEL tipo (tipo = BYTE, WORD, DWORD, NEAR o FAR). Ejemplo:

```
palabra     LABEL  WORD
byte_bajo   DB     0
byte_alto   DB     0
```

En el ejemplo, con MOV AX,palabra se accederá a ambos bytes a la vez (el empleo de MOV AX,byte_bajo daría error: no se puede cargar un sólo byte en un registro de 16 bits y el ensamblador no supone que realmente pretendíamos tomar dos bytes consecutivos de la memoria).

STRUC - ENDS: permite definir registros al estilo de los lenguajes de alto nivel, para acceder de una manera más elegante a los campos de una información con cierta estructura. Estos campos pueden componerse de cualquiera de los tipos de datos simples (DB, DW, DD, DQ, DT) y pueden ser modificables o no en función de si son simples o múltiples, respectivamente:

```
alumno      STRUC
mote        DB '0123456789'          ; modificable
edadaltura  DB 20,175                ; no modificable
peso        DB 0                     ; modificable
otros       DB 10 DUP(0)             ; no modificable
telefono    DD ?                     ; modificable
alumno      ENDS
```

La anterior definición de estructura no lleva implícita la reserva de memoria necesaria, la cual ha de hacerse expresamente utilizando los ángulos '<' y '>':

```
felipe      alumno <'Gordinflas',,101,,251244>
```

En el ejemplo se definen los campos modificables (los únicos definibles) dejando sin definir (comas consecutivas) los no modificables, creándose la estructura 'felipe' que ocupa 27 bytes. Las cadenas de caracteres son rellenadas con espacios en blanco al final si no alcanzan el tamaño máximo de la declaración. El TASM es más flexible y permite definir también el primer elemento de los campos múltiples sin dar error. Tras crear la estructura, es posible acceder a sus elementos utilizando un (.) para separar el nombre del campo:

```
MOV    AX,OFFSET felipe.telefono
LEA    BX,felipe
MOV    CL,[BX].peso      ; equivale a [BX+12]
```

RECORD: similar a STRUC pero operando con campos de bits. Permite definir una estructura determinada de byte o palabra para operar con comodidad. Sintaxis:

```
nombre      RECORD nombre_de_campo:tamaño[=valor],...
```

Donde *nombre* permitirá referenciar la estructura en el futuro, *nombre_de_campo* identifica los distintos campos, a los que se asigna un tamaño (en bits) y opcionalmente un valor por defecto.

```
registro    RECORD a:2=3, b:4=5, c:1
```

La estructura *registro* totaliza 7 bits, por lo que ocupa un byte. Está dividida en tres campos que ocupan los 7 bits **menos** significativos del byte: el campo A ocupa los bits 6 y 5, el campo B ocupa los bits 1 al 4 y el campo C el bit 0:

6 5	4 3 2 1	0
1 1	0 1 0 1	?

La reserva de memoria se realiza, por ejemplo, de la siguiente manera:

```
reg1        registro <2,,1>
```

Quedando *reg1* con el valor binario 1001011 (el campo B permanece inalterado y el A y C toman los valores indicados). Ejemplos de operaciones soportadas:

```
MOV   AL, A          ; AL = 5 (desplazamiento del bit
                     ;      menos significativo de A)
MOV   AL, MASK A     ; AL = 01100000b (máscara de A)
MOV   AL, WIDTH A    ; AL = 2 (anchura de A)
```

DIRECTIVAS CONDICIONALES

Se emplean para que el ensamblador evalúe unas condiciones y, según ellas, ensamble o no ciertas zonas de código. Es frecuente, por ejemplo, de cara a generar código para varios ordenadores: pueden existir ciertos símbolos definidos que indiquen en un momento dado si hay que ensamblar ciertas zonas del listado o no de manera condicional, según la máquina. En los fragmentos en ensamblador del código que generan los compiladores también aparecen con frecuencia (para actuar de manera diferente, por ejemplo, según el modelo de memoria). Es interesante también la posibilidad de definir un símbolo que indique que el programa está en fase de pruebas y ensamblar código adicional en ese caso con objeto de depurarlo. Sintaxis:

```
IFxxx    [símbolo/exp./arg.]    ; xxx es la condición
...
ELSE                            ; el ELSE es opcional
...
ENDIF
```

```
IF      expresion       (expresión distinta de cero)
IFE     expresión       (expresión igual a cero)
IF1                     (pasada 1 del ensamblador)
IF2                     (pasada 2 del ensamblador)
IFDEF   símbolo         (símbolo definido o declarado como externo)
IFNDEF  símbolo         (símbolo ni definido ni declarado como externo)
IFB     <argumento>     (argumento en blanco en macros -incluir '<' y '>'-)
IFNB    <argumento>     (lo contrario, también es obligado poner '<' y '>')
IFIDN   <arg1>, <arg2>  (arg1 idéntico a arg2, requiere '<' y '>')
IFDIF   <arg1>, <arg2>  (arg1 distinto de arg2, requiere '<' y '>')
```

DIRECTIVAS DE LISTADO

* PAGE num_lineas, num_columnas: Formatea el listado de salida; por defecto son 66 líneas por página (modificable entre 10 y 255) y 80 columnas (seleccionable de 60 a 132). PAGE salta de página e incrementa su número. «PAGE +» indica capítulo nuevo (y se incrementa el número).

*TITLE título: indica el título que aparece en la 1ª línea de cada página (máximo 60 caracteres).

*SUBTTL subtítulo: Ídem con el subtítulo (máx. 60 caracteres).

*.LALL: Listar las macros y sus expansiones.

*.SALL: No listar las macros ni sus expansiones.

*.XALL: Listar sólo las macros que generan código objeto.

*.XCREF: Suprimir listado de referencias cruzadas (listado alfabético de símbolos junto al nº de línea en que son definidos y referenciados, de cara a facilitar la depuración).

*.CREF: Restaurar listado de referencias cruzadas.

*.XLIST: Suprimir el listado ensamblador desde ese punto.

*.LIST: Restaurar de nuevo la salida de listado ensamblador.

*COMMENT delimitador comentario delimitador: Define un comentario que puede incluso ocupar varias líneas, el delimitador (primer carácter no blanco ni tabulador que sigue al COMMENT) indica el inicio e indicará más tarde el final del comentario. ¡No olvidar cerrar el comentario!.

* %OUT mensaje: escribe en la consola el mensaje indicado durante la fase de ensamblaje y al llegar a ese punto del listado, excepto cuando el listado es por pantalla y no en fichero.

*.LFCOND: Listar los bloques de código asociados a una condición falsa (IF).

*.SFCOND: suprimir dicho listado.

* TFCOND: Invertir el modo vigente de listado de los bloques asociados a una condición falsa.

SOFTWARE NECESARIO PARA CREAR UN PROGRAMA EN ENSAMBLADOR

Para poder crear un programa se requieren varias herramientas.
Primero un editor para crear el programa fuente. Segundo un compilador que no es más que un programa que "traduce" el programa fuente a un programa objeto. Y tercero un enlazador o linker, que genere el programa ejecutable a partir del programa objeto.
El editor puede ser cualquier editor de textos que se tenga a la mano, como compilador utilizaremos el MASM (macro ensamblador de Microsoft) ya que es el más común, y como enlazador utilizaremos el programa Tlink.
La extensión usada para que MASM reconozca los programas fuente en ensamblador es .ASM; una vez traducido el programa fuente, el MASM crea un archivo con la extensión .OBJ, este archivo contiene un "formato intermedio" del programa, llamado así porque aún no es ejecutable pero tampoco es ya un programa en lenguaje fuente. El enlazador genera, a partir de un archivo .OBJ o la combinación de varios de estos archivos, un programa ejecutable, cuya extensión es usualmente .EXE aunque también puede ser .COM, dependiendo de la forma en que se ensambló.

UTILIZACIÓN DEL MASM

Una vez que se creó el programa fuente se debe pasar al MASM para crear el código intermedio, el cual queda guardado en un archivo con extensión .OBJ. El comando para realizar esto es:

MASM Nombre_Archivo; [Enter]

Donde Nombre_Archivo es el nombre del programa fuente con extensión .ASM que se va a traducir.

El punto y coma utilizado después del nombre del archivo le indican al macro ensamblador que genere directamente el código intermedio, de omitirse este carácter el MASM pedirá el nombre del archivo a traducir, el nombre del archivo que se generará así como opciones de listado de información que puede proporcionar el traductor.
Es posible ejecutar el MASM utilizando parámetros para obtener un fin determinado, toda la lista de los mismos se encuentra en el manual del programa. Luego se indicará la forma de pasar dichos parámetros al MASM:

Todo parámetro va después del símbolo "/". Es posible utilizar varios parámetros a la vez. Una vez tecleados todos los parámetros se escribe el nombre del archivo a ensamblar. Por ejemplo, si queremos que el MASM ensamble un programa llamado prueba, y además deseamos que despliegue el número de líneas fuente y símbolos procesados (eso lo realiza con el parámetro /v), y si ocurre un error que nos diga en que línea ocurrió (con el parámetro /z), entonces tecleamos:

MASM /v /z prueba;

Las opciones pueden ser:

- **/A** escribe los segmentos en orden alfabético
- **/S** escribe los segmentos en orden del fuente
- **/B***num* fija buffer de tamaño *num*
- **/C** especifica un archivo de referencias cruzadas
- **/L** especifica un listado de ensamble
- **/D** crea listado del paso 1
- **/D***sym* define un símbolo que puede usarse en el ensamble
- **/I***path* fija path para búscar archivos a incluir
- **/ML** mantiene sensitividad de letras (mayús./minús) en nombres
- **/MX** mantiene sensitividad en nombre publicos y externos
- **/MU** convierte nombres a mayúsculas
- **/N** suprime tablas en listados
- **/P** checa por código impuro
- **/R** crea código para instrucciones de punto flotante
- **/E** crea código para emular instrucciones de punto flotante
- **/T** suprime mensajes de ensable exitoso

/V despliega estadísticas adicionales en pantalla
/X incluir condicionales falsos en pantalla
/Z despliega líneas de error en pantalla

USO DEL ENLAZADOR (LINKER)

El MASM únicamente puede crear programas en formato. OBJ, los cuales no son ejecutables por si solos, es necesario un enlazador que genere el código ejecutable.
La utilización del enlazador es muy parecida a la del MASM, únicamente se teclea en el indicador del DOS:

TLINK Nombre_Archivo ;

Donde Nombre_Archivo es el nombre del programa intermedio (OBJ). Esto generara directamente un archivo con el nombre del programa intermedio y la extensión .EXE

FORMATO INTERNO DE UN PROGRAMA

Funcionamiento interno (ejecución de un programa)

Para que un microprocesador ejecute un programa es necesario que éste haya sido ensamblado, enlazado y cargado en memoria.
Una vez que el programa se encuentra en la memoria, el microprocesador ejecuta los siguientes pasos:

1.- Extrae de la memoria la instrucción que va a ejecutar y la coloca en el registro interno de instrucciones.

2.- Cambia el registro apuntador de instrucciones (IP) de modo que señale a la siguiente instrucción del programa.

3.- Determina el tipo de instrucción que acaba de extraer.

4.- Verifica si la instrucción requiere datos de la memoria y, si es así, determina donde están situados.

5.- Extrae los datos, si los hay, y los carga en los registros internos del CPU.

6.- Ejecuta la instrucción.

7.- Almacena los resultados en el lugar apropiado.

8.- Regresa al paso 1 para ejecutar la instrucción siguiente.

Este procedimiento lo lleva a cabo el microprocesador millones de veces por segundo.

FORMATO EXTERNO DE UN PROGRAMA

Además de definir ciertas reglas para que el ensamblador pueda entender una instrucción es necesario darle cierta información de los recursos que se van a utilizar, como por ejemplo los segmentos de memoria que se van a utilizar, datos iniciales del programa y también donde inicia y donde termina nuestro código.

Un programa sencillo puede ser el siguiente:

```
.MODEL SMALL
.CODE
Programa:
MOV AX,4C00H
INT 21H
.STACK
END Programa
```

El programa realmente no hace nada, únicamente coloca el valor 4C00H en el registro AX, para que la interrupción 21H termine el programa, pero nos da una idea del formato externo en un programa de ensamblador.
La directiva .MODEL define el tipo de memoria que se utilizará; la directiva .CODE nos indica que lo que está a continuación es nuestro programa; la etiqueta Programa indica al ensamblador el inicio del programa; la directiva STACK le pide al ensamblador que reserve un espacio de memoria para las operaciones de la pila; la instrucción END Programa marca el final del programa.

Ejemplo práctico de un programa

Aquí se ejemplificará un programa que escriba una cadena en pantalla:

```
.MODEL SMALL
.CODE
Programa:
MOV AX, @DATA
MOV DS, AX
MOV DX, Offset Texto
MOV AH, 9
INT 21H
MOV AX,4C00H
INT 21H
.DATA
Texto DB 'Mensaje en pantalla.$'
.STACK
END Programa
```

Los primeros pasos son iguales a los del programa anterior: se define el modelo de memoria, se indica donde inicia el código del programa y en donde comienzan las instrucciones.

A continuación se coloca @DATA en el registro AX para después pasarlo al registro DS ya que no se puede copiar directamente una constante a un registro de segmento. El contenido de @DATA es el número del segmento que será utilizado para los datos. Luego se guarda en el registro DX un valor dado por "Offset Texto" que nos da la dirección donde se encuentra la cadena de caracteres en el segmento de datos. Luego utiliza la opción 9 (Dada por el valor de AH) de la interrupción 21H para desplegar la cadena posicionada en la dirección que contiene DX. Por último utiliza la opción 4CH de la interrupción 21H para terminar la ejecución del programa (aunque cargamos al registro AX el valor 4C00H la interrupción 21H solo toma como opción el contenido del registro AH)
La directiva .DATA le indica al ensamblador que lo que está escrito a continuación debe almacenarlo en el segmento de memoria destinado a los datos. La directiva DB es utilizada para Definir Bytes, esto es, asignar a cierto identificador (en este caso "Texto") un valor, ya sea una constante o una cadena de caracteres, en este último caso deber estar entre comillas sencillas (') y terminar con el símbolo "$".

SEGMENTOS

La arquitectura de los procesadores x86 obliga al uso de segmentos de memoria para manejar la información, el tamaño de estos segmentos es de 64kb.
La razón de ser de estos segmentos es que, considerando que el tamaño máximo de un número que puede manejar el procesador esta dado por una palabra de 16 bits o registro, no sería posible accesar a más de 65536 localidades de memoria utilizando uno solo de estos registros, ahora, si se divide la memoria de la PC en grupos o segmentos, cada uno de 65536 localidades, y utilizamos una dirección en un registro exclusivo para localizar cada segmento, y entonces cada dirección de una casilla específica la formamos con dos registros, nos es posible accesar a una cantidad de 4294967296 bytes de memoria, lo cual es, en la actualidad, más memoria de la que veremos instalada en una PC.
Para que el ensamblador pueda manejar los datos es necesario que cada dato o instrucción se encuentren localizados en el área que corresponde a sus respectivos segmentos. El ensamblador accesa a esta información tomando en cuenta la localización del segmento, dada por los registros DS, ES, SS y CS, y dentro de dicho registro la dirección del dato específico. Es por ello que cuando creamos un programa empleando el Debug en cada línea que vamos ensamblando aparece algo parecido a lo siguiente:

1CB0:0102 MOV AX, BX

En donde el primer número, 1CB0, corresponde al segmento de memoria que se está utilizando, el segundo se refiere la dirección dentro de dicho segmento, y a continuación aparecen las instrucciones que se almacenaran a partir de esa dirección.
La forma de indicarle al ensamblador con cuales de los segmentos se va a trabajar es por medio de las directivas .CODE, .DATA y .STACK.
El ensamblador se encarga de ajustar el tamaño de los segmentos tomando como base el número de bytes que necesita cada instrucción que va ensamblando, ya que sería un desperdicio de memoria utilizar los segmentos completos. Por ejemplo, si un programa únicamente necesita 10 kb para almacenar los datos, el segmento de datos únicamente será de 10 kb y no de los 64 kb que puede manejar.

TABLA DE SÍMBOLOS

A cada una de las partes de una línea de código en ensamblador se le conoce como token, por ejemplo en la línea de código

MOV AX,Var

Tenemos tres tokens, la instrucción MOV, el operando AX, y el operando VAR. El ensamblador lo que hace para generar el código OBJ es leer cada uno de los tokens y buscarlo en una tabla interna de "equivalencias" conocida como tabla de palabras reservadas, que es donde se encuentran todos los significados de los mnemónicos que utilizamos como instrucciones.

Siguiendo este proceso, el ensamblador lee MOV, lo busca en su tabla y al encontrarlo lo identifica como una instrucción del procesador, así mismo lee AX y lo reconoce como un registro del procesador, pero al momento de buscar el token Var en la tabla de palabras reservadas no lo encuentra y entonces lo busca en la tabla de símbolos que es una tabla donde se encuentran los nombres de las variables, constantes y etiquetas utilizadas en el programa donde se incluye su dirección en memoria y el tipo de datos que contiene.

Algunas veces el ensamblador se encuentra con algún token no definido en el programa, lo que hace en estos casos es dar una segunda pasada por el programa fuente para verificar todas las referencias a ese símbolo y colocarlo en la tabla de símbolos. Existen símbolos que no los va a encontrar ya que no pertenecen a ese segmento y el programa no sabe en que parte de la memoria se encontrara dicho segmento, en este momento entra en acción el enlazador, el cual crea la estructura que necesita el cargador para que el segmento y el token sean definidos cuando se cargue el programa y antes de que el mismo sea ejecutado.

MOVIMIENTO DE DATOS

En todo programa es necesario mover datos en la memoria y en los registros de la UCP; existen diversas formas de hacer esto: puede copiar datos de la memoria a algún registro, de registro a registro, de un registro a una pila, de la pila a un registro, transmitir datos hacia dispositivos externos así como recibir datos de dichos dispositivos.
Este movimiento de datos está sujeto a reglas y restricciones. Algunas de ellas son las que se citan a continuación.

- No es posible mover datos de una localidad de memoria a otra directamente, es necesario primero mover los datos de la localidad origen hacia un registro y luego del registro a la localidad destino.

- No se puede mover una constante directamente a un registro de segmentos, primero se debe mover a un registro de la UCP.

- Es posible mover bloques de datos por medio de las instrucciones movs, que copia una cadena de bytes o palabras; movsb que copia n bytes de una localidad a otra; y movsw copia n palabras de una localidad a otra. Las dos últimas instrucciones toman los valores de las direcciones definidas por DS:SI como grupo de datos a mover y ES:DI como nueva localización de los datos.

Para mover los datos también existen las estructuras llamadas pilas, en este tipo de estructuras los datos se introducen con la instrucción PUSH y se extraen con la instrucción POP.
En una pila el primer dato introducido es el último que podemos sacar, esto es, si en nuestro programa utilizamos las instrucciones:

```
PUSH AX
PUSH BX
PUSH CX
```

Para devolver los valores correctos a cada registro al momento de sacarlos de la pila es necesario hacerlo en el siguiente orden:

```
POP CX
POP BX
POP AX
```

Para la comunicación con dispositivos externos se utilizan el comando OUT para mandar información a un puerto y el comando IN para leer información recibida desde algún puerto.
La sintaxis del comando OUT es:

```
OUT DX,AX
```

Donde DX contiene el valor del puerto que se utilizará para la comunicación y AX contiene la información que se mandará.

La sintaxis del comando IN es:

```
IN AX,DX
```

Donde AX es el registro donde se guardará la información que llegue y DX contiene la dirección del puerto por donde llegará la información.

OPERACIONES LÓGICAS Y ARITMÉTICAS

Las instrucciones de las operaciones lógicas son: AND, NOT, OR y XOR; estas trabajan sobre los bits de sus operandos.

Para verificar el resultado de operaciones recurrimos a las instrucciones CMP y TEST.

Las instrucciones utilizadas para las operaciones algebraicas son: para sumar ADD, para restar SUB, para multiplicar MUL y para dividir DIV.

Casi todas las instrucciones de comparación están basadas en la información contenida en el registro de banderas. Normalmente las banderas de este registro que pueden ser directamente manipuladas por el programador son la bandera de dirección de datos DF, usada para definir las operaciones sobre cadenas. Otra que también puede ser manipulada es la bandera IF por medio de las instrucciones STI y CLI, para activar y desactivar respectivamente las interrupciones.

SALTOS, CICLOS Y PROCEDIMIENTOS

Los saltos incondicionales en un programa escrito en lenguaje ensamblador están dados por la instrucción JMP; un salto es alterar el flujo de la ejecución de un programa enviando el control a la dirección indicada.
Un ciclo, conocido también como iteración; es la repetición de un proceso un cierto número de veces hasta que alguna condición se cumpla. En estos ciclos se utilizan los brincos "condiciónales" basados en el estado de las banderas. Por ejemplo la instrucción JNZ que salta solamente si el resultado de una operación es diferente de cero y la instrucción JZ que salta si el resultado de la operación es cero.
Por último tenemos los procedimientos o rutinas, que son una serie de pasos que se usaran repetidamente en el programa y en lugar de escribir todo el conjunto de pasos únicamente se les llama por medio de la instrucción CALL.

Un procedimiento en ensamblador es aquel que inicie con la palabra PROC y termine con la palabra RET.
Realmente lo que sucede con el uso de la instrucción CALL es que se guarda en la pila el registro IP y se carga la dirección del procedimiento en el mismo registro, conociendo que IP contiene la localización de la siguiente instrucción que ejecutara la UCP, entonces podemos darnos cuenta que se desvía el flujo del programa hacia la dirección especificada en este registro. Al momento en que se llega a la palabra RET se saca de la pila el valor de IP con lo que se devuelve el control al punto del programa donde se invocó al procedimiento.
Es posible llamar a un procedimiento que se encuentre ubicado en otro segmento, para esto el contenido de CS (que nos indica que segmento se está utilizando) es empujado también en la pila.

INSTRUCCIONES DE OPERACIÓN SOBRE DATOS

INSTRUCCIONES DE TRANSFERENCIA

Son utilizadas para mover los contenidos de los operandos. Cada instrucción se puede usar con diferentes modos de direccionamiento.

MOV
MOVS (MOVSB) (MOVSW)

INSTRUCCIONES DE CARGA

Son instrucciones específicas de los registros. Son usadas para cargar en algún registro bytes o cadenas de bytes.

LODS (LODSB) (LODSW)
LAHF
LDS
LEA
LES

INSTRUCCIONES DE LA PILA

Estas instrucciones permiten el uso de la pila para almacenar y extraer datos.

POP
POPF
PUSH
PUSHF

INSTRUCCIÓN MOV

Propósito: Transferencia de datos entre celdas de memoria, registros y acumulador.

Sintaxis:

MOV Destino,Fuente

Donde Destino es el lugar a donde se moverán los datos y fuente es el lugar donde se encuentran dichos datos.

Los diferentes movimientos de datos permitidos para esta instrucción son:

Destino: memoria. Fuente: acumulador
Destino: acumulador. Fuente: memoria
Destino: registro de segmento. Fuente: memoria/registro
Destino: memoria/registro. Fuente: registro de segmento
Destino: registro. Fuente: registro
Destino: registro. Fuente: memoria
Destino: memoria. Fuente: registro
Destino: registro. Fuente: dato inmediato
Destino: memoria. Fuente: dato inmediato

Ejemplo:

```
MOV AX,0006h
MOV BX,AX
MOV AX,4C00h
INT 21H
```

Este pequeño programa mueve el valor 0006H al registro AX, luego mueve el contenido de AX (0006h) al registro BX, por último mueve el valor 4C00h al registro AX para terminar la ejecución con la opción 4C de la interrupción 21h.

INSTRUCCIÓN MOVS (MOVSB) (MOVSW)

Propósito: Mover cadenas de bytes o palabras desde la fuente, direccionada por SI, hasta el destino direccionado por DI.

Sintaxis:

MOVS

Este comando no necesita parámetros ya que toma como dirección fuente el contenido del registro SI y como destino el contenido de DI. La secuencia de instrucciones siguiente ilustran esto:

```
MOV SI, OFFSET VAR1
MOV DI, OFFSET VAR2
MOVS
```

Primero inicializamos los valores de SI y DI con las direcciones de las variables VAR1 y VAR2 respectivamente, después al ejecutar MOVS se copia el contenido de VAR1 a VAR2.

Los comandos MOVSB y MOVSW se utilizan de la misma forma que MOVS, el primero mueve un byte y el segundo una palabra.

INSTRUCCIÓN LODS (LODSB) (LODSW)

Propósito: Cargar cadenas de un byte o palabra al acumulador.

Sintaxis:

LODS

Esta instrucción toma la cadena que se encuentre en la dirección especificada por SI, la carga al registro AL (o AX) y suma o resta 1 (según el estado de DF) a SI si la transferencia es de bytes o 2 si la transferencia es de palabras.

```
MOV SI, OFFSET VAR1
LODS
```

La primera línea carga la dirección de VAR1 en SI y la segunda línea lleva el contenido de esa localidad al registro AL.

Los comandos LODSB y LODSW se utilizan de la misma forma, el primero carga un byte y el segundo una palabra (utiliza el registro completo AX)

INSTRUCCIÓN LAHF

Propósito: Transfiere al registro AH el contenido de las banderas

Sintaxis:

LAHF

Esta instrucción es útil para verificar el estado de las banderas durante la ejecución de nuestro programa.

Las banderas quedan en el siguiente orden dentro del registro:

SF ZF À? AF À? PF À? CF

El símbolo "À?" significa que en esos bits habrá un valor indefinido.

INSTRUCCIÓN LDS

Propósito: Cargar el registro del segmento de datos

Sintaxis:

LDS destino, fuente

El operando fuente debe ser una palabra doble en memoria. La palabra asociada con la dirección más grande es transferida a DS, o sea que se toma como la dirección del segmento. La palabra asociada con la dirección menor es la dirección del desplazamiento y se deposita en el registro señalado como destino.

INSTRUCCIÓN LEA

Propósito: Carga la dirección del operando fuente.

Sintaxis:

LEA destino, fuente

El operando fuente debe estar ubicado en memoria, y se coloca su desplazamiento en el registro índice o apuntador especificado en destino.

Para ilustrar una de las facilidades que tenemos con este comando pongamos una equivalencia:

MOV SI, OFFSET VAR1

Equivale a:

LEA SI, VAR1

Es muy probable que para el programador sea más sencillo crear programas extensos utilizando este último formato.

INSTRUCCIÓN LES

Propósito: Carga el registro del segmento extra

Sintaxis:

LES destino, fuente

El operando fuente debe ser un operando en memoria de palabra doble. El contenido de la palabra con la dirección mayor se interpreta como la dirección del segmento y se coloca en ES. La palabra con la dirección menor es la dirección del desplazamiento y se coloca en el registro especificado en el parámetro destino.

INSTRUCCIÓN POP

Propósito: Recupera un dato de la pila

Sintaxis:

POP destino

Esta instrucción transfiere el último valor almacenado en la pila al operando destino, después incrementa en dos el registro SP.

Este incremento se debe a que la pila va creciendo desde la dirección mas alta de memoria del segmento hacia la mas baja, y la pila solo trabaja con palabras (2 bytes), entonces al incrementar en dos el registro SP realmente se le esta restando dos al tamaño real de la pila.

INSTRUCCIÓN POPF

Propósito: Extrae las banderas almacenadas en la pila.

Sintaxis:

POPF

Este comando transfiere bits de la palabra almacenada en la parte superior de la pila hacia el registro de banderas.

La forma de transferencia es la siguiente:

BIT	BANDERA
0	CF
2	PF
4	AF
6	ZF
7	SF
8	TF
9	IF
10	DF
11	OF

Estas localizaciones son las mismas para el comando PUSHF

Una vez hecha la transferencia se incrementa en 2 el registro SP disminuyendo así el tamaño de la pila.

INSTRUCCIÓN PUSH

Propósito: Coloca una palabra en la pila.

Sintaxis:

PUSH fuente

La instrucción PUSH decrementa en dos el valor de SP y luego transfiere el contenido del operando fuente a la nueva dirección resultante en el registro recién modificado.

El decremento en la dirección se debe a que al agregar valores a la pila esta crece de la dirección mayor a la dirección menor del segmento, por lo tanto al restarle 2 al valor del registro SP lo que hacemos es aumentar el tamaño de la pila en dos bytes, que es la única cantidad de información que puede manejar la pila en cada entrada y salida de datos.

INSTRUCCIÓN PUSHF

Propósito: Coloca el valor de las banderas en la pila

Sintaxis:

PUSHF

Este comando decrementa en 2 el valor del registro SP y luego se transfiere el contenido del registro de banderas a la pila, en la dirección indicada por SP.

Las banderas quedan almacenadas en memoria en los mismos bits indicados en el comando POPF

INSTRUCCIONES LÓGICAS Y ARITMÉTICAS

INSTRUCCIONES LÓGICAS

Son utilizadas para realizar operaciones lógicas sobre los operandos.

AND
NEG
NOT
OR
TEST
XOR

INSTRUCCIONES ARITMÉTICAS

Se usan para realizar operaciones aritméticas sobre los operandos.

ADC
ADD
DIV
IDIV
MUL
IMUL
SBB
SUB

INSTRUCCIÓN AND

Propósito: Realiza la conjunción de los operandos bit por bit.

Sintaxis:

AND destino, fuente

Con esta instrucción se lleva a cabo la operación "y" lógica de los dos operandos:

Fuente	Destino	Destino
1	1	1
1	0	0
0	1	0
0	0	0

El resultado de la operación se almacena en el operando destino.

INSTRUCCIÓN NEG

Propósito: Genera el complemento a 2

Sintaxis:

NEG destino

Esta instrucción genera el complemento a 2 del operando destino y lo almacena en este mismo operando. Por ejemplo, si AX guarda el valor de 1234H, entonces:

NEG AX

Dejaría almacenado en el registro AX el valor EDCCH.

INSTRUCCIÓN NOT

Propósito: Lleva a cabo la negación bit por bit del operando destino.

Sintaxis:

NOT destino

El resultado se guarda en el mismo operando destino.

INSTRUCCIÓN OR

Propósito: OR inclusivo lógico

Sintaxis:

OR destino, fuente

La instrucción OR lleva a cabo, bit por bit, la disyunción inclusiva lógica de los dos operandos:

Fuente	Destino	Destino
1	1	1
1	0	1
0	1	1
0	0	0

INSTRUCCIÓN TEST

Propósito: Comparar lógicamente los operandos

Sintaxis:

TEST destino, fuente

Realiza una conjunción, bit por bit, de los operandos, pero a diferencia de AND esta instrucción no coloca el resultado en el operando destino, solo tiene efecto sobre el estado de las banderas.

INSTRUCCIÓN XOR

Propósito: OR exclusivo

Sintaxis:

XOR destino, fuente

Su función es efectuar bit por bit la disyunción exclusiva lógica de los dos operandos.

Fuente	Destino	Destino
1	1	0
1	0	1
0	1	1
0	0	0

INSTRUCCIÓN ADC

Propósito: Adición con acarreo.

Sintaxis:

ADC destino, fuente

Lleva a cabo la suma de dos operandos y suma uno al resultado en caso de que la bandera CF esté activada, esto es, en caso de que exista acarreo.

El resultado se guarda en el operando destino.

INSTRUCCIÓN ADD

Propósito: Adición de los operandos.

Sintaxis:

ADD destino, fuente

Suma los dos operandos y guarda el resultado en el operando destino.

INSTRUCCIÓN DIV

Propósito: División sin signo

Sintaxis:

DIV fuente

El divisor puede ser un byte o palabra y es el operando que se le da a la instrucción.
Si el divisor es de 8 bits se toma como dividendo el registro de 16 bits AX y si el divisor es de 16 bits se tomara como dividendo el registro par DX:AX, tomando como palabra alta DX y como baja AX.

Si el divisor fue un byte el cociente se almacena en el registro AL y el residuo en AH, si fue una palabra el cociente se guarda en AX y el residuo en DX.

INSTRUCCIÓN MUL

Propósito: Multiplicación sin signo

Sintaxis:

MUL fuente

El ensamblador asume que el multiplicando será del mismo tamaño que el del multiplicador, por lo tanto multiplica el valor almacenado en el registro que se le da como operando por el que se encuentre contenido en AH si el multiplicador es de 8 bits o por AX si el multiplicador es de 16 bits.

Cuando se realiza una multiplicación con valores de 8 bits el resultado se almacena en el registro AX y cuando la multiplicación es con valores de 16 bits el resultado se almacena en el registro par DX:AX.

INSTRUCCIÓN IMUL

Propósito: Multiplicación de dos enteros con signo.

Sintaxis:

IMUL fuente

Este comando hace lo mismo que el anterior, solo que si toma en cuenta los signos de las cantidades que se multiplican.

INSTRUCCIÓN IDIV

Propósito: División con signo

Sintaxis:

IDIV fuente

Consiste básicamente en lo mismo que la instrucción DIV, solo que esta última realiza la operación con signo.

Para sus resultados utiliza los mismos registros que la instrucción DIV.

Los resultados se guardan en los mismos registros que en la instrucción MOV.

INSTRUCCIÓN SBB

Propósito: Substracción con acarreo

Sintaxis:

SBB destino, fuente

Esta instrucción resta los operandos y resta uno al resultado si CF está activada. El operando fuente siempre se resta del destino.

Este tipo de substracción se utiliza cuando se trabaja con cantidades de 32 bits.

INSTRUCCIÓN SUB

Propósito: Substracción

Sintaxis:

SUB destino, fuente

Resta el operando fuente del destino.

INSTRUCCIONES DE SALTO

Son utilizadas para transferir el flujo del proceso al operando indicado.

JMP
JA (JNBE)
JAE (JNBE)
JB (JNAE)
JBE (JNA)
JE (JZ)
JNE (JNZ)
JG (JNLE)
JGE (JNL)
JL (JNGE)
JLE (JNG)
JC
JNC
JNO
JNP (JPO)
JNS
JO
JP (JPE)
JS

INSTRUCCIONES PARA CICLOS: LOOP

Transfieren el flujo del proceso, condicional o incondicionalmente, a un destino repitiéndose esta acción hasta que el contador sea cero.

LOOP
LOOPE
LOOPNE

INSTRUCCIONES DE CONTEO

Se utilizan para decrementar o incrementar el contenido de los contadores.

DEC
INC

INSTRUCCIONES DE COMPARACIÓN

Son usadas para comparar operandos, afectan al contenido de las banderas.

CMP
CMPS (CMPSB) (CMPSW)

INSTRUCCIONES DE BANDERAS

Afectan directamente al contenido de las banderas.

CLC
CLD
CLI
CMC
STC
STD
STI

Instrucción JMP

Propósito: Salto incondicional

Sintaxis:

JMP destino

Esta instrucción se utiliza para desviar el flujo de un programa sin tomar en cuenta las condiciones actuales de las banderas ni de los datos.

Instrucción JA (JNBE)

Propósito: Brinco condicional

Sintaxis:

JA Etiqueta

Después de una comparación este comando salta si está arriba o salta si no está abajo o si no es igual.

Esto significa que el salto se realiza solo si la bandera CF esta desactivada o si la bandera ZF esta desactivada (que alguna de las dos sea igual a cero).

Instrucción JB (JNAE)

Propósito: salto condicional

Sintaxis:

JB etiqueta

Salta si está abajo o salta si no está arriba o si no es igual.

Se efectúa el salto si CF esta activada.

Instrucción JBE (JNA)

Propósito: salto condicional

Sintaxis:

JBE etiqueta

Salta si está abajo o si es igual o salta si no está arriba.

El salto se efectúa si CF est activado o si ZF est activado (que cualquiera sea igual a 1).

Instrucción JAE (JNB)

Propósito: salto condicional

Sintaxis:

JAE etiqueta

Salta si está arriba o si es igual o salta si no está abajo.

El salto se efectúa si CF esta desactivada.

Instrucción JE (JZ)

Propósito: salto condicional

Sintaxis:

JE etiqueta

Salta si es igual o salta si es cero.

El salto se realiza si ZF está activada.

Instrucción JNE (JNZ)

Propósito: salto condicional

Sintaxis:

JNE etiqueta

Salta si no es igual o salta si no es cero.

El salto se efectúa si ZF está desactivada.

Instrucción JG (JNLE)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JG etiqueta

Salta si es más grande o salta si no es menor o igual.

El salto ocurre si ZF = 0 u OF = SF.

Instrucción JGE (JNL)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JGE etiqueta

Salta si es más grande o igual o salta si no es menor que.

El salto se realiza si SF = OF

Instrucción JL (JNGE)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JL etiqueta

Salta si es menor que o salta si no es mayor o igual.

El salto se efectúa si SF es diferente a OF.

Instrucción JLE (JNG)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JLE etiqueta

Salta si es menor o igual o salta si no es más grande.

El salto se realiza si ZF = 1 o si SF es diferente a OF

Instrucción JC

Propósito: salto condicional, se toman en cuenta las banderas.

Sintaxis:

JC etiqueta

Salta si hay acarreo.

El salto se realiza si CF = 1

Instrucción JNC

Propósito: salto condicional, se toma en cuenta el estado de las banderas.

Sintaxis:

JNC etiqueta

Salta si no hay acarreo.

El salto se efectúa si CF = 0.

Instrucción JNO

Propósito: salto condicional, se toma en cuenta el estado de las banderas.

Sintaxis:

JNO etiqueta

Salta si no hay desbordamiento.

El salto se efectúa si OF = 0.

Instrucción JNP (JPO)

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JNP etiqueta

Salta si no hay paridad o salta si la paridad es non.

El salto ocurre si PF = 0.

Instrucción JNS

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JNP etiqueta

Salta si el signo esta desactivado.

El salto se efectúa si SF = 0.

Instrucción JO

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JO etiqueta

Salta si hay desbordamiento (Overflow).

El salto se realiza si OF = 1.

Instrucción JP (JPE)

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JP etiqueta

Salta si hay paridad o salta si la paridad es par.

El salto se efectúa si PF = 1.

Instrucción JS

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JS etiqueta

Salta si el signo está prendido.

El salto se efectúa si SF = 1.

Instrucción LOOP

Propósito: Generar un ciclo en el programa.

Sintaxis:

LOOP etiqueta

La instrucción loop decrementa CX en 1, y transfiere el flujo del programa a la etiqueta dada como operando si CX es diferente a 1.

Instrucción LOOPE

Propósito: Generar un ciclo en el programa considerando el estado de ZF

Sintaxis:

LOOPE etiqueta

Esta instrucción decrementa CX en 1. Si CX es diferente a cero y ZF es igual a 1, entonces el flujo del programa se transfiere a la etiqueta indicada como operando.

Instrucción LOOPNE

Propósito: Generar un ciclo en el programa, considerando el estado de ZF

Sintaxis:

LOOPNE etiqueta

Esta instrucción decrementa en uno a CX y transfiere el flujo del programa solo si ZF es diferente a 0.

INSTRUCCIÓN DEC

Propósito: Decrementar el operando

Sintaxis:

DEC destino

Esta operación resta 1 al operando destino y almacena el nuevo valor en el mismo operando.

INSTRUCCIÓN INC

Propósito: Incrementar el operando.

Sintaxis:

INC destino

La instrucción suma 1 al operando destino y guarda el resultado en el mismo operando destino.

INSTRUCCIÓN CMP

Propósito: Comparar los operandos.

Sintaxis:

CMP destino, fuente

Esta instrucción resta el operando fuente al operando destino pero sin que este almacene el resultado de la operación, solo se afecta el estado de las banderas.

INSTRUCCIÓN CMPS (CMPSB) (CMPSW)

Propósito: Comparar cadenas de un byte o palabra.

Sintaxis:

CMPS destino, fuente

Con esta instrucción la cadena de caracteres fuente se resta de la cadena destino.

Se utilizan DI como índice para el segmento extra de la cadena fuente y SI como índice de la cadena destino.

Solo se afecta el contenido de las banderas y tanto DI como SI se incrementan.

INSTRUCCIÓN CLC

Propósito: Limpiar bandera de acarreo.

Sintaxis:

CLC

Esta instrucción apaga el bit correspondiente a la bandera de acarreo, o sea, lo pone en cero.

INSTRUCCIÓN CLD

Propósito: Limpiar bandera de dirección

Sintaxis:

CLD

La instrucción CLD pone en cero el bit correspondiente a la bandera de dirección.

INSTRUCCIÓN CLI

Propósito: Limpiar bandera de interrupción

Sintaxis:

CLI

CLI pone en cero la bandera de interrupciones, deshabilitando así aquellas interrupciones enmascarables.

Una interrupción enmascarable es aquella cuyas funciones son desactivadas cuando IF = 0.

INSTRUCCIÓN CMC

Propósito: Complementar la bandera de acarreo.

Sintaxis:

CMC

Esta instrucción complementa el estado de la bandera CF, si CF = 0 la instrucción la iguala a 1, y si es 1 la instrucción la iguala a 0.

Podemos decir que únicamente "invierte" el valor de la bandera.

INSTRUCCIÓN STC

Propósito: Activar la bandera de acarreo.

Sintaxis:

STC

Esta instrucción pone la bandera CF en 1.

INSTRUCCIÓN STD

Propósito: Activar la bandera de dirección.

Sintaxis:

STD

La instrucción STD pone la bandera DF en 1.

INSTRUCCIÓN STI

Propósito: Activar la bandera de interrupción.

Sintaxis:

STI

La instrucción activa la bandera IF, esto habilita las interrupciones externas enmascarables (las que funcionan únicamente cuando IF = 1).

INTERRUPCIONES

DEFINICIÓN DE INTERRUPCIÓN

Una interrupción es una instrucción que detiene la ejecución de un programa para permitir el uso de la CPU a un proceso prioritario. Una vez concluido este último proceso se devuelve el control a la aplicación anterior.
Por ejemplo, cuando estamos trabajando con un procesador de palabras y en ese momento llega un aviso de uno de los puertos de comunicaciones, se detiene temporalmente la aplicación que estábamos utilizando para permitir el uso del procesador al manejo de la información que está llegando en ese momento. Una vez terminada la transferencia de información se reanudan las funciones normales del procesador de palabras.
Las interrupciones ocurren muy seguido, sencillamente la interrupción que actualiza la hora del día ocurre aproximadamente 18 veces por segundo.
Aunque no podemos manejar directamente esta interrupción (no podemos controlar por software las actualizaciones del reloj), es posible utilizar sus efectos en la computadora para nuestro beneficio, por ejemplo para crear un "reloj virtual" actualizado continuamente gracias al contador del reloj interno. Únicamente debemos escribir un programa que lea el valor actual del contador y lo traduzca a un formato entendible para el usuario.
Para lograr administrar todas estas interrupciones, la computadora cuenta con un espacio de memoria, llamado memoria baja, donde se almacenan las direcciones de cierta localidad de memoria donde se encuentran un juego de instrucciones que la CPU ejecutará para después regresar a la aplicación en proceso.
El manejo directo de interrupciones es una de las partes más fuertes del lenguaje ensamblador, ya que con ellas es posible controlar eficientemente todos los dispositivos internos y externos de una computadora gracias al completo control que se tiene sobre operaciones de entrada y salida.

UN POCO DE HISTORIA

IBM tomó una decisión respecto a la arquitectura de sus computadoras personales destinada a marcar un cambio notable en la historia de la tecnología. Adoptó una arquitectura abierta, esto es, utilizó componentes que estaban en el mercado en lugar de fabricar chips propietarios. Al tomar esta resolución, Intel pasó a ser la opción más clara como proveedor de procesadores y periféricos: por aquél entonces acababa de salir al mercado la línea de 16 bits 8086 y existían muchos periféricos de 8 bits de su predecesor, el 8085, tales como el controlador de interrupciones 8259, el PPI 8255, DMA 8237, la UART 8251, el timer 8253.
En los procesadores Intel de la línea X86, hay dos tipos de interrupciones: por hardware y por software. En las primeras, una señal llega a uno de los terminales de un controlador de interrupciones 8259 y éste se lo comunica al procesador mediante una señal LOW en su pin INT. El procesador interroga al 8259 cuál es la fuente de la interrupción (hay 8 posibles en un 8259) y este le muestra en el bus de datos un vector que la identifica. Por instrucciones de programa, se puede instruir al 8086 para que ignore la señal en el pin INT, por lo que estas interrupciones se denominan "enmascarables". Hay un pin adicional llamado NMI, que se comporta como una interrupción, pero imposible de bloquear (Non-Maskable-Interrupt).

TIPOS DE INTERRUPCIONES

Las interrupciones por software se comportan de igual manera que las de hardware pero en lugar de ser ejecutadas como consecuencia de una señal física, lo hacen con una instrucción.
Hay en total 256 interrupciones, de la 0 a la 7 (excepto la 5) son generadas directamente por el procesador. Las 8 a 0Fh son interrupciones por hardware primitivas de las PC. Desde la AT en adelante, se incorporó un segundo controlador de interrupciones que funciona en cascada con el primero a través de la interrupción 2 (de ahí que en la tabla siguiente se la denomine múltiplex). Las 8 interrupciones por hardware adicionales de las AT se ubican a partir del vector 70h.

Decimal	Hexa	Generada	Descripción
0	0	CPU	División por cero
1	1	CPU	Single-step
2	2	CPU	NMI
3	3	CPU	Breakpoint
4	4	CPU	Desbordamiento Aritmético
5	5	BIOS	Imprimir Pantalla
6	6	CPU	Código de operación inválido
7	7	CPU	Coprocesador no disponible
8	8	HARD	Temporizador del sistema (18,2 ticks por seg)
9	9	HARD	Teclado
10	0A	HARD	Multiplex
11	0B	HARD	IRQ3 (normalmente COM2)
12	0C	HARD	IRQ4 (normalmente COM1)
13	0D	HARD	IRQ5
14	0E	HARD	IRQ6
15	0F	HARD	IRQ7 (normalmente LPT1)
112	70	HARD	IRQ8 (reloj de tiempo real)
113	71	HARD	IRQ9
114	72	HARD	IRQ10
115	73	HARD	IRQ11
116	74	HARD	IRQ12
117	75	HARD	IRQ13 (normalmente coprocesador matemático)
118	76	HARD	IRQ14 (normalmente Disco Duro)
119	77	HARD	IRQ15

En cuanto a las interrupciones por software, están divididas entre las llamadas por el BIOS (desde la 10h a la 1Fh) y las llamadas por el DOS (desde la 20h hasta la 3Fh). Esto es sólo la versión oficial, ya que en realidad las interrupciones entre BIOS y DOS se extienden hasta la 7Fh.

CÓMO FUNCIONA UNA INTERRUPCIÓN

A partir del offset 0 del segmento 0 hay una tabla de 256 vectores de interrupción, cada uno de 4 bytes de largo (lo que significa que la tabla tiene una longitud de 1KB). Cada vector está compuesto por dos partes: offset (almacenado en la dirección más baja) y segmento (almacenado en la dirección más alta). Cuando se llama a una interrupción (no importa si es por hardware o por software), el procesador ejecuta las siguientes operaciones:

1. PUSHF (guarda las banderas en el stack)
2. CTF/DI (borra la bandera de Trap y deshabilita interrupciones)
3. CALL FAR [4 * INT#] (salta a nueva CS:IP, almacenando dirección de retorno en stack)

La expresión 4 * INT# es la forma de calcular la dirección de inicio del vector de interrupción a utilizar en el salto. Por ejemplo, el vector de la INT 21h estará en la dirección 84h. Al efectuarse el salto, la palabra almacenada en la dirección más baja del vector sustituye al contenido del registro IP (que previamente fue salvado en el stack) y la palabra almacenada en la dirección más alta sustituye al contenido del registro CS (también salvado en el stack). Por ejemplo:
Supongamos que en la posición de memoria 0000:0084 está almacenada la palabra 1A40h y en la dirección 0000:0086 está almacenada la palabra 208Ch. La próxima instrucción que se ejecute es la que está en la posición 208C:1A40 (nuevo CS:IP).
La instrucción INT 21h es la usada para efectuar llamadas a las funciones del DOS.
El final de una rutina de interrupción debe terminarse con la instrucción IRET, que recupera del stack los valores de CS, IP y Flags.

Notemos que un llamado a interrupción implica el cambio de estado automático de la bandera de habilitación de interrupciones. En pocas palabras, esto significa que al producirse una interrupción, esta bandera inhabilita futuras interrupciones. Como la instrucción IRET restablece el registro de flags al estado anterior que tenia antes de producirse la interrupción, las próximas interrupciones se habilitan en el mismo momento en que se produce el retorno desde la rutina de servicio.

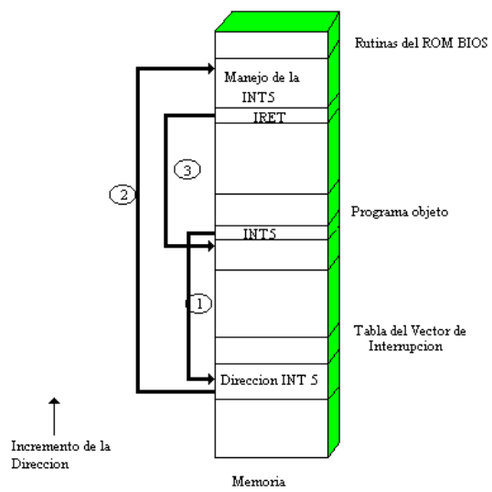
PASO DE PARÁMETROS DESDE EL PROGRAMA A LA ISR

Cuando las interrupciones son llamadas por software mediante la instrucción INT xx, por lo general se le deben pasar parámetros a la rutina de servicio de interrupción (ISR). Estos parámetros definen la tarea que debe cumplir la ISR y son pasados en los registros del procesador, lo que es una opción muy veloz.
Un ejemplo casi extremo, en donde muchos de los registros del 8086 son utilizados son algunos servicios cumplidos por la INT 13h (disco). Para tomar sólo un caso, en una operación de escritura de un sector, los parámetros se pasan de la siguiente manera:

Registro	Asignación
AH	03 (servicio de escritura de sectores)
AL	cantidad de sectores a escribir
CH	8 bits más bajos del número de cilindro
CL(bits 0-5)	número de sector
CL(bits 6 y 7)	2 bits más altos del número de cilindro
DH	número de cabeza
DL	número de unidad de disco (hard: mayor a 80h)
BX	offset del buffer de datos
ES	segmento del buffer de datos

Si bien no está escrito en ningún lado, las interrupciones utilizan el registro AH para identificar el tipo de operación que deben ejecutar. Cuando una interrupción devuelve códigos de error siempre vienen en el registro AL, AX y/o en la bandera de Carry.

ACCESO A LAS INTERRUPCIONES DEL BIOS Y DOS DESDE ROM



El ROM BIOS y DOS contienen rutinas que pueden ser usadas en los programas. Estas rutinas usualmente no son invocadas por procedimientos usuales, pero pueden ser accedidos por mecanismos de interrupción. La mayoría de los programadores típicamente organizan los programas por instrucciones CALL. El BIOS y las funciones del DOS están en forma de código objeto, y se encuentran en direcciones de memoria; en el lenguaje ensamblador hay una instrucción denominada *INT* que genera una interrupción de software en un microprocesador 80x86 que provee una solución a determinado código de interrupción. El 80x86 usa código de interrupciones como índice en una tabla para localizar la rutina a ejecutar cuando la interrupción ocurre. Esta tabla de funciones son conocidas como Tabla del Vector de Interrupción (IVT) y las funciones son conocidas como Interrupciones Rutinarias de Servicio (ISR's) El IVT esta localizado en el primer mega (1,024 Bytes) de Memoria y

contiene 256 entradas. Desde cada dirección ISR es de la forma CS:IP cada entrada en el IVT requiere de 4 Byte de almacenamiento (256 * 4 = 1,024 B). Cuando el 80x86 recibe la señal de interrupción primero empuja (PUSH) los Flags, CS y el registro IP en la pila en ese orden, luego el CPU usa el número de interrupción para indexarlo en el vector de interrupción (IVT) y luego salta a las rutinas de servicio de interrupción (ISR's) para esa interrupción. El ISR's termina con IRET (Interrup RETRY) los cual remueve los datos de la pila (POP) el Instruccion Pointer (IP), el Code Segment (CS) y Flags de la Stack (pila) por la cual retorna el control a la interrupción del programa. Ej: 1) Ejecutando la interrupción 5 ocasiona que el microprocesador grabe el siguiente estado y salta a la función de la tabla IVS en la entrada de la interrupción 5, 2) El microprocesador ejecuta el código que maneja en esa interrupción (imprimir pantalla), 3). Cuanto IRET es ejecutado se devuelve el control justo después del comando colocado en el programa objeto.

PRINCIPALES INTERRUPCIONES DEL BIOS Y DEL DOS

InT	*TIPO*	*DESCRIPCIÓN*
2	BIOS	Este tipo de interrupción no se puede evitar. Utiliza el BIOS NEM2, procedimiento NMI-INT y aparece cuando se detectan errores en la memoria sobre la tarjeta del sistema (Parity Check 1) o se tiene problemas con tarjetas que se añaden al sistema (Parity Check2)
5	BIOS	Esta interrupción se encarga de imprimir el contenido de la pantalla bajo el control del programa. EL llamado al procedimiento tipo FAR en PRINT SCREEN y la dirección 0050;0000 contiene el estado
8	BIOS	Esta rutina maneja la interrupción del temporizador proveniente del canal 0 del temporizador 8253. La rutina lleva el conteo del número de interrupciones desde que se energizó la computadora.
9	BIOS	Esta rutina es un procedimiento FAR KB-INT. La rutina continua en la dirección F000:EC32 y constituye la interrupción del teclado. La INT 16h es la rutina de E/S del teclado y es más flexible.
E	BIOS	Este procedimiento de tipo FAR, DISK-INT maneja la interrupción del diskette.
F	DOS	Activa la misma llamada que Tipo 4.
10	BIOS	El conjunto de rutinas asociado con este procedimiento NEAR VIDEO-E/S, constituye la interfaz con el TRC.
11	BIOS	El procedimiento proporciona él numero de puertos para la impresora, adaptadores de juegos, interfaces RS-232C, número de unidades de Diskettes, modos de video y tamaños del RAM
12	BIOS	Proporciona el tamaño de la memoria
13	BIOS	Llama a varias rutinas para llevar operaciones de entrada y salidas del disco.
14	BIOS	Este procedimiento permite al usuario la entrada y salida de datos desde el puerto de comunicaciones RS-232C.
15	BIOS	Interrupción empleada para controlar las operaciones de E/S en casettes.
16	BIOS	Esta interrupción utiliza a AX para leer el teclado.
17	BIOS	Esta rutina proporciona la comunicación con la impresora. Los parámetros necesarios son colocados en los registros AX y DX.
18	BIOS	Esta interrupción llama al casette de basic.

19	BIOS	La rutina asociada con esta interrupción, lee el sector 1 de la pista 0 del disco en la unidad A, a la que le transfiere el control
1A	BIOS	Esta rutina permite seleccionar o leer el contenido del reloj que lleva la hora. El registro CX contiene la palabra más significativa del conteo mientras que en el DX se encuentra la menos significativa.
1B	DOS	Esta interrupción se presenta cada vez que se genera una interrupción proveniente del teclado.
1C	BIOS	Esta interrupción provoca la ejecución IRET.
1D	BIOS	Esta tabla de bytes y rutinas necesarias para establecer varios parámetros para gráficos.
1E	DOS	Tabla de Diskette.
1F	DOS	Tabla de gráficos.
20	DOS	Esta interrupción es generada por DOS para salirse un programa, es la primera dirección del área correspondiente al segmento prefijo del programa.
21	DOS	Esta interrupción consta de varias opciones, una de ellas es solicitar funciones.
22	DOS	Cuando termina la ejecución de un programa esta interrupción transfiere el control a la dirección especificada por el vector de interrupción. Esta interrupción nunca debe generarse de manera directa.
23	DOS	Esta interrupción es generada como respuesta a un CRTL BREAK.
24	DOS	Esta interrupción se llama cada vez que ocurre un error crítico dentro de DOS, como puede ser un error de disco.
25	DOS	Esta interrupción transfiere el control, para lectura, al manejador del dispositivo (driver).
26	DOS	Esta interrupción transfiere el control, para escritura, a manejador del dispositivo.
27	DOS	Este vector es empleado, para que al término de un programa este permanezca residente en la memoria del sistema una vez que DOS toma de nuevo el control.
2F	DOS	Esta interrupción define una interfaz general entre dos procesos, el número especificado en AH indica a cada manejador y AL contiene la función del manejador.

TIPOS DE INTERRUPCIONES

- Interrupciones internas de hardware
- Interrupciones externas de hardware
- Interrupciones de software

Las Interrupciones más usuales son:

Int 21H (interrupción del DOS)
Múltiples llamadas a funciones del DOS.

Int 10H (interrupción del BIOS)
Entrada/salida de vídeo.

Int 16H (Interrupción del BIOS)
Entrada/salida de teclado.

Int 17H (Interrupción del BIOS)
Entrada/salida de la impresora.

INTERRUPCIONES INTERNAS DE HARDWARE

Las Interrupciones Internas son generadas por ciertos eventos que surgen durante la ejecución de un programa.
Este tipo de Interrupciones son manejadas, en su totalidad, por el hardware y no es posible modificarlas.

INTERRUPCIONES EXTERNAS DE HARDWARE

Las Interrupciones Externas las generan los dispositivos periféricos, como pueden ser: teclado, impresoras, tarjetas de comunicaciones, etc. También son generadas por los coprocesadores.
No es posible desactivar a las Interrupciones externas.
Estas Interrupciones no son enviadas directamente a la CPU, sino que se mandan a un circuito integrado cuya función es exclusivamente manejar este tipo de Interrupciones. El circuito, llamado PIC 8259A, si es controlado por la CPU utilizando para tal control una serie de vías de comunicación llamadas puertos.

INTERRUPCIONES DE SOFTWARE

Las Interrupciones de software pueden ser activadas directamente por el ensamblador invocando al número de interrupción deseada con la instrucción INT.
El uso de las Interrupciones nos ayuda en la creación de programas; utilizándolas nuestros programas son más cortos, es más fácil entenderlos y usualmente tienen un mejor desempeño debido en gran parte a su menor tamaño.
Este tipo de interrupciones podemos separarlas en dos categorías: las Interrupciones del sistema operativo DOS y las Interrupciones del BIOS.
La diferencia entre ambas es que las interrupciones del sistema operativo son más fáciles de usar pero también son más lentas ya que estas interrupciones hacen uso del BIOS para lograr su cometido, en cambio las interrupciones del BIOS son mucho más rápidas pero tienen la desventaja que, como son parte del hardware, son muy específicas y pueden variar dependiendo incluso de la marca del fabricante del circuito.
La elección del tipo de interrupción a utilizar dependerá únicamente de las características que le quiera dar a su programa: velocidad (utilizando las del BIOS) o portabilidad (utilizando las del DOS).

INTERRUPCIONES DE DOS

INTERRUPCIÓN 21H

Propósito: Llamar a diversas funciones del DOS.

Sintaxis:

Int 21H

Nota: Cuando trabajamos en MASM es necesario especificar que el valor que estamos utilizando es hexadecimal.

Esta interrupción tiene varias funciones, para accesar a cada una de ellas es necesario que el registro AH se encuentre el número de función que se requiera al momento de llamar a la interrupción.

Funciones para desplegar información al vídeo

- 02H Exhibe salida
- 09H Impresión de cadena (vídeo)
- 40H Escritura en dispositivo/Archivo

Funciones para leer información del teclado

- 01H Entrada desde teclado
- 0AH Entrada desde teclado usando buffer
- 3FH Lectura desde dispositivo/archivo

Funciones para trabajar con archivos

En esta sección únicamente se expone la tarea específica de cada función, para una referencia acerca de los conceptos empleados refiérase al tema titulado: "Introducción al manejo de archivos".

Método FCB

0FH Abrir archivo
14H Lectura secuencial
15H Escritura secuencial
16H Crear archivo
21H Lectura aleatoria
22H Escritura aleatoria

Handles

3CH Crear archivo
3DH Abrir archivo
3EH Cierra manejador de archivo
3FH Lectura desde archivo/dispositivo
40H Escritura en archivo/dispositivo
42H Mover apuntador de lectura/escritura en archivo

FUNCIONES PARA DESPLEGAR INFORMACIÓN AL VÍDEO

Función 02H

Uso: Despliega un carácter a la pantalla.

Registros de llamada:

AH = 02H
DL = Valor del carácter a desplegar.

Registros de retorno:

Ninguno

Esta función nos despliega el carácter cuyo código hexadecimal corresponde al valor almacenado en el registro DL, no se modifica ningún registro al utilizar este comando.

Es recomendado el uso de la función 40H de la misma interrupción en lugar de esta función.

Función 09H

Uso: Despliega una cadena de caracteres en la pantalla.

Registros de llamada:

AH = 09H
DS:DX = Dirección de inicio de una cadena de caracteres

Registros de retorno:

Ninguno.

Esta función despliega los caracteres, uno a uno, desde la dirección indicada en el registro DS: DX hasta encontrar un carácter $, que es interpretado como el final de la cadena.

Se recomienda utilizar la función 40H en lugar de esta función.

Función 40H

Uso: Escribir a un dispositivo o a un archivo.

Registros de llamada:

AH = 40H
BX = Vía de comunicación
CX = Cantidad de bytes a escribir
DS:DX = Dirección del inicio de los datos a escribir

Registros de retorno:

CF = 0 si no hubo error
AX = Número de bytes escritos
CF = 1 si hubo error
AX = Código de error

El uso de esta función para desplegar información en pantalla se realiza dándole al registro BX el valor de 1 que es el valor preasignado al vídeo por el sistema operativo MS-DOS.

FUNCIONES PARA LEER INFORMACIÓN DEL TECLADO

Función 01H

Uso: Leer un carácter del teclado y desplegarlo.

Registros de llamada:

AH = 01H

Registros de retorno:

AL = Carácter leído

Con esta función es muy sencillo leer un carácter del teclado, el código hexadecimal del carácter leído se guarda en el registro AL.

Función 0AH

Uso: Leer caracteres del teclado y almacenarlos en un buffer.

Registros de llamada:

AH = 0AH
DS:DX = Dirección del área de almacenamiento
BYTE 0 = Cantidad de bytes en el área
BYTE 1 = Cantidad de bytes leídos
desde BYTE 2 hasta BYTE 0 + 2 = caracteres leídos

Registros de retorno:

Ninguno

Los caracteres son leídos y almacenados en un espacio predefinido de memoria. La estructura de este espacio le indica que en el primer byte del mismo se indican cuantos caracteres serán leídos. En el segundo byte se almacena el número de caracteres que ya se leyeron, y del tercer byte en adelante se escriben los caracteres leídos.

Cuando se han almacenado todos los caracteres indicados menos uno la bocina suena y cualquier carácter adicional es ignorado. Para terminar la captura de la cadena es necesario darle [ENTER].

Función 3FH

Uso: Leer información de un dispositivo o archivo.

Registros de llamada:

AH = 3FH
BX = Número asignado al dispositivo
CX = Número de bytes a procesar
DS:DX = Dirección del área de almacenamiento

Registros de retorno:

CF = 0 si no hay error y AX = número de bytes leídos.
CF = 1 si hay error y AX contendrá el código del error.

FUNCIONES PARA TRABAJAR CON ARCHIVOS

Función 0FH

Uso: Abrir archivo FCB

Registros de llamada:

AH = 0FH
DS:DX = Apuntador a un FCB

Registros de retorno:

AL = 00H si no hubo problema, de lo contrario regresa 0FFH

Función 14H

Uso: Leer secuencialmente un archivo FCB.

Registros de llamada:

AH = 14H
DS:DX = Apuntador a un FCB ya abierto.

Registros de retorno:

AL = 0 si no hubo errores, de lo contrario se regresara el código correspondiente de error: 1 error al final del archivo, 2 error en la estructura del FCB y 3 error de lectura parcial.

Esta función lo que hace es que lee el siguiente bloque de información a partir de la dirección dada por DS:DX, y actualiza este registro.

Función 15H

Uso: Escribir secuencialmente a un archivo FCB

Registros de llamada:

AH = 15H
DS:DX = Apuntador a un FCB ya abierto

Registros de retorno:

AL = 00H si no hubo errores, de lo contrario contendrá el código del error:
1 disco lleno o archivo de solo lectura, 2 error en la formación o especificación del FCB.

La función 15H después de escribir el registro al bloque actual actualiza el FCB.

Función 16H

Uso: Crear un archivo FCB.
Registros de llamada:

AH = 16H
DS:DX = Apuntador a un FCB ya abierto.

Registros de retorno:

AL = 00H si no hubo errores, de lo contrario contendrá el valor 0FFH

Se basa en la información proveída en un FCB para crear un archivo en el disco.

Función 21H

Uso: Leer en forma aleatoria un archivo FCB.

Registros de llamada:

AH = 21H
DS:DX = Apuntador a un FCB ya abierto.

Registros de retorno:

A = 00H si no hubo error, de lo contrario AH contendrá el código del error: 1 si es fin de archivo, 2 si existe error de especificación de FCB y 3 si se leyó un registro parcial o el apuntador del archivo se encuentra al final del mismo.

Esta función lee el registro especificado por los campos del bloque actual y registro actual de un FCB abierto y coloca la información en el DTA (área de transferencia de disco o Disk Transfer Area).

Función 22H

Uso: Escribir en forma aleatoria en un archivo FCB.

Registros de llamada:

AH = 22H
DS:DX = Apuntador a un FCB abierto.

Registros de retorno:

AL = 00H si no hubo error, de lo contrario contendrá el código del error: 1 si el disco está lleno o es archivo de solo lectura y 2 si hay error en la especificación de FCB.

Escribe el registro especificado por los campos del bloque actual y registro actual de un FCB abierto. Escribe dicha información a partir del contenido del DTA (área de transferencia de disco).

Función 3CH

Uso: Crear un archivo si no existe o dejarlo en longitud 0 si existe. (Handle)

Registros de llamada:

AH = 3CH
CH = Atributo de archivo
DS:DX = Apuntador a una especificación ASCIIZ

Registros de retorno:

CF = 0 y AX el número asignado al handle si no hay error, en caso de haberlo CF ser 1 y AX contendrá el código de error: 3 ruta no encontrada, 4 no hay handles disponibles para asignar y 5 acceso negado.

Esta función sustituye a la 16H. El nombre del archivo es especificado en una cadena ASCIIZ, la cual tiene como característica la de ser una cadena de bytes convencional terminada con un carácter 0.

El archivo creado contendrá los atributos definidos en el registro CX en la siguiente forma:

Valor	Atributos
00H	Normal
02H	Escondido
04H	Sistema
06H	Escondido y de sistema

El archivo se crea con los permisos de lectura y escritura. No es posible crear directorios utilizando esta función.

Función 3DH

Uso: Abre un archivo y regrese un handle

Registros de llamada:

AH = 3DH
AL = modo de acceso
DS:DX = Apuntador a una especificación ASCIIZ

Registros de retorno:

CF = 0 y AX = número de handle si no hay errores, de lo contrario CF = 1 y AX = código de error: 01H si no es válida la función, 02H si no se encontró el archivo, 03H si no se encontró la ruta, 04H si no hay handles disponibles, 05H en caso de acceso negado, y 0CH si el código de acceso no es válido.

El handle regresado es de 16 bits.

El código de acceso se especifica en la siguiente forma:

BITS
7 6 5 4 3 2 1

. . . . 0 0 0 Solo lectura
. . . . 0 0 1 Solo escritura
. . . . 0 1 0 Lectura/Escritura
. . . X . . . RESERVADO

Función 3EH

Uso: Cerrar archivo (Handle).

Registros de llamada:

AH = 3EH
BX = Handle asignado

Registros de retorno:

CF = 0 si no hubo errores, en caso contrario CF ser 1 y AX contendrá el código de error: 06H si el handle es inválido.

Esta función actualiza el archivo y libera o deja disponible el handle que estaba utilizando.

Función 3FH

Uso: Leer de un archivo abierto una cantidad definida de bytes y los almacena en un buffer específico.

Registros de llamada:

AH = 3FH
BX = Handle asignado
CX = Cantidad de bytes a leer
DS:DX = Apuntador a una área de trabajo.

Registros de retorno:

CF = 0 y AX = número de bytes leídos si no hubo error, en caso contrario CF = 1 y AX = código de error: 05H si acceso negado y 06H si no es válido el handle.

Función 40H

Uso: Escribe a un archivo ya abierto una cierta cantidad de bytes a partir del buffer designado.

Registros de llamada:

AH = 40H
BX = Handle asignado
CX = Cantidad de bytes a escribir.
DS:DX = Apuntador al buffer de datos.

Registros de retorno:

CF = 0 y AX = número de bytes escritos si no hay errores, en caso de existir CF = 1 y AX = código del error: 05H si el acceso es negado y 06H si el handle es inválido.

Función 42H

Uso: Mover apuntador al archivo (Handle)

Registros de llamada:

AH = 42H
AL = método utilizado
BX = Handle asignado
CX = La parte m s significativa del offset
DX = La parte menos significativa del offset

Registros de retorno:

CF = 0 y DX:AX = la nueva posición del apuntador. En caso de error CF ser 1 y AX = código de error: 01H si la función no es v lida y 06H si el handle no es válido.

El método utilizado se configura como sigue:

Valor de AL	Método
00H	A partir del principio del archivo
01H	A partir de la posición actual
02H	A partir del final del archivo

INTERRUPCIONES DE BIOS

Interrupción 10H

Propósito: Llamar a diversas funciones de vídeo del BIOS.

Sintaxis:

Int 10H

Esta interrupción tiene diversas funciones, todas ellas nos sirven para controlar la entrada y salida de vídeo, la forma de acceso a cada una de las opciones es por medio del registro AH.

Funciones comunes de la interrupción 10H:

02H Selección de posición del cursor
09H Escribe atributo y carácter en el cursor
0AH Escribe carácter en la posición del cursor
0EH Escritura de caracteres en modo alfanumérico

Función 02H

Uso: Posiciona el cursor en la pantalla dentro de las coordenadas válidas de texto.

Registros de llamada:

AH = 02H
BH = Página de vídeo en la que se posicionará el cursor.
DH = Fila
DL = Columna

Registros de retorno:

Ninguno.

Las posiciones de localización del cursor son definidas por coordenadas iniciando en 0,0, que corresponde a la esquina superior izquierda hasta 79,24 correspondientes a la esquina inferior derecha. Tenemos entonces que los valores que pueden tomar los registros DH y DL en modo de texto de 80 x 25 son de 0 hasta 24 y de 0 hasta 79 respectivamente.

Función 09H

Uso: Desplegar un carácter un determinado número de veces con un atributo definido empezando en la posición actual del cursor.

Registros de llamada:

AH = 09H
AL = Carácter a desplegar
BH = Página de vídeo en donde se desplegará
BL = Atributo a usar
Número de repeticiones.

Registros de retorno:

Ninguno

Esta función despliega un carácter el número de veces especificado en CX pero sin cambiar la posición del cursor en la pantalla.

Función 0AH

Uso: Desplegar un carácter en la posición actual del cursor.

Registros de llamada:

AH = 0AH
AL = Carácter a desplegar
BH = Página en donde desplegar
BL = Color a usar (sólo en gráficos).
CX = Número de repeticiones

Registros de retorno:

Ninguno.

La única diferencia entre esta función y la anterior es que esta no permite modificar los atributos, simplemente usa los atributos actuales.
Tampoco se altera la posición del cursor con esta función.

Función 0EH

Uso: Desplegar un carácter en la pantalla actualizando la posición del cursor.

Registros de llamada:

AH = 0EH
AL = Carácter a desplegar
BH = Página donde se desplegara el carácter
BL = Color a usar (solo en gráficos)

Registros de retorno:

Ninguno

Interrupción 16H

Propósito: Manejar la entrada/salida del teclado.

Sintaxis:
Int 16H

Veremos dos opciones de la interrupción 16H. Estas opciones, al igual que las de otras Interrupciones, son llamadas utilizando el registro AH.

Funciones de la interrupción 16H

00H Lee un carácter de teclado
01H Lee estado del teclado

Función 00H

Uso: Leer un carácter del teclado.

Registros de llamada:

AH = 00H

Registros de retorno:

AH = código de barrido (scan code) del teclado
AL = Valor ASCII del carácter.

Cuando se utiliza esta interrupción se detiene la ejecución del programa hasta que se introduzca un carácter desde el teclado, si la tecla presionada es un carácter ASCII su valor será guardado en el registro AH, de lo contrario el código de barrido será guardado en AL y AH contendrá el valor 00H.

El código de barrido fue creado para manejar las teclas que no tienen una representación ASCII como [ALT], [CONTROL], las teclas de función, etc.

Función 01H

Uso: Leer estado del teclado.

Registros de llamada:

AH = 01H

Registros de retorno:

Si la bandera de cero, ZF, está apagada significa que hay información en el buffer, si se encuentra prendida es que no hay teclas pendientes.

En caso de existir información el registro AH contendrá el código de la tecla guardada en el buffer.

Interrupción 17H

Propósito: Manejar la entrada/salida de la impresora.

Sintaxis:

Int 17H

Esta interrupción es utilizada para escribir caracteres a la impresora, inicializarla y leer su estado.

Funciones de la interrupción 17H

00H Imprime un carácter ASCII
01H Inicializa la impresora
02H Proporciona el estado de la impresora

Función 00H

Uso: Escribir un carácter a la impresora.

Registros de llamada:

AH = 00H
AL = Carácter a imprimir
DX = Puerto a utilizar

Registros de retorno:

AH = Estado de la impresora.

El puerto a utilizar, definido en DX, se especifica así: LPT1 = 0, LPT2 = 1, LPT3 = 2 ...

El estado de la impresora se codifica bit por bit como sigue:

BIT	1/0	SIGNIFICADO
0	1	Se agotó el tiempo de espera
1	-	
2	-	
3	1	Error de entrada/salida
4	1	Impresora seleccionada
5	1	Papel agotado
6	1	Reconocimiento de comunicación
7	1	La impresora se encuentra libre

Los bits 1 y 2 no son relevantes.
La mayoría de los BIOS únicamente soportan 3 puertos paralelos aunque existen algunos que soportan 4.

Función 01H

Uso: Inicializar un puerto de impresión.

Registros de llamada:

AH = 01H
DX = Puerto a utilizar

Registros de retorno:

AH = Status de la impresora

El puerto a utilizar, definido en DX, se especifica así: LPT1 = 0, LPT2 = 1, etc.

El estado de la impresora se codifica bit por bit como sigue:

BIT	1/0	SIGNIFICADO
0	1	Se agotó el tiempo de espera
1	-	
2	-	
3	1	Error de entrada/salida
4	1	Impresora seleccionada
5	1	Papel agotado
6	1	Reconocimiento de comunicación
7	1	La impresora se encuentra libre

Los bits 1 y 2 no son relevantes.

La mayoría de los BIOS únicamente soportan 3 puertos paralelos aunque existen algunos que soportan 4.

Función 02H

Uso: Obtener el estado de la impresora.

Registros de llamada:

AH = 01H
DX = Puerto a utilizar

Registros de retorno:

AH = Status de la impresora.

El puerto a utilizar, definido en DX, se especifica así: LPT1 = 0, LPT2 = 1, etc.
El estado de la impresora se codifica bit por bit como sigue:

BIT	1/0	SIGNIFICADO
0	1	Se agotó el tiempo de espera
1	-	
2	-	
3	1	Error de entrada/salida
4	1	Impresora seleccionada
5	1	Papel agotado
6	1	Reconocimiento de comunicación
7	1	La impresora se encuentra libre

Los bits 1 y 2 no son relevantes.
La mayoría de los BIOS únicamente soportan 3 puertos paralelos aunque existen algunos que soportan 4.

PROCEDIMIENTOS Y MACROS

PROCEDIMIENTOS

Definición de procedimiento

Un procedimiento es un conjunto de instrucciones a los que podemos dirigir el flujo de nuestro programa, y una vez terminada la ejecución de dichas instrucciones se devuelve el control a la siguiente línea a procesar del código que mando llamar al procedimiento.
Los procedimientos nos ayudan a crear programas legibles y fáciles de modificar.
Al momento de invocar a un procedimiento se guarda en la pila la dirección de la siguiente instrucción del programa para que, una vez transferido el flujo del programa y terminado el procedimiento, se pueda regresar a la línea siguiente del programa original (el que llamó al procedimiento).

Sintaxis de un procedimiento

Existen dos tipos de procedimientos, los intrasegmentos, que se encuentran en el mismo segmento de instrucciones y los intersegmentos que pueden ser almacenados en diferentes segmentos de memoria.
Cuando se utilizan los procedimientos intrasegmentos se almacena en la pila el valor de IP y cuando se utilizan los intersegmentos se almacena el valor CS:IP
Para desviar el flujo a un procedimiento (llamarlo) se utiliza la directiva:

CALL NombreDelProcedimiento

Las partes que componen a un procedimiento son:

- Declaración del procedimiento
- Código del procedimiento
- Directiva de regreso
- Terminación del procedimiento

Por ejemplo, si queremos una rutina que nos sume dos bytes, almacenados en AH y AL cada uno y guardar la suma en el registro BX:

```
 Suma Proc Near  ;Declaración del procedimiento
   Mov Bx, 0     ;Contenido del procedimiento
   Mov Bl, Ah
   Mov Ah, 00
   Add Bx, Ax
   Ret           ;Directiva de regreso
 Suma Endp       ;Declaración de final del procedimiento
```

En la declaración la primera palabra, Suma, corresponde al nombre de nuestro procedimiento, Proc lo declara como tal y la palabra Near le indica al MASM que el procedimiento es intrasegmento. La directiva Ret carga la dirección IP almacenada en la pila para regresar al programa original, por último, la directiva Suma Endp indica el final del procedimiento.
Para declarar un procedimiento intersegmento sustituimos la palabra Near por la palabra FAR.
El llamado de este procedimiento se realiza de la siguiente forma:

Call Suma

Las macros ofrecen una mayor flexibilidad en la programación comparadas con los procedimientos, pero no por ello se dejarán de utilizar estos últimos.

MACROS

Cuando un conjunto de instrucciones en ensamblador aparece frecuentemente repetido a lo largo de un listado, es conveniente agruparlas bajo un nombre simbólico que las sustituirá en aquellos puntos donde aparezcan. Esta es la misión de las macros; por el hecho de soportarlas el ensamblador eleva su categoría a la de macroensamblador, al ser las macros una herramienta muy cotizada por los programadores.
No conviene confundir las macros con subrutinas: en éstas últimas, el conjunto de instrucciones aparece una sola vez en todo el programa y luego se invoca con CALL. Sin embargo, cada vez que se referencia a una macro, el código que ésta representa se *expande* en el programa definitivo, duplicándose tantas veces como se use la macro. Por ello, aquellas tareas que puedan ser realizadas con subrutinas siempre será más conveniente realizarlas con las mismas, con objeto de economizar memoria. Es cierto que las macros son algo más rápidas que las subrutinas (se ahorra un CALL y un RET) pero la diferencia es tan mínima que en la práctica es despreciable en el 99,99% de los casos. Por ello, es absurdo e irracional realizar ciertas tareas con macros que pueden ser desarrolladas mucho más eficientemente con subrutinas: es una pena que en muchos manuales de ensamblador aún se hable de macros para realizar operaciones sobre cadenas de caracteres, que generarían programas gigantescos con menos de un 1% de velocidad adicional.

DEFINICIÓN Y BORRADO DE LAS MACROS

La macro se define por medio de la directiva MACRO. Es necesario definir la macro **antes** de utilizarla. Una macro puede llamar a otra. Con frecuencia, las macros se colocan juntas en un fichero independiente y luego se mezclan en el programa principal con la directiva INCLUDE:

```
IF1
  INCLUDE fichero.ext
ENDIF
```

La sentencia IF1 asegura que el ensamblador lea el fichero fuente de las macros sólo en la primera pasada, para acelerar el ensamblaje y evitar que aparezcan en el listado (generado en la segunda fase). Conviene hacer hincapié en que la definición de la macro no consume memoria, por lo que en la práctica es indiferente declarar cientos que ninguna macro:

```
nombre_simbólico MACRO [parámetros]
 ...
 ...            ; instrucciones de la macro
ENDM
```

El nombre simbólico es el que permitirá en adelante hacer referencia a la macro, y se construye casi con las mismas reglas que los nombres de las variables y demás símbolos. La macro puede contener parámetros de manera opcional. A continuación vienen las instrucciones que engloba y, finalmente, la directiva ENDM señala el final de la macro. No se debe repetir el nombre simbólico junto a la directiva ENDM, ello provocaría un error un tanto curioso y extraño por parte del ensamblador (algo así como «Fin del fichero fuente inesperado, falta directiva END»), al menos con MASM 5.0 y TASM 2.0.
En realidad, y a diferencia de lo que sucede con los demás símbolos, el nombre de una macro puede coincidir con el de una instrucción máquina o una directiva del ensamblador: a partir de ese momento, la instrucción o directiva *machacada* pierde su significado original. El ensamblador dará además un aviso de advertencia si se emplea una instrucción o directiva como nombre de macro, aunque tolerará la operación. Normalmente se les asignará nombres normales, como a las variables. Sin embargo, si alguna vez se redefiniera una instrucción máquina o directiva, para restaurar el significado original del símbolo, la macro puede ser borrada -o simplemente porque ya no va a ser usada a partir de cierto punto del listado, y así ya no consumirá espacio en las tablas de macros que mantiene en memoria el ensamblador al ensamblar-. No es necesario borrar las macros antes de redefinirlas. Para borrarlas, la sintaxis es la siguiente:

```
PURGE  nombre_simbólico[,nombre_simbólico,...]
```

EJEMPLO DE UNA MACRO SENCILLA

Desde el 286 existe una instrucción muy cómoda que introduce en la pila 8 registros, y otra

que los saca (PUSHA y POPA). Quien esté acostumbrado a emplearlas, puede crear unas macros que simulen estas instrucciones en los 8086:

```
SUPERPUSH   MACRO
            PUSH   AX
            PUSH   CX
            PUSH   DX
            PUSH   BX
            PUSH   SP
            PUSH   BP
            PUSH   SI
            PUSH   DI
            ENDM
```

La creación de SUPERPOP es análoga, sacando los registros en orden inverso. El orden elegido no es por capricho y se corresponde con el de la instrucción PUSHA original, para compatibilizar. A partir de la definición de esta macro, tenemos a nuestra disposición una nueva instrucción máquina (SUPERPUSH) que puede ser usada con libertad dentro de los programas.

PARÁMETROS FORMALES Y PARÁMETROS ACTUALES

Para quien no haya tenido relación previa con algún lenguaje estructurado de alto nivel, se hará un breve comentario acerca de lo que son los parámetros formales y actuales en una macro, similares a los de los procedimientos de los lenguajes de alto nivel.
Cuando se llama a una macro se le pueden pasar opcionalmente un cierto número de parámetros de cierto tipo. Estos parámetros se denominan *parámetros actuales*. En la definición de la macro, dichos parámetros aparecen asociados a ciertos nombres arbitrarios, cuya única misión es permitir distinguir unos parámetros de otros e indicar en qué orden son entregados: son los *parámetros formales*. Cuando el ensamblador expanda la macro al ensamblar, los parámetros formales serán sustituidos por sus correspondientes parámetros actuales. Considere el siguiente ejemplo:

```
SUMAR       MACRO  a,b,total
            PUSH   AX
            MOV    AX,a
            ADD    AX,b
            MOV    total,AX
            POP    AX
            ENDM
            ....
            SUMAR positivos, negativos, total
```

En el ejemplo, «a», «b» y «total» son los parámetros formales y «positivos», «negativos» y «total» son los parámetros actuales. Tanto «a» como «b» pueden ser variables, etiquetas, etc. en otro punto del programa; sin embargo, dentro de la macro, se comportan de manera independiente. El parámetro formal «total» ha coincidido en el ejemplo y por casualidad con su correspondiente actual. El código que genera el ensamblador al expandir la macro será el siguiente:

```
PUSH   AX
MOV    AX,positivos
ADD    AX,negativos
MOV    total,AX
POP    AX
```

Las instrucciones PUSH y POP sirven para no alterar el valor de AX y conseguir que la macro se comporte como una *caja negra*; no es necesario que esto sea así pero es una buena costumbre de programación para evitar que los programas hagan cosas raras. En general, las macros de este tipo no deberían alterar los registros y, si los cambian, hay que tener muy claro cuáles.
Si se indican más parámetros de los que una macro necesita, se ignorarán los restantes. En cambio, si faltan, el MASM asumirá que son nulos (0) y dará un mensaje de advertencia, el

TASM es algo más rígido y podría dar un error. En general, se trata de situaciones atípicas que deben ser evitadas.
También puede darse el caso de que no sea posible expandir la macro.
En el ejemplo siguiente, no hubiera sido posible ejecutar SUMAR AX,BX,DL porque DL es de 8 bits y la instrucción MOV DL,AX sería ilegal.

ETIQUETAS DENTRO DE MACROS. VARIABLES LOCALES

Son necesarias normalmente para los saltos condicionales que contengan las macros más complejas. Si se pone una etiqueta a donde saltar, la macro sólo podría ser empleada una vez en todo el programa para evitar que dicha etiqueta aparezca duplicada. La solución está en emplear la directiva LOCAL que ha de ir colocada justo después de la directiva MACRO:

```
MINIMO     MACRO  dato1, dato2, resultado
           LOCAL  ya_esta
           MOV    AX,dato1
           CMP    AX,dato2      ; ¿es dato1 el menor?
           JB     ya_esta       ; sí
           MOV    AX,dato2      ; no, es dato2
ya_esta:   MOV    resultado,AX
           ENDM
```

En el ejemplo, al invocar la macro dos veces el ensamblador no generará la etiqueta «ya_esta» sino las etiquetas ??0000, ??0001, ... y así sucesivamente. La directiva LOCAL no sólo es útil para los saltos condicionales en las macros, también permite declarar variables internas a los mismos. Se puede indicar un número casi indefinido de etiquetas con la directiva LOCAL, separándolas por comas.

OPERADORES DE MACROS

* Operador ;;

Indica que lo que viene a continuación es un comentario que no debe aparecer al expansionar la macro. Cuando al ensamblar se genera un listado del programa, las macros suelen aparecer expandidas en los puntos en que se invocan; sin embargo sólo aparecerán los comentarios normales que comiencen por (;). Los comentarios relacionados con el funcionamiento interno de la macro deberían ir con (;;), los relativos al uso y sintaxis de la misma con (;). Esto es además conveniente porque durante el ensamblaje son mantenidos en memoria los comentarios de macros (no los del resto del programa) que comienzan por (;), y no conviene desperdiciar memoria...

* Operador &

Utilizado para concatenar texto o símbolos. Es necesario para lograr que el ensamblador sustituya un parámetro dentro de una cadena de caracteres o como parte de un símbolo:

```
SALUDO     MACRO  c
           MOV    AL,"&c"
etiqueta&c: CALL  imprimir
           ENDM
```

Al ejecutar SALUDO A se producirá la siguiente expansión:

```
           MOV    AL,"A"
etiquetaA:  CALL  imprimir
```

Si no se hubiera colocado el & se hubiera expandido como MOV AL,"c"

Cuando se utilizan estructuras repetitivas REPT, IRP o IRPC (que se verán más adelante) existe un problema adicional al intentar crear etiquetas, ya que el ensamblador *se come* un & al hacer la primera sustitución, generando la misma etiqueta a menos que se duplique el operador &:

```
MEMORIA    MACRO  x
           IRP    i, <1, 2>
```

```
x&i         DB  i
        ENDM
        ENDM
```

Si se invoca MEMORIA ET se produce el error de "etiqueta ETi repetida", que se puede salvar añadiendo tantos '&' como niveles de anidamiento halla en las estructuras repetitivas empleadas, como se ejemplifica a continuación:

```
MEMORIA    MACRO  x
        IRP   i, <1, 2>
x&&i        DB  i
        ENDM
        ENDM
```

Lo que con MEMORIA ET generará correctamente las líneas:

```
ET1     DB 1
ET2     DB 2
```

* Operador ! o <>

Empleado para indicar que el carácter que viene a continuación debe ser interpretado literalmente y no como un símbolo. Por ello, !; es equivalente a <;>.

* Operador %

Convierte la expresión que le sigue -generalmente un símbolo- a un número; la expresión debe ser una constante (no relocalizable). Sólo se emplea en los argumentos de macros. Dada la macro siguiente:

```
PSUM       MACRO  mensaje, suma
        %OUT   * mensaje, suma *
        ENDM
```

(Evidentemente, el % que precede a OUT forma parte de la directiva y no se trata del % operador que estamos tratando)

Supuesta la existencia de estos símbolos:

```
SIM1    EQU   120
SIM2    EQU   500
```

Invocando la macro con las siguientes condiciones:

```
PSUM       < SIM1 + SIM2 = >, (SIM1+SIM2)
```

Se produce la siguiente expansión:

```
%OUT * SIM1 + SIM2 = (SIM1+SIM2) *
```

Sin embargo, invocando la macro de la siguiente manera (con %):

```
PSUM < SIM1 + SIM2 = >, %(SIM1+SIM2)
```

Se produce la expansión deseada:

```
%OUT * SIM1 + SIM2 = 620 *
```

DIRECTIVAS ÚTILES PARA MACROS

Estas directivas pueden ser empleadas también sin las macros, aumentando la comodidad de la programación, aunque abundan especialmente dentro de las macros.

* REPT veces ... ENDM (Repeat)

Permite repetir cierto número de veces una secuencia de instrucciones. El bloque de instrucciones se delimita con ENDM (no confundirlo con el final de una macro). Por ejemplo:

```
REPT   2
  OUT  DX,AL
ENDM
```

Esta secuencia se transformará, al ensamblar, en lo siguiente:

```
OUT    DX,AL
OUT    DX,AL
```

Empleando símbolos definidos con (=) y apoyándose además en las macros se puede llegar a crear pseudo-instrucciones muy potentes:

```
SUCESION    MACRO n
        num = 0
        REPT n
          DB num
          num = num + 1
        ENDM            ; fin de REPT
        ENDM            ; fin de macro
```

La sentencia SUCESION 3 provocará la siguiente expansión:

```
DB     0
DB     1
DB     2
```

* IRP simbolo_control, <arg1, arg2, ..., arg_n> ... ENDM (Indefinite repeat)

Es relativamente similar a la instrucción FOR de los lenguajes de alto nivel. Los ángulos (<) y (>) son obligatorios. El símbolo de control va tomando sucesivamente los valores (no necesariamente numéricos) arg1, arg2, ... y recorre en cada pasada todo el bloque de instrucciones hasta alcanzar el ENDM (no confundirlo con fin de macro) sustituyendo simbolo_control por esos valores en todos los lugares en que aparece:

```
IRP    i, <1,2,3>
  DB   0, i, i*i
ENDM
```

Al expansionarse, este conjunto de instrucciones se convierte en lo siguiente:

```
DB     0, 1, 1
DB     0, 2, 4
DB     0, 3, 9
```

Nota: Todo lo encerrado entre los ángulos se considera un único parámetro. Un (;) dentro de los ángulos no se interpreta como el inicio de un comentario sino como un elemento más. Por otra parte, al emplear macros anidadas, deben indicarse tantos símbolos angulares '<' y '>' consecutivos como niveles de anidamiento existan.

Lógicamente, dentro de una macro también resulta bastante útil la estructura IRP:

```
TETRAOUT    MACRO  p1, p2, p3, p4, valor
        PUSH   AX
        PUSH   DX
        MOV    AL,valor
        IRP    cn, <p1, p2, p3, p4>
          MOV  DX, cn
          OUT  DX, AL
        ENDM            ; fin de IRP
        POP    DX
        POP    AX
        ENDM            ; fin de macro
```

Al ejecutar TETRAOUT 318h, 1C9h, 2D1h, 1A4h, 17 se obtendrá:

```
PUSH   AX
PUSH   DX
MOV    AL, 17
MOV    DX, 318h
OUT    DX, AL
MOV    DX, 1C9h
OUT    DX, AL
MOV    DX, 2D1h
OUT    DX, AL
MOV    DX, 1A4h
OUT    DX,AL
POP    DX
POP    AX
```

Cuando se pasan listas como parámetros hay que encerrarlas entre '<' y '>' al llamar, para no confundirlas con elementos independientes. Por ejemplo, supuesta la macro INCD:

```
INCD      MACRO  lista, p
          IRP    i, <lista>
            INC  i
          ENDM              ; fin de IRP
          DEC    p
          ENDM              ; fin de macro
```

Se comprende la necesidad de utilizar los ángulos:

INCD AX, BX, CX, DX se expandirá:

```
INC    AX
DEC    BX ; CX y DX se ignoran (4 parámetros)
```

INCD <AX, BX, CX>, DX se expandirá:

```
INC    AX
INC    BX
INC    CX
DEC    DX          ; (2 parámetros)
```

* IRPC simbolo_control, <c1c2 ... cn> ... ENDM (Indefinite repeat character)

Esta directiva es similar a la anterior, con una salvedad: los elementos situados entre los ángulos (<) y (>) -ahora opcionales, por cierto- son caracteres ASCII y no van separados por comas:

```
IRPC   i, <813>
  DB   i
ENDM
```

El bloque anterior generará al expandirse:

```
DB     8
DB     1
DB     3
```

Ejemplo de utilización dentro de una macro (en combinación con el operador &):

```
INICIALIZA  MACRO  a, b, c, d
          IRPC   iter, <&a&b&c&d>
            DB   iter
          ENDM                ; fin de IRPC
          ENDM                ; fin de macro
```

Al ejecutar INICIALIZA 7, 1, 4, 0 se produce la siguiente expansión:

```
DB     7
DB     1
DB     4
```

```
        DB    0
```

* EXITM

Sirve para abortar la ejecución de un bloque MACRO, REPT, IRP ó IRPC. Normalmente se utiliza apoyándose en una directiva condicional (IF...ELSE...ENDIF). Al salir del bloque, se pasa al nivel inmediatamente superior (que puede ser otro bloque de estos). Como ejemplo, la siguiente macro reserva n bytes de memoria a cero hasta un máximo de 100, colocando un byte 255 al final del bloque reservado:

```
MALLOC    MACRO n
        maximo=100
        REPT n
          IF maximo EQ 0      ; ¿ya van 100?
            EXITM             ; abandonar REPT
          ENDIF
          maximo = maximo - 1
          DB 0                ; reservar byte
        ENDM
        DB 255                ; byte de fin de bloque
        ENDM
```

MACROS AVANZADAS CON NUMERO VARIABLE DE PARÁMETROS

Como se vio al estudiar la directiva IF, existe la posibilidad de chequear condicionalmente la presencia de un parámetro por medio de IFNB, o su ausencia con IFB. Uniendo esto a la potencia de IRP es posible crear macros extraordinariamente versátiles. Como ejemplo, valga la siguiente macro, destinada a introducir en la pila un número variable de parámetros (hasta 10): es especialmente útil en los programas que gestionan interrupciones:

```
XPUSH     MACRO R1,R2,R3,R4,R5,R6,R7,R8,R9,R10
        IRP reg, <R1,R2,R3,R4,R5,R6,R7,R8,R9,R10>
          IFNB <reg>
            PUSH reg
          ENDIF
        ENDM                  ; fin de IRP
        ENDM                  ; fin de XPUSH
```

Por ejemplo, la instrucción:

```
        XPUSH  AX,BX,DS,ES,VAR1
```

Se expandirá en:

```
        PUSH   AX
        PUSH   AX
        PUSH   DS
        PUSH   ES
        PUSH   VAR1
```

El ejemplo anterior es ilustrativo del mecanismo de comprobación de presencia de parámetros. Sin embargo, este ejemplo puede ser optimizado notablemente empleando una lista como único parámetro:

```
XPUSH     MACRO lista
        IRP i, <lista>
          PUSH i
        ENDM
        ENDM

XPOP      MACRO lista
        IRP i, <lista>
          POP i
        ENDM
        ENDM
```

La ventaja es el número indefinido de parámetros soportados (no sólo 10). Un ejemplo de uso puede ser el siguiente:

```
XPUSH  <AX, BX, CX>
XPOP   <CX, BX, AX>
```

Que al expandirse queda:

```
PUSH   AX
PUSH   BX
PUSH   CX
POP    CX
POP    BX
POP    AX
```

BIBLIOTECAS DE MACROS

Una de las facilidades que ofrece el uso de las macros es la creación de bibliotecas, las cuales son grupos de macros que pueden ser incluidas en un programa desde un archivo diferente.
La creación de estas bibliotecas es muy sencilla, únicamente tenemos que escribir un archivo con todas las macros que se necesitan n y guardarlo como archivo de texto.
Para llamar a estas macros solo es necesario utilizar la instrucción Include NombreDelArchivo, en la parte de nuestro programa donde escribiríamos normalmente las macros, esto es, al principio de nuestro programa (antes de la declaración del modelo de memoria).
Suponiendo que se guardó el archivo de las macros con el nombre de MACROS.TXT la instrucción Include se utilizaría de la siguiente forma:

```
  ;Inicio del programa
Include MACROS.TXT
.MODEL SMALL
.DATA
  ;Aquí van los datos
.CODE
Inicio:
  ;Aquí se inserta el código del programa
.STACK
  ;Se define la pila
End Inicio
  ;Termina nuestro programa
```

Programación híbrida

Pascal y ensamblador

Como ya se mencionó, la programación en lenguaje ensamblador proporciona un mayor control sobre el hardware de la computadora, pero también dificulta la buena estructuración de los programas.
La programación híbrida proporciona un mecanismo por medio del cual podemos aprovechar las ventajas del lenguaje ensamblador y los lenguajes de alto nivel, todo esto con el fin escribir programas más rápidos y eficientes.
En esta sección se mostrará la forma para crear programas híbridos utilizando el lenguaje ensamblador y Turbo Pascal.
Turbo Pascal permite escribir procedimientos y funciones en código ensamblador e incluirlas como parte de los programas en lenguaje Pascal; para esto, Turbo Pascal cuenta con dos palabras reservadas: Assembler y Asm.
Assembler permite indicarle a Turbo Pascal que la rutina o procedimiento que se está escribiendo está totalmente escrita en código ensamblador.

Ejemplo de un procedimiento híbrido:

```
Procedure Limpia_Pantalla;
Assembler;
Asm
Mov AX,0600h
Mov BH,18h
Mov CX,0000h
Mov DX,184Fh
Int 10h
End;
```

El procedimiento del listado anterior usa la función 06h de la Int 10h del BIOS para limpiar la pantalla, este procedimiento es análogo al procedimiento ClrScr de la unidad CRT de Turbo Pascal.
Por otro lado, Asm nos permite incluir bloques de instrucciones en lenguaje ensamblador en cualquier parte del programa sin necesidad de escribir procedimientos completos en ensamblador.

Ejemplo de un programa con un bloque de instrucciones en ensamblador:

Este programa muestra como se construye un programa híbrido utilizando un bloque Asm... End; en Turbo Pascal.
El programa solicita que se introduzcan dos números, después calcula la suma por medio de la instrucción Add de ensamblador y finalmente imprime el resultado en la pantalla.

```
Program hibrido;
Uses Crt;
Var
    N1, N2, Res : integer;
Begin
Write("Introduce un número:  ");
Readln(N1);
Write("Introduce un número:  ");
Readln(N2);
Asm
Mov AX, N1;
Add AX, N2;
Mov Res, AX
End;
Writeln("El resultado de la suma es: ",Res);
Readln;
End.
```

El programa del listado anterior realiza la suma de dos cantidades enteras (N1 y N2) introducidas previamente por el usuario, después almacena el resultado en la variable Res y finalmente presenta el resultado en la pantalla.
El lenguaje ensamblador no cuenta con funciones de entrada y salida formateada, por lo cual es muy complicado escribir programas que sean interactivos, es decir, programas que soliciten información o datos al usuario. Es aquí donde podemos explotar la facilidad de la programación híbrida, en el programa anterior se utilizan las funciones Readln y Writeln para obtener y presentar información al usuario y dejamos los cálculos para las rutinas en ensamblador.
En el siguiente listado nos muestra la forma de escribir programas completos utilizando procedimientos híbridos.

```
{Este programa solicita al usuario que presione alguna tecla, cuando la tecla es presionada,
ésta se utiliza para rellenar la pantalla.
El programa termina cuando se presiona la tecla enter.
El programa utiliza tres procedimientos:
Limpia_Pantalla: Este se encarga de borrar la pantalla
Cursor_XY: Este procedimiento reemplaza al GotoXY de Pascal
Imprime_Car: Este procedimiento imprime en pantalla el carácter que se le pasa como
parámetro. }

Program Hibrido2;
Uses Crt;
Var
      Car: Char;
       i,j : integer;
{Este procedimiento limpia la pantalla y pone blanco sobre azul}
Procedure Limpia_Pantalla;
Assembler;
Asm
Mov AX, 0600h
Mov Bh, 17h
Mov CX, 0000h
Mov DX, 184Fh
Int 10h
End;

{Este procedimiento imprime el carácter en la pantalla}
Procedure Imprime_Car(C: Char);
Assembler;
Asm
Mov Ah, 02h
Mov Dl, C
Int 21h
End;

{Este procedimiento tiene la misma función que el procedimiento GotoXY de Turbo Pascal}
Procedure Cursor_XY(X,Y: Byte);
Assembler;
Asm
Mov Ah, 02h
Mov Bh, 00h
Mov Dh, Y
Mov Dl, X
Int 10h
End;

Begin
Limpia_Pantalla;
Repeat
Limpia_Pantalla;
Cursor_XY(0,0);
Write('Introduce un carácter: ');
Car:=ReadKey;
```

```
Imprime_Car(Car);
Limpia_Pantalla;
If car <> #13 then
                Begin
                    For i:=0 to 24 do
                        For j:=0 to 79 do
                                        Begin
                                           Cursor_XY(j,i);
                                           Imprime_Car(Car);
                                        End;
Cursor_XY(30,24);
Write('Presiona enter para salir u otro para seguir...');
Readln;
Until car = #13;
End.
```

INTRODUCCIÓN AL MANEJO DE ARCHIVOS

FORMAS DE TRABAJAR CON ARCHIVOS

MÉTODOS DE TRABAJO CON ARCHIVOS:

MÉTODO FCB:
- Introducción
- Abrir archivos
- Crear un archivo nuevo
- Escritura secuencial
- Lectura secuencial
- Lectura y escritura aleatoria
- Cerrar un archivo

MÉTODO DE CANALES DE COMUNICACIÓN:
- Trabajando con handles
- Funciones para utilizar handles

MÉTODOS DE TRABAJO CON ARCHIVOS

Existen dos formas de trabajar con archivos, la primera es por medio de bloques de control de archivos o "FCB" y la segunda es por medio de canales de comunicación, también conocidos como "handles".
La primera forma de manejo de archivos se viene utilizando desde el sistema operativo CPM, antecesor del DOS, por lo mismo asegura cierta compatibilidad con archivos muy antiguos tanto del CMP como de la versión 1.0 del DOS. Además este método nos permite tener un número ilimitado de archivos abiertos al mismo tiempo. Si se quiere crear un volumen para el disco la única forma de lograrlo es utilizando este método.
Aún considerando las ventajas del FCB el uso de los canales de comunicación es mucho más sencillo y nos permite un mejor manejo de errores, además, por ser más novedoso es muy probable que los archivos así creados se mantengan compatibles a través de versiones posteriores del sistema operativo.
Para una mayor facilidad en las explicaciones posteriores se hará referencia al método de bloques de control de archivos como FCBs y al método de canales de comunicación como handles.
Existen dos tipos de FCB, el normal, cuya longitud es de 37 bytes y el extendido de 44 bytes. Únicamente se tratará el primer tipo, así que de ahora en adelante cuando me refiera a un FCB realmente estoy hablando de un FCB de 37 bytes.
El FCB se compone de información dada por el programador y por información que toma directamente del sistema operativo. Cuando se utilizan este tipo de archivos únicamente es posible trabajar en el directorio actual ya que los FCB no proveen apoyo para el uso de la organización por directorios del DOS.

El FCB está formado por los siguientes campos:

POSICIÓN	LONGITUD	SIGNIFICADO
00H	1 Byte	Drive
01H	8 Bytes	Nombre del archivo
09H	3 Bytes	Extensión
0CH	2 Bytes	Número de bloque
0EH	2 Bytes	Tamaño del registro
10H	4 Bytes	Tamaño del archivo
14H	2 Bytes	Fecha de creación
16H	2 Bytes	Hora de creación
18H	8 Bytes	Reservados
20H	1 Byte	Registro actual
21H	4 Bytes	Registro aleatorio

Para seleccionar el drive de trabajo se sigue el siguiente formato: drive A = 1; drive B = 2; etc. Si se utiliza 0 se tomará como opción el drive que se este utilizando en ese momento.
El nombre del archivo debe estar justificado a la izquierda y en caso de ser necesario se deberán rellenar los bytes sobrantes con espacios, la extensión del archivo se coloca de la misma forma.

El bloque actual y el registro actual le dicen a la computadora que registro será accesado en operaciones de lectura o escritura. Un bloque es un grupo de 128 registros. El primer bloque del archivo es el bloque 0. El primer registro es el registro 0, por lo tanto el último registro del primer bloque sería 127, ya que la numeración inició con 0 y el bloque puede contener 128 registros en total.

ABRIR ARCHIVOS

Para abrir un archivo FCB se utiliza la interrupción 21H, función 0FH. La unidad, el nombre y extensión del archivo deben ser inicializados antes de abrirlo.
El registro DX debe apuntar al bloque. Si al llamar a la interrupción ,esta regresa valor de FFH en el registro AH es que el archivo no se encontró, si todo salió bien se devolverá un valor de 0.
Si se abre el archivo DOS inicializa el bloque actual a 0, el tamaño del registro a 128 bytes y el tamaño del mismo y su fecha se llenan con los datos encontrados en el directorio.

CREAR UN ARCHIVO NUEVO

Para la creación de archivos se utiliza la interrupción 21H función 16H.
DX debe apuntar a una estructura de control cuyos requisitos son que al menos se encuentre definida la unidad lógica, el nombre y la extensión del archivo.
En caso de existir algún problema se devolverá el valor FFH en AL, de lo contrario este registro contendrá el valor de 0.

ESCRITURA SECUENCIAL

Antes de que podamos realizar escrituras al disco es necesario definir el área de transferencia de datos utilizando para tal fin la función 1AH de la interrupción 21H.
La función 1AH no regresa ningún estado del disco ni de la operación, pero la función 15H, que es la que usaremos para escribir al disco, si lo hace en el registro AL. Si este regidtro es igual a cero no hubo error y se actualizan los campos del registro actual y bloque.

LECTURA SECUENCIAL

Antes que nada debemos definir el área de transferencia de archivos o DTA.
Para leer secuencialmente utilizamos la función 14H de la int 21H.
El registro a ser leído es el que se encuentra definido por el bloque y el registro actual. El registro AL regresa el estado de la operación, si AL contiene el valor de 1 o 3 es que hemos llegado al final del archivo. Un resultado de 2 significa que el FCB está mal estructurado.
En caso de no existir error AL contendrá el valor de 0 y los campos bloque actual y registro actual son actualizados.

LECTURA Y ESCRITURA ALEATORIA

La función 21H y la función 22H de la interrupción 21H son las encargadas de realizar las lecturas y escrituras aleatorias respectivamente.
El número de registro aleatorio y el bloque actual son usados para calcular la posición relativa del registro a leer o escribir.
El registro AL regresa la misma información que para lectura o escritura secuencial. La información que será leída se regresaría en el área de transferencia de disco, así mismo la información que será escrita reside en el DTA.

CERRAR UN ARCHIVO

Para cerrar un archivo utilizamos la función 10H de la interrupción 21H.
Si después de invocarse esta función el registro AL contiene el valor de FFH significa que el archivo ha cambiado de posición, se cambió el disco o hay un error de acceso al disco.

TRABAJANDO CON HANDLES

El uso de handles para manejar los archivos facilita en gran medida la creación de archivos y el programador puede concentrarse en otros aspectos de la programación sin preocuparse en detalles que pueden ser manejados por el sistema operativo.

La facilidad en el uso de los handles consiste en que para operar sobre un archivo únicamente es necesario definir el nombre del mismo y el número del handle a utilizar, toda la demás información es manejada internamente por el DOS.
Cuando utilizamos este método para trabajar con archivos no existe una distinción entre accesos secuenciales o aleatorios, el archivo es tomado simplemente como una cadena de bytes.

FUNCIONES PARA UTILIZAR HANDLES

Las funciones utilizadas para el manejo de archivos por medio de handles son descritas en el tema Interrupciones, en la sección dedicada a la interrupción 21H.

EJEMPLOS PRÁCTICOS DE PROGRAMAS

DESPLEGAR UN MENSAJE EN PANTALLA

Uno de los programas más sencillos, pero en cierta forma práctico, es uno que despliegue una cadena de caracteres en la pantalla. Eso es lo que hace el siguiente programa:

Programa :

```
; Primero definimos el modelo de memoria, en este caso small
.MODEL SMALL
.CODE              ; Declaramos el área que contendrá el código
Inicio:            ; Etiqueta de inicio del programa:
MOV AX,@DATA       ; Vamos a colocar la dirección del segmento de datos
MOV DS,AX          ;  en DS, usando como intermediario a AX
MOV DX,OFFSET Cadena ; Colocamos en DX la dirección, dentro del
                   ;  segmento, de la cadena a desplegar
MOV AH,09          ; Utilizaremos la función 09 de la interrupción
INT 21H            ;  21H para desplegar la cadena.
MOV AH,4CH         ; Por medio de la función 4CH de la interrupción
INT 21H            ;  21H terminaremos nuestro programa
.DATA              ; Declaramos el segmento de datos
Cadena DB 'Mensaje del programa.$' ; Cadena a desplegar
.STACK             ; Declaramos la pila
END Inicio         ; Final de nuestro programa
```

Cuando se crea un programa no es necesario escribir los comentarios que van después de las comillas, sin embargo es una técnica recomendable para que en caso de errores o mejoras al código sea más sencillo encontrar la parte deseada.
Para ensamblar este programa primero se guarda en formato ASCII con un nombre válido, por ejemplo: program1.asm
Para ensamblarlo se utiliza el MASM, el comando de ensamble es: masm program1.
Para enlazarlo y hacerlo ejecutable tecleamos: TLINK program1.
Una vez terminados estos pasos es posible ejecutarlo tecleando: program1 [Enter].
Para utilizar el programa directamente en su computadora guarde este archivo como ASCII o texto, llévelo a su PC, con algún editor elimine todos estos comentarios y los comentarios del principio y ensámblelo.

DESPLEGAR NÚMEROS HEXADECIMALES DEL 15 AL 0

Este programa despliega los 16 caracteres correspondientes al código hexadecimal en orden descendente.

Programa:

```
; Inicio del programa, definimos el modelo de memoria a usar y el segmento
; de código
.MODEL SMALL       ; Modelo de memoria
.CODE              ; Area de código
Inicio:            ; Etiqueta de inicio del programa
MOV AX,@DATA       ; Inicializa el registro DS con la dirección dada
MOV DS,AX          ;  por @DATA (Segmento de datos).
MOV DX, OFFSET Titulo ; Obtiene la dirección de la cadena de caracteres
```

```
MOV AH,09           ; Usamos la función 09H de la interrupción 21H
INT 21H             ;  para desplegar la cadena cuya dirección obtuvimos.
MOV CX,16           ; Contador de caracteres que se mostrar n
MOV BX, OFFSET Cadena ; Permite acceso a la cadena donde se encuentran los
                    ;  valores a desplegar
Ciclo:              ; Etiqueta para generar un ciclo
MOV AL,CL           ; Coloca en AL el número a traducir y lo traduce
XLAT                ; usando la instrucción XLAT
MOV DL,AL           ; Coloca en DL el valor a ser desplegado por medio de la
MOV AH,02           ;  función 2 de la interrupción 21H
INT 21H             ; Despliega el carácter
MOV DL,10           ; Salta una línea desplegando el carácter 10
INT 21H             ; Despliega el carácter
MOV DL,13           ; Produce un retorno de carro desplegando el carácter 13
INT 21H             ; Despliega el retorno de carro
LOOP Ciclo          ; Decrementa en uno a CX y brinca a la etiqueta Ciclo
                    ;  siempre y cuando CX no sea igual a cero
MOV AH,4C           ; Utiliza la función 4C de la interrupción 21H para
INT 21H             ; finalizar el programa

; Inicio del segmento de datos
.DATA               ;Define el segmento de datos
Titulo DB 13,10,'Desplegar los números hexadecimales del 15 al 1'
  DB 13,10,'$'      ; Cadena a desplegar al inicio del programa
Cadena DB '0123456789ABCDEF'  ; Cadena con los dígitos hexadecimales
; Declaración del segmento de la pila
.STACK
END Inicio          ;Declaración del final del programa
```

El comando XLAT busca la cadena o tabla localizada en BX, el registro AL contiene el número de bytes, a partir de la dirección de inicio, que se recorrerá el apuntador para buscar un dato, el contenido de AL es remplazado por el byte donde se encuentra el apuntador.
El proceso de ensamblado es igual al del ejemplo anterior.

OPERACIONES BÁSICAS

En el siguiente ejemplo se utilizan la mayor parte de las instrucciones vistas anteriormente; su objetivo es realizar las operaciones de suma, resta, multiplicación o división de dos cantidades.
Para accesar a cada una de las opciones disponibles se hace uso de un menú en el que se presentan las operaciones disponibles.

Programa:

```
.MODEL SMALL            ; Define el modelo de memoria
.DATA                   ; Define el segmentos de datos

ErrorCAP DB 0           ;Bandera de error en la captura de las cantidades
Cantidad DB 0           ;Cantidad sobre la que se opera. Si es 0 la cantidad
                        ; será la 1, y si es 1 será la 2.
CantUnoR DW 0           ;Guardará la cantidad 1 convertida en binario
CantDosR DW 0           ;Guardará la cantidad 2 convertida en binario
CantUnoN DB 6,0,6 DUP(?) ;Variable que almacena la cantidad 1
CantDosN DB 6,0,6 DUP(?) ;Variable que almacena la cantidad 2
Función  DB 0           ;Variable que almacena la opción a realizar
Resulta  DB 13,10,'Resultado: $'
ResultaR DB 11 DUP(?)
Mensaje  DB 13,10,'Operaciones básicas entre dos números'
         DB 13,10,13,10,'$'
Pregunta DB 13,10,'Presione: ',13,10
         DB ' 1  Multiplicación ',13,10
         DB ' 2  División ',13,10
         DB ' 3  Suma ',13,10
         DB ' 4  Resta ',13,10
```

```
        DB ' 5  Salir ',13,10,'$'
Error    DB 7,13,10,'Selección inválida (1-5)',13,10,'$'
Error1   DB 7,13,10,'Cantidad 1 inválida.          ',13,10,'$'
Error2   DB 7,13,10,'Cantidad 2 inválida.          ',13,10,'$'
Error3   DB 7,13,10,'Cantidad fuera de rango (65535)      ',13,10,'$'
Error4   DB 7,13,10,'Intento de división por cero. ',13,10,'$'
CantUnoM DB 13,10,'Introduzca la cantidad 1 (Menor a 65535): $'
CantDosM DB 13,10,'Introduzca la cantidad 2 (Menor a 65535): $'

; Tabla de potencias para conversión binaria/ASCII

Potencia DW 0001h, 000Ah, 0064h, 03E8h, 2710h
PotenciaF DW $

  .CODE                 ;Define el  rea de código
  Empieza:              ;Etiqueta de inicio del programa

  Mov AH, 0Fh           ;Obtiene modo de video actual
  Int 10h
  Mov AH, 00            ;Cambia el modo de video al mismo anterior
  Int 10h               ; con la finalidad de que se borre la pantalla
  Mov AX, @Data         ;Obtiene la dirección del segmento de datos
  Mov Ds, Ax            ;Inicializa a DS con esa dirección
  Mov Dx, Offset Mensaje  ;Despliega el título del programa
  Call Imprime          ;Llama a un procedimiento
  Mov Si, Offset ResultaR ;Inicializa la variable ResultaR
  Add Si,11
  Mov Al,'$'
  Mov [Si], Al
OTRA:
  Mov Dx,Offset Pregunta  ;Despliega menú de opciones
  Call Imprime
  Call ObtenTecla       ;Espera a que se presione la opción deseada
  Cmp Al, 49            ;Compara la selección con el dígito 1 ASCII
  Jae SIGUE             ;Si la opción es mayor a 1 brinca a SIGUE
  Mov Dx, Offset Error    ;Despliega mensaje de error
  Call Imprime
  Jmp OTRA              ;Brinca a OTRA para volver a preguntar
SIGUE:
  Cmp Al,53             ;Compara la selección con el dígito 5 ASCII
  Jbe TODOBIEN          ;Si es menor a 5 brinca a TODOBIEN, sino continúa
  Mov Dx, Offset Error    ;Si la opción fu, mayor a 5 despliega el error
  Call Imprime
  Jmp OTRA
TODOBIEN:
  Cmp Al,53             ;Compara la selección con el dígito 5 ASCII
  Jnz CHECATODO         ;Si no es igual brinca a CHECATODO
  Jmp FUNCIÓN5          ;Si es igual brinca a FUNCIÓN5 para terminar
CHECATODO:
  Mov Función, Al       ;Guarda el número de función a realizar
CAPCANT01:
  Mov Dx, Offset CantUnoM ;Mensaje de captura de la cantidad 1
  Call Imprime
  Mov Ah, 0Ah           ;Captura la cantidad (hasta 8 dígitos)
  Mov Dx, Offset CantUnoN
  Int 21h
  Mov ErrorCAP, 0       ;Supone que no hay errores y que se est
  Mov Cantidad, 0       ; operando sobre la cantidad 1
  Call ConvNUM          ;Convierte cantidad 1 a binario
  Cmp ErrorCAP, 1       ;Verifica si hubo error
  Jz CAPCANT01          ;En caso afirmativo regresa a la captura.
  Mov CantUnoR, Bx      ;Guarda el resultado de la conversión
CAPCANT02:
  Mov ErrorCAP, 0       ;Supone que no hay error
```

```
  Mov Cantidad, 1        ;Indica a ConvNUM que se trabajar  con cantidad 2
  Mov Dx, Offset CantDosM ;Mensaje de captura de cantidad 2
  Call Imprime
  Mov Ah, 0Ah            ;Captura de la cantidad 2
  Mov Dx, Offset CantDosM
  Int 21H
  Call ConvNum           ;Convierte la cantidad 2 a binario
  Cmp ErrorCAP, 1        ;Verifica si existió algún error
  Jz, CAPCANT02          ;En caso afirmativo regresa a la captura
  Mov CantDosR, Bx       ;Almacena el valor binario de cantidad 2

;La siguiente parte es el proceso de selección de la operación
;Que se realizará:

  Mov Al, Función        ;Carga en Al la función que seleccionó el usuario
  Cmp Al, 31h            ;Revisa si es 1
  Jne FUNCIÓN2           ;Si no es brinca a FUNCIÓN2
  Call Multiplica        ;Multiplica las cantidades
  Jmp OTRA               ;Regresa al menú principal

FUNCIÓN2:
  Cmp Al, 32h            ;Revisa si es 2
  Jne FUNCIÓN3           ;Si no es brinca a FUNCIÓN3
  Call Divide            ;Divide las cantidades
  Jmp OTRA
FUNCIÓN3:
  Cmp Al, 33h            ;Revisa si es 3
  Jne FUNCIÓN4           ;Si no es brinca a FUNCIÓN4
  Call Suma              ;Suma las cantidades
  Jmp OTRA

FUNCIÓN4:
  Cmp Al, 34h            ;Revisa si es 4
  Jne FUNCIÓN5           ;Si no es brinca a FUNCIÓN5
  Call Resta             ;Resta las cantidades
  Jmp OTRA

FUNCIÓN5:
  Mov Ax, 4C00h          ;Esta función termina la ejecución
  Int 21h                ;del programa

; Procedimientos o rutinas del programa

Multiplica Proc Near     ;Indicador de inicio de procedimiento
  Xor Dx, Dx             ;Dx = 0
  Mov Ax, CantUnoR       ;Primera cantidad
  Mov Bx, CantDosR       ;Segunda cantidad
  Mul Bx ;Multiplica
  Call ConvASCII         ;Convierte en ASCII
  Mov Dx, Offset Resulta  ;Imprime mensaje del resultado
  Call Imprime
  Mov Dx, Offset ResultaR ;Imprime resultado
  Call Imprime
  Ret                    ;Regresa al programa principal
Multiplica Endp          ;Indicador de fin de procedimiento
Divide Proc Near
  Mov Ax, CantUnoR       ;Carga la cantidad 1 (dividendo)
  Mov Bx, CantDosR       ;Carga la cantidad 2 (divisor)
  Cmp Bx, 0              ;Revisa que el divisor no sea cero
  Jnz DIVIDE01           ;Si no es cero brinca a DIVIDE01
  Mov Cantidad, 3        ;Hubo error así que despliega el mensaje y regresa al
programa
  Call HuboError
  Ret
```

```
DIVIDE01:
  Div Bx              ;Efectúa la división
  Xor Dx, Dx          ;Dx = 0. El residuo no es utilizado
  Call ConvASCII      ;Convierte en ASCII el resultado
  Mov Dx, Offset Resulta  ;Despliega el mensaje del resultado
  Call Imprime
  Mov Dx, Offset ResultaR ;Despliega el resultado
  Call Imprime
  Ret
Divide Endp

Suma Proc Near
  Xor Dx, Dx          ;Dx = 0 por si acaso existe acarreo
  Mov Ax, CantUnoR    ;Cantidad 1
  Mov Bx, CantDosR    ;Cantidad 2
  Add Ax, Bx          ;Realiza la suma
  Jnc SUMACONV        ;Si no existió acarreo brinca a SUMACONV
  Adc Dx,0            ;Si existió
SUMACONV:
  Call ConvASCII      ;Convierte en ASCII el resultado
  Mov Dx, Offset Resulta  ;Despliega el mensaje del resultado
  Call Imprime
  Mov Dx, Offset ResultaR ;Despliega el resultado
Call Imprime
  Ret
Suma Endp

Resta Proc Near
  Xor Dx, Dx          ;Dx = 0 por si existe acarreo
  Mov Ax, CantUnoR    ;Ax = cantidad 1
  Mov Bx, CantDosR    ;Bx = cantidad 2
  Sub Ax, Bx          ;Realiza la resta
  Jnc RESTACONV       ;Si no hay acarreo brinca a RESTACONV
  Sbb Dx,0            ;Si hay acarreo
RESTACONV:
  Call ConvASCII      ;Convierte en ASCII el resultado
  Mov Dx, Offset Resulta  ;Despliega el mensaje del resultado
  Call Imprime
  Mov Dx, Offset ResultaR ;Despliega el resultado
  Call Imprime
  Ret
Resta Endp

Imprime Proc Near
  Mov Ah, 09          ;Utiliza la función 9 de la interrupción
  Int 21h             ;21h para desplegar una cadena
  Ret
Imprime Endp

ObtenTecla Proc Near
  Mov ah, 0           ;Utiliza la interrupción 16h para
  Int 16h             ; leer una tecla
  Ret
ObtenTecla Endp
ConvNum Proc Near
  Mov Dx, 0Ah         ;Multiplicador es 10
  Cmp Cantidad, 0     ;Verifica si es la cantidad 1
  Jnz CONVNUM01       ;No fu,, entonces es cantidad 2 y brinca
  Mov Di, Offset CantUnoN + 1 ;Bytes leídos en la cantidad 1
  Mov Cx, [Di]
  Mov Si, Offset CantUnoN + 2 ;La cantidad 1
  Jmp CONVNUM02
CONVNUM01:
  Mov Di, Offset CantDosN + 1 ;Bytes leídos de la cantidad 2
```

```
  Mov Cx, [Di]
  Mov Si, Offset CantDosN + 2 ;La cantidad 2
CONVNUM02:
  Xor Ch, Ch ;CH = 0
  Mov Di, Offset Potencia     ;Dirección de la tabla de potencias
  Dec Si
  Add Si, Cx
  Xor Bx, Bx
  Std
CONVNUM3:
  Lodsb                ;Carga en AL el byte cuya dirección es DS:SI
  Cmp Al, "0"          ;Compara el byte con el dígito 0
  Jb CONVNUM04         ;Si es menor es inválido y brinca
  Cmp Al, "9"          ;Compara el byte con el dígito 9
  Ja CONVNUM04         ;Si es mayor es inválido y brinca
  Sub Al, 30h          ;Convierte el dígito de ASCII a binario
  Cbw                  ;Convierte a palabra
  Mov Dx, [Di]         ;Obtiene la potencia a ser usada para multiplicar
  Mul Dx               ;Multiplica el número
  Jc CONVNUM05         ;Si hay acarreo brinca (fu, mayor a 65535)
  Add Bx, Ax           ;Suma el resultado a BX
 Jc CONVNUM05          ;Si hay acarreo brinca
  Add Di, 2            ;Va a la siguiente potencia de 10
  Loop CONVNUM03       ;Brinca hasta que CX sea igual a 0
  Jmp CONVNUM06
CONVNUM04:
  Call HuboERROR       ;Hubo algún error, despliega mensaje
  Jmp CONVNUM06        ;Brinca a CONVNUM06
CONVNUM05:
  Mov Cantidad, 2      ;Hubo acareo en la conversión, lo que
  Call HuboERROR       ; significa que la cantidad es mayor a 65535
CONVNUM06:
  Cld                  ;Regresa la bandera de dirección a su estado
  Ret                  ; normal
ConvNum Endp

ConvASCII Proc Near
  Push Dx
  Push Ax              ;Guarda el resultado en la pila
  Mov Si, Offset ResultaR ;Inicializa la variable ResultaR
  Mov Cx, 10           ; llenándola con asteriscos
  Mov Al, '*'
ConvASCII01:
  Mov [Si], Al
  Inc Si
  Loop ConvASCII01
  Pop Ax
  Pop Bx
  Mov Bx, Ax           ;Palabra baja de la cantidad
  Mov Ax, Dx           ;Palabra alta de la cantidad
  Mov Si, Offset ResultaR ;Cadena donde se guarda el resultado
  Add Si, 11
  Mov Cx, 10           ;Divisor = 10
OBTENDIGITO:
  Dec Si
  Xor Dx, Dx           ;DX contendrá el residuo
  Div Cx               ;Divide la palabra alta
  Mov Di, Ax           ;Guarda cociente en DI
  Mov Ax, Bx           ;Carga palabra baja en AX
  Div Cx               ;DX contiene registro de la división
  Mov Bx, Ax           ;Guarda el cociente
  Mov Ax, Di           ;Regresa la palabra alta
  Add Dl, 30h          ;Convierte residuo en ASCII
  Mov [Si], Dl         ;Lo almacena
```

```
  Or Ax, Ax             ;Si la palabra alta no es cero
  Jnz OBTENDIGITO       ; brinca a OBTENDIGITO
  Or Bx, Bx             ;Si la palabra baja no es cero
  Jnz OBTENDIGITO       ; brinca a OBTENDIGITO
  Ret
ConvASCII Endp

HuboERROR Proc Near
  Cmp Cantidad,0        ;Es la cantidad 1?
  Jnz HUBOERROR02       ;no
  Mov Dx, Offset Error1
  Call Imprime
  Mov ErrorCAP, 1       ;Enciende la bandera de error
  Jmp HUBOERROR05
HUBOERROR02:
  Cmp Cantidad, 1       ;Es la cantidad 2?
  Jnz HUBOERROR03       ;no
  Mov Dx, Offset Error2
  Call Imprime
  Mov ErrorCAP, 1
 Jmp HUBOERROR05
HUBOERROR03:
  Cmp Cantidad, 2       ;Es una cantidad fuera de rango?
  Jnz HUBOERROR04       ;no
  Mov Dx, Offset Error3
  Call Imprime
  Mov ErrorCAP, 1
  Jmp HUBOERROR05
HUBOERROR04:
  Mov Dx, Offset Error4   ;Error de división por cero
  Call Imprime
  Mov ErrorCAP, 1
HUBOERROR05:
  Ret
HuboERROR Endp
.STACK
End Empieza
```