PROGRAMACIÓN BÁSICA.

Lenguajes de Alto Nivel:

codifican comandos poderosos.
cada uno de los cuales puede generar muchas instrucciones en lenguaje de máquina.

Un lenguaje de alto nivel utiliza un *compilador* para traducir el código fuente a lenguaje de máquina (técnicamente, código objeto).

Lenguajes de Bajo Nivel:

lenguaje ensamblador de bajo nivel codifican instrucciones simbólicas,
cada una de las cuales genera una instrucción en lenguaje de máquina.

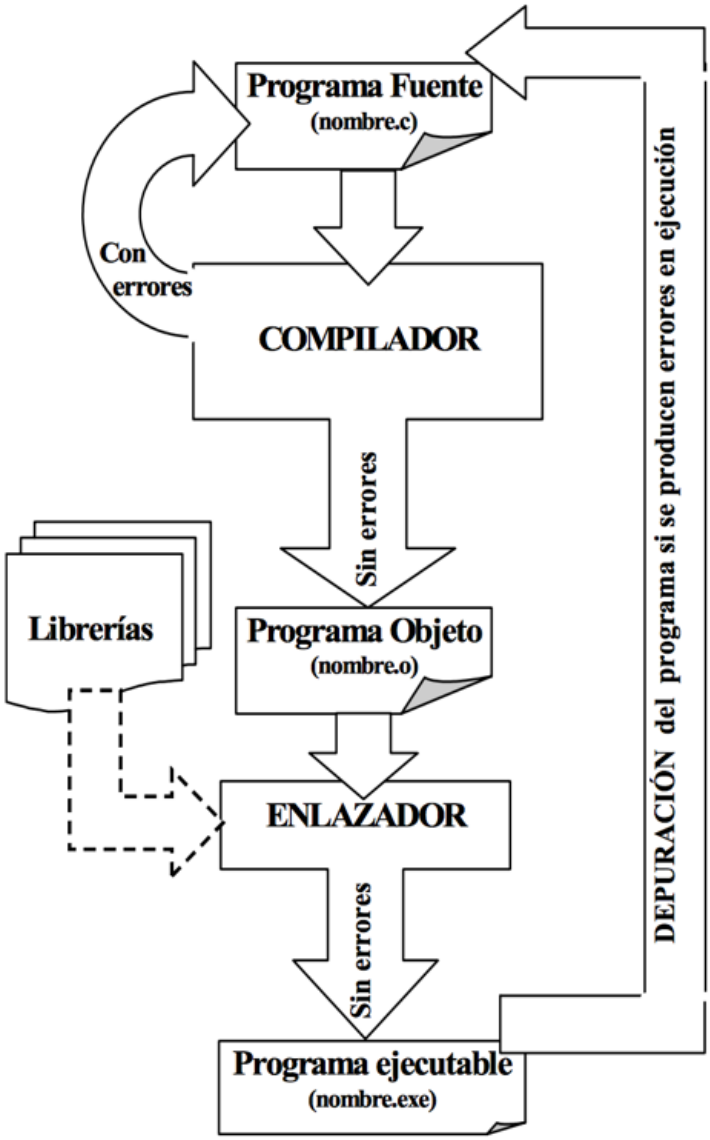
Un lenguaje de bajo nivel utiliza un *ensamblador* para realizar la traducción.
Un programa *enlazador* para ambos niveles, alto y bajo ; completa el proceso al convertir el código objeto en lenguaje ejecutable de máquina.

- Proporciona más control sobre el manejo particular de los requerimientos de hardware.
- Genera módulos ejecutables más pequeños y más compactos.
- Con mayor probabilidad tiene una ejecución más rápida.

Introducción

Un programa escrito en un lenguaje de programación, no puede ser ejecutado directamente por un ordenador, sino que debe ser traducido a lenguaje máquina.

Las etapas por las que debe pasar un programa escrito en un lenguaje de programación, hasta poder ser ejecutable son:



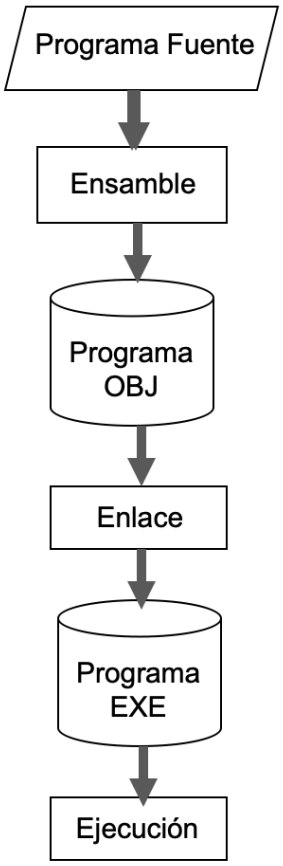
1. **Programa fuente:** Programa escrito en un lenguaje de alto nivel (texto ordinario que contiene las sentencias del programa en un lenguaje de programación). Necesita ser traducido a código máquina para poder ser ejecutado.

2. **Compilador:** Programa encargado de traducir los programas fuentes escritos en un lenguaje de alto nivel a lenguaje máquina y de comprobar que las llamadas a las funciones de librería se realizan correctamente.

3. **Programa (o código) objeto:** Es el programa fuente traducido (por el compilador) a código máquina. Aún no es directamente ejecutable.

4. **Programa Ejecutable:** Traducción completa a código máquina, realizada por el enlazador, del programa fuente y que ya es directamente ejecutable.

Este escencario aplica para los programas escritos en Lenguajes de Alto Nivel.



Creación de programa fuente, mediante un editor de texto.
(archivo con extención .ASM

El proceso de ensamble consiste en la traducción del código fuente en código objeto y la generación de un archivo intermedio .OBJ (objeto).

Se crea el encabezado del programa en el módulo OBJ.

El módulo .OBJ aún no está en forma ejecutable.

El paso de enlace implica convertir el módulo .OBJ en un módulo de código de máquina .EXE (ejecutable).

Una de las tareas del enlazador es combinar los programas ensamblados en forma separada en un módulo ejecutable.

El último paso es cargar el programa para su ejecución.

Ya que el cargador conoce en dónde está el programa a punto de ser cargado,

El cargador desecha el encabezado.

AMBIENTE DE DESAROLLO

Editor para crear el programa fuente.

puede ser cualquier editor de textos que se tenga a la mano,
Notepad, Texpad (sugerido), Edit -DOS-, VI (Linux).

Compilador programa que "traduce" el programa fuente a un programa objeto.

Compilador que en nuestro caso es un ensamblador, utilizaremos el MASM (macro ensamblador de Microsoft) ya que es el mas común

Enlazador *(*linker), que genere el programa ejecutable a partir del programa objeto.

utilizaremos el programa **link**.

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.

Formato de un programa

Para poder comunicarnos en cualquier lenguaje, incluyendo los lenguajes de programación, es necesario seguir un conjunto de reglas, de lo contrario no podríamos expresar lo que deseamos.

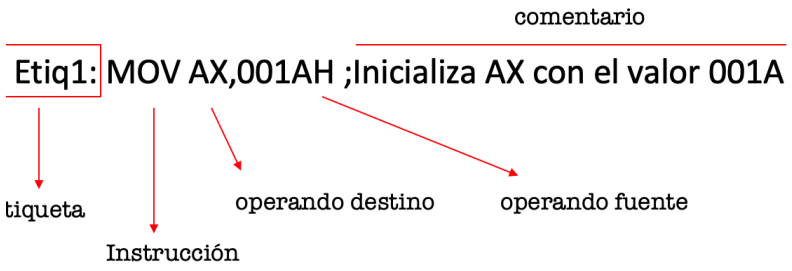
Básicamente el formato de una línea de código en lenguaje ensamblador consta de cuatro partes:

1. **Etiqueta, variable o constante:** No siempre es definida, si se define es necesario utilizar separadores para diferenciarla de las otras partes, usualmente espacios, o algún símbolo especial.

2. **Directiva o instrucción:** es el nombre con el que se conoce a la instrucción que queremos que se ejecute.

3. **Operando(s):** la mayoría de las instrucciones en ensamblador trabajan con dos operandos, aunque hay instrucciones que funcionan solo con uno. El primero normalmente es el operando destino, que es el depósito del resultado de alguna operación; y el segundo es el operando fuente, que lleva el dato que será procesado. Los operandos se separan uno del otro por medio de una coma ",".

4. **Comentario**: como su nombre lo indica es tan solo un escrito informativo, usado principalmente para explicar que está haciendo el programa en determinada línea; se separa de las otras partes por medio de un punto y coma ";". Esta parte no es necesaria en el programa, pero nos ayuda a depurar el programa en caso de errores o modificaciones.



declaración de una constante esta dado por:

UNO EQU 0001H

"EQU" es la directiva utilizada para usar a "UNO" como constante

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      -----> define el tipo de memoria que se utilizará.
.STACK 64         -----> pide al ensamblador que reserve un espacio de memoria para las                                operaciones de la pila
.CODE             -----> indica que lo que esta a continuación es nuestro programa
Programa:
   MOV AX,4C00H
   INT 21H
END Programa      -----> final del programa
```

El programa 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.

Modelo de Memoria	Descripción
tiny	El código y los datos del programa deben ajustarse dentro del mismo segmento de 64 Kb. Código y datos son near.
small	El código del programa debe ajustarse dentro de un segmento simple de 64 Kb, y los datos del programa deben estar en otro segmento separado de 64 Kb. Código y datos son near.
medium	El código del programa puede ser mayor que 64 Kb, pero los datos del programa deben ajustarse en un sólo segmento de 64 Kb. El código es far, mientras que los datos son near.
compact	El código del programa debe estar dentro de un segmento de 64 Kb, pero los datos del programa pueden ocupar más de 64 Kb. El código es near, mientras que los datos son far. Ningún arreglo de datos puede ser mayor de 64 Kb.
large	El código y los datos del programa pueden ocupar más de 64 Kb, pero ningún arreglo de datos puede ser mayor de 64 Kb. El código y los datos son far.
huge	El código y los datos del programa pueden ocupar más de 64 Kb y los arreglos de datos puede exceder los 64 Kb. El código y los datos son far. Los apuntadores a elementos dentro de un arreglo son far.

Tabla 2.1. Modelos de memoria

2.1 Ensamblador (y Ligador) a Utilizar.

Al construir un programa algunos de sus módulos pueden colocarse en el mismo módulo fuente y ensamblarse juntos, otros pueden estar en módulos diferentes y ser ensamblados separadamente. En cualquier caso, los módulos objeto resultantes, algunos de los cuales pueden estar agrupados en librerías, deben ser enlazados para formar el módulo de carga, antes de que se pueda ejecutar el programa. Además de dar como salida el módulo de carga, el linker o enlazador imprime un mapa de memoria que indica donde serán cargados los módulos objeto en la memoria.

Una práctica común es combinar los beneficios de ambos niveles de programación:
codificar el grueso de un proyecto en un lenguaje de alto nivel y los módulos críticos que requieren administrar memoria y regitros de hardware; en lenguaje ensamblador.

Aunque todos los ensambladores realizan básicamente las mismas tareas, podemos clasificarlos de acuerdo a características.

Ejemplo: Hola Mundo (DOSBox)

```
.model small
.stack 64
.data
  msj db 'Hola Mundo','$'

.code
inicio:
```

; guardar en el registro de segmento DS, la posición del segmento de

```
  mov ax,@data

;imprime el mensaje
  mov ds,ax
  mov ah,09h
  mov dx,offset msj
  int 21h

Salir:
mov ah,04ch
int 21h

end
```

Ensambladores Cruzados (Cross-Assembler).
Se denominan así los ensambladores que se utilizan en una computadora que posee un procesador diferente al que tendrán las computadoras donde va a ejecutarse el programa objeto producido.

El empleo de este tipo de traductores permite aprovechar el soporte de medios físicos (discos, impresoras, pantallas, etc.), y de programación que ofrecen las máquinas potentes para desarrollar programas que luego los van a ejecutar sistemas muy especializados en determinados tipos de tareas.

Ensambladores Residentes.
Son aquellos que permanecen en la memoria principal de la computadora y cargan, para su ejecución, al programa objeto producido. Este tipo de ensamblador tiene la ventaja de que se puede comprobar inmediatamente el programa sin necesidad de transportarlo de un lugar a otro, como se hacía en cross-assembler, y sin necesidad de programas simuladores.

Sin embargo, puede presentar problemas de espacio de memoria, ya que el traductor ocupa espacio que no puede ser utilizado por el programador. Asimismo, también ocupará memoria el programa fuente y el programa objeto. Esto obliga a tener un espacio de memoria relativamente amplio. Es el indicado para desarrollos de pequeños sistemas de control y sencillos automatismos empleando microprocesadores.

La ventaja de estos ensambladores es que permiten ejecutar inmediatamente el programa; la desventaja es que deben mantenerse en la memoria principal tanto el ensamblador como el programa fuente y el programa objeto.

Macroensambladores.
Son ensambladores que permiten el uso de macroinstrucciones (macros).

Debido a su potencia, normalmente son programas robustos que no permanecen en memoria una vez generado el programa objeto. Puede variar la complejidad de los mismos, dependiendo de las posibilidades de definición y manipulación de las macroinstrucciones, pero normalmente son programas bastantes complejos, por lo que suelen ser ensambladores residentes.

Microensambladores.
Generalmente, los procesadores utilizados en las computadoras tienen un
repertorio fijo de instrucciones, es decir, que el intérprete de las mismas interpretaba de igual forma un determinado código de operación.

El programa que indica al intérprete de instrucciones de la UCP cómo debe actuar se denomina microprograma. El programa que ayuda a realizar esta microprograma se llama microensamblador. Existen procesadores que permiten la modificación de sus microprogramas, para lo cual se utilizan microensambladores.

Ensambladores de una fase.
Estos ensambladores leen una línea del programa fuente y la traducen directamente para producir una instrucción en lenguaje máquina o la ejecuta si se trata de una pseudoinstrucción. También va construyendo la tabla de símbolos a medida que van apareciendo las definiciones de variables, etiquetas, etc.

Debido a su forma de traducción, estos ensambladores obligan a definir los símbolos antes de ser empleados para que, cuando aparezca una referencia a un determinado símbolo en una instrucción, se conozca la dirección de dicho símbolo y se pueda traducir de forma correcta. Estos ensambladores son sencillos, baratos y ocupan poco espacio, pero tiene el inconveniente indicado.

Ensambladores de dos fases.
Los ensambladores de dos fases se denominan así debido a que realizan la traducción en dos etapas. En la primera fase, leen el programa fuente y construyen una tabla de símbolos; de esta manera, en la segunda fase, vuelven a leer el programa fuente y pueden ir traduciendo totalmente, puesto que conocen la totalidad de los símbolos utilizados y las posiciones que se les ha asignado. Estos ensambladores son los más utilizados en la actualidad.

COMENTARIOS EN LENGUAJE ENSAMBLADOR

- El uso de comentarios a lo largo de un programa puede mejorar su claridad .
- Un comentario empieza con punto y coma (;) y, en donde quiera que lo codifique; el ensamblador supone que todos los caracteres a la derecha en esa línea son comentarios.
- Un comentario puede aparecer sólo en una línea o a continuación de una instrucción en la misma línea, como lo muestran los dos ejemplos siguientes:

```
1. ;Toda esta línea es un comentario
2. ADD AX,BX /Comentario en la misma línea que la instrucción
```

- Un comentario aparece sólo en un listado de un programa fuente en ensamblador y no genera código de máquina.
- El número de comentarios no afecta el tamaño o la ejecución del programa ensamblado.
- Otra manera de insertar comentarios es mediante la instrucción COMMENT.

Un programa escrito en lenguaje ensamblador se ensambla por medio de un programa ensamblador: Los programas mas comunes para el 8088, 8086 con el Turboensamblador (TASM) de la compañía Borland y el de la compañía Microsoft (MASM).

Un programa en lenguaje ensamblador consiste en un conjunto de enunciados. Los dos tipos de enunciados o líneas de programación son:

- **Instrucción**, tal como MOV y ADD, que el ensamblador traduce a código objeto
- **Directiva**, que indican al ensamblador que realice una acción especifica, como definir un elemento de dato.

Enunciado O Linea De Programación:

[Etiqueta] <Operación o directiva> [operando (s)] [; comentarios]

NOTA: Los corchetes indican una entrada opcional

Etiqueta o Identificador.- Es un nombre para designar un dato y la dirección donde se encuentra dicho dato. Puede consistir de los siguientes caracteres:

- Letras del alfabeto: A-Z a -z
- Dígitos: (0 9) (no puede ser el primer carácter)
- Caracteres especiales: signo de interrogación (¿)
- Subrayado (_)
- Signo de pesos ($)
- Arroba (@)
- Punto (.) (no puede ser el primer carácter)

no debe tener espacios en blanco en medio de la etiqueta. Para definir una etiqueta se usan las directivas DW, DB. Una etiqueta puede tener los siguientes usos:

Como variable.- Ejemplo:

```
num db 65      ; A num se le asocia el valor 65 y la dirección donde se encuentra dicho dato.
```

También podemos hacer:

```
num dw 6567H
```

Como dirección.- En la cual puede continuar el programa:

```
SUMA:  ADD  AX, DX
       _____________

       _____________

       JMP   SUMA
```

OPERACIÓN o **DIRECTIVA.-** En este campo deberá estar el nombre de la operación (Mnemónico) que el micro deberá realizar, o el nombre de una orden (directiva) que el programa ensamblador deberá ejecutar al momento de ensamblar nuestro programa. Al ensamblarse nuestro programa se genera un código ejecutable, que el micro entiende. La directiva no genera código ejecutable solo el mnemónico.

(Instrucción) Etiq1: MOV AX, 20H

(Directiva) Constante Equ 100

Algunas directivas serían:

- .Model Indica el modelo de memoria que usara el programa
- .Stack Indica el tamaño del Stack o pila
- .Data Indica el inicio del segmento de datos
- .Code Indica el inicio del segmento de código
- End Indica el fin del programa

OPERANDO.- Es una fuente de datos para una instrucción. Algunas instrucciones, como CLC y RET, no necesitan un operando, mientras que otras pueden tener uno o dos operandos. Donde existan dos operandos, el segundo es el fuente, que contiene ya sea datos que serán entregados (inmediatos) o bien la dirección (de un registro o en memoria) de los datos. El dato fuente no es cambiado por la operación. El primer operando es el destino, que contiene datos en un registro o en memoria y que será procesado.

```
operación operandol, operando2
```

Los operandos pueden afectar el direccionamiento de datos.

Operandos registro.

Para este tipo, el registro proporciona el nombre de alguno de los registros de 8, 16 o 32 bits. Dependiendo de la instrucción, el registro puede codificarse en el primero o segundo operandos, o en ambos:

```
WORDX DW 12


MOV CX,WORDX          ;Registro en el primer operando

MOV WORDX, BX         ;Registro en el segundo operando



MOV CL, AH            ;Registros en ambos operandos
```

El procesamiento de datos entre registros es el tipo de operación más rápida, ya que no existe referencia a memoria.

Operandos inmediatos.

En formato inmediato, el segundo operando contiene un valor constante o una expresión constante. El campo destino en el primer operando define la longitud de los datos y puede ser un registro o una localidad de memoria. A continuación se dan algunos ejemplos:

```
ADD CX, 12 ;          ;Suma 12 al CX
MOV SAVE, 25          ;Mueve 25 a SAVE
```

2.2 Ciclos Numéricos.

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 condicionales 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.

Los ciclos numéricos que se utilizan son los siguientes: (instrucciones).

- **jmp**
- **loop**
- **cmp**
- **cmps**

Instrucción JMP

Es una instrucción basada comúnmente para la transferencia de control en un salto, es incondicional ya que la operación transfiere el control bajo cualquier circunstancia. También vacía el resultado de la instrucción previamente procesada.

Un programa con muchas operaciones de saltos puede perder velocidad de procesamiento.

El formato general para la instrucción JMP es:

```
[etiqueta] |jmp| dirección corta, cercana o lejana|
```

Ejemplo:

```
JMP EtiquetaFin  ; desde aquí saltará
mov ax,5        ; no se ejecutará
mov bx,8        ; no se ejecutará
add ax,bx       ; no se ejecutará

EtiquetaFin:           ; hasta aquí
           mov dx,1         ; sí se ejecutará
```

programa completo:

```
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.STACK 4096

option casemap:none             ; Treat labels as case-sensitive
ExitProcess PROTO, dwExitCode:DWORD

.data

.code

      main PROC

                  JMP EtiquetaFin     ; desde aqui saltara
                        mov ax,5        ; no se ejecutara
                        mov bx,8        ; no se ejecutara
                        add ax,bx       ; no se ejecutara

                  EtiquetaFin:               ; hasta aqui
                        mov dx,1        ; si se ejecutara


            Etiqueta:
                  mov eax,00
```

```
            FIN:

            INVOKE ExitProcess,0

      main ENDP

END main
```

Instrucción LOOP

La instrucción Loop requiere un valor inicial en el registro CX, en cada interación Loop de forma automática disminuye 1 de CX.

Si el valor en el CX es cero, el flujo del programa pasa a la instrucción que sigue. Si el valor en el CX no es cero, el control pasa a la dirección del operando.

La distancia debe ser un salto corto, desde -128 hasta +127 bits. Para una operación que exceda este limite, el ensamblador envia un mensaje como un salto relativo fuera de rango.

Ejemplo:

```
mov CX,10          ; Asignamos al registro contador el valor de 10 para que
LOOP genere 10 ciclos.
EtiquetaCiclo:   ; aquí comienza ciclo
    instrucción1.
    instrucción2.
    instrucción3.
     ...                                    ; en cada ciclo el registro CX
decrementa automáticamente en 1
     instrucciónN.
LOOP EtiquetaCiclo      ; mientras CX sea diferente de cero (CX <> 0),
                                          volverá   hasta   la   etiqueta.
EtiquetaCiclo:
```

Instrucción JZ

Salta hasta la etiqueta Destino si después de una operación aritmética el resultado es cero.

Ejemplo:

```
SUB op1,op2        ; se realiza la operación aritmética resta entre los
operandos op1 y op2
JZ Etiqueta        ; si el resultado es cero, entonces desde aquí saltará
mov ax,5        ; caso contrario vendrá por aquí
mov bx,8
add ax,bx
```

```
Etiqueta:         ; hasta aquí
                     mov dx,1
ejemplo2:
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.STACK 4096

option casemap:none             ; Treat labels as case-sensitive
ExitProcess PROTO, dwExitCode:DWORD

.data
       op1 dw 13

.code

       main PROC


              mov cx,5

              EtiquetaCiclo:
                     mov ax,5        ; no se ejecutará
                     mov bx,8        ; no se ejecutará
                     add ax,bx       ; no se ejecutará

                     SUB ax,op1      ; se realiza la operación aritmética resta entre los
operandos op1 y ax
                     JZ EtiquetaJZ   ; si el resultado es cero, entonces desde aquí saltará
                     mov ax, 4       ; caso contrario vendrá por aquí
                     mov bx, 100
                     add ax,bx

                     LOOP EtiquetaCiclo    ; ciclo hasta que CX=0

              EtiquetaFin:             ; hasta aquí
                     mov dx,1         ; sí se ejecutará

              EtiquetaJZ:             ; hasta aquí
                     mov dx,1



              Etiqueta:
                     mov eax,00

              FIN:
              INVOKE ExitProcess,0
       main ENDP

END main
```

Instrucción JNE

Salta hasta la etiqueta Destino si después de una comparación los operandos resultan ser diferentes.

Ejemplo:

```
CMP op1, 2     ; se realiza la comparación de los operandos op1 y op2
JNE Etiqueta    ; si los operandos son diferentes desde aquí saltará
mov ax,5          ; caso contrario vendrá por aquí
mov bx,8
add ax,bx

Etiqueta:       ; hasta aquí
mov dx,1
```

Instrucción JNZ

Salta hasta la etiqueta Destino si después de una operación aritmética el resultado es diferente de cero.

Ejemplo:

```
SUB op1,op2     ; resta entre los operandos op1 y op2
JNZ Etiqueta    ; si el resultado es diferente de cero, entonces desde aquí
saltará
mov ax,5            ; caso contrario vendrá por aquí
mov bx,8
add ax,bx

Etiqueta:       ; hasta aquí
mov dx,1        ; sí se ejecutará
```

Instrucción CMP

La instrucción CMP por lo común es utilizada para comparar dos campos de datos, uno de los cuales están contenidos en un registro.

El formato general para el CMP es:

```
    |[etiqueta]|CMP |{registro/memoria}, {registro/memoria/inmediato}|
```

Si los operandos son iguales entonces pone a la bandera ZF en cero (ZF=0 o ZR=1), si los operandos son diferentes pone a ZF en uno (ZF=1).

Ejemplo:

```
     CMP ax,cx  ; la operación que realiza es: Si(ax==bx) entonces ZF=0
si_no ZF=1;
```

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.

2.3 Captura Básica de Cadenas.

En el mundo de la programación se denomina cadena a la secuencia de caracteres y se utilizan para definir y almacenar textos. Dado que la longitud es dinámica se usan algunos bits extras con información sobre la longitud real de las cadenas o un indicador de fin de cadena.

En el leguaje ensamblador, el tipo de dato cadena (string) no está definido. Para fines de programación una cadena es definida como un conjunto de localidades de memoria consecutivas que se reservan bajo el nombre de una variable.

Instrucciones para el Manejo de Cadenas.

- **MOVSB**: Mueve un byte desde una localidad de memoria a otra.
- **MOVSW**: Mueve un palabra desde una localidad de memoria a otra.
- **LODS** (cargar un byte o palabra): carga el registro acumulador (AX o AL) con el valor de la localidad de memoria determinada por DS:SI se incrementa tras la transferencia.
- **STOS**: Almacena el contenido del registro AL o AX en la memoria.
- **CMPS**: Compara localidades de memoria de un byte o palabra.
- **SCAS**: Compara el contenido de AL o AX con el contenido de alguna localidad de memoria.

Las instrucciones para cadenas trabajan en conjunto con la instrucción CLD, la cual permite establecer que el sentido en el que las cadenas será de izquierda a derecha.

Otra instrucción importante es el prefijo de repetición REP, el cual permite que una instrucción para manejo de cadenas pueda ser repetida un número determinado de veces.

Los registros índice juegan un papel importante en el procesamiento de cadenas de datos; el par de registros CS:SI indican la dirección de la cadena original que erá procesada, y el par ES:DI contienen la dirección donde las cadenas pueden ser almacenadas.

ejemplo: programa usuando un vector para almacenar una cadena digitada por el usuario.

```
 .model small
.stack 64
.data
    inicio db 'Repetir Texto..',10,13,'$'
    ingresar db 10,13,'Ingresa tu nombre', 10,13,'$'
    nombre db 'Su nombre es: ','$'
    repetir db 10,13,'Quiere repetir s/n?',10,13,'$'
    vtext db 100 dup('$'),10,13  ;Declaracion del vector

.code
```

```
    mov ax,@data
    MOV DS,AX
    lea dx,inicio
    mov ah,09
    int 21h

    lea dx,ingresar
    mov ah,09
    int 21h   ; impr de simbolos

    ;Iniciamos nuestro conteo de si en la posicion 0.
    mov si,00h

leer:
    mov ax,0000
    mov ah,01h
    int 21h
    ;Guardamos el valor tecleado por el usuario en la posicion
si del vector.
    mov vtext[si],al
    inc si   ;Incrementamos nuestro contador
    cmp al,0dh  ;Se repite el ingreso de datos hasta que se
teclee un Enter.
    ja leer           ;ja salta mientras lo que se teclee es
menor <
    lea dx,nombre
    mov ah,09
    int 21h

ver:
    mov dx,offset vtext  ;Imprime el contenido del vector con la
misma instrucción de una cadena
    mov ah,09h
    int 21h
    lea dx,repetir  ;imprime si quiere ver de nuevo el texto
escrito.
    mov ah,09
    int 21h
    mov ah,01h
    int 21h
    cmp al,73h ;Si la tecla presiona es una s se repite la
impresión del vector.
    je ver

salir:
    mov ah,4ch
    int 21h
```

```
    end
```

2.4 Comparación y Prueba.

La instrucción **CMP** por lo común es utilizada para comparar dos campos de datos, uno o ambos de los cuales están contenidos en un registro.

El formato general para **CMP** es

```
[etiqueta:] CMP {registro/memoria},{registro/memoria/inmediato}
```

El resultado de una operación **CMP** afecta las banderas AF, CF, OF, PF, SF y ZF, aunque no tiene que probar estas banderas de forma individual. El código siguiente prueba el registro BX por un valor cero:

```
X           CMP       BX,00     ,-Compara BX con cero
            JZ        B50       ,-Si es cero salta a B50

                      (acción   es diferente de cero)
                      si


B50: ...    ...       /Destino  del salto, si BX es cero
```

Si el BX tiene cero, CMP establece el ZF a 1 y puede o no cambiar la configuración de otras banderas. La instrucción JZ (salta si BX es cero) sólo prueba la bandera ZF. Ya que ZF tiene 1 (que significa una condición cero), JZ transfiere el control (salta) a la dirección indicada por el operando B50.

Observe que la operación compara el primer operando con el segundo, por ejemplo el valor del primer operando es **mayor que**, **igual** a o **menor** que el valor del segundo operando. En el siguiente capitulo, se analizaran las diferentes formas de transferencia de control con base en condiciones probadas.

Algunas derivaciones de CMPS son las siguientes:

- **CMPSB** Compara bytes
- **CMPSD** Compara palabras dobles
- **CMPSW** Compara palabras

```
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.STACK 4096

option casemap:none             ; Treat labels as case-sensitive
ExitProcess PROTO, dwExitCode:DWORD

.data

.code

      main PROC

                  mov eax,10000h    ; EAX = 10000h

                  mov bx, 05

                  cmp bx, 00
                  jz Etiqueta

                  add eax,40000h    ; EAX = 50000h
                  sub eax,20000h    ; EAX = 30000h

            Etiqueta:
                  mov eax,00


            FIN:

            INVOKE ExitProcess,0

      main ENDP

END main
```

2.5 Saltos.

INTRODUCCIÓN

Hasta ahora los programas que hemos examinado han sido ejecutados en forma lineal, esto es, con una instrucción secuencialmente a continuación de otra. Sin embargo, rara vez un problema programable es tan sencillo. Aunque si hemos visto algunas instrucciones de saltos en el capítulo Ciclos Numéricos y en Comparación y Prueba.

La mayoría de los programas constan de varios ciclos en los que una serie de pasos se repite hasta alcanzar un requisito específico y varias pruebas para determinar qué acción se realiza de entre varias posibles.

Requisitos como éstos implican la transferencia de control a la dirección de una instrucción que no sigue de inmediato de la que se está ejecutando actualmente. Una transferencia de control puede ser hacia adelante, para ejecutar una serie de pasos nuevos, o hacia atrás, para volver a ejecutar los mismos pasos.

Ciertas instrucciones pueden transferir el control fuera del flujo secuencial normal añadiendo un valor de desplazamiento al IP. A continuación están las instrucciones introducidas en este capítulo, por categorías:

OPERACIONES DE COMPARACIÓN	OPERACIONES DE TRANSFERENCIA	OPERACIONES LÓGICAS	CORRIMIENTO Y ROTACIÓN
CMP	CALL	AND	SAR/SHR
TEST	JMP	NOT	SAL/SHL
Jnnn	OR	RCR/ROR	
LOOP	XOR	RCL/ROL	

DIRECCIONES CORTA, CERCANA Y LEJANA

Una operación de salto alcanza una dirección **corta** por medio de un desplazamiento de un byte, limitado a una distancia de -128 a 127 bytes.

Una operación de salto alcanza una dirección **cercana** por medio de un desplazamiento de una palabra, limitado a una distancia de -32,768 a 32,767 bytes dentro del mismo segmento.

Una dirección **lejana** puede estar en otro segmento y es alcanzada por medio de una dirección de segmento y un desplazamiento; **CALL** es la instrucción normal para este propósito.

La tabla siguiente indica las reglas sobre distancias para las operaciones **JMP, LOOP y CALL**. Hay poca necesidad de memorizar esta reglas, ya que el uso normal de estas instrucciones en rara ocasión causa problemas.

	Corta	Cercana	Lejana
Instrucciones	Mismo segmento -128 a 127	Mismo segmento -32,768 a 32,767	Otro segmento
JMP	sí	sí	sí
Jnnn	sí	sí: 80386 y posteriores	no
LOOP	sí	no	no
CALL	N/A	sí	sí

ETIQUETAS DE INSTRUCCIONES

Las instrucciones **JMP**, **Jnnn** (salto condicional) y **LOOP** requieren un operando que se refiere a la etiqueta de una instrucción. El ejemplo siguiente salta a *A90*, que es una etiqueta dada a una instrucción MOV:

```
                JMP         A90
A90:            MOV.        AH, 00
```

La etiqueta de una instrucción, tal como A90: terminada con dos puntos (:) para darle el atributo de cercana esto es, la etiqueta está dentro de un procedimiento en el mismo segmento de código. Un error común es la omisión de los dos puntos. Note que una etiqueta de dirección en un operando de instrucción (como JMP A20) no tiene un carácter de dos puntos.

También puede codificar una etiqueta en una línea separada como

```
A90:
                MOV.        AH, 00
```

En ambos casos, la dirección A90 se refiere al primer byte de la instrucción MOV.

LA INSTRUCCIÓN JMP

Una instrucción usada comúnmente para la transferencia de control es la instrucción **JMP** (jump, salto, bifurcación). Esta instrucción hace un salto incondicional, ya que la operación transfiere el control bajo cualquier circunstancia.

JMP vacía el resultado de la instrucción previamente procesada; por lo que, un programa con muchas operaciones de salto puede perder velocidad de procesamiento. El formato general para JMP es

```
[etiqueta:] JMP dirección corta, cercana o lejana
```

En este caso, el ensamblador genera una instrucción de máquina de dos bytes. Una JMP que excede -128 a +127 bytes se convierte en un salto cercano:

```
A50:
                    JMP         A50
```

En un salto hacia adelante, el ensamblador aún no ha encontrado el operando designado:

```
                    JMP         A90
A90:
```

Ya que algunas versiones del ensamblador no saben en este punto si el salto es corto o cercano, generan de forma automática una instrucción de tres bytes. Sin embargo, estipulando que en realidad el salto es corto se puede utilizar el operador SHORT para forzar un salto corto y una instrucción de dos bytes codificando

```
                    JMP         SHORT   A90
A90:
```

Existen diferentes tipos de saltos:

- JG salta si es mayor.
- JGE salta si es mayor o igual.
- JE salta si es igual.
- JL salta si es menor.
- JZ salta si es 0 (cero).
- JMP salta de manera incondicional.

Ejemplo de JZ

```
.model small
.stack 64
.data
    v1 dw 16
    msg1 db 'El numero es par','$'
    msg2 db 'El numero es impar','$'
    aux db 2

.code
    begin proc far
    ; guardar en el registro de segmento DS, la posici0n del segmento de datos
    mov ax,@data
    mov ds,ax

    mov ax,v1
```

```
    div aux

    cmp ah,0
    jz par
    mov ah,09
    lea dx,msg2
    int 21h
    jmp fin

    par:
        mov ah,09
        lea dx,msg1
        int 21h

    fin:
        mov ah,4ch
        int 21h

        begin endp
end
```

ejemplo usando JG

```
.model small
.stack 64
.data
msg1 db 'El numero1 es mayor','$'
msg2 db 'El numero1 es menor','$'

.code
    begin proc far

    ; guardar en el registro de segmento DS, la posición del segmento de datos
    mov ax,@data
    mov ds,ax

    mov ax,5
    mov bx,8

    cmp ax,bx
    jg mayor
    mov ah,09
    lea dx,msg2
    int 21h
    jmp fin

    mayor:
        mov ah,09
        lea dx,msg1
        int 21h

    fin:
        mov ah,4ch
        int 21h
        begin endp
```

```
end
```

2.6 Ciclos Condicionales.

El ensamblador permite usar una variedad de instrucciones de salto condicional que transfieren el control dependiendo de las configuraciones en el registro de banderas. Por ejemplo, puede comparar dos campos y después saltar de acuerdo con los valores de las banderas que la comparación establece. El formato general para el salto condicional es:

```
[etiqueta:] Jnnn dirección corta
```

La instrucción **LOOP** decrementa *CX* en 1, y transfiere el flujo del programa a la etiqueta dada como operando si CX es diferente a 0. Cuando CX sea 0 entonces el control se va a la instrucción que sigue después de la instrucción LOOP.

ejemplo:

```
mov CX,10          ; Asignamos al registro contador el valor de 10 para que LOOP genere
                   10 ciclos.
EtiquetaCiclo:     ; aquí comienza ciclo
instrucción1.
instrucción2.
instrucción3.
.......
instrucciónN.
LOOP               ; mientras CX sea diferente de cero (CX <> 0),
EtiquetaCiclo      volverá hasta la etiqueta. EtiquetaCiclo:
```

en el siguiente ejemplo se utilizan dos instrucciones para realizar un salto condicional.

```
DEC         CX          ;Equivalente a LOOP
JNZ         A20
```

ejemplo2:

```
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data

.code
main PROC

        MOV AX, 01          ; iniciar AX
        MOV BX, 01          ; iniciar BX
        MOV DX, 01          ; iniciar DX
        MOV CX, 10          ; numero de iteraciones

              ETIQ:
                            INC AX              ; incrementar 01 A AX
                            ADD BX, AX          ; sumar AX a BX
                            SHL DX,1            ; multiplicar por dos a DX
                                                ; desplazamiento a la izquierda 1 bit

                            DEC CX              ; decrementa CX
                            JCXZ FIN
                            JNZ ETIQ            ; salta si CX es no es 0

              FIN:

INVOKE ExitProcess,0
main ENDP
END main
```

ejemplo 3: tabla de multiplicar del 3 hasta el 10.

```
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
      n2 DW 3                 ; valor del multiplicando
      n1 DW 1                 ; valor inicial del multiplicador

.code
main PROC

      MOV CX, 10                      ; inicializa contador en 10

      ETI1:
            MOV AX,n1           ; contador y valor del multiplicador
            MUL N2          ; ax = ax * n2
            INC N1                  ; incrementar n2
            DEC CX              ; decrementa CX
            JCXZ FIN

            JNZ ETI1            ; regrea si CX es no es 0

      FIN:

      INVOKE ExitProcess,0
main ENDP
END main
```

Este programa si se hace en emu8086, el apuntador al segmento de datos se pierde en un principio. Es por eso que se agregan las instrucciones para crear el apuntador al segmento de datos:

```
 mov ax,@data
 mov ds,ax
```

el codigo queda:

```
; multi-segment executable file template.

data segment
    ; add your data here!
    num2 dw 4                 ; valor del multiplicando
    num1 dw 1                 ; valor inicial del multiplicado
ends

stack segment
    dw   128  dup(0)
ends

code segment

   ; guardar en el registro de segmento DS, la posición del segmento de datos
    mov ax,@data
    mov ds,ax

    MOV CX, 10                      ; inicializa contador en 10

      ETI1:
            MOV AX,num1           ; contador y valor del multiplicador

            MUL num2               ; ax = ax * n2
            INC num1               ; incrementar n2
            DEC CX                          ; decrementa CX
            JCXZ FIN

            JNZ ETI1             ; regrea si CX es no es 0

      FIN:

    ; wait for any key....
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h
ends

end
```

2.7 Incremento y Decremento.

Son las instrucciones más básicas a la hora de hacer operaciones con registros.

INC incrementa el valor de un registro, o de cualquier posición en memoria, en una unidad, y **DEC** lo decrementa.

Instrucción ***INC***

```
INC AX              ;incrementa AX en uno.

INC WORD PTR        ;Incrementa la palabra situada en CS.
```

Instrucción ***DEC***

```
DEC AX              ;decremento AX, le resta uno.

DEC WORD PTR        ;decrementa la palabra situada en CS.
```

Estas dos instrucciones, equivalentes a "a++" en C, nos servirán bastante como contadores para bucles.

Declaración de variables enteras

Cuando se declara una variable entera, el assembler reserva un lugar en la memoria para dicho entero. El nombre que se le asigna a la variable se transforma en una etiqueta que identifica ese lugar de memoria. Las siguientes directivas indican el tamaño del entero:

Tipo de dato	Bytes
BYTE , **DB** (byte)	1 (de 0 a 255)
SBYTE (byte con signo)	1 (de –128 a +127)
WORD, **DW** (word)	2 (de 0 a 65.535)
SWORD (word con signo)	2 (de –32.768 a +32.767)
DWORD, **DD** (doubleword)	4 (de 0 a 4.294.967.295)
SDWORD (doubleword con signo)	4 (de –2.147.483.648 a +2.147.483.647)

Declaración de datos

Se pueden inicializar variables cuando se las declara con constantes o con expresiones que evalúan dichas constantes. El assembler generará un error si se especificó un valor inicial que es demasiado grande para el tipo de variable. Si se utiliza un ? en lugar de una constante , entonces se está indicando que no se requiere una inicialización y el assembler reservará automáticamente un espacio de memoria pero no escribirá ningún valor en él. Se pueden declararar e inicializar variables en un solo paso como se muestra a continuación:

```
entero      BYTE      16          ; inicializa un byte en 16
negativo    SBYTE     -16         ; inicializa un byte con signo en -16
expression  WORD      4*3         ; inicializa un word en 12 a través de la expresión 4*3
positivo    SWORD     4*3         ; inicializa un worq con signo en 12
vacio       DWORD     ?           ; declara una variable dword pero no la inicializa
lista       BYTE      1,2,3,4,5,6 ; inicializa 6 bytes
```

Una vez que se declararon las variables en el programa, se pueden realizar diferentes operaciones con ellas como copiar, mover, sumar, restar, multiplicar y dividir. Puede ocurrir que sea necesario trabajar con un tamaño de dato distinto del que fue declarado originalmente. Como las instrucciones de MASM requieren que los operandos sean del mismo tamaño, se puede realizar cualquiera de las operaciones mencionadas, utilizando el operador PTR . Por ejemplo, se puede usar PTR para acceder a la parte más significativa de una variable de tamaño DWORD. La sintaxis para este operador es la siguiente: `tipo de dato` **`PTR`** `expresión`

donde el operador PTR fuerza a expresión a ser tratada como si fuera del tipo de dato especificado. Un ejemplo de su uso es el sgte:

```
.DATA
num    DWORD   0
.CODE
mov    AX, WORD PTR num[0]   ; carga en AX  los 2 bytes menos significativos de num
mov    DX, WORD PTR num[2]   ; carga en DX los 2 bytes más significativos de num
```

Definición de datos

DB - Definir bytes

DW - Definir words

DD - Definir doublewords

DQ - Definir quadwords

DT - Definir grupos continuos de 16 bytes

.DATA marca el inicio del segmento de datos. En este segmento deberán colocarse las variables de memoria. Por ejemplo:

```
...
.DATA
ErrorMessage DB 0Dh,0Ah,'*** Error ***',0Dh,0Ah,'$'
...
.CODE
...
```

.DATA presenta cierta complejidad en el código del programa, debe cargarse explícitamente el registro DS con el símbolo @data, antes de que se realicen accesos a localizaciones de memoria existentes en el segmento definido por .DATA. Puesto que un registro de segmento puede ser cargado con el contenido de un registro de propósito general o una localización de memoria, pero no con una constante, el registro de segmento es generalmente cargado con una secuencia de dos instrucciones:

```
...
mov   ax,@data
mov   ds,ax
...
```

Esta secuencia de instrucciones inicializa DS para que apunte al segmento de datos que inicia con la directiva .DATA.

Sin las dos instrucciones que inicializan el registro DS con el segmento definido por .DATA, las variables de memoria que residen en el segmento .DATA no podrán ser accesadas a menos que DS apunte a este segmento.

Los registros de segmento CS y SS son inicializados por el DOS cuando un programa inicia, por lo tanto no deberán ser cargados explícitamente como se hace con el registro DS.

Mientras que CS apunta a las instrucciones y SS apunta a la pila, DS apunta a los datos. Los programas no pueden manipular directamente el código y la pila; pero trabajan constantemente con los datos de manera directa. Además, los programas pueden tener datos en diferentes segmentos a la vez; recuerde que el 8086 permite accesar cualquier localidad de memoria en un rango de 1 MB, pero únicamente en bloques de 64KB a la vez (un registro de segemento). Por lo tanto, al principio el programador querrá iniciar el registro DS con un segmento, accesar datos en ese segmento y depués cargar DS con otro segmento para accesar un bloque diferente de datos. Este esquema es utilizado generalmente en los programas más grandes de lenguaje ensamblador.

```
... .DATA
DB 'A' ; Definir un byte. DB 65
DB 41h
DB 101o
DB 1000001b
DB 'ABCDE' ; Definir una cadena de bytes
DB 'A','B','C','D','E'
...
```

El ensamblador almacena los caracteres en formato ASCII, sin apóstrofes. Si la cadena debe contener un apóstrofes o una comilla, usted puede definirlo en una de las forma siguientes:

```
DB "Honest Ed's PC Emporium"    ;Comillas para la cadena
                                ;una comilla para el apóstrofe

DB 'Honest Ed''s PC Emporium' ; Una comilla para la cadena,
                                ;dos comillas seguidas para el apóstrofe
```

2.8 Captura de Cadenas con Formato.

(leer capitulo 12 Operaciones con cadenas)

Las cadenas de caracteres son usadas para datos descriptivos como nombres de personas, etc. La cadena está definida dentro de apostrofes como 'PC'. El ensamblador traduce las cadenas de caracteres en código objeto en formato ASCII normal.

El capturar cadenas con formato permite el movimiento, comparación o búsqueda rápida entre bloques de datos, las instrucciones son las siguientes:

MOVS
Mueve un byte, palabra o palabra doble desde una localidad en memoria a otra.

LODS
Carga desde memoria un byte en el *AL*, una palabra en el *AX* o una palabra doble en el *EAX*. (apuntador)

CMPS
Compara localidades de memoria de un byte, palabra o palabra doble.

SCAS
Compara el contenido de *AL*, *AX* o *EAX* con el contenido de una localidad de memoria.

Una instrucción de cadena puede especificar el procesamiento repetitivo de un byte, palabra o (en el 80386 y procesadores posteriores) palabra doble a un tiempo.

Cada instrucción de cadena tiene una versión para byte, palabra o palabra doble y supone el uso de los registros ***ES:DI*** o ***DS:SI.***
Donde: ***DI*** y ***SI*** deben contener direcciones de desplazamiento válidas.

Básicamente existen dos maneras de codificar instrucciones de cadena. En la tabla siguiente, la segunda columna muestra el formato básico para cada operación, la cual utiliza los operandos implicados listados en la tercer columna (por ejemplo, si codifica una instrucción MOVS, incluya operandos como MOVS BYTE1 ,BYTE2, en donde la definición de los operandos indican la longitud del movimiento):

Operación	Instrucción básica	Operandos implicados	Operación con bytes	Operación con palabra	Operación con palabra doble
Mover	MOVS	ES:DI,DS:SI	MOVSB	MOVSW	MOVSD
Cargar	LODS	AX,DS:SI	LODSB	LODSW	LODSD
Almacenar	STOS	ES:DI,AX	STOSB	STOSW	STOSD
Comparar	CMPS	DS:SI,ES:DI	CMPSB	CMPSW	CMPSD
Rastrear	SCAS	ES:DI,AX	SCASB	SCASW	SCASD

La segunda manera de codificar instrucciones de cadena es la práctica usual, como se mostró en las columnas cuarta, quinta y sexta. Usted carga las direcciones de los operandos en los registros ***DI*** y ***SI*** y codifica, por ejemplo, *MOVSB, MOVSW* y *MOVSD* sin operandos.

Las instrucciones de cadena suponen que el ***DI*** y el ***SI*** contienen direcciones de desplazamiento válidas que hacen referencia a bytes en memoria.
El registro ***SI*** está asociado por lo común con el ***DS*** (segmento de datos) como ***DS:SI.***
El registro ***DI*** siempre está asociado con el registro ***ES*** (segmento extra) como ***ES:DI.***

REP: **Prefijo de Repetición de Cadena**
El prefijo REP inmediatamente antes de una instrucción de cadena, como REP MOVSB, proporciona una ejecución repetida con base en un contador inicial que se debe establecer en el registro CX. REP ejecuta la instrucción de cadena, disminuye el CX y repite la operación hasta que el contador en el CX sea cero. De esta manera, puede manejar cadenas de caracteres de casi cualquier longitud.

La bandera de dirección (DF) determina la dirección de la operación que se repite:

- Para procesamiento de izquierda a derecha (la manera normal de procesar), utilice CLD para poner en cero a DF.
- Para procesamiento de derecha a izquierda, utilice STD para poner uno en DF.

MOVS (MOVSB) (MOVSW)
Mueve cadenas de bytes o palabras desde la fuente, direccionada por SI, hasta el destino direccionado por DI.

Sintaxis:
Este comando no necesita parámetros ya que toma como dirección fuente el contenido del registro SI y como destino el contenido de DI.

Ejemplo:
```
MOV SI, OFFSET VARIABLE1  ; se inicializa valor de SI con offset de variable1
MOV DI, OFFSET VARIABLE2  ; se inicializa valor de DI con offset de variable2
MOVS  ; se copia el contenido de VARIABLE1 a VARIABLE2.
```

OFFSET, devuelve el desplazamiento de la variable o etiqueta especificada. El desplazamiento es la posición desde el principio del segmento hasta la expresión indicada.

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

LODS (LODSB) (LODSW)
Carga cadenas de un byte o palabra al acumulador.

Sintaxis: LODS
Toma la cadena que se encuentre en la dirección especificada por SI, la carga al registro AL (o AX) y suma o resta 1 (segun el estado de DF) a SI si la transferencia es de bytes o 2 si la transferencia es de palabras.

Ejemplo:
MOV SI, OFFSET VARABLE1
LODS

La primer linea carga la dirección de VARIABLE1 en SI y la segunda linea 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).

LAHF Transfiere al registro AH el contenido de las banderas

Sintaxis: LAHF
Se utiliza para verificar el estado de las banderas durante la ejecución de un programa.

Las banderas quedan en el siguiente orden dentro del registro:
SF ZF __ AF __ PF __ CF

LEA 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.

Ejemplo:
MOV SI, OFFSET VAR1

Que es equivalente a:
LEA SI, VAR1

Ejemplo usando interrupciones. Programar en emulador DosBOX.
NO se pueden usar interrupciones en Windows.
En el ejemplo se definen dos variables y se les da un valor de texto, luego se utiliza MOVS para copiar la cadena 1 a la cadena2

```
.model small
.stack 64
.data
             cad1 db 'Esta es la cadena1','$'
      cad2 db 'Esta es la cadena2','$'


.code
inicio:

      mov ax, @data    ;inicia registros
      mov ds, ax       ;de segmento
      mov es, ax

      cld              ;Procesamiento de cadenas de izq->der.
      mov cx,18        ;longitud de la cadena original
      lea di,cad2      ;ES:DI contienen la direccion de Cad2
      lea si,cad1      ;DS:SI contienen la direccion de Cad1

      rep movsb             ;DS:SI->ES:DI, SI=SI+1, DI=DI+1

      lea dx,cad1      ;Imprimir Cad1 en pantalla
      mov ah,09h            ;
      int 21h          ;

      lea dx,cad2      ;Imprimir Cad2 en pantalla
      mov ah,09h            ;
      int 21h          ;


Salir:
mov ah,04ch
int 21h

end
```

El programa debe copiar Cad3 dentro de Cad1 usando 18 repeticiones con MOVSB, después realiza lo mismo con Cad4 y Cad2 pero usando solo nueve repeticiones de la instrucción MOVSW.

```
.model small
.stack 64
.data
    cad1 db 'Cadena de prueba1 ','$'
    cad2 db 'Cadena de prueba2 ','$'
    cad3 db 18 dup ('d')
    cad4 db 18 dup ('l')

.code
inicio:

    mov ax, @data     ;inicia registros
     mov ds, ax       ;de segmento
     mov es, ax

    cld              ;procesamiento de izq->der.
    mov cx,18        ;Longitud de la cadena
    lea si,cad3      ;DS:SI->Cad3
    lea di,cad1      ;ES:DI->Cad1
    rep movsb        ;Cad3->Cad1

    mov cx,9         ;Longitud de la cadena por pares de bytes
    lea si,cad4      ;DS:SI->Cad4
    lea di,cad2      ;ES:DI->Cad2
    rep movsw        ;Cad4->Cad2

    lea dx,cad1      ;
    mov ah,09h       ;Imprimir Cad1
    int 21h          ;

    lea dx,cad2      ;
    mov ah,09h       ;Imprimir Cad2
    int 21h          ;

Salir:
mov ah,04ch
int 21h

end
```

El programa invierte el orden de los elementos de una cadena y los almacena en otra cadena que originalmente esta inicializada con espacios. Al final se imprimen las dos cadenas.

```
.model small
.stack 64

.DATA
    cad1 db 'Cadena de prueba','$'
    cad2 db 16 dup (' '),'$'

.CODE
inicio: ;Punto de entrada al programa

    mov ax, @data     ;inicia registros
      mov ds, ax      ;de segmento
      mov es, ax

    cld             ;Procesamiento de izq->der.
    mov cx,16       ;Longitud de la cadena
    lea si,cad1     ;DS:SI->Cad1
    lea di,cad2+15  ;ES:DI apuntan al final del área reservada para
                    ;almacenar la cadena invertida
otro:               ;
    lodsb           ;Obtener el primer carácter de Cad1
    mov [di],al     ;almacenarlo en la posición actual de DI
    dec di          ;Disminuir DI
    loop otro       ;Obtener siguiente carácter de Cad1

    lea dx,cad1     ;
    mov ah,09h      ;Imprimir cadena original
    int 21h         ;

    lea dx,cad2     ;
    mov ah,09h      ;Imprimir cadena invertida
    int 21h         ;

    mov ax,4c00h    ;Terminar programa y regresar al DOS
    int 21h         ;

    END inicio
END
```

Este programa utiliza la instrucción STOSB para rellenar un registro de memoria con el contenido del registro AL.
En este caso, el área de memoria reservado para la variable Cad1 es rellenada con el carácter ASCII '*'.

```
.model small
.stack 64
.data
    cad1 db 'Cadena de prueba',13,10,'$'

.code
inicio:

     mov ax, @data    ;inicia registros
     mov ds, ax       ;de segmento
     mov es, ax

     lea dx,cad1 ;Imprimir Cad1 antes de que sea borrada
    mov ah,09h  ;
    int 21h     ;

    cld         ;Procesamiento de izq->der
    mov al,'*'  ;Inicializar AL con '*'
    mov cx,16   ;Longitud de la cadena que se va a rellenar
    lea di,cad1 ;ES:DI->Cad1
    rep stosb   ;Rellenar 16 bytes de memoria con '*'

    lea dx,cad1 ;
    mov ah,09h ;Imprimir Cad1 después de ser borrada
    int 21h ;

Salir:
mov ah,04ch
int 21h

end
```

Descripción: Este programa utiliza la instrucción CMPSB para comparar cadena 1 con otras dos cadenas. En este programa se declaran 3 cadenas de prueba.

El registro BH sirve como bandera
BH=0 No hay cadenas iguales
BH=1 Cad1 es igual a Cad2
BH=2 Cad1 es igual a Cad3

Se puede cambiar el contenido de las cadenas de prueba para comprobar los tres posibles resultados.

```
.model small
.stack 64
.data
    cad1 db 'CADENA1',13,10,'$'
    cad2 db 'CADENA2',13,10,'$'
    cad3 db 'CADENA3',13,10,'$'
    error1 db 'No hay cadenas iguales...','$'
    error2 db 'Cadena 1 = Cadena 2','$'
    error3 db 'Cadena 1 = Cadena 3','$'


.code
inicio:
     mov ax, @data       ;inicia registros
     mov ds, ax          ;de segmento
     mov es, ax

     xor bh,bh           ;BH=0
    cld                 ;Comparación de izq->der.
    mov cx,7            ;Longitud de la cadena
    lea di,cad2         ;ES:DI-> Cad2
    lea si,cad1         ;DS:SI-> Cad1
    repe cmpsb          ;Comparar Cad1 y Cad2
    jne otra            ;Son iguales ?No, Comparar Cad1 y Cad3
    mov bh,1            ;Si, entonces BH=1

    otra:
        mov cx,7        ;Longitud de la cadena
        lea di,cad3     ;ES:DI->Cad3
        lea si,cad1     ;DS:SI->Cad1
        repe cmpsb      ;Comparar Cad1 y Cad3
        jne salir       ;Son iguales ?No, imprimir mensajes.
        mov bh,2        ;Si, entonces BH=2

    salir:
        cmp bh,0        ;Es BH=0?
        je ninguna      ;Si, Entonces no hay cadenas iguales
        cmp bh,1        ;No. Es BH=1?
        je cad1_cad2    ;Si. Entonces Cad1 es igual a Cad2
        lea dx,error3   ;Si no es ninguna de las anteriores
        mov ah,09h      ;entonces debe ser que Cad1 es igual que Cad3
```

```
        int 21h         ;imprimir mensaje
        jmp Fin

    cad1_cad2:
        lea dx,error2   ;
        mov ah,09h      ;Imprimir mensaje
        int 21h         ;
        jmp Fin         ;

    ninguna:
        lea dx,error1   ;
        mov ah,09h      ;Imprimir mensaje
        int 21h         ;

    Fin:
        mov ah,04ch     ;Terminar programa y regresar al DOS
        int 21h         ;

end
```

este programa no compara cadena 2 con cadena 3 y tampoco se valida si las tres cadenas son iguales.

2.9 Instrucciones Aritméticas.

ADD Suma los dos operandos y guarda el resultado en el operando destino.
ADD permite realizar la suma de dos números enteros a nivel de bits.
Sintaxis: ADD destino, fuente
Ejemplo:
```
ADD ah,bh       ; la operación que realiza es: AH = AH + BL
ADD ax,bx       ; la operación que realiza es: AX = AX + BX
ADD eax,ebx     ; la operación que realiza es: EAX = EAX + EBX
```

ADC Adición con acarreo, es decir lleva a cabo la suma de dos operanos y suma 1 al resultado en caso de que la bandera CF este activa (existe acarreo).

El resultado se guarda en el operando destino.
Sintaxis: ADC destino, fuente
Ejemplo:
```
ADC ah,bl       ; la operación que realiza es: AH = ( AH + BL ) + CF
ADC ax,bx       ; la operación que realiza es: AX = ( AX + BX ) + CF
ADC eax,ebx     ; la operación que realiza es: EAX = ( EAX + EBX ) + CF
```

SUB Resta el operando fuente del destino (trabaja con números enteros).
Sintaxis: SUB destino, fuente
Ejemplo:
```
SUB ah,bh       ; la operación que realiza es: AH = AH - BL
SUB ax,bx       ; la operación que realiza es: AX = AX - BX
SUB eax,ebx     ; la operación que realiza es: EAX = EAX - EBX
```

SSB Resta los operandos y resta uno al resultado si CF está activado.
El operando fuente siempre se resta del destino.
Este tipo de substracción se utiliza cuando se trabaja con cantidades de 32 bits.
Sintaxis: SBB destino, fuente
Ejemplo:
```
SSB ah,34h      ; la operación que realiza es: AH = ( AH – 34h ) - CF
SSB ax,bx       ; la operación que realiza es: AX = ( AX - BX ) - CF
SSB eax,ebx     ; la operación que realiza es: EAX = ( EAX - EBX ) - CF
```

Ambas operaciones aritméticas tanto suma como resta se pueden realizar entre:

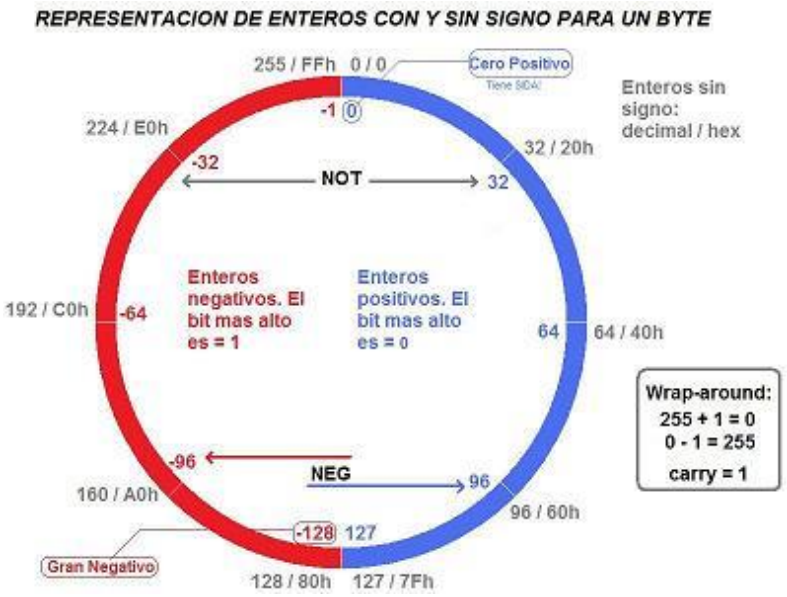
- Dos registros.
- Un registro y una ubicación de memoria.
- Una ubicación de memoria y un registro.
- Un registro y una constante.
- Una ubicación de memoria y una constante.

Las instrucciones ADD y SUB no distinguen ente datos con y sin signo: en realidad, sólo suman y restan bits.

En registros de 8 bits (2^8 =256):
Números sin signo.- Estos son los de siempre. Cuenta de cero a 255.
Números con signo.- Con los mismos 8 bits de un byte, podemos representar los números desde el -128 al 127. Si se fijan, estos son 256 valores diferentes.

Para poder representar un número negativo por ejemplo -10 en un registro de 8 bits:

1. 256 – 10 = 246 (en 16 bits → 65,536 en 32 bits→ 4,294,967,296)
2. 246 en Hex = F6



Estar alerta con los desbordamientos en las operaciones aritméticas. Ya que un byte sólo permite el uso de un bit de signo y siete bits de datos (desde -128 hasta +127), una operación aritmética, puede exceder con facilidad la capacidad de un registro de un byte. Y una suma en el registro AL que exceda su capacidad puede provocar resultados inesperados. Por ejemplo, suponga que el AL contiene 60H. Entonces la instrucción

ADD AL, 20H

genera una suma de 80H en el AL. Como hemos sumado dos números positivos, esperamos qué la suma sea positiva, pero la operación pone en uno la bandera de desbordamiento y la bandera de signo en negativa. ¿La razón? El valor 80H, o 10000000 binario, es un número negativo; en lugar de +128 la suma es -128. El problema es que el registro AL es muy pequeño para la suma, que debe estar en el registro AX completo.

emu8086:

```
.model small
```

```
.stack 64
.data

.code

            mov ah, 60h
            add ah, 20h

    Salir:
        mov ah,04ch
        int 21h

end
```

En la división tenga cuidado especial del desbordamiento. El divisor debe ser mayor que el contenido del AH si el divisor es un byte, que el DX si el divisor es una palabra, o que el EDX si el divisor es una palabra doble.

MUL Multiplicación sin signo

El ensamblador asume que el multiplicando será del mismo tamaño que el multiplicador, Para multiplicar dos números de un byte, el multiplicando está en el registro **AL** y el multiplicador es un byte en memoria o en otro registro. Para la Instrucción MUL DL, la operación multiplica el contenido del AL por el contenido del DL. El producto generado está en el registro AX. La operación ignora y borra cualquier información que pueda estar en el AH.

	AH	AL
Antes de multiplicar:		**Multiplicando**
Después de multiplicar:	AX ^——————**Producto**——————^	

Para multiplicar dos números de una palabra, el multiplicando está en el registro AX y el multiplicador es una palabra en memoria o en otro registro. Para la instrucción MUL DX, la operación multiplica el contenido del AX por el contenido del DX. El producto generado es una palabra doble que necesita dos registros: la parte de orden alto (más a la izquierda) en el DX y la parte de orden bajo (más a la derecha) en el AX. La operación ignora y borra cualquier información que pueda estar en el DX.

	DX	AX
Antes de multiplicar:	**Ignorado**	**Multiplicando**
Después de multiplicar:	**Parte alta de producto**	**Parte baja de producto**

Para multiplicar dos números de palabras dobles, el multiplicando está en el registro EAX y el multiplicador es una palabra doble en memoria o en otro registro. El producto es generado en el par EDX:EAX. La operación ignora y borra cualquier información que ya esté en el EDX.

	EDX	EAX
Antes de multiplicar:	**Ignorado**	**Multiplicando**
Después de multiplicar:	**Parte alta de producto**	**Parte baja de producto**

El operando de MUL o IMUL sólo hace referencia al multiplicador, que determina el tamaño del campo. En los ejemplos siguientes, el multiplicador está en un registro, el cual especifica el tipo de operación:

INSTRUCCIÓN	MULTIPLICADOR	MULTIPLICANDO	PRODUCTO
MUL CL	byte	AL	AX
MUL BX	palabra	AX .	DX:AX
MUL EBX	palabra doble	EAX	EDX: EAX

Sintaxis: MUL fuente

Ejemplo: MUL DL ; la operación que realiza es: AX = AL * DL
MUL SI ; la operación que realiza es: DX:AX = AX * SI
MUL ESI ; la operación que realiza es: EDX:EAX = EAX * ESI

IMUL Multiplicación de dos enteros con signo.
Este comando hace lo mismo que el anterior, toma en cuenta los signos de las cantidades que se multiplican.
Los resultados se guardan en los mismos registros que en la instrucción MUL.

Sintaxis: IMUL fuente

Ejemplo: IMUL BL ; la operación que realiza es: AX = AL * BL
IMUL SI ; la operación que realiza es: DX:AX = AX * SI
IMUL ESI ; la operación que realiza es: EDX:EAX = EAX * ESI

DIV División sin signo

Las operaciones de división básicas son palabra entre byte, palabra doble entre palabra y (para 80386 y posteriores) palabra cuádruple entre palabra doble.

Palabra entre byte

Aquí, el dividendo está en el AX y el divisor es un byte en memoria o en otro registro. Después de la división, el residuo está en el AH y el cociente está en el AL. Ya que un cociente de un byte es muy pequeño —si es sin signo, un máximo de +255 (FFH) y con signo +127 (7FH)— esta operación tiene un uso limitado.

Antes de la división:	**AX**	
	- Dividendo -	
Después de la división:	**AH**	**AL**
	Residuo	**Cociente**

Palabra doble entre palabra

Para esta operación, el dividendo está en el par DX: AX y el divisor es una palabra en memoria o en otro registro. Después de la división, el residuo está en el DX y el cociente está en el AX. Tenemos:

Antes de la división:	**DX**	**AX**
Después de la división:	**Parte alta del dividendo**	**Parte baja del dividendo**
	Residuo	**Cociente**

Palabra cuádruple entre palabra doble

Al dividir una palabra cuádruple entre una palabra doble, el dividendo está en el par EDX:EAX y el divisor está en una palabra doble en memoria o en otro registro. Después de la división, el residuo está en el EDX y el cociente en el EAX.

Antes de la división:	**EDX**	**EAX**
Después de la división:	**Parte alta del dividendo**	**Parte baja del dividendo**
	Residuo	**Cociente**

Sintaxis: DIV fuente

Ejemplo:

```
DIV DL    ; la operación que realiza es: AL = AX div DL          AH = AX mod DL
DIV SI    ; la operación que realiza es: AX = DX:AX div SI       DX = DX:AX mod SI
DIV ESI   ; la operación que realiza es: EAX = EDX:EAX div ESI   EDX = EDX:EAX mod ESI
```

IDIV División con signo

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.

Sintaxis: IDIV fuente

Ejemplo:

```
IDIV BL    ; la operación que realiza es: AL = AX div BL          AH = AX mod BL
IDIV SI    ; la operación que realiza es: AX = DX:AX div SI       DX = DX:AX mod SI
IDIV ESI   ; la operación que realiza es: EAX = EDX:EAX div ESI   EDX = EDX:EAX mod ESI
```

El operando de DIV o de IDIV hace referencia al divisor, que especifica el tamaño del campo. En los ejemplos siguientes de DIV, los divisores están en un registro, que determina el tipo de operación:

OPERACIÓN	DIVISOR	DIVIDENDO	COCIENTE	RESIDUO
DIV CL	byte	AX	AL	AH
DIV CX	palabra	DX:AX	AX	DX
DIV EBX	palabra doble	EDX: EAX	EAX	EDX

ejemplo:

```
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data

.code
main PROC

            ; suma de registros de 8 bits
            MOV AH, 255
            MOV BL, 10
            ADD AH, BL

            ; suma de registros de 16 bits
            ;MOV AX, 5
            ;MOV BX, 10
            ;ADD AX, BX

            ; suma de registros de 32 bits
            ;MOV EAX, 5
            ;MOV EBX, 10
            ;ADD EAX, EBX


            ; suma con acarreo de registros de 8 bits
            ;MOV AH, 255
            ;MOV BL, 1
            ;ADD AH, BL   ; activar bandera de acarreo

            ;ADC AH, BL
            ;MOV AH, 254

            ; suma con acarreo de registros de 16 bits
            ;MOV AX, 100
            ;MOV BX, 100

            ;ADC AX, BX

            ; suma con acarreo registros de 32 bits
            ;MOV EAX, 5
            ;MOV EBX, 10
            ;ADC EAX, EBX


            ; resta de registros de 8 bits
            ;MOV AH, 55
            ;MOV BL, 10
```

```
;SUB AH, BL

; resta de registros de 16 bits
;MOV AX, 55
;MOV BX, 10
;SUB AX, BX

; resta de registros de 32 bits
;MOV EAX, 55
;MOV EBX, 10
;SUB EAX, EBX

; resta de registros y acarreo de 8 bits

;MOV AH, 255
;MOV BL, 1
;ADD AH, BL
;
;MOV AH, 55
;ssb AH, bl

; resta de registros y acarreo de 16 bits
;MOV AX, 65534
;MOV BX, 1
;SSB AX, BX


; resta de registros y acarreo de 32 bits
;MOV EAX, 55
;MOV EBX, 10
;SSB EAX, EBX

; resta de registros de 8 bits
;MOV AH, 55
;MOV BL, 10
;SUB AH, BL

; multiplicacion de registros de 8 bits, resultado en AX
;MOV AL, 5
;MOV DL, 2
;MUL DL

; multiplicacion de registros de 16 bits, resultado en par DX:AX
;MOV AX, 5000
;MOV SI, 20
;MUL SI

; multiplicacion de registros de 32 bits, resultado en EDX:EAX
;MOV EAX, 5000
;MOV ESI, 20
;MUL ESI

; IMUL multiplicacion  de registros de 8 bits, resultado en AX
;MOV AL, 5
;MOV DL, 2
;IMUL DL
```

```
            ; IMUL multiplicacion de registros de 16 bits, resultado en par
DX:AX
            ;ax:dx = 0000 E000
            ;MOV AX, -80H
            ;MOV SI, 40H
            ;IMUL SI

            ; IMUL multiplicacion de registros de 16 bits, resultado en par
DX:AX
            ; dx:ax = F000 0000
            ;MOV AX, -8000H
            ;MOV SI, 2000H
            ;IMUL SI

            ; multiplicacion de registros de 32 bits, resultado en EDX:EAX
            ; EDX:EAX  FFFFFFFF F0000000
            ;MOV EAX, -8000H
            ;MOV ESI, 2000H
            ;IMUL ESI

            ; División  de registros de palabra /Byte, cociente en AL residuo
en AH
            ;MOV AX, 4
            ;MOV DL, 2
            ;DIV DL
            ;conciente AL 2   residuo AH 0

            ; División  de registros de palabra /Byte, cociente en AL residuo
en AH
            ;MOV AX, 5
            ;MOV DL, 2
            ;DIV DL
            ;conciente AL 2   residuo AH 01


            ; División  de registros de Palabra / Byte, cociente en AX
residuo en DX
            ;MOV AX, 2000h
            ;MOV DL, 80H
            ;DIV DL
            ; cociente AL 40

            ; División  de registros de Palabra / Byte, cociente en AX
residuo en DX
            ;MOV AX, 2010h
            ;MOV DL, 80H
            ;DIV DL
            ; cociente AL 40  residuo AH 10

            ; División  de registros de Palabra Doble / Palabra, cociente en
AX residuo en DX
            ;MOV DX, 10H
            ;MOV AX, 1000h
            ;MOV DL, 2000H
            ;DIV DL
            ; cociente AX 40
```

```
            ; División  de registros de Palabra Doble / Palabra, cociente en
AX residuo en DX
            ;MOV DX, 10H
            ;MOV AX, 1030h
            ;MOV SI, 2000H
            ;DIV SI
            ; cociente AX 80  reciduo DX 1021

            ; División  de registros Palabra Quadruple / Palabra Doble,
cociente en EAX residuo en EDX
            ;MOV EDX, 1030H
            ;MOV EAX, 10001000H
            ;MOV SI, 2000H
            ;DIV SI
            ; cociente EAX = 81808000   reciduo EDX = 00001000


            FIN:

INVOKE ExitProcess,0
main ENDP
END main
```

2.10 Manipulación de la Pila.

La pila es un grupo de localidades de memoria que se reservan para contar con un espacio de almacenamiento temporal cuando el programa se está ejecutando. La pila en tiempo de ejecución es un arreglo de memoria que la CPU administra directamente, mediante el uso de dos registros: SS y SP.

SS: registro del segmento de la pila.
SP: registro apuntador de la pila (16bits).

En modo protegido, el registro SS guarda un apuntador a un descriptor de segmento y no se modifica en los programas de usuario. El registro ESP guarda un desplazamiento de 32 bits hacia alguna ubicación en la pila. Muy raras veces es necesario manipular el registro ESP en forma directa; en vez de ello, se modifica de manera indirecta mediante las instrucciones tales como CALL, RET, PUSH y POP.

La pila es una estructura de datos de tipo LIFO (Last In First Out), esto quiere decir que el último dato que es introducido en ella, es el primero que saldrá al sacar datos de la pila.

Para la manipulación de la pila ensamblador cuenta con dos instrucciones especificas:

Push:
Esta instrucción permite almacenar el contenido del operando dentro de la última posición de la pila.

Ejemplo:
Push ax El valor contenido en ax es almacenado en el último espacio de la pila.

Pop:
Esta instrucción toma el último dato almacenado en la pila y lo carga al operando.

Ejemplo:
Pop bx El valor contenido en el último espacio de la pila se almacena en el registro

ejemplo: Programa: PushPop.ASM
Descripción: Este programa demuestra el uso de las instrucciones para el manejo de la pila, implementando la instrucción **Push** y **Pop**

```
.386
```

```
.model flat,stdcall
.STACK 4096

option casemap:none             ; Treat labels as case-sensitive
ExitProcess PROTO, dwExitCode:DWORD

.code

       main PROC

              Mov AX,5            ;AX=5
              Mov BX,10           ;BX=10
              Push AX             ;Pila=5
              Mov AX,BX           ;AX=10
              Pop BX              ;BX=5
              Mov AX,4C00h        ;Terminar programa y salir al DOS

              FIN:
              INVOKE ExitProcess,0
       main ENDP
END main
```

Cada elemento de dato en la pila es una palabra (dos bytes). El registro **SS**, como es inicializado por el DOS, contiene la dirección del inicio de la pila. Inicialmente, el **SP** contiene el tamaño de la pila.

La pila difiere de otros segmentos en su método de almacenar los datos: empieza en la localidad más alta y almacena los datos hacia abajo por la memoria.

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

La instrucción **PUSH**; disminuye el ***SP*** en 2 hacia abajo, hacia la siguiente palabra almacenada de la pila y coloca (o empuja, **push**) un valor ahí.

La instrucción **POP**; regresa el valor de la pila e incrementa el ***SP*** en 2 hacia arriba, hacia la siguiente palabra almacenada.

El ejemplo siguiente ilustra cómo meter el contenido de los registros **AX** y **BX** a la pila y la subsecuente extracción de ellos.

Suponga que:

- AX = 015AH
- BX = 03D2H
- SP = 28H

1. Al comienzo, la pila está vacía y se ve así: SS SP = A30
(dirección del segmento de la pila tope de la pila)

PUSH AX: disminuye el SP en 2 (A2E) y almacena el contenido del AX (015AH), en la pila.

3. PUSH BX: disminuye el SP en 2 (a A2CH) y almacena el contenido del BX, 03D2H,

POP BX: regresa la palabra que se encuentra en la pila, en donde apunta el SP, y la envía al registro BX e incrementa el SP en 2 (a A2EH).

```
; Author:
; Program Name:
; Program Description:
; Date:

.386
.model flat,stdcall
.STACK 4096

option casemap:none             ; Treat labels as case-sensitive
ExitProcess PROTO, dwExitCode:DWORD

.data

.code

      main PROC

            Mov AX,015AH
            Mov BX,03D2H


            Push AX              ;disminuye el SP en 2 y almacena en la pila el
contenido de AX
            PUSH BX             ;disminuye el SP en 2 y almacena en la pila el
contenido del BX
            POP AX
            POP BX

            FIN:

            INVOKE ExitProcess,0
      main ENDP

END main
```

Siempre debe asegurarse que su programa coordine los valores que guarda en la pila con los valores que saca de ella. Como éste es un requisito directo, un error puede causar que un programa no funcione. También, para un programa .EXE usted tiene que definir una pila que sea suficientemente grande para contener todos los valores que podrían ser guardados en ella.

Otras instrucciones relacionadas con los valores que guarda y saca de la pila son:
• **PUSHF** y **POPF**: Guarda y restablece el estado de los banderas.
• **PUSHA** y **POPA** (para el 80286 y posteriores): Guarda y restaura el contenido de todos los registros de propósito general.

PUSH

La instrucción PUSH siempre transfiere 2 bytes de información a la pila. Cuando un dato es metido a la pila, el primer byte (mas significativo) es almacenado en la pila en la localidad direccionada por SP-1. El segundo byte (menos significativo) es almacenado en la pila en la localidad direccionada por SP-2. Después de que el dato se a empujado a la pila, el registro SP es decrementado en 2. La Figura 1 muestra la ejecución de la instrucción PUSH AX, la cual transfiere el contenido de AX en la pila ([SP-1]=AH, [SP-2]=AL y SP=SP-2).