# Introducción al ensamblador 80x86

Por: Sadot Alexandres Fernández
Colaboran: Carlos Rodríguez Morcillo, Javier Goyanes.
Departamento de Electrónica y Automática
Escuela Técnica Superior de Ingeniería
Universidad Pontificia Comillas de Madrid

Febrero de 2004.

## Índice



Referencias recomendadas al texto:

- Gavin Estey, 1996. http://burks.brighton.ac.uk/burks/language/asm/asmtut/asm1.htm
- “Borland Turbo Assembler Quick Referente”. Borland.
- "Intel Pentium Family User's Manual”: Volumen 3.
- URL de Borland: http://www.borland.com
- URL de Intel: http://www.intel.com
- “The Revolutionary Guide to Assembly Language”. Vitaly Maljugin, Jacov Izrailevich et al Wrox Press. ISBN: 1-874416-12-5

# I – Hardware 80x86

Un poco de historia. El 80x86 de Intel comienza su andadura en 1981 con la versión 8086, hasta las nuevas versiones del Pentium, la primera en 1994. Todas ellas compatibles hacia atrás una con otra. Cada generación ha sumado nuevos atributos y mayor velocidad que la anterior, tal que hoy en día será muy difícil encontrar un microprocesador en operación de la primera generación, el 8088 o el 8086, puesto que son "muy lentos" y obsoletos. Sobre las versiones 286 y 386, se puede decir lo mismo, pero aquí es el software el que pide más proceso de cálculo. Hace unos años, los 486's se han reemplazado en su totalidad por los Pentiums, los Pentium PRO, los MMX, II, III, IV y V. Todos los CPUs de Intel mantienen un incremento en el rendimiento. En adelante se usará exclusivamente el modo más sencillo de operación de la arquitectura 80x86; el llamado modo real.

## Recordando el binario

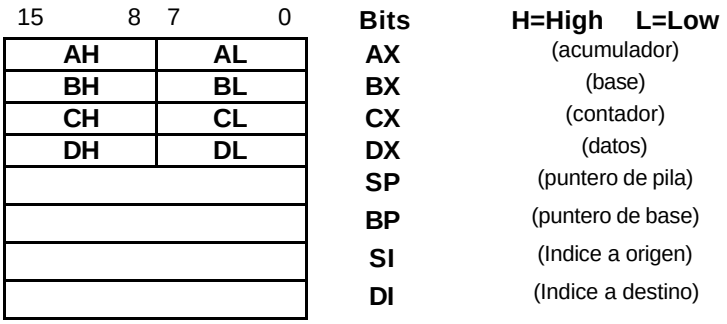
Antes de comenzar a describir lo que es el lenguaje ensamblador en 80x86, vamos a recordar cómo se representan los números en una CPU. Una CPU es la Unidad Central de Proceso dentro de un microprocesador, tal que la información en una CPU se representa en binario, esto es, usando base 2. Un **BIT** es el elemento que representa el elemento básico unidad. A partir de ahí tenemos:

1 NIBBLE: 0000 (4 BITS ). Es base del hexadecimal, e.j. 1111 = Fh.
1 BYTE: 00000000 (8 BITS) ó 2 NIBBLES
1 WORD: 0000000000000000 (16 BITS) ó 2 BYTES ó 4 NIBBLES. Es el tamaño usado para un registro de 16-bit.

## Registros

Los registros son los elementos principales de almacenamiento de la CPU. Pueden almacenar y mantener la información. En INTEL, existen tres tamaños de registros: de 8-bit, de 16-bit y de 32-bit (para versiones superiores al 386). Además, se especifican 4 tipos de registros básicos y registros de control:

- REGISTROS DE PROPÓSITO GENERAL,
- REGISTROS DE PILA,
- REGISTROS DE ÍNDICE y
- REGISTROS DE SEGMENTO




*Registros de propósito general, de pila e índice.*


*Registros de control*

## - Registros de propósito general.

Son cuatro registros de 16-bits etiquetados como **AX, BX, CX y DX**. Éstos se dividen en registros de 8-bits etiquetados como **AH**, que contiene el BYTE ALTO y **AL** que contiene el BYTE BAJO. A partir del 386 hay registros de 32 bits, que conservan al mismo nombre y una E que le precede, esto es: **EAX**. Estos registros se pueden usar indistintamente como: AL, AH, AX y EAX.

Por tanto si, AX contiene el número 24689 en decimal, tenemos que:

| AH | AL | |
|---|---|---|
| 01100000 | 01110001 | 6071 (hexadecimal) |

## - Registros de índices.

Estos registros de 16 bits también son llamados "registros punteros". Su utilidad es para el uso de instrucciones que operan con cadenas de caracteres. Lo forman el registro de índice SI (origen) y el registro DI (destino). Al igual que los registros de propósito general, en las versiones del 386 y superiores de Intel, existen los registros ESI y EDI.

## - Registros de Pila

Los registros de control de pila, BP y SP, se usan cuando manejamos una pila (stack). Estos registros se ven más adelante en el párrafo que corresponde al STACK.

## - Registros de Segmentos y offset

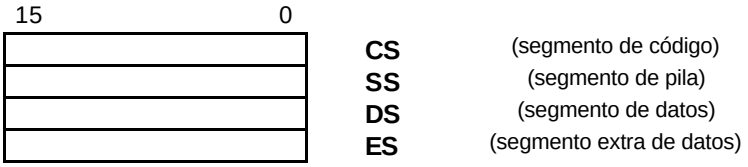
En los orígenes del 80x86 se pensó que nadie iba a usar más de 1 MByte de memoria, tal que los primeros microprocesadores (el 8088) no permiten direccionar más allá de este número. Es decir, se utilizan 20 bits de direcciones, tal que si queremos más direccionamiento se podrían usar 2 registros de 16 bits. Esto nos da un número binario de 32 bits. Sin embargo, ¡esto es mucho!, pensaron los ingenieros de diseño. Aparecen entonces para resolver este dilema, los segmentos y el desplazamiento (offset). Este modo de operación es el denominado modo real y se generó de la siguiente manera:

Dos registros de 16 bits, uno que contiene el segmento y otro el offset, generan una dirección física de 20 bits. Para ello, se colocan en posición tal que, el registro segmento se mueve 4 posiciones a la izquierda y el de offset a la derecha, también 4 posiciones, dejando 4 bits a 0 respectivamente. Si se unen (se suman) los 2 registros, se obtiene una dirección de 20 bits, que nos da la dirección física en memoria.

| | | |
|---|---|---|
| SEGMENTO | 0010010000010000 --- | Dirección base del segmento + 4 bits a 0 |
| OFFSET | ----0100100000100010 | 4 bits a 0 + Offset |
| DIRECCION FISICA | 00101000100100100010 | 20 bits total para dirección física a memoria |

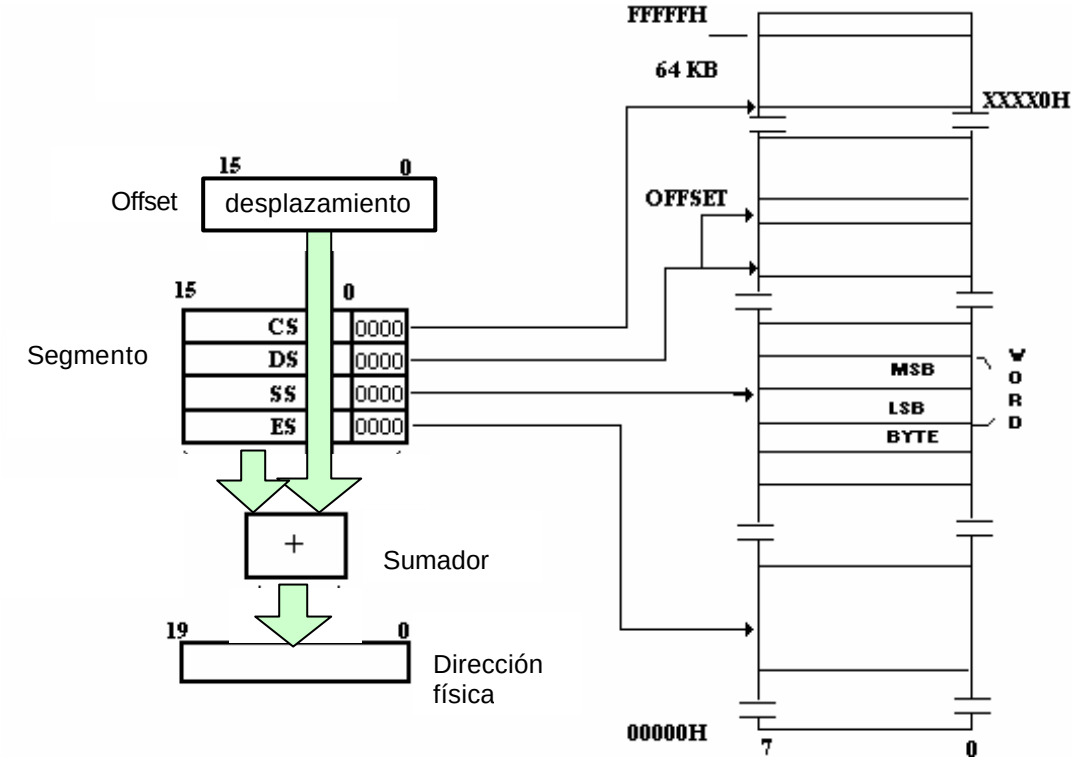
Por ejemplo, el registro DS guarda el Segmento y el registro SI guarda el offset, ambos son de 16 bits, por tanto **un SEGMENTO tiene un tamaño de 64KByte**. Sin embargo ambos registros se usan en conjunto para generar una dirección de 20 bits. Se usa para ello la notación DS:SI, es decir SEGMENTO:OFFSET. ¡Vaya chapuza !, pero hay que saber usarlo cuando se trabaja en el nivel más bajo de un PC.

Los registros para segmentos son: CS, SS, DS, ES y los registros que se usan para offset son: DX,BX, DI, SI, BP, SP




*Registros de segmentos*

El mecanismo para obtener la dirección física a memoria es el reflejado en el siguiente dibujo.

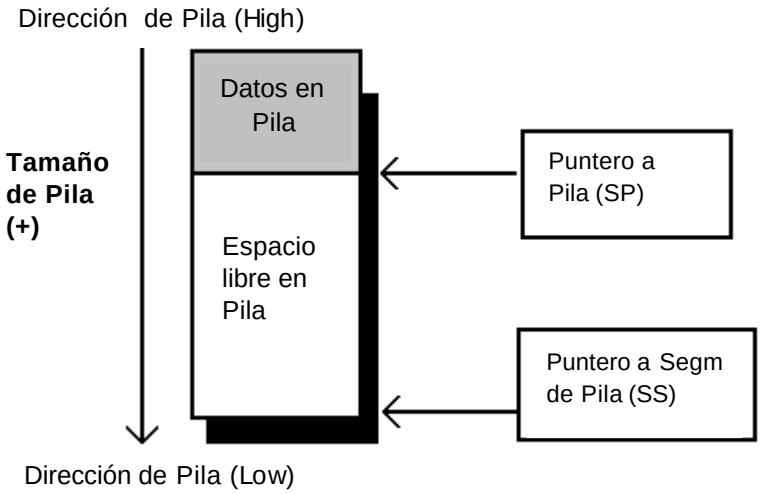



# Otros registros

Existen además otros registros de control. Los más importantes son, el registro IP (puntero de instrucción) y el registro de estado (FLAGS), que contiene la información del estado de la CPU.

# La pila (stack)

En general, en la arquitectura de Intel, seis registros de propósito general, son insuficientes para almacenar todos los valores requeridos para las operaciones en ensamblador. Para el caso de mantener y recuperar los valores de estos registros, existe un área de memoria llamada PILA (stack). El nombre de PILA no es por otra cosa sino por su forma de uso. Es decir, el último que entra es el primero que sale, comúnmente también llamado “Last In First Out (LIFO)”.




*Organización de la PILA (stack)*

Si entran más datos en la pila de los que puede almacenar, ésta desborda. Obsérvese que la PILA tiene asociada una dirección base alta y crece en sentido decreciente (en dirección). Es labor del usuario controlar el número de datos que entran en la PILA para no desbordar.

# II - Instrucciones básicas. Ensamblador

El número de instrucciones en ensamblador es muy grande, pero hay unas 20 instrucciones que son las usadas más frecuentemente. La mayoría de estas instrucciones se indican con tres letras en donde se especifica el tipo de instrucción y lleva asociada una serie de operandos separados por coma. Por ejemplo, para escribir en un registro usamos la instrucción MOV (de *move* en inglés).

```
mov ax,10      ; escribe 10 (decimal) en el registro  ax
mov bx,20      ; escribe 20 (decimal) en el registro  bx
mov cx,30      ; escribe 30 (decimal) en el registro  cx
mov dx,40      ; escribe 40 (decimal) en el registro  dx
```

Nótese que en ensamblador el punto y coma (;) separa los comentarios.

## Tipos de Operandos

Hay tres tipos de operandos en ensamblador:

- TIPO INMEDIATO
- TIPO REGISTRO
- TIPO MEMORIA

El operando inmediato es un número conocido en compilación, es decir una constante.
El operando tipo registro es cualquier registro de propósito general o registro de índice.
Un operando tipo memoria es una variable almacenada en memoria.

## Instrucciones básicas

Hay una serie de instrucciones importantes y necesarias para cualquier programa en ensamblador. La más importante es la instrucción MOV.

**MOV:** mueve un valor de un lugar a otro.
MOV destino, origen
Ejemplos:

```
mov ax,10          ; mueve un valor inmediato al registro ax
mov bx,cx          ; mueve en valor registro desde cx a bx
mov dx,Number      ; mueve el valor memoria definido como Number a dx
```

## Instrucciones Push y Pop. Uso del Stack

Conocido el Stack o pila, veremos cómo escribir datos y recuperalos. Hay dos instrucciones para estas operaciones, que son:
**PUSH:** Pone el dato de un registro en la pila.
**POP:** Recupera el dato de la pila y lo escribe en un registro.

```
push cx         ; escribe el contenido del registro cx en la pila
push ax         ; escribe el contenido del registro ax en la pila
pop cx          ; recupera el valor de la pila en el registro cx
pop ax          ; recupera el valor de la pila en el registro ax
```

Nótese que los valores de CX y AX se intercambian en el código anterior. Existe una instrucción especial para intercambiar valores de dos registros: XCHG, la cual produce el mismo resultado con una única instrucción:

```
xchg ax,cx      ; intercambia el valor de ax por el de cx
```

## Llamada a interrupción

Los PC están compuestos físicamente por: monitor, teclado, CPU, discos (flexibles y duros), periféricos y componentes adicionales. Todos estos periféricos y otras funciones se pueden leer o escribir desde programa a través de una solicitud de servicio llamada interrupción. Una interrupción es una operación que invoca la ejecución de una rutina específica que suspende la ejecución del programa que la llamó, de tal manera que el sistema toma control del computador colocando en el stack el contenido de los registros CS e IP. El programa suspendido vuelve a activarse cuando termina la ejecución de la interrupción y son restablecidos los registros salvados. Existen dos razones para ejecutar una interrupción: (1) intencionalmente como petición para la entrada o salida de datos de un dispositivo, y (2) un error serio y no intencional, como un resultado de desbordamiento de la ALU o de división por cero. En un PC el control de estos servicios se hace a través del BIOS (Basic Input-Output System) o del sistema operativo MS-DOS (MicroSoft Disk Operating System), que le llamaremos DOS. Existen interrupciones *software* e interrupciones *hardware*, los ejemplos siguientes muestran cómo llamar al DOS y a la BIOS.

**INT:** llama a una función del DOS o BIOS a través de una subrutina para realizar un servicio especial, por ejemplo manipular el vídeo, la pantalla, abrir un fichero, escribir en un puerto, etc. En este texto usaremos una de ellas, la salida a pantalla, que se corresponde con la 21h. Las demás se pueden consultar en la bibliografía.

INT interrupt number
Ejemplos:

```
int 21h         ; Llama a la interrupción número 21 (hexadecimal)
int 10h         ; Llama a la interrupción del Video en la BIOS
```

Casi todas las interrupciones tienen asociados más de un servicio, por ello es necesario pasar el número de servicio deseado. Se utiliza para ello el registro AH. El ejemplo más usado es escribir algo en la pantalla del monitor:

```
mov ah,9        ; el servicio número 9 (decimal) se pasa al registro AH
int 21h         ; llamada a la interrupción 21
```

El servicio, o función, número 9 se corresponde a la impresión de caracteres en pantalla. Para ello, antes es necesario especificar qué vamos a escribir en pantalla. Se hace uso de DS:DX que será el puntero de la dirección física en memoria indicando dónde se

encuentra la cadena de caracteres a imprimir en pantalla. La cadena de caracteres se termina con el símbolo ($). Sería sencillo si pudiésemos escribir directamente en DS, como no es posible, tenemos que usar el registro AX para generar la dirección física. El ejemplo es el siguiente:

```
mov dx,OFFSET Message    ; DX contiene el offset de message
mov ax,SEG Message       ; AX contiene el segmento del message
mov ds,ax                ; DS:DX apunta a la dirección física del mensaje
mov ah,9                 ; carga el servicio 9 – caracteres a pantalla
int 21h                  ; llamada a la interrupción de DOS
```

Las directivas OFFSET y SEG indican al compilador que el segmento y el offset de *Message* se use para obtener la dirección física en memoria la cadena de datos, para posteriormente escribir a pantalla. Hecho esto, podremos enviar el mensaje a la pantalla, escribiendo previamente en el segmento de datos el mensaje con la etiqueta y la directiva DB de la siguiente forma:

```
Message DB "Estructura y Tecnología de Computadores. Práctica 1$"
```

Nótese que la cadena de caracteres termina con el símbolo ($). DB es una directiva para asignar memoria a variables del tipo byte y el contenido del mensaje en un array de bytes (ASCII). Además de formato bytes (DB) se puede usar formatos tipo word (DW) y doble word (DD).

```
Number1 db ?
Number2 dw ?
```

El símbolo (?) al final de la línea indica que no hay inicialización.

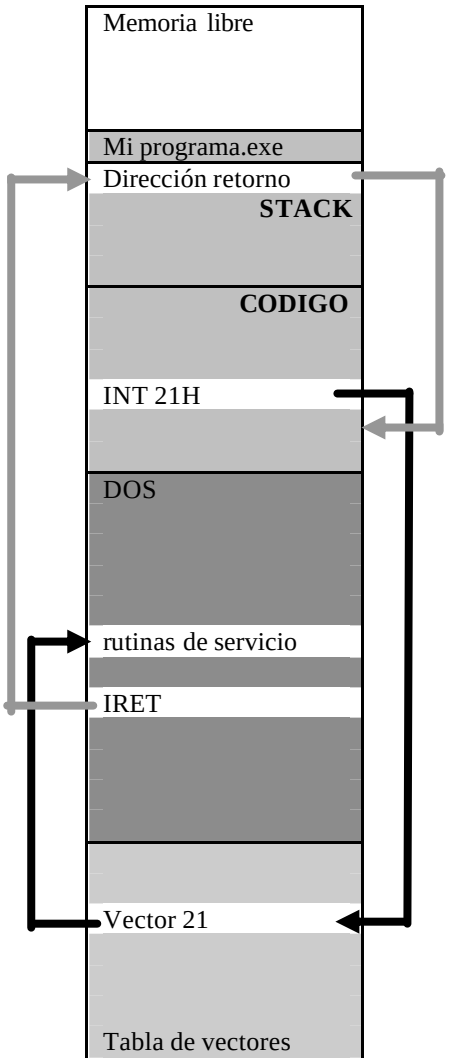
```
Number1 db 0
Number2 dw 1
```

En este caso Number1 es igual a 0 y Number2 es igual a 1. Hay que tener en cuenta el tamaño de los datos, por ejemplo:

```
mov al,Number1           ; ok
mov ax,Number1           ; error

mov bx,Number2           ; ok
mov bl,Number2           ; error
```

Lo que indica que, únicamente se pueden usar bytes en registros de 8 bits y palabras en registros de 16 bits.

En resumen, el proceso de una interrupción *software* es el siguiente:




## Escribiendo un sencillo programa

Es suficiente con lo anterior para escribir un sencillo programa en ensamblador, compilarlo y ejecutarlo.

```
; Ejem_1 : UNO.ASM
; Un programa en ensamblador que envía un mensaje
; a pantalla. A ver que pasa….

.model tiny                 ; directiva de modelo de programación
.stack                      ; directiva de asignación de segmento
.data                       ; directiva de asignación de datos

Message db "Práctica 1. EyTC$"      ; mensaje a escribir en pantalla

.code                       ; directiva para el segmento de código

start:                              ; inicio de programa
    mov dx,OFFSET Message           ; offset en DX
    mov ax,SEG Message              ; segmento en AX
```

```
    mov ds,ax                       ; DS:DX apunta al mensaje

    mov ah,9          ; función 9-llamada a servicio de pantalla
    int 21h           ; llamada a servicio del DOS
    mov ax,4c00h      ; instrucción de comprobación para
    int 21h           ; volver al DOS

END start             ; fin
```

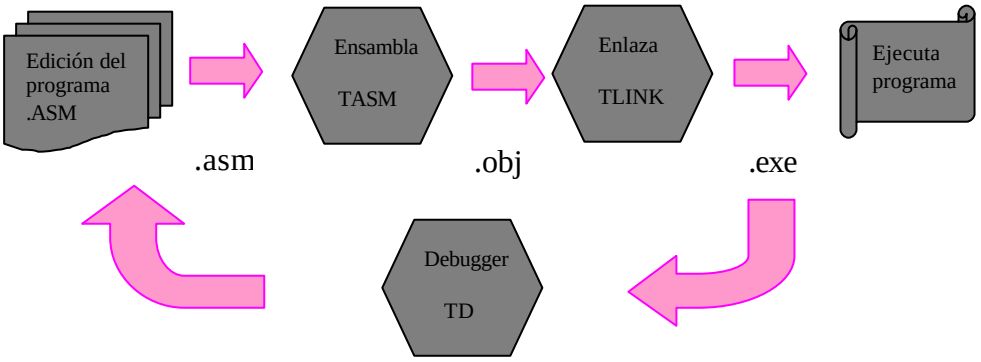
Las dos últimas instrucciones:

```
mov ax,4c00h          ; instrucción de comprobación de error
int 21h               ; y volver del DOS, correctamente.
```

Es un servicio adicional del DOS, llamado de terminación (4Ch). Se emplea como comprobación de error, dejando siempre en el registro AL un valor de 00h.

## El Ensamblador

Para la compilación de un programa en ensamblador y su enlazado, hay que seguir una serie de pasos. Usaremos el ensamblador llamado TurboAssembler (TASM). Este ensamblador lo que hace es obtener un programa "ejecutable" para un ordenador que tenga integrado un microprocesador de la familia 80x86. La figura siguiente detalla los cuatro pasos que lleva a cabo el compilador:




El fichero.asm contiene el programa en texto. Es el fichero origen para el compilador.
El fichero.obj contiene un fichero intermedio. Este o varios ficheros son generados por el compilador.
El fichero.exe es el fichero ejecutable desde el DOS. Es el fichero ejecutable generado por el enlazador o "linker".

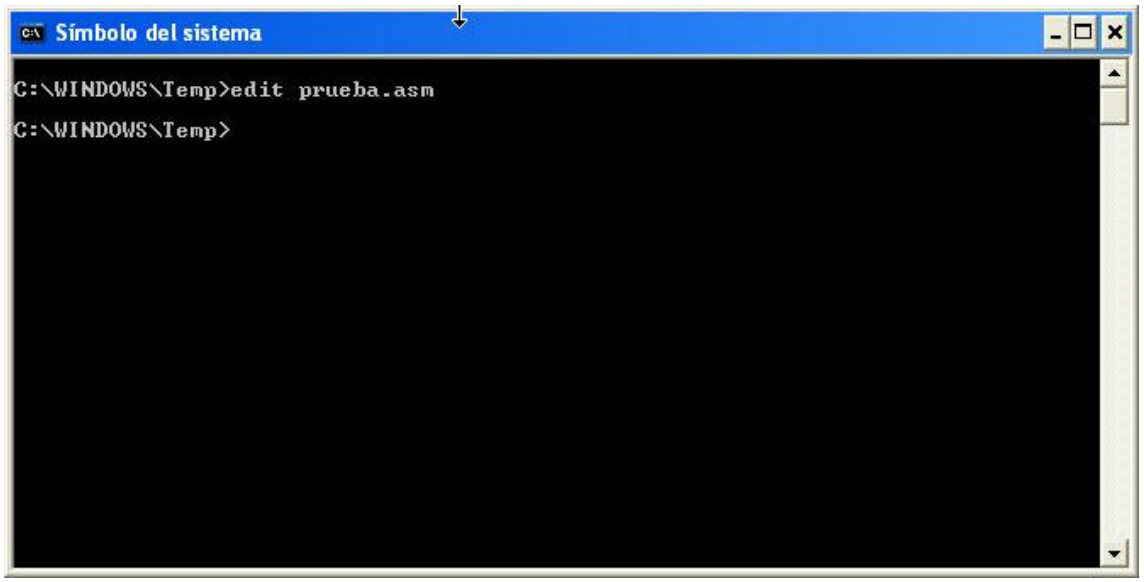
Estos cuatro pasos se llevan a cabo con las dos instrucciones siguientes:

```
tasm file.asm
tlink file
```

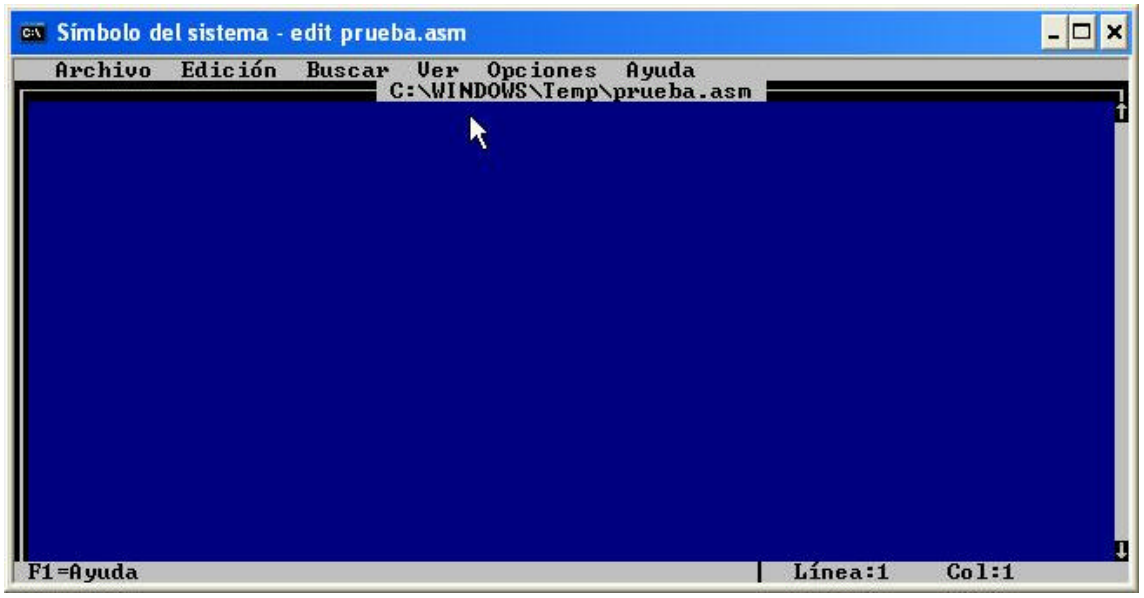
Antes de esto es necesario preparar y organizar el programa **:**

1.- Edición del programa. Se utiliza cualquier editor de texto para escribir el código del programa en ensamblador. Para que pueda ser interpretado por el compilador TASM, la condición es que el fichero de este programa lleve extensión .ASM y el nombre del fichero no debe ser mayor a ocho caracteres. Para la edición, por ejemplo, se puede usar EDIT como programa editor de texto. En este caso, vaya a Inicio -> Accesorios->Símbolo del sistema. Aparecerá una ventana de DOS. Vaya al directorio de trabajo que esté usando e invoque al editor.

C:\MiDirectorio\TASM> edit prueba.asm




A continuación aparecerá la siguiente ventana de edición. Pulsando ALT se accede al a la barra del menú superior. Para desplazarse se emplean las teclas de dirección (indicadas por una flecha).




Continúe editando el siguiente programa, termine y guarde el fichero. No olvide confirmar que tiene extensión .asm,.

```
; Ejem_1 : PRUEBA.ASM
; Un programa en ensamblador que envía un mensaje
; a pantalla. A ver que pasa….

.model tiny
.stack
.data

Message db "Práctica 1. EyTC$"      ; mensaje a pantalla

.code

start:
mov dx,OFFSET Message               ; offset en DX
mov ax,SEG Message                  ; segmento en AX
mov ds,ax                           ; DS:DX apunta al mensaje

mov ah,9            ; función 9–llamada a servicio de pantalla
int 21h             ; llamada a interrupción del DOS
mov ax,4c00h        ; vuelta al DOS
int 21h

END start           ; fin
```

Ya hemos generado un pequeño programa en ensamblador. Compilamos, enlazamos y ejecutamos el mismo.

C:\MiDirectorio\TASM> tasm /z /zi prueba.asm
C:\MiDirectorio\TASM> tlink prueba
C:\MiDirectorio\TASM> prueba

Debe aparecer el texto "Práctica 1. EyTC" el resultado en la ventana de DOS, si no hay errores en la compilación.
**NOTA: es importante tener el path al fichero ejecutable tasm.exe o estar en el mismo directorio**

# El depurador (debugger)

El depurador es una herramienta que permite observar lo que está haciendo un programa. El ejemplo que veremos aquí tiene como objetivo una breve introducción y uso del "depurador" con la herramienta Turbo Debugger (TD). El TD nos permite comprobar lo que está haciendo el programa paso a paso. Antes de esto se necesita generar un programa. Para ello, invoque un editor de texto y a continuación escriba el siguiente programa que nos permitirá supervisar el programa paso a paso y con mayor amplitud el depurador. Este programa consiste en comprobar instrucciones de suma, de multiplicación, el uso del stack y la llamada a una función.

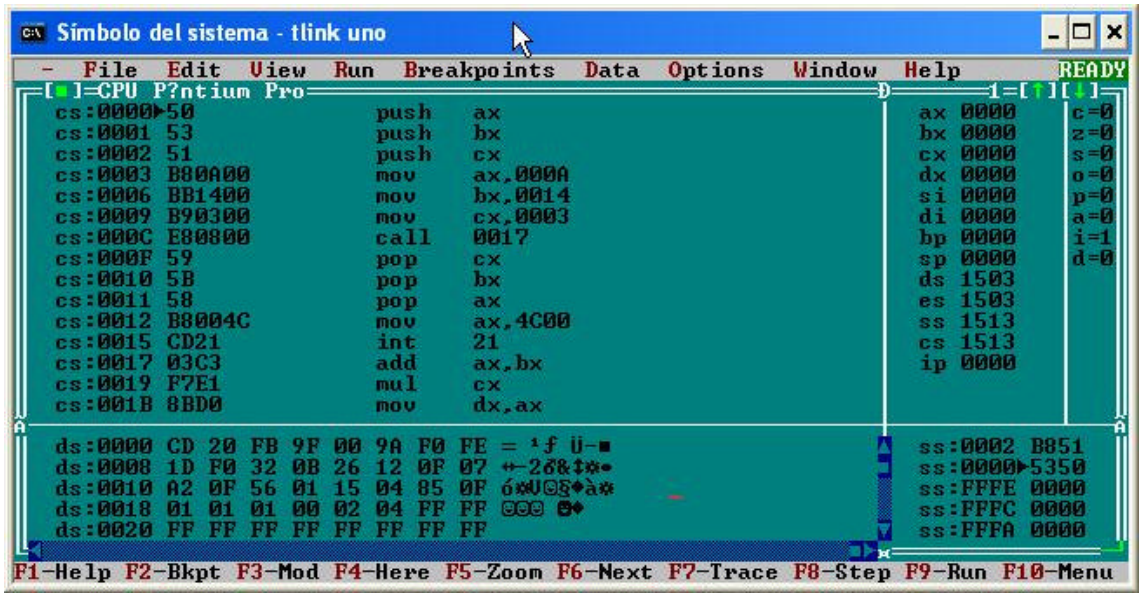
```
; Ejem_1 : UNO.ASM
; Ejemplo para demostrar el uso del depurador TD.

.model tiny     ; directiva de modelo de programación
.code           ; directiva de asignación de código
.stack          ; directiva de asignación de snack

start:          ; inicio de programa
```

```
push ax          ; guarda el valor de ax en la pila
push bx          ; guarda el valor de bx en la pila
push cx          ; guarda el valor de cx en la pila

mov ax,10        ; ax = 10
mov bx,20        ; bx = 20
mov cx,3         ; cx= 3

Call ChangeNumbers      ; llamada a subrutina

pop cx           ; recupera cx
pop bx           ; recupera bx
pop ax           ; recupera dx

mov ax,4C00h     ; terminación del programa
int 21h          ; vuelve al DOS

; subrutina

ChangeNumbers PROC      ; inicio de subrutina

add ax,bx        ; suma al valor de ax, el valor de bx
mul cx           ; multiplica ax por cx
mov dx,ax        ; guarda el resultado en dx
ret              ; vuelve de subrutina

ChangeNumbers ENDP ; fin de subrutina

end start        ; fin de programa
```

Para trabajar con el depurador se necesita un fichero ensamblado. Este fichero, con extensión .EXE, es un fichero del tipo “ejecutable”. Invoque al depurador:

```
td uno.exe
```

A continuación podemos ver una ventana con las siguientes características:



En primera instancia hay 5 recuadros o ventanas. De arriba a abajo e izquierda a derecha:

- El primero, se corresponde al segmento de código.
- El segundo, se corresponde al segmento de datos.
- El tercero, se corresponde con el contenido de los registros.
- El cuarto, se corresponde con el control de la pila (stack).
- El quinto, con el contenido de registros de estado.

También hay dos menús. El principal (fila superior) que es una barra de control de ventanas, así como de funciones de usuario y el de la fila inferior, que incluye una barra con un menú de funciones rápidas a través de las teclas F1 a F10.

Volviendo a las ventanas principales. Se ve en primera instancia en el primer recuadro o ventana que contiene las primeras líneas:

```
cs:0000 50       push ax
cs:0001 53       push bx
cs:0002 51       push cx
```

La primera línea - `cs:0000 50 push ax`– corresponde al segmento de código y contiene la posición de la línea de código (una posición de memoria de 1 byte), el código de instrucción en hexadecimal y la instrucción en código ensamblador. Por tanto, la posición **0000** del segmento de código contiene la instrucción **50** de longitud un byte y se corresponde con la instrucción del tipo **push ax**.

A continuación vemos las siguientes líneas:

```
cs:0003 B80A00   mov ax,000A
cs:0006 BB1400   mov bx,0014
cs:0009 B90300   mov cx,0003
```

Obsérvese que la información representada está en binario y mostrada en hexadecimal. Será más fácil si el hexadecimal lo convertimos a binario para conocer su valor en decimal. Mire con atención las líneas de código y los cambios respecto al código que previamente ha escrito en el editor. Se pueden ver seis dígitos en el código de instrucción, por tanto la secuencia de las direcciones de memoria aumenta en tres posiciones de 1 byte y posteriormente el código de la instrucción expresado en parte en formato hexadecimal. Nótese la descripción en ensamblador y su correspondencia con el programa descrito en texto. ¿Qué diferencias encuentra usted?

Pasando ahora a otra ventana, en la parte derecha de la figura, se muestra el contenido de todos los registros del microprocesador en una columna. En este momento están vacíos. Comience pulsando la tecla F7 (ejecución paso a paso). Se ejecuta la primera línea del programa. Véase también como el contenido de AX se coloca en el STACK. En esta ventana se observa el registro SP (stack pointer) cómo ha cambiado. Continué con F7 hasta que los contenidos de los registros AX y BX vuelvan a valor de 00h.

Anote paso a paso, los contenidos de AX, BX, CX, SS, SP.

# III – Control de flujo y Entrada/Salida

## Haciendo las cosas más sencillas

La forma de escribir a pantalla no es fácil de entender. A continuación se explica paso a paso la rutina para escribir en pantalla. Recuerde que es la dirección del mensaje la que nosotros especificamos. Se comienza con las siguientes instrucciones básicas para enviar un mensaje a pantalla:

```
mov dx,OFFSET MyMessage  ;apunta a la dirección de MyMessage
mov ax,SEG MyMessage     ;se coloca el segmento de datos en ax
mov ds,ax                ;obtiene la dirección física del mensaje
```

A partir de aquí solicitamos el servicio a la BIOS para que se "interrumpa" el programa y se llame al un servicio de la BIOS que escribe en pantalla (identificado como 9). Esto se corresponde con las siguientes instrucciones en ensamblador;

```
mov ah,9          ;el servicio número 9 se pasa al registro AH
int 21h           ;llamada a la interrupción 21, que interrumpe
                  ; el programa y ejecuta la rutina en la BIOS
```

Sin embargo, como se verá más adelante, en el modelo de programación más simple (TINY), existe un único segmento para datos, programa y pila. En este modelo se simplifica la salida a pantalla, tal que una cadena de caracteres se envía a pantalla como:

```
mov dx,OFFSET Message    ; mensaje a la pantalla
mov ah,9                 ; usando la función 09h
int 21h                  ; e interrupción 21h
```

## Lectura por teclado

Se emplea la interrupción 16h y la función de servicio 00h para leer un carácter de teclado. Esta rutina devuelve el carácter al registro AH y su valor hexadecimal (ASCII) al registro AL.

```
xor ah,ah        ; ah=00H, función 00h – lee carácter
int 16h          ; interrupción 16h
```

Al final, lo más interesante es conocer el contenido de AL, que corresponde al carácter tecleado.

Nota: La instrucción XOR es la función booleana de la OR exclusiva. Nótese que la instrucción anterior pone a valor 00h un registro.

## Sacando a pantalla un carácter

Para obtener en pantalla un carácter que previamente se ha tecleado, no se puede usar la función 9h. Para ello, se usa lo siguiente:

```
; después de leer el teclado, 00h e interrupción 16h

mov dl,al        ; mueve al (ASCII) a dl
mov ah,02h       ; servicio 02h de interrupción 21h
int 21h          ; llamada a la interrupción 21h
```

Esta instrucción se puede usar en conjunto con push y pop.

## Control de Flujo

En ensamblador hay un conjunto de instrucciones para el control de flujo de un programa, como en cualquier otro lenguaje. El más común es el salto incondicional, por ejemplo:

```
jmp Alabel
```

Esta instrucción lo que hace es indicar a través de una etiqueta (Alabel) el cambio del control de flujo del programa. Ejemplo:

```
jmp ALabel
.
.
.
ALabel:
```

En caso de que un programa esté sujeto a una condición, este salto es del tipo condicional a través de la instrucción correspondiente. Primeramente comparamos y luego usamos la instrucción de salto que corresponda. Ejemplo:

```
cmp ax,3          ; AX = 3?
je correcto       ; si
```

En la tabla a continuación se pueden observar las principales condiciones de salto y a continuación la de comparación.

## Instrucciones de Salto con Condición

| | |
|---|---|
| JA | Salta si el primer número es mayor que el segundo |
| JAE | Salta si el primer número es mayor o igual que el segundo |
| JB | Salta si el primer número es menor que el segundo |
| JBE | Salta si el primer número es menor o igual que el segundo |
| JNA | Salta si el primer número no es mayor (JBE) |
| JNAE | Salta si el primer número no es mayor o igual como (JNB) |
| JNB | Salta si el primer número no es menor (JAE) |
| JNBE | Salta si el primer número no fue menor o el mismo como (JA) |
| JZ | Salta si los dos números son iguales |
| JE | Igual que JZ, |
| JNZ | Salta si los dos número NO son iguales |
| JNE | Igual que JNZ, |
| JC | Salta si el bit de acarreo está activo |

Nota: el salto puede ser de máximo 127 bytes posiciones de memoria.

**CMP:** comparación de un valor

```
CMP registro o variable, valor
jxx destino
```

Ejemplo;

```
cmp al,'Y'       ; compara el valor en al con Y
je  IGUAL        ; si es igual entonces salta a IGUAL
```

Si el salto es mayor a 127 posiciones de memoria el compilador nos generará un "warning" o un error. S

Por último, hay que ser cuidadoso en ordenar el código. Para resolver este tipo de problemas en la organización de los programas en ensamblador, existen métodos y "buenas formas" que demuestran el control de flujo de un programa. Véase el siguiente:

```
; Ejem_2 : DOS.ASM
; Este programa demuestra el control de flujo
; Entrada y salida por pantalla

.model tiny
.code
.stack

start:

mov dx,OFFSET Message    ; mensaje a la pantalla
mov ah,9                 ; usando 09h
int 21h                  ; e interrupción 21h

mov dx,OFFSET Prompt     ; mensaje a la pantalla
mov ah,9                 ; usando la función 09h
int 21h                  ; e interrupción 21h

jmp First_Time           ; continua en la etiqueta FirsTime
Prompt_Again:

mov dx,OFFSET Another    ; mensaje a la pantalla
mov ah,9                 ; usando 09h
int 21h                  ; e interrupción 21h

First_Time:

mov dx,OFFSET Again      ; mensaje a la pantalla
mov ah,9                 ; usando la función 09h
int 21h                  ; e interrupción 21h

xor ah,ah                ; limpia ah a valor 00h
int 16h                  ; interrupción 16h  lee carácter

mov bl,al                ; almacena en bl
mov dl,al                ; mueve al a dl
mov ah,02h               ; función 02h–caracter a pantalla
int 21h                  ; llamada al servicio de DOS

cmp bl,'Y'               ; al=Y?
je Prompt_Again          ; si si, nuevo mensaje a pantalla
```

```
cmp bl,'y'              ; al=y?
je Prompt_Again         ; si si, nuevo mensaje a pantalla

theEnd:

mov dx,OFFSET GoodBye   ; adiós
mov ah,9                ; función 9
int 21h                 ; interrupción 21h

mov ah,4Ch              ; fin
int 21h

.data                   ; directiva de declaración de datos
CR equ 13               ; carácter intro
LF equ 10               ; retorno de línea

Message DB "Programa de Entrada-Salida$"
Prompt  DB CR,LF,"Entrada.$"
Again   DB CR,LF,"De nuevo? $"
Another DB CR,LF,"Aquí esta de nuevo!$"
GoodBye DB CR,LF,"Adiós."

end start
```

# IV - Instrucciones aritméticas y procedimientos

## Instrucciones aritméticas

Además de las instrucciones vistas anteriormente, hay otro grupo muy importante que se introducen a continuación:

**ADD:** Suma un número a otro.

```
ADD operando1,operando2
```

Suma el operando2 al operando1. El resultado se almacena en operando1. Un valor inmediato no puede ser usado como operando1, pero sí como operando2.

**SUB:** Resta un número a otro.

```
SUB operando1,operando2
```

Resta el operando1 al operando2. El resultado se almacena en operando1. Un valor inmediato no puede ser usado como operando1, pero sí como operando2.

**MUL:** Multiplica dos números enteros sin signo (positivos)
**IMUL:** Multiplica dos números enteros con signo (positivo o negativo)

```
MUL registro o variable
IMUL registro o variable
```

Esto es, multiplica AL o AX por el valor del registro o variable dado. Si se emplea un operando de tamaño un byte, se emplea la parte del registro correspondiente a AL y el resultado se almacena en AX. Si el operando es de tamaño word, entonces se multiplica AX y el resultado se almacena en DX:AX.

**DIV:** Divide dos números enteros sin signo (positivos)
**IDIV:** Divide dos números enteros con signo (positivo o negativo)

```
DIV registro o variable
IDIV registro o variable
```

El proceso es similar a MUL e IMUL, dividiendo el número de AX por el registro o variable dado. El resultado se almacena en dos lugares, AL almacena el resultado de la división y el resto se almacena en AH. Si el operando es un registro de 16 bits, entonces el número en DX:AX se divide por el operando, almacenando el resultado en AX y el resto en DX.

# Procedimientos

En ensamblador un procedimiento es equivalente a una función en C o Pascal. Es también un mecanismo sencillo de encapsular partes de código. Para definir un procedimiento se siguen los siguientes pasos:

```
PROC AProcedure
.
.                ; código
.
ret
ENDP AProcedure
```

La llamada para ejecutar este procedimiento es:

```
call Aprocedure
```

Ejemplo:

```
; Ejem_3 : TRES.ASM
; Ejemplo de un procedimiento que envía a pantalla Hola

.model tiny
.stack
.code

Start:
call Display_Hola        ; Llamada al procedimiento Hola
mov ax,4C00h             ; vuelta  al DOS
int 21h                  ; fin de programa

Display_Hola PROC
mov dx,OFFSET HolaMuyBuenas
mov ah,9
int 21h
ret
Display_Hola ENDP

.data
HolaMuyBuenas DB 13,10 "Que pasa chaval! $"     ; mensaje
end Start
```

# Paso de parámetros a un procedimiento

La utilidad de un procedimiento es la modificación de valores desde el procedimiento. Para ello es necesario el paso de parámetros. Existen tres formas de hacer esto: por registro, por memoria o por pila (stack).

A continuación hay 3 ejemplos con los 3 mecanismos anteriores. El ejemplo saca por pantalla un valor (el valor ASCII de 254).

## - por registro

Es el más sencillo y el más rápido. Se necesita mover los parámetros a los registros y después hacer la llamada al procedimiento.

```
; Ejem_4: CUATRO.ASM
```

```
; imprime en pantalla un cuadro usando registros
; como paso de parámetros

.model tiny
.code
.stack

Start:

mov dh,4          ; posición de la fila en pantalla
mov dl,5          ; posición  columna en pantalla
mov al,254        ; ASCII del carácter a pantalla
mov bl,4          ; color

call PrintChar    ; a pantalla
mov ax,4C00h      ; fin del programa
int 21h

PrintChar PROC

push bx           ; salvaguarda registros
push cx

xor bh,bh         ; limpia registro bh – página video 0
mov ah,2          ; funtion 2 - mover cursor
int 10h           ; fila x columnas ya en dx

pop bx            ; recupera bx

xor bh,bh         ; limpia registro bh – página video 0
mov ah,9          ; funcion 09h escribe caracter
mov cx,1          ; cx se usa como contador, se pone un 1
int 10h           ; llamada a servicio

pop cx            ; recupera registro

ret               ; return

PrintChar ENDP

end Start
```

## - por memoria

También es sencillo, pero el programa resultante es más grande y más lento que el anterior.

Para pasar parámetros a través de memoria es necesario copiar estos en variables almacenados en memoria.

```
;Ejem_5: CINCO.ASM

; lo mismo pero por paso de parámetros

.model tiny
.code
.stack

Start:
```

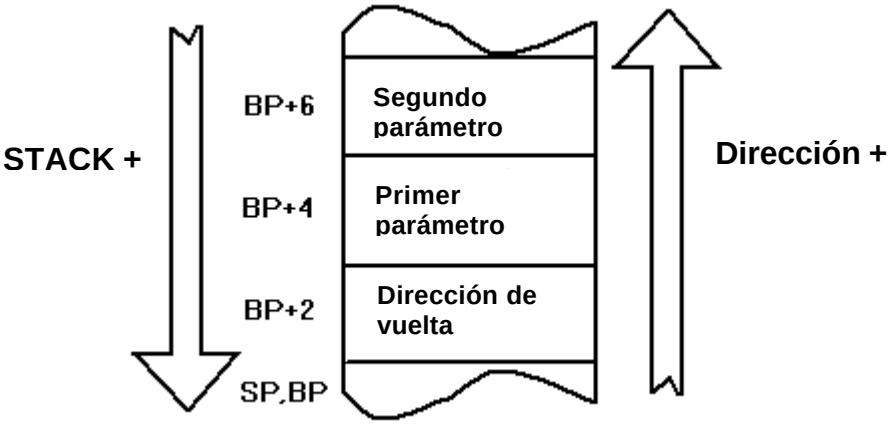
```
mov Row,4        ; fila
mov Col,5        ; columna
mov Char,254     ; ASCII
mov Colour,4     ; color

call PrintChar   ; llamada a función
mov ax,4C00h     ; termina programa
int 21h

PrintChar PROC

push ax cx bx    ; guarda registros

xor bh,bh        ; limpia bh – página de video 0
mov ah,2         ; funcion 2 - mover cursor
mov dh,Row
mov dl,Col
int 10h          ; llamada a la Bios

mov al,Char
mov bl,Colour
xor bh,bh        ;
mov ah,9         ; funcion 09h escribe
mov cx,1         ;
int 10h          ; llamada a servicio de la Bios

pop bx cx ax     ; recupera registros

ret              ; return
PrintChar ENDP

; variables para almacenar datos

Row db           ?
Col db           ?
Colour db        ?
Char db          ?

end Start
```

## - por pila (stack)

Debido a sus características, es el más flexible. No por ello más complicado.

```
;Ejem_6 : SEIS.ASM

; lo mismo usando el Stack

.model tiny
.code
.stack

Start:

mov dh,4                ; fila
mov dl,5                ; columna
mov al,254              ; ascii
mov bl,4                ; color
push dx ax bx           ; guarda parametros en stack

call PrintString        ; funcion a pantalla
```

```
pop bx ax dx            ;recupera registros
mov ax,4C00h            ;termina programa
int 21h

PrintString PROC

push bp                 ; guarda bp
mov bp,sp               ; mueve sp a bp
push cx                 ; guarda registro (se va a usar cx)

xor bh,bh               ; limpia bh
mov ah,2                ; funcion 2 - mover cursor
mov dx,[bp+8]           ; recupera dx
int 10h                 ; llamada a Bios

mov ax,[bp+6]           ; caracter
mov bx,[bp+4]           ; atributo
xor bh,bh               ; pantalla en página 0
mov ah,9                ; funcion 09h escribe
mov cx,1
int 10h                 ; llamada de servicio a bios

pop cx                  ; recupera de stack
pop bp

ret                     ; return
PrintString ENDP

end Start
```




*Proceso que sigue la utilización del STACK*

## Modelos de Memoria

Se ha venido utilizando la directiva .MODEL a lo largo de los programas de este texto. Para el MSDOS solamente existen dos tipos de ficheros, los .COM y los .EXE. Los del tipo .EXE que pueden usar hasta cuatro segmentos (STACK, DATA, EXTRA y CODE) cada uno de 64 KB de tamaño. Un fichero .COM solo tiene un segmento de 64KB y

siempre inicia en la dirección 0100H. A continuación se explica el significado de .MODEL:

.MODEL MemoryModel

Donde MemoryModel puede ser del tipo TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE, o FLAT. En este texto estamos usando el modelo más simple Tiny.

**Tiny**

Existe únicamente un segmento para ambos, código y datos. El resultado de la compilación puede ser un fichero tipo .COM.

**Small**

Por defecto todo el código se coloca en un segmento y los datos declarados en el segmento de datos se colocan en otro segmento. Además todos los procedimientos y variables son del tipo NEAR (dentro del mismo segmento).

**Compact**

Por defecto todo el código está organizado en un segmento y los datos pueden estar en otro segmento independiente. Esto significa que los datos se direccionan apuntando a la base del segmento y su offset correspondiente. El código es del tipo NEAR y las variables son del tipo FAR (pueden estar en segmentos diferentes).

**Medium**

Opuesto al modelo compact. Los datos son NEAR y el código es FAR.

**Large**

Ambos, los datos y el código son FAR. Es necesario apuntar a ambos con el segmento y el offset.

**Flat**

Esto no es usado, a menos que se use un espacio de memoria no-segmentado de 32 bits. Para esto se necesita una extensión de DOS.

## Otras instrucciones de manejo de pila. Push y Pop

Ya se ha comentado el manejo de las instrucciones de push y pop para el control de la pila. Es posible, en una misma línea incluir varias instrucciones de almacenamiento y de recuperación de datos. Esto hace más fácil el entendimiento del código.

```
push ax bx cx dx        ; guarda registros
pop dx cx bx ax         ; recupera registros
```

## PUSHA y POPA

PUSHA es una instrucción que coloca todos los registros de propósito general en la pila. Al igual que el mecanismo anterior, facilita la codificación. Su equivalente es el siguiente:

```
push ax
push cx
push dx
push bx
push sp
push bp
push si
push di
```

POPA es la instrucción inversa. Recupera todos los registros de la pila.

## Desplazamientos

Las instrucciones de desplazamiento mueven bit a bit un registro. Lo hay del tipo lógico (sin signo) y aritmético (con signo). Además esas instrucciones permiten hacer operaciones en base 2 de multiplicación y división, aunque son muy lentas comparativamente con otro tipo de instrucciones aritméticas. Para hacer este tipo de operaciones aritméticas, se puede emplear el desplazamiento binario hacia la izquierda o derecha una o varias posiciones. Cada desplazamiento equivale a dividir o multiplicar en potencia de 2. Su equivalente son los operadores "<<" y ">>" en lenguaje C.

Existen cuatro formas diferentes para efectuar estos desplazamientos.

**SHL** Desplazamiento lógico a la izquierda. Multiplicación por 2 sin signo.
**SHR** Desplazamiento lógico a la derecha. División por 2 sin signo.
**SAR** Desplazamiento aritmético a la derecha. División por 2 con signo
**SAL** Igual a SHL

Las instrucciones **ROR, RCR, ROL y RCL** Son las correspondientes al desplazamiento con rotación. Se incluyen las instrucciones que insertan el bit de acarreo ("C" del registro de estado).

## Bucles (loops)

Una forma de optimizar el uso de instrucciones de salto anidadas con instrucciones del tipo JMP, es por medio de una instrucción de tipo bucle. Esto es, usando la instrucción LOOP y colocando el valor del número de veces que se tiene que hacer el bucle en el registro CX. En cada pasada del bucle se va decrementado el valor de CX (CX-1). Esto significa hacer un salto corto, con un valor de hasta 128 bytes, es decir 127 bytes o anterior a la posición donde se localiza la instrucción LOOP.

```
mov cx,100       ; bucle de 100 iteraciones
Label:
.
.
.
Loop Label:      ; decrementa CX y salta a la posición Label
```

El equivalente sin usar la instrucción de LOOP es:

```
mov cx,100       ; bucle de 100 iteraciones

Label:
dec cx           ; CX = CX-1
jnz Label        ; continua hasta que el valor de CX=0
```

# V - Salida a Pantalla

## Modo texto

Además de la salida a pantalla de vista anteriormente, existen otros mecanismos para trabajar con la pantalla.

```
;Ejem_8: TEXTO_1.ASM
;este ejemplo posiciona el cursor en un punto concreto de la pantalla
.model tiny
.code
.stack
start:

      mov dh,12               ; cursor a columna 12
      mov dl,32               ; cursor a fila 32
      mov ah,02h              ; pone en ah el número de servicio
      xor bh,bh               ; bh se pone a 0 (pagina 0 del video)
      int 10h                 ; llamada al bios

      mov dx,OFFSET Texto     ; DS:DX apunta la mensaje
      mov ah,9                ; función 9 – salida de texto a pantalla
      int 21h                 ; llamada al bios

      mov ax,4C00h            ; salida a dos
      int 21h

.data
      Texto DB "Esta es una prueba para enviar texto a pantalla”$"

end start
```

Escribiendo a pantalla usando la función 40h de la interrupción 21h.

```
;Ejem_10: TEXTO2.ASM
;este ejemplo saca uno a uno los caracteres a pantalla
.model small
.stack
.code

      mov ax,@data            ; setup ds como segmento de datos
      mov ds,ax
      mov ah,40h              ; función 40h – escribe fichero
      mov bx,1                ; bx = 1 (pantalla)
      mov cx,21               ; longitud de la cadena
      mov dx,OFFSET Texto     ; DS:DX apunta al texto
      int 21h                 ; llamada al servicio de dos

      mov ax,4C00h            ; fin
      int 21h

.data
      Texto DB "Un poco más de texto"
end
```

El siguiente ejemplo utiliza la función 13h de la interrupción 10h. Su ventaja es poder escribir en cualquier parte de la pantalla y en cualquier color.

```
; Ejem_11: TEXTO3.ASM

.model small
.stack
.code

      mov ax,@data            ; set-up ds como segmento de datos
      mov es,ax               ; set-up el segmento es

      mov bp,OFFSET Texto     ; ES:BP apunta al mensaje
      mov ah,13h              ; función 13 - escribe
      mov al,01h              ; mueve cursor
      xor bh,bh               ; video a pagina 0
      mov bl,5                ; atributo - magenta
      mov cx,21               ; longitud del texto
      mov dh,5                ; fila
      mov dl,5                ; columna
      int 10h                 ; llamada BIOS

      mov ax,4C00h            ; vuelve al DOS
      int 21h

.data

      Text DB "Un poco más de texto”

end
```

Otra posibilidad, es escribir en memoria de video directamente.

```
;Ejem_12: TEXTO4.ASM
;escribe directamente en memoria de video
.model small
.stack
.code

      mov ax,0B800h    ; segmento del buffer de video
      mov es,ax        ; copia al segmento extra es
      xor di,di        ; “limpia” di, ES:DI apunta al video buffer
      mov ah,4         ; atributo - rojo
      mov al,"G"       ; carácter a memoria de video
      mov cx,4000      ; numero de veces a repetir
      cld              ; dirección – hacia adelante
      rep stosw        ; salida del carácter en ES:[DI]

      mov ax,4C00h     ; vuelve al DOS
      int 21h

end
```

Lo mismo pero ahora con una cadena de caracteres escribiendo en la memoria de video.

```
;Ejem_13: TEXTO5.ASM

.model small
.stack
.code
```

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

        mov ax,0B800h           ; segmento del video buffer
        mov es,ax               ; copia en el segmento extra
        mov ah,3                ; atributo - azul
        mov cx,21               ; longitud de la cadena
        mov si,OFFSET Texto     ; DX:SI apunta a la cadena
        xor di,di

        Wr_Char:

        lodsb                   ; siguiente carácter a al
        mov es:[di],al          ; escribe en memoria de video
        inc di                  ; mueve a la siguiente columna
        mov es:[di],ah          ; atributo del carácter
        inc di

        loop Wr_Char            ; bucle definido en cx

        mov ax,4C00h            ; vuelve al DOS
        int 21h

        .data

        Texto DB "Esto es un poco más de texto"

end
```

# VI – Listado de Interrupciones

| DIRECCION (hex) | INTERRUPCION(hex) | FUNCION |
|---|---|---|
| 0-3 | 0 | Division by zero |
| 4-7 | 1 | Single step trace |
| 8-B | 2 | Nonmaskable interrupt |
| C-F | 3 | Breakpoint instruction |
| 10-13 | 4 | Overflow |
| 14-17 | 5 | Print screen |
| 18-1F | 6,7 | Reserved |
| 20-23 | 8 | Timer |
| 24-27 | 9 | Keyboard interrupt |
| 28-37 | A,B,C,D | Reserved |
| 38-3B | E | Diskette interrupt |
| 3C-3F | F | Reserved |
| 40-43 | 10 | Video screen I/O |
| 44-47 | 11 | Equipment check |
| 48-4B | 12 | Memory size check |
| 4C-4F | 13 | Diskette I/O |
| 50-53 | 14 | Communication I/O |
| 54-57 | 15 | Cassete I/O |
| 58-5B | 16 | Keyboard input |
| 5C-5F | 17 | Printer Output |
| 60-63 | 18 | ROM Basic entry code |
| 64-67 | 19 | Bootstrap loader |
| 68-6B | 1A | Time of day |
| 6C-6F | 1B | Get control on keyboard break |
| 70-73 | 1C | Get control on timer interrupt |
| 74-77 | 1D | Pointer to video initialization table |
| 78-7B | 1E | Pointer to diskette parameter table |
| 7C-7F | 1F | Pointer to table for graphics characters ASCII 128-255 |
| 80-83 | 20 | DOS program terminate |
| 84-87 | 21 | DOS function call |
| 88-8B | 22 | DOS terminate address |
| 90-93 | 24 | DOS fatal error vector |
| 94-97 | 25 | DOS absolute disk read |
| 98-9B | 26 | DOS absolute disk write |
| 9C-9F | 27 | DOS terminate, fix in storage |
| A0-FF | 28-3F | Reserved for DOS |
| 100-1FF | 40-7F | Not used |
| 200-217 | 80-85 | Reserved by BASIC |
| 218-3C3 | 86-F0 | Used by BASIC interpreter |
| 3C4-3FF | F1-FF | Not Used |