

# Ensamblador 80x86



Departamento de  
**Arquitectura y**  
Tecnología de Computadores  
UNIVERSIDAD DE SEVILLA

1. Introducción
2. Registros internos
3. Almacenamiento de datos
4. Modos de direccionamiento
5. Juego de Instrucciones
6. Etiquetas, comentarios y directivas
7. Problemas

## 1. Introducción

El lenguaje ensamblador como cualquier lenguaje de programación es un conjunto de palabras que le indican al ordenador lo que tiene que hacer. Sin embargo la diferencia fundamental es que cada instrucción escrita en lenguaje ensamblador tiene una correspondencia exacta con una operación en el procesador. Por lo que son operaciones muy sencillas tales como: “Cargar 32 en el registro BX” o “Transferir el contenido del registro CL al CH”. Así pues, las palabras del lenguaje ensamblador son nemotécnicos que representan al *código máquina*, lenguaje que entiende el procesador.

### 1.1. Tamaño de los datos

En el 80x86 se definen los siguientes tamaños de datos:

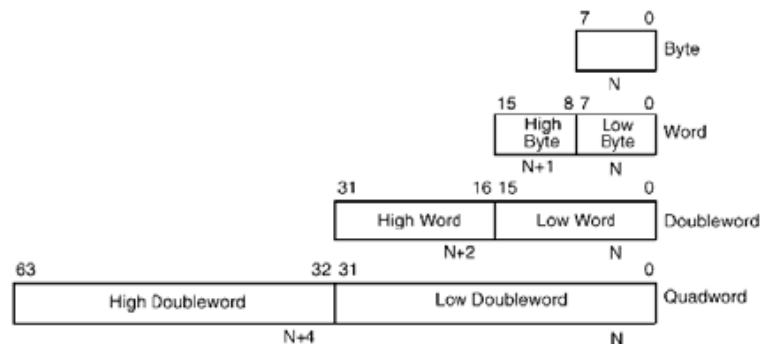


Fig. 1: Tipos de datos que puede manejar el procesador

## 2. Registros internos

El 80x86 dispone de 8 registros de propósito general (4 de datos y 4 índice), 6 registros de segmento, 1 registro de índice de programa y 1 registro de estado.

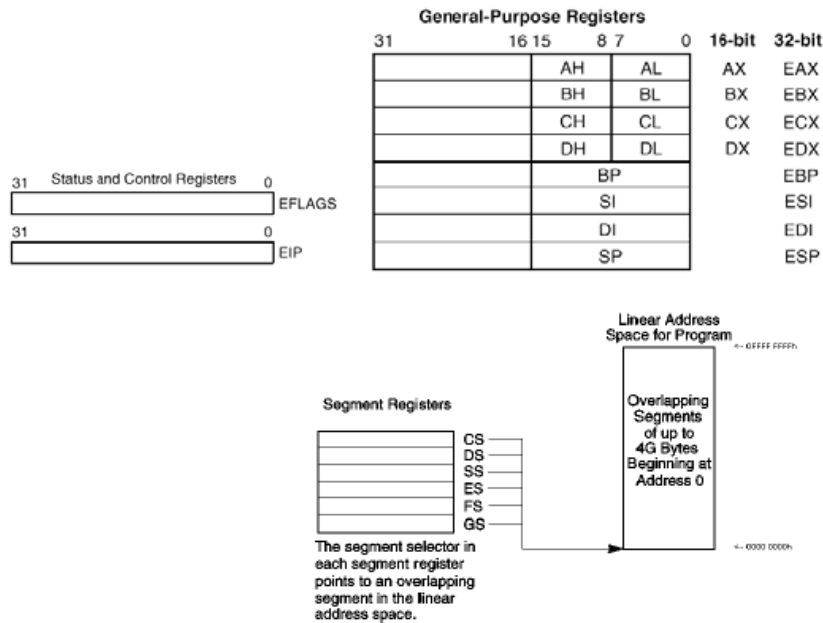
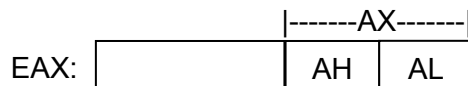


Fig. 2: Registros internos del 80x86

## 2.1. Registros de datos

Los registros de datos son de 32 bits, aunque están divididos, lo que permite su acceso a 8 bits o a 16 bits. Estos registros son de propósito general aunque todos tienen alguna función por defecto.

**EAX** (acumulador) se usa para almacenar el resultado de las operaciones, es el único registro con el que se puede hacer divisiones y multiplicaciones. Puede ser accedido en 8 bits como AH y AL; y a 16 bits como AX.



**EBX** (registro base) almacena la dirección base para los accesos a memoria. También puede accederse como BH y BL, o BX de la misma forma que el anterior.

**ECX** (contador) actúa como contador en los bucles de repetición. CL (parte baja del registro) almacena el desplazamiento en las operaciones de desplazamiento y rotación de múltiples bits.

**EDX** (datos) es usado para almacenar los datos de las operaciones. También es accesible en 8, 16 y 32 bits.

## 2.2. Registros de segmento

Los registros de segmento son de 16 bits y contienen el valor de segmento. El uso de segmentos para dividir el espacio de la memoria es una reminiscencia del procesador 8086, el primero de la familia. En los procesadores 80386 y posteriores los registros de segmentos son usados de manera automática por el procesador, sin que el programador deba preocuparse por el valor que en cada momento tiene, salvo para algunas instrucciones especiales.

**CS** (segmento de código) contiene el valor de segmento donde se encuentra el código. Actúa en conjunción con el registro EIP para obtener la dirección de memoria que contiene la próxima instrucción. Este registro es modificado por las instrucciones de saltos lejanos.



**DS** (segmento de datos) contiene el segmento donde están los datos.

**ES** (segmento extra de datos) es usado para acceder a otro segmento que contiene más datos.

**SS** (segmento de pila) contiene el valor del segmento donde está la pila. Se usa conjuntamente con el registro ESP para obtener la dirección donde se encuentra el último valor almacenado en la pila por el procesador.

**FS, GS:** se usan como registros de segmento adicionales

## 2.3. Registros de índice

Estos registros son de 32 bits y son usados como índices por algunas instrucciones, pueden ser accedidos solo a 16 bits. También pueden ser usados como operandos.

**ESI** (índice de origen) almacena el desplazamiento del operando de origen en memoria en algunos tipos de operaciones (operaciones con operandos en memoria).

**EDI** (índice de destino) almacena el desplazamiento del operando de destino en memoria en algunos tipos de operaciones (operaciones con operandos en memoria).

**ESP** (índice de pila) almacena el desplazamiento dentro del segmento de pila, y apunta al último elemento introducido en la pila. Se usa conjuntamente con el registro SS.

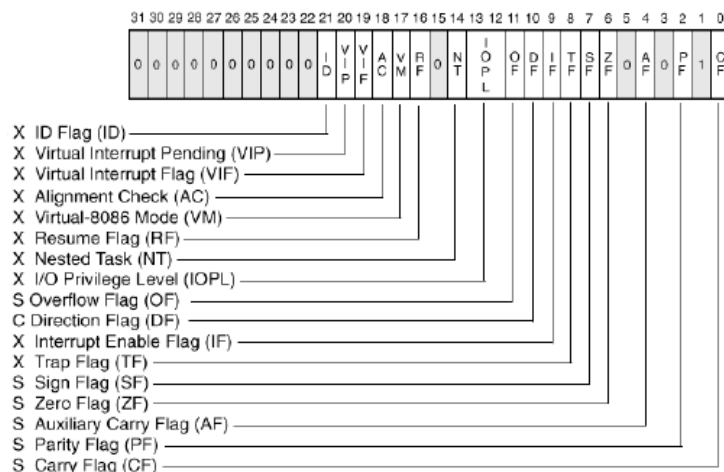
**EBP** (índice de base) se usa para almacenar desplazamiento en los distintos segmentos. Por defecto es el segmento de la pila

## 2.4. Registro contador de programa

**EIP** (índice de programa) almacena el desplazamiento dentro del segmento de código. Este registro junto al registro CS apunta a la dirección de la próxima instrucción. No puede ser usado como operando en operaciones aritmético/lógicas.

## 2.5. Registro de banderas (EFLAGS)

El registro de estado contiene una serie de *banderas* que indican distintas situaciones en las que se encuentra el procesador, de las cuales solo usaremos las que se detallan más abajo.



**Fig. 3: Registro de banderas (S: bit de estado, C: bit de control. X: bit del sistema)**

**OF** (desbordamiento) es el principal indicador de error producido durante las operaciones con signo. Vale 1 cuando:

- La suma de dos números con igual signo o la resta de dos números con signo opuesto producen un resultado que no se puede guardar (más de 32 bits).

- El bit más significativo (el signo) del operando ha cambiado durante una operación de desplazamiento aritmético.
- El resultado de una operación de división produce un cociente que no cabe en el registro de resultado.

**DF** (dirección en operaciones con cadenas) si es 1 el sentido de recorrido de la cadena es de izquierda a derecha, si es 0 irá en sentido contrario.

**IF** (indicador de interrupción) cuando vale 1 permite al procesador reconocer interrupciones. Si se pone a 0 el procesador ignorará las solicitudes de interrupción.

**TF** (modo traza) indica al procesador que la ejecución es paso a paso. Se usa en la fase de depuración.

**SF** (indicador de signo) solo tiene sentido en las operaciones con signo. Vale 1 cuando en una de estas operaciones el signo del resultado es negativo.

**ZF** (indicador de cero) vale 1 cuando el resultado de una operación es cero.

**AF** (acarreo auxiliar) vale 1 cuando se produce acarreo o acarreo negativo en el bit 3.

**PF** (paridad) vale 1 si el resultado de la operación tiene como resultado un número con un número par de bits a 1. Se usa principalmente en transmisión de datos.

**CF** (bit de acarreo) vale 1 si se produce acarreo en una operación de suma, o acarreo negativo en una operación de resta. Contiene el bit que ha sido desplazado o rotado fuera de un registro o posición de memoria. Refleja el resultado de una comparación.

### 3. Almacenamiento de datos

El 8086/88 usa el formato de almacenamiento denominado *"little endian"*, esto quiere decir que el byte menos significativa (LSB) del dato es guardada en la parte baja de la memoria.

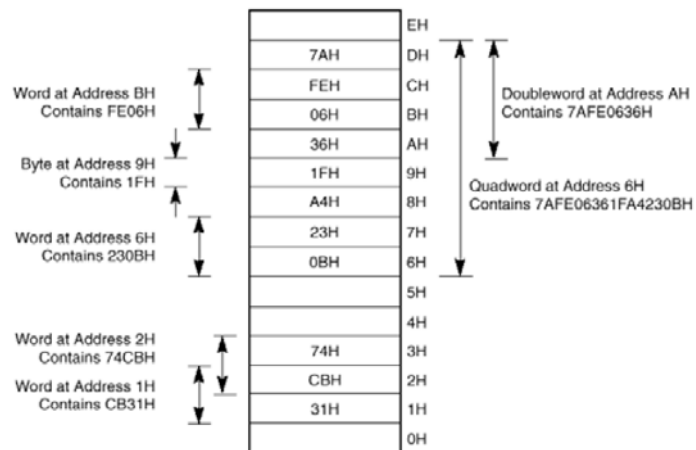


Fig. 4: Almacenamiento en memoria de los datos

Es importante tener esto en cuenta a la hora de acceder a los datos para operar con ellos.

### 4. Modos de direccionamiento

Los modos de direccionamiento indican la manera de obtener los operandos y son:

- Direccionamiento de registro
- Direccionamiento inmediato
- Direccionamiento relativos a EIP (se tratarán en los saltos)



## Direccionamiento de memoria

El tipo de direccionamiento se determina en función de los operandos de la instrucción.

La instrucción `MOV` realiza transferencia de datos desde un operando origen a un operando destino (se verá más con más detalle en los siguientes apartados). Su formato es el siguiente:

```
MOV destino, origen
```

### 4.1. Direccionamiento de registro

Cuando ambos operando son un registro.

Ejemplo:

```
MOV EAX, EBX ;transfiere el contenido de EBX en EAX
```

### 4.2. Direccionamiento inmediato

Cuando el operando origen es una constante.

Ejemplo:

```
MOV EAX, 500 ;carga en EAX el valor 500.
```

### 4.3. Direccionamiento de memoria

La dirección de memoria donde se encuentra el operando se especifica como una combinación de varios elementos, tal y como se muestra en el figura.

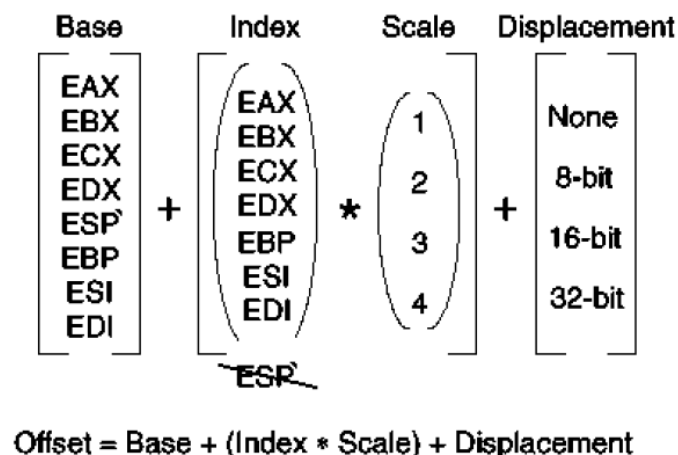


Fig. 5: Direccionamiento de memoria

Ejemplo:

```
MOV EAX, [EBX]; carga en EAX lo almacenado en la posición apuntada
                por EBX
```

## 5. Juegos de Instrucciones

Las instrucciones del 80x86 se pueden dividir en varios grupos:

Instrucciones de transferencia de datos

Instrucciones aritméticas

Instrucciones lógicas  
 Instrucciones de desplazamiento y rotaciones  
 Instrucciones de E/S  
 Instrucciones de control del flujo del programa  
 Instrucciones de cadena de caracteres

## 5.1. Instrucciones de transferencia de datos

Las instrucciones de transferencia de datos copian datos de un sitio a otro y son: MOV, XCHG, XLAT, LEA, LDS, LES, LAHF, SAHF, PUSH, PUSHF, POP, POPF.

**MOV** realiza la transferencia de datos del operando de origen al destino. Como ya hemos visto en la parte de los modos de direccionamiento, MOV admite todos los tipos de direccionamiento. Ambos operandos deben ser del mismo tamaño y no pueden estar ambos en memoria.

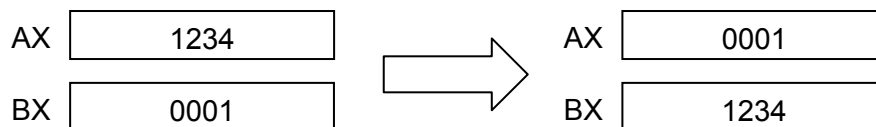
```
MOV reg, reg ; reg es cualquier registro.
MOV mem, reg ; mem indica una posición de memoria
MOV reg, mem
MOV mem, dato ; dato es una constante
MOV reg, dato
MOV seg-reg, mem ;seg-reg es un registro de segmento
MOV seg-reg, reg
MOV mem, seg-reg
MOV reg, seg-reg
```

**XCHG** realiza el intercambio entre los valores de los operandos. Puede tener operando en registros y en memoria:

```
XCHG reg, mem
XCHG reg, reg
XCHG mem, reg
```

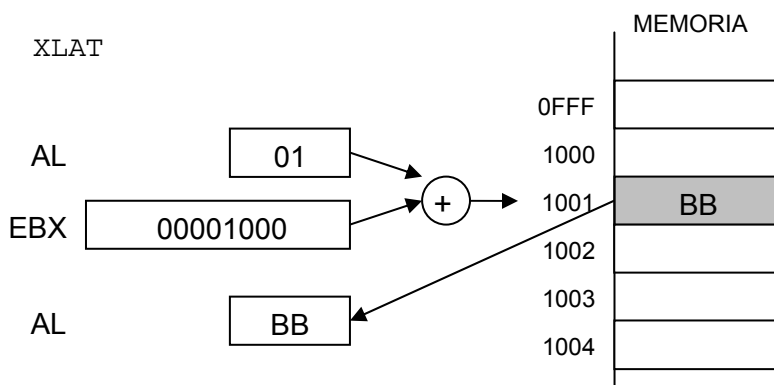
Ejemplo:

```
XCHG AX, BX
```



**XLAT** carga en AL el contenido de la dirección apuntada por [EBX+AL].

Ejemplo:



**LEA** carga en un registro especificado la dirección efectiva especificada como en el operando origen:

```
LEA reg, mem
```

Ejemplos:

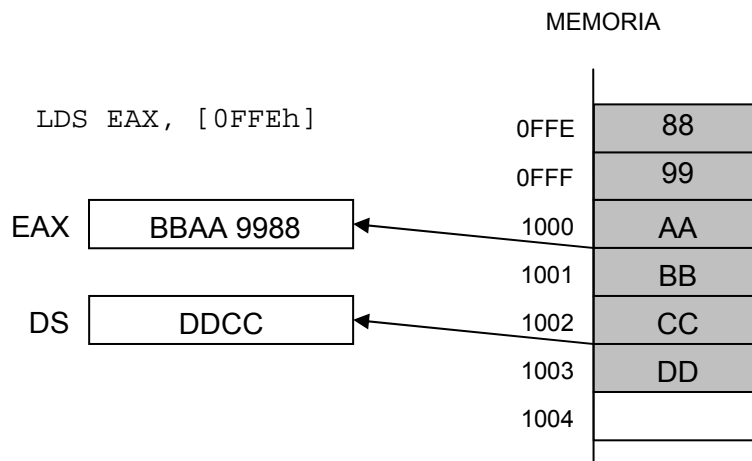
```
LEA EAX, [EBX] ; carga en AX la dirección apuntada por EBX.
LEA EAX, 3[EBP+ESI] ; carga en EAX la dirección resultado de
                    multiplicar por 3 la suma de los contenido de EBP y
                    SI.
```

Para este tipo de acceso (el del segundo ejemplo) la instrucción LEA es más eficiente, que usando la instrucción MOV e instrucciones de multiplicación.

**LDS, LES, LFS, LGS, LSS** carga el contenido de una dirección de memoria de 48 bits de la siguiente manera: la parte baja en el registro especificado y la parte alta en el registro DS, ES, FS, GS, SS respectivamente.

```
LDS reg, mem32
```

Ejemplo:



**PUSH** y **POP** realizan las operaciones de apilado y desapilado en la pila del procesador respectivamente, admiten todos los tipos de direccionamiento (excepto inmediato). Los operandos deben ser siempre de 16 bits

```
PUSH reg
PUSH mem
PUSH seg-reg
POP reg
POP mem
POP seg-reg
```

Ejemplos:

```
PUSH AX ; envía a la pila AX
POP DS ; carga el primer elemento de la pila en DS
```

**PUSHF** y **POPF** apila y desapila el registro de estado, respectivamente.

**LAHF** carga la parte baja del registro de estado en AH.

**SAHF** carga AH en el la parte baja del registro de estado.

## 5.2. Instrucciones aritméticas

Este tipo de instrucciones realizan operaciones aritméticas con los operandos. Y son: ADD, ADC, DAA, AAA, SUB, SBB, DAS, AAS, NEG, MUL, IMUL, AAM, DIV, IDIV, AAD, CBW, CWB, INC, DEC.

**ADD** y **ADC** realizan la suma y la suma con acarreo (bit CF del registro de estado) de dos operandos, respectivamente, y guardan el resultado en el primero de ellos. Admiten todos los tipos de direccionamiento (excepto que ambos operando estén en memoria).

```
ADD/ADC reg, reg
ADD/ADC mem, reg
ADD/ADC reg, mem
ADD/ADC reg, inmediato
ADD/ADC mem, inmediato
```

Ejemplo:

```
; J = 34+f
MOV EAX, F
ADD EAX, 34
MOV J, EAX
```

Estas instrucciones afectan a los bits OF, SF, ZF, AF, PF, CF del registro de estado.

**SUB** y **SBB** realizan la resta y la resta con acarreo, respectivamente, de dos operandos y guardan el resultado en el primero de ellos. Admiten todos los modos de direccionamiento, excepto dos operando en memoria.

```
SUB/SBB reg, reg
SUB/SBB mem, reg
SUB/SBB reg, mem
SUB/SBB reg, inmediato
SUB/SBB mem, inmediato
```

Ejemplo:

```
; J = F-34
MOV AX, F
SUB AX, 34
MOV J, AX
```

Estas instrucciones afectan a los bits OF, SF, ZF, AF, PF, CF del registro de estado.

**NEG** realiza la operación aritmética de negado de un operando y guarda el resultado en el mismo operando. Admite todos los tipos de direccionamiento, excepto inmediato.

```
NEG reg
NEG mem
```

La operación que realiza es: 0 – operando.

Afecta a todos los bits del registro de estado, poniendo el bit AF a 1.

**MUL** e **IMUL** realizan la multiplicación y multiplicación con signo, respectivamente, de contenido de EAX y del operando indicado, guardando el resultado en EAX, para operaciones de 16 bits; y en EDX:EAX para operaciones de 32 bits. Los formatos son:

```
MUL/IMUL reg
MUL/IMUL mem
```

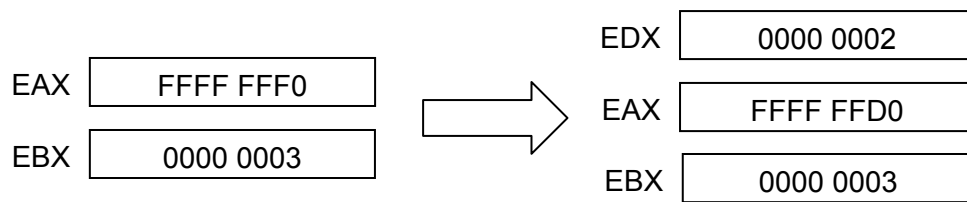
Ejemplo:

```
MOV EAX, FFFF FFF0h
```





```
MOV EBX, 3
MUL EBX          ; EDX:EAX = FFFF FFF0h*3
```



**DIV** e **IDIV** realizan la división y la división con signo, respectivamente.

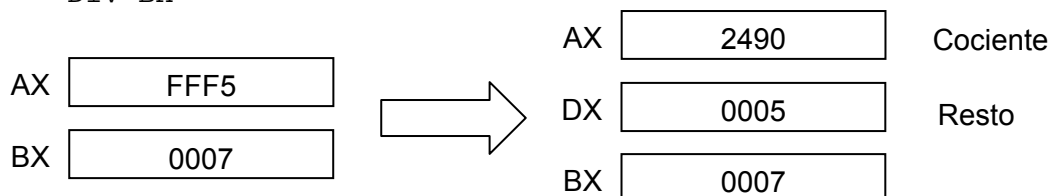
Dependiendo del tamaño de los operandos el resultado se guarda tal y como se muestra en la tabla.

Long. op	Dividendo	Operando (divisor)	Cociente	Resto
8 bits	AX	Op	AL	AH
16 bits	EAX	Op	AX	DX
32 bits	EDX:EAX	Op	EAX	EDX

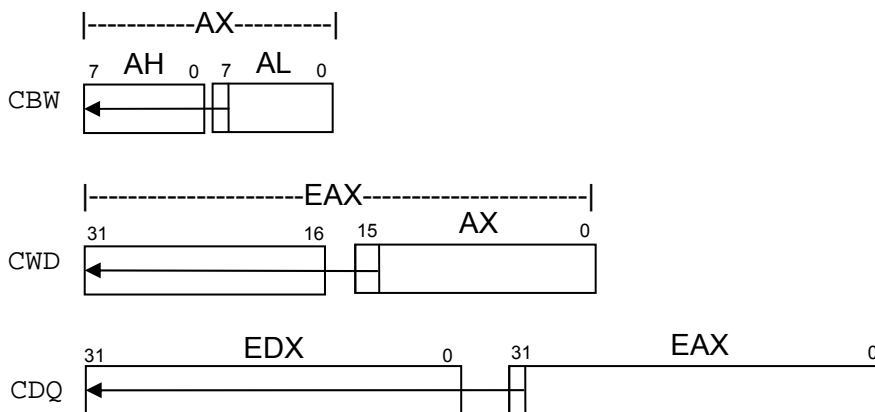
```
DIV/IDIV reg
DIV/IDIV mem
```

Ejemplo:

```
MOV AX, FFF1h
MOV BX, 7
DIV BX
```



**CBW**, **CWD** y **CDQ** realizan la extensión del bit de signo de byte a WORD, de WORD a DWORD y de DWORD a QUADWORD actuando sobre AX, EAX o EDX:EAX respectivamente. Tal y como se muestra en el figura. Replica el bit de signo en las posiciones altas del dato.



Ejemplo:

```
MOV AL, -3
CBW          ; AX = -3
```

**INC** y **DEC** realizan las operaciones de incremento y decremento, respectivamente, de un operando, guardando el resultado en el mismo operando. Admiten todos los modos de direccionamiento excepto el inmediato.

```
INC/DEC reg
INC/DEC mem
```

Afectan a todos los bits de estado del registro de estado.

### 5.3. Instrucciones lógicas

Realizan las operaciones lógicas y son: OR, XOR, AND, NOT, TEST, CMP.

**OR**, **XOR** y **AND** realizan las operaciones lógicas “O”, “O exclusiva” e “Y”, respectivamente, de dos operandos, guardando el resultado en el primero de ellos. Estas operaciones son bit a bit. La tabla de verdad de estas funciones es:

Operandos		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Admiten todos los modos de direccionamiento excepto los dos operandos en memoria.

```
AND/OR/XOR reg, reg
AND/OR/XOR reg, mem
AND/OR/XOR mem, reg
AND/OR/XOR reg, inmediato
AND/OR/XOR mem, inmediato
```

Afectan a los bits SF, ZF, PF del registro de estado. Además ponen a cero los bits CF y OF.

**NOT** realiza la operación de negado lógico de los bits del operando, guardando el resultado en el mismo operando. Admite todos los modos direccionamiento excepto inmediato.



NOT reg  
NOT mem

No afecta a ningún bit del registro de estado.

## 5.4. Instrucciones de comparación

Estas instrucciones realizan funciones de comparación sin guardar el resultado, pero si afecta al registro de estado (no cambian a los operandos). Son muy útiles en las instrucciones de salto que se verán más adelante.

**TEST** realiza la operación lógica “Y” de dos operandos, pero NO afecta a ninguno de ellos, SÓLO afecta al registro de estado. Admite todos los tipos de direccionamiento excepto los dos operandos en memoria

TEST reg, reg  
TEST reg, mem  
TEST mem, reg  
TEST reg, inmediato  
TEST mem, inmediato

Afecta a todos los bits del registro de estado, de la misma manera que la instrucción AND.

**CMP** realiza la resta de los dos operandos (como la instrucción SUB) pero NO afecta a ninguno de ellos, SÓLO afecta al registro de estado. Admite todos los modos de direccionamiento, excepto los dos operandos en memoria.

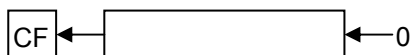
CMP reg, reg  
CMP reg, mem  
CMP mem, reg  
CMP reg, inmediato  
CMP mem, inmediato

Se usa con las instrucciones de salto que veremos más adelante.

## Instrucciones de desplazamiento y rotaciones

Realizan operaciones de desplazamiento y rotaciones de bits, y son: SAL/SHL, SAR, SHR, ROL, ROR, RCL, RCR.

**SAL/SHL** realiza el desplazamiento a la izquierda del primer operando tantos bits como indique el segundo operando, introduciendo un 0 y guardando el bit que sale en el bit CF del registro de estado.

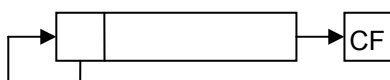


Admite los siguientes formatos:

SAL/SHL reg, 1; desplaza 1 vez el contenido de reg  
SAL/SHL mem, 1  
SAL/SHL reg, CL; desplaza tantas veces el contenido de reg como indique CL.  
SAL/SHL mem, CL

Afecta a los bits OF, CF del registro de estado.

**SAR** realiza el desplazamiento a la derecha del operando, repitiendo el bit de signo y guardando el resultado en el bit CF del registro de estado.



Admite los siguientes formatos:

```
SAR reg, 1; desplaza 1 vez el contenido de reg
SAR mem, 1
SAR reg, CL; desplaza tantas veces el contenido de reg como indique
              CL.
SAR mem, CL
```

Afecta a todos los bit del registro de estado.

**SHR** realiza el desplazamiento a la derecha del operando, introduciendo un 0 y guardando el resultado en el bit CF del registro de estado.

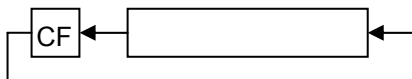


Admite los siguientes formatos:

```
SHR reg, 1; desplaza 1 vez el contenido de reg
SHR mem, 1
SHR reg, CL; desplaza tantas veces el contenido de reg como indique
              CL.
SHR mem, CL
```

Afecta a los bit OF, CF del registro de estado.

**RCL** realiza la rotación a la izquierda de los bits del operando a través del bit CF (acarreo) del registro de estado.

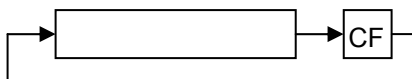


Admite los siguientes formatos:

```
RCL reg, 1; desplaza 1 vez el contenido de reg
RCL mem, 1
RCL reg, CL; desplaza tantas veces el contenido de reg como indique
              CL.
RCL mem, CL
```

Afecta a los bit OF, CF del registro de estado.

**RCR** realiza la rotación a la derecha de los bits de operando a través del bit CF del registro de estado.

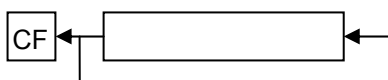


Admite los siguientes formatos:

```
RCR reg, 1; desplaza 1 vez el contenido de reg
RCR mem, 1
RCR reg, CL; desplaza tantas veces el contenido de reg como indique
              CL.
RCR mem, CL
```

Afecta a los bit OF, CF del registro de estado.

**ROL** realiza la rotación a la izquierda de los bits del operando, ignorando el bit CF del registro de estado, aunque en CF se almacena el bit que se rota.

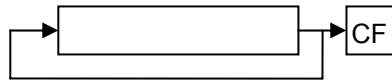


Admite los siguientes formatos:

```
ROL reg, 1; desplaza 1 vez el contenido de reg
ROL mem, 1
ROL reg, CL; desplaza tantas veces el contenido de reg como indique
              CL.
ROL mem, CL
```

Afecta a los bit OF, CF del registro de estado.

**ROR** realiza la rotación a la derecha de los bits del operando, ignorando el bit CF del registro de estado, aunque en CF se almacena el bit que se rota.



Admite los siguientes formatos:

```
ROL reg, 1; desplaza 1 vez el contenido de reg
ROL mem, 1
ROL reg, CL; desplaza tantas veces el contenido de reg como indique
              CL.
ROL mem, CL
```

Afecta a los bit OF, CF del registro de estado.

**NOTA:** Las instrucciones SHL/SAL y SAR se suelen usar para hacer divisiones y multiplicaciones, respectivamente, por números potencia de dos (2, 4, 8, 16, 32, 64 y 128), de manera más eficiente que las instrucciones DIV y MUL.

Ejemplo:

```
MOV AX, 40h
MOV CL, 2
SHL AX, CL
; es equivalente y mas eficiente que
MOV AX, 40h
MOV DX, 00h
MOV BX, 04h
DIV BX
```

## 5.5. Instrucciones de E/S

Se usan para la comunicación con los dispositivos periféricos. Y son IN, OUT.

**IN** lee de un puerto (sólo si la dirección del puerto es menor que 255). Admite las siguientes formas:

```
IN AX, inmediato    ;obtiene un WORD del puerto especificado y lo
                      guarda en AX
IN AX, DX            ;obtiene un WORD del puerto especificado en DX y lo
                      guarda en AX
```

**OUT** escribe en un puerto (sólo si la dirección del puerto es menor que 255). Admite las siguientes formas:

```
OUT inmediato, AX   ;escribe un WORD (contenido en AX) en el
                      puerto especificado
OUT DX, AX           ;escribe un WORD (contenido en AX) en el puerto
                      especificado en DX.
```

## 5.6. Instrucciones de control del programa

Se utilizan para el control del programa, son instrucciones de salto, bucles y llamadas a procedimientos.

### Instrucciones de salto

Estas instrucciones permiten saltar a otras partes del código. Todas cambian el registro EIP (contador de programa) y el registro CS (segmento de código) si es un salto lejano. Un salto es lejano cuando la dirección a la que se salta no está en el mismo segmento de código.

Existen dos tipos de saltos: los absolutos; en lo que se especifica la dirección absoluta a la que se salta; y los relativos; que son saltos hacia delante o hacia atrás desde el valor de EIP.

**JMP** realiza un salto incondicional a la dirección especificada. La siguiente tabla relaciona los tipos de saltos y los argumentos que puede tomar esta instrucción.

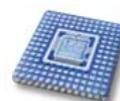
	Cercano	Lejano
Relativo	8 ó 16 bits	-
Absoluto	Mem reg	Inmediato mem

**Saltos condicionales** estas instrucciones realizan el salto a la dirección especificada en función de si se cumple o no una condición. Para evaluar la condición se considera el registro de estado, esto quiere decir que la condición depende directamente de la instrucción anterior. En la siguiente tabla se presentan estas instrucciones en función del tipo de operandos y la condición que se quiere evaluar.

Condición	Sin signo	Con signo
=	<b>JE/JZ</b>	<b>JE/JZ</b>
>	<b>JA/JNBE</b>	<b>JG</b>
<	<b>JB/JNAE</b>	<b>JL</b>
≥	<b>JAE/JNB</b>	<b>JGE</b>
≤	<b>JBE/JNA</b>	<b>JLE</b>
≠	<b>JNE/JNZ</b>	<b>JNE/JNZ</b>

También existen instrucciones que evalúan sólo un bit del registro de estado.

BIT	= 1	= 0
ZF	<b>JZ</b>	<b>JNZ</b>
CF	<b>JC</b>	<b>JNC</b>
OF	<b>JO</b>	<b>JNO</b>
SF	<b>JS</b>	<b>JNS</b>



SP

JP

JNP

**Instrucción de llamada a procedimiento CALL y RET**

La instrucción CALL se usa para realizar una llamada a un procedimiento y la instrucción RET se usa para volver de un procedimiento. Cuando se realiza una llamada a procedimiento con CALL, se guardan en la pila el valor de EIP en caso de un salto corto, y de CS e EIP en caso de un salto lejano.

Cuando se ejecuta la instrucción RET se recuperan de la pila los valores de EIP o de CS e EIP dependiendo del caso.

Al salir de un procedimiento es necesario dejar la pila como estaba; para ello podemos utilizar la instrucción pop, o bien ejecutar la instrucción RET n donde n es el número de posiciones que deben descartarse de la pila.

**Bucles**

Las instrucciones de bucle se usan para realizar estructuras repetitivas, y utilizan el registro CX como contador.

**LOOP** esta instrucción hace que el programa salte a la dirección especificada (salto dentro del segmento), mientras que CX sea distinto de 0 y decrementa CX en 1 cada vez.

```
LOOP salto
```

Ejemplo:

```
                MOV CX, 100
COMIENZO:      ...
                ...
                LOOP COMIENZO; este bucle se repite 100
```

**LOOPNE/LOOPNZ** esta instrucción salta a la dirección especificada mientras que ECX sea distinto de 0 y si ZF = 0.

```
LOOPNE/LOOPNZ salto
```

Esta instrucción proporciona una ruptura del bucle adicional.

**LOOPE/LOOPZ** esta instrucción actúa como la anterior pero la condición adicional es ZF = 1.

```
LOOPE/LOOPZ salto
```

**JCXZ** esta instrucción realiza un salto si ECX = 0.

```
JCXZ salto
```

Ninguna de estas instrucciones afectan al registro de estado.

**5.7. Instrucciones de cadena de caracteres**

Realizan operaciones con cadenas de caracteres. Antes de ver las instrucciones que manipulan cadenas, es necesario comentar el uso de los prefijos de repetición, modificadores que sólo se pueden usar con las instrucciones de manipulación de cadenas.

**REP** este modificador repite la instrucción a la que acompaña mientras que ECX sea distinto de 0 (decrementa CX cada vez). Las instrucciones con las que se puede usar son MOVS, MOVDSW o STOS.

```
                MOVS destino, fuente
REP  MOVS destino, fuente
```



**REPE/REPZ** este modificador repite la instrucción a la que acompaña mientras que ECX sea distinto de 0 y ZF = 1 (decrementa ECX cada vez). Las instrucciones con las que se puede usar son CMPS o SCAS.

```
CMPS destino, fuente
REPE/REPZ CMPS destino, fuente
```

**REPNE/REPZ** este modificador repite la instrucción a la que acompaña si ECX es distinto de 0 y ZF = 0 (decrementa CX cada vez). Las instrucciones con las que se puede usar son CMPS o SCAS.

```
REPNE/REPZ CMPS destino, fuente
```

**MOVS/MOVSX/MOVSQ** copia un byte, WORD o DWORD de una posición a otra de la memoria.

```
MOVS/MOVSX/MOVSQ destino, fuente
```

donde destino es ES:EDI y fuente es DS:ESI, lo que quiere decir que antes de utilizar la instrucción hay que cargar en SI y DI los valores apropiados.

Ejemplo:

```
LEA ESI, fuente
LEA EDI, ES:destino
MOV CX, 100
REP MOVS destino, fuente
```

Por lo tanto, para usar esta instrucción hay que seguir los siguientes pasos:

- 1.- Colocar el bit DF (dirección de recorrido) al valor correcto (lo veremos más adelante).
- 2.- Cargar en SI el desplazamiento de la fuente.
- 3.- Cargar en DI el desplazamiento del destino.
- 4.- Cargar en CX el número de elementos a mover.
- 5.- Ejecutar la instrucción MOVS/MOVSX con el prefijo REP.

Esta instrucción no afecta al registro de estado.

**CMPS** realiza la comparación de dos cadenas, devuelve el resultado en el registro de estado.

```
CMPS destino, fuente
```

Hay que realizar los mismos pasos para usar esta instrucción que en el caso de la instrucción MOVS/MOVSX, la única diferencia es que el modificador que usa es REPE/REPZ o REPNE/REPZ.

Esta instrucción afecta a todos los bits del registro de estado.

**SCAS/SCASX/SCASQ** localiza el valor contenido en AL, AX o EAX (según sea byte, WORD o DWORD) en una cadena, si encuentra el elemento, devuelve en EDI el desplazamiento del siguiente elemento.

```
SCAS/SCASX/SCASQ destino
```

Al igual que las instrucciones anteriores es necesario cargar en EDI el desplazamiento del primer elemento de la cadena.

Ejemplo

```
; busca en CADENA un espacio en blanco
LEA EDI, ES:CADENA
MOV AL, ' '
MOV CX, 100
REP SCAS CADENA
```

Esta instrucción afecta a todos los bits del registro de estado.



**LODS/LODSW/LODS** transfiere un elemento de una cadena (fuente) a AL, AX o EAX, respectivamente.

`LODS/LODSW/LODS fuente`

Esta instrucción también necesita que la fuente esté cargada en ESI.

**STOS/STOSW/STOS** transfiere el contenido de AL, AX o EAX, respectivamente, a una cadena (destino).

`STOS/STOSW/STOS destino`

También debe cargarse en EDI el desplazamiento de la cadena, y puede usarse con el modificador REP.

Ejemplo:

```
; busca en una cadena un 0 y si lo encuentra rellena las siguientes
5 posiciones con ceros.
    LEA DI, ES:CADENA
    MOV AX, 0
    MOV CX, 200
    REPNE SCASW
    JCXZ no_encon
    SUB DI, 2
    MOV CX, 6
    REP STOS CADENA
no_encon: ...
...
```

## 5.8. Otras instrucciones

**HLT** parada del procesador, solo es posible salir de este estado reiniciando o por medio de una interrupción externa.

`HLT`

**LOCK** bloquea el acceso al bus por parte de otro dispositivo mientras dure la ejecución de la instrucción a la que acompaña.

`LOCK instrucción`

**WAIT** genera estados de espera en el procesador hasta que se active la línea TEST, generalmente usada por el coprocesador.

`WAIT`

**CLC/STC** pone a 0 ó a 1, respectivamente, el bit CF del registro de estado.

`CLC/STC`

**CMC** cambia el valor del bit CF del registro de estado.

`CMC`

**CLI/STI** pone a 0 ó a 1, respectivamente, el bit IF del registro de estado.

`CLI/STI`

**CLD/STD** pone a 0 ó a 1, respectivamente, el bit DF del registro de estado. Este bit es el que se usa para recorrer una cadena de manera ascendente o descendente en memoria.

`CLD/STD`

**NOP** no operación, hace que el procesador ejecute NADA.

## 6. Etiquetas, comentarios y directivas

Las **etiquetas** asignan un nombre a una instrucción. Esto permite hacer referencia a ellas en el resto del programa. Pueden tener una máximo de 31 caracteres y deben terminar en “:”.

Los **comentarios** permiten describir las sentencias de un programa, facilitando su comprensión. Comienzan por “;”, el ensamblador ignora el resto de la línea.

Ejemplo:

```
INI_CONT: MOV CX, DI ; inicia el contador
```

Las **directivas** son comandos que afectan al ensamblador, no al procesador. Se puede usar para preparar segmentos y procedimientos, definir símbolos, reservar memoria, etc. La mayoría de las directivas no generan código objeto.

Las directivas más comunes son:

**.386** para indicar el juego de instrucciones que se va a usar, se pueden especificar para todos los procesadores de la familia, nosotros veremos el 80386.

**.OPTION CASEMAP:NONE:** hace que el ensamblador distinga entre mayúsculas y minúsculas en la tabla de símbolos.

Las **directivas simplificadas** se utilizan para la definición de segmentos.

**.MODEL** para usar las directivas simplificadas es necesario incluir esta directiva que define el modelo de memoria que debe usarse. Algunos de los argumentos que puede tomar son:

TINY: para programa con un solo segmento para datos y código (tipo .COM)

SMALL: para programas con un solo segmento de datos (64K, incluida la pila) y otro de código (64K).

LARGE: varios segmentos de datos y código (1Mb para cada uno).

MEDIUM: Varios segmentos de código y 1 de datos.

COMPACT: 1 segmento de código y varios de datos.

**FLAT: modelo de memoria plano (es el que usaremos siempre)**

Con esta directiva se preparan todos los segmentos y el ensamblador reconoce, a partir de este momento, las directivas .DATA, .STACK y .CODE.

**.STACK n** sirve para fijar un tamaño **n** del segmento de pila, por defecto 1K.

**.DATA** abre el segmento de datos.

**.CODE** abre el segmento de código, al final código debe aparecer **END**.

Una vez inicializado los segmento se permite usar los símbolos @CODE y @DATA en lugar del nombre de los segmentos de código y datos respectivamente.

Justo después de la directiva .CODE hay que inicializar el segmento de datos (ya que la directiva no genera código):

```
MOV AX, @DATA
MOV DS, AX
```

**PROC** y **ENDP** definen un procedimiento (o subprograma).

**EQU** asigna nombre a números, combinaciones de direccionamiento y a otras cosas que se vayan a usar repetidas veces en el código.

Ejemplo:

```
K EQU 1024           ; especifica una constante
TABLA EQU DS:[BP][SI] ; especifica una combinación de direc.
VELOCIDAD EQU TOCINO ; da un nombre alternativo
```

**DB, DW y DD** se usan para asignar espacio a las variables en memoria. DB tamaño byte, DW tamaño WORD y DD tamaño DWORD.

Ejemplo:

```
MSG_ERROR DB 'has cometido un error, zoquete' ;reserva para MSG
           constante de tamaño byte con valor 255, este es un caso
           especial de la directiva DB, que también puede ser usada
           para declarar cadenas de caracteres.
PESO_MEDIO DW ? ; Reserva para PESO_MEDIO tamaño DWORD pero no
                inicializa el valor (?)
```

**n DUP** reserva tantas posiciones del tamaño que se indique (DB, DW, DD) como indique n.

Ejemplo de un código fichero .asm:

```
.386
.MODEL FLAT
.OPTION CASEMAP:NONE
.STACK 100H
.DATA
    max EQU 10 ; declara un constante con valor 100
    cad DB DUP 'racecar', 00H ; declara una variable llamada cad
                               ; con valor inicial la cadena "racecar"
    dac DB max DUP ? ; declara una variable llamada cad de tamaño
                     ; max sin valor inicial
.CODE
    MOV AX, @DATA
    MOV DS, AX
    ...
    ...
    END
```

## 7. Bibliografía

- ▶ John H. Crawford y Patrick P. Gelsinger . *Programación del 80386/87* Ed. Anaya. 1991
- ▶ Jon Beltrán de Heredia. *Ensamblador de los 80x86*. Ed Anaya. 1995