

# **ASSEMBLER DESDE CERO**

e

## **INTERRUPCIONES**

**MARIO CARLOS GINZBURG**

INGENIERO ELECTRONICO (UBA)

DIRECTOR DE LA CATEDRA DE "ESTRUCTURA  
DEL COMPUTADOR" E INVESTIGADOR EN LA  
FACULTAD DE INGENIERIA DE LA  
UNIVERSIDAD DE BUENOS AIRES

A Piyi, Jerónimo y Rafael

A mis alumnos



**INTRODUCCION GENERAL A LA INFORMATICA:**

**3 ASSEMBLER DESDE CERO**

**M.C.Ginzburg**

Es propiedad - Queda hecho el depósito que marca la ley  
Impreso en la Argentina - Printed In Argentina

DERECHOS RESERVADOS © 2000

**I.S.B.N. 987-43-2203-9**

Impreso en talleres prensa & acabado, Concepción Arenal 4866, Capital Federal  
en el mes de agosto de 2000

No se permite la reproducción total o parcial de esta obra, ni el almacenamiento en un sistema de informática,  
ni transmisión en cualquier forma o por cualquier medio electrónico, mecánico, fotocopia, registro u otros  
medios, sin el permiso previo y la autorización escrita del autor.

## INDICE

UTILIZACION DEL LENGUAJE ASSEMBLER .....	1
REGISTROS DE LA UCP A UTILIZAR .....	1
EJERCICIOS .....	2
EJERCICIOS DONDE SE DEFINEN INSTRUCCIONES DE SALTO.....	4 a 8
EJERCICIO CON UNA LISTA (VECTOR) .....	9
EJERCICIO DONDE SE DEFINE LA INSTRUCCIÓN DE COMPARACIÓN Y DOS LISTAS	10
EJERCICIOS CON NUEVAS INSTRUCCIONES DE SALTO .....	12 a 15
EJERCICIOS CON SUMA DE NUMEROS NATURALES Y ENTEROS DE UNA LISTA.....	16 a 17
EJERCICIO CON TRES LISTAS.....	17
EJERCICIO PARA VERIFICAR MEMORIA Y EXTENDER SIGNO.....	19
EJERCICIO CON MODO INDIRECTO POR REGISTRO CON DESPLAZAMIENTO MEDIANTE REGISTRO Y EJERCICIO CON DIVISIONES .....	20
EJERCICIO CON USO DE INSTRUCCIONES AND, ROL Y OR.....	22
EJERCICIOS CON REORDENAMIENTO DE LISTAS .....	23 a 26
EJERCICIOS DE MULTIPLICACION Y DE SUMA EN BCD .....	25
EJERCICIO PARA OBTENER RAIZ CUADRADA .....	27
EJERCICIOS PARA MULTIPLICAR ENTEROS .....	28
DIRECCIONES EFECTIVAS Y REGISTROS DE SEGMENTO .....	29 a 30
LLAMADOS A SUBRUTINAS Y USO DE LA PILA .....	31 a 38
PASAJE DE PARAMETROS .....	36 a 37
INTERRUPCIONES POR SOFTWARE (INSTRUCCIÓN INT).....	39 a 41
INTERRUPCIONES POR HARDWARE .....	41
MNEMONICOS DE LAS INSTRUCCIONES MAS USADAS .....	43

CUSPIDE

LIBROS



## PROLOGO

Esta corta edición previa salió a pedido de los alumnos, dada la inexistencia de textos que expliquen en forma progresiva la enseñanza del Assembler.

Se trata en realidad de la mitad del texto final de la Unidad 3, que usa el programa Debug del DOS para ensamblar los programas escritos en Assembler. El objetivo de esta mitad es que el alumno incorpore las estructuras básicas de programación en Assembler, conozca las instrucciones más comunes, maneje los flags y se vincule con el hardware. Se trata de aportarle las "7 notas musicales", para que pueda combinarlas en estructuras cada vez más complejas. Esto es, se definen los modos de direccionamiento básicos que combinados permiten operar cualquier estructura de datos.

También se ha tratado que los ejercicios vayan progresando en complejidad.

No podía faltar el llamado a subrutinas con el consiguiente uso de la pila, instrucciones PUSH y POP, y el pasaje de parámetros. Dado que tanto las interrupciones por hardware (mediante las líneas IRQ), como las interrupciones por software (mediante la instrucción INT xx) son formas de llamado a subrutinas (del BIOS o del sistema operativo), por extensión se tratan éstas, indicando su proceso en detalle.

Si bien el Assembler está dedicado a procesadores de Intel (dada la facilidad de acceder al DOS desde cualquier PC para practicarlo) los conceptos desarrollados pueden aplicarse a otros microprocesadores y microcontroladores.

La otra mitad (en preparación), a cargo del Ing. Carlos Robello, trata en esencia los mismos ejercicios desarrollados en la primer mitad, pero realizados con directivas para programas ensambladores más poderosos, como el TASM y el MASM. Esta mitad se incorporará a la primera en una próxima edición completa.

De esta forma, el alumno primero se concentra en lo fundamental del lenguaje, para luego manejar un Assembler de más alto nivel, con mayores posibilidades y flexibilidad de programación. Asimismo está planeado agregar nuevos ejercicios más complejos.

## UTILIZACION DEL LENGUAJE ASSEMBLER

Conviene aclarar que en nuestro país se habla de programar en "assembler", siendo que en realidad el lenguaje se denomina *assembly*. Assembler ("ensamblador") es el programa traductor de assembly a código de máquina, por lo que lenguaje assembly puede traducirse como lenguaje "ensamblable" o lenguaje para ensamblador. Siguiendo la costumbre, seguiremos llamando assembler al lenguaje simbólico de máquina, y ensamblador al programa traductor citado.

Recordemos (Historia de la Computación, unidad 1) que el assembler fue el primer lenguaje simbólico creado. Permitió escribir programas desde el teclado de un computador (salida código ASCII), lo cual suponía la existencia de un programa ensamblador, para pasar los símbolos en ASCII a código de máquina.

Cada modelo de procesador tiene su correspondiente lenguaje assembler, y su ensamblador. Por lo general, los nuevos modelos de un mismo fabricante conservan instrucciones en assembler de modelos anteriores. Así, un Pentium tiene, en su repertorio de instrucciones en assembler, instrucciones del 80286, por razones de compatibilidad.

Si bien cada fabricante de microprocesadores define notaciones y aspectos particulares para simbolizar instrucciones en assembler, con un poco de práctica no resulta difícil para quien sabe programar un determinado assembler, pasar a otro. Dado que la mayoría de las PC usa procesadores de Intel o sus clones, y que desde una PC se puede programar cómodamente en assembler, desarrollaremos el assembler de Intel como lenguaje representativo.

En el presente, se programa en Assembler para distintas aplicaciones. Lo más corriente quizás sea programar porciones de un programa que necesitan ser ejecutadas en corto tiempo, siendo que la mayor parte del programa se desarrolla con un lenguaje de alto nivel (C, Pascal, Basic, etc). Esto se debe a que un Compilador para CISC (unidad 1), al pasar de alto nivel a código de máquina genera código en exceso, en comparación con el obtenido a partir de assembler, lo cual redundaría en mayores tiempos de ejecución de porciones de programa cuyo tiempo de ejecución es crítico. Como contrapartida, en general lleva más tiempo programar en assembler que en alto nivel.

También suele usarse el assembler para desarrollar manejadores de periféricos, y para controlar directamente el hardware, dada la flexibilidad de este lenguaje.

El lenguaje assembler, por otra parte, es una herramienta imprescindible para dominar a fondo el funcionamiento de un computador, de modo de sacarle el máximo provecho.

### REGISTROS DE LA UCP A UTILIZAR

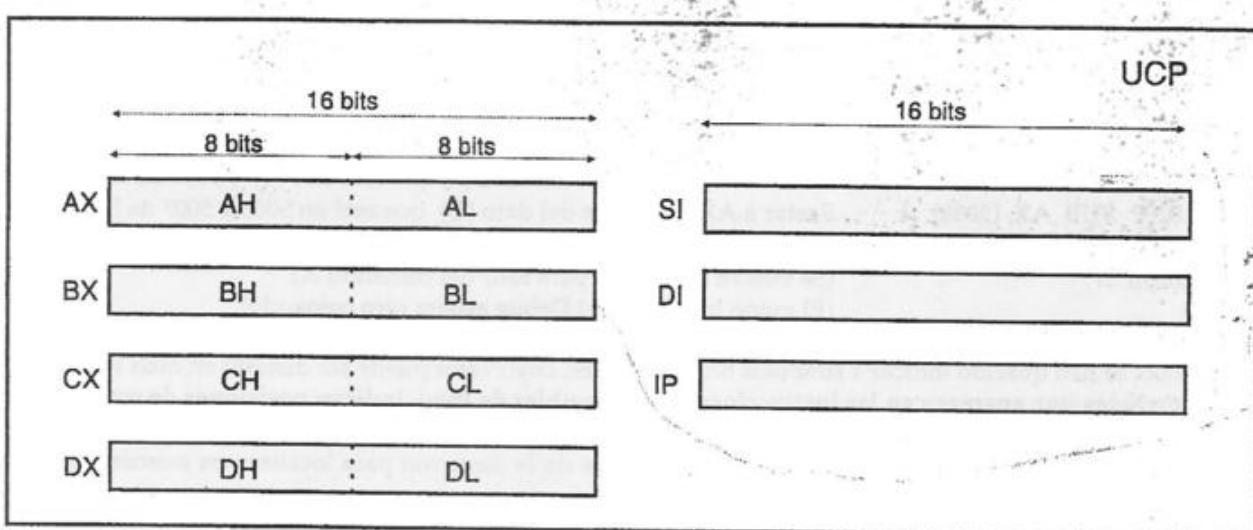


Figura 3.1

En las unidades 1 y 2 se trataron distintos ejemplos que empleaban el registro AX, de 16 bits, de la UCP como acumulador, y para operaciones de entrada y salida. Este registro también es indispensable usarlo cuando se multiplica o divide, según se verá. Pero en la UCP existen otros registros de 16 bits que pueden elegirse indistintamente como acumuladores, como BX, CX y DX (figura 3.1), que serán empleados en assembler a partir del ejercicio 3.

También se dispone en la UCP de los registros SI y DI para guardar exclusivamente direcciones que apunten a posiciones de memoria (registros punteros), como se exemplifica a partir del ejercicio 6.

BX es el único acumulador que también puede utilizarse como registro puntero (ejercicio 16).

El registro IP (instruction pointer) es afectado por las instrucciones de salto (jump).

Si se necesita operar datos de 8 bits, los registros de 16 bits: AX, BX, CX o DX (o sea los terminados en X), pueden dividirse en dos registros de 8 bits. Por ejemplo, AX se descompone en AH y AL; BX en BH y BL, etc.

Las letras H (de "high"), y L (de "low") hacen referencia a la parte alta y baja de un registro de 16 bits.

Entonces, en assembler, cuando se necesita definir un registro para datos o resultados de 16 bits, el mismo tendrá su segunda letra terminada en X; y en caso de ser necesario emplear un registro para datos o resultados de 8 bits, se usará una mitad del mismo, la cual se identificará por ser su segunda letra una H o una L, según se elija.

En los procesadores 386, 486 y Pentium, los registros citados pueden tener 32 bits, indicándose EAX, EBX, ..., EDI, ESI, EIP. La letra E hace referencia a "extended". Dichos registros también pueden elegirse para guardar 16 u 8 bits, mediante la simbología antes definida. *El Debug sólo permite definir registros de 16 u 8 bits.*

### EJERCICIO 1

Escribir en assembler las instrucciones necesarias para codificar la operación  $R = P + P - Q$

Suponer que, como en la página 31 de la Unidad I, las variables R, P y Q están en las direcciones 5010, 5000 y 5006, respectivamente, y que son magnitudes.

Razonando de igual forma que en la Unidad I, se debe también tener siempre presente en assembler el esquema del calculador de bolsillo, en el cual los datos están en su memoria, y los cálculos se efectúan usando como acumulador el visor. Según se explicó, en un computador también por un lado se tiene los datos en memoria, y por otro lado se tiene el procesador (supuesto de Intel en esta obra). Como se describió, en este último, se encuentran distintos registros que pueden ser usados a elección como acumuladores, designados AX, BX, CX y DX.

Usando el programa Debug para codificar en assembler, es necesario partir siempre de una asignación de direcciones de las variables (datos) en memoria, o sea suponer que previamente están escritas en determinadas direcciones de memoria, pues dichas direcciones (en general arbitrarias) son las que usaremos directamente en assembler para identificar variables en las instrucciones que escribiremos.

En este caso se asume que las variables se encuentran en las direcciones 5010, 5000 y 5006, según se enunció. Mediante el comando A del Debug se pueden escribir programas en assembler. Al lado de la letra A se debe colocar la dirección (en general arbitraria) a partir de la cual se quiere escribir el programa. A los fines de continuidad con la Unidad I, usaremos la misma dirección (0200) para comienzo del programa. En otros ejercicios posteriores hemos elegido arbitrariamente la dirección 0100.

Escribiremos a continuación en assembler, para el Debug, las 4 instrucciones que se corresponden con I<sub>1</sub> a I<sub>4</sub> de la página 31 citada.

A 0200 ↴

xxxx:0200 MOV AX, [5000] ↴	Llevar a AX una copia del dato (P) que está en 5000 y 5001 de la memoria.
xxxx:0203 ADD AX, [5000] ↴	Sumar a AX una copia del dato (P) que está en 5000 y 5001 de la memoria.
xxxx:0207 SUB AX, [5006] ↴	Restar a AX una copia del dato (Q) que está en 5006 y 5007 de la memoria.
xxxx:020B MOV [5010], AX ↴	Transferir a 5010 y 5011 de memoria una copia del contenido de AX
xxxx:020E ↴	(Se vuelve a pulsar ↴ para salir del comando A)
-	(El guión indica que el Debug espera otro comando)

Con xxxx se han querido indicar 4 símbolos hexadecimales, cuyo valor puede ser distinto en cada PC.

**Los corchetes que aparecen en las instrucciones del assembler de Intel, indican posiciones de memoria.**

Cuando una instrucción tiene un número entre corchetes se dice que está en "modo directo", o en modo de direccionamiento directo, siendo que directamente así se da la dirección para localizar en memoria el dato a procesar.

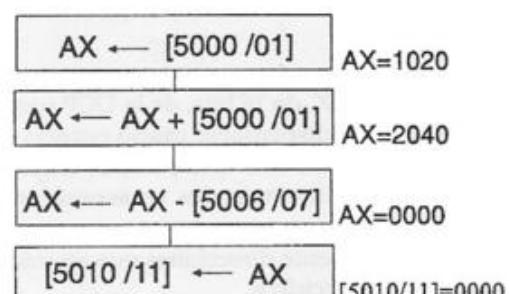


Figura 3.2

En la figura 3.2 aparece un diagrama lógico, que es conveniente realizar antes de codificar en assembler. El mismo también muestra los valores que iría tomando AX durante el proceso. Con [5000/01] se quiere indicar los valores contenidos en las posiciones de memoria 5000 y 5001 (dos bytes de información). A la derecha de cada rectángulo aparece el valor que tendría el registro o posición si se ejecutaría la instrucción. Luego de tipar una instrucción en assembler, y de pulsar "Enter" (↵), el programa traductor Ensamblador (que viene con el Debug) traduce los caracteres de la instrucción en assembler (que están en ASCII), en el correspondiente código de máquina. Así, a partir de la dirección 0200 la instrucción MOV AX, [5000]

queda codificada como A10050, según aparece en la página 31 de la Unidad I. Puesto que el Debug ahora sabe qué dicha instrucción ocupa 3 bytes, luego del Enter escribirá automáticamente xxxx:0203. Esto mismo se repite con cada instrucción.

En definitiva, después de tipar las 4 instrucciones, en memoria quedará la secuencia de códigos de máquina indicada en dicha página 31, no quedando rastro alguno de lo tipeado en assembler. De esta forma, se evita tener que escribir con el comando E todos los códigos de instrucción, como se hizo en la Unidad I, página 34. Debe notarse que en lugar del registro AX podríamos haber usado BX, CX o DX; sólo hubiesen cambiado los códigos de máquina de las instrucciones.

Si se quiere ejecutar las instrucciones de la secuencia anterior escrita en assembler, primero se debe dar valores a las variables en las direcciones, mediante el comando E, como se hizo en la página 33 de la Unidad I. Luego se debe proceder conforme a lo realizado de la página 36 a 39 de la Unidad I.

En cualquier momento que se quiere verificar si la secuencia a ejecutar o en ejecución es la correcta se usa el comando U seguido por la dirección donde comienza la secuencia:

```
U 0200 ↓
xxxx:0200 A10050      MOV AX, [5000]
xxxx:0203 03060050    ADD AX, [5000]
xxxx:0207 2B060650    SUB AX, [5006]
xxxx:020B A31050      MOV [5010], AX
```

Los puntos suspensivos indican otras instrucciones que arroja el comando U, pero que no interesan a nuestro problema. Son "basura". Este comando realiza lo que se denomina "**disassembler**" o "**unassembler**". Esto es, a partir de la dirección dada (0200 en este caso) interpreta los códigos de máquina de un programa en sus correspondientes instrucciones en assembler, efectuando el *proceso contrario* al que realiza el programa traductor Ensamblador.

Los conceptos hasta acá vertidos sirven para todos los ejercicios planteados.

## EJERCICIO 2

Escribir una secuencia para realizar la suma  $R = M + N$  de dos números enteros de 32 bits ("integers"), dado que mediante ADD AX, [xxxx] sólo pueden sumarse dos números de 16 bits

Los valores de M y N en decimal son:  $M = -2.650.000$  y  $N = 3.250.876$

A continuación se indican los valores de las variables con bit de signo, como los dejaría en memoria un programa traductor. Con fines didácticos han sido sumados y expresados en hexa.

		C=1	
M =	1111111110101111001000001110000	=	FFD790 70
N =	00000000001100011001101010111100	=	00 319ABC
R = M+N =	100000000000010010010101100101100	=	10009 2B2C
	↓	↓	
	SZVC	SZVC	
	0 0 0 1	0 0 1 1	
2000	70		
1	90		
2002	D7		
3	FF		
2008	BC	AX ← [2000 /01]	AX=9070
9	9A	AX ← AX + [2008 /09]	AX=2B2C
200A	31	SZVC	
B	00	[2010 /11] ← AX	[2010/11]=2B2C
2010		AX ← [2002 /03]	AX=FFD7
1			
2012		AX ← AX + [200A/0B] + C	AX=0009
3		SZVC	
		[2012 /13] ← AX	[2012/13]=0009

Figura 3.3

Los número M y N de 32 bits serán los datos para el programa a desarrollar. Separados en dos grupos de 16 bits se sumarán en forma fraccionada, habiéndose indicado en negrita los que se sumarán primero.

Se supone una UAL como la del 80286, que sólo puede sumar hasta dos números de 16 bits cada uno.

Esto es, el programa sumará primero los 16 bits menos significativos de M (arbitrariamente escritos en las direcciones 2000 y 2001 como se indica en la figura 3.3 con los de N (en las direcciones 2008 y 2009), y almacenará los 16 bits del resultado parcial obtenido en las direcciones 2010 y 2011 (que después de ejecutarse el programa deberán contener 2C y 2B, respectivamente, como se deduce de la suma anticipada realizada con fines didácticos)

Luego se sumarán los 16 bits superiores de M (localizados en 2002 y 2003) con los correspondientes de N (en 2001 y 200B), a los cuales se les adicionará el valor 0 ó 1 con que resulta el carry C generado de la suma de los 16 bits de la mitad inferior (en nuestro ejemplo resultó C=1, como se indica en la suma didáctica). El resultado de la suma de la mitad superior (que en hexa deberá ser 0009) se guardará en 2012 y 2013, a continuación de donde estarán los 16 bits de la suma de la mitad inferior. Estos pasos se indican en el diagrama lógico de la figura 3.3. Arbitrariamente escribiremos en assembler la secuencia a partir de la dirección 0100, existiendo correspondencia entre los 6 pasos del diagrama lógico y las 6 instrucciones siguientes:

A 100

xxxx:0100	MOV AX, [2000]	Llevar a AX una copia del dato que está en memoria en 2000 y 2001
xxxx:0103	ADD AX, [2008]	Sumar a AX una copia del dato que está en memoria en 2008 y 2009
xxxx:0107	MOV [2010], AX	Transferir a 2010 y 2011 de memoria una copia del contenido de AX
xxxx:010A	MOV AX, [2002]	Llevar a AX una copia del dato que está en memoria en 2002 y 2003
xxxx:010D	ADC AX, [200A]	Sumar a AX copia del dato que está en 200A y 200B más Carry anterior
xxxx:0111	MOV [2012], AX	Transferir a 2012 y 2013 de memoria una copia del contenido de AX

La instrucción ADC ("Add with Carry") es distinta de ADD, pues ordena sumar a AX el contenido de dos posiciones consecutivas de memoria (200A y 200B) más el valor que tenía el flag C de carry antes de ejecutar la instrucción. Puesto que una instrucción tipo MOV, como MOV [2010], AX, no cambia ningún flag, el valor de C es el proveniente de realizar ADD AX,[2008].

Podemos decir que la secuencia anterior duplica por software el formato con que opera una UAL de 16 bits (hardware) en una suma multibyte.

### EJERCICIO 3: uso de una instrucción de salto condicional (conditional jump)

Emplearemos una instrucción de salto condicional para repetir la ejecución de una secuencia.. Esta estructura será usada repetidamente en la mayoría de los ejercicios.

La usaremos para realizar el producto  $R = M \times N$ , repitiendo N veces un procedimiento que consiste en una suma. Lo hacemos por razones didácticas, dado que en el presente los microprocesadores poseen instrucciones para multiplicar y dividir mediante una sola instrucción.

Efectuaremos  $R = M \times N = 0 + M + M + \dots + M$ ; ó sea mediante N sumas con sumando M.

En un acumulador, que primero se pone en 0, iremos registrando los resultados de las sumas sucesivas.

A fin de que resulte una secuencia sencilla, supondremos: que M y N son números enteros<sup>1</sup> positivos de dos bytes, que  $N \neq 0$ , y que es el resultado  $R \leq 32767$ . Luego se verá como detectar si se supera este máximo<sup>2</sup>.

Ejemplificaremos el programa a desarrollar para la operación  $M \times N = 3 \times 4 = (11 \times 100)_B$ , a efectuarse como

$$R = M \times N = 0 + 11 + 11 + 11 + 11 \quad (N = 4 \text{ sumas con el sumando } M = 11_B)$$

Asignaremos (figura 3.4) a la variable M (con valor  $0003_H$ ) las direcciones de memoria B300 y B301; y a la variable N (con valor  $0004_H$ ) las direcciones B310 y B311. Se ha reservado para el resultado R la B320 y B321.

<sup>1</sup> Los números enteros se tratan en la Unidad 4 de esta obra.

<sup>2</sup> Otras limitaciones que tendrá la secuencia a desarrollar, es que si  $M=0$ , si bien el resultado será  $P=0$ , se repetirá el lazo N veces. Surge así la necesidad de un programa más genérico, capaz de detectar desde su inicio si M ó N valen cero, se asigne inmediatamente a la variable R el valor 0. Esto podrá concretarse luego de que definamos la instrucción comparar. En este ejemplo se ha buscado la simplicidad didáctica en lugar de la rigurosidad.

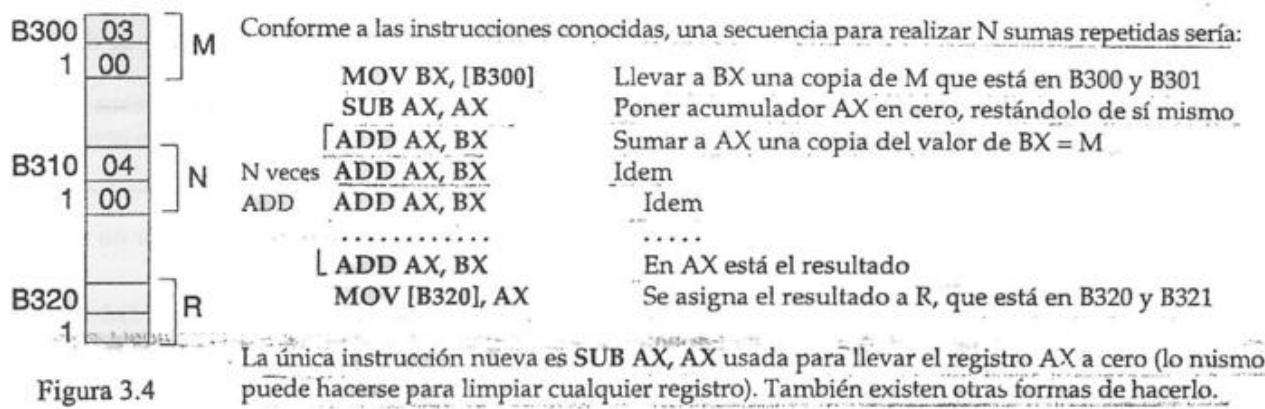


Figura 3.4

Cuando el dato a operar no está en memoria sino en un registro de la UCP, el modo de direccionamiento se denomina "modo registro"

A fin de evitar escribir N veces ADD AX, BX se construirá (ver secuencia siguiente en assembler) un lazo repetitivo controlable, que obligue a ejecutar N veces dicha instrucción escrita una sola vez, con la ayuda de dos instrucciones nuevas: DEC CX y JNZ (jump if not zero), que aparecen en la secuencia escrita. Previamente, mediante MOV CX, [B310] se cargará en CX una copia de N, para contar las N veces que se ejecutará el lazo, valor que disminuirá cada vez que se ejecute DEC CX, instrucción en modo registro, que ordena decrementar (disminuir en uno) el valor de CX.

Tipicamente se usa el registro acumulador CX para controlar las veces que se debe repetir un lazo.

En definitiva, una secuencia en assembler puede ser:

```

A 100
xxxx:0100  MOV CX, [B310]
xxxx:0104  MOV BX, [B300]
xxxx:0108  SUB AX, AX
xxxx:010A  ADD AX, BX
xxxx:010C  DEC CX
xxxx:010D  JNZ 010A
xxxx:010F  MOV [B320], AX
xxxx:0112  INT 20

```

Llevar a CX una copia de N que está en B310 y B311  
Llevar a BX una copia de M que está en B300 y B301  
Poner acumulador AX en cero, restándolo de sí mismo  
Sumar a AX el valor de M que está en BX  
Restar uno a CX (con lo cual pueden cambiar los flags SZVC)  
Si luego de la instrucción anterior Z=0 (CX no zero) saltar a 010A  
Llevar a B320 y B321 de memoria, una copia del valor de AX  
Instrucción de final'

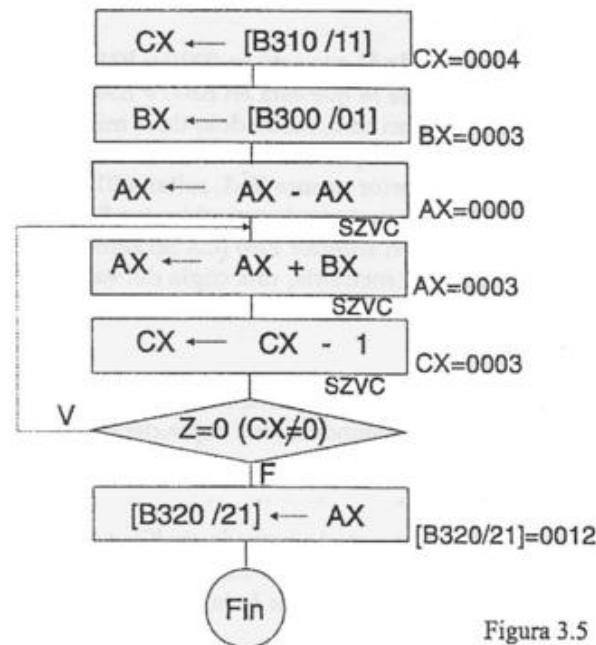


Figura 3.5

(cierto) que Z=0, y que si es falso que Z=0 ordena continuar con la instrucción que sigue a JNZ

Como se observa en el diagrama lógico de la figura 3.5, el procedimiento básico que se debe repetir N veces consiste en este caso en la instrucción ADD AX, BX. Cada vez que se realiza esta suma, la operación siguiente descontúa uno al contenido del registro CX (que inicialmente es N); y la operación subsiguiente determina si el número contenido en CX alcanzó o no el valor cero. Mientras ("while") este número en CX no sea cero (not zero = NZ), o sea mientras el resultado de restar uno a CX le corresponda la indicación Z=0 (condición de salto de JNZ)<sup>1</sup>, se salta a ejecutar nuevamente la instrucción ADD AX, BX de comienzo de la secuencia que se debe repetir. La dirección de esta instrucción a la que se quiere saltar es el número que acompaña a JNZ.

En definitiva, la instrucción de saltar si resultado anterior no es cero, que en assembler se escribe JNZ hhhh, ordena saltar a la instrucción de dirección hhhh sólo si Z=0 (cero no); caso contrario (Z=1) ordena continuar con la instrucción que sigue a JNZ.

También puede decirse que JNZ hhhh ordena saltar a la instrucción de dirección hhhh si es verdadero

<sup>1</sup> Esta instrucción no conviene ejecutarla mediante el comando T del Debug, pues puede hacer perder el programa tipeado.

<sup>2</sup> No confundir el hecho de que un resultado no sea cero (NZ), con la indicación del flag para ese caso: Z=0 que significa Zero no Tener presente que cuando un resultado si es cero, el flag es Z=1 (zero si), como se explica en la unidad 4 de esta obra.

Al lado de cada rectángulo del diagrama lógico (figura 3.5) se indica el contenido del lugar de destino para la primer ejecución de las instrucciones del lazo.

Después de realizar N veces el lazo, el contenido de CX, que era N, se le habrá restado N veces uno. La última resta que efectuará la UAL con el valor de CX será  $1-1 = 0$ , con lo cual será **Z=1**, en correspondencia con CX=0.

En consonancia se habrán realizado N sumas.

A esta altura del proceso, el siguiente salto con la condición que sea Z=0 no se realizará, por no verificar esa condición, y la secuencia continuará con MOV [B320], AX escrita a continuación.

En este ejemplo, con N=4 se ejecutará 4 veces el lazo, realizándose en cada oportunidad las siguientes operaciones que ordenarán las instrucciones dadas, con los resultados en hexadecimal que se indican.

	ADD AX, BX	DEC CX	JNZ 010A
1ra vez:	AX $\leftarrow 0 + 3$ AX=3	CX $\leftarrow 4 - 1$ (Z=0)	CX=3
2da vez:	AX $\leftarrow 3 + 3$ AX=6	CX $\leftarrow 3 - 1$ (Z=0)	CX=2
3ra vez:	AX $\leftarrow 6 + 3$ AX=9	CX $\leftarrow 2 - 1$ (Z=0)	CX=1
4ta vez:	AX $\leftarrow 9 + 3$ AX=C	CX $\leftarrow 1 - 1$ (Z=1)	CX=0
			No salta, sigue, pues Z=1

Esto puede verificarse paso a paso, ejecutando la secuencia dada, mediante el comando T del Debug.

Debe observarse en la secuencia dada, o en cualquiera con una instrucción de salto condicional, que debe escribirse debajo de la instrucción de salto aquella que hay que ejecutar en caso de que NO se cumpla la condición que ella estipula.

#### EJERCICIO 4: Mejora del ejercicio 3 usando JO (Jump if overflow), manejo de constantes, empleo de la instrucción JMP, y uso de etiquetas

En la sección N.3 de la Unidad 4 se define el overflow para números enteros. Para un formato de 16 bits, como el que tienen los registros AX, BX, CX, DX, existe overflow si por ejemplo, una suma de enteros positivos supera el valor 32767. En ese caso la UAL genera la indicación V=1 de existencia de overflow.

Modificaremos la secuencia del ejercicio anterior para que si por los valores M y N alguna suma parcial o total excede 32767, con lo cual la UAL genera indicación de overflow, se indique esto escribiendo FFFF en B320 y B321. Para tal fin, en la secuencia anterior luego de la instrucción de suma agregamos la instrucción JO como se indica a continuación.

A 100		
xxxx:0100	MOV CX, [B310]	Llevar a CX una copia de N que está en B310 y B311
xxxx:0104	MOV BX, [B300]	Llevar a BX una copia de M que está en B300 y B301
xxxx:0108	SUB AX, AX	Poner acumulador AX en cero, restándolo de sí mismo
xxxx:010A	ADD AX, BX	Sumar a AX el valor de M que está en BX
xxxx:010C	JO 0130	Si de la instrucción anterior resulta V=1, saltar a 0130
xxxx:010E	DEC CX	Restar uno a CX (con lo cual pueden cambiar los flags SZVC)
xxxx:010F	JNZ 010A	Si luego de la instrucción anterior Z=0 (CX no zero) saltar a 010A
xxxx:0111	MOV [B320], AX	Llevar a B320 y B321 de memoria, una copia del valor de AX
xxxx:0114	INT 20	Instrucción de final
A 0130		
xxxx:0130	MOV AX, FFFF	Llevar a AX una copia de la constante FFFF
xxxx:0133	MOV [B320], AX	Llevar a B320 y B321 de memoria, una copia del valor de AX
xxxx:0136	INT 20	Instrucción de final

Figura 3.6

Entonces la secuencia que empieza en 0100 es la del ejercicio anterior con el agregado de JO. La instrucción de saltar si el resultado anterior generó overflow que en assembler se escribe JO hhhh, ordena saltar a la instrucción de dirección hhhh sólo si V=1; caso contrario (V=0), ordena continuar con la instrucción que sigue a JO.

En definitiva, si luego de una suma es V=0 se ejecuta la secuencia del ejercicio anterior, y si V=1 se pasa a ejecutar la secuencia que empieza en 130 y termina en 136. Esto puede verse en el diagrama de la figura 3.7. La instrucción JO 0130 ordena saltar (si V=1) a una instrucción que no forma parte de la secuencia. Por tal motivo, mediante A 0130 hubo que escribir la otra secuencia, que empieza con MOV AX, FFFF.

Esta última ordena pasar a AX una copia de la constante FFFF, la cual será luego escrita en memoria mediante la ejecución de MOV [B321], AX. La dirección 130 fue elegida arbitrariamente.

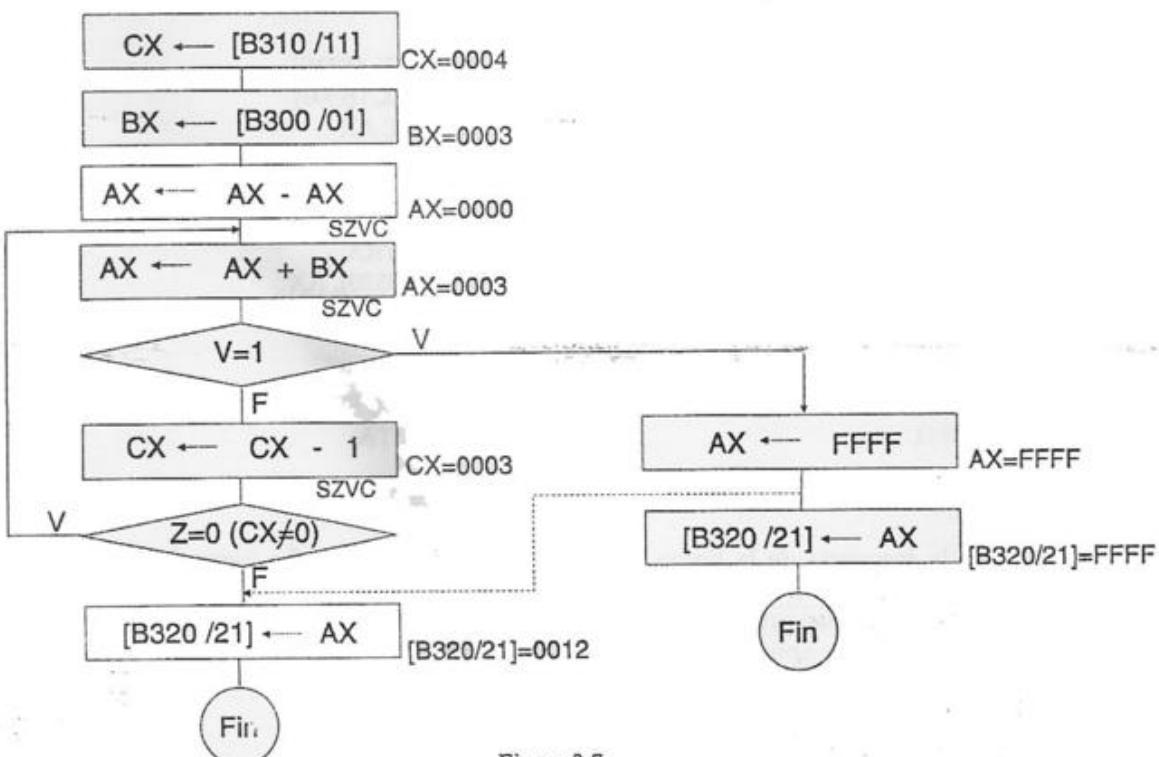


Figura 3.7

No existe una instrucción como MOV [B320], FFFF que ordene pasar directamente una constante a memoria. Ó sea que para ello se debe pasar primero a un registro (en este caso AX), y luego de éste a memoria.

Las instrucciones que ordenan pasar una constante a un registro están en modo de direccionamiento "inmediato".

Obsérvese la diferencia entre MOV AX, FFFF en modo inmediato, y MOV AX, [FFFF] en modo directo. La primera ordena cargar en AX el número FFFF, mientras que la segunda ordena cargar en AX el número que está en la dirección de memoria FFFF.

La denominación "inmediato" puede explicarse a partir del código de máquina de una instrucción en este modo de direccionamiento. Mediante el comando U del Debug hallaremos el código de MOV AX, FFFF (luego de haber escrito la secuencia en assembler)

U 0130		
xxxx:0130	B8FFFF	MOV AX, FFFF
xxxx:0133	A320B3	MOV [B320], AX
xxxx:0136	CD20	INT 20

Se observa que en memoria, luego del código de operación B8 inmediatamente le sigue la constante FFFF, de donde proviene el nombre "inmediato" de este modo. En el mismo el dato (constante) en lugar de estar en una zona de datos se encuentra formando parte de la instrucción. Es el único caso que un dato se halla en la zona de instrucciones.

#### Uso de la instrucción JMP (jump).

El programa de la figura 3.6 puede escribirse con menos instrucciones, como en la figura 3.8 dado que las secuencias que empiezan en 100 y en 130 terminan con las mismas dos instrucciones. En el diagrama lógico de la figura 3.7 se indica una línea en punteado que luego de la orden de pasar FFFF a AX sigue con la orden de pasar de AX a [B320]. De esta manera, luego de ejecutar la primer instrucción que está en 0130 se saltaría sin condición alguna a ejecutar las dos últimas instrucciones de la secuencia que empieza en 0100, que son también las dos últimas de la secuencia iniciada en 0130. La instrucción JMP 0111 de salto (jump) cumplirá este cometido. No es necesario indicar con un rectángulo la existencia de la instrucción JMP.

En general una instrucción JMP hhhh ordena saltar incondicionalmente ("si o si") a la instrucción que está en la dirección hhhh, sin posibilidad de seguir con la instrucción que sigue a JMP en memoria. Es equivalente a un "GO TO" en un lenguaje de alto nivel.

Conforme al uso de JMP, las secuencias de la figura 3.6 quedarían como indica la figura 3.8

```

A 100
xxxx:0100 MOV CX, [B310]
xxxx:0104 MOV BX, [B300]
xxxx:0108 SUB AX, AX
xxxx:010A ADD AX, BX
xxxx:0130 JO 0130
xxxx:010E DEC CX
xxxx:010F JNZ 010A
xxxx:0111 MOV [B320], AX
xxxx:0114 INT 20

```

```

A 0130
xxxx:0130 MOV AX, FFFF
xxxx:0133 JMP 0111

```

Figura 3.8

```

OTRA ADD AX, BX
JO ALFA
DEC CX
JNZ OTRA
BETA MOV [B320], AX
INT 20

```

```

ALFA MOV AX, FFFF
JMP BETA

```

Figura 3.9

La figura 3.9 repite la secuencia de la figura 3.8 pero sin las direcciones. En lugar de éstas aparecen las **etiquetas** ("labels"), como "otra", "beta", "alfa", que son admitidas por otros traductores, pero que no son permitidas por el traductor que provee el Debug. Está claro que dichas etiquetas se usan para indicar direcciones simbólicas a las que apuntan las instrucciones de salto. En la presente etapa usaremos etiquetas para escribir secuencias en assembler con papel y lápiz, pero no para el Debug. De esta forma nos independizamos del Debug y de las direcciones.

#### EJERCICIO 5: cómo cambiaría el ejercicio 3 si se usara JZ en vez de JNZ.

El lector puede preguntarse qué ocurriría si en lugar de poner la condición que  $Z=0$  se hubiese puesto que  $Z=1$ .

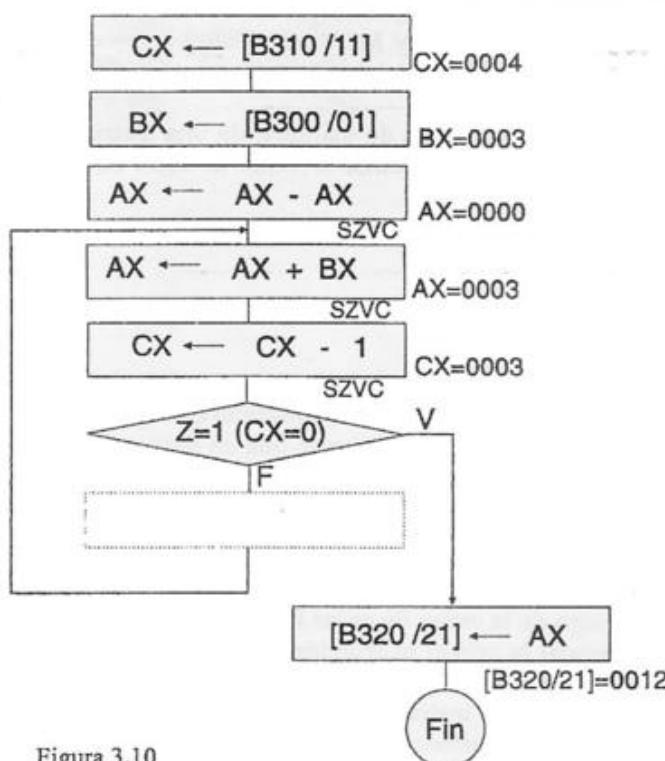


Figura 3.10

En ese caso (diagrama lógico de la figura 3.10), se saltaría cuando sea  $CX = 0$ , pues recién entonces sería  $Z=1$ , y se pasaría a ejecutar la secuencia que empieza con el envío del valor de  $AX$  a la dirección  $B320$ , a la que sigue el fin de la secuencia. Pero mientras tanto, cada vez que se le resta uno a  $CX$  no se salta, sino que se continúa (por el no, dado que es  $Z=0$ ) con la instrucción siguiente (en punteado en la figura 3.10) a la del salto condicional, que debe ser un `JMP`. Esta instrucción ordena saltar incondicionalmente a la instrucción que indica sumar  $AX$  con  $BX$ . Así se vuelve a armar un ciclo como en la figura 3.5).

Esto es, puesto que el rombo que representa una instrucción de salto condicional sólamente permite saltar por uno solo de sus vértices laterales, siendo que su vértice inferior está ligado a la continuación de una secuencia, ésta se continúa con una instrucción que ordena saltar incondicionalmente, "si o si", a otra instrucción.

Obsérvese (figura 3.10) que el salto condicional no se realiza sobre la misma secuencia, como en la figura 3.5, sino que se salta a otra secuencia, compuesta por dos instrucciones. Es por ello que en la figura 3.11 esta secuencia se escribe aparte.

Con el Debug fue necesario indicar A 130, habiéndose elegido la dirección 130 en forma arbitraria:

<sup>1</sup> Si bien, como se dijo más arriba, no es necesario representar mediante un rectángulo esta instrucción, esta vez lo haremos por razones didácticas, para mostrar que cuando no se cumple la condición en una instrucción de salto incondicional, la secuencia continúa con otra instrucción, que en este caso particular es de salto incondicional.

A 100		
xxxx:0100	MOV CX, [B310]	Llevar a CX una copia de N que está en B310 y B311
xxxx:0104	MOV BX, [B300]	Llevar a BX una copia de M que está en B300 y B301
xxxx:0108	SUB AX, AX	Poner acumulador AX en cero, restándolo de sí mismo
xxxx:010A	ADD AX, BX	Sumar a AX el valor de M que está en BX
xxxx:010C	DEC CX	Restar uno a CX (con lo cual pueden cambiar los flags SZVC)
xxxx:010D	JZ 130	Si luego de la instrucción anterior Z=1 (CX zero) saltar a 0130
xxxx:010F	JMP 10A	Saltar incondicionalmente a 10A
A130		
xxxx:0130	MOV [B320], AX	Llevar a B320 y B321 de memoria, una copia del valor de AX
xxxx:0133	INT 20	Instrucción de final

Figura 3.11

**EJERCICIO 6: manejo de un listado de datos consecutivos en memoria (lista o vector)**

Se supone (figura 3.14) que a partir de la dirección 1000 de memoria, y en posiciones consecutivas, se tiene una lista de números enteros de dos bytes cada uno:  $N_1, N_2, N_3 \dots N_n$ . La cantidad  $n$  de números de la lista (longitud) se da en la dirección 1500. Todos los números de la lista deben ser sumados, y el resultado total de la suma debe ser asignado a una variable R que está en las direcciones 2000 y 2001. La posibilidad de overflow se considera en otro ejercicio posterior.

De tener que codificarse la secuencia solicitada con los modos de direccionamiento definidos hasta el presente, y no considerando el overflow, habría que escribir (figura 3.12) tantas instrucciones para sumar como sumandos conforman la lista.

SUB AX, AX	MOV SI, 1000	Cargar en modo inmediato SI con el valor 1000
ADD AX, [1000]	SUB AX, AX	Llevar AX a cero
ADD AX, [1002]	ADD AX, [SI]	Sumar a AX el contenido de las direcciones 1000 y 1001. (AX=N <sub>1</sub> )
ADD AX, [1004]	ADD SI, 2	En modo inmediato sumar 2 al contenido de SI. (SI=1002)
ADD AX, [1006]	ADD AX, [SI]	Sumar a AX el contenido de las direcciones 1002 y 1003. (AX=N <sub>2</sub> )
.....	ADD SI, 2	En modo inmediato sumar 2 al contenido de SI (SI=1004)
.....	ADD AX, [SI]	Sumar a AX el contenido de las direcciones 1004 y 1005. (AX=N <sub>3</sub> )
MOV [2000], AX	ADD SI, 2	En modo inmediato sumar 2 al contenido de SI (SI=1006)
	ADD AX, [SI]	Sumar a AX el contenido de las direcciones 1006 y 1007 (AX=N <sub>4</sub> )
	etc...	hasta que SI contenga la dirección del último dato $N_n$ de la lista

Figura 3.12

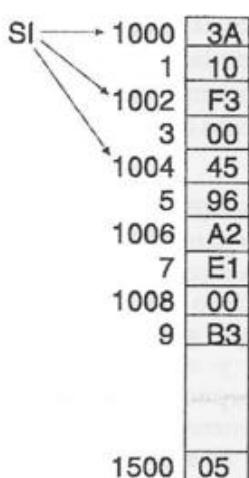


Figura 3.14

Figura 3.13

- N1 En el corchete de cada instrucción ADD AX, [hhhh] hay un número de valor fijo (hhhh), que es la dirección de una celda de memoria. Dicho número aumenta en dos de una instrucción a la siguiente. Una instrucción como ADD AX, [SI] presenta el valor del registro SI como dirección de una celda de memoria, dado que lo que está entre corchetes es siempre una dirección. Se comprende que el valor de esta dirección sería el valor que tenga el registro SI.
- N2 Para nuestro problema habrá que cargar SI con el valor inicial 1000, y luego cada vez irle sumando 2 para que SI vaya tomando sucesivamente los valores 1002, 1004, 1006, ... etc, de modo que ADD AX, [SI] cumpla la misma función de todas las instrucciones ADD AX, [hhhh] de la secuencia de la figura 3.13. La idea "en bruto", pensada como una secuencia lineal de instrucciones, sin lazo de repetición, sería la de la figura 3.13. Mediante un lazo (figuras 3.15 y 3.16) se evita repetir las instrucciones ADD AX, [SI] y ADD SI, 2.
- N3 La instrucción ADD AX, [SI] ordena cargar en AX el contenido de dos celdas consecutivas de memoria, siendo que la dirección de la primera de ellas está dada por el registro SI. Una instrucción de este tipo corresponde al modo de direccionamiento indirecto por registro. Esto es, la dirección de una celda de memoria se da indirectamente a través de un registro, como SI, DI, o BX.

*Las instrucciones en modo indirecto por registro permiten solucionar el problema de recorrer uno tras otro datos contenidos en posiciones sucesivas de memoria.*

A 100  
 xxxx:0100 MOV CL, [1500]  
 xxxx:0104 MOV SI, 1000  
 xxxx:0107 SUB AX, AX  
 xxxx:0109 ADD AX, [SI]  
 xxxx:010B ADD SI, 2  
 xxxx:010E DEC CL  
 xxxx:0110 JNZ 109

Carga en CL (mitad inferior de CX) la longitud de la lista que está en 1500  
 SI apunta al comienzo de la lista de datos  
 Pone AX en cero  
 Suma a AX un número de la lista apuntada por SI  
 Suma 2 a SI  
 Decrementa CL  
 Mientras Z sea 0, volver a 109

xxxx:0112 MOV [2000], AX Carga en 2000 y 2001 el resultado de la suma  
 xxxx:0115 INT 20 Fin

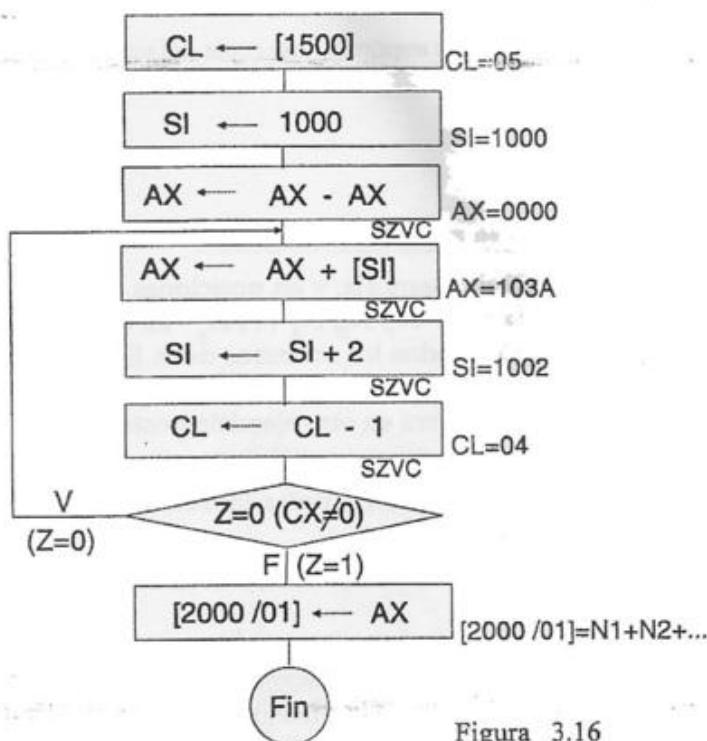


Figura 3.16

Figura 3.15

Obsérvese en el diagrama o en la secuencia, que al registro puntero SI (o cualquiera que sea) primero es necesario darle un valor inicial conforme a la dirección donde comienza la lista (mediante la instrucción en modo inmediato MOV SI, 1000). Luego en el lazo es necesario incrementar según corresponda (en este caso mediante ADD SI, 2) el valor de SI a fin de barrer toda la lista. Es importante notar que esto no se hace automáticamente, sino que debe estar a cargo del programador.

En el ejercicio 13 se amplía esta secuencia considerando la posibilidad de overflow.

#### EJERCICIO 7: uso de las instrucciones de comparación y JZ (Jump if Z=1); y escritura en una lista

A partir de la dirección 2000 se tiene una lista de caracteres codificados en ASCII (figura 3.17), siendo que su longitud está en la dirección 1500. Encontrar el número de veces que en la lista se encuentra la letra E (código 45), y dicho número guardarla en la dirección 1600. Asimismo, cada vez que se encuentra una E, indicar en una segunda lista que empieza en 6000, la dirección donde se encontraba dicha E (figura 3.18).

1500	06	
SI	→ 2000	06
	→ 2001	34 (4)
	→ 2002	41 (A)
	→ 2003	45 (E)
	→ 2004	30 (0)
	→ 2005	45 (E)
		47 (G)

DI	→ 6000	02
	→ 6002	1
	→ 6003	20
	→ 6004	04
	→ 6005	20

Figura 3.17

Figura 3.18

Como indica el diagrama lógico (figura 3.19), primero se inicializan los registros. El registro CL con la longitud de la lista; el registro SI con la dirección inicial de la lista de caracteres ASCII; el registro DI con la dirección inicial de la segunda lista, que guarda las direcciones donde se encontró una A; y el registro BL (mitad inferior de BX) se pone a cero, pues va a ser usado como contador de las veces que se encontró una A.

Luego, cada elemento de la lista es llevado en modo indirecto por registro hacia AL (mitad inferior de AX, pues AX tiene 16 bits y cada dato es de 8 bits).

A continuación se compara el elemento cargado en AX con el valor 45 (valor en hexa del código ASCII de la letra E). Para tal fin se resta en la instrucción de

comparación AX - 45, a fin de que la UAL genere valores de los flags SZVC.

Por si el valor del flag Z es 1, se pregunta en la instrucción de salto siguiente JZ. Si es Z=1 implica que AX=45, esto es, que el elemento cargado en AX es la letra E. Entonces se salta a la instrucción que incrementa a BL (contador), y se escribe en la lista que comienza en 6000 la dirección donde se encontró una E, mediante la instrucción MOV [DI], SI. Luego se debe aumentar en dos al puntero DI (pues cada dirección ocupa 2 bytes) y saltar (mediante JMP) a la secuencia principal. En ésta se incrementa el puntero SI a fin de apuntar al siguiente elemento de la lista, y se decrementa CL, para determinar con JNZ si no se terminó con la lista de datos. Si el valor de Z es 0 implica que en AX no está la letra E, por lo que deberán realizarse las acciones indicadas en la frase anterior.

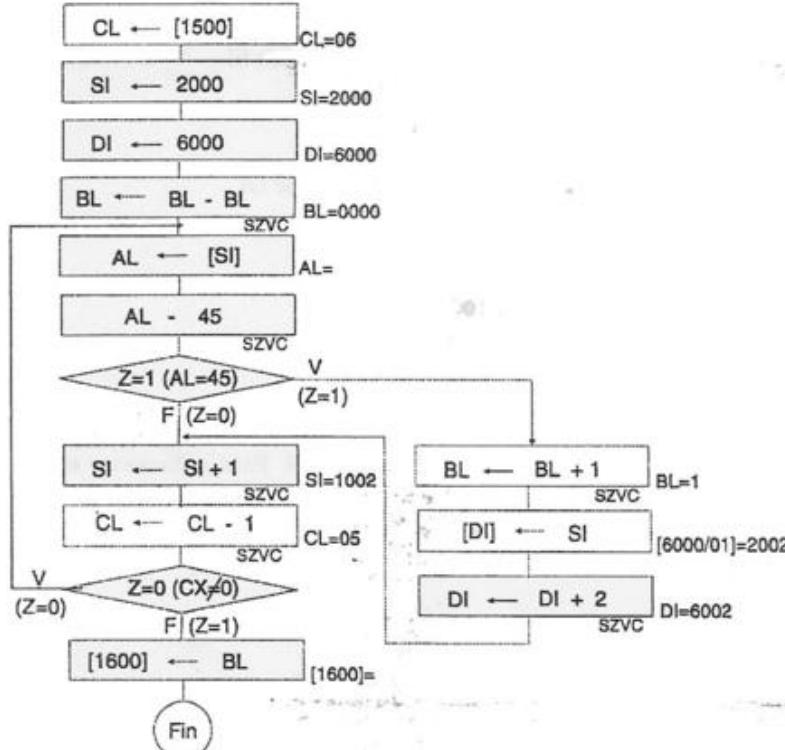


Figura 3.19

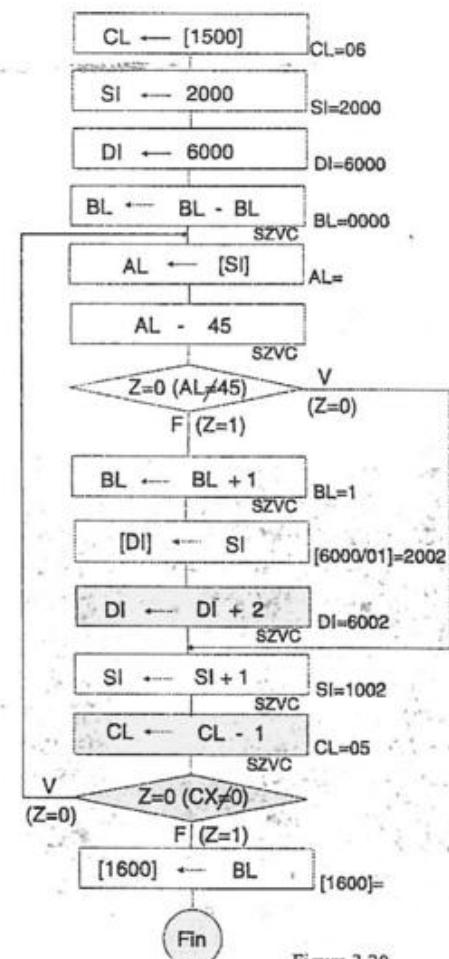


Figura 3.20

Del diagrama lógico de la figura 3.19 resultan las secuencias escritas a continuación.

A 100

```

xxxx:0100 MOV CL, [1500]
xxxx:0104 MOV SI, 2000
xxxx:0107 MOV DI, 6000
xxxx:010A SUB BL, BL
xxxx:010C MOV AL, [SI]
xxxx:010E CMP AL, 41
xxxx:0110 JZ 130
xxxx:0112 INC SI
xxxx:0113 DEC CL
xxxx:0115 JNZ 10C
xxxx:0117 MOV [1600], BL
xxxx:011B INT 20

```

Carga en CL (mitad inferior de CX) la longitud de la lista que está en 1500  
SI apunta al comienzo de la lista de datos

DI apunta al comienzo de la otra lista

Pone AL en cero

Lleva a AL un carácter de la lista apuntada por SI

Compara el valor de AL con 41

Si son iguales (Z=1) saltar a 130

Incrementa SI

Decrementa CL

Mientras Z sea 0, volver a 10C

Carga en 1600 el valor del contador

Fin

A 130

```

xxxx:0130 INC BL
xxxx:0132 MOV [DI], SI
xxxx:0134 ADD DI, 2
xxxx:0137 JMP 112

```

Incrementa BL

Escribe el valor de SI en la lista apuntada por DI

Suma 2 al puntero DI

Saltar a la instrucción que está en 112

Obsérvese que una instrucción como MOV AL, [SI] ordena leer un elemento de una lista, y cargarlo en AL. En cambio MOV [DI], SI ordena escribir un elemento de una lista cuyo valor está por SI. Las dos secuencias anteriores pueden escribirse en una sola, evitando la instrucción JMP, cambiando JZ por JNZ, conforme a las figura 3.20 y a la secuencia siguiente:

A 100	
xxxx:0100	MOV CL, [1500]
xxxx:0104	MOV SI, 2000
xxxx:0107	MOV DI, 6000
xxxx:010A	SUB BL, BL
xxxx:010C	MOV AL, [SI]
xxxx:010E	CMP AL, 41
xxxx:0110	JNZ 119
xxxx:0112	INC BL
xxxx:0114	MOV [DI], SI
xxxx:0116	ADD DI, 2
xxxx:0119	INC SI
xxxx:011A	DEC CL
xxxx:011C	JNZ 10C
xxxx:011E	MOV [1600], BL
xxxx:0122	INT 20
	Carga en CL (mitad inferior de CX) la longitud de la lista que está en 1500 SI apunta al comienzo de la lista de datos DI apunta al comienzo de la otra lista Pone AL en cero Lleva a AL un carácter de la lista apuntada por SI Compara el valor de AL con 41 Si no son iguales (Z=0) saltar a 119 Incrementa BL Escribe el valor de SI en la lista apuntada por DI Suma 2 al puntero DI Incrementa SI Decrementa CL Mientras Z sea 0, volver a 10C Carga en 1600 el valor del contador Fin

Para la instrucción JNZ 119, nos encontramos con un problema nuevo: se debe saltar hacia la instrucción INC SI, pero en el momento de escribir JNZ no conocemos la dirección 119. Para solucionar esto primero escribimos JNZ 100 (elegimos 100 por ser una dirección próxima; podría haber sido cualquier otra cercana), y cuando terminamos de escribir toda la secuencia, entonces conoceremos que 119 es la dirección de INC SI. Luego corregimos como sigue la instrucción JNZ 100 (siendo conocida su dirección 110):

A 110  
xxxx: 0110 JNZ 119

Este procedimiento debe hacerse siempre que haya que saltar "hacia adelante"

#### EJERCICIO 8: uso de la instrucción JA (Jump if above = saltar si está por arriba) para números naturales, y del modo indirecto por registro con desplazamiento

Se tiene una lista de números naturales de un byte cada uno, que empieza en la dirección 2000, siendo que su longitud está en la dirección 1500. Encontrar el mayor de los números de la lista y guardarlo en la dirección 3000.

A 100	
xxxx:0100	MOV CL, [1500]
xxxx:0104	MOV SI, 2000
xxxx:0107	MOV AL, [SI]
xxxx:0109	INC SI
xxxx:010A	DEC CL
xxxx:010C	CMP AL, [SI]
xxxx:010E	JA 112
xxxx:0110	MOV AL, [SI]
xxxx:0112	INC SI
xxxx:0113	DEC CL
xxxx:0115	JNZ 10C
xxxx:0117	MOV [3000], AL
xxxx:011A	INT 20
	Carga en CL la longitud de la lista que está en 1500 SI apunta al comienzo de la lista de datos Lleva el primer número de la lista a AL, suponiendo que es el mayor Incrementa el puntero SI para tomar el segundo elemento de la lista Decrementa CL, pues ya se tomó el primer número de la lista Compara el nro supuesto mayor en AL con otro nro apuntado por SI Saltar a 112 si el nro que está en AL está por arriba del apuntado por SI Carga en AL el nuevo número mayor apuntado por SI Incrementa SI preparándose para apuntar al próximo elemento Decrementa CL Mientras Z sea 0, volver a 10C Carga en 3000 el número mayor Fin

La instrucción JA 112 que sigue a CMP AL, [SI], ordena saltar a la instrucción que está en 112 si al hacer la resta AL - [SI] en la comparación anterior, resulta C=0, o sea si el número natural que está en AL está por arriba (es mayor) del número natural apuntado por SI; caso contrario continuar con la instrucción que sigue a JA 112. Si el número que está en AL es igual al apuntado en la lista por SI, como no se cumple la condición, la secuencia sigue por INC SI. De haberse realizado un diagrama lógico, en el rombo correspondiente a la condición de salto habría que escribir C=0. En la Unidad 4 (pag 88) se trata en detalle el uso del flag C.

Se puede escribir la secuencia anterior de la manera siguiente, donde las instrucciones INC SI y CMP AL, [SI] se han reemplazado por la instrucción CMP AL, [SI + 1]. Esto es, en una misma instrucción se incrementa SI, y se compara AL contra el número apuntado por SI + 1. También se puede escribir CMP AL, [SI] + 1.

Esta forma de direccionar el operando se conoce como "Based Addressing Mode"

A 100

```

xxxx:0100 MOV CL, [1500]
xxxx:0104 MOV SI, 2000
xxxx:0107 MOV AL, [SI]
xxxx:0109 DEC CL
xxxx:010B CMP AL, [SI + 1]
xxxx:010E JA 113
xxxx:0110 MOV AL, [SI + 1]
xxxx:0113 INC SI
xxxx:0114 DEC CL
xxxx:0116 JNZ 10B
xxxx:0118 MOV [3000], AL
xxxx:011B INT 20

```

Carga en CL la longitud de la lista que está en 1500  
 SI apunta al comienzo de la lista de datos  
 Lleva el primer número de la lista a AL, suponiendo que es el mayor  
 Decrementa CL, pues ya se tomó el primer número de la lista  
 Compara el nro supuesto mayor en AL con otro nro apuntado por SI + 1  
 Saltar a 113 si el nro que está en AL está por arriba del apuntado por SI  
 Carga en AL el nuevo número mayor apuntado por SI + 1  
 Incrementa SI preparándose para apuntar al próximo elemento  
 Decrementa CL  
 Mientras Z sea 0, volver a 10B  
 Carga en 3000 el número mayor  
 Fin

**EJERCICIO 9: uso de la instrucción JG (Jump if Greater – Saltar si es mayor) para números enteros**

Se tiene una lista de números enteros de dos bytes cada uno, que empieza en la dirección 2000, siendo que su longitud está en la dirección 1500. Encontrar el mayor de los números de la lista, y guardarlo en la dirección 3000.

Este ejercicio es similar al anterior, con la diferencia que se trata de números enteros que ocupan dos bytes. Por un lado el puntero SI habrá que incrementarlo en dos, y usar AX en vez de AL. Por otro, usar la instrucción de salto condicionado JG para números enteros en lugar de JA, propia de números naturales.

A 100

```

xxxx:0100 MOV CL, [1500]
xxxx:0104 MOV SI, 2000
xxxx:0107 MOV AX, [SI]
xxxx:0109 DEC CL
xxxx:010B CMP AX, [SI + 2]
xxxx:010E JG 113
xxxx:0110 MOV AX, [SI + 2]
xxxx:0113 ADD SI, 2
xxxx:0116 DEC CL
xxxx:0118 JNZ 10B
xxxx:011A MOV [3000], AX
xxxx:011D INT 20

```

Carga en CL la longitud de la lista que está en 1500  
 SI apunta al comienzo de la lista de datos  
 Lleva el primer número de la lista a AL, suponiendo que es el mayor  
 Decrementa CL, pues ya se tomó el primer número de la lista  
 Compara el nro supuesto mayor en AL con otro nro apuntado por SI + 2  
 Saltar a 113 si el nro que está en AL es mayor que el apuntado por SI  
 Carga en AL el nuevo número mayor apuntado por SI + 2  
 Incrementa SI preparándose para apuntar al próximo elemento  
 Decrementa CL  
 Mientras Z sea 0, volver a 10B  
 Carga en 3000 y 3001 el número mayor  
 Fin

La instrucción JG 113 que sigue a CMP AX, [SI+2], ordena saltar a la instrucción que está en 113 si al hacer la resta  $AX - [SI+2]$  en la comparación anterior, resulta  $S=V$  ( $S=0$  y  $V=0$  ó  $S=1$  y  $V=1$ , esto es, resultado positivo) y  $Z=0$ ; o sea si el número entero que está en AX es mayor ("greater") que el número entero apuntado por SI+2; caso contrario se debe continuar con la instrucción que sigue a JL 113. Vale decir, en un diagrama lógico, en el rombo correspondiente a JG habría que escribir  $S=V$  y  $Z=0$  en función de los flags (unidad 4, pág. 88).

*Es importante saber usar para cada clase de números (enteros o naturales) la instrucción de salto que corresponda: cuando se ordena saltar si un primer número es mayor que otro segundo, se debe usar JA para naturales, y JG para enteros. Se supone que la instrucción anterior a la de salto es una comparación (resta del primero menos el segundo).*

**EJERCICIO 10: uso de las instrucciones JB (jump if below) para naturales, y JL (jump if less) para enteros**

Modificar las secuencias de los ejercicios 8 y 9 para encontrar el menor de una lista de números.

En el ejercicio 8 se debe cambiar la instrucción JA 113 por JB 113, y en el ejercicio 9 reemplazar JG 113 por JL 113

La instrucción JB 113 que seguiría a CMP AL, [SI], ordena saltar a la instrucción que está en 113 si al hacer la resta  $AL - [SI]$  en la comparación anterior, resulta  $C=1$ , o sea si el número natural que está en AL está por debajo (es menor) del número natural apuntado por SI; caso contrario continuar con la instrucción que sigue a JB 113.

La instrucción JL 113 que seguiría a CMP AX, [SI+2], ordena saltar a la instrucción que está en 113 si al hacer la resta  $AX - [SI+2]$  en la comparación anterior, resulta  $S \neq V$  ( $S=1$  y  $V=0$  ó  $S=0$  y  $V=1$ , esto es, resultado negativo) y  $Z=0$ , o sea si el número entero que está en AX es menor ("less") que el número entero apuntado por SI+2; caso contrario se debe continuar con la instrucción que sigue a JG 113. De lo anterior, resulta que en un diagrama lógico habría que escribir la condición  $S \neq V$  y  $Z=0$  en el rombo que corresponde a JL.

#### EJERCICIO 11: uso de JBE (Jump if below or equal)

Se tiene una lista de combinaciones binarias de un byte que corresponde cada una a dígitos hexadecimales aislados (0, 1, 2, ..., F), la cual empieza en la dirección 2001, siendo que su longitud está en la dirección 2000. Convertir cada dígito hexadecimal a su correspondiente carácter ASCII según se exemplifica a continuación. Los caracteres ASCII obtenidos guardarlos en otra lista que comienza en 3001

	Binario	Hexa		ASCII	
2001	00000000	(00)		00110000	(30)
2002	00000100	(04)		00110100	(34)
2003	00001001	(09)	=>	00111001	(39)
2004	00001010	(0A)		01000001	(41)
2005	00001101	(0D)		01000100	(44)
2006	00001111	(0F)		01000110	(46)

Si dato < 9 es en ASCII igual a dicho dato + 30

Si dato > 9 es en ASCII igual a dicho dato + 7 + 30

Por ejemplo:

Dato = 00001010 = A

$$+7 = + \underline{\hspace{2cm}} 111$$

00010001

$$+30 = \underline{\hspace{2cm}} 00110000$$

01000001 = 41 en ASCII

A 100

```
xxxx:0100 MOV CL, [2000]
xxxx:0104 MOV SI, 2001
xxxx:0107 MOV DI, 3001
xxxx:010A MOV AL, [SI]
xxxx:010C CMP AL, 9
xxxx:010E JBE 112
xxxx:0110 ADD AL, 7
xxxx:0112 ADD AL, 30
xxxx:0114 MOV [DI], AL
xxxx:0116 INC DI
xxxx:0117 INC SI
xxxx:0118 DEC CL
xxxx:011A JNZ 10A
xxxx:011C INT 20
```

Carga en CL la longitud de la lista

SI apunta al comienzo de la lista de datos

DI apunta al comienzo de la lista de resultados

Carga en AL un dato de la lista apuntada por SI

Compara el dato en AL contra 9

Si es menor o igual que 9 salta a 112

Suma 7 a AL

Suma 30 a AL

Guarda un resultado en la lista apuntada por DI

Incrementa DI

Incrementa SI

Decrementa CL

Mientras Z sea 0, volver a 10A

Fin

La instrucción JBE 112 que sigue a CMP AL, 9, ordena saltar a la instrucción que está en 112 si al hacer la resta  $AL - 9$  en la comparación anterior, resulta  $C=1$  ó  $Z=1$ , o sea si el número natural que está en AL está por debajo (es menor) o es igual al número 9; caso contrario continuar con la instrucción que sigue a JBE 112. O sea que en un diagrama lógico, en el rombo correspondiente a JBE habría que escribir  $C=1$  ó  $Z=1$ .

#### EJERCICIO 12:

Este ejercicio es contrario al anterior. Esto es, se tiene una lista de caracteres ASCII que empieza en 2001, y cuya longitud está en 2000. Se la quiere convertir en otra lista que empiece en 3001 con los números hexadecimales (en binario) correspondientes a dichos caracteres en ASCII.

A 100	
xxxx:0100 MOV CL, [2000]	Carga en CL la longitud de la lista
xxxx:0104 MOV SI, 20001	SI apunta al comienzo de la lista de datos
xxxx:0107 MOV DI, 3001	DI apunta al comienzo de la lista de resultados
xxxx:010A MOV AL, [SI]	Carga en AL un dato de la lista apuntada por SI
xxxx:010C SUB AL, 30	Resta 30 a AL
xxxx:010E CMP AL, 9	Compara el contenido de AL con 9
xxxx:0110 JBE 114	Si es menor o igual salta a 114 (Si AL < 9 es un símbolo del 0 al 9)
xxxx:0112 SUB AL, 7	Resta 7 a AL (Si AL > 9 es un símbolo de A a F, que se halla restando 37, siendo que antes ya se restaron 30)
xxxx:0114 MOV [DI], AL	Guarda el resultado en la lista apuntada por DI
xxxx:0116 INC DI	Incrementa DI
xxxx:0117 INC SI	Incrementa SI
xxxx:0118 DEC CL	Decrementa CL
xxxx:011A JNZ 10A	Mientras Z sea 0, volver a 10A
xxxx:011C INT 20	Fin

#### EJERCICIO 13: uso de la instrucción JPE (Jump if parity is even = Saltar si la paridad es par)

Se tiene una lista de caracteres ASCII a partir de la dirección 2001, siendo que su longitud está en la dirección 2000. Si la paridad de unos de cada uno de los caracteres es par, escribir 00 en la dirección 3000; y si algún carácter presenta paridad impar de unos, escribir FF<sub>H</sub> en la dirección 3000

A100	
xxxx:0100 MOV CL, [2000]	Carga en CL la longitud de la lista
xxxx:0103 MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
xxxx:0107 MOV DL, 00	Carga 00 en DL suponiendo que todo está bien
xxxx:0109 SUB AL, AL	Hace cero el registro AL
xxxx:010B ADD AL, [SI]	Suma contra AL para que pueda cambiar el indicador de paridad P
xxxx:010D JPE 111	Si paridad n AL es par, saltar a 111
xxxx:010F MOV DL, FF	Indicación de un elemento con paridad errada
xxxx:0111 INC SI	Incrementa SI
xxxx:0112 DEC CL	Decrementa CL
xxxx:0114 JNZ 109	Mientras Z=0, volver a 109
xxxx:0116 MOV [3000], DL	Guarda indicación de paridad
xxxx:011A INT 20	Fin

La instrucción JPE 111 que sigue a ADD AL, [SI], ordena saltar a la instrucción que está en 110 si al hacer la suma AL + [SI] en la suma anterior, resulta PE (paridad even, o sea paridad par); caso contrario (paridad impar) continuar con la instrucción que sigue a JPE 111.

La instrucción ADD AL, [SI] debe insertarse a los fines de sumar AL = 0 más el número apuntado por SI, para que de acuerdo con este número pueda detectarse la paridad del mismo.

#### Sistematización: INSTRUCCIONES DE SALTO CONDICIONAL

Lo que hace la primer instrucción de salto condicional JNZ ejemplificada, sirve de modelo para comprender cualquier otra instrucción de salto.

En todas ellas si se cumple una determinada condición establecida en función de uno o varios flags combinados, se salta a ejecutar otra instrucción, en lugar de seguir con la que está escrita a continuación. Se ejecutará esta última si no se cumple la condición establecida.

Una instrucción de salto condicional sólo difiere de otra en la condición establecida para saltar.

Si bien cuando se ejecuta una instrucción de salto condicional la UC determina el valor de uno o varios flags combinados (realizando en este último caso operaciones lógicas entre los mismos), en assembler la condición puede establecerse en función de si un número es mayor, igual, o menor que otro, debiendo existir siempre una instrucción de comparación (resta) anterior a la de salto.

Esto hace que ciertas instrucciones de salto puedan escribirse de dos o más maneras.

Por ejemplo, la instrucción JZ (jump if Z=1) usada en el ejercicio 6 es equivalente a JE (Jump if equal), dado que la instrucción de comparación anterior resta dos números. Si el resultado de esta resta es cero (Z=1) implica que los números son iguales. Usar JE en vez de JZ permite que el programador se desentienda de los flags. Del mismo modo, puede usarse JNE (Jump if not equal) en vez de JNZ (Jump if Z=0).

Las instrucciones JZ/JE y JNZ/JNE se usan tanto para números naturales como para enteros, puesto que el número cero tiene igual representación simbólica en ambas clases de números.

Según se ha visto en ejercicios anteriores, las instrucciones de salto condicional para números enteros, tales como JL<sup>1</sup> y JG<sup>2</sup>, no utilizan en sus letras alusión a los flags interviniéntes, dado que éstos están relacionados mediante operaciones lógicas que el programador no necesita necesariamente conocer ni recordar. Esto es, en principio si al restar A - B el resultado es negativo (S=1) y no cero (Z=0) implicaría que A < B. Pero en un computador (unidad 4, página 88) ello no es suficiente, dado que además no debe haber overflow (V=0), pues significaría que el signo es incorrecto, o sea por un lado para que A < B, debe ser: S=1, V=0 y Z=0. Por otra parte si el resultado es positivo (S=0), pero si hay overflow (V=1) implica que en realidad es S=1, por lo que también será A < B si S=0, V=1 y Z=0. Las dos situaciones se pueden resumir así: A < B si S≠V y Z=0. También para enteros se pueden fijar las condiciones de salto JLE (Jump if less or equal)<sup>3</sup>, y JGE (Jump if greater or equal)<sup>4</sup>. La condición JLE puede resumirse así en función de los flags: A ≤ B si S≠V ó Z=1.

Resumiendo, en principio para los números es importante tener presente las siguientes instrucciones de salto condicionado:

Para enteros: JL, JLE, JG, JGE, JZ, JNZ, JO, JNO

Para naturales: JB<sup>5</sup>, JBE, JA, JAE<sup>6</sup>, JZ, JNZ

Como ya se observó, los flags no son indispensables de manejar para escribir en assembler.

#### EJERCICIO 14

Se tiene una lista de números enteros (2 bytes cada uno) que empieza en 2001, siendo que en 2000 está su longitud. Sumar algebraicamente dichos números, y el resultado guardarlo en 1500 y 1501. Si al sumar algún número se produce overflow, guardar en 1500 y 1501 la suma parcial anterior correcta, y guardar en 1600 y 1601 la dirección de dicho número que produjo overflow. Asimismo, si hay overflow, escribir FF en la dirección 1502; y si no lo hay, escribir 00 en dicha dirección.

##### A 100

```
xxxx:0100 MOV CL, [2000]
xxxx:0104 MOV SI, 2001
xxxx:0107 SUB AX, AX
xxxx:0109 MOV DX, AX
xxxx:010B ADD AX, [SI]
xxxx:010D JO 130
xxxx:010F ADD SI, 2
xxxx:0112 DEC CL
xxxx:0114 JNZ 109
xxxx:0116 MOV [1500], AX
xxxx:0119 MOV AL, 00
xxxx:011C MOV [1502], AL
xxxx:011F INT 20
```

Carga en CL la longitud de la lista  
 SI apunta al comienzo de la lista de datos  
 Pone AX en cero  
 Guarda una copia de AX en DX  
 Suma contra AX un número de la lista apuntado por SI  
 Si overflow (V=1) saltar a 130  
 Incrementa SI en 2, pues cada dato ocupa 2 bytes.  
 Decrementa CL  
 Mientras Z sea 0, volver a 109  
 Guarda en 1500 y 1501 la suma total  
 Carga 00 en AL  
 Guarda 00 de AL en 1502  
 Fin

##### A 130

```
xxxx:0130 MOV [1500], DX
xxxx:0134 MOV AL, FF
xxxx:0137 MOV [1502], AL
xxxx:013A MOV [1600], SI
xxxx:013E INT 20
```

Guarda la suma parcial en 1501 y 1502  
 Carga FF en AL  
 Guarda FF de AL en 1502  
 Guarda en 1600 la dirección apuntada por SI  
 Fin

En este ejercicio, o en cualquiera donde se comienza cargando la longitud de una lista, se podría, mediante una instrucción de comparación, determinar si dicha longitud es cero. En este caso, una instrucción de salto (siguiente) ordenaría ir al final de la secuencia.

<sup>1</sup> JL (Jump if Less) también es equivalente a Jump if Not Greater than or Equal (JNGE)

<sup>2</sup> JG (Jump if Greater) también es equivalente a Jump if Not Less than or Equal (JNLE)

<sup>3</sup> JLE también es equivalente a Jump if Not Greater (JNG)

<sup>4</sup> JGE también es equivalente a Jump if Not Less (JNL)

<sup>5</sup> JB es equivalente a Jump if Not Above or Equal (JNAE) y también a Jump if Carry (JC)

<sup>6</sup> JAE (Jump if above or Equal) es equivalente a Jump if Not Below (JNB) y también a Jump if Not Carry (JNC)

**EJERCICIO 15:**

Se tiene una lista de números naturales de 2 bytes cada uno, que empieza en 2001, siendo que su longitud está la dirección 2000. Sumarlos, y el resultado (de hasta 32 bits) indicarlo en las direcciones 1500 a 1503.

A100

```
xxxx:0100 MOV CL, [2000]
xxxx:0104 MOV SI, 20001
xxxx:0107 SUB AX, AX
xxxx:0109 SUB DX, DX
xxxx:010B ADD AX, [SI]
xxxx:010D ADC DX, 00
```

Carga en CL la longitud de la lista  
 Registro SI apunta al comienzo de la lista de datos  
 Pone AX en cero  
 Pone DX en cero  
 Suma contra AX un dato de la lista  
 Lleva en DX la suma de acarreos que ocurren cuando las sumas superan el formato de AX (Hace  $DX \leftarrow DX + 00 + \text{Carry} = DX + \text{Carry}$ )  
 Incrementa SI en 2, pues cada dato ocupa 2 bytes.  
 Decrementa CL  
 Mientras Z sea 0, volver a 10B  
 Guarda en 1500 y 1501 los 16 bits inferiores de la suma  
 Guarda en 1502 y 1503 los 16 bits superiores de la suma  
 Fin.

La instrucción ADC DX, 00 es necesaria, dado que la suma de dos números naturales de 16 bits puede dar como resultado un número de 17 bits si hay carry. En ese caso, cada vez que ello ocurra hay que ir sumando el bit de carry al registro DX, elegido para ir acumulando todos los carry que se produzcan. De esta forma, la suma parcial y la total se componen de 32 bits, de los cuales los 16 menos significativos están en AX, siendo que los 16 más significativos se guardan en DX.

**EJERCICIO 16**

Se tiene una lista de números enteros que comienza en la dirección 2000, y cuya longitud está en la dirección 1500. Contar cuántos negativos y positivos hay (los ceros no se cuentan) y separarlos en dos listas que empiecen en 3000 y 4000, respectivamente. La cuenta de negativos y positivos guardarla en las direcciones 1600 y 1700, respectivamente.

A 100

```
xxxx:0100 MOV CL, [1500]
xxxx:0104 MOV SI, 2000
xxxx:0107 MOV DL, 3000
xxxx:010A MOV BX, 4000
xxxx:010D SUB DX, DX
xxxx:010F MOV AX, [SI]
xxxx:0111 CMP AX, 0
xxxx:0114 JL 130
xxxx:0116 JG 140
xxxx:0118 ADD SI, 2
xxxx:011B DEC CL
xxxx:011D JNZ 10F
xxxx:011F MOV [1600], DL
xxxx:0123 MOV [1700], DH
xxxx:0127 INT 20
```

Carga en CL la longitud de la lista  
 El registro SI apunta al comienzo de la lista de enteros  
 DI apunta al comienzo de la lista de negativos  
 BX apunta al comienzo de la lista de positivos  
 DX se lleva a cero  
 Carga en AX un dato de la lista apuntada por SI  
 Compara el dato en AX contra 0  
 Si es negativo salta a 130  
 Si es positivo salta a 140  
 Suma 2 a SI  
 Decrementa CL  
 Mientras Z=0, volver a 10F  
 Guarda la cantidad de negativos en 1600  
 Guarda la cantidad de positivos en 1700  
 Fin

A 130

```
xxxx:011B INC DL
xxxx:011F MOV [DI], AX
xxxx:0118 ADD DI, 2
xxxx:0118 JMP 118
```

Incrementa DL  
 Guarda número negativo en lista apuntada por DI  
 Suma 2 a DI  
 Salta a 118

A 140

```
xxxx:011B INC DH
xxxx:011F MOV [BX], AX
xxxx:0118 ADD BX, 2
xxxx:0118 JMP 118
```

Incrementa DH  
 Guarda el número positivo en lista apuntada por BX  
 Suma 2 a BX  
 Salta a 118

De haberse realizado un diagrama lógico, existirían dos rombos consecutivos, uno para JL y otro para JG.

**EJERCICIO 17**

Existen dos listas de igual longitud de números enteros (2 bytes cada uno), que empiezan en las direcciones 2000 y 3000 respectivamente. La longitud está en la dirección 1500. Comparar los números correspondientes, y dejar todos los elementos menores en una lista, y los mayores en otra.

A100

```

xxxx:0100 MOV CL, [1500]
xxxx:0104 MOV SI, 2000
xxxx:0107 MOV DI, 3000
xxxx:010A MOV AX, [SI]
xxxx:010C MOV BX, [DI]
xxxx:010E CMP AX, BX
xxxx:0110 JL 116
xxxx:0112 MOV [DI], AX
xxxx:0114 MOV [SI], BX
xxxx:0116 ADD DI,2
xxxx:0119 ADD SI,2
xxxx:011C DEC CL
xxxx:011E JNZ 10A
xxxx:0120 INT 20

```

Carga en CL la longitud de las listas  
 Registro SI apunta al comienzo de una lista  
 Registro DI apunta al comienzo de la otra lista  
 Carga en AX un número de la lista apuntada por SI  
 Carga en BX un número de la lista apuntada por DI  
 Compara los números correspondientes de ambas listas  
 Si AX < BX saltar a 116  
 Lleva a la lista apuntada por DI el número mayor  
 Lleva a la lista apuntada por SI el número menor  
 Incrementa DI en 2  
 Incrementa SI en 2  
 Decrementa CL  
 Mientras Z=0, volver a 10A  
 Fin

**EJERCICIO 18**

Se tiene un número decimal codificado en ASCII, con parte entera y parte fraccionaria separadas por un punto (codificación 2E en ASCII). El número, como se indica, ocupa n posiciones consecutivas de memoria, a partir de la dirección 2001, siendo que en 2000 se indica cuántas posiciones ocupa. En caso que se trate de un número entero, sin parte fraccionaria, no lleva punto. Reemplazar los dígitos fraccionarios que están a la derecha del punto, por el código ASCII del espacio (SP = 20).

Ejemplo:

2000	05
2001	34
2002	33
2003	2E
2004	35
2005	30

(4) (3) (. ) ==> (5) (0)

Antes

2000	05
2001	34
2002	33
2003	2E
2004	20
2005	20

(SP) (SP)

Después

2000	03
2001	39
2002	36
2003	38

Antes

2000	03
2001	39
2002	36
2003	38

Después

A100

```

xxxx:0100 MOV CL, [2000]
xxxx:0104 MOV SI, 2001
xxxx:0107 MOV AL, 2E
xxxx:0109 MOV BL, 20
xxxx:010B CMP AL, [SI]
xxxx:010D JNZ 114
xxxx:010F DEC CL
xxxx:0111 INC SI
xxxx:0112 MOV [SI], BL
xxxx:0114 INC SI
xxxx:0115 DEC CL
xxxx:0117 JNZ 10B
xxxx:0119 INT 20

```

Carga en CL la longitud de la lista  
 Registro SI apunta al comienzo de la lista de datos  
 Carga 2E en AL  
 Carga 20 en BL  
 Compara AL con el número apuntado por SI  
 Si no son iguales (no hay un punto) saltar a 114  
 Decrementa CL  
 Incrementa SI  
 El contenido de BL (20) se guarda donde apunta SI  
 Incrementa SI  
 Decrementa CL  
 Mientras Z=0, volver a 10B  
 Fin

**EJERCICIO 19**

La siguiente secuencia puede ser parte de un programa para verificar el estado de la memoria de un computador. Se trata de escribir ocho "unos" en cada una de las posiciones de memoria que van de la dirección 4000 a la 5000. Luego se debe verificar que realmente existan ocho "unos" en cada una de las posiciones escritas. La dirección de las posiciones que no presenten ocho "unos" (posiciones defectuosas) deben guardarse en una lista que comienza en la dirección 2000.

A 100

```
xxxx:0100 MOV SI, 4000
xxxx:0103 MOV DI, 2000
xxxx:0106 MOV AL, FF
xxxx:0108 MOV [SI], AL
xxxx:011A INC SI
xxxx:011B CMP SI, 5001
xxxx:011F JNZ 108
xxxx:0111 MOV SI, 4000
xxxx:0114 MOV AL, [SI]
xxxx:0116 CMP AL, FF
xxxx:0118 JZ 11A
xxxx:011A MOV [DI], SI

xxxx:011C ADD DI, 2
xxxx:011F INC SI
xxxx:0120 CMP SI, 5001
xxxx:0124 JNZ 114
xxxx:0126 INT 20
```

El registro SI apunta a la dirección 4000  
 DI apunta al comienzo de la lista de direcciones  
 AL se carga con 8 "unos"  
 El contenido de AL pasa a la posición apuntada por SI  
 Incrementa SI  
 Compara SI con 5000 + 1  
 Mientras SI no sea 5001 volver a 108  
 El registro SI apunta nuevamente a la dirección 4000  
 El contenido de la posición apuntada por SI pasa a AL  
 Compara el contenido de AL con FF  
 Si Z=1, ir a 11A  
 Guarda en la lista apuntada por DI la dirección (de 16 bits de SI) de una posición que no contenga 8 "unos"  
 Suma 2 a DI (las direcciones ocupan 2 bytes)  
 Incrementa SI  
 Compara SI con 5000 + 1  
 Mientras SI no sea 5001 volver a 114  
 Fin

Carga 8 unos en la lista de 16 bits de SI

**EJERCICIO 20**

Se tiene una lista de números enteros de un byte cada uno, que empieza en la dirección 2000, siendo que su longitud está en la dirección 1500. Pasar los números a otra lista, donde cada número ocupe 16 bits, de modo que a los números con bit de signo 0 se le agreguen 8 ceros a la izquierda, y a los números con bit de signo 1 se le agreguen 8 unos a la izquierda (propagación de signo).

A 100

```
xxxx:0100 MOV CL, [1500]
xxxx:0104 MOV SI, 2000
xxxx:0107 MOV DI, 3000
xxxx:010A MOV AL, [SI]
xxxx:010C CMP AL, 0
xxxx:010E JGE 130
xxxx:0110 JL 140
xxxx:0112 INC SI
xxxx:0113 ADD DI, 2
xxxx:0116 DEC CL
xxxx:0118 JNZ 10A
xxxx:011A INT 20
```

Carga en CL la longitud de la lista  
 El registro SI apunta al comienzo de la lista de datos  
 DI apunta al comienzo de la lista de resultados  
 Carga en AL un dato de la lista apuntada por SI  
 Compara el dato en AL contra 0  
 Si es mayor o igual que 0 (positivo) salta a 130  
 Si es menor que 0 (negativo) salta a 140  
 Incrementa SI  
 Suma 2 a DI (los números resultantes ocupan 2 bytes)  
 Decrementa CL  
 Mientras Z=0, volver a 10A  
 Fin

A 130

```
xxxx:0104 MOV AH, 00
xxxx:010A MOV [DI], AX
xxxx:010A JMP 112
```

Carga 8 ceros en AH para propagar signo  
 Carga en la lista apuntada por DI el número de 16 bits  
 Salta a 112

A 140

```
xxxx:0104 MOV AH, FF
xxxx:010A MOV [DI], AX
xxxx:010A JMP 112
```

Carga 8 unos en AH para propagar signo  
 Carga en la lista apuntada por DI el número de 16 bits  
 Salta a 112

**EJERCICIO 21**

Se tiene dos listas L1 y L2 de caracteres ASCII de igual longitud, que empiezan en las direcciones 2000 y 3000 respectivamente. La longitud de ambas está en la dirección 1500. Determinar el orden alfabético de una respecto de la otra, de forma tal que si L1 sea primera en orden alfabético, o tiene igual orden alfabético que L2, poner 00 en la dirección 1600. Si L2 va primero, colocar FF en 1600.

2000	<b>41</b>	(A)	3000	<b>41</b>	(A)
2001	<b>42</b>	(B)	3001	<b>44</b>	(D)
2002	<b>41</b>	(A)	3002	<b>41</b>	(A)

A 100

xxxx:0100 MOV CL, [1500]  
 xxxx:0104 MOV SI, 2000  
 xxxx:0107 MOV DI, 3000  
 xxxx:010A MOV AL, [SI]  
 xxxx:010C CMP AL, [DI]  
 xxxx:010E JB 120  
 xxxx:0110 JA 130  
 xxxx:0112 INC SI  
 xxxx:0113 INC DI  
 xxxx:0114 DEC CL  
 xxxx:0116 JNZ 10A  
 xxxx:0118 MOV BL, 00  
 xxxx:011A MOV [1600], BL  
 xxxx:011E INT 20

Carga en CL la longitud de las listas  
 Registro SI apunta al comienzo de una lista  
 DI apunta al comienzo de la otra lista  
 Carga en AL un dato de una lista  
 Compara el dato en AL con el correspondiente dato de la otra lista  
 Si es menor salta a 120  
 Si es mayor salta a 130  
 Incrementa SI  
 Incrementa DI  
 Decrementa CL  
 Mientras Z=0, volver a 10A  
 Pone BL en cero  
 Guarda resultado en BL  
 Fin

A120

xxxx:0120 MOV BL, 00  
 xxxx:0122 MOV [1600], BL  
 xxxx:0126 INT 20

Pone BL en cero  
 Guarda resultado en BL  
 Fin

A130

xxxx:0130 MOV BL, FF  
 xxxx:0132 MOV [1600], BL  
 xxxx:0136 INT 20

Pone BL en FF  
 Guarda resultado en BL  
 Fin

**EJERCICIO 22: uso del modo indirecto por registro con desplazamiento mediante valor de registro**

De las direcciones de memoria 2000 a 2009 se tiene una lista con los cuadrados de los números binarios del 0 al 9. En las direcciones 3000 y 3001 se tiene dos números N1 y N2, naturales de un byte, comprendidos entre 0 y 9 inclusive. Se necesita sumar los cuadrados de N1 y N2, y el resultado dejarlo en 3002 y 3003

2000	<b>00000000</b>	(0 = 0 <sup>2</sup> )	3000	<b>N1</b>
2001	<b>00000001</b>	(1 = 1 <sup>2</sup> )	3001	<b>N2</b>
2002	<b>00000100</b>	(4 = 2 <sup>2</sup> )	3002	
2003	<b>00001001</b>	(9 = 3 <sup>2</sup> )	3003	
2004	<b>00010000</b>	(16 = 4 <sup>2</sup> )		
2005	<b>00011001</b>	(25 = 5 <sup>2</sup> )		
2006	<b>00100100</b>	(36 = 6 <sup>2</sup> )		
2007	<b>00110001</b>	(49 = 7 <sup>2</sup> )		
2008	<b>01000000</b>	(64 = 8 <sup>2</sup> )		
2009	<b>01010001</b>	(81 = 9 <sup>2</sup> )		

A 100

xxxx:0100 MOV SI, 2000  
 xxxx:0103 SUB AX, AX  
 xxxx:0105 SUB BX, BX  
 xxxx:0107 SUB DX, DX  
 xxxx:0109 MOV DL, [3000]  
 xxxx:010D ADD SI, DX  
 xxxx:010F MOV AL, [SI]  
 xxxx:0111 SUB SI, DX  
 xxxx:0113 MOV DL, [3001]  
 xxxx:0117 ADD SI, DX  
 xxxx:0119 MOV BL, [SI]  
 xxxx:011B ADD AX, BX  
 xxxx:011D MOV [3002], AX  
 xxxx:0120 INT 20

SI apunta al comienzo de la lista de datos  
 Pone AX en cero  
 Pone BX en cero  
 Pone DX en cero  
 Carga N1 en DL  
 Registro SI apunta a 2000 + N1  
 $(N1)^2$  se lleva a la mitad inferior de AX  
 El puntero SI vuelve al valor 2000  
 Carga N2 en DL  
 Registro SI apunta a 2000 + N2  
 $(N2)^2$  se lleva a la mitad inferior de BX  
 Suma los cuadrados, y reserva para el resultado un registro de 16 bits  
 Guarda el resultado en 3002 y 3003  
 Fin

Otra forma más compacta de escribir la secuencia anterior

A 100

xxxx:0100 MOV SI, 2000  
 xxxx:0103 SUB AX, AX  
 xxxx:0105 SUB BX, BX  
 xxxx:0107 SUB DX, DX  
 xxxx:0109 MOV BL, [3000]  
 xxxx:010D MOV AL, [SI + BX]  
 xxxx:010F MOV BL, [3001]  
 xxxx:0113 MOV DL, [SI + BX]  
 xxxx:0115 ADD AX, DX  
 xxxx:0117 MOV [3002], AX  
 xxxx:011A INT 20

SI apunta al comienzo de la lista de datos  
 Pone AX en cero  
 Pone BX en cero  
 Pone DX en cero  
 Carga N1 en BL (con BH = 0)  
 $(N1)^2$  se lleva a la mitad inferior de AX  
 Carga N2 en BL (con BH = 0)  
 $(N2)^2$  se lleva a la mitad inferior de DX  
 Suma los cuadrados, y reserva para el resultado un registro de 16 bits  
 Guarda el resultado en 3002 y 3003  
 Fin

La instrucción MOV DL, [SI + BX], que también se puede escribir MOV DL, [SI] + [BX], está en "modo de direccionamiento indexado".

### EJERCICIO 23: instrucción de división de naturales

Dividir un número natural de 32 bits que está a partir de la dirección 2000 por otro natural de 16 bits que está en la dirección 1000. El resultado guardarlo en 1500 y 1501; y el resto en 1502 y 1503. Los 16 bits de más peso de los 32 bits del dividendo deben ir a DX; y los 16 de menor peso a AX. El divisor debe ir a CX. El resultado queda en AX (no debe superar 16 bits), y el resto va a DX.

A 100

xxxx:0100 MOV AX, [2000]  
 xxxx:0103 MOV DX, [2002]  
 xxxx:0107 MOV CX, [1000]  
 xxxx:010B DIV CX  
 xxxx:010D MOV [1500], AX  
 xxxx:0110 MOV [1502], DX  
 xxxx:0114 INT 20

Carga en AX los 16 bits inferiores del dividendo  
 Carga en DX los 16 bits superiores del dividendo  
 Carga en CX el divisor  
 Realiza la división  
 Guarda en 1500 el cociente  
 Guarda en 1502 el resto  
 Fin

### EJERCICIO 24

Se tiene una lista de números naturales de un byte, que comienza en la dirección 2001, cuya longitud se indica en la dirección 2000. Hallar el promedio de dichos números, y guardarlo en la dirección 1500

<u>A100</u>	
xxxx:0100 MOV CL, [2000]	Carga en CL la longitud de la lista
xxxx:0104 MOV SI, 2001	Sí apunta al comienzo de la lista de datos
xxxx:0107 SUB AX, AX	Pone AX en cero
xxxx:0109 ADD AL, [SI]	Suma a AL un número de la lista apuntada por SI
xxxx:0108 ADC AH, 00	Lleva en AH la suma de acarreos que ocurren cuando las sumas superan el formato de AL (Hace AH <= AH + 00 + Carry = AH + Carry)
xxxx:010E INC SI	Incrementa SI
xxxx:010F DEC CL	Decrementa CL
xxxx:0111 JNZ 109	Mientras Z sea 0, volver a 109
xxxx:0113 MOV BL, [2000]	Carga en BL la cantidad de números de la lista
xxxx:0117 DIV BL	Hace AL / BL; el cociente va a AL, y el resto a AH
xxxx:0119 MOV [2000], AL	Guarda en 2000 el resultado de la división (el promedio)
xxxx:011C INT 20	Fin

**EJERCICIO 25: uso de instrucciones AND, ROL y OR**

Se tiene a partir de la dirección 2001 una lista de dígitos aislados codificados en ASCII, que ocupan un número par de posiciones de memoria. Este número se indica en la dirección 2000. Convértilos cada dos dígitos ASCII consecutivos en un byte BCD empaquetado, que se almacenan a partir de la dirección 3001

Ejemplo:

DATOS

xxxx:2000 04. 31. 32. 35. 38

RESULTADOS

xxxx:3001 12. 58.

A 100

xxxx:0100 MOV DL, [2000]	Carga en DL la longitud de la lista
xxxx:0104 MOV SI, 2001	Inicializa SI para que apunte al comienzo de la lista de datos
xxxx:0107 MOV DI, 3001	Inicializa DI para que apunte al comienzo de la lista de resultados
xxxx:010A MOV AL, [SI]	Carga en AL un dato de la lista
xxxx:010C MOV BL, [SI+1]	Carga en BL el dato siguiente de la lista
xxxx:010F AND AL, 0F	Pone en 0 los 4 bits más altos de AL
xxxx:0111 AND BL, 0F	Pone en 0 los 4 bits más altos de BL
xxxx:0114 MOV CL, 4	Prepara CL para indicar 4 rotaciones
xxxx:0116 ROL AL, CL	Rota 4 veces el contenido de AL
xxxx:0118 OR AL, BL	Forma el byte empaquetado
xxxx:011A MOV [DI], AL	Guarda el byte empaquetado en la otra lista
xxxx:011C INC DI	DI apunta al siguiente lugar
xxxx:011D ADD SI, 2	SI se incrementa 2, pues ya se tomaron dos datos consecutivos
xxxx:0120 SUB DL, 2	A la longitud se le resta 2, pues se tomaron dos datos consecutivos
xxxx:0122 JNZ 10A	Mientras Z sea 0, volver a 10A
xxxx:0125 INT 20	Fin

La instrucción AND AL, 0F realiza bit a bit la operación lógica And ( $\Lambda$ ) entre el valor de AL y el número 0F, conforme a dicha operación And:  $0 \Lambda 0 = 0$ ;  $0 \Lambda 1 = 0$ ;  $1 \Lambda 0 = 0$ ;  $1 \Lambda 1 = 1$ .

Suponiendo que en AL se tenga, como en este ejemplo 31, la operación será bit a bit:  $00110001 = 31$   
 $\Lambda 00001111 = 0F$   
 $00000001$

De esta manera, mediante la "máscara" 0F se enmascaran los 4 bits superiores de 31, obligándolos a ser ceros, manteniéndose sus 4 bits inferiores.

La instrucción -no usada en este ejemplo- ROL AL, 1 (ROlates a register Left = rotar un registro a izquierda) ordena rotar el contenido del registro AL hacia la izquierda conforme indican las flechas de la figura 3.21.a. Esto es, cada bit se mueve una posición hacia la izquierda, siendo que el bit de la extrema izquierda se reinyecta a la derecha. La figura 3.21.b muestra el resultado en AL de ejecutar la instrucción.

El flag C toma el valor del bit que queda en el extremo izquierdo de AL (o el registro que sea).

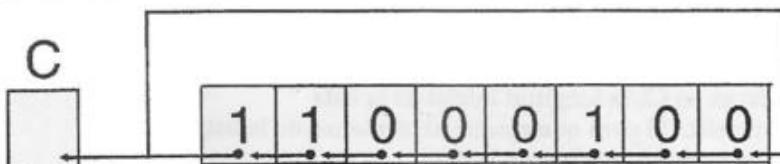


Figura 3.21.a



Figura 3.21.b

Para rotar más que una posición, se debe poner el número de posiciones a rotar en el registro CL en la instrucción anterior (en nuestro caso mediante `MOV CL, 4`) y luego escribir `ROL AL, CL` (si se usa AL).

En el ejemplo de la secuencia anterior, antes de ROL en AL se tenía 00000001, por lo que luego de ejecutar ROL se tendrá 00010000, resultando C = 0.

También existe la instrucción ROR (ROtate a register Right), de función semejante a ROL.

#### EJERCICIO 26:

Un número que está en la dirección 1500 agregarlo al final de una lista no ordenada de números de un byte que empieza en 2001 si dicho número no se encuentra en la lista. Asimismo, aumentar en uno el número de elementos de la lista que está en la dirección 2001.

A 100

```

xxxx:0100 MOV DL, 00
xxxx:0102 MOV CL, [2000]
xxxx:0106 MOV SI, 2001
xxxx:0109 MOV AL, [SI]
xxxx:010B CMP AL, [1500]
xxxx:010F JNZ 113
xxxx:0111 MOV DL, FF
xxxx:0113 INC SI
xxxx:0114 DEC CL
xxxx:0116 JNZ 109
xxxx:0118 CMP DL, FF
xxxx:011B JZ 12A
xxxx:011D MOV AL, [1500]
xxxx:0120 MOV [SI], AL
xxxx:0122 MOV AL, [2000]
xxxx:0113 INC AL
xxxx:0122 MOV [2000], AL
xxxx:012A INT 20

```

Asume con la indicación `DL = 00` que el número no está en la lista  
Carga en CL la longitud inicial de la lista

Inicializa SI para que apunte al comienzo de la lista

Carga en AL un dato de la lista

Compara el número que está en AL con el que está en 1500

Si Z=0 (los números comparados son distintos), saltar a 113

Si Z=1 indica con FF en DL que el número está en la lista

Incrementa SI

Decrementa CL

Mientras Z=0, volver a 109

Compara el número que está en DL con FF

Si es FF (Z=1) salta a fin pues el número está en la lista

Si Z=0 el número no está en la lista, y lo pasa a AL

Lleva el número al final de la lista (El reg. SI fue incrementado en el lazo)

Carga en AL la longitud de la lista

Incrementa la longitud

Guarda en 2000 la nueva longitud de la lista

Fin

#### EJERCICIO 27:

Se tiene una lista de números naturales de un byte, que empieza en la dirección 2001, y se la quiere ordenar en forma creciente. La longitud de la lista está en la dirección 2000

El mecanismo de ordenación que usaremos es el de "burbujeo", como se ejemplifica a continuación

Inicio Primer Luego del 1er Segundo Luego del 2do Tercer Luego del 3er Cuarto Luego 4to  
Ordenamiento ordenam. ordenam. Ordenamiento ordenam ordenamiento ordenam ordenam.



Los círculos indican comparaciones realizadas; y las flechas, los casos en que hubo intercambio de lugar entre dos números.

A 100

xxxx:0100	MOV DL, 1	Asume con la indicación DL = 1 que no se produjo ningún cambio
xxxx:0102	MOV CL, [2000]	Carga en CL la longitud inicial de la lista
xxxx:0106	MOV SI, 2001	Inicializa SI para que apunte al comienzo de la lista
xxxx:0109	MOV AL, [SI]	Carga en AL un dato de la lista
xxxx:010B	CMP AL, [SI+1]	Compara el número que está en AL con el siguiente
xxxx:010E	JBE 117	Si es menor o igual saltar a 117 para seguir recorriendo la lista
xxxx:0110	XCHG AL, [SI+1]	Intercambia el contenido de AL con el de memoria apuntado por SI+1
xxxx:011B	MOV [SI], AL	Guarda el nuevo contenido de AL en la dirección apuntada por SI
xxxx:0115	MOV DL, 0	Asume con la indicación DL = 0 que hubo un intercambio
xxxx:0117	INC SI	Incrementa SI
xxxx:0118	DEC CL	Decrementa CL
xxxx:011A	JNZ 109	Mientras Z=0, volver a 109
xxxx:011C	CMP DL, 0	Compara el número que está en DL con 00
xxxx:011F	JZ 100	Si es 00 (Z=1) salta a 100 para iniciar un nuevo ordenamiento
xxxx:0121	INT 20	Fin (recorriendo la lista no se produjeron más intercambios)

**EJERCICIO 28:**

Se tiene una lista no ordenada de números de un byte que empieza en 2001, siendo que su longitud está en 2000. Si el número que está en la dirección 1500 se encuentra en la lista eliminarlo, desplazando una posición hacia arriba los números que están debajo de dicho número. En caso que no se encuentre en la lista, no modificarla

2000	05
2001	20
2002	52
2003	16
2004	98
2005	90

2000	04
2001	20
2002	16
2003	98
2004	90

Se ha supuesto que el número que está en la dirección 1500 es el 52

A100

xxxx:0100	MOV CL, [2000]	Carga en CL la longitud de la lista
xxxx:0104	MOV SI, 20001	SI apunta al comienzo de la lista de datos
xxxx:0107	MOV AL, [SI]	Carga en AL un dato de la lista apuntada por SI
xxxx:0109	CMP AL, [1500]	Compara el número que está en AL con el que está en 1500
xxxx:010D	JZ 120	Si Z es 1 (los números comparados son iguales), saltar a 120
xxxx:010E	INC SI	Incrementa SI
xxxx:0110	DEC CL	Decrementa CL
xxxx:0112	JNZ 107	Mientras Z=0, volver a 107
xxxx:0114	INT 20	Fin

A 120

xxxx:0120	MOV AL, [SI+1]	Carga en AL el número que sigue al que apunta SI
xxxx:0123	MOV [SI], AL	Guarda donde apunta SI el número que estaba en la dirección siguiente
xxxx:0125	INC SI	Incrementa SI
xxxx:0126	DEC CL	Decrementa CL
xxxx:0129	JNZ 120	Mientras Z=0, volver a 120
xxxx:012A	MOV AL, [2000]	Carga en AL la longitud de la lista
xxxx:012D	DEC AL	Decrementa la longitud
xxxx:012F	MOV [2000], AL	Guarda en 2000 la nueva longitud de la lista
xxxx:0132	INT 20	Fin

**EJERCICIO 29: uso de la instrucción MUL para multiplicar números naturales.**

Secuencia que convierte una lista de números de dos dígitos que están en BCD empaquetado a binario puro. La lista empieza en 2001, y su longitud está en 2000.

Ejemplo:

$$98 = 9 \times 10 + 8$$

$$10011000_{BCD} = 1001 \times 1010 + 1000 = 1100010 = 62\text{ H}$$

Si se multiplican dos números de 4 bits ( $XXXX \times 1010$ ), el resultado entra en 8 bits

A 100

```
xxxx:0100 MOV DL, [2000]
xxxx:0104 MOV SI, 2001
xxxx:0107 MOV DI, 3001
xxxx:010A MOV AL, [SI]
xxxx:010C MOV BL, AL
xxxx:010E AND BL, 0F
xxxx:0111 AND AL, F0
xxxx:0113 MOV CL, 4
xxxx:0115 ROR AL, CL
xxxx:0117 MOV BH, 0A
xxxx:0119 MUL BH
xxxx:010B ADD AL, BL
xxxx:011D MOV [DI], AL
xxxx:011F INC SI
xxxx:0120 INC DI
xxxx:0121 DEC DL
xxxx:0123 JNZ 10A
xxxx:0125 INT 20
```

Carga en DL la longitud de la lista  
 Inicializa SI para que apunte al comienzo de la lista de datos  
 Inicializa DI para que apunte al comienzo de la lista de resultados  
 Carga en AL los dos dígitos en BCD  
 Copia en BL el contenido de AL  
 Pone en 0 los 4 bits más altos de BL  
 Pone en 0 los 4 bits más bajos de AL  
 Prepara CL para indicar 4 rotaciones  
 Rota 4 posiciones a la derecha el contenido de AL  
 El multiplicador BH se hace de valor diez  
 Multiplica AL por BH y el resultado va a AL  
Suma las unidades al producto antes efectuado  
 Guarda el byte empaquetado en la otra lista  
El registro SI apunta al siguiente lugar  
DI apunta al siguiente lugar  
 Decrementa DL  
Mientras Z=0, volver a 10A  
 Fin

**EJERCICIO 30:**

A partir de la dirección 2001 se tiene una lista de 11 dígitos, del 0 al 9, codificados en BCD. Sumarlos en BCD, y el resultado guardararlo en la dirección 1500. El número  $11_b = 0B_H$  está en la dirección 2000.

A100

```
xxxx:0100 MOV CL, [2000]
xxxx:0104 MOV SI, 2001
xxxx:0109 SUB AL, AL
xxxx:0109 ADD AL, [SI]
xxxx:010B DAA
xxxx:010C INC SI
xxxx:010D DEC CL
xxxx:010F JNZ 109
xxxx:0111 MOV [1500], AL
xxxx:0114 INT 20
```

Carga en CL la longitud de la lista  
Registro SI apunta al comienzo de la lista de datos  
Hace cero el registro AL  
Suma un elemento de la lista contra AL  
Corrección para suma BCD ← PARA SE UNA DESPLAZAMIENTO DE UNA UNIDAD EN LA LISTA  
 Incrementa SI  
 Decrementa CL  
Mientras Z no sea 1, volver a 109  
Guarda indicación de paridad  
 Fin

**EJERCICIO 31:**

Se tiene una lista de números de un byte que empieza en la dirección 2001, siendo que su longitud está en la dirección 2000. Desplazar cada número de la lista una posición más abajo.

Determinar si el número contenido en la dirección 1500 se halla en la lista que comienza en la dirección 2001, siendo que la longitud de la lista está en la dirección 2000. En caso que el número citado sea uno de los de la lista (salvo el último), reemplazarlo por el que le sigue, éste por el siguiente, y así sucesivamente, de modo de recibir todos los números que siguen al hallado, subiéndolos una posición. Asimismo, restarle uno a la longitud de la lista que está en la dirección 2000.

## A100

xxxx:0100 MOV SI, 2001  
 xxxx:0107 MOV AL, [1500]  
 xxxx:010A CMP AL, [SI]  
 xxxx:010C JZ 120  
 xxxx:010E INC SI  
 xxxx:010F DEC CL  
 xxxx:0111 JNZ 10A  
 xxxx:0113 INT 20

Registro SI apunta al comienzo de la lista de datos  
 Carga 2E en AL  
 Compara AL con el número apuntado por SI  
 Si son iguales saltar a 120  
 Incrementa SI  
 Decrementa CL  
 Mientras Z=0, volver a 10A  
 Fin

## A 120

xxxx:0120 MOV BL, [2000]  
 xxxx:0124 DEC BL  
 xxxx:0126 MOV [2000], BL  
 xxxx:0129 DEC CX  
 xxxx:012A JZ 136  
 xxxx:012C MOV BL, [SI+1]  
 xxxx:012F MOV [SI], BL  
 xxxx:0131 INC SI  
 xxxx:0132 DEC CL  
 xxxx:0134 JNZ 12C  
 xxxx:0136 INT 20

Carga en BL la longitud de la lista  
 Decrementa la longitud de la lista  
 Guarda la nueva longitud de la lista  
 Decrementar CX  
 Si es el último elemento de la lista saltar a 136  
 Si no es el último, cargar en BL el elemento siguiente  
 Guarda en el lugar de un elemento, el que le sigue  
 Incrementa SI  
 Decrementa CL  
 Mientras Z=0, volver a 12C  
 Fin

## EJERCICIO 32:

Se tiene una lista de números naturales de un byte, ordenados en forma creciente, a partir de la dirección 2001, siendo que en la dirección 2000 está la longitud de dicha lista. En la dirección 1500 se tiene un número natural de un byte. Intercalarlo según su valor en la lista citada, reubicando una posición más abajo cada uno de los números de la lista que le siguen (salvo que sea el último). Asimismo, incrementar en uno la longitud de la lista, que está en la dirección 2000. Si el número a intercalar ya se encuentra en la lista, ésta no debe modificarse

## A100

xxxx:0100 MOV CL, [2000]  
 xxxx:0103 MOV SI, 2001  
 xxxx:0107 MOV AL, [1500]  
 xxxx:010A CMP [SI], AL  
 xxxx:010C JA 120  
 xxxx:010E JZ 115  
 xxxx:0110 INC SI  
 xxxx:0111 DEC CL  
 xxxx:0113 JNZ 10A  
 xxxx:0115 INT 20

Carga en CL la longitud de la lista  
 Registro SI apunta al comienzo de la lista de datos  
 Carga el número a intercalar en AL  
 Compara el número apuntado por SI con AL  
 Si el número de la lista es mayor hay que intercalar, saltar a 120  
 Si son iguales (está en la lista) saltar a 115  
 Incrementa SI  
 Decrementa CL  
 Mientras Z=0, volver a 10A  
 Fin

## A120

xxxx:0120 INC CL  
 xxxx:0122 MOV AH, [SI]  
 xxxx:0124 MOV [SI], AL  
 xxxx:0126 MOV AL, AH  
 xxxx:0128 INC SI  
 xxxx:0129 DEC CL  
 xxxx:012B JNZ 122  
 xxxx:012D MOV AL, [2000]  
 xxxx:0130 INC AL  
 xxxx:0132 MOV [2000], AL  
 xxxx:0135 INT 20

Incrementa CL, pues hay un elemento más  
 Salva el número que apuntaba SI en AH  
 Guarda el número a intercalar (u otro) donde apunta SI  
 Lleva a AL el próximo número a "bajar"  
 Incrementa SI  
 Decrementa CL  
 Mientras Z=0, volver a 122  
 Carga en AL la longitud de la lista  
 Incrementa AL  
 Guarda en 2000 la longitud de la lista incrementada  
 Fin

**EJERCICIO 33:**

Se tiene una lista de caracteres ASCII que empieza en 2000. Determinar el número de caracteres entre el carácter 02 = STX (start of text), y el carácter 03 = ETX (end of text), e indicarlo en la dirección 1500.

A100

```
xxxx:0100 MOV SI, 1FFF
xxxx:0103 INC SI
xxxx:0104 MOV AL, 02
xxxx:0106 CMP [SI], AL
xxxx:0108 JNZ 103
xxxx:010A MOV CX, -1
xxxx:010D MOV AL, 03
xxxx:010F INC CX
xxxx:0110 INC SI
xxxx:0111 CMP [SI], AL
xxxx:0113 JNZ 10F
xxxx:0115 MOV [1500], CX
xxxx:0119 INT 20
```

Registro SI apunta al comienzo de la lista menos uno  
 Incrementa SI  
 Carga 02 = STX en AL  
 Compara el número apuntado por SI con AL  
 Si el carácter de la lista es distinto, saltar a 103  
 Pone en -1 al contador  
 Carga 03 = ETX en AL  
 Incrementa CX  
 Decrementa SI  
 Compara el número apuntado por SI con AL  
 Mientras Z=0, volver a 10F  
 Guarda en 1500 el resultado  
 Fin

**EJERCICIO 34:**

En las direcciones 2000 a 2003 se tiene un número N de 32 bits. Hallar su raíz cuadrada, y el resultado dejarlo en el registro CX.

Primera aproximación  $\sqrt{N} = A_1 = (N/200) + 2$

Segunda aproximación  $\sqrt{N} = A_2 = [(N/A_1) + A_1]/2$

Tercera aproximación  $\sqrt{N} = A_3 = [(N/A_2) + A_2]/2$

.....

Enésima aproximación  $\sqrt{N} = A_n = [(N/A_{n-1}) + A_{n-1}]/2$

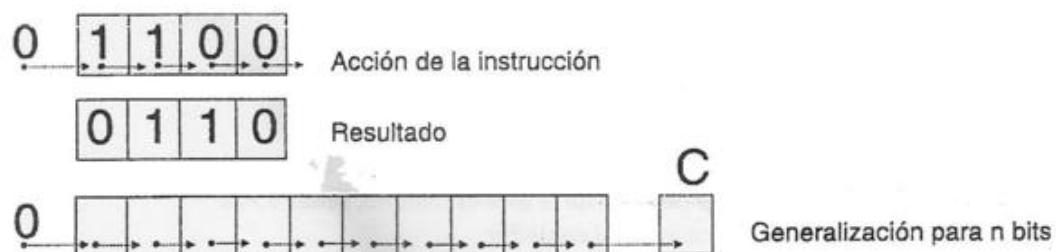
El algoritmo se detendrá cuando la diferencia entre dos aproximaciones sucesivas sea cero, 16-1

A 100

```
xxxx:0100 MOV SI, 2000
xxxx:0103 MOV AX, [SI]
xxxx:0105 MOV DX, [SI+2]
xxxx:0108 MOV CX, C8
xxxx:010B DIV CX
xxxx:010D ADD AX, 2
xxxx:0110 MOV CX, AX
xxxx:0112 MOV AX, [SI]
xxxx:0114 MOV DX, [SI+2]
xxxx:0117 DIV CX
xxxx:0119 ADD AX, CX
xxxx:011B SHR AX, 1
xxxx:011D CMP AX, CX
xxxx:011F JZ 12D
xxxx:0121 SUB CX, AX
xxxx:0123 CMP CX, 1
xxxx:0126 JZ 12D
xxxx:0128 CMP CX, -1
xxxx:012B JNZ 110
xxxx:012D MOV CX, AX
xxxx:012F INT 20
```

Inicializa SI  
 Carga en AX los 16 bits inferiores de N  
 Carga en DX los 16 bits superiores de N  
 El divisor CX toma el valor  $C8_{16} = 200_{10}$   
 Divide por 200, y el resultado queda en AX  
 Suma 2 como pide el algoritmo  
 Lleva A1 (cociente) que está en AX hacia CX (divisor)  
 Vuelve a cargar en AX los 16 bits inferiores de N  
 Carga en DX los 16 bits superiores de N  
 Divide por A1  
 Suma N/A1 que está en AX con A1 que está en CX  
 Divide por 2 la suma antes hallada, para hallar A2  
 Compara A2 con A1  
 Si son iguales, terminar  
 Resta A1 - A2  
 Compara (A1 - A2) contra 1  
 Si es igual a 1, terminar  
 Si no es igual a 1 determina si es -1 = FFFF  
 Si no es igual a -1 se hace otra aproximación  
 Guarda An válida en CX  
 Fin

La instrucción SHR AX, 1 permite rápidamente dividir por dos un número binario, merced a que desplaza hacia la derecha ("shift right"), una posición, los bits del registro AX, y agrega un cero a la izquierda. Esto se indica en la figura que sigue, que ejemplifica para 4 bits, cómo el número 1100 (12) se transforma en el 0110 (6).



En general, para un registro de  $n$  bits, SHR desplaza un lugar hacia la derecha el contenido del registro, llevando un cero a la izquierda. El bit de la extrema derecha pasa a ser el nuevo valor del flag C (Carry). De manera inversa, SHL AX, 1 desplaza un lugar hacia la izquierda (shift left) cada uno de los 16 bits de AX, pone un cero en la posición extrema izquierda, y el bit extremo izquierdo pasa a ser el valor del flag C. Así es posible multiplicar por dos en binario, mediante una sola instrucción, el contenido de AX.

#### EJERCICIO 35: Multiplicación de dos números enteros de 8 bits cada uno mediante IMUL

Multiplicar el número entero que está en la dirección 2000 por el número  $-6_D = 11111010_B = FA_{16}$

Para multiplicar dos números enteros de 8 bits, mediante la instrucción IMUL, uno de ellos puede estar en cualquier registro de 8 bits, y el otro en AL. El resultado, que puede ser un número de hasta 16 bits, se encuentra en el registro AX.

A100

```
xxxx:0100 MOV CL, [2000]
xxxx:0104 MOV AL, FA
xxxx:0106 IMUL CL
xxxx:010A INT 20
```

Carga en CL uno de los números enteros de 8 bits

Carga en AL el otro número entero de 8 bits

Ordena multiplicar CL por AL, con resultado en AX

Fin

#### EJERCICIO 36 : Multiplicación de dos números enteros de 16 bits cada uno mediante IMUL

Multiplicar el número entero que está en la dirección 2000 y 2001 por el número  $-6_D = FFFA_{16}$

Para multiplicar dos números enteros de 16 bits, mediante la instrucción IMUL, uno de ellos puede estar en cualquier registro de 16 bits, y el otro en AX. El resultado, que puede ser un número de hasta 32 bits, se encuentra en los registros DX (mitad superior), y DX (mitad inferior)

A100

```
xxxx:0100 MOV BX, [2000]
xxxx:0104 MOV AX, FFFA
xxxx:0107 IMUL BX
xxxx:0109 INT 20
```

Carga en BX el número entero de 16 bits que está en 2000 y 2001

Carga en AX el otro número entero de 16 bits

Ordena multiplicar BX por AX, con resultado en DX y AX

Fin

#### EJERCICIO 37 : Multiplicación de un número entero de 16 bits por otro de 8 bits, y uso de CBW

Multiplicar el número entero que está en la dirección 2000 y 2001 por el número  $-6_D = FA_{16}$

El número entero de 16 bits, puede estar en cualquier registro de 16 bits, y el de 8 bits en AL. A este último se le debe propagar el signo (unidad 4), mediante la instrucción CBW (Convert byte to word), para que resulte una multiplicación entre dos números de 16 bits.

A100

```
xxxx:0100 MOV BX, [2000]
xxxx:0104 MOV AL, FA
xxxx:0106 CBW
xxxx:0107 IMUL BX
xxxx:0109 INT 20
```

Carga en BX el número entero de 16 bits que está en 2000 y 2001

Carga en AL el otro número entero de 8 bits

Convierte FA que está en AL, en FFFA que ocupa AX

Ordena multiplicar BX por AX, con resultado en DX y AX

Fin

## DIRECCIONES EFECTIVAS Y REGISTROS DE SEGMENTO<sup>1</sup>

Hasta el presente, en una dirección que en el Debug aparecía como 2B16:1723 sólo considerábamos didácticamente la porción derecha 1723. Ello suponía considerar una memoria (figura 3.22) cuyas direcciones iban de 0000 a FFFF = 65.536, o sea una memoria de 64 KB.

En modo real<sup>2</sup>, una dirección efectiva es formada con las dos porciones (2B16 y 1723) que la componen, para lo cual la UCP realiza una suma binaria, que en hexadecimal es:

$$\begin{array}{r}
 2B160 \text{ (dirección base)} \\
 + \underline{1723 \text{ (desplazamiento)}} \\
 \hline
 2C883 \text{ (dirección efectiva)}
 \end{array}$$

Se observa que a 2B16 se le agregó un cero a la derecha (equivalente a cuatro ceros en binario). Lo anterior implica (figura 3.23) que la UCP (80286) direcciona una memoria cuyas direcciones van de 00000 a FFFFF = 1.048.576, o sea una memoria de 1 MB. Las direcciones que van de 0000 a FFFF constituyen un "segmento" de 64 KB de la memoria de 1 MB, dentro del cual el valor 1723 representa un "desplazamiento" (D) u "offset" respecto del origen relativo 0000. Este origen tiene por dirección efectiva 2B160 (2B160 + 0000), siendo que la celda localizada con desplazamiento 1723 tiene como dirección efectiva 2C883 = 2B160 + 1723 = 2B16:1723.

En el espacio de memoria del 80286 cada programa dispone de cuatro segmentos independientes de 64 KB cada uno, direccionados en su origen por un registro denominado **registro de segmento**:

1. **Segmento de código:** donde se guardan los códigos de máquina de las instrucciones que constituyen el programa. Cada instrucción es localizada dentro del segmento por medio del IP, que proporciona el valor del desplazamiento respecto del origen del segmento (figura 3.24). El origen de este segmento es direccionado (apuntado) por el "*registro de segmento de código*" (CS = code segment register). Por lo tanto, una instrucción se localiza en el segmento de código mediante el par de valores CS:IP, que conforman lo que se denomina el "contador de programa". En la figura 3.24 se ha supuesto CS:IP = 302B:B01C.
2. **Segmento de datos:** que guarda los datos que el programa debe operar y los resultados que resulten de la ejecución del mismo. Cada dato es localizado dentro del segmento mediante alguno de estos tres punteros: SI, DI o BX, que proporcionan el valor del desplazamiento en relación con el origen del segmento (figura 3.24). La dirección donde comienza este segmento es proporcionada por el "*registro de segmento de datos*" (DS = data segment register). Conforme a lo anterior, un dato se localiza en el segmento de datos mediante alguno de los siguientes pares de valores: DS:SI, DS:DI o DS:BX. En la figura 3.24 se ha supuesto DS:DI = 5048:251A.
3. **Segmento de pila ("stack"):** almacena direcciones y datos que se ponen en juego durante la ejecución de cada información, información que es localizada dentro del segmento mediante un registro denominado SP ("stack pointer"). El inicio del segmento de pila es direccionado por el "*registro de segmento de pila*" (SS = stack segment register). Por lo tanto el par de valores SS:SP permite localizar en el segmento de pila información dentro de la pila (figura 3.24). En la figura 3.24 se ha supuesto SS:SP = A3232:762A.
4. **Segmento extra:** usado mayormente para guardar datos tipo "strings" (cadenas de caracteres), o como prolongación del segmento de datos. Cada dato es apuntado dentro del segmento por el registro puntero DI (figura 3.24). La dirección de inicio de este segmento es proporcionada por el "*registro de segmento extra*" (ES = extra segment register). Así, el par de valores ES:DI sirve para localizar un dato en el segmento extra. En la figura 3.24 se ha supuesto ES:DI = E230:5420.

Los registros de segmento permiten cambiar de lugar programas, con tan solo cambiar el valor contenido en los mismos.

Si observamos el Debug vemos que normalmente los registros CS, DS, SS y ES están con el valor de la parte izquierda de la dirección (2B16 en un ejemplo anterior). Ello implica que los cuatro segmentos citados empiezan en la misma dirección (2B160), sea que están superpuestos (figura 3.25).

<sup>1</sup> Este punto es necesario para desarrollar los temas siguientes.

<sup>2</sup> Desde el 80286 hasta el Pentium, los procesadores pueden operar en modo real o en modo protegido. Este último es el que se utiliza en multitasking (multiprogramación).

Esta es la forma en que hemos estado usando dichos segmentos, como un solo segmento de 64 KB, útil cuando se tiene un programa pequeño, como son los .COM. En este segmento pueden convivir, en áreas separadas, un programa, datos, y una pila (figura 3.25)

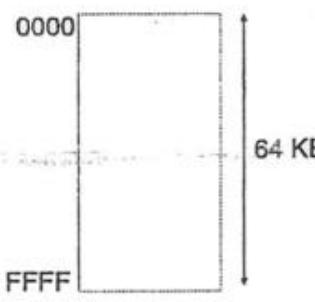


Figura 3.22

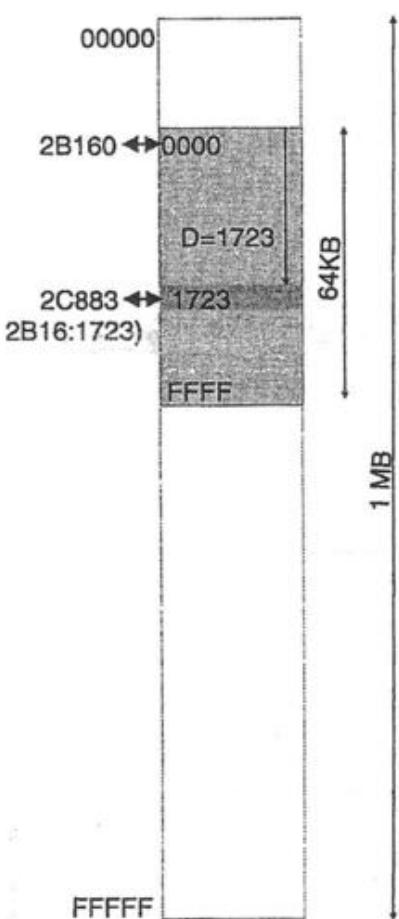
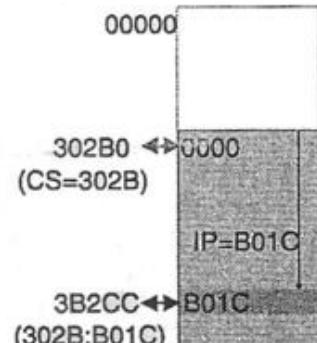


Figura 3.23

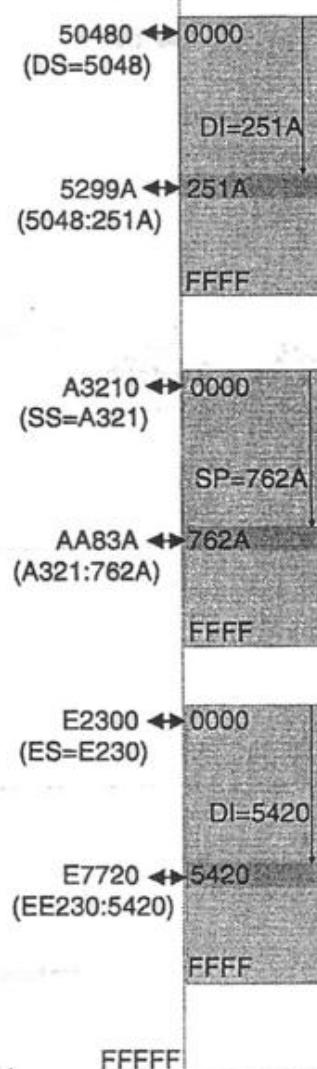


Figura 3.24

Figura 3.25

### LLAMADO A SUBRUTINAS

Una subrutina es una porción específica de un programa, que puede ser incorporada, para ser ejecutada, en cualquier punto de un programa. La acción de requerir la ejecución de una subrutina que se incorpora a un programa, se conoce como "llamado" ("call") a subrutina.

En general, una subrutina es llamada sucesivamente desde distintos puntos del programa principal para proveer una función. En lugar de incluir repetidamente dicha porción de programa en el programa principal, cada vez que se la necesita se la invoca como subrutina, en cada oportunidad que la función se necesite. Así, una misma porción de programa escrito una sola vez puede ser usada muchas veces, con la consiguiente economía de memoria, y en beneficio de una mejor estructuración de los programas.

Otra razón para usar subrutinas es la división de programas largos en módulos más pequeños. Las causas más comunes para codificar un programa en varios subprogramas separados —que luego serán combinados en un único programa ejecutable— pueden ser:

- Facilitar la solución del problema que se quiere resolver
- Agilizar la fase de depuración de errores
- La no existencia de suficientes registros de la UCP para un programa extenso
- La división de un programa largo entre varios programadores
- El aprovechamiento de subrutinas ya existentes

Existen subrutinas que se escriben como programas independientes. Como tales pueden traducirse, probarse y almacenarse para formar parte de una "biblioteca de subrutinas". Esta técnica permite compartir subrutinas entre distintos programas.

#### VISION GLOBAL DEL PROCESO DE LLAMADA A UNA SUBRUTINA (figura 3.26)

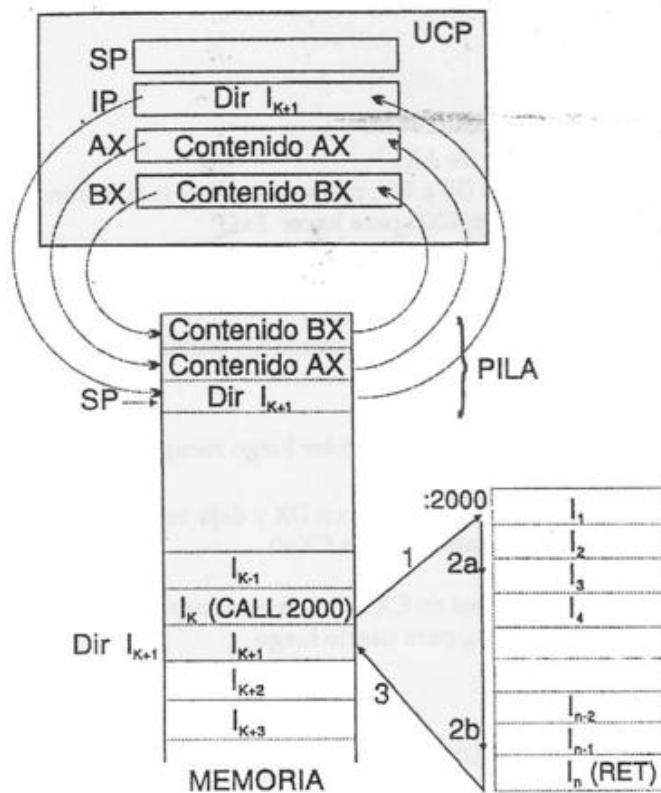


Figura 3.26

1. Cuando desde un programa se quiere pasar a ejecutar una subrutina (SR), una instrucción  $I_k$  del mismo denominada de "llamada a subrutina" (CALL en assembler de Intel), debe ordenar saltar a la primera instrucción ( $I_n$ ) de dicha SR.

Previamente (paso 0)  $I_k$  ordena guardar en memoria principal la dirección de  $I_{k+1}$ , instrucción con la que continuará la ejecución del programa que llamó a la SR, una vez que ésta se haya ejecutado.

2. Luego se ejecutan  $I_n$  y las siguientes instrucciones de la SR.

3. La última de estas instrucciones  $I_n$  (RET en assembler) ordena saltar a la dirección de  $I_{k+1}$  del programa llamador de la SR, o sea retornar a este programa.

La dirección de  $I_{k+1}$ , o sea la "dirección de retorno" es guardada –paso 0 de la ejecución de la instrucción  $I_k$  de llamada– en una zona de la memoria principal denominada "pila" ("stack").

*La pila se usa durante la ejecución de cada subrutina que haya sido llamada.*

La dirección de memoria donde se escribe (o lee) la dirección de retorno es proporcionada por un registro de la UCP denominado "puntero de pila" ("stack pointer" = SP).

Como se verá, la denominación "pila" se debe a que al comienzo de la ejecución de la SR (2a) instrucciones ordenan guardar en la pila contenidos de registros de la UCP, que venía usando el programa llamador. Dichos contenidos se "apilan" –como una pila de platos– encima de la dirección de retorno citada. Las últimas instrucciones de la SR (2b) ordenan "desapilar", o sea ir

pasando ordenadamente, de la pila hacia los registros, los contenidos que tenían en el momento del llamado a SR. El registro SP es usado para apilar y desapilar, en forma ordenada, los contenidos de la pila. El mismo es manejado automáticamente por la UC, la cual por su intermedio controla la pila.

## DESCRIPCION MAS DETALLADA DEL PROCESO

### I Codificación de instrucciones

Ejercicio 38:

Las siguientes sentencias en alto nivel:

Unsigned

$N = R + Q$

$R = P \times Q$

$T = N \times Q$

serán traducidas a instrucciones en assembler, con llamadas a SR para efectuar las multiplicaciones. Siempre con fines didácticos seguiremos usando para multiplicar (con sus limitaciones) la secuencia de la figura 3.5, asumiendo para las variables las siguientes direcciones de memoria:

$R \rightarrow 4000/1$

$Q \rightarrow 5000/1$

$P \rightarrow 5500/1$

$T \rightarrow 6000/1$

$N \rightarrow 6500/1$

Entonces, una forma de presentar las instrucciones es la siguiente:

A 500

xxxx:0500 MOV CX, [5000]  
 xxxx:0504 MOV AX, [4000]  
 xxxx:0507 ADD AX, CX  
 xxxx:0509 MOV [6500], AX  
 xxxx:050C MOV DX, [5500]  
 xxxx:0510 CALL 2000  
 xxxx:0513 MOV [4000], AX  
 xxxx:0516 MOV DX, [6500]  
 xxxx:0519 CALL 2000  
 xxxx:051C MOV [6000], AX

La variable Q pasa a CX<sup>1</sup>  
 La variable R pasa a AX  
 Realiza en AX:  $N = R + Q$   
 Asigna a la variable N el valor de AX  
 Pasa el valor del multiplicando (P) a DX, pues así lo exige la subrutina  
 Llama a la subrutina que está en 2000 para hacer PxQ  
 Asigna a la variable R el resultado de PxQ que la SR acumuló en AX  
 Pasa el valor del multiplicando (N) a DX, pues así lo exige la subrutina  
 Vuelve a llamar a la subrutina para hacer NxQ  
 Asigna a la variable T el resultado de PxQ que la SR acumuló en AX

A 2000

xxxx:2000 PUSH CX  
 xxxx:2001 SUB AX, AX  
 xxxx:2003 ADD AX, DX  
 xxxx:2005 DEC CX  
 xxxx:2006 JNZ 2003  
 xxxx:2008 POP CX  
  
 xxxx:2009 RET

Guarda en la pila el valor de CX=Q para poder luego recuperarlo  
 Pone AX en cero  
 Suma el valor del multiplicando supuesto en DX y deja resultado en AX  
 Resta en cada ciclo uno a CX, hasta que sea CX=0  
 Mientras Z sea 0, volver a 2003  
 Pasa el valor Q de la pila a CX, así en CX se vuelve a recuperar el valor de Q (que en la subrutina se hace 0), para usarlo luego  
 Vuelve al programa llamador.

Figura 3.27

En el programa llamador, antes de cada CALL, se pasa al registro DX el valor del multiplicando (R ó P según sea), y al comienzo del programa se pasó el valor del multiplicador (Q) a CX. Esta exigencia surge de observar la subrutina, dado que ésta emplea los registros DX y CX para dichos datos. Asimismo, la subrutina deja en AX el resultado, el cual en el programa es asignado en memoria a la variable correspondiente.

Este pase de datos del programa a la subrutina, y de resultados de la subrutina al programa se conoce como "pasaje de parámetros". Los parámetros se han pasado a registros. Existe otra forma de pasaje de parámetros por medio de la pila, que luego se ilustrará.

<sup>1</sup> Esto, como se verá, también lo exige la subrutina para hacer el producto

## II. Uso de la pila ("stack") y del stack pointer (SP)

Como se planteó anteriormente, la pila es una zona de memoria principal, usada para almacenar temporalmente direcciones y datos relacionados con la ejecución de subrutinas. El puntero de pila (SP) es un registro de la UCP que proporciona la dirección del último dato que se ha almacenado. En lo que sigue veremos cómo progresa el SP desde que se llama a una subrutina hasta que se retorna al programa llamador. *Es conveniente seguir este proceso usando el Debug.*

Observando el Debug (ejemplificado a continuación), al comenzar la secuencia, cuando no se usa la pila, permanece  $SP = FFEE$ , o sea que determina la dirección de memoria FFEE (figura 3.28). Si se ejecuta la instrucción CALL 2000 de la secuencia anterior, luego constataríamos con el Debug que en la pila se guarda la dirección de retorno 0513, y que en correspondencia  $SP=FFEC$  (figura 3.29), indicando la nueva cima de la pila, siendo que se ha apilado el valor 0513. El valor de SP cambia, pues, *automáticamente*, mientras se ejecuta CALL 2000: no es necesario insertar en el programa ninguna instrucción que modifique el valor del SP.

La siguiente instrucción que se pasa a ejecutar es la primera de la subrutina, o sea PUSH CX. Esta instrucción ordena guardar en la pila el contenido (supuesto 0002) del registro CX (figura 3.30). En consonancia con el nuevo apilamiento, el valor del SP pasará al valor FFEA, apuntando al dato que está en la cima de la pila.

Las instrucciones SUB AX, AX; ADD AX, DX; DEC CX y JNZ 2003 no afectan a la pila. Hacia el final de la subrutina, la instrucción POP CX ordena un movimiento contrario a PUSH CX. Esto es, retornar el dato (0002) que está en la cima de la pila (apuntado por SP) al registro CX (figura 3.31). Además POP produce un desapilamiento, siendo que el SP apunta ahora a FFEC. Siempre que se desapila un dato, el mismo deja de formar parte de la pila, aunque momentáneamente siga en la memoria. Al desapilar sólo cambia el valor del SP.

La instrucción RET, última de la subrutina también afecta a la pila. El desapilamiento que ocasiona hace que el valor de la dirección de retorno se restaure en el IP, de modo que se puede reanudar el programa en la instrucción que sigue a CALL (figuras 3.27 y 3.32). Asimismo, luego de ejecutar RET, el SP pasa al valor FFEE, que tenía antes que se llamase a la subrutina.

Con el Debug, mediante el comando E se examina la pila en distintas oportunidades.

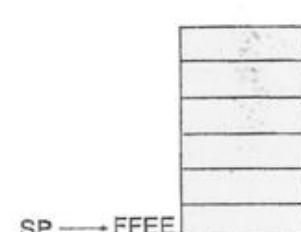


Figura 3.28

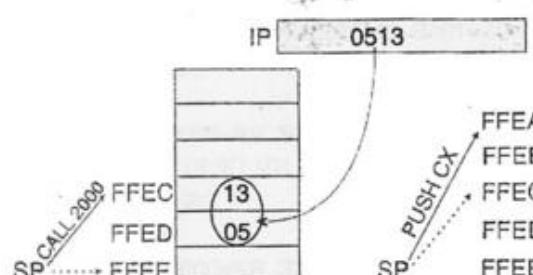


Figura 3.29

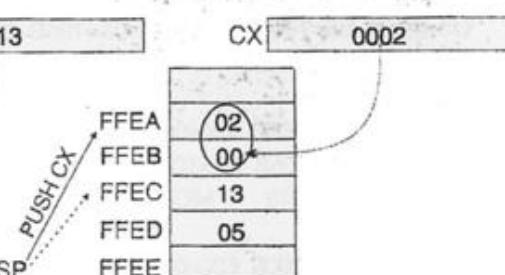


Figura 3.30

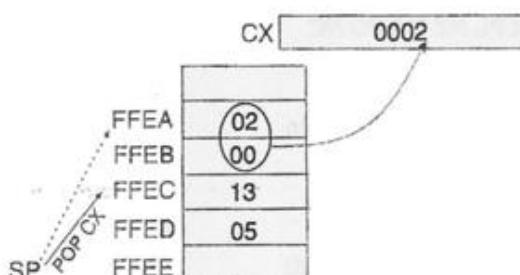


Figura 3.31

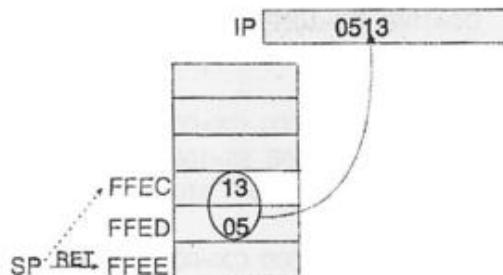


Figura 3.32

De la descripción anterior resultan las siguientes características de la pila:

La pila guarda temporalmente datos en forma ordenada, mientras se ejecuta una subrutina. Los datos tienen una restricción de acceso: sólo pueden agregarse o eliminarse por un extremo de la pila, conocido como "cima".

El último dato colocado en la pila es el primero en quitarse, cuando se comienza a retirar datos. En inglés esto lo expresan las siglas LIFO: "last in, first out".

Se usa la palabra introducir o empujar ("push") para indicar la operación de colocar un nuevo dato en la cima de la pila; y sacar o tirar ("pop") para señalar la operación de retirar el dato de la cima.

El puntero de pila (SP) proporciona la dirección del último dato almacenado en la pila, que es el que está en la cima de ella.

Cada vez que 2 ó 4 bytes son almacenados ("apilados") en la pila, el valor del SP es previamente decrementado en 2 ó 4, respectivamente. Cuando 2 ó 4 bytes son desapilados de la cima, a posteriori el SP es incrementados en 2 ó 4, respectivamente.

El SP controla el orden de la pila, y el SP es controlado por la UC durante la ejecución de instrucciones que afectan a la pila (como CALL, PUSH, POP y RET)

A los efectos de visualizar mediante el Debug el proceso indicado en las figuras 3.28 a 3.32 se ejecutará la secuencia de la figura 3.27, desde la dirección 0500 a 0513. Previamente será necesario suponer valores para las variables. Asumiremos:

R (4000/1) = 0005<sub>H</sub>  
 Q (5000/1) = 0002<sub>H</sub>  
 P (5500/1) = 0004<sub>H</sub>

Estos valores se escriben en memoria, en las posiciones de las variables, mediante el comando E, como se hizo en la unidad 1.

```

IP 0100
:500
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0500 NV UP EI PL NZ NA PO NC
106E:0500 8B0E0050 MOV CX,[5000] DS:5000=0002
-T
AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0504 NV UP EI PL NZ NA PO NC
106E:0504 A10040 MOV AX,[4000] DS:4000=0005
-T
AX=0005 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0507 NV UP EI PL NZ NA PO NC
106E:0507 01C8 ADD AX,CX
-T
AX=0007 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0509 NV UP EI PL NZ NA PO NC
106E:0509 A30065 MOV [6500],AX DS:6500=0000
-T
AX=0007 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=050C NV UP EI PL NZ NA PO NC
106E:050C 8B160055 MOV DX,[5500] DS:5500=0004
-T
AX=0007 BX=0000 CX=0002 DX=0004 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0510 NV UP EI PL NZ NA PO NC
106E:0510 E8ED1A CALL 2000
-T
AX=0007 BX=0000 CX=0002 DX=0004 SP=FFEC BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2000 NV UP EI PL NZ NA PO NC
106E:2000 51 PUSH CX

```

-E FFEC (Examina la pila)

106E:FFEC 13. 05. (Se verifica que en la pila guardó en FFED el 05, y en FFEC el 13, de 0513, que es la dirección de retorno)

-T

```
AX=0007 BX=0000 CX=0002 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2001 NV UP EI PL NZ NA PO NC
106E:2001 29C0 SUB AX, AX
```

-E FFEA

106E:FFEA 02. 00. 13. 05. (Se verifica que en la pila guardó en FFEB el 00, y en FFEA el 02, de 0002 contenido en CX, manteniéndose 13 en FFEC, y 05 en FFED)

-T

```
AX=0000 BX=0000 CX=0002 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2003 NV UP EI PL ZR NA PE NC
106E:2003 01D0 ADD AX, DX
```

(Se observa que la pila no cambió, pues SP continúa apuntando a la cima que está en FFEA)

-T

```
AX=0004 BX=0000 CX=0002 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2005 NV UP EI PL NZ NA PO NC
106E:2005 49 DEC CX
```

-T

```
AX=0004 BX=0000 CX=0001 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2006 NV UP EI PL NZ NA PO NC
106E:2006 75FB JNZ 2003
```

-T

```
AX=0004 BX=0000 CX=0001 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2003 NV UP EI PL NZ NA PO NC
106E:2003 01D0 ADD AX, DX
```

-T

```
AX=0008 BX=0000 CX=0001 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2005 NV UP EI PL NZ NA PO NC
106E:2005 49 DEC CX
```

-T

```
AX=0008 BX=0000 CX=0000 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2006 NV UP EI PL ZR NA PE NC
106E:2006 75FB JNZ 2003
```

-T

```
AX=0008 BX=0000 CX=0000 DX=0004 SP=FFE A BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2008 NV UP EI PL ZR NA PE NC
106E:2008 59 POP CX
```

-T

```
AX=0008 BX=0000 CX=0002 DX=0004 SP=FFEC BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=2009 NV UP EI PL ZR NA PE NC
106E:2009 C3 RET
```

-E FFEC

106E:FFEC 13. 05. (El SP pasó a FFEC donde guarda 13, siendo que en FFED guarda 05)

-T

```
AX=0008 BX=0000 CX=0002 DX=0004 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0513 NV UP EI PL ZR NA PE NC
106E:0513 A30040 MOV [4000],AX DS:4000=0005
```

(Luego de ejecutar RET, el SP volvió a su valor inicial FFEE, y la dirección 0513, guardada en la pila, pasó al IP)

-T

```
AX=0008 BX=0000 CX=0002 DX=0004 SP=FFEE BP=0000 SI=0000 DI=0000
DS=106E ES=106E SS=106E CS=106E IP=0516 NV UP EI PL ZR NA PE NC
106E:0516 8B160065 MOV DX,[6500] DS:6500=0007
```

**EJERCICIO 39:**

A partir de la dirección 2000 existe una lista de números naturales de 2 bytes, cuya longitud está en la dirección 1500. Determinar la raíz cuadrada de cada uno, sumarle 3, y al número resultante guardarlo en otra lista a partir de la dirección 3000. Este problema se basa en el ejercicio 34.

a) Resolución por medio de pasaje de parámetros a través de registros (en este caso se usó BX)

A 100

xxxx:0100	MOV CL, [1500]	Carga en CL la longitud de la lista
xxxx:0104	MOV SI, 2000	El registro SI apunta al comienzo de la lista de datos
xxxx:0107	MOV DI, 3000	DI apunta al comienzo de la lista de resultados
xxxx:010A	MOV BX, [SI]	Carga en BX (para la subrutina) un dato de la lista apuntada por SI
xxxx:010C	CALL 5000	Llama a subrutina que está en 5000 para calcular raíz cuadrada
xxxx:010F	MOV AX, 3	Carga en AX el valor 3
xxxx:0112	ADD AX, BX	Suma 3 al resultado de la raíz cuadrada dejado por la subrutina en BX
xxxx:0114	MOV [DI], AX	Guarda el número resultante en la lista apuntada por DI
xxxx:0116	ADD SI, 2	Incrementa en 2 el puntero SI
xxxx:0119	ADD DI, 2	Incrementa en 2 el puntero DI
xxxx:011C	DEC CL	Decrementa CL
xxxx:011E	JNZ 10A	Mientras Z=0, volver a 10A
xxxx:0120	INT 20	Fin

A 5000

xxxx:5000	PUSHF	Guarda los flags (Registro de Estado) en la pila <sup>1</sup>
xxxx:5001	PUSH AX	Guarda AX en la pila
xxxx:5002	PUSH CX	Guarda BX en la pila
xxxx:5003	PUSH DX	Guarda DX en la pila
xxxx:5004	SUB DX, DX	Lleva a cero el registro DX
xxxx:5006	MOV AX, BX	Carga en AX el número N de 16 bits (dejado en BX) cuya raíz se quiere hallar
xxxx:5008	MOV CX, C8	El divisor CX toma el valor $C8_{16} = 200_{10}$
xxxx:500B	DIV CX	Divide por 200, y el resultado queda en AX
xxxx:5010D	ADD AX, 2	Suma 2 como pide el algoritmo
xxxx:5010	MOV CX, AX	Lleva A1 (cociente) que está en AX hacia CX (divisor)
xxxx:5012	SUB DX, DX	Lleva a cero el registro DX
xxxx:5014	MOV AX, BX	Vuelve a cargar en AX el número N de 16 bits
xxxx:5016	DIV CX	Divide por A1
xxxx:5018	ADD AX, CX	Suma N/A1 que está en AX con A1 que está en CX
xxxx:501A	SHR AX, 1	Divide por 2 la suma antes hallada, para hallar A2
xxxx:501C	CMP AX, CX	Compara A2 con A1
xxxx:501E	JZ 502C	Si son iguales, terminar
xxxx:5020	SUB CX, AX	Resta A1 - A2
xxxx:5022	CMP CX, 1	Compara (A1 - A2) contra 1
xxxx:5025	JZ 502C	Si es igual a 1, terminar
xxxx:5027	CMP CX, -1	Si no es igual a 1 determina si es -1 = FFFF
xxxx:502A	JNZ 5010	Si no es igual a -1 se hace otra aproximación
xxxx:502C	MOV BX, AX	Guarda An válida en BX
xxxx:502E	POP DX	Restaura DX desde la pila
xxxx:502F	POP CX	Restaura CX desde la pila
xxxx:5030	POP AX	Restaura AX desde la pila
xxxx:5031	POPF	Restaura los flags en el Registro de Estado desde la pila
xxxx:5032	RET	Retorna al programa principal

b) Resolución por medio de pasaje de parámetros a través de la pila

<sup>1</sup> En general, esto es necesario hacerlo, dado que cuando se ejecute la subrutina cambiará el valor de los flags, siendo que se requiere volver a la secuencia principal con los mismos valores que tenían los flags antes del llamado a la subrutina. Como se verá, en cualquier tipo de interrupción se guardan automáticamente los flags, o sea el Registro de Estado, en la pila.

A 100	
xxxx:0100 MOV CL, [1500]	Carga en CL la longitud de la lista
xxxx:0104 MOV SI, 2000	El registro SI apunta al comienzo de la lista de datos
xxxx:0107 MOV DI, 3000	DI apunta al comienzo de la lista de resultados
xxxx:010A MOV AX, [SI]	Carga en AX un dato de la lista apuntada por SI
xxxx:500C PUSH AX	Guarda AX en la pila (es el número N cuya raíz se quiere hallar) <sup>1</sup>
xxxx:010D CALL 5000	Llama a subrutina para calcular raíz cuadrada que está en 5000
xxxx:0110 POP AX	Restaura AX desde la pila (la raíz cuadrada calculada)
xxxx:0112 ADD AX, 3	Suma 3 a la raíz cuadrada calculada
xxxx:0114 MOV [DI], AX	Guarda el número resultante en la lista apuntada por DI
xxxx:0116 ADD SI, 2	Incrementa en 2 el puntero SI
xxxx:0119 ADD DI, 2	Incrementa en 2 el puntero DI
xxxx:011C DEC CL	Decrementa CL
xxxx:011E JNZ 10A	Mientras Z no sea 1, volver a 10A
xxxx:0120 INT 20	Fin
A 5000	
xxxx:5000 PUSHF	Guarda los flags en la pila (designados REH y REL en la figura 3.33)
xxxx:5001 PUSH AX	Guarda AX en la pila (designado por sus porciones AH y AL en la figura 3.33)
xxxx:5002 PUSH CX	Guarda CX en la pila (en sus porciones CH y CL en la figura 3.33)
xxxx:5003 PUSH DX	Guarda DX en la pila (en sus porciones DH y DL en la figura 3.33)
xxxx:5004 PUSH BP	Guarda BP en la pila (en sus porciones BPH y BPL en la figura 3.33)
xxxx:5005 MOV BP, SP	Carga en BP el valor de SP, así también BP apunta a la cima de la pila
xxxx:5007 SUB DX, DX	Lleva a cero el registro DX, pues los números son de 16 bits
xxxx:5009 MOV AX, [BP+C]	Carga en AX el número cuya raíz se quiere hallar <sup>2</sup>
xxxx:500C MOV CX, C8	El divisor CX toma el valor $C8_{16} = 200_8$
xxxx:500F DIV CX	Divide por 200, y el resultado queda en AX
xxxx:0111 ADD AX, 2	Suma 2 como pide el algoritmo
xxxx:5014 MOV CX, AX	Lleva A1 (cociente) que está en AX hacia CX (divisor)
xxxx:5016 SUB DX, DX	Lleva a cero el registro DX
xxxx:5018 MOV AX, [BP+C]	Carga en AX el número N cuya raíz se quiere hallar
xxxx:501B DIV CX	Divide por A1
xxxx:501D ADD AX, CX	Suma N/A1 que está en AX con A1 que está en CX
xxxx:501F SHR AX, 1	Divide por 2 la suma antes hallada, para hallar A2
xxxx:5021 CMP AX, CX	Compara A2 con A1
xxxx:5023 JZ 5031	Si son iguales, terminar
xxxx:5025 SUB CX, AX	Resta A1 - A2
xxxx:5027 CMP CX, 1	Compara (A1 - A2) contra 1
xxxx:502A JZ 5031	Si es igual a 1, terminar
xxxx:502C CMP CX, -1	Si no es igual a 1, determina si es $-1 = FFFF$
xxxx:502F JNZ 5014	Si no es igual a $-1$ , se hace otra aproximación
xxxx:5031 MOV [BP+C], AX	Guarda An válida en la pila, en lugar del dato cuya raíz se quería calcular <sup>3</sup>
xxxx:5034 POP BP	Restaura BP desde la pila
xxxx:5035 POP DX	Restaura DX desde la pila
xxxx:5036 POP CX	Restaura CX desde la pila
xxxx:5037 POP AX	Restaura AX desde la pila
xxxx:5038 POPF	Restaura los flags desde la pila
xxxx:5039 RET	Retorna al programa principal

<sup>1</sup> Indicado como AH y AL en la base de la pila de la figura 3.33<sup>2</sup> Este nuevo apilamiento de AX, o sea de N, indicado también como AH y AL, puede servir, por ejemplo, para hacer $\sqrt{N + N}$  (en vez de  $\sqrt{N + 3}$ ) en cuyo caso la secuencia principal se podría modificar como sigue:

CALL 5000

MOV DX, AX

POP AX

ADD AX, DX

<sup>3</sup> Figura 3.3: el número N cuya raíz se quiere hallar está en la base de la pila, y siendo que ahora BP apunta a la cima, entre ésta y el comienzo de N en este caso existen doce posiciones (C posiciones en hexa).<sup>4</sup> Serán los nuevos valores que toman los denominados AH y AL en la base de la pila de la figura 3.33

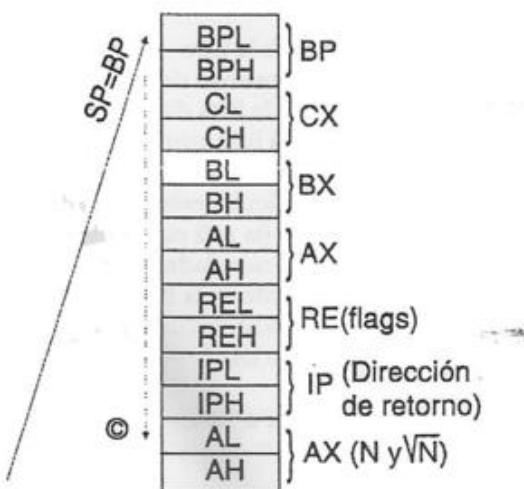


Figura 3.33

### Anidamiento de subrutinas

Una subrutina puede llamar a otra, ésta a una tercera, y así sucesivamente. Este proceso se conoce como *"anidamiento de subrutinas"*. Cuando la última subrutina llamada termina de ejecutarse, se retorna a la subrutina que la llamó, ésta a la que la llamó, y así de seguido. De este modo los retornos se van sucediendo en orden inverso al de las llamadas. El último retorno será hacia el programa principal, que efectuó el primer llamado. Conforme a lo visto, el mecanismo de operación de la pila permite cuidar el orden en el anidamiento de subrutinas.

### DETALLE DE LA EJECUCION DE INT (INTERRUPCION POR SOFTWARE)

En la Unidad 1, sección 1.11, se introdujeron los conceptos fundamentales acerca de las interrupciones por hardware y software, habiéndose planteado que en esencia son una forma de llamado a subrutina. Como se estableció, la ejecución de la instrucción INT xx da lugar a una interrupción por software, en el sentido que desde un programa de usuario no se llama a una porción del mismo, sino a un subprograma de la ROM BIOS o del sistema operativo. Vale decir, se suspende momentáneamente la ejecución de un programa, y en su lugar se pasa a ejecutar dicho subprograma de la ROM BIOS o del sistema operativo llamado por el programa en cuestión. Por tal motivo se habla de *"autointerrupción"*, provocada por la ejecución de INT xx.

Puesto que una interrupción es una forma de llamado a subrutina, en relación con el uso de la pila valen las consideraciones hechas al tratar ese tema, como se exemplificará.

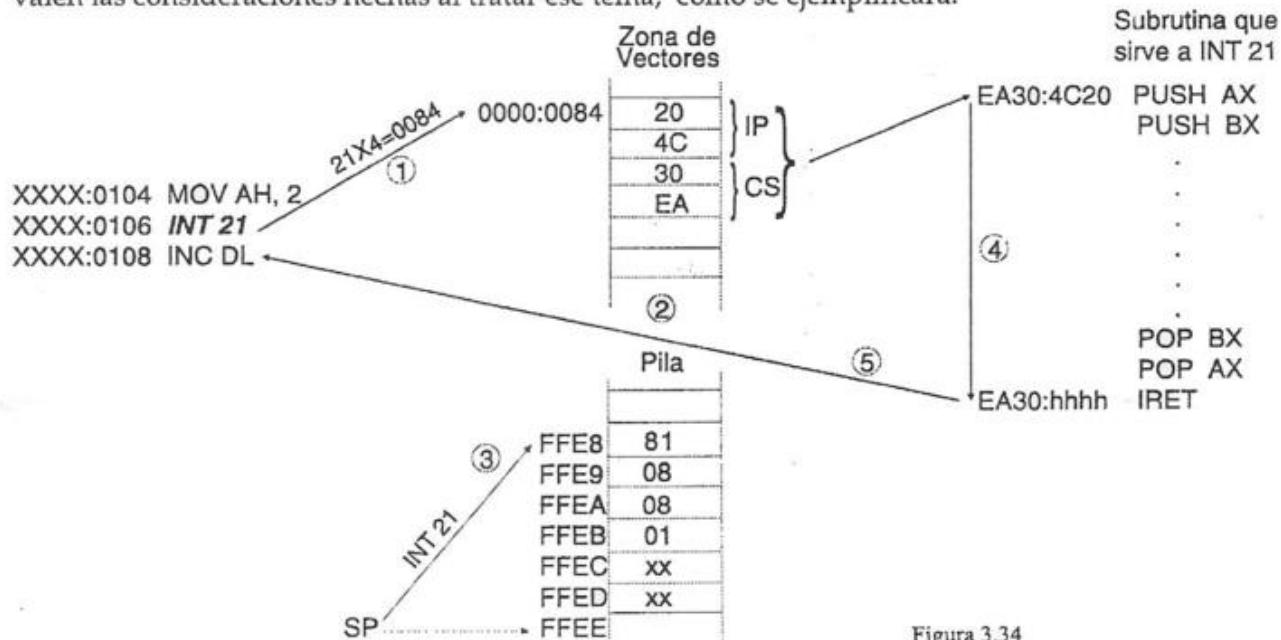


Figura 3.34

En otros procesadores la instrucción INT tiene otros nombres, como SVC, TRAP, etc.

Dada la secuencia:

```
xxxx:0104 MOV AH, 2
xxxx:0106 INT 21
xxxx:0108 INC DL
```

durante la ejecución de una instrucción INT xx, como INT 21, la UC realiza los siguientes pasos (figura 3.34):

1. Multiplica por 4 el número que acompaña a INT. Por ejemplo, (en hexa)  $21 \times 4 = 84 = 0084$
2. El resultado obtenido es el desplazamiento dentro del primer segmento donde empieza la memoria principal (CS = 0000) denominada "zona de vectores interrupción". Esto es, la UC forma la dirección 0000:0084. Esta es la dirección donde se encuentra la dirección CS:IP de la subrutina que atiende a INT 21. O sea que 0084 es la dirección, de la dirección (CS:IP) de la subrutina que sirve a INT 21, supuesta en la dirección CS:IP = EA30:4C20.  
La dirección CS:IP apunta al comienzo de la subrutina, por lo cual se llama "vector interrupción"
3. La UC guarda en la pila la dirección de retorno (xxxx:0108) de la instrucción que sigue a INT 21. Sobre ella apila el registro de estado (RE) conteniendo los flags (entre otros SZVC<sup>1</sup>). Por lo tanto, el SP pasa de FFEE a FFE8.  
El RE en el 80286 tiene 16 bits. Los flags están dispuestos como sigue (figura 3.35)



Figura 3.35

Suponiendo que los flags SZVC tengan los valores supuestos a continuación, el contenido del RE en hexa será (como se ha guardado en la figura 3.34):

0	0	0	0	1	0	0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

= 0881

4. Ejecuta la subrutina llamada por INT 21. Las instrucciones PUSH con que empieza ésta, ordenarán guardar en la pila el contenido de registros de la UCP. Asimismo, las instrucciones POP que están al final de la subrutina ordenarán restaurar desde la pila los contenidos de los registros guardados mediante las instrucciones PUSH.
5. La última instrucción de la subrutina, que necesariamente debe ser IRET ordena retornar a la instrucción que sigue a INT 21 en el programa llamador (o sea a INC DL). La dirección de esta instrucción está en la pila. También retorna desde la pila el contenido del RE. Luego de ejecutarse IRET, el SP vuelve a su valor inicial FFEE.

Salvo pequeñas diferencias, se observan grandes similitudes con lo visto al tratar el llamado a subrutina. Las diferencias consisten, en primer lugar, en que con INT xx la dirección de la subrutina se halla a través de la zona de vectores (figura 3.33). Por otro lado, con INT además de salvarse la dirección de retorno (CS:IP), también se guarda en la pila el registro RE (flags). Por ello, la instrucción de retorno se llama IRET ("interrupt return"), dado que además de la dirección de retorno, ordena restaurar los flags en RE. A continuación se dan casos de uso de INT 21<sup>2</sup>

Secuencia para leer los contenidos (binarios) de dos posiciones consecutivas de memoria, de direcciones 2000 y 2001, y mostrarlos en hexa en la pantalla en orden inverso (como se establece en Intel)

A 100

xxxx:0100 MOV BX, [2000]	Carga el contenido de 2000 y 2001 en BL y BH, respectivamente
xxxx:0104 MOV CH, 4	Carga en CH el número de loops a repetir
xxxx:0106 MOV CL, 4	Carga en CL la cantidad de bits a rotar en BX
xxxx:0108 ROL BX, CL	Rota a izquierda 4 veces en BX
xxxx:010A MOV AL, BL	Carga en AL el valor de BL

<sup>1</sup> El flag A, es el Auxiliar Carry, usado para operar en BCD. El flag T (Trap) se usa por ejemplo cuando en el Debug se ejecuta instrucción por instrucción.. Los flags I y D se describen más adelante. No se han dibujado dos flags para modo protegido.

<sup>2</sup> Los valores que se dan a AH (pasaje de parámetros) se obtienen de Internet y de manuales.

xxxx:010C AND AL, 0F	Pone en 0 el cuarteto superior de AL
xxxx:010E ADD AL, 30	Suma 30 para convertir el número de AL en ASCII
xxxx:0110 CMP AL, 39	Compara con $39_{ASCII} = 9$
xxxx:0112 JBE 116	Pregunta si es < 39
xxxx:0114 ADD AL, 07	Si es mayor suma 7 (por ej. $3A + 7 = 41 = A$ )
xxxx:0116 MOV DL, AL	Para el BIOS carga el número de AL en DL
xxxx:0118 MOV AH, 2	Inicializa parámetros para ver en pantalla caracteres ASCII
xxxx:011A INT 21	Llama a subrutina del BIOS
xxxx:011C DEC CH	Decrementa cantidad de loops a realizar
xxxx:011E JNZ 106	Mientras Z=0, volver a 106
xxxx:0120 INT 20	Fin

Secuencia para visualizar en pantalla  $255 = FF_H$  caracteres ASCII, a partir del carácter ASCII 00

A 100

xxxx:0100 MOV CL, FF	Carga en CL el total de repeticiones a realizar
xxxx:0102 MOV DL, 00	Carga en DL el código ASCII del primer carácter a visualizar
xxxx:0104 MOV AH, 2	Inicializa parámetros para ver en pantalla caracteres ASCII
xxxx:0106 INT 21	Llama a subrutina del BIOS
xxxx:0108 INC DL	Para el BIOS carga el número de AL en DL
xxxx:010A DEC CL	Determina el siguiente carácter ASCII a visualizar
xxxx:010C JNZ 104	Mientras Z=0, volver a 104
xxxx:010E INT 20	Fin

Secuencia para ingresar caracteres por teclado, cuyo número se indica en la dirección 2000, los cuales a medida que van ingresando se observan en pantalla y se deben guardar a partir de la dirección 2001. El programa se queda esperando hasta que se han tipeado todos los caracteres. Tiene la limitación que al apretar la tecla Enter se vuelve al comienzo de renglón, sin saltar al renglón siguiente.

A 100

xxxx:0100 MOV CL, [2000]	Carga en CL el número de caracteres a tipear
xxxx:0104 MOV DI, 2001	Inicializa a DI en 2001
xxxx:0107 MOV AH, 1	Inicializa parámetros para entrar un carácter por teclado, quedando esperando el programa hasta que se tipee el carácter, el cual luego aparece por pantalla
xxxx:0109 INT 21	Llama a subrutina del DOS
xxxx:010B MOV [DI], AL	Guarda cada carácter tipeado en lista apuntada por DI
xxxx:010D INC DI	Incrementa DI
xxxx:010E DEC CL	Decrementa cantidad de repeticiones a realizar
xxxx:0110 JNZ 109	Mientras Z=0, volver a 109
xxxx:0112 INT 20	Fin

Secuencia que combina la secuencia anterior para ingresar caracteres por teclado y visualizarlos, con la que visualiza un carácter que está en el registro DL. El número de caracteres a tipear se indica en la dirección 2000, siendo que los mismos a medida que van ingresando deben observarse repetidos en pantalla. El programa se queda esperando hasta que se han tipeado todos los caracteres. Tiene la limitación que al apretar la tecla Enter, en la pantalla se vuelve al comienzo de renglón, sin saltar al renglón siguiente.

A 100

xxxx:0100 MOV CL, [2000]	Carga en CL el número de caracteres a tipear
xxxx:0104 MOV AH, 1	Inicializa parámetros para entrar un carácter por teclado. El programa queda esperando hasta que se tipee el carácter, el cual luego aparece por pantalla
xxxx:0106 INT 21	Llama a subrutina del DOS
xxxx:0108 MOV DL, AL	Guarda en DL el carácter dejado en AL por la subrutina anterior
xxxx:010A MOV AH, 2	Inicializa parámetros para que un carácter en DL se vea en pantalla
xxxx:010C INT 21	Llama a subrutina del DOS
xxxx:010E DEC CL	Decrementa cantidad de repeticiones a realizar
xxxx:0110 JNZ 104	Mientras Z=0, volver a 104
xxxx:0112 INT 20	Fin

## ESQUEMA DETALLADO DE INTERRUPCIONES POR HARDWARE ENMASCARABLES EN PC

Habiendo tratado en forma global las interrupciones por hardware en la sección 11 de la unidad 1, pasaremos a ver más en detalle el proceso que tiene lugar durante las mismas.

1. Segundo se estableció (figura 3.36), las líneas  $IRQ_n$  de solicitud de interrupción ("interrupt request"), salen de las interfaces y van a un chip donde, entre otros circuitos, se encuentra un "árbitro". Este, en caso de que haya varias líneas  $IRQ_n$  activadas simultáneamente, le otorga prioridad a la de menor subíndice  $n$ .
2. En correspondencia con la priorización de una línea  $IRQ_n$ , el "árbitro" activa la línea INTR, que llega al microprocesador, para ser sensada por la UC.
3. Como se especifica en el diagrama lógico de la figura 3.36, luego de terminar de ejecutar cada instrucción, como ser la  $I_n$ , la UC sensa si la línea INTR está activa o no. En caso negativo, pasa a ejecutar la instrucción siguiente,  $I_{n+1}$ .  
Si INTR está activada, la UC se fija en el valor del Interrupt flag I, contenido en el Registro de Estado (RE), bit que mediante instrucciones se lo puede poner en 1 ó 0.  
En caso de que sea  $I=1$ , no da curso a la solicitud de interrupción indicada por INTR (y comenzada por la activación de la línea  $IRQ_2$ ), pasando a ejecutar la instrucción  $I_{n+1}$ .
4. Si  $I=0$ , se aceptará la solicitud de interrupción que llegó a la UC a través de la activación de INTR, y la UC enviará al árbitro INTA (A de acknowledge = aceptación).
5. El árbitro envía hacia la UC por el bus de datos, una combinación binaria, que es la dirección de la dirección donde se encuentra la subrutina preparada exclusivamente para atender la activación de la línea  $IRQ_2$ . La dirección de la subrutina es el vector interrupción.
6. La UC ordena escribir en la pila (como en las interrupciones por software) el CS e IP de la dirección de retorno, y luego el valor de los flags contenidos en RE.
7. Con la dirección obtenida en el paso 5, la UC direcciona a la primer instrucción de la subrutina que atiende la interrupción, y pasa a ejecutar ésta y las siguientes.
8. Cuando ejecuta la instrucción IRET, de la pila la UC lee el valor del registro RE del programa interrumpido, y luego el IP y CS de la dirección de retorno. Todos estos valores se restauran en los correspondientes registros de la UCP, con lo cual se reanuda la ejecución del programa interrumpido.

El tipo de interrupción tratado se denomina "enmascarable", pues según el valor del flag I, a una interrupción se le puede dar curso o "enmascarar". Esto último significa que la interrupción se tiene que quedar esperando hasta que sea  $I = 0$ , no pudiendo ser interrumpido el programa.

Como se anticipó, el valor de I puede establecerse por software. Así, si un programa no debe ser interrumpido, empezará con la instrucción STI (Set I), que hace  $I = 1$ , y al terminar debe hacerse  $I = 0$  mediante la instrucción CLI (Clear I).

En el Debug, el flag I aparece en sus dos valores como EI (Enable I) y DI (Disable I).

Existen también interrupciones por hardware "no enmascarables" mediante el bit I, que son sensadas por el microprocesador a través de su "pata" NMI (Not Maskable Interrupt). Un caso de activación de la línea NMI tiene lugar cuando la fuente de alimentación deja de proveer tensión. En este caso la interrupción no enmascarable llamará a una subrutina preparada para tal evento.

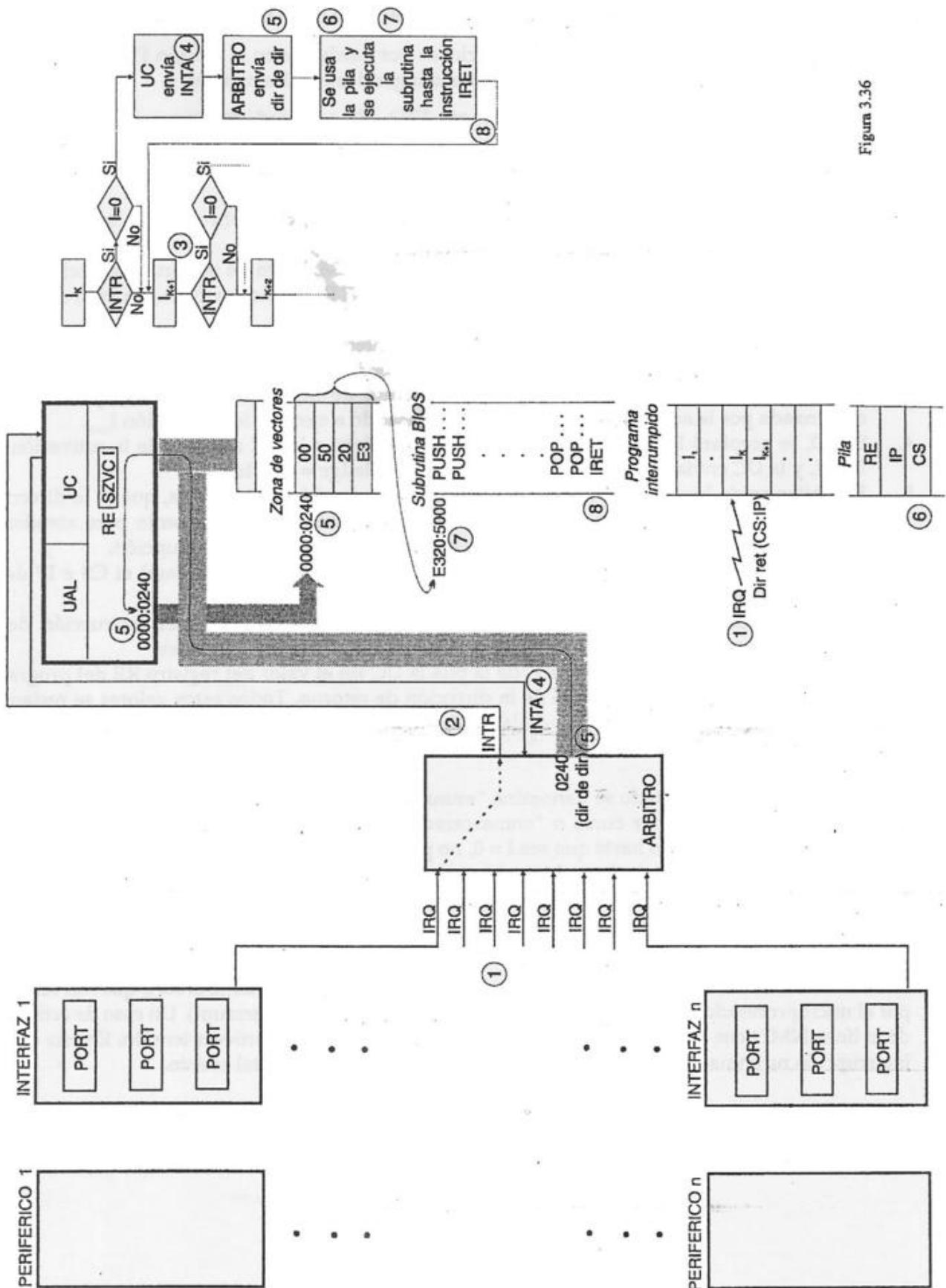


Figura 3.36

## MNEMONICOS DE LAS INSTRUCCIONES MAS USADAS

### **AAA** (ASCII Adjust for Addition):

Ordena obtener una suma en BCD a partir de dos dígitos que están en ASCII, uno de los cuales está en el registro AL, que actúa como acumulador. Esta instrucción puede hacer cambiar los flags AC y C.

Ejemplo: AL = 00110110 = 6; BL = 00111000 = 8

ADD AL, BL luego de ejecutar esta instrucción resulta: AL = 01101110 (resultado incorrecto en BCD)

AAA luego de ejecutar esta instrucción resulta: AL = 00000100 = 4 (desempaquetado) y C=1  
Esto implica que el resultado en BCD es  $14 = 6 + 8$ , siendo el 1 indicado por C.

### **AAD** (Conversión de BCD a binario antes de dividir)

Dados dos dígitos BCD desempaquetados en AH y AL, la instrucción AAD los convierte en un número binario puro contenido en AL. Esta conversión se usa antes de dividir dichos dígitos BCD por un dígito BCD desempaquetado en CX. El resultado de la división aparece desempaquetado en AL, y en AH aparece el resto. AAD puede cambiar los flags S y Z.

Ejemplo: AX = AH AL = 0000001000001000 = 0208 = 28 desempaquetado; CX = 0009

AAD luego de ejecutar esta instrucción resulta: AX = 0000000000011100 (28 en binario)

DIV CX luego de ejecutar esta instrucción resulta: AL = 00000011 = 03 (cociente)  
AH = 00000001 = 01 (resto)

### **AAM** (Conversión de binario a BCD después de multiplicar dos dígitos desempaquetados)

Ejemplo: AL = 00000110 = 6 BCD desempaquetado; BH = 00001000 = 8 BCD desempaquetado

MUL BH luego de ejecutar esta instrucción resulta AX = 0000000000110000 = 48

AAM luego de ejecutar esta instrucción resulta AX = 00000100 00001000 = 0408 desempaquetado

AAM puede cambiar los flags S y Z.

### **AAS** (ASCII Adjust for Subtraction): semejante a AAA, pero para una resta

Ejemplo: AL = 00110101 = ASCII 5 BL = 001111000 = ASCII 8

SUB AL, BL luego de ejecutar esta instrucción resulta AL = 11111101 = -3 y C=1

AAS luego de ejecutar esta instrucción resulta AL = 00000011 = 03 desempaquetado, y C=1

Si dos números de varios dígitos son restados, el flag C es tenido en cuenta para los dígitos siguientes, usando la instrucción SBB.

AAS opera sólo sobre el registro AL. Puede cambiar los flags AC y C.

**ADC** Instrucción exemplificada en el ejercicio 2 para sumar dos operandos más el carry existente.

**ADD** Instrucción exemplificada en el ejercicio 1 para sumar dos operandos.

**AND** Instrucción exemplificada en el ejercicio 25 para realizar la operación lógica AND.

### **CALL** Ordena transferir la ejecución a una subrutina.

Un *near* o intrasegmento CALL llama una subrutina que está en el mismo CS que el de la instrucción CALL, como se exemplifica en el ejercicio 38

Un *far* o intersegmento CALL llama a una subrutina que está en un CS distinto que el de la instrucción CALL. En este caso primero se decrementa SP en dos y se copia en la pila el valor de CS. Luego otra vez se decrementa SP en dos, y se copia en la pila el IP con el offset de la instrucción que sigue a CALL. Despues se caarga CS con el valor de CS de la subrutina, y se carga IP con el offset de la primera instrucción de la subrutina.

**CBW** Instrucción exemplificada en el ejercicio 37, para duplicar el tamaño del registro AL

**CLC** (Clear Carry) Ordena poner en cero el flag C

**CLD** (Clear Direction flag)

Ordena poner en cero el flag D. De esta forma los registros SI y DI pueden autoincrementarse automáticamente (en uno o dos, según sea) en las instrucciones para operar sobre strings (como MOVS o CMPS; ejemplo en instrucción CMPB)

**CLI** (Clear Interrupt flag). Ordena poner a cero el flag I, para que la UCP no responda (enmascarar) los pedidos de interrupción que activan las líneas IRQ.

**CMC** (Complement Carry flag). Ordena poner el flag C al valor contrario al que tenía antes de ejecutar esta instrucción.

**CMP** Instrucción exemplificada en el ejercicio 7, para comparar dos operandos.

Se trata de una instrucción de resta, para que se enciendan los flags según el resultado (que no se asigna a ningún destino). Siempre es seguida por una instrucción de salto condicionado

**CMPSB**

Ordena compara un byte en un string origen (source) con un byte en otro string (destination), siendo que un string es una cadena de caracteres del mismo tipo (como ser caracteres ASCII). La comparación se realiza tomando el byte apuntado por SI, y restándolo del byte apuntado por DI, a fin de que se enciendan los flags. Los operandos no son afectados por la resta. Después de la comparación, SI y DI pueden ser incrementados automáticamente a fin de apuntar a los siguientes elementos correspondientes de los strings. Para ello previamente el flag D (de dirección) debe ser llevado a cero con la instrucción CLD. El string apuntado por SI debe estar en el segmento de datos, y el apuntado por DI, en el segmento extra.

La instrucción CMPSB puede ser usado con REPE para comparar todos los elementos sucesivos de un string.

Ejemplo:

MOV SI, 3000	SI apunta al comienzo del primer string
MOV DI, 1000	DI apunta al comienzo del otro string
CLD	Pone el flag D en cero para indicar autoincrementar a DI y SI
MOV CX, 50	Pone en CX el número de elementos del string
REPE CMPSB	Repite la comparación de los bytes correspondientes de los strings, hasta completar 50 elementos, o hasta que los bytes comparados sean distintos

Si en lugar de usar CLD se hubiera usado STD (Set D; poner a uno a D), los punteros SI y DI se hubieran decrementado.

**CMPSW**

Semejante a CMPB, pero para operar elementos de strings que tengan 2 bytes. También, agregando CLD o STD, los punteros SI y DI pueden autoincrementarse o decrementarse en dos, respectivamente.

**CWD** (Convert signed Word to signed Doble word)

Extiende el signo de AX a todos los bits de DX. Semejante a CBW

**DAA** (Decimal Adjust AL after BCD Addition). Luego de sumar dos números BCD y dejar el resultado en AL, convierte dicho resultado en dos dígitos BCD. Instrucción exemplificada en el ejercicio 30

**DAS** (Decimal Adjust AL after BCD Subtraction)

Realiza correcciones semejantes a DAA si va luego de la instrucción SUB AL, ....

**DEC** Decrementa uno el operando. Instrucción exemplificada en el ejercicio 3

**DIV** (Divide magnitudes)

Divide una magnitud de 16 bits por otra de 8 bits especificada en la instrucción, ó una de 32 q bits por otra de 16 bits especificada en la instrucción. Instrucción exemplificada en el ejercicio 23

**IDIV** (Divide integers)

Cuando el dividendo tiene 32 bits, su parte más significativa tiene que estar en DX, y la menos significativa en AX. El divisor puede estar en cualquier otro registro de 16 bits. El cociente resultante está en AX, y DX contiene el resto, del mismo signo que el dividendo.

**IMUL** Instrucción para multiplicar números signados, exemplificada en el ejercicio 36

**IN** (Input). Ordena copiar el contenido de un port en AL ó AX, según que el port sea de 8 ó 16 bits, respectivamente. Ejemplo: IN AL, 3DB0; IN AX, 4CDA (Ver aplicaciones en la Unidad 2)

**INC** Instrucción exemplificada en el ejercicio 7

**INT** Instrucción exemplificada en el ejercicio 39

**IRET** Instrucción exemplificada en el ejercicio 39

**JA = JNBE** Instrucción exemplificada en el ejercicio 8

**JAE = JNB = JNC** (Jump if Above or Equal = Jump if Not Below = Jump if Not Carry)

**JB = JC = JNAE** Instrucción exemplificada en el ejercicio 10

**JBE = JNA** Instrucción exemplificada en el ejercicio 11

**JCXZ** (Jump if the CX register is Zero)

**JE = JZ** Instrucción exemplificada en el ejercicio 5

**JG = JNLE** Instrucción exemplificada en el ejercicio 9

**JGE = JNL** (Jump if Greater or Equal = Jump Not Less)

**JL = JNGE** Instrucción exemplificada en el ejercicio 10

**JLE = JNG** (Jump if Less or Equal = Jump Not Greater)

**JMP** Instrucción exemplificada en el ejercicio 4

**JNE = JNZ** Instrucción exemplificada en el ejercicio 3

**JNO** (Jump if Not Overflow)

**JNP = JPO** (Saltar si no hay paridad par = saltar si hay paridad impar)

**JNS** (Jump if not signed = jump if positive)

**JO** Instrucción exemplificada en el ejercicio 4

**JPE = JP** Instrucción exemplificada en el ejercicio 13

**JS** (Jump if Sign is negative (S=1))

**LEA** Transfiere a un registro el valor de la dirección del operando fuente.  
Instrucción exemplificada más adelante, en el assembler para TASM y MASM

**LODSB** (Load String Byte into AL)

Ordena copiar en AL un byte de una locación de un string apuntada por SI. Mediante el flag D=0, el registro SI se incrementa automáticamente para apuntar al siguiente elemento del string. Si D=1, SI se decrementa automáticamente.

**LODSW** (Load String Word into AX)

Ordena copiar en AX dos bytes consecutivos de un string, apuntados por SI.

**LOOP**

Ordena repetir una secuencia de instrucciones un número de veces dado por el contenido de CX. Cada vez que LOOP se ejecuta, CX es automáticamente decrementada en 1. Mientras CX≠0 salta a ejecutar la instrucción

ejemplificada por la dirección que acompaña a LOOP. (Esta dirección no puede superar el rango de +127 a -128 bytes desde la instrucción que sigue a LOOP)

#### **LOOPE = LOOPZ**

Ordena repetir la secuencia mientras sea CX ≠ 0 y Z = 1

#### **LOOPNE = LOOPNZ**

Ordena repetir la secuencia mientras sea CX ≠ 0 y Z = 1

**MOV** Transfiere el valor del operando fuente hacia el operando destino. Instrucción exemplificada en el ejercicio 1

#### **MOVSB (Move String Byte)**

Ordena transferir un byte de un string apuntado por SI en el segmento de datos, hacia una posición en el segmento extra apuntado por DI. El número de elementos a ser transferidos es puesto en CX.

Si flag D=0, entonces DI y SI serán automáticamente incrementados en uno.

Si flag D=1, entonces DI y SI serán automáticamente decrementados en uno.

**MOVSW** Semejante a MOVSB, pero copia 2 bytes.

**MUL** Instrucción exemplificada en el ejercicio 29

**NEG** Genera el complemento a dos del número contenido en un registro

**NOP** Su ejecución emplea hasta 3 ciclos reloj, y luego incrementa el IP para apuntar la próxima instrucción. Se usa para aumentar el retardo de un loop que genera retardo.

**NOT** Ordena invertir cada bit dell operando.

**OR** Realiza una operación OR. Instrucción exemplificada en el ejercicio 25

**OUT xxxx** Transfiere el contenido del registro AL (ó AX) hacia el port de dirección xxxx

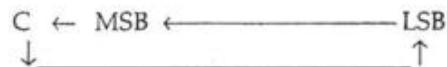
**POP** Copia 2 bytes desde la cima de la pila (apuntadada por SP), y los transfiere al registro o destino indicado en la instrucción. Luego se incrementa en dos el SP. Ejemplificada en el ejercicio 38

**POPF** Instrucción exemplificada en el ejercicio 39. Ningún flag es afectado por esta instrucción.

**PUSH** Primero ordena decrementar en dos al SP, y luego salva 2 bytes (que típicamente estaban en un registro) en la cima de la pila. Ejemplificada en el ejercicio 38.

**PUSHF** Instrucción exemplificada en el ejercicio 39

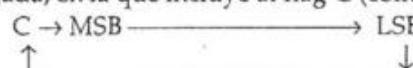
**RCL** Ordena la rotación a izquierda dibujada un cierto número de veces, en la que se incluye al flag C (comparar con ROL)



Ejemplos: RCL BX, 1

MOV CL,4  
RCL BX, CL

**RCR** Ordena la rotación a derecha dibujada, en la que incluye al flag C (comparar con ROR)



#### **REPZ = REPE**

Ordena repetir una instrucción para stringsd hasta que se de una cierta condición. A menudo usada con CMPS, para comparar dos cadenas hasta que CX = 0 ó hasta que los elementos del string no sean iguales.

Ejemplo: REPE CMPSB compara bytes de cadenas hasta el fin de la cadena, o hasta que bytes de las cadenas no sean iguales.

**RET**      Ordena saltar a la instrucción que sigue a la última CALL ejecutada.  
Instrucción exemplificada en el ejercicio 38

**ROL**      (Rotate Left)  
Ordena la rotación a izquierda dibuja, un número de veces



**ROR**      (Rotate Right)  
Ordena la rotación a derecha dibujada, un cierto número de veces



**SHL = SAL**    (Shift Left)

C ← MSB ←                LSB ← 0

Como indica el dibujo, ordena desplazar hacia la izquierda cada bit del operando un cierto número de posiciones (indicado por CL<sup>1</sup>). Conforme un bit es desplazado de la posición menos significativa (LSB), un cero es puesto en dicha posición, a la par que el valor del bit que está en la posición más significativa (MSB) es puesto como valor del flag C. En el caso de varios desplazamientos sucesivos, el flag C toma el valor del bit más recientemente desplazado desde la posición más significativa.

Esta operación puede usarse para multiplicar un número natural por dos, o una potencia de dos.

**SAR**

MSB → MSB →                LSB → C

Esta instrucción puede usarse para dividir un número signado por dos, o una potencia de dos, dado que el MSB es copiado nuevamente como MSB, conforme el que era bit de signo es desplazado a la derecha.

**SBB**      (Substract with borrow). Como ADC, pero para la resta.

**SCASB**      (Scan a String Byte)

Ordena comparar un byte en AL con un byte de un string apuntado por DI en ES.

Si flag D=0 entonces DI será incrementado luego de SCASB; y si D=1 entonces DI puede ser decrementado luego de SCASB

**SCASW**      (Scan a String Word)

Ordena comparar AX con 2 bytes consecutivos de un string apuntados por DI en ES. El flag D puede usarse para incrementar o decrementar DI en dos.

**SHR**      Instrucción exemplificada en el ejercicio 34, para números naturales.

**STC**      (Set Carry flag): ordena hacer C=1

**STD**      (Set Direction flag): ordena hacer 1 el flag D, de dirección.

**STI**      (Set Interrupt flag):

Ordena hacer 1 el flag I, para habilitar las interrupciones por hardware enmascarables.

**STOSB**      (Store Byte in String)

Ordena almacenar una copia de un byte desde AL en una locación de memoria apuntada por DI en el ES, reemplazando un byte de un string. Luego DI puede ser automáticamente incrementado o decrementado según que el flag D valga 0 ó 1, respectivamente.

**STOSW**      (Store Word in String)

Ordena almacenar una copia de AX en dos bytes sucesivos apuntados por DI en el ES, reemplazando 2 bytes de un string. DI puede autoincrementarse o decrementarse en 2 mediante el flag D.

<sup>1</sup> Si el número de desplazamiento es uno, ello puede especificarse colocando un uno a la derecha de la coma en la instrucción.

**SUB** Ordena restar dos operandos. Instrucción exemplificada en el ejercicio 1

**TEST**

Realiza una operación AND para activar los flags, sin asignar resultados. Semejante CMP respecto de SUB.

**WAIT**

Ordena que el procesador pase al estado IDLE, en el cual no procesa hasta que se active la entrada de interrupción INTR o NMI, o hasta que la entrada TEST resulte baja. Se usa para sincronizar al procesador con hardware exterior.

**XCHG** (Exchange source and destination)

Ordena intercambiar el contenido de un registro con el otro, o el contenido de un registro con contenidos de locaciones de memoria.

**XLATB** (Translate Byte in AL)

Translada un byte de un código a otro código. Reemplaza un byte en AL con un byte punteado por BX, en una tabla de memoria.

**XOR**

Ordena realizar la operación X-OR entre cada bit de un operando fuente con cada bit del operando destino, y el resultado colocarlo en el destino. Si se hace XOR AX, AX, el registro AX se pone en cero.

# TÉCNICAS DIGITALES II

Instrucción	Condición probada	Comentario
JAE	C = 0	Brinca si está por arriba
JBE	C = 1 o Z = 1	Brinca si está por arriba o es igual a
JC	C = 1	Brinca si está por abajo
JGE	Z = 1	Brinca si es igual o menor
JG	Z = 0 y S = 0	Brinca si es mayor que
JGE	S = 0	Brinca si es mayor que, o igual a
JL	S ≠ 0	Brinca si es menor que
JLE	Z ≠ 1 o S ≠ C	Brinca si es menor que, o igual a
JNC	C = 0	Brinca si no hay acarreo
JNE o JNZ	Z = 0	Brinca si no es igual a, o si no es cero
JNO	O = 0	Brinca si no hay sobreflujo
JNS	S = 0	Brinca si no hay signo
JNP/JPO	P = 0	Brinca si no hay paridad o con paridad impar
JO	O ≠ 1	Brinca si hay sobreflujo
JP/JPE	P ≠ 1	Brinca si hay paridad o paridad par
JS	S ≠ 1	Brinca si hay signo
JCXZ	CX = 0	Brincar si CX = 0

Números sin signo

255	FFH
254	FEH

132	84H
131	83H
130	82H
129	81H
128	80H

4	04H
3	03H
2	02H
1	01H
0	00H

Números con signo

+127	7FH
+126	7EH

+2	02H
+1	01H
+0	00H
-1	FFH
-2	FEH

+124	84H
+125	83H
+126	82H
-127	81H
-128	80H