

ASSEMBLER DESDE CERO

e

INTERRUPCIONES

MARIO CARLOS GINZBURG
INGENIERO ELECTRONICO (UBA)

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

INDICE

UTILIZACION DEL LENGUAJE ASSEMBLER	1
REGISTROS DE LA UCP A UTILIZAR	1
EJERCICIO 1 introductory	2
EJERCICIO 2: uso de una instrucción de salto condicional (conditional jump) JNZ	4
EJERCICIO 3: ejercicio 2 mejorado usando JO (Jump if overflow), constantes, JMP y etiquetas	7
EJERCICIO 4: perfeccionamiento del ejercicio 2 teniendo en cuenta que M & N pueden valer cero	9
EJERCICIO 5: cómo cambiaría el ejercicio 3 si se usara JZ en vez de JNZ	9
GENERALIZACION: instrucciones de salto condicional:	10 a 12
EJERCICIO 6: suma de 2 números de 32 bits mediante la UAL de 16 bits (3 soluciones)	12
EJERCICIO 7: lectura y escritura de una sucesión de datos consecutivos en memoria (lista o vector)	14
EJERCICIO 8: sumar los elementos de una lista (vector)	16
EJERCICIO 9: búsqueda en lista usando la instrucción comparación (resta sin asignación de resultado)	17
EJERCICIO 10 basado en el ejercicio 8 y en un paso del 9	20
EJERCICIO 11: otra variante del ejercicio 8	20
EJERCICIO 12: encontrar el mayor de una lista de números naturales usando JA (Jump if above)	21
EJERCICIO 13: diagrama lógico con rombos consecutivos y ejecución del programa con el Debug	23
FORMA DE COPIAR UNA PANTALLA DEL DEBUG EN EL WORD	28
EJERCICIO 14 ordenar alfabéticamente 2 listas cuya cantidad de caracteres no es un dato directo	28
EJERCICIO 15 validación de caracteres de una lista cuya cantidad no es un dato directo	30
EJERCICIO 16 intercambio de elementos entre dos listas, y variante con ahorro de un puntero	32
EJERCICIO 17 redondeo de números con parte entera (de longitud no limitada) y fraccionaria (un dígito)	32
EJERCICIO 18 técnicas para repetir o no la ejecución de secuencias mediante indicadores de pasada	34
EJERCICIOS 19 A 25: ejercicios varios con cambios internos en los elementos de vectores	34 a 39
EJERCICIO 25 a 28: ejercicios con traducciones de sentencias IF y FOR a bajo nivel	39 a 40
CONSIDERACIONES GENERALES PARA PROGRAMAR EN ASSEMBLER	41
EJERCICIOS PARA REALIZAR	44
EJERCICIOS 30 y 31: uso de JBE (Jump if below or equal)	45
EJERCICIO 32: uso de la instrucción JPE (Jump if parity is even = Saltar si la paridad es par)	46
EJERCICIO 34 ordenamiento de una lista por burbujeo	46
EJERCICIOS VARIOS CON OPERACIONES ARITMÉTICAS	47 a 51
DIRECCIONES EFECTIVAS Y REGISTROS DE SEGMENTO EN MODO REAL	52
LLAMADO A SUBRUTINAS	54
INTERRUPCIONES POR SOFTWARE	72
EJERCICIOS BASICOS CON INSTRUCCIONES INT 21	77
INTERRUPCIONES POR HARDWARE	80
EJERCICIO INTEGRADOR DE SUBRUTINAS E INTERRUPCIONES	83
MANEJO DEL PORT PARALELO	85
MNEMONICOS DE LAS INSTRUCCIONES MAS USADAS	88

UTILIZACION DEL LENGUAJE ASSEMBLER

Conviene aclarar que se suele hablar de programa “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 Assembler fue el primer lenguaje que usó símbolos alfábéticos. Permitió así escribir programas con letras desde el teclado de un computador (salida código ASCII), lo cual suponía la existencia de un programa traductor ensamblador, para pasar los símbolos codificados 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, por razones de compatibilidad, instrucciones en Assembler del 80286.

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 AMD, 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 redunda 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, que permite manejar recursos en general no accesibles a lenguajes de alto nivel.

El lenguaje Assembler, por otra parte, es una herramienta imprescindible para dominar a fondo el funcionamiento de un computador (*visualizar movimientos internos que deben tener lugar*), y para 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).

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 7.

BX es el único acumulador que también puede utilizarse como registro puntero (ejemplificado en el ejercicio 13).

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.*

El siguiente ejercicio básico, analizado y realizado directamente en código de máquina, sin usar Assembler en la Unidad 1 de esta obra, muestra los pasos necesarios que se deben tener presente en los restantes ejercicios que se plantearán. Asimismo conviene aclarar que un desarrollo en Assembler puede realizarse en una infinidad de formas. Las que se presentan en este texto sólo pretenden ser una guía, y no un modelo que limite potencialidades creativas.

EJERCICIO 1

En un lenguaje de alto nivel se tiene la sentencia $R = P + P - Q$ para operar números enteros (variable "Integers"). Codificarla en lenguaje Assembler, lo cual también servirá para poner en evidencia cómo se concreta a nivel de hardware y cómo podría traducirla un programa traductor tipo Compilador cuando realiza la traducción en seudo assembler.

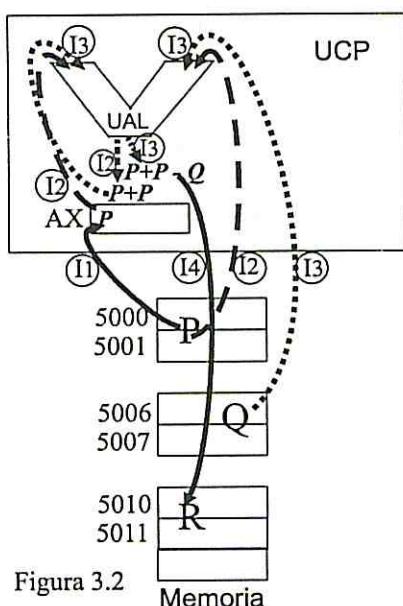


Figura 3.2
Memoria

1a) Cualquier secuencia de instrucciones a desarrollar operará números (valores de las variables) que estarán en memoria principal, por lo que primero siempre deben elegirse direcciones de memoria arbitrarias donde se encuentran los valores de las variables (R, P y Q). Como se verá esta elección arbitraria se utiliza luego en la codificación de las instrucciones.

En lo que sigue, supondremos que las variables tipo "Integers" ocupan 2 células consecutivas de memoria, salvo que se indique otra cosa. Asignaremos (fig. 3.2) a R, P y Q las direcciones 5010/11, 5000/1 y 5006/7 respectivamente.

No interesa en esta etapa el valor de P ó Q, pues los programas se desarrollan para valores cualesquiera de las variables, dentro de ciertos límites.

1b) La figura 3.2 también sirve para plantear la secuencia de los movimientos y operaciones que deben ordenar las instrucciones a codificar, para ir llevando a cabo en orden los pasos necesarios para concretar la suma algebraica que ordena la sentencia $R = P + P - Q$ teniendo presente que dichos movimientos y operaciones son los que puede efectuar típicamente un computador. Estos últimos, como se planteó en la Unidad 1 de esta obra, pueden comprenderse conceptualmente en relación con lo que puede hacer una simple calculadora de bolsillo. Asimilaremos su visor-acumulador al registro AX del procesador arriba definido, y sus circuitos para sumar/restar a la Unidad Aritmético Lógica (UAL).

En un computador los números que se suman provienen directamente de su memoria principal en vez de ser generados desde un teclado como en la calculadora, y los resultados que aparecen en AX pueden ser guardados en dicha memoria. Esto lo indican las flechas dibujadas en la figura 3.2.

Si en una calculadora quisieramos hacer $P + P - Q$ los pasos en esencia serían:

- 1) Llevar el valor de P al visor.
- 2) Sumar a P el valor de P, y el resultado de $P + P$ queda en el visor.
- 3) Restar al resultado del paso 2 el valor de Q, y el resultado final de $P + P - Q$ queda en el visor.

En un computador los pasos equivalentes serían ordenados por 3 instrucciones que designaremos I_1 , I_2 e I_3 :

I_1 : ordena llevar hacia AX (registro de 16 bits) una copia del valor (P) que está en memoria en las direcciones 5000/1. I_2 : ordena sumar en la UAL al valor (P) presente en AX una copia del valor (P) que está en memoria en las direcciones 5000/1 y el resultado ($P + P$) dejarlo en AX.

I_3 : ordena restar en la UAL al valor ($P + P$) presente en AX una copia del valor (Q) que está en memoria en las direcciones 5006/7 y el resultado ($P + P - Q$) dejarlo en AX.

Puesto que $R = P + P - Q$ ordena guardar el resultado de $P + P - Q$ donde está la variable R, es necesaria una instrucción I_4 que ordena llevar hacia las direcciones 5010/11 (donde está R) una copia del valor ($P + P - Q$) que está en AX. Obsérvese que las variables R, P y Q definidas en alto nivel pasan a ser ahora contenidos de direcciones de memoria.

Estos movimientos y operaciones se indican en la figura 3.2 en relación con lo que sucedería si las instrucciones correspondientes se ejecutarían, sin que aparezcan por simplicidad los movimientos para pedir esas instrucciones. En dicha figura sobre el registro AX aparecen los nuevos contenidos ($P + P$ y $P + P - Q$) que AX va tomando.

2) Habiendo elegido las direcciones de las variables en memoria, el registro AX como acumulador, y planificado todos los pasos (futuras instrucciones) necesarios, se debe seguir con la realización de un **diagrama lógico** donde se codifique lo que ordena cada instrucción, con vistas a ayudar a la codificación final de las instrucciones en lenguaje Assembler.

Así (fig. 3.3), lo que ordena I_1 se ha indicado $AX \leftarrow [5000/1]$ pudiéndose también escribir $AX \leftarrow M5000/1$, con M signi-

ficando memoria, pero la simbología [5000/1] prepara el terreno para codificar esta instrucción en Assembler (ver abajo).

Los corchetes siempre simbolizan memoria. Esto es lo que guardan una o más celdas sucesivas de memoria. Si bien esta vez entre corchetes se indican direcciones de 2 celdas, en Assembler siempre se codifica la dirección de la primera celda.

A la derecha de cada paso es muy conveniente ir controlando los resultados parciales esperados, como AX = P.

No es necesario en los pasos 1, 2 y 3 indicar el valor concreto de las variables P y Q, pues en general un programa se desarrolla para una amplia gama de valores de sus variables. Pero puede ayudar a depurarlo si antes del paso siguiente se hace una prueba representativa “de escritorio” con números, para verificar que una secuencia funcionará correctamente.

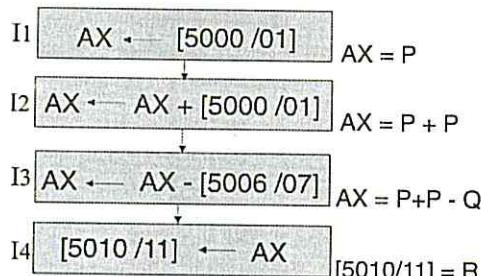


Figura 3.3

3) Mediante el comando A del Debug¹ 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. Se ha elegido ésta en 0200, y en ejercicios siguientes usaremos 0100. Escribiremos a continuación en Assembler, para el Debug, las 4 instrucciones que se corresponden una a una con I₁ a I₄ de la figura 3.3. Al tipear A 0200 ↴ aparecerá xxxx : 0200, y el Debug esperará que se tipée MOV AX, [5000] ↴, luego de lo cual aparecerá xxxx : 0203 a la espera de otra instrucción, y así sucesivamente. Si se tipea mal una instrucción, aparece una indicación de error para repetir su escritura. Entre corchetes sólo se escribe la primer dirección de cada variable. Las instrucciones donde aparece AX (ó BX, CX, DX) implican que

los datos a operar son de 16 bits. AX es un “recipiente” que recibe necesariamente el contenido de 2 celdas sucesivas.

Cada instrucción ordena una operación (MOV; ADD, etc.), seguida del destino del resultado con una coma final. Luego continúa otro campo con el lugar de origen de un dato a operar. El campo de comentarios de la derecha es optativo.

- A 0200 ↴
- xxxx:0200 MOV AX, [5000] ↴² 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 ha querido indicar 4 símbolos hexadecimales a tratar, cuyo valor puede ser distinto en cada PC.

Cuando se tipa MOV AX, [5000], a partir de la dirección 0200 quedarán en celdas sucesivas de memoria codificadas en ASCII cada uno de los caracteres tipeados. Al tipear ↴ el programa Ensamblador del Debug traducirá MOV AX, [5000] en la instrucción de máquina 10100001 00000000 00000101 = A10050h que se escribirá en 3 celdas a partir de dicha dirección 0200, desapareciendo así todos los caracteres en ASCII antes tipeados desde 0200, y la dirección que aparecerá para escribir la segunda instrucción será 0203. El código A1 y el código ASCII se tratan en la Unidad 1 de esta obra. Al terminar de tipear las 4 instrucciones, las mismas quedarán traducidas en código de máquina listas para ser ejecutadas. Si se quiere ver estos códigos en hexa, se debe tipear el comando U 0200 ↴.

4) Como se exemplifica luego del ejercicio 13 y al igual que en los ejemplos de la Unidad 1 de esta obra, antes de ejecutar con el Debug esta secuencia de instrucciones, se deben escribir en memoria con el comando E los datos a procesar (valores elegidos para las variables P y Q). De seguido, con el comando RIP se debe dar el valor 0200 al IP; con el comando R se debe verificar que todo está en orden; y con 4 comandos T (uno por instrucción) se ejecutará la secuencia.

OTRA FORMA DE REALIZAR EL EJERCICIO 1

En un computador R = P + P - Q se puede hacer de múltiples maneras, como ser (figura 3.4):

- 1) Llevar hacia AX una copia del valor (P) que está en memoria en las direcciones 5000/1.
- 2) Llevar hacia BX una copia del valor (Q) que está en memoria en las direcciones 5006/7.
- 3) Realizar la suma AX ← AX + AX, o sea sumarle a AX su propio valor, y el resultado P + P dejarlo en AX.
- 4) Realizar AX ← AX - BX, o sea restar a AX el valor de BX, y el resultado P + P - Q dejarlo en AX.
- 5) Llevar hacia las direcciones 5010/11 (donde está R) una copia del valor (P + P - Q) que está en AX.

¹ Este programa Debug está presente en la mayoría de las PC, vinculado al DOS. Ya fue usado por su simplicidad didáctica en la Unidad 1 de esta obra para codificar en memoria datos e instrucciones en código de máquina y luego ejecutarlas una a una. Ahora lo usaremos también para programar directamente en Assembler, pues no requiere directivas para el traductor Ensamblador, como otros Debug más potentes (como el MASM o el TASM) que permiten entre otras cosas usar nombres de variables en lugar de direcciones numéricas, acercando al Assembler a lenguajes de alto nivel. Sin necesidad de usar ni pensar en esas directivas, el alumno desde el comienzo centra su esfuerzo en construir directamente secuencias de instrucciones en Assembler.

² Si una instrucción tiene un número entre corchetes se dice que está en “modo directo”, o en modo de direccionamiento directo, dado que directamente dicho número es la dirección para localizar en memoria el dato a procesar.

Las 5 instrucciones correspondientes en Assembler serían:

```
MOV AX, [5000]      (AX = P)
MOV BX, [5006]      (BX = Q)
ADD AX, AX          (AX = P + P)
SUB AX, BX          (AX = P + P - Q)
MOV [5010], AX
```

A la derecha de cada instrucción se indica qué ocurre cuando ella se ejecuta.
En lugar de utilizar el registro AX como acumulador, podríamos haber usado BX, CX o DX; sólo hubiesen cambiado los códigos de máquina de las instrucciones.
Por ejemplo:

```
MOV BX, [5000]
MOV AX, [5006]
ADD BX, BX
SUB BX, AX
MOV [5010], BX
```

En ninguna de las tres secuencias de instrucciones anteriores se planteó la posibilidad que por los valores de los datos puestos en juego, un resultado en el registro AX de 16 bits supere alguno de los máximos +32767 ó -32768 que AX puede contener (determinados en la Unidad 1 de esta obra), situación considerada en el ejercicio 3. En la codificación de dichas secuencias se pudo verificar que la elección arbitraria de las direcciones de las variables no interesa, dado que al formar parte esas direcciones de las instrucciones, éstas permitían localizar las variables en las direcciones elegidas.

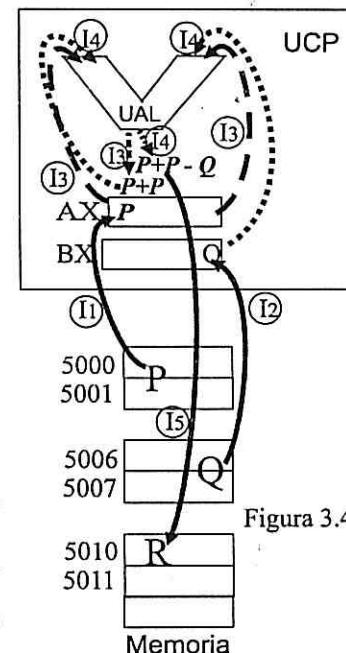


Figura 3.4

EJERCICIO 2 : uso de una instrucción de salto condicional (conditional jump)

En un lenguaje de alto nivel se escribió la sentencia $R = M \times N$ para multiplicar números enteros positivos. Codificarla en lenguaje Assembler, suponiendo un microprocesador básico que no tiene instrucciones para multiplicar, y para plantear la correspondencia entre las estructuras tipo "FOR i = 1 to i = n ..." y su equivalencia en Assembler.

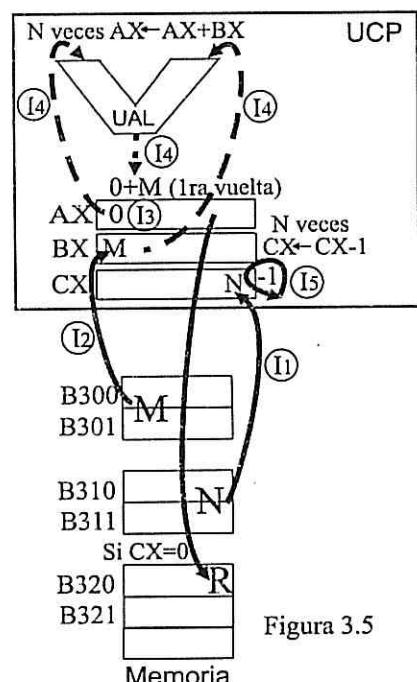


Figura 3.5

Este ejercicio es esencial, pues empezaremos a usar las instrucciones de salto condicional, en este caso para repetir la ejecución de una secuencia, estructura que será usada en muchos ejercicios para repetir n veces un procedimiento.

Emplearemos una instrucción de salto condicional para realizar el producto $R = M \times N$, repitiendo N veces una operación de suma.

Por definición de producto es $R = M \times N = M + M + \dots + M$ expresión con N sumandos iguales a M y N - 1 sumas.

Si bien puede partirse de esta expresión, a los fines didácticos haremos $R = M \times N = 0 + M + M + \dots + M$; ó sea mediante N sumas con sumando M, en vez de N - 1 sumas.

Si $N = 4$, es $M \times 4 = 0 + M + M + M + M$ ($N = 4$ sumas con el sumando M)

A fin de que resulte una secuencia sencilla, supondremos: que M y N son números enteros positivos de dos bytes, que N es distinto de cero, y que es $R \leq 32767$. En el ejercicio 3 se trata cómo detectar si se supera este máximo, y en el ejercicio 4 la detección de si M ó N valen cero.

1) Asignaremos (figura 3.5) a la variable M las direcciones de memoria B300 y B301; a la variable N las direcciones B310 y B311; y se han reservado B320 y B321 para el resultado R.

Para evidenciar la necesidad de una instrucción de salto, codificaremos en assembler $R = 0 + M + M + \dots$ generalizando lo aprendido en el ejercicio 1. La figura 3.5, pensada para el ejercicio 2, muestra el paso para llevar una copia del valor de M al registro BX (podía haber sido AX, CX, o DX); y el

paso para inicializar en cero el registro AX usado como acumulador acorde con el cero presente en $R = 0 + M + M + \dots$ A fin de realizar las N sumas necesarias, un sumando es aportado por AX, y el otro por BX, siendo que el resultado "pisa" el anterior valor existente en el acumulador AX, como en una calculadora. En la primera suma se hace $0 + M$. Una vez realizadas las N sumas en AX quedará el resultado de $M \times N$, el cual debe ir a R como indica la flecha. Con las instrucciones definidas en el ejercicio 1, una secuencia para realizar N sumas repetidas sería:

	MOV BX, [B300]	Llevar a BX una copia de M que está en B300/1
	SUB AX, AX	Acumulador AX en cero, restándolo de si mismo
N veces	ADD AX, BX ¹	Hacer AX+ BX y dejar el <u>resultado</u> en AX
	ADD AX, BX	Idem
ADD	ADD AX, BX	Idem
.....
	ADD AX, BX	En AX está acumulado el resultado
	MOV [B320], AX	Se asigna el resultado a R, que está en B320/1

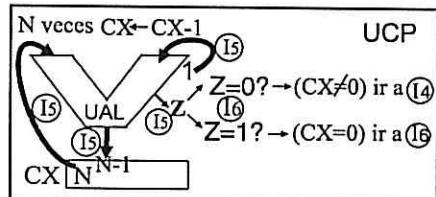


Figura 3.5.a

También se podría haber hecho la secuencia: MOV AX, [B300], seguida por ADD AX, [B300] repetida N – 1 veces, y por último MOV [B300], AX, pero sería más lenta de ejecutar, pues en cada suma hay que acceder a memoria a buscar el dato. La única instrucción nueva es SUB AX, AX usada para llevar el registro AX a cero (lo mismo puede hacerse para limpiar cualquier registro). También existen otras formas de hacerlo.

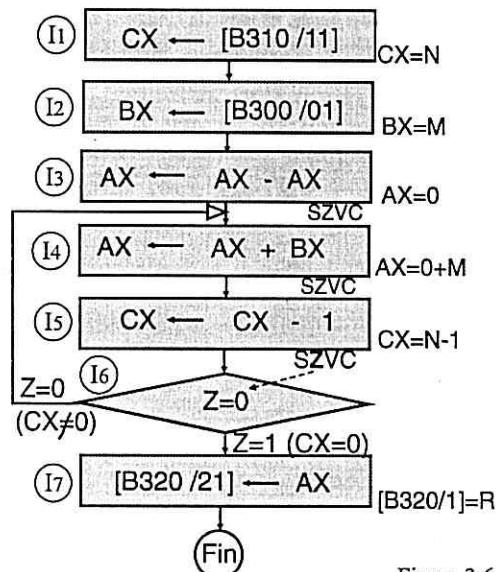


Figura 3.6

Como aparece en el programa escrito más abajo, la instrucción de salto JNZ (jump if not zero) evita escribir N veces ADD AX, BX, permitiendo escribirla una sola vez. Conforme se discutirá en la etapa 2), cuando se ejecute este programa, JNZ posibilitará repetir N veces la ejecución de ADD AX, BX escrita una sola vez, por estar ésta dentro de un “lazo” o secuencia corta de instrucciones que también se ejecutará repetida N veces.

- 2) En la etapa indicada 1) se asignaron direcciones a las variables y se plantearon todos los movimientos y acciones que deberán suceder, por lo que ahora se construirá el diagrama lógico de la figura 3.6. En general éste se puede dividir en las siguientes fases:
- 2a) Elección e inicialización de registros: para preparar el control del número N de veces que se repetirá la ejecución del lazo citado, en un primer paso (I1) se llevará a un registro una copia del valor de N, que está en B310/11. Es usual usar con este fin el registro CX al cual se le restará uno cada vez que se ejecute dicho lazo. El paso siguiente (I2) es inicializar BX (podría haber sido DX) con el valor de M a fin de no acceder a memoria en la N sumas. Luego (I3) el acumulador elegido (AX) se inicializa con el valor 0, restando al valor cualquiera que pueda tener AX ese mismo valor, a fin de generar el cero de $0 + M + M + \dots$.
- En general el orden de la inicialización no es crucial.

2b) Procedimiento: en este caso (I4) es la operación de sumar cada vez al registro AX el valor de M, dejando en AX el resultado, operación que se repetirá N veces, y que constituye el cuarto paso.

2c) Control: como hay un lazo que se repetirá N veces, al registro CX (que contiene el valor N) se le debe restar uno (I5) luego de cada vez que se ejecuta la instrucción de suma. Cuando sea CX = 0 el lazo se habrá ejecutado N veces.

Mientras esto no ocurra, (mientras CX ≠ 0), se deberán seguir ejecutando una y otra vez las instrucciones del lazo, y cada vez en la UAL se le resta uno a CX. Así CX controla las veces que se repite la ejecución de las instrucciones del lazo.

El último paso (I6) que forma parte del lazo de ejecución repetitiva es la instrucción de salto (“jump”), simbolizada por un rombo donde se pregunta indirectamente² a través del flag Z³ generado por la UAL (figura 3.5.a), si el resultado de la última operación (I5) que ella hizo (restarle uno a CX) fue cero ó distinto de cero. Si esto último se cumple, esta instrucción ordena saltar a ejecutar otra vez la suma (I4), como indica la línea que sale del rombo hacia esa instrucción. Más en detalle, si Z = 0 (que debe entenderse como si Zero “no”, o sea si el resultado de CX – 1 que quedó en CX no es cero)⁴ la instrucción de salto ordena un salto en el valor del registro Puntero de Instrucciones (IP) para que indique que la dirección de memoria de la próxima instrucción a pedir y ejecutar sea la de suma citada (I4). Con ese fin, la instrucción de salto contiene el valor que debe restarse al IP para que la ejecución retroceda a la dirección de la instrucción de suma. Cuando sea Z = 1 (o sea si Zero “si”, es decir si es cero el resultado de hacer CX – 1, con lo cual CX = 0), se tendrá la situación contraria a la condición planteada en el rombo que sea Z = 0. Entonces, la instrucción de salto ordena no saltar, sino seguir con el pedido y ejecución de la instrucción que le sigue en memoria (I7), como en una instrucción corriente.

Esta instrucción (I6) en Assembler es JNZ hhhh, siglas de Jump if Not Zero (saltar si zero no) a la dirección hhhh.

2d) Almacenamiento: por lo general los resultados alcanzados en la UCP se guardan en memoria principal, como ocurre con el paso (I7) que sigue en el diagrama al rombo, cuando se almacena en B320/1 el resultado final de la multiplicación.

¹ Cuando el dato a operar no está en memoria sino en un registro de la UCP, y el resultado va también a un registro de la UCP, el modo de direccionamiento se denomina “modo registro”.

² Como se plantea en la Unidad 1 de esta obra, la Unidad de Control (UC) no puede ver ni interpretar el contenido concreto de un registro, en este caso CX, por lo que el programador se debe valer de los flags que generó la UAL, en este caso Z, para determinar si el contenido es o no es cero. Mediante la instrucción de saltar si Z=0 se ordena a la UC que lea el valor de Z, y que se salte si Z=0.

³ El tema de los flags se desarrolla en detalle en la Unidad 1 de esta obra, en el capítulo sobre representación de la información.

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

Al lado de cada rectángulo del diagrama lógico de la figura 3.6 se indica el resultado a obtener cuando ocurra la primera ejecución de las instrucciones del lazo; y abajo los pasos que modifican el valor de los flags SZVC.

La secuencia en assembler correspondiente al diagrama de la figura 3.6 será:

A 0100

xxxx:0100	MOV CX, [B310]
xxxx:0104	MOV BX, [B300]
xxxx:0108	<u>SUB AX, AX</u>
xxxx:010A	<u>ADD AX, BX</u>
xxxx:010C	<u>DEC CX</u>
xxxx:010D	<u>JNZ 010A</u>
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 010 ^a .
Llevar a B320 y B321 de memoria, una copia del valor de AX.
Instrucción de final de secuencia ¹ .

Al igual que en el diagrama lógico de la figura 3.6, se debe repetir N veces la instrucción ADD AX, BX. Cada vez que se realiza esta suma, la instrucción siguiente DEC CX decrementa uno al contenido del registro CX (que inicialmente es N); y la instrucción JNZ sirve para determinar por un lado si el número contenido en CX alcanzó o no el valor cero.

Mientras este número en CX **no sea cero** (not zero = NZ), o sea mientras el resultado de restar uno a CX le corresponda que sea Z=0 (condición de salto de JNZ), se salta a ejecutar otra vez la instrucción ADD AX, BX de inicio de la secuencia a 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** (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*. Esta mal escribir debajo de JNZ la instrucción JZ hhhh (saltar si Z=1), pues esto ya está contemplado en JNZ.

También puede decirse que JNZ hhhh ordena saltar a la instrucción de dirección hhhh si es verdadero (cierto) que Z=0, y que si es falso que Z=0 (o sea si Z=1) ordena continuar con la instrucción que sigue a JNZ.

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

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 verificarse esa condición, y la secuencia continuará con MOV [B320], AX escrita a continuación.

Si fuera R = MxN = 3x4, puesto que N=4, se ejecutará 4 veces el lazo, realizándose en cada oportunidad las siguientes operaciones que ordenarán las instrucciones indicadas, con los resultados en hexadecimal que aparecen.

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

Esto puede verificarse paso a paso, ejecutando la secuencia anterior en Assembler, mediante el comando T del Debug. La secuencia en Assembler, como cualquier otra, puede dividirse en las fases de *inicialización, procedimiento, control de éste y almacenamiento*, de acuerdo con las líneas horizontales que aparecen en esa secuencia, planteadas en el diagrama.

Este tipo de secuencia para repetir un procedimiento será muy utilizada en ejercicios posteriores.

Resulta ilustrativo analizar qué sucede si se cambia el orden de alguna de las instrucciones de la presente secuencia. Las tres instrucciones de la inicialización pueden escribirse en cualquier orden, pues no afecta al desarrollo siguiente.

Vale la pena señalar que las instrucciones tipo MOV, como no ordenan operaciones en la UAL, no afectan el valor que tenían los flags SZVC antes de su ejecución. Sólo pueden cambiar el valor de los flags las instrucciones que usan la UAL.

Si bien luego de ejecutar SUB AX, AX resulta AX=0, y por lo tanto Z=1, como la instrucción siguiente no es una de salto que pregunte por el valor de Z, sino que es ADD AX, BX, no interesa que sea Z=1.

¿Qué pasa si se intercambian los lugares de DEC CX y ADD AX, BX ? Sucederá que como ADD AX, BX cambia el valor de los flags, y JNZ pregunta por el valor de Z luego de la operación anterior de la UAL (como indica la flecha que va del flag Z al rombo), se estaría preguntando por el valor de Z de dicha suma, en vez del que resulta de DEC CX.

¿Qué sucede si MOV [B320], AX se coloca dentro del lazo ? Si bien el resultado final en R sería correcto, la secuencia tardaría más en ejecutarse, pues en cada vuelta habría que acceder a memoria para escribir cada suma parcial.

¹ Esta instrucción no conviene ejecutarla mediante el comando T del Debug, pues puede perder el programa tipeado. Su uso indica que una secuencia termina, pero que la UCP no se detiene, pues INT 20 es una instrucción que ordena una interrupción para que se pase a ejecutar el módulo del sistema operativo que determina cuál es el siguiente programa que se seguirá ejecutando a continuación.

Los pasos desarrollados en la figura 3.6 ordenan que una suma (y en general un procedimiento) se repita un número N de veces conocido de antemano. Se los puede hacer corresponder con la siguiente estructura en un lenguaje de alto nivel:

```
R = 0
FOR i = 1 to N
    R = R + M
```

EJERCICIO 3: Mejora del ejercicio 2 usando JO (Jump if overflow), manejo de constantes, empleo de la instrucción JMP, y uso de etiquetas

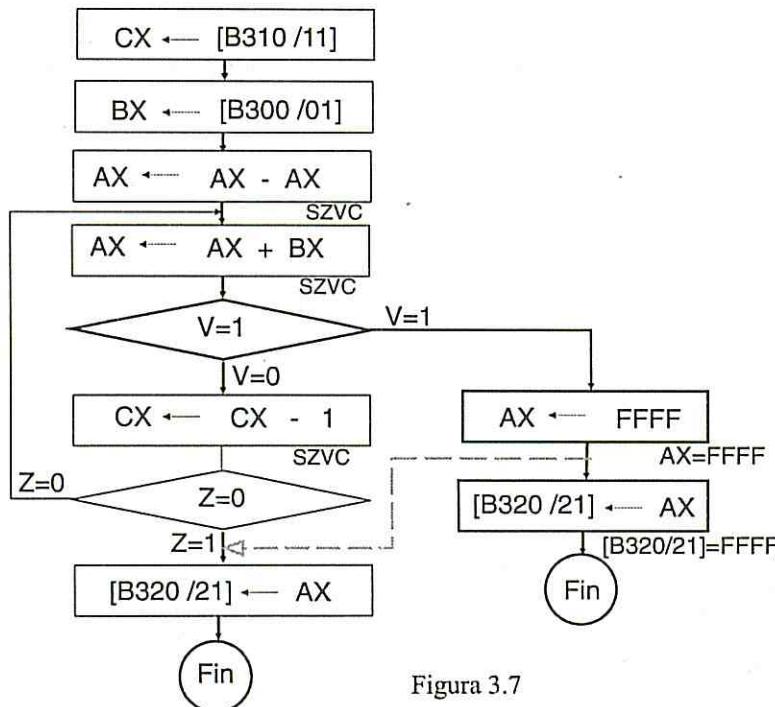


Figura 3.7

En la Unidad 1 de esta obra se define el **overflow** (desborde), término que sólo debe usarse para números enteros.

Para un formato de 16 bits, como el de los registros AX, BX, CX, DX, existe overflow si por ejemplo, una suma de enteros positivos supera el valor 32767. De ser así la UAL genera la indicación **V=1** de resultado errado por 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 la indicación **V=1**, se escriba como resultado en B320 y B321 el entero negativo FFFF (que comienza con bit uno), resultado **imposible** para MxN, dado que M y N son enteros positivos.

Para tal fin (figura 3.7), se modifica la fig 3.6 para preguntar si vale uno el flag V generado luego de hacer la suma.

Si V=1 se crea otra secuencia que pasa el número FFFF hacia AX; y que después en otro paso el valor FFFF de AX lo copia en B320 y B321.

No hay problema que se “pise” con FFFF el resultado acumulado por AX, el curso de ejecución sigue por los mismos pasos del diagrama de la figura 3.6.

dado que éste es errado por ser V=1. Si V=0 (resultado parcial o final correcto) las dos secuencias codificadas en Assembler que se corresponden con las de la figura 3.7 son las siguientes:

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

A 0130	Llevar a AX una copia de la constante FFFF
xxxx:0130 MOV AX, FFFF	Llevar a B320 y B321 de memoria, una copia del valor FFFF de AX
xxxx:0133 MOV [B320], AX	
xxxx:0136 INT 20	Instrucción de final

Figura 3.8

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, conforme al diagrama de la figura 3.7.

JO 0130 ordena saltar (si V=1) a otra instrucción que en este caso no forma parte de la secuencia donde JO se encuentra. Por ello mediante el comando A 0130 hubo que escribir otra secuencia, que empieza con MOV AX, FFFF. Esta última ordena pasar el número FFFF hacia AX y MOV [B320],AX ordena pasar a memoria lo que está en AX, o sea FFFF.

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

La dirección 130 fue elegida arbitrariamente, pero dentro de los límites máximos permitidos para saltar, tema a tratar.

Nueva mejora: se puede obviar la secuencia de 0130 MOV AX, FFFF, si se hace JO 0111 (salto a *MOV [B320], AX*). Dado que con enteros positivos si V=1 el resultado erróneo siempre es negativo (S=1), al quedar éste en B320/1, su signo negativo es suficiente para indicar al programa que lo utilizará que ese resultado es erróneo, lo cual implica que V=1.

GENERALIZACION:

Instrucciones en modo “inmediato” con un operando que siempre tiene el mismo valor (constante):

Obsérvese la diferencia entre MOV AX, FFFF y MOV AX, [FFFF]. La primera ordena cargar en AX el número fijo FFFF, mientras que la segunda ordena cargar en AX el número que contenga entonces la dirección de memoria FFFF.

MOV AX, FFFF al igual que todas las instrucciones que después de la coma sigue un número (sin corchetes) se dice que están en modo “inmediato”. Esta denominación puede explicarse observando el código de máquina de una instrucción en este modo de direccionamiento. Mediante el comando U del Debug podemos hallar por ejemplo el código de máquina de la instrucción MOV AX, FFFF (luego de haber escrito la secuencia en assembler donde ella se encuentra):

```
- 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 formando parte de la instrucción sigue el número fijo FFFF que pasará a AX cuando B8FFFF se ejecute, de donde proviene el nombre “inmediato” de este modo. También pertenecen a este modo por ejemplo ADD BX, 2 (sumar siempre 2 al contenido de BX, y el resultado dejarlo en BX), o MOV AX, 0 (llevar el número cero a AX, lo cual es otra manera de llevar un registro a cero), etc.

En todas las instrucciones en modo inmediato de un programa, un número que ellas operarán (cuando sean ejecutadas) forma parte (en memoria) de estas instrucciones. Cuando cada una de ellas sea ejecutada y se acceda a memoria para pedirla, dicho número llegará a la UCP desde memoria junto con la instrucción, pues el mismo está incluido en ella. O sea que este tipo de instrucciones proveen a la UCP el valor (constante) de un número que intervendrá en la operación que ordenan, y por lo tanto no será necesario acceder otra vez a memoria (mayor tiempo de ejecución) para obtenerlo. Por consiguiente, los valores fijos (constants) que operará un programa forman parte en memoria del mismo y están con las instrucciones en modo inmediato en lugar de estar en memoria en la zona de datos variables de ese programa, por ser datos conocidos de antemano se hallan en la zona de instrucciones. Cambiar el valor de uno de estos números fijos, que se usan típicamente para incrementar o decrementar el valor de un registro usado como puntero de datos de una lista, o para dar valores iniciales a variables en registros al comienzo de un programa, implica modificarlo.

EJERCICIO 3 bis: Mejora del ejercicio 3 empleando la instrucción JMP. Uso de etiquetas

El programa de la figura 3.8 puede escribirse con menos instrucciones, como en la figura 3.9 dado que las secuencias que empiezan en 100 y en 130 tienen las mismas dos últimas instrucciones en común. En el diagrama de la figura 3.7 aparece una línea grisada en punteado que va desde abajo del paso AX ← FFFF hasta el comienzo del paso [B320/1] ← AX. O sea que esta línea indicaría saltar sin condición alguna, de una secuencia a otra (o dentro de una misma secuencia).

Así, luego de ejecutar la primer instrucción que está en 0130 se saltaría incondicionalmente 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) de la figura 3.9 cumplirá este cometido. No es necesario indicar con un rectángulo la existencia de la instrucción JMP, aunque en lo sucesivo por razones didácticas lo haremos (ver figura 3.11).

En general 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. Equivale a un “GO TO” en lenguajes de alto nivel.

Conforme al uso de JMP, las secuencias de la figura 3.7 con la línea en punteado quedarían como indica la figura 3.9.

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

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

Figura 3.9

Figura 3.10

La figura 3.10 repite la secuencia de la figura 3.9 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. Para desarrollar programas en Assembler con papel y lápiz sin usar el Debug usaremos etiquetas. De esta forma podemos independizarnos del Debug y de las direcciones.

EJERCICIO 4: perfeccionamiento del ejercicio 2 teniendo en cuenta que M ó N pueden valer cero.

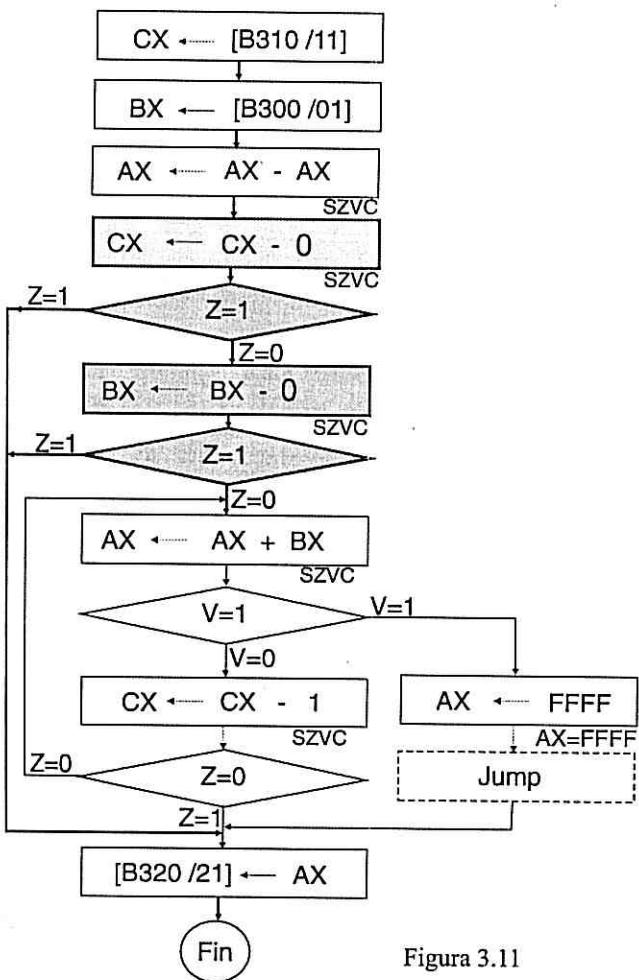


Figura 3.11

variable que está en el registro DX tiene por ejemplo el valor 50, se debe ordenar mediante instrucciones primero hacer la resta $DX - 50$, y luego indirectamente mediante una instrucción de salto preguntar si el resultado de esa resta fue cero ($Z=1$, o sea cero "si"). De ser $Z=0$ (o sea cero "no") dicho resultado será un número que no interesa, pero no será 50. Igualmente (figura 3.11) luego de la inicialización de registros, para saber si N que está en CX vale cero, primero se ordena hacer $CX - 0$ y luego preguntar si $Z=1$. Si $Z=1$ implica que $R = M \times N = 0$. Entonces como antes de hacer $CX - 0$ se hizo $AX=0$ aprovecharemos este cero que está en AX para pasarlo a la variable R que está en B320/1. A tal fin mediante un salto condicional se ordena que si $Z=1$ saltar a la instrucción $[B320/1] \leftarrow AX$ que está al final. Si $Z=0$ indica que N no es cero, por lo que se pasa a evaluar si $M=0$ con los mismos dos pasos empleados para N . Asimismo, si $Z=0$ implica que M tampoco es cero, y por lo tanto se puede seguir con el lazo para realizar las N sumas.

Si se escribe con el Debug **JZ BETA** (con salto hacia adelante), el número con la dirección que va en vez de BETA no se conoce, pues aún no se ha escrito **MOV [B320], AX** con esa dirección (a diferencia de un salto hacia atrás, donde esa dirección se ve en pantalla). En el uso del Debug que acompaña al ejercicio 13 se indica cómo proceder en esos casos.

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 $Z=1$.

En ese caso (diagrama de la figura 3.12), se saltaría cuando sea $CX = 0$, pues recién entonces sería $Z=1$, y se pasaría a ejecutar la secuencia que empieza con $[B320/1] \leftarrow AX$, a la que sigue el fin de secuencia. Hasta que no sea $Z=1$, cada vez que se le resta uno a CX y mientras no sea $CX = 0$ (o sea $Z=0$), no se salta, sino que se continúa con la instrucción siguiente, que en este caso es de salto incondicional (jump en el diagrama lógico de la figura 3.12) hacia la instrucción $AX \leftarrow AX + BX$. Así se vuelve a armar un lazo como en la figura 3.6.

En el ejercicio 2 se planteó que si $N=0$ el lazo con la suma se repetiría indefinidamente pues nunca sería $CX=0$.

Esto último no sucedería en el ejercicio 3, pues al repetirse cierto número de veces la suma, llegaría un punto en que en AX no entraría el resultado ($V=1$), y se escribiría como resultado (errado) FFFF en vez de cero, y el programa terminaría.

Ahora completaremos el ejercicio 2, para que si M ó N valgan cero inmediatamente sea R=0, sin efectuar ninguna suma.

	MOV CX, [B310]
	MOV BX, [B300]
	SUB AX, AX
	SUB CX, 0
OTRA	JZ BETA
	SUB BX, 0
	JZ BETA
OTRA	ADD AX, BX
	JO OVER
	DEC CX
BETA	JNZ OTRA
	MOV [B320], AX
	INT 20
OVER	MOV AX, FFFF
	JMP BETA

Antes de que se realice el lazo detectaremos si vale cero M ó N, que en la inicialización fueron puestos en BX y CX.

La Unidad de Control, encargada de ejecutar las instrucciones, no está capacitada -como podemos estar nosotros- para conocer por sus unos y ceros, el valor de cualquier número que está en un registro o en memoria. Si se quiere conocer si una

variable que está en el registro DX tiene por ejemplo el valor 50, se debe ordenar mediante instrucciones primero hacer la resta $DX - 50$, y luego indirectamente mediante una instrucción de salto preguntar si el resultado de esa resta fue cero ($Z=1$, o sea cero "si"). De ser $Z=0$ (o sea cero "no") dicho resultado será un número que no interesa, pero no será 50.

Igualmente (figura 3.11) luego de la inicialización de registros, para saber si N que está en CX vale cero, primero se ordena hacer $CX - 0$ y luego preguntar si $Z=1$. Si $Z=1$ implica que $R = M \times N = 0$. Entonces como antes de hacer $CX - 0$ se hizo $AX=0$ aprovecharemos este cero que está en AX para pasarlo a la variable R que está en B320/1.

A tal fin mediante un salto condicional se ordena que si $Z=1$ saltar a la instrucción $[B320/1] \leftarrow AX$ que está al final. Si $Z=0$ indica que N no es cero, por lo que se pasa a evaluar si $M=0$ con los mismos dos pasos empleados para N .

Asimismo, si $Z=0$ implica que M tampoco es cero, y por lo tanto se puede seguir con el lazo para realizar las N sumas.

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

En la figura 3.12 el salto condicional no se realiza sobre la misma secuencia, como en la fig. 3.6, sino que se salta a otra secuencia, compuesta por dos instrucciones. Con el comando A 0130 del Debug esta secuencia en la fig. 3.13 se escribe aparte, a partir de la dirección 0130, elegida arbitrariamente, pero que no supera el valor límite permitido para un salto.

Por lo tanto, el "precio" de elegir el valor contrario de un flag, es una destrucción del diagrama lógico, por la aparición de una secuencia nueva, *innecesaria*, con la consiguiente complicación de la codificación con el Debug.

En general cuando se elige la condición de salto y aparece JMP, se debe probar qué pasa si se elige la condición contraria.

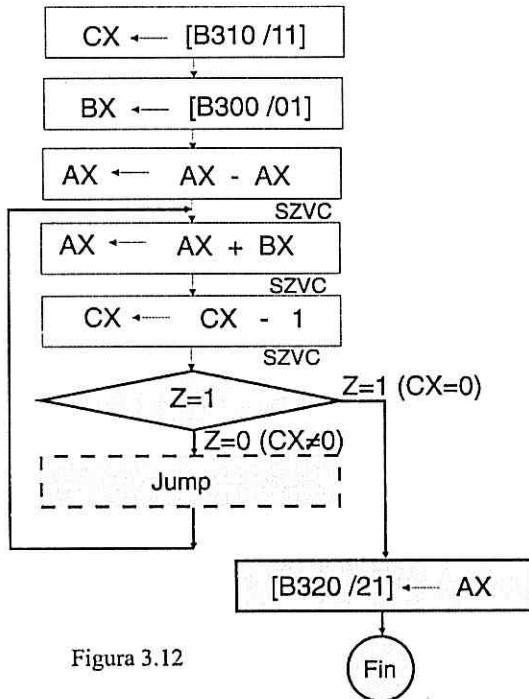


Figura 3.12

A 0100	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	JZ 0130
xxxx:010F	JMP 10A
A 0130	xxxx:0130 MOV [B320], AX
	xxxx:0133 INT 20

Figura 3.13

GENERALIZACION: *Instrucciones de salto condicional:*

En los ejercicios anteriores hemos exemplificado el uso de este tipo de instrucciones, sin las cuales no se podría repetir automáticamente sin intervención humana ni veces una secuencia, ni pasar automáticamente de una secuencia a otra, ni detectar si se superó un formato de representación de datos.

Una instrucción de este tipo en esencia permite, en función del valor de los flags generados por la última operación que hizo la UAL, que la máquina, después de ejecutar esta instrucción, pueda seguir la ejecución con una entre 2 alternativas:

- 1) si uno o más flags tienen el valor planteado en la instrucción (condición para que ocurra el salto), la UC sigue con la ejecución de la instrucción a la que se ordena saltar (cuya dirección permite determinar la instrucción de salto)
- 2) si uno o más flags no tienen el valor por el que pregunta la instrucción, la UC continúa con la instrucción que sigue en memoria a la instrucción de salto (como ocurre en las instrucciones que no son de salto condicionado).

Una instrucción de salto sólo pregunta por los valores de los flags, pero no los modifica, pues en ella la UAL no opera.

La decisión de saltar al comienzo de otra secuencia o de continuar con la secuencia que se venía ejecutando, como una alternativa entre dos caminos posibles, no depende de la UC (que no tiene inteligencia alguna para decidir nada), sino que depende que el resultado anterior que generó la UAL sea o no cero, sea o no positivo, sea o no representable en un formato dado, etc., resultado que se sintetiza en el valor de los flags que también genera la UAL. Es el programador quién establece, al elegir la condición presente en una instrucción de salto, en qué condiciones se va a producir (o no) el salto.

Estas instrucciones primero preguntan por el valor de los flags presentes en la condición de salto, y sólo si esta condición se cumple, ordenan saltar a la instrucción cuya dirección ellas permiten determinar.

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

Codificación y ejecución de una instrucción de salto que tipeada luego de A 0100 es xxxx : 0100 JNZ 0112 ↴

Codificación: como describimos anteriormente, luego de tipar ↴ el programa traductor del Debug reemplaza los caracteres tipeados que quedaron codificados en ASCII a partir de 0100 por el código de máquina de JNZ 112, que es 01110101 00010000 = 75 10 h, codificación que puede verificarse si se tipa el comando U 0100.

75 es el código de operación correspondiente a JNZ, y el 10 que lo acompaña es la diferencia 0112 – 0102 = 10, o sea la dirección (0112) donde se ordena saltar si se cumple que Z=0 menos la dirección (0102) de la instrucción que sigue a JNZ, que es la instrucción que se ejecuta luego de la instrucción de salto si Z=1 (JNZ ocupa las direcciones 0100 y 0101).

Ejecución: cuando la UC pide de memoria la instrucción 7510, el registro IP (Instruction Pointer) tiene el valor 0100. Durante la ejecución de 7510 primero se ordena sumar 2 al IP, pues 7510 ocupa 2 bytes. Con IP = 0102 el IP apunta a la dirección de la instrucción que sigue a 7510 por si Z=0. Luego 75 ordena leer el valor de Z. Si Z=0 al IP se le suma 10, o sea IP = 0102 + 10 = 0112 (dirección del salto); y si Z=1 no se hace nada, con lo cual el IP permanece en 0102.

Dado que 0112 se obtiene sumando 10 al valor 0102 que tiene el IP, el 0112 está en relación, es relativo al valor del IP. Por tal motivo se dice que las instrucciones de salto condicional están en *modo de direccionamiento relativo*, y todas las instrucciones de esta clase se ejecutan de la misma manera ilustrada para JNZ, y todas ocupan 2 bytes de memoria.

El modo relativo no requiere que la dirección 0112 forme parte de la instrucción, lo cual implicaría que la instrucción ocuparía 3 bytes, y en general el tamaño de ella dependería del número de bits que tienen las direcciones de memoria. Asimismo, siempre se debería saltar a 0112, con lo cual los programas no se podrían cambiar de lugar en memoria.

El número (10h = 00010000) que acompaña al código de operación (75 para JNZ) ocupa un byte, y puede ser positivo (empieza con 0 como en este ejemplo) para saltar hacia delante, o negativo si se salta hacia atrás (como en el ejercicio 2). Según se vió en la Unidad 1 de esta obra, un número binario entero de un byte tiene los valores extremos +127 y -128, que serán el número máximo de posiciones de memoria que se puede saltar hacia delante y atrás (7F y 80 en hexa).

Si no se tiene presente los límites citados, el traductor ensamblador del Debug no aceptará la instrucción de salto. Por ejemplo, si se codifica 0500 JNZ 0600, la diferencia de casi 100 posiciones entre 600 y 500 en hexa (o sea 256 en decimal) supera el valor máximo +127, por lo que en este caso JNZ 0600 no será aceptada por el ensamblador del Debug.

Instrucción de salto condicional que sigue a una de resta para determinar si minuendo > < sustraendo

1) PARA NÚMEROS ENTEROS (POSITIVOS O NEGATIVOS)

Cuando en un papel restamos dos números enteros A – B, sabemos que si el signo del resultado es positivo (flag S=0) es $A \geq B$; y si es negativo (S=1) es $A < B$. Esto mismo puede decirse si dicho resultado es correcto por haber sido V=0. En un computador el signo solo no es suficiente para esta determinación. Conforme se trató en la Unidad 1 de esta obra, el resultado de una suma o resta de enteros puede ser erróneo por no poder representarse en el formato (número de bits) con que opera la UAL, lo cual denominamos overflow (V=1). Y si esto ocurre, el signo del resultado, si hubiese sido V=0, es contrario al signo del resultado erróneo cuando es V=1. Vale decir que aunque el resultado esté mal por ser V=1, igualmente se puede determinar el signo que éste hubiera tenido, aunque no el valor completo del resultado.

Pero basta con conocer el signo del resultado de una resta de enteros para poder afirmar que $A \geq B$ ó $A < B$.

Resumiendo lo anterior, luego de efectuar una resta de enteros A – B, el valor de los flags S y V permite determinar:

Si S=0 y V=0 está bien que sea $A \geq B$

Si S=1 y V=0 está bien que sea $A < B$

Si S=0 y V=1 debió haber sido S=1 y por lo tanto se deduce que $A < B$

Si S=1 y V=1 debió haber sido S=0 y por lo tanto se deduce que $A \geq B$

De los 4 casos anteriores resulta que:

Si S y V tienen igual valor ($S=V$) es $A \geq B$; y si S y V tienen distinto valor ($S \neq V$), resulta $A < B$.

Igualmente, para saber si $A = B$ debe hacerse antes $A – B$ para conocer Z. En el ejercicio 4 usamos JZ para ver si el contenido de CX era igual o no a cero, para lo cual antes de JZ que pregunta por el valor del flag Z, se hizo la resta CX – 0.

A continuación se tratan todos los casos posibles para saber como es A respecto de B, siendo A y B enteros.

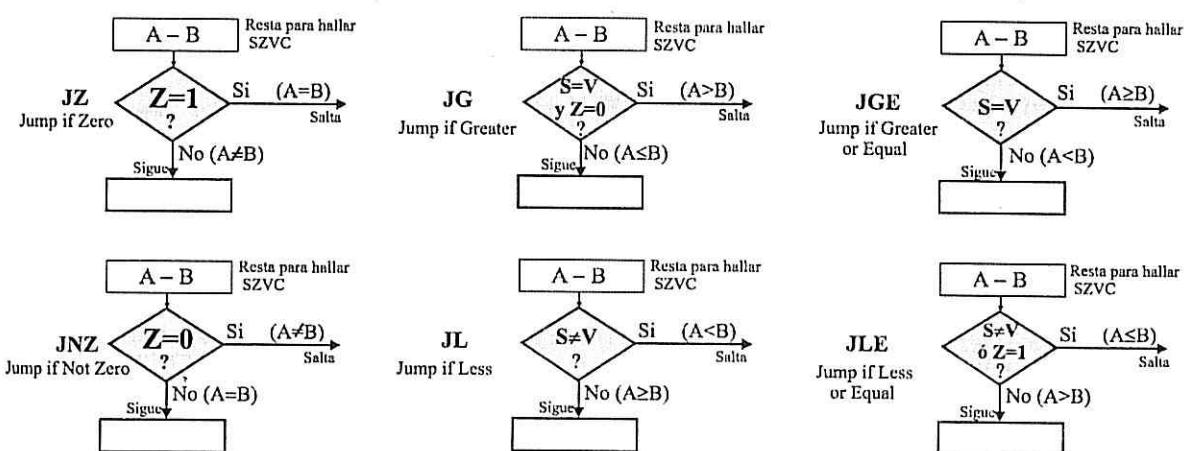


Figura 3.15

Estos diagramas también sirven de recordatorio que si **no** se cumple la condición planteada, siempre debajo del rombo debe seguir otra instrucción, o sea que **no** se puede saltar. Esto último sólo es posible si la siguiente es JMP (ejercicio 5).

Si bien cuando se ejecuta una instrucción de salto condicional la UC lee en el registro de estado el valor de uno o varios flags, en Assembler la condición puede establecerse con dos o tres letras fáciles de recordar (ya hemos usado JZ, JNZ, JO), para preguntar si un número es mayor, igual, o menor que otro, como aparecen al lado de los rombos en la fig. 3.15.

Esto permite desentenderse de los flags, especialmente si la combinación de flags es compleja, y es factible poder codificar una instrucción de salto de dos o más maneras (ver mnemónicas de instrucciones al final de esta obra).

Por ejemplo, la instrucción JZ (jump if Z=1) usada en el ejercicio 4 es equivalente a JE (Jump if equal) si se piensa en la instrucción anterior de resta, pues si el resultado de ésta es cero (Z=1), implica que los números restados son iguales.

2) PARA NÚMEROS NATURALES

Estos números son usados para cuantificar magnitudes (litros, metros, ...) o para codificar caracteres en ASCII, etc. comparten con los enteros el flag Z en las instrucciones JZ y JNZ, pues el cero se codifica igual para enteros o naturales. Con los naturales no se pueden usar los flags S y V, usados para enteros, so pena de resultados con errores inadmisibles. El único flag que es propio de números naturales es el flag C. Su significado para una resta A - B de naturales con n bits, es que si C=0 se "pediría 0 prestado" si la resta continuaría con n+1 bits, lo cual implica que $A \geq B$; y si C=1 se "pediría 1 prestado" en dicha resta, lo cual implica que $A < B$.

Este tema se desarrolló en la Unidad 1 de esta obra, y se sistematiza en las siguientes porciones de diagramas lógicos:

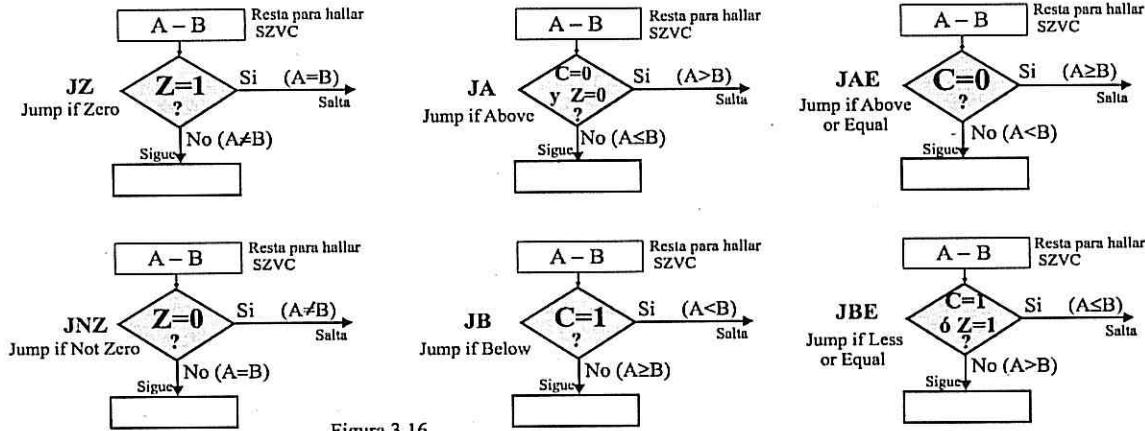


Figura 3.16

Por ejemplo (figura central superior), si al hacer $A - B$ resulta $C=0$ implica que en esa resta de naturales se pidió "0 prestado" por lo que $A \geq B$, y si además $Z=0$ implica que el resultado no es cero, o sea que no puede ser $A = B$ (lo cual ocurriría si $C=0$ y $Z=1$), por lo que en definitiva es $A > B$. Si al hacer $A - B$ resulta $C=1$, implica que en esa resta se pidió "1 prestado", por lo que $A < B$.

EJERCICIO 6

Codificar en Assembler una secuencia para realizar la suma $R = M + N$ de dos números enteros de 32 bits ("long integers"), siendo que con ADD AX, [xxxx] sólo pueden sumarse en la UAL dos números de 16 bits.

Para de poder visualizar y concretar el ejercicio, esta vez supondremos que los datos típicados son $M = -2.650.000_d$ y $N = 3.250.876_d$. En lo que sigue se indican los valores de las variables M y N representados con bit de signo (tema visto en la Unidad 1 de esta obra), sus direcciones, y la suma en cuestión planteada en 32 bits, convertida luego a hexa.

$$\begin{array}{rcl}
 C=1 \\
 + M = + 111111110101110010000001110000 = + FFD790\ 70 \\
 \underline{N} = \underline{00000000001100011001101010111100} = \underline{00\ 319ABC} \\
 R = M+N = 100000000000010010010101100101100 = 100\ 092B\ 2C \\
 \downarrow \quad \downarrow \\
 \text{SZVC} \quad \text{SZVC} \\
 0\ 0\ 0\ 1 \quad 0\ 0\ 1\ 1
 \end{array}$$

Suponemos una UAL que sólo puede sumar dos números de 16 bits como la del 80286. Con esta UAL y mediante instrucciones simularemos una UAL de 32 bits, dividiendo la suma en dos mitades de 16 bits, lo cual implica más tiempo que si la suma se realiza de una sola vez con instrucciones para operar 32 bits en una UAL apropiada. Para cada mitad se dan los valores de los flags SZVC que la UAL generaría. Primero sumaremos las mitades de M y N en negrita, suma que puede arrojar o no un transporte hacia la otra mitad. Como los binarios enteros se suman como naturales, el flag C es propio de estos últimos.

Dado que en una suma con 16 bits el valor del flag C es "lo que me llevo" si la suma continúa en un bit 17, usaremos el flag C para memorizar si me llevo 1 ó 0 hacia la primer columna de la segunda mitad a sumar. En este ejemplo se indica en grisado dicha columna, y la mitad en negrita luego de sumarse arrojará C=1 ("me llevo 1") hacia la segunda mitad.

En definitiva lo que haremos es que si C=1, sumaremos uno en la suma de la segunda mitad, y si C=0, no sumamos uno.

Desarrollaremos este ejercicio de tres maneras diferentes, siendo que los tres primeros pasos son comunes a todas ellas. De la forma conocida se sumará primero los 16 bits menos significativos de M (arbitriamente supuestos más arriba en las direcciones 2000/1) con los de N (en 2008/9), y se almacenarán en 2010/11 los 16 bits del resultado parcial obtenido, que si se ejecuta el programa, en hexa serían 2C y 2B respectivamente, que aparecen en la suma realizada antes en hexa. A continuación plantearemos la suma de la segunda mitad de M y N de tres formas distintas:

2000	70	M
1	90	
2002	D7	
3	FF	
2008	BC	N
9	9A	
200A	31	
B	00	
2010	1	R
2012	2C	

S
L
S
r
e
S
S
c
L
C

S
d
p
t
d
l
p
a

V
V
3.
A

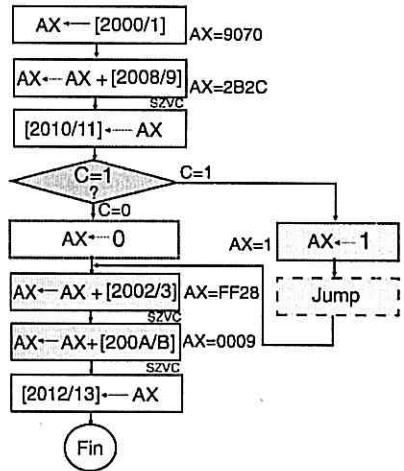


Figura 3.17.a

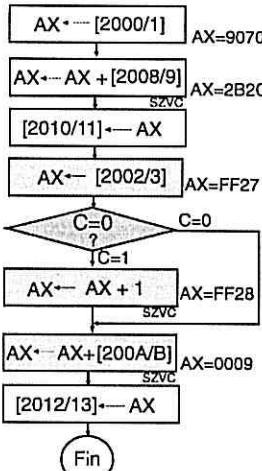


Figura 3.17.b

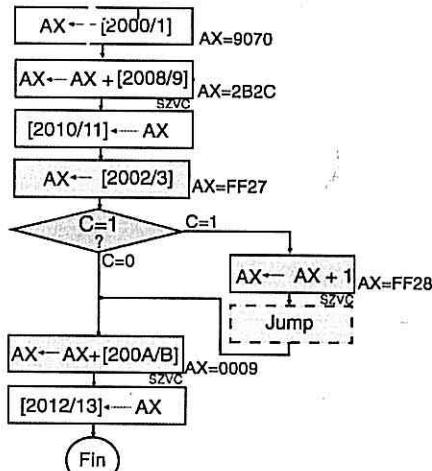


Figura 3.17.(b)

Solución a): conforme a la figura 3.17.a , luego de $[2010/11] \leftarrow AX$ se pregunta por el valor del flag C (generado por la UAL luego de $AX \leftarrow AX + [2008/9]$ y que no cambia luego del paso $[2010/11] \leftarrow AX$, en el cual la UAL no opera). Si C=1 (como ocurriría con estos valores de M y N), mediante $AX \leftarrow 1$ se debe poner en uno a AX (“pisando” a 2B2C resultado del paso $AX \leftarrow AX + [2008/9]$). Luego se debe saltar incondicionalmente al paso $AX \leftarrow AX + [2002/3]$ que en este caso sería $AX \leftarrow 1 + FFD7 = FFD8$, para sumar el uno de acarreo de la posición en grisado de la suma binaria citada. Si C=0 se debe poner AX en cero, mediante $AX \leftarrow 0$, luego de lo cual ahora sería $AX \leftarrow AX + [2002/3] = 0 + [2002/3]$. Sigue $AX \leftarrow AX + [2009/A]$ para sumar al registro AX los 16 bits superiores de N, y luego $[2012/13] \leftarrow AX$ para almacenar el resultado de esta suma (0009 en este caso) en 2012/13. Estos dos pasos serán comunes a las 3 soluciones.

La codificación correspondiente al diagrama lógico de la figura 3.17.a resulta:

Codificaremos en Assembler la secuencia correspondiente a los pasos del diagrama, con las siguientes instrucciones:

MOV AX, [2000]	Llevar a AX una copia del dato que está en memoria en 2000 y 2001
ADD AX, [2008]	Sumar a AX una copia del dato que está en memoria en 2008 y 2009
MOV [2010], AX	Transferir a 2010 y 2011 de memoria una copia del contenido de AX
JC ALFA	Si C=1 saltar a la instrucción que está en ALFA
MOV AX, 0	Si C=0 poner en cero el registro AX
SUMA ADD AX, [2002]	Sumar al valor (1 ó 0) de AX una copia del dato que está en 2002 y 2003
ADD AX, [200A]	Sumar a AX copia del dato que está en 200A y 200B
MOV [2012], AX	Transferir a 2012 y 2013 de memoria una copia del contenido de AX
INT 20	Finalizar
ALFA MOV AX, 1	Como es C=1, poner en uno el registro AX
JMP SUMA	Saltar incondicionalmente a la instrucción que está en SUMA

Solución b): conforme a la figura 3.17.b, luego de $[2010/11] \leftarrow AX$ sigue $AX \leftarrow [2002]$ para llevar hacia AX una copia de la mitad superior de M (para estos datos FFD7) “pisando” la suma parcial anterior existente en AX (2B2C). Como este paso, como el anterior ($[2010/11] \leftarrow AX$) tampoco cambia el valor del flag C, pues en él no interviene la UAL, se pregunta en el paso siguiente por el valor de C. Si C=0 no se suma uno al valor de AX, se “puentea” este paso, y se salta a los dos últimos pasos de la solución a). Si C=1 (o sea si no se cumple la condición C=0) se debe sumar uno (de acarreo a la segunda mitad de la suma) a AX con el paso $AX \leftarrow AX + 1$ (en este caso FFD7 + 1). Luego siguen los dos últimos pasos de la solución a). Las instrucciones que se corresponden con los pasos del diagrama de la figura 3.17.b son:

MOV AX, [2000]	Llevar a AX una copia del dato que está en memoria en 2000 y 2001
ADD AX, [2008]	Sumar a AX una copia del dato que está en memoria en 2008 y 2009
MOV [2010], AX	Transferir a 2010 y 2011 de memoria una copia del contenido de AX
MOV AX, [2002]	Llevar a AX una copia del dato que está en 2002 y 2003
JNC SUMA	Si C=0 saltar a la instrucción que está en SUMA
ADD AX, 1	Si C=1 sumar uno al registro AX
SUMA ADD AX, [200A]	Sumar al valor de AX una copia del dato que está en 200A y 200B
MOV [2012], AX	Transferir a 2012 y 2013 de memoria una copia del contenido de AX
INT 20	Finalizar

Variante menos efectiva de la solución b) según el diagrama lógico de la figura 3.17.(b)

Veremos cómo se modifica la solución b) si en el rombo se plantea C=1 en vez de C=0 como aparece en la figura 3.17.(b). Entonces si C=1 se debe saltar a otra secuencia con el paso $AX \leftarrow AX + 1$, seguido por un salto incondicional a $AX \leftarrow AX + [2009/A]$, suma que sigue al rombo si C=0.

	MOV AX, [2000]	Llevar a AX una copia del dato que está en memoria en 2000 y 2001
	ADD AX, [2008]	Sumar a AX una copia del dato que está en memoria en 2008 y 2009
	MOV [2010], AX	Transferir a 2010 y 2011 de memoria una copia del contenido de AX
	MOV AX, [2002]	Llevar a AX una copia del dato que está en memoria en 2002 y 2003
	JC ALFA	Si C=1 saltar a la instrucción que está en SUMA
SUMA	ADD AX, [200A]	Si C=0 sumar al valor de AX una copia del dato que está en 200A y 200B
	MOV [2012], AX	Transferir a 2012 y 2013 de memoria una copia del contenido de AX
	INT 20	Finalizar
ALFA	ADD AX, 1	Si C=1 sumar uno al registro AX
	JMP SUMA	Saltar incondicionalmente a la instrucción que está en SUMA

Resulta así que el hecho de haber elegido en un rombo como condición de salto la contraria a la de la solución b) trajo aparejado la aparición de una nueva secuencia y un salto incondicional. Por lo tanto, si luego de elegir una condición de salto aparece una nueva secuencia que termina en un salto incondicional hacia la parte inferior de dicho rombo, esta secuencia puede evitarse si se cambia en el rombo a la condición contraria, con lo cual como en la solución b) los pasos que estaban en la secuencia nueva pasan a una secuencia única, y se aprovecha la instrucción de salto para evitar JMP.

Solución c): no emplea instrucción de salto. Las 4 primeras instrucciones son las mismas que las de la solución b), pero no se pregunta por el valor del flag C, dado que 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 último valor que tenía el flag C antes de ejecutar la instrucción, o sea que automáticamente incorpora a la suma el valor 0 ó 1 del dicho acarreo.

MOV AX, [2000]	Llevar a AX una copia del dato que está en memoria en 2000 y 2001
ADD AX, [2008]	Sumar a AX una copia del dato que está en memoria en 2008 y 2009
MOV [2010], AX	Transferir a 2010 y 2011 de memoria una copia del contenido de AX
MOV AX, [2002]	Llevar a AX una copia del dato que está en memoria en 2002 y 2003
ADC AX, [200A]	Sumar a AX copia del dato que está en 200A y 200B <u>más Carry anterior</u>
MOV [2012], AX	Transferir a 2012 y 2013 de memoria una copia del contenido de AX
INT 20	

Con cualquiera de las soluciones planteadas se duplica por software el formato con que opera una UAL de 16 bits (hardware) en una suma multibyte.

EJERCICIO 7: Lectura y escritura de una sucesión de datos consecutivos en memoria (lista o vector)

En memoria se tiene un vector conformado por una lista de **n** elementos N1, N2, N3 ... Nn, que son números de 2 bytes cada uno ordenados sucesivamente en posiciones consecutivas de memoria. La lista (figura 3.18) comienza en la dirección 1000, siendo que en 1500 se da la cantidad n de elementos de la lista. Esta cantidad n en todos los ejercicios se asumirá siempre mayor que uno, para evitar su verificación con la metodología del ejercicio 4. Se pide copiar la lista a partir de la dirección 2000.

La figura 3.18 indica (para N1) los dos movimientos que permiten pasar cada elemento de una lista a la otra, usando AX como registro intermedio. Se han dibujado las dos listas a la par, siendo que en memoria están una debajo de otra. Con las instrucciones que conocemos hasta ahora, por un lado no existen instrucciones como **MOV [2000], [1000]** para el pasaje de una copia de N1 hacia la otra lista. Para tal fin (figura 3.18) primero se debe pasar una copia de N1 hacia AX, usando **MOV AX, [1000]**; y luego una copia de AX hacia 2000/1, mediante **MOV [2000], AX**. Por otra parte, si con esas instrucciones codificáramos los pasos para pasar los **n** elementos que están en memoria en la primer lista hacia la otra, se tendría que escribir **n** veces cada par de instrucciones MOV:

MOV AX, [1000]	Una copia de N1 que está en 1000/1 pasa a AX	
MOV [2000], AX	Una copia de N1 que está en AX pasa a 2000/1	(primer par de instrucciones MOV)
MOV AX, [1002]	Una copia de N2 que está en 1002/3 pasa a AX	
MOV [2002], AX	Una copia de N2 que está en AX pasa a 2002/3	(segundo par de instrucciones MOV)
.....		
MOV AX, [1 ...]	Una copia de Nn que está en 1 ... pasa a AX	
MOV [2 ...], AX	Una copia de Nn que está en AX pasa a 2 ...	(enésimo par de instrucciones MOV)

En el corchete de cada instrucción **MOV AX, [1hhh]** hay un número de valor fijo (1hhh), que es la dirección de una celda de memoria. Lo mismo en **MOV [2hhh], AX**. Este número forma parte de la instrucción. Así, el código de máquina de **MOV AX, [1000]** en hexa es: A1 0010 (0010 codifica a 1000).

Con el fin de evitar como en este caso repetir n veces la escritura de instrucciones, para localizar datos que están en direcciones consecutivas de memoria se usan los registros SI, DI y BX definidos en la figura 1.

Los registros **SI** ("source indexation") y **DI** ("destination indexation") son los que se eligen en primer lugar como punteros, y como **BX** también puede usarse como acumulador, conviene utilizarlo sólo en el caso que haya tres listas a puntear.

En correspondencia con el uso de estos registros para direccionar celdas de memoria, existen instrucciones para poner entre corchetes el nombre del registro que proveerá la dirección de memoria a acceder, como ser **MOV AX, [SI]**. Como siempre los corchetes simbolizan memoria, y la dirección a acceder es el valor que en este caso tiene SI. **MOV AX, [SI]** ordena pasar a AX una copia del número contenido en la celda cuya dirección indica (apunta) entre corchetes el registro SI, y también una copia del número contenido en la celda siguiente, pues el registro AX es de 16 bits.

Si al registro SI se le da el valor inicial 1000 mediante la instrucción en modo inmediato **MOV SI, 1000**, resultará $SI=1000$; y si después se ejecuta **MOV AX, [SI]**, será equivalente a ejecutar **MOV AX, [1000]**, para que N1 pase a AX. Dado que el elemento siguiente a pasar está en 1002, este valor se halla sumando 2 a SI ($SI \leftarrow SI + 2 = 1000 + 2 = 1002$), y así indexando en 2 a SI se pueden determinar las direcciones (1004, 1006, ...), de los siguientes elementos a pasar. Ahora si se vuelve a ejecutar **MOV AX, [SI]**, será equivalente a ejecutar **MOV AX, [1002]**, como se necesita.

Igualmente, si a DI se le da el valor inicial 2000 mediante **MOV DI, 2000**, las sucesivas ejecuciones de **MOV [DI], AX** seguidas de sumarle 2 a DI, son equivalentes a **MOV [2000], AX**; **MOV [2002], AX**; **MOV [2004], AX**, etc.

Entonces, si repetidamente se ejecutan n veces 4 instrucciones: **MOV AX, [SI]** y **MOV [DI], AX**, AX seguidas por el par de sumas **ADD SI, 2** y **ADD DI, 2**, se habrán copiado todos los elementos de una lista en la otra. Para lograr la repetición de estos 4 pasos, en el diagrama lógico de la figura 3.18 que luego desarrollaremos se armó un lazo semejante al de la figura 3.6.

Sistematizando, hasta ahora hemos usado los registros AX, BX, CX, DX para contener valores de datos o resultados, y a partir de este ejercicio usaremos SI, DI y BX para contener números que son direcciones de datos o resultados. Los registros, al igual que las posiciones de memoria, se corresponden con variables, cuyo valor es el contenido de los mismos. Las instrucciones del tipo **ADD AX, [SI]**; **MOV AX, [BX]**; **MOV [DI], AX**; etc., corresponden 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, que son los únicos registros que pueden aparecer entre corchetes. Estos registros serán indexados con instrucciones, a diferencia del IP que es actualizado automáticamente por la UC. Las instrucciones en modo indirecto por registro permiten recorrer uno tras otro a datos contenidos en posiciones sucesivas de memoria, como es el caso de una lista o vector.

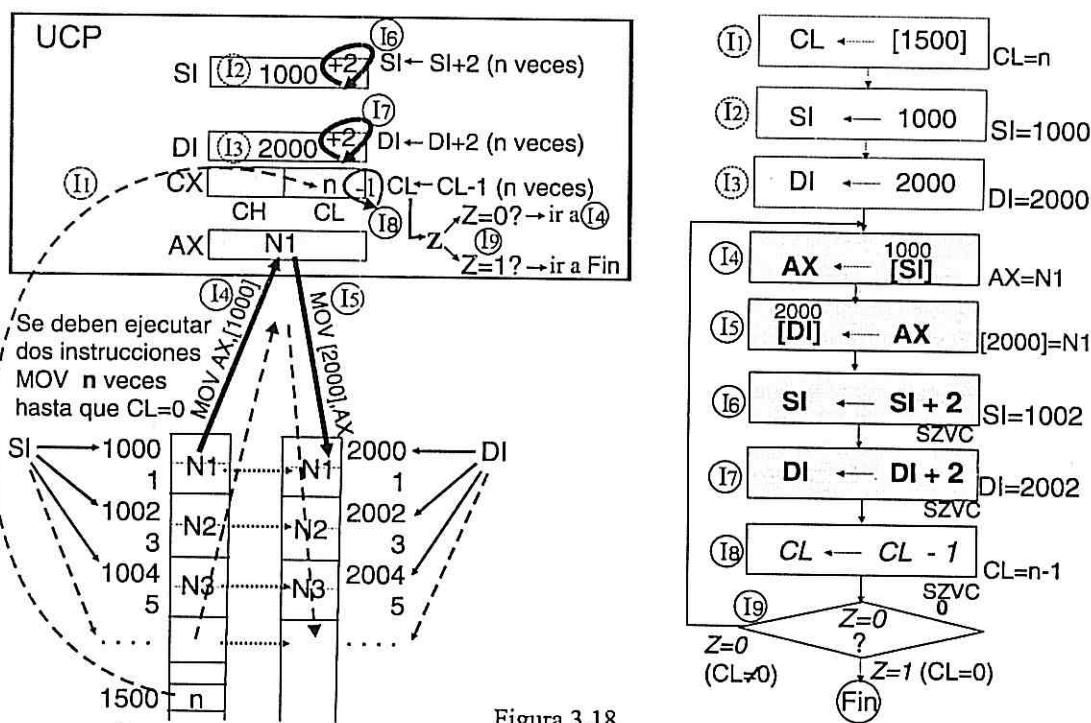


Figura 3.18

El diagrama de la figura 3.18 tiene la misma concepción que el de la figura 3.6 para repetir un procedimiento y se basa en el esquema que lo acompaña. En este caso en cada vuelta (*ciclo*) del lazo se copia un nuevo elemento de una lista a otra.

En el primer paso (11) de la inicialización, en el esquema y en el diagrama lógico se eligió ordenar que una copia del número n que está en 1500 pase a los 8 bits de CL (mitad inferior de CX que puede contener hasta $11111111 = 255$). Esta asignación del número n al registro CL la efectuaremos también en ejercicios siguientes, por si hace falta usar CH.

Al igual que N del ejercicio 2, el valor de n es el número de veces que se repetirá el lazo.

Siguen los pasos 2 y 3 con la asignación de los valores 1000 a SI, y 2000 a DI.

El procedimiento (14, 15, 16 e 17) son los 4 pasos antes planteados. Comienza con 14, que en el primer ciclo ordena que una copia de N1, que está en 1000/1, pase a AX. En este ciclo el paso 5 luego ordena que N1 pase de AX hacia 2000/1.

Luego de los pasos 6 y 7 los registros **SI** y **DI** quedan apuntando a 1002 y 2002 direcciones necesarias para copiar a N2. Con $DI \leftarrow DI + 2$ termina el procedimiento a repetir, y como en el ejercicio 2 se resta uno al contador regresivo de ciclos ($CL \leftarrow CL - 1$), y se pregunta al final de cada ciclo si $Z=0$ o sea si $CL \neq 0$. De ser así se vuelve a repetir el ciclo para que el elemento siguiente se copie de una lista a la otra. En caso de ser $Z=1$ ($CL=0$) se finaliza el programa.

Esta forma de repetir ciclos para operar elementos de una lista será usada en los ejercicios que traten este tipo de datos.

Como dirección inicial de la lista y la cantidad **n** de elementos son datos, no hace falta dar la dirección final de la misma.

A continuación se dan las instrucciones correspondientes al diagrama lógico de la figura 3.18.

MOV CL, [1500]	Llevar a CL una copia de n que está en 1500.
MOV SI, 1000	Llevar a SI el valor 1000.
MOV DI, 2000	Llevar a DI el valor 2000.
OTRO	
MOV AX, [SI]	Llevar hacia AX una copia de un número de 2 bytes de la lista apuntado por SI.
MOV [DI], AX	Llevar hacia la dirección apuntada por DI el número de 2 bytes contenido en AX.
ADD SI, 2	Sumar 2 al contenido de SI.
ADD DI, 2	Sumar 2 al contenido de DI.
DEC CL	Restarle uno a CL para que cambien los flags SZVC.
JNZ OTRO	Mientras $Z=0$ volver al procedimiento que empieza en OTRO.
INT 20	Fin de secuencia.

Es importante manejar correctamente el uso de los corchetes. Si en vez de **MOV CL, [1500]** escribimos **MOV CL, 1500** al registro CL irá el valor 1500 en vez de ir **n**. Y si en lugar de **MOV SI, 1000** se escribe **MOV SI, [1000]**, hacia el registro SI irá el elemento N1 en vez de 1000.

EJERCICIO 8: Sumar los elementos de una lista (vector)

En un lenguaje de alto nivel con variable "enteros" se escribió la sentencia: $R(n) = N(1) + N(2) + N(3) + \dots + N(n)$ que ordena sumar los valores (de 2 bytes cada uno) de los **n** elementos de un vector, localizados en celdas consecutivas de memoria, conformando una *lista* a partir de la dirección 1000 (figura 3.19).

La cantidad **n** de números de la lista está en la dirección 1500. La variable **R** está en las direcciones 2000/1. Si al sumar un nuevo número de la lista se produce overflow, no seguir sumando y sólo enviar a la dirección 2002 la cantidad **k** de elementos que faltaban sumar. En caso de que se sumen los **n** números, enviar a **R** el resultado y escribir en 2002 el valor de **k** que en ese caso será cero, como indicativo de resultado correcto. Codificar en Assembler un programa que obtenga los resultados requeridos.

El diagrama de la figura 3.19 tiene la misma estructura que el de las figuras 3.6 y 3.18 para repetir un procedimiento y se basa en el esquema que lo acompaña. En este caso en cada vuelta (*ciclo*) del lazo se suma un nuevo elemento de la lista. Como siempre a la derecha de los rectángulos se indican los valores resultantes correspondientes a la primera vuelta.

En el ejercicio 2 se codificó $R = 0 + M + M + \dots$ (**N** sumas); y ahora se codificará: $R = 0 + N1 + N2 + N3 + \dots$ (**n** sumas). El registro **SI** irá apuntando (mientras no haya overflow) al elemento siguiente a sumar. Para tal fin (I1) a **SI** se le dió el valor 1000, luego (I2) el número **n** se llevó hacia **CL**, y después (I3) AX fue llevado a cero (pues $R = 0 + N1 + \dots$). El procedimiento comienza (I4) con **AX ← AX + [SI]** que ordena sumar a AX una copia de los 2 bytes del número que está en la dirección indicada entre corchetes por **SI**; y dejar el resultado en AX. En Assembler es **ADD AX, [SI]**.

En el primer ciclo se tendría $AX = 0 + N1 = N1$. Le sigue la pregunta (I5) si $V=1$. Si es así (fig. 3.19), se salta a la instrucción (I9) que ordena copiar el valor de **CL** (que indica la cantidad **k** de elementos que falta sumar) en la dirección 2002, para después finalizar la ejecución del programa, sin guardar lo acumulado en AX.

Si no hay overflow ($V=0$) prosigue el ciclo (I6) sumando 2 a **SI**, que en el primer ciclo quedaría en 1002, apuntando a N2. Sigue el control de las veces que se ejecutará el lazo: se le resta uno a **CL** (I7), y se pregunta si $Z=0$ (I8). Mientras así sea, se salta a ejecutar otra vez al comienzo del ciclo donde está la instrucción (I4) que suma el elemento siguiente.

Cuando sea $Z=1$ ($CL=0$) si bien **SI** está apuntando a un supuesto elemento **n+1**, éste no se suma, dado que la secuencia continúa con dos almacenamientos: primero se guarda (I8) una copia en 2000/1 (**R**) la suma total que está en AX, y luego (I9) se guarda en 2002 una copia del valor **k** de **CL**, que ahora debe ser cero, indicación de que el valor de R es correcto.

En definitiva, si $k = 0$ el resultado presente en 2000/1 está bien, y si $k \neq 0$ el mismo es errado por que antes fue $V=1$.

Debe consignarse que en el ejercicio 2 se operó con enteros positivos, y si era $V=1$ se escribía como resultado un número negativo, que empezaba con uno, lo cual indicaba que estaba mal, pues ello no es posible. En este caso como el resultado puede ser positivo o negativo, no puede hacerse lo mismo si $V=1$, por lo que se optó por la indicación que provee **CL**.

La secuencia correspondiente al diagrama lógico de la figura 3.19 sigue a continuación del mismo. En dicho diagrama aparecen los resultados obtenidos luego de cada paso, y en el lazo los correspondientes a la primer vuelta. Este diagrama finaliza de forma semejante al del ejercicio anterior, y que se repetirá en todos los ejercicios con listas.

Se debe tener presente que **ADD AX, [SI]** no es lo mismo que **ADD AX, SI**, dado que la primera es de modo indirecto por registro, y la segunda es de modo registro, y ordena sumar al contenido de AX una copia del registro SI, dejando el resultado en AX.

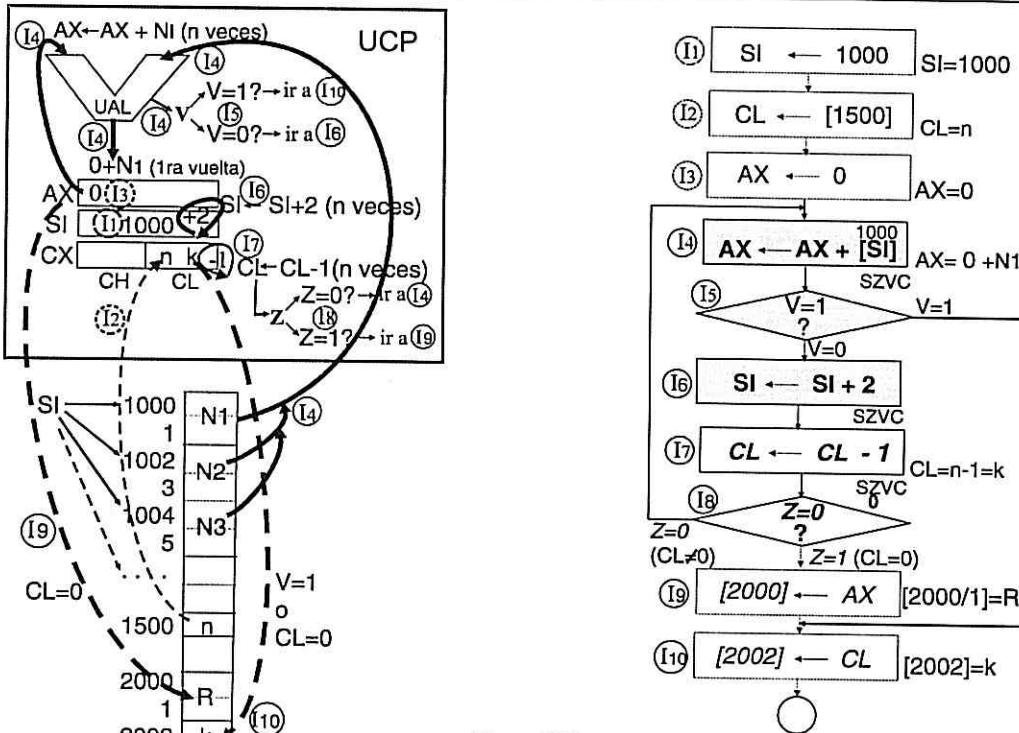


Figura 3.19

```

MOV SI, 1000
MOV CL, [1500]
MOV AX, 0
OTRO ADD AX, [SI]
JO ALFA
ADD SI, 2
DEC CL
JNZ OTRO
MOV [2000], AX
ALFA MOV [2002], CL
INT 20
    
```

SI apunta al comienzo de la lista de datos
Carga en CL (mitad inferior de CX) la cantidad n que está en 1500
Lleva AX a cero
Suma a AX el número de la lista apuntado por SI
Si hay overflow saltar a ALFA
Suma 2 a SI para que apunte al elemento siguiente de la lista
Decrementa CL
Mientras Z sea 0, volver a OTRO
Carga en 2000 y 2001 el resultado de la suma
Carga en 2002 el contenido k de CL
Fin

EJERCICIO 9: búsqueda en lista usando la instrucción comparación, con resta sin asignación de resultado
A partir de la dirección 2000 se tiene una lista de caracteres codificados en ASCII (figura 3.20), siendo que su número n está en la dirección 1500. Encontrar el número k de veces que en la lista se encuentra e letra E y dicho número guardararlo en la dirección 3000. Asimismo, cada vez que se encuentra una E, indicar su código ASCII (45h) en una segunda lista que empieza en 2000, seguido de la dirección donde se encontraba dicha letra E.

En el esquema y diagrama de las figuras 3.20 y 3.21 primero se inicializan los registros. El registro CL con el número n de caracteres (I1); el registro SI con la dirección inicial (2000) de la lista de caracteres ASCII (I2); el registro DI con la dirección inicial (3000) de la segunda lista (I3), que guarda las direcciones donde se encontró una letra E; y el registro AH (mitad superior de AX) se pone a cero (I4), pues va a ser usado como contador de las veces que se encontró una letra E. Luego (I5), una copia del valor x de la lista apuntado por SI es llevado hacia AL (mitad inferior de AX, pues AX tiene 16 bits y cada dato x es un carácter ASCII de 8 bits). Si se usara AX irían hacia AH y AL dos caracteres consecutivos. No hace falta llevar AL a cero, pues no se usa como acumulador, y cada MOV que envía un valor a AL “pisá” el anterior. Entonces usaremos AL para enviarle cada uno de los n valores de la lista, paso previo a la resta AL - 45 en la UAL. En la primer vuelta en AL queda el valor x apuntado por SI = 1000. De forma análoga al ejercicio 4, para saber si una variable tiene un valor determinado, en este caso si x vale 45, primero se ordena hacer AL - 45 (I6), con lo cual tomarán valor los flags SZVC. Hasta ahora en todas las instrucciones de resta, su resultado “pisaba” el valor que antes de ella tenía el acumulador. Si $AL = x = 45$, y se ordena la resta con asignación $AL \leftarrow AL - 45$, ésta sería $45 - 45 = 0$, resultando $AL = 0^1$. Entonces si se quiere enviar desde AL el 45 a la otra lista, como pide el ejercicio, en vez del 45 se enviaría un cero.

¹ Inclusive en el ejercicio 4 como primer paso de la comparación también se usó la resta con asignación $CX \leftarrow CX - 0$, pero como se resta cero, el resultado que quedará en CX será el mismo que el existente antes de esa resta.

A fin de evitar usar más instrucciones para salvar a 45 que está en AL antes que el resultado de la resta lo destruya, existe la resta sin asignación de resultado, que se usa como primer paso para determinar si un valor es igual, mayor o menor que otro. Este tipo de resta para comparar, se denomina no muy felicitamente “comparación” (operación CMP en Assembler). O sea que una comparación es una resta, pero sin asignación, una resta en la que no interesa el resultado, y que solo sirve para que la UAL genere valores de los flags SZVC correspondientes a dicha resta, sin afectar los números que se operan. Por tal motivo en el diagrama de la figura 3.21 en el paso AL - 45 se indica que AL sigue conteniendo el valor de x.

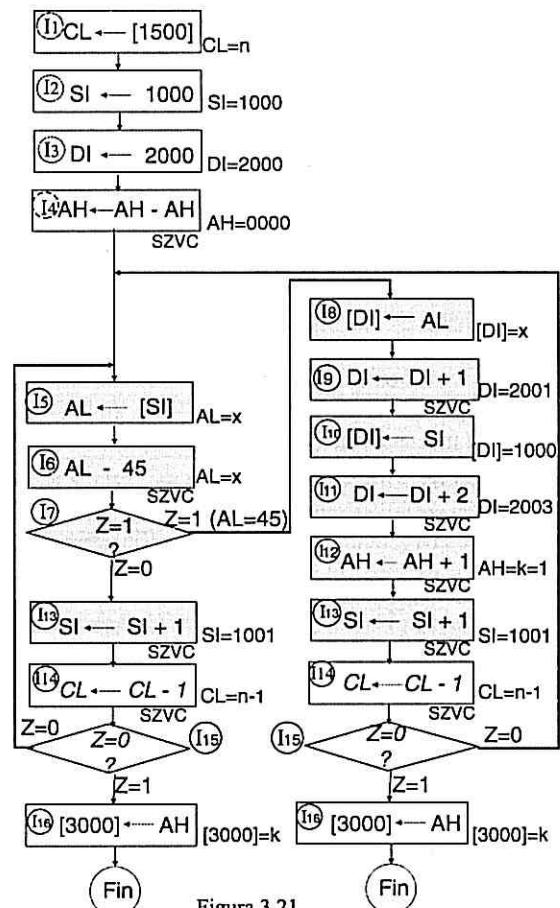
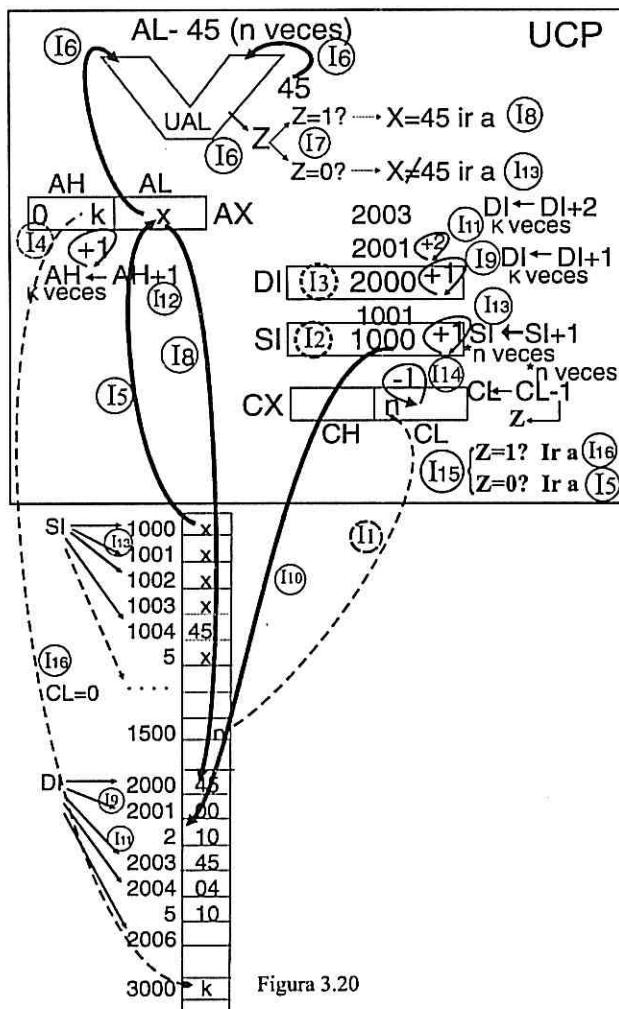
Decimos “no muy felicitamente” por que en realidad la comparación se determina con la instrucción de salto que siempre sigue a la resta sin asignación, que en este caso (I7) ordena preguntar por el valor de Z, pues se necesita saber si AL = 45. En otros casos luego de dicha resta se pregunta por otros flags (figs 15 y 16) para saber si uno es mayor o menor que otro. Si Z=1 implica que el resultado es cero, y por lo tanto x = 45, o sea que x es el código ASCII de la letra E; y por lo tanto, como se pide, una copia de x que está en AL debe ser llevada (I8) a la dirección de la lista apuntada por DI=2000. De ser Z=0, será x ≠ 45, por lo que el valor de x no es el ASCII de la E, y como se tratará, hay que pedir otro carácter.

Siguiendo con Z=1, como luego de I8 el valor x = 45 ocupó la celda 2000 apuntada por DI, hay que sumar uno a DI (I9) para que en el paso I10 que ordena que una copia del contenido de SI que sigue apuntando a la dirección (1000) donde se localizó x = 45, se guarde en la segunda lista luego de dicho 45. De no ser así, el 1000 se escribiría en 2000 y 2001, pisando el 45. Entonces haciendo DI ← DI + 1 = 2000 + 1 = 2001, se escribiría en 2001 y 2001 la dirección 1000.

Como luego de escribir las celdas 2001 y 2002, DI sigue en 2001, se debe ordenar (I11) sumarle 2, de modo que apunte 2 celdas más abajo (a 2003), por si se encuentra otro código x = 45 en la lista de los n caracteres, como se supone luego.

Conforme a lo pedido, como se encontró el código 45 de una letra E, sólo falta ordenar sumar 1 al contador del número de veces k que se halló E, el cual (inicialmente en cero) está en AH. Por lo tanto (I12), en cada oportunidad que se encuentra un 45 se hace AH ← AH + 1 (0 + 1 = 1, la primera vez que se localiza 45).

Habiendo realizado todos los pasos solicitados cada vez que se localiza el 45, con I13 se ordena CL ← CL - 1 = n - 1, para registrar que faltan analizar n - 1 códigos de caracteres de la lista. Mientras el resultado de esta resta no sea cero (Z=0), se debe volver al paso I5 para leer el siguiente carácter de la lista y pasarlo al registro AL.



En el paso I7 el que se pregunta por el valor de Z, seguimos los pasos si era Z=1, en cuyo caso (figura 3.21) se salta a I8. Si es Z=0 dijimos que x ≠ 45, o sea x no es el ASCII de la E, por lo que simplemente se debe ordenar que SI apunte a la

dirección (2001) del elemento siguiente a analizar, para luego llevarlo hacia AL mientras CL no sea cero. Por lo tanto, se deben realizar los mismos pasos 13, 14 y 15 antes desarrollados para tal fin en el caso de ser Z=1. Se ha asumido en la figura 3.20 que en la dirección 1004 se localizó un segundo código 45, y por consiguiente en la dirección 2003 (hacia la cual quedó apuntando DI en el supuesto que en 1000 sea x = 45) se guardó 45; y 1004 en 2004/5. Entonces en el contador debería ser k = AH = 2.

Si los n códigos fueran todos 45, en la figura 3.20 se observa que n veces se repetirían los pasos 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 5, 6, 7 . . . resultando k = n. De hallarse k veces el 45 se repetirían estos pasos k veces; y de no encontrarse ningún código 45 en la lista, se repetirían n veces los pasos 5, 6, 7, 13, 14, 15, resultando k = 0. Cualquiera sea el valor de k alcanzado en AH, luego de analizar los n caracteres el programa finaliza (I16) con k en 3000.

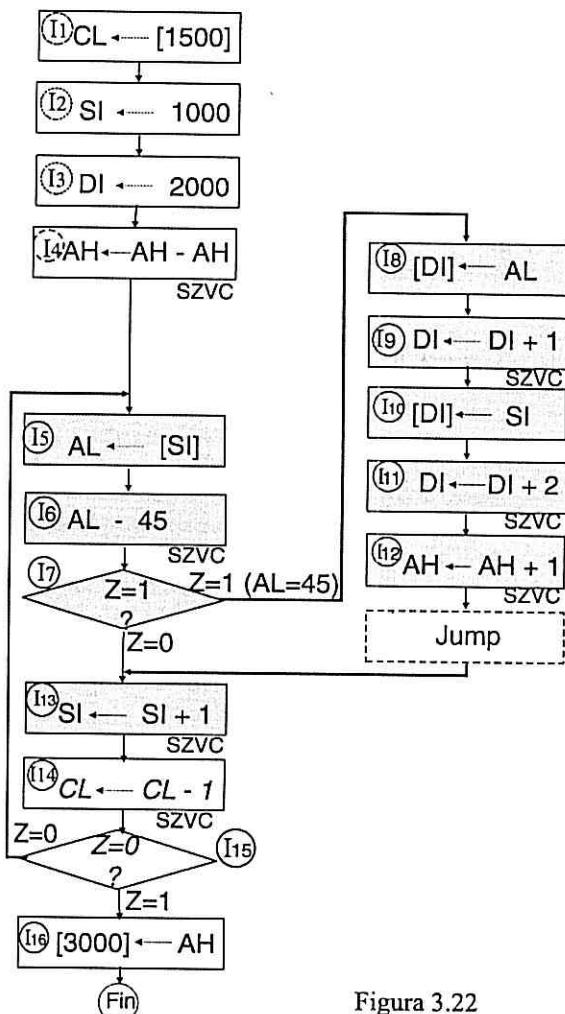


Figura 3.22

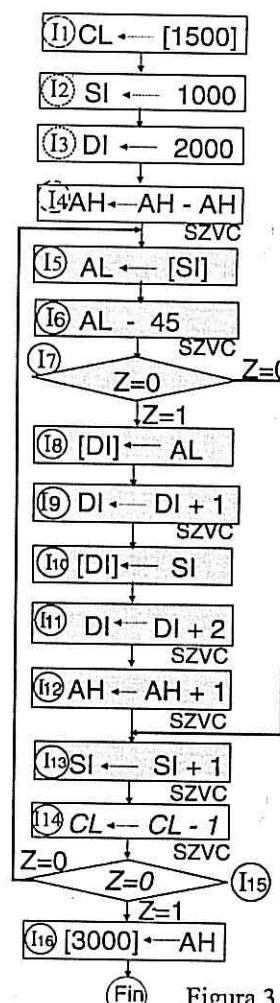


Figura 3.23

Mejoras al diagrama lógico de la figura 3.21

Las dos ramas del diagrama lógico de la figura 3.21 fueron construidas de forma independiente, una para Z=1 y otra para Z=0, sin discutir pasos comunes que pueden tener. Sin embargo los pasos 13, 14, 15 y 16 son idénticos, y pueden compartirse como indica la figura 3.22 si mediante una instrucción de salto incondicional (jump) se ordena saltar de una secuencia a la otra. Esto ya se planteó en la figura 3.7, siendo que en este ejemplo es más evidente el ahorro de instrucciones en el momento de codificar en Assembler.

Pero es factible una estructuración más sencilla aún, con una sola secuencia y sin el salto incondicional, aún más fácil de codificar en Assembler, planteada en el diagrama de la figura 3.23, y que fue expuesta en relación con las figuras 3.17.b.

Cuando desde un rombo se salta a otra secuencia, que termina en un jump hacia ese mismo rombo, si se invierte el valor del flag(s) por el cual se pregunta en el rombo dicha secuencia puede escribirse debajo de dicho rombo, para seguir con el paso a partir del cual comienza la secuencia común, y conformar así una sola secuencia en vez de dos. En ésta el rombo con la condición modificada ordena saltar a la instrucción a partir del cual empieza dicha secuencia común.¹

En la figura 3.23, si Z=1 ahora se sigue hacia abajo con los pasos 8 al 14, como se desarrollaron más arriba; y si Z=0 se

¹ Esta regla de simplificación práctica es fruto de la experiencia del autor sobre el tema, y no la ha visto enunciada en ninguno de los libros sobre Assembler que ha consultado.

ordena saltar al paso 13, "puenteando" los pasos 8 al 12, que sólo deben ordenarse si Z=1. Suponiendo que los n elementos sean todos letras E, se repetirían n veces los pasos I5 al I15, y si no hubiera ninguna E, los pasos I5, I6, I7, I13, I14 e I15.

Del diagrama lógico de la figura 3.23 resultan la secuencia escrita a continuación.

	MOV CL, [1500]	Carga en CL (mitad inferior de CX) el número n de elementos contenido en 1500
	MOV SI, 1000	SI apunta al comienzo de la lista de datos
	MOV DI, 2000	DI apunta al comienzo de la otra lista
	SUB AH, AH	Pone AH en cero
OTRO	MOV AL, [SI]	Lleva a AL una copia del carácter de la lista apuntada por SI
	CMP AL, 45	Compara el valor de AL con 45
	JNZ ALFA	Si AL no es igual a 45 (Z=0) saltar a ALFA
	MOV [DI], AL	Lleva a la dirección apuntada por DI una copia del número 45 que está en AL
	INC DI	Incrementa DI
	MOV [DI], SI	Escribe una copia del valor de SI en la lista apuntada por DI
	ADD DI, 2	Suma 2 al puntero DI
	INC AH	Incrementa AH
ALFA	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [3000], AH	Carga en 3000 el valor del contador contenido en AH
	INT 20	Fin

Una instrucción como **MOV AL,[SI]** con corchete a la derecha ordena **leer** un elemento de una lista en memoria, y cargarlo en AL. En cambio **MOV [DI],SI** ordena **escribir** en otra lista apuntada por DI, una copia del valor de SI. Si por error se escribe **MOV DI, SI** se ordenaría pasar a DI el contenido de SI (movimiento dentro de la UCP).

EJERCICIO 10 basado en el ejercicio 8 y en un paso del 9.

En un lenguaje de alto nivel con variable "enteros" se escribió la sentencia: $R(n) = N(1) + N(2) + N(3) + \dots + N(n)$ que ordena sumar los valores (de 2 bytes cada uno) de los n elementos de un vector, localizados en celdas consecutivas de memoria, conformando una lista a partir de la dirección 1000 (figura 3.19).

La cantidad n de números de la lista está en la dirección 1500. La variable R está en las direcciones 2000/1. Si al sumar un nuevo número de la lista se produce overflow, no seguir sumando y enviar a la variable R la suma parcial correcta anterior a la suma que generó overflow; y en 2002/3 la dirección del sumando que produjo overflow. En caso de que se sumen los n números, guardar en R el resultado, y escribir 0000 en 2002/3. Codificar en Assembler un programa que permita obtener los resultados requeridos.

	MOV SI, 1000	SI apunta al comienzo de la lista de datos
	MOV CL, [1500]	Carga en CL (mitad inferior de CX) la cantidad n que está en 1500
	MOV AX, 0	Lleva AX a cero
OTRO	MOV BX, AX	Llevar a BX una copia del último valor acumulado sin overflow en AX
	ADD AX, [SI]	Suma a AX el número de la lista apuntado por SI
	JO ALFA	Si hay overflow saltar a ALFA
	ADD SI, 2	Suma 2 a SI para que apunte al elemento siguiente de la lista
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [2000], AX	Carga en 2000 y 2001 el resultado de la suma
	MOV SI, 00	Lleva SI a cero
	MOV [2002], SI	Se escribe 0000 en 2002/3
	INT 20	Fin
ALFA	MOV [2000], BX	Carga en 2000 el contenido k de BX
	MOV [2002], SI	Se escribe en 2002/3 la dirección del sumando que generó overflow
	INT 20	Fin

EJERCICIO 11: otra variante del ejercicio 8 |

Idem ejercicio 8, pero si al sumar un nuevo número de la lista se produce overflow, no seguir sumando y enviar a la dirección 2002 el número de orden en la lista (que va de uno hasta n) del sumando que generó overflow. En caso de que se sumen los n números, enviar a R el resultado y escribir 00 en 2002.

Codificar en Assembler un programa que permita obtener los resultados requeridos.

	MOV SI, 1000	SI apunta al comienzo de la lista de datos
	MOV CL, [1500]	Carga en CL (mitad inferior de CX) la cantidad n que está en 1500
	MOV AX, 0	Lleva el acumulador AX a cero
	MOV CH, 0	Lleva CH a cero, para usarlo como generador de los números de orden 1, 2, ..., n
OTRO	INC CH	Suma uno a CH, para que en la primera vuelta indique 1, en la segunda 2, etc.
	ADD AX, [SI]	Suma a AX el número de la lista apuntado por SI
	JO ALFA	Si hay overflow saltar a ALFA
	ADD SI, 2	Suma 2 a SI para que apunte al elemento siguiente de la lista
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [2000], AX	Carga en 2000 y 2001 el resultado de la suma
	MOV [2002], CL	Pone un cero en 2002, aprovechando que CL llegó a cero
	INT 20	
ALFA	MOV [2002], CH	Carga en 2002 el número de orden contenido en CH del sumando que generó V=1
	INT 20	Fin

EJERCICIO 12: determinación si un número natural es mayor que otro, usando la instrucción JA (Jump if above, o sea saltar si está por arriba) para números naturales para encontrar el mayor de una lista de números.

Se tiene una lista de números naturales de dos bytes cada uno, que empieza en la dirección 1000, siendo que la cantidad n de números está en la dirección 1500, siendo que n números naturales de la lista, y guardar el mismo en la dirección 2000.

El presente ejercicio lo realizaremos de 3 formas diferentes

Primera solución: mantiene la forma conocida de contar la repetición del lazo partiendo del número n . Didácticamente es más sencilla, pero hay una repetición extra de la ejecución del lazo, por que en la primera vuelta se compara el primer número con sí mismo. Usamos AX por que los números son de 2 bytes. En las 3 soluciones al inicio comparten los pasos 1, 2 y 3; se supone que N1 es el mayor y se lo lleva hacia AX. Si $N2 > N1$, N2 sustituye a N1 en AX. De no ser así, en AX sigue N1. Si $N3 > AX$, N3 sustituye al mayor que indicaba AX. Así en AX debe ir quedando el número mayor encontrado a medida que se recorre la lista, mientras no sea superado por el valor de otro elemento. En la 2da vuelta de la 1er solución (esquema de la fig 3.24), N2 reemplaza a N1 en AX.

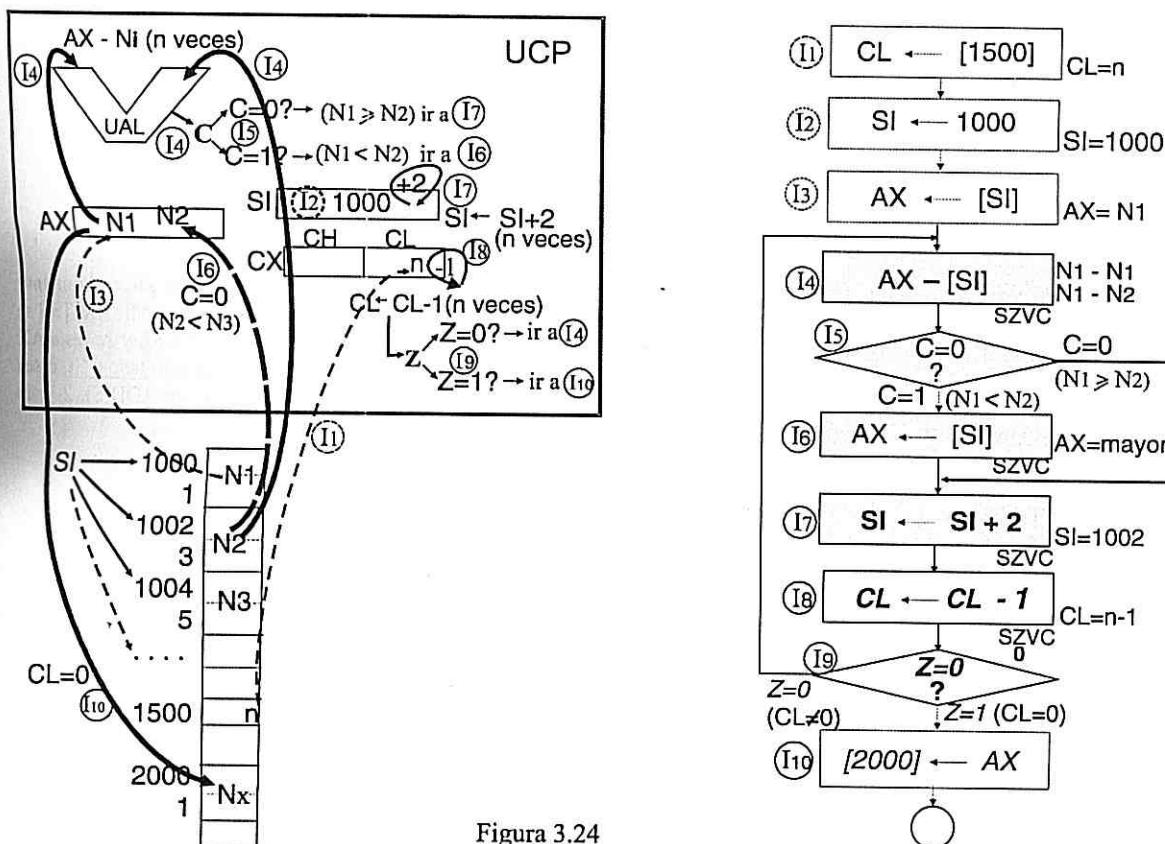


Figura 3.24

Para conocer en esa 2da vuelta si $N1 \geq N2$ se hace (ver figura 3.16): $N1 - N2 = AX - N2$ (paso 4), y si $C=0$ es $N1 \geq N2$. De ser así (paso 5) de la figura 3.22 se debe saltar al paso 7, salteando el paso 6 que ordena reemplazar el valor ($N1$) que contenía AX por una copia del valor ($N2$) contenido en la dirección de memoria (1002) apuntada por SI . Si $C=0$ es $N1 < N2$ por lo que $N2$ es el mayor, y debe seguirse con el paso 6 para que quede en AX el nuevo mayor ($N2$). En el caso que se comparan dos números iguales, como se indica para la primer vuelta del diagrama con $N1 - N1$, será $C=0$ y $Z=1$ por ser cero el resultado. Entonces también se salteará el paso 6, evitando el tiempo que requiere su ejecución. Obsérvese que la resta $AX - [SI]$ ordenada, debe ser sin asignación del resultado en AX (resta para comparar, planteada en el ejercicio 9), pues de no ser así dicho resultado "pisaría" el valor del número mayor hallado hasta ese momento. La codificación en Assembler correspondiente al diagrama lógico de la figura 3.24 resulta:

	MOV CL, [1500]	Carga en CL la cantidad n de elementos que está en 1500
	MOV SI, 1000	SI apunta al comienzo de la lista de datos
	MOV AX, [SI]	Lleva el primer número de la lista a AX, suponiendo que es el mayor
OTRO	CMP AX, [SI]	Resta del valor supuesto mayor, indicado en AX, el número apuntado por SI
	JA BETA	Saltar a BETA si el valor que está en AX está por arriba (mayor) del apuntado por SI
	MOV AX, [SI]	Carga en AX el nuevo número mayor apuntado por SI
BETA	ADD SI, 2	Suma 2 a SI preparándolo para que apunte al próximo elemento
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [2000], AX	Guarda en 2000 el número que resultó ser el mayor
	INT 20	Fin

Nota: en vez de JA puede usarse JNC (Jump if not Carry, o sea si $C=0$). Esta equivalencia de mnemónicos puede verse al final del texto.

Segunda solución: no se pierde tiempo para comparar el primer elemento consigo mismo. En la inicialización, dado que $N1$ se lleva hacia AX con $MOV AX, [SI]$, y que el lazo comienza restando $N1 - N2$, el lazo se debe repetir $n - 1$ veces, pues con $N1$ en AX ya se consideró un elemento. Por tal motivo, luego de $DEC CL$ se parte de $n - 1$.

En Assembler resulta la siguiente secuencia, que sólo difiere en la inicialización de la primera solución:

	MOV CL, [1500]	Carga en CL la cantidad n de elementos que está en 1500
	MOV SI, 1000	SI apunta al comienzo de la lista de datos
	MOV AX, [SI]	Lleva el primer número de la lista a AX, suponiendo que es el mayor
	DEC CL	Decrementa CL, pues ya con $MOV AX, [SI]$ se tomó un número de la lista (el 1ro)
	ADD SI, 2	Suma 2 a SI para que apunte al segundo elemento de la lista
OTRO	CMP AX, [SI]	Resta al valor supuesto mayor, indicado en AX, el número apuntado por SI
	JA BETA	Saltar a BETA si AX es mayor que el valor que apunta SI, sino seguir con $MOV AX, [SI]$
	MOV AX, [SI]	Carga en AX el nuevo número mayor apuntado por SI
BETA	ADD SI, 2	Suma 2 a SI preparándolo para que apunte al próximo elemento
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [2000], AX	Guarda en 2000 el número que resultó ser el mayor
	INT 20	Fin

Tercera solución: reducción de la longitud del programa usando instrucciones con registro indexado en la misma. Se puede escribir la secuencia anterior de la manera siguiente, donde las instrucciones $ADD SI, 2$ y $CMP AX, [SI]$ se han reemplazado por la instrucción $CMP AX, [SI+2]$. Esto es, en una misma instrucción se suma 2 a SI , y se resta AX menos el número apuntado por $SI + 2$. El corchete con $SI + 2$ implica una suma sin asignación del resultado a SI , o sea que después de ejecutar $CMP AX, [SI+2]$ la dirección que guarda SI no cambia. Por ello se requiere $ADD SI, 2$ para actualizar el contenido de SI , como se codificó en las dos soluciones anteriores.

	MOV CL, [1500]	Carga en CL la cantidad n de elementos que está en 1500
	MOV SI, 1000	SI apunta al comienzo de la lista de datos
	MOV AX, [SI]	Lleva el primer número de la lista a AX, suponiendo que es el mayor
	DEC CL	Decrementa CL, pues ya con $MOV AX, [SI]$ se tomó un número de la lista (el 1ro)
OTRO	CMP AX, [SI+2]	Resta al valor supuesto mayor, indicado en AX, el número apuntado por SI
	JA BETA	Saltar a BETA si AX es mayor que el valor que apunta SI, sino seguir con $MOV AX, [SI]$
	MOV AX, [SI+2]	Carga en AX el nuevo número mayor apuntado por SI
BETA	ADD SI, 2	Suma 2 a SI preparándolo para que apunte al próximo elemento
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [2000], AX	Guarda en 2000 el número que resultó ser el mayor
	INT 20	Fin

EJERCICIO 13: diagrama lógico con rombos consecutivos

Se tiene una lista de números enteros que comienza en la dirección 2000, siendo que la cantidad total n de dichos números 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 cantidad de negativos y positivos hallada guardarla en las direcciones 1600 y 1700, respectivamente.

Desarrollar en Assembler un programa que permita obtener los resultados requeridos. Luego, usando el Debug, escribirlo en memoria y ejecutarlo con los valores dados para las variables (datos).

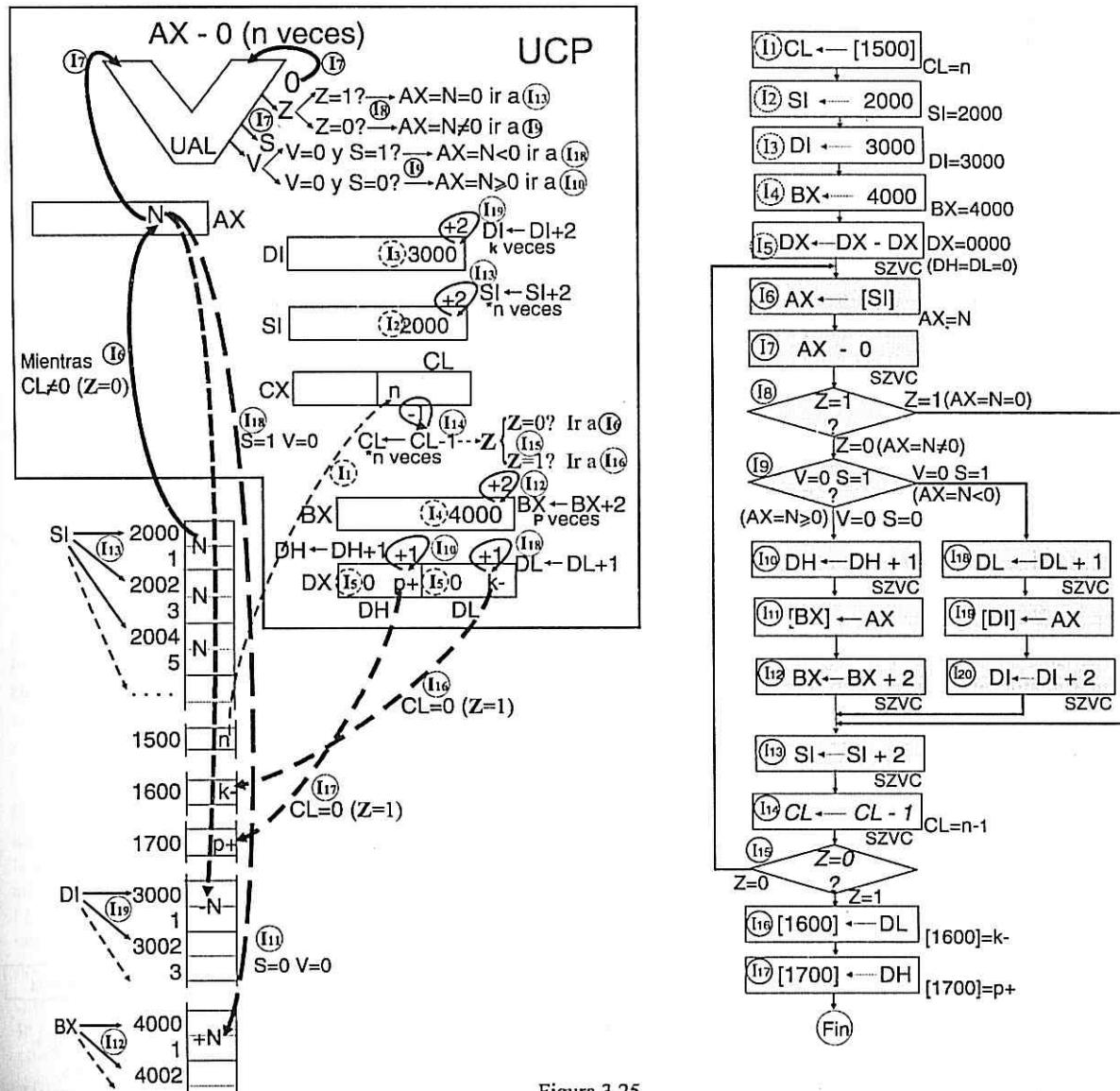


Figura 3.25

Este ejercicio cuyo esquema de pasos a efectuar y diagrama lógico correspondiente aparecen en la figura 3.25 se basa en los ejercicios 7 y 9. En los 5 primeros pasos se inicializan: CL con el número n ; SI, DI y BX con las direcciones iniciales elegidas en el esquema para las 3 listas a utilizar; y DX se pone en cero para que sus mitades DH y DL queden en cero como contadores de cantidad de números positivos y negativos, respectivamente.

Para saber si cada número entero N de la lista (apuntado por SI) es positivo, negativo o cero, se debe restar $N - 0$.

Como para ordenar esta resta no existe la instrucción CMP [SI],0, primero (paso 6) se debe enviar N (apuntado por SI) hacia AX, con lo cual $AX = N$; y luego (paso 7) hacer $AX - 0 = N - 0$. Si en el paso 8 se detecta $Z=1$ implica (fig. 3.15) que $N - 0 = 0$, ó sea que $N=0$, en cuyo caso ese cero no se debe guardar, según se pide, y se deben hacer los pasos 12 a 15 para pedir otro número; y si $Z=0$ (cero no) implica que el resultado de $N - 0$ no es cero, por lo que N tampoco es cero.

Entonces, el paso 8 sirve para determinar si el valor de N que se analiza es igual o distinto de cero. En este último caso hay que detectar en el paso siguiente (9) si N es negativo o positivo, siendo que en el paso 7 se hizo $AX - 0 = N - 0$.

Los valores de los flags SZVC que resultan de esta resta no cambian en el paso 8, puesto que una instrucción de salto sólo pregunta por los valores de los flags, pero no los modifica, dado que en la misma la UAL no hace ninguna operación.

El diagrama central inferior de la figura 3.15 para números enteros sirve de modelo para detectar si $A < B$, y proporciona los flags por lo que se pregunta y el código Assembler (JL) de la instrucción. Según ese diagrama, si $V \neq S$ (o sea si $S=0$ y $V=1$ ó $S=1$ y $V=0$) es $A < B$; y si $V=S$ (o sea si $S=0$ y $V=0$ ó $S=1$ y $V=1$) es $A \geq B$. En nuestro caso el paso $A - B$ se corresponde con $AX - 0$ del paso 6, en el cual es $S=1$ y $V=0$, o bien es $S=0$ y $V=0$, pues no puede haber overflow si a un número se le resta cero. Si en el paso 8 se detecta que fue $S=1$ y $V=0$, el resultado es negativo por empezar con uno, y si fue $S=0$ y $V=0$, el resultado es positivo o cero, por comenzar con cero. Como que al paso 9 se llega si en el paso 8 se detectó $Z=0$, o sea que N no es cero, no puede ser $N \geq 0$, y sólo puede ser $N > 0$ (positivo). Por lo tanto, si en el paso 8 se descarta que N sea cero, en el paso 9 se determina si N es negativo o positivo. Si N es positivo, se siguen con los pasos 10 a 12, que incrementan el contador CH de positivos. Si N es positivo, el curso del diagrama seguirá con el paso 10 que ordena incrementar el contador DH de positivos. Luego (paso 11) se envía una copia de AX hacia la lista de positivos apuntada por DI, y como N ocupa 2 bytes, en el paso 12 se le suma 2 a DI, por si aparece otro número positivo en la lista, como en el ejercicio 9. Los pasos 13 al 15 son los mismos que cuando se detecta un cero en la lista, a fin de obtener el siguiente número que corresponde en la lista, o si se terminó con los n números de la lista ($Z=1$), guardar en los valores de los contadores de positivos y negativos (pasos 16 y 17). Si N es negativo ($V=0$ y $S=1$, o sea $V \neq S$ pues tienen distinto valor) saltar a otra secuencia que consta de los pasos 18 al 20 para: incrementar el contador DL de negativos, pasar una copia de $AX = N$ a la lista de negativos apuntada por BX, y luego indexar en 2 a BX por si aparece otro negativo, para terminar esta secuencia con un salto incondicional hacia la secuencia principal, a fin de que se ejecuten los pasos 13 al 15 comunes para $N=0$, $N > 0$ y $N < 0$. Como se solicita, este ejercicio además de ser codificado con el Debug, luego se ejecutará mediante el mismo, con valores supuestos para las variables (datos). Para ello primero con el comando E se entrarán los datos, y luego con el comando A el programa, aunque el orden es indistinto. Se siguen las directivas para uso del Debug dadas en la Unidad I de esta obra. Los caracteres que debe tipar el operador están en negrita itálica, mientras que la respuesta del Debug se muestra en negrita normal.

C:\WINDOWS>*debug*

Con el comando E se escribirán un numero positivo (0025), un cero, un negativo (8033), y otro positivo (0042)

-E 2000 ↴

xxxx:2000 BF.25 C9.00 E1.00 AC.00 AA.33 0A.80 C0.42 75.00 ↴

En 1500 se escribe la cantidad de números ($n=4$)

-E 1500 ↴

xxxx:1500 E8.04 ↴

Mediante el comando A se escriben las instrucciones en Assembler correspondientes a la figura 3.25. Luego de escribir cada instrucción, con el Enter (↵) el traductor Ensamblador pasa los códigos ASCII de la misma a código de máquina.

-A 0100 ↴

xxxx:0100 MOV CL, /1500/ ↴
 xxxx:0104 MOV SI, 2000 ↴
 xxxx:0107 MOV DI, 3000 ↴
 xxxx:010A MOV BX, 4000 ↴
 xxxx:010D SUB DX, DX ↴
 xxxx:010F MOV AX, /SI/ ↴
 xxxx:0111 CMP AX,0 ↴

Carga en CL la cantidad de números 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, para poner a cero los contadores DH (+) y DL (-)
 Carga en AX una copia del numero entero de la lista apuntado por SI
 Al número entero contenido en AX le resta 0 para que los flags tomen valor

xxxx:0114 JZ 0100 ↴

Si el resultado de la resta es cero salta a una instrucción de esta misma secuencia¹

xxxx:0116 JL 0130 ↴
 xxxx:0118 INC DH ↴
 xxxx:011A MOV /BX/, AX ↴
 xxxx:011C ADD BX, 2 ↴
 xxxx:011F ADD SI, 2 ↴
 xxxx:0122 DEC CL ↴
 xxxx:0124 JNZ 010F ↴
 xxxx:0126 MOV /I600/, DL ↴
 xxxx:012A MOV /I700/, DH ↴
 xxxx:012E INT 20 ↴

Si dicho resultado es negativo saltar a una secuencia que empieza en 0130²
 Si dicho resultado no es negativo (o sea es positivo) incrementa el contador de +
 Guarda una copia del número entero contenido en AX en la lista de positivos
 Se actualiza BX sumándole 2 para guardar el próximo positivo
 Se actualiza SI en 2 por si hay que pedir otro número entero de la lista
 Se decrementa CL por haber considerado un elemento de la lista
 Mientras $Z=0$ volver a 010F
 Guarda la cantidad de negativos en 1600
 Guarda la cantidad de positivos en 1700

¹ Si antes de codificar en Assembler se hizo el diagrama lógico (figura 3.25), en el mismo se ve la instrucción a la que se quiere saltar. Pero en el Debug como aún no se escribió dicha instrucción (que cuando así haga recién se verá que está en la dirección 011F) en la instrucción JZ se escribe 0100, dirección cercana falsa que luego se corregirá, cuando más abajo se escribe A 0114 JNZ 011F.

Al hacer luego U 0100 para revisar el programa, se verifica que todo lo escrito en Assembler está bien.

² Se eligió arbitrariamente 130 como una dirección cercana, siendo que en una instrucción de salto sólo se puede saltar 127 (80 en hexa) hacia delante expresado en decimal (ó 128 hacia atrás) en relación con la dirección de la instrucción que sigue a la de salto.

-A 0114 ↴ (para corregir la instrucción de salto que está en 0114)
 xxxx:0114 JZ 011F ↴ Se corrige JZ 0100 por JZ 011F
 xxxx:116 ↴

-A 0130 ↴ (para escribir la secuencia a la que se salta con JL 0130). Cada secuencia lateral exige un comando A
 xxxx:0130 INC DL ↴ Se incrementa el contado de negativos
 xxxx:0132 MOV [DI], AX ↴ Guarda una copia del número entero contenido en AX en la lista de negativos
 xxxx:0134 ADD DI, 2 ↴ Se actualiza DI sumándole 2 para guardar el próximo negativo
 xxxx:0137 JMP 011F ↴ Salta a 011F
 xxxx:0139 ↴

Para ver si una secuencia a ejecutar está bien escrita, se usa el comando U seguido por la dirección donde comienza

	Programa escrito usando etiquetas
xxxx:0100 8A0E0015	MOV CL,[1500]
xxxx:0104 BE0020	MOV SI, 2000
xxxx:0107 BF0030	MOV DI, 3000
xxxx:010A BB0040	MOV BX, 4000
xxxx:010D 29D2	SUB DX, DX
xxxx:010F 8B04	MOV AX, [SI]
xxxx:0111 3D0000	CMP AX, 0000
xxxx:0114 7409	JZ 011F
xxxx:0116 7C18	JL 0130
xxxx:0118 FEC6	INC DH
xxxx:011A 8907	MOV [BX], AX
xxxx:011C 83C302	ADD BX, +02
xxxx:011F 83C602	ADD SI, +02
-U (para seguir viendo instrucciones restantes)	
xxxx:0122 FEC9	DEC CL
xxxx:0124 75E2	JNZ 010F
xxxx:0126 88160016	MOV [1600], DL
xxxx:012A 88360017	MOV [1700], DH
xxxx:012E CD20	INT 20
U 0130 ↴	
xxxx:0130 FEC2	INC DL
xxxx:0132 8905	MOV [DI], AX
xxxx:0134 83C702	ADD DI, +02
xxxx:0137 EBE6	JMP 011F

Los puntos suspensivos de la secuencia izquierda indican otras instrucciones que arroja el comando U, pero que no interesan. Son "basura". Este comando realiza lo que se denomina "disassembler" o "unassembler". Esto es, a partir de la dirección dada (0100) interpreta los códigos de máquina en sus correspondientes instrucciones en assembler, efectuando el *proceso contrario* al que realiza el programa traductor Ensamblador.

La secuencia de la derecha, que viene de la página anterior, es otra forma de resolver el ejercicio mediante una sola secuencia, sin usar instrucciones jump ("go to"). En la codificación se han usado etiquetas.

A continuación se ejecutará el programa arriba codificado con A 0100 y con flechas se detallan los movimientos.

-RIP ↴
 IP 1246
 :0100-R ↴ AX=0025 BX=4000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3002

Tener presente que el comando T ejecuta la instrucción que se ve escrita arriba del mismo, como indica la primer flecha hacia la derecha

```

DS=106D ES=106D SS=106D CS=106D IP=0100 NV UP EI PL NZ NA PO NC
106D:0100 8A0E0015 → MOV CL, [1500] DS:1500=04 (indica que antes de ejecutar MOV CL,[1500] en memoria 1500 hay 04)
-T ↴ (04 pasa de memoria a CL)
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=0104 NV UP EI PL NZ NA PO NC
106D:0104 BE0020 MOV SI, 2000
-T ↴ (2000 que vino con la instrucción pasa a SI)
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=0107 NV UP EI PL NZ NA PO NC
106D:0107 BF0030 MOV DI, 3000
-T ↴
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=010A NV UP EI PL NZ NA PO NC
106D:010A BB0040 MOV BX, 4000
  
```

- T ↴
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=010D NV UP EI PL NZ NA PO NC
106D:010D 29D2 SUB DX, DX

- T ↴
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=010F NV UP EI PL ZR NA PE NC
106D:010F 8B04 MOV AX, [SI] DS:2000-0025 (antes de ejecutar MOV AX, [SI] en memoria 2000/1 hay 0025)

- T ↴ (De memoria pasó hacia AX una copia de 0025 apuntado por SI=2000)
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0111 NV UP EI PL ZR NA PE NC
106D:0111 3D0000 CMP AX, 0000 (se ordena restar AX - 0 = 25 - 0 para ver el valor de los flags)

- T ↴
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0114 NV UP EI PL NZ NA PO NC
106D:0114 7409 JZ 011F Con JZ se salta si Z=1; como fue NZ (Z=0 ó sea el resultado de AX - 0 no fue cero) no se saltará

- T ↴
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0116 NV UP EI PL NZ NA PO NC
106D:0116 7C18 JL 0130 Con JL se salta si S≠V; como en AX - 0 fue NV (V=0) y PL (S=0) o sea S=V no se saltará a 0130

- T ↴
AX=0025 BX=4000 CX=0004 DX=0000 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0118 NV UP EI PL NZ NA PO NC
106D:0118 FEC6 INC DH ↓ (DH pasa de 00 a 01 pues se encontró un positivo)

- T ↴
AX=0025 BX=4000 CX=0004 DX=0100 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=011A NV UP EI PL NZ NA PO NC
106D:011A 8907 MOV [BX], AX (Se ordena guarda en memoria apuntada por BX=4000 el número positivo 0025 hallado)

- T ↴
AX=0025 BX=4000 CX=0004 DX=0100 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=011C NV UP EI PL NZ NA PO NC
106D:011C 83C302 ADD BX,+02

E 2000_/
106D:2000 25. 00. (Así se verifica antes de ejecutar ADD BX,+02, que desde AX, en 2000/1 de memoria se escribió 0025)

- T ↴
AX=0025 BX=4002 CX=0004 DX=0100 SP=FFEE BP=0000 SI=2000 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=011F NV UP EI PL NZ NA PO NC
106D:011F 83C602 ADD SI,+02 ↓ (SI pasó de 2000 a 2002)

- T ↴
AX=0025 BX=4002 CX=0004 DX=0100 SP=FFEE BP=0000 SI=2002 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0122 NV UP EI PL NZ NA PO NC
106D:0122 FEC9 DEC CL (Se ordena hacer CL ← CL - 1 ó sea 4 - 1)
- T ↴ (CL pasa de 04 a 03)

AX=0025 BX=4002 CX=0003 DX=0100 SP=FFEE BP=0000 SI=2002 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0124 NV UP EI PL NZ NA PE NC
106D:0124 75E9 JNZ 010F NZ (Z=0) indica que CL - 1 = 4 - 1 no fue cero, y se saltará, pues JNZ ordena hacerlo si es Z=0
- T ↴ (IP salta de 0124 a 010F como ordena JNZ 010F)

AX=0025 BX=4002 CX=0003 DX=0100 SP=FFEE BP=0000 SI=2002 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=010F NV UP EI PL NZ NA PE NC
106D:010F 8B04 MOV AX,[SI] DS:2002-0000 (antes de ejecutar MOV AX, [SI] en memoria 2002/3 hay 0000)

- T ↴ (De memoria pasó hacia AX una copia de 0000 que está apuntado por SI=2002)

AX=0000 BX=4002 CX=0003 DX=0100 SP=FFEE BP=0000 SI=2002 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0111 NV UP EI PL NZ NA PE NC
106D:0111 3D0000 CMP AX, 0000 (se ordena restar AX - 0 = 0 - 0 para ver el valor de los flags)

- T ↴
AX=0000 BX=4002 CX=0003 DX=0100 SP=FFEE BP=0000 SI=2002 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0114 NV UP EI PL ZR NA PE NC
106D:0114 7409 JZ 011F ZR (Z=1) indica que AX - 0 = 0 - 0 fue cero, y se saltará, pues JZ ordena hacerlo si es Z=1
- T ↴ (IP salta de 0114 a 011F como ordena JZ 01FF)

AX=0000 BX=4002 CX=0003 DX=0100 SP=FFEE BP=0000 SI=2002 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=011F NV UP EI PL ZR NA PE NC
106D:011F 83C602 ADD SI,+02 ↓ (SI pasó de 2002 a 2004)

AX=0000 BX=4002 CX=0003 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0122 NV UP EI PL NZ NA PO NC
106D:0122 FEC9 DEC CL CL (Se ordena hacer CL ← CL - 1 ó sea 3 - 1)
- T ↴ (CL pasa de 03 a 02)

AX=0000 BX=4002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
DS=106D ES=106D SS=106D CS=106D IP=0124 NV UP EI PL NZ NA PO NC
106D:0124 75E9 JNZ 010F NZ (Z=0) indica que CL - 1 = 3 - 1 no fue cero, y se saltará, pues JNZ ordena hacerlo si es Z=0

- T ↴ (IP salta de 0124 a 010F como ordena JZ 010F)
 AX=0000 BX=4002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=010F NV UP EI PL NZ NA PO NC
 106D:010F 8B04 MOV AX, [SI] DS:2004-8033 (antes de ejecutar MOV AX, [SI] en memoria 2004/5 hay 8033)

- T ↴ (De memoria pasó hacia AX una copia de 8033 que está apuntado por SI=2004)
 AX=8033 BX=4002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=0111 NV UP EI PL NZ NA PO NC
 106D:0111 3D0000 CMP AX, 0000 (se ordena restar AX - 0 = 8034 - 0 para ver el valor de los flags)

- T ↴
 AX=8033 BX=4002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=0114 NV UP EI NG NZ NA PE NC
 106D:0114 7409 JZ 011F Con JZ se salta si Z=1; como fue NZ (Z=0 o sea el resultado de AX - 0 no fue cero) no se saltará

- T ↴
 AX=8033 BX=4002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=0116 NV UP EI NG NZ NA PE NC
 106D:0116 7C18 JL 0130 Con JL se salta si S≠V; como en AX - 0 fue NV (V=0) y NG (S=1) o sea S≠V se saltará a 0130
 - T ↴ (IP salta de 0116 a 0130 como ordena JZ 0130)
 AX=8033 BX=4002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=0130 NV UP EI NG NZ NA PE NC
 106D:0130 FEC2 INC DL
 ↓(DL pasa de 00 a 01 pues se encontró un negativo)
 AX=8033 BX=4002 CX=0002 DX=0101 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=0132 NV UP EI PL NZ NA PO NC
 106D:0132 8905 MOV [DI], AX (Se ordena guarda en memoria apuntada por DI el número negativo 8033 hallado)

- T ↴
 AX=8033 BX=4002 CX=0002 DX=0101 SP=FFEE BP=0000 SI=2004 DI=3000
 DS=106D ES=106D SS=106D CS=106D IP=0134 NV UP EI PL NZ NA PO NC
 106D:0134 83C702 ADD DI,+02 (Antes de ejecutar ADD DI, 02 se podía hacer E 3000 para ver si en 3000/I está 8033)

- T ↴
 AX=8033 BX=4002 CX=0002 DX=0101 SP=FFEE BP=0000 SI=2004 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0137 NV UP EI PL NZ NA PO NC
 106D:0137 EB66 JMP 011F (Se ordena saltar incondicionalmente a 011F)
 - T ↴ (IP salta de 0137 a 011F como ordena JMP 011F)
 AX=8033 BX=4002 CX=0002 DX=0101 SP=FFEE BP=0000 SI=2004 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=011F NV UP EI PL NZ NA PO NC
 106D:011F 83C602 ADD SI,+02
 ↓(SI pasó de 2004 a 2006)
 AX=8033 BX=4002 CX=0002 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0122 NV UP EI PL NZ NA PE NC
 106D:0122 FEC9 DEC CL (Se ordena hacer CL←CL - 1 ó sea 2 - 1)
 - T ↴ (CL pasa de 02 a 01 indicando que sólo queda un número sin analizar en la lista)
 AX=8033 BX=4002 CX=0001 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0124 NV UP EI PL NZ NA PO NC
 106D:0124 75E9 JNZ 010F NZ (Z=0) indica que CL - 1 = 2 - 1 no fue cero, y se saltará, pues JNZ ordena hacerlo si es Z=0
 - T ↴ (IP salta de 0124 a 010F como ordena JNZ 010F)
 AX=8033 BX=4002 CX=0001 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=010F NV UP EI PL NZ NA PO NC
 106D:010F 8B04 MOV AX, [SI] DS:2002-0042 (antes de ejecutar MOV AX, [SI] en memoria 2006/7 hay 0042)
 - T ↴ (De memoria pasó hacia AX una copia de 0042 que está apuntado en memoria por SI)
 AX=0042 BX=4002 CX=0001 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0111 NV UP EI PL NZ NA PE NC
 106D:0111 3D0000 CMP AX, 0000 (se ordena restar AX - 0 = 0042 - 0 para ver el valor de los flags)

- T ↴
 AX=0042 BX=4002 CX=0001 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0114 NV UP EI PL NZ NA PE NC
 106D:0114 7409 JZ 011F Con JZ se salta si Z=1; como fue NZ (Z=0 o sea el resultado de AX - 0 no fue cero) no se saltará

- T ↴
 AX=0042 BX=4002 CX=0001 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0116 NV UP EI PL NZ NA PE NC
 106D:0116 7C18 JL 0130 Con JL se salta si S≠V; como en AX - 0 fue NV (V=0) y PL (S=0) o sea S=V no se saltará a 0130
 - T ↴
 AX=0042 BX=4002 CX=0001 DX=0101 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=0118 NV UP EI PL NZ NA PE NC
 106D:0118 FEC6 INC DH
 ↓(DH pasa de 01 a 02 pues se encontró otro positivo)
 AX=0042 BX=4002 CX=0001 DX=0201 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=011A NV UP EI PL NZ NA PO NC
 106D:011A 8907 MOV [BX],AX
 - T ↴
 AX=0042 BX=4002 CX=0001 DX=0201 SP=FFEE BP=0000 SI=2006 DI=3002
 DS=106D ES=106D SS=106D CS=106D IP=011C NV UP EI PL NZ NA PO NC
 106D:011C 83C302 ADD BX,+02 (Antes de ejecutar ADD BX, 02 se podía hacer E 4002 para ver si en 4002/3 está 0042)

- T ↴ (BX pasa de 4002 a 4004)
AX=0042 BX=4004 CX=0001 DX=0201 SP=FFEE BP=0000 SI=2006 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=011F NV UP EI PL NZ NA PO NC
106D:011F 83C602 ADD SI,+02
(SI pasó de 2006 a 2008)

- T ↴
AX=0042 BX=4004 CX=0001 DX=0201 SP=FFEE BP=0000 SI=2008 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=0122 NV UP EI PL NZ NA PO NC
106D:0122 FEC9 DEC CL (Se ordena hacer CL ← CL - 1 ó sea 1 - 1 = 0)
(CL pasa de 01 a 00 indicando que ya no quedan números sin analizar en la lista)

- T ↴
AX=0042 BX=4004 CX=0000 DX=0201 SP=FFEE BP=0000 SI=2008 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=0124 NV UP EI PL ZR NA PE NC
106D:0124 75E9 JNZ 010F Con JNZ se salta si Z=0; como fue ZR (Z=1 o sea el resultado de CL - 1 fue cero) no se saltará

- T ↴
AX=0042 BX=4004 CX=0000 DX=0201 SP=FFEE BP=0000 SI=2008 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=0126 NV UP EI PL ZR NA PE NC
106D:0126 A30016 MOV [1600], AX

- T ↴
AX=0042 BX=4004 CX=0000 DX=0201 SP=FFEE BP=0000 SI=2008 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=0129 NV UP EI PL ZR NA PE NC
106D:0129 A30013 MOV [1300], AX

- T ↴
AX=0042 BX=4004 CX=0000 DX=0201 SP=FFEE BP=0000 SI=2008 DI=3002
DS=106D ES=106D SS=106D CS=106D IP=012C NV UP EI PL ZR NA PE NC
106D:012C CD20 INT 20

-E3000 ↴ Examina resultados: lista de negativos
xxxx:3000 33. 80.

-E 4000 ↴ Examina resultados: lista de positivos
xxxx:4000 25. 00. 42. 00.

-E 1600 ↴ Examina resultados: contador de negativos
xxxx:1600 01.

-E 1700 ↴ Examina resultados: contador de positivos
xxxx:1700 02.

Forma de copiar una pantalla del Debug en el Word

1. Cliquear con el botón derecho del mouse en la barra superior azul de la pantalla del DOS.
2. Cliquear con el botón izquierdo del mouse en Edit.
3. Cliquear con el botón izquierdo en Mark y aparecerá un punto luminoso en el rincón superior izquierdo.
4. Mover la flecha del mouse sobre dicho punto y ampliarlo con el botón izquierdo apretado, hasta abarcar la parte de la pantalla que se quiere copiar. Luego apretar Enter, con lo cual desaparecerá la parte iluminada de la pantalla.
5. Abrir un archivo en el Word (u otro) y pulsar Ctrl V para copiar lo seleccionado en el punto anterior.

Forma de ejecutar el programa de una sola vez mediante el comando G

-G = 0100 012E ↴ (También puede primero hacerse IP=0100 mediante RIP y luego hacer G = 0100 ↴)
AX=0042 BX=4002 CX=0000 DX=0102 SP=FFE6 BP=FFE6 SI=2008 DI=3004
DS=106D ES=106D SS=106D CS=106D IP=012E NV UP DI PL ZR NA PE NC
106D:012E CD20 INT 20

EJERCICIO 14 Con listas cuya cantidad n de caracteres que las conforman no es un dato directo.
En las direcciones 1000 y 2000 comienzan dos listas de caracteres ASCII de mayúsculas que corresponden a nombres de personas, las cuales pueden ser de igual o distinta longitud, siendo que cada lista termina con el carácter ASCII correspondiente al Enter (0D), como se exemplifica. Determinar en cuál de esas dos listas correspondientes a nombres, está el nombre que va primero por orden alfabetico, escribiendo en la dirección 3000 la dirección donde comienza dicha lista. Se supone que los nombres han sido bien tipeados, con caracteres ASCII de mayúsculas, y que terminan con Enter.

1000 41 (A) 2000 4D (M)	1000 41 (A) 2000 41 (A)	1000 41 (A) 2000 41 (A)
4E (N) 45 (A)	4E (N) 4C (L)	4E (N) 4E (N)
41 (A) 52 (R)	41 (A) 42 (B)	41 (A) 41 (A)
0D (J) 41 (A)	0D (J) 41 (A)	0D (J) 48 (H)
0D (J)	0D (J)	49 (I)
		0D (J)

Figura 3.26.a

Figura 3.26.b

Figura 3.26.c

Este ejercicio, por desconocerse de entrada la cantidad n de caracteres que componen cada lista, no empezará como otros anteriores con M0V CL, [...] y terminará con DEC CL y JNZ forma típica que usamos para las estructuras tipo FOR en alto nivel. En este caso el ciclo se repetirá mientras ("while") no se detecte un final de nombre (carácter 0D) o no bien se detecte que son diferentes los códigos ascii de dos letras que se están comparando.

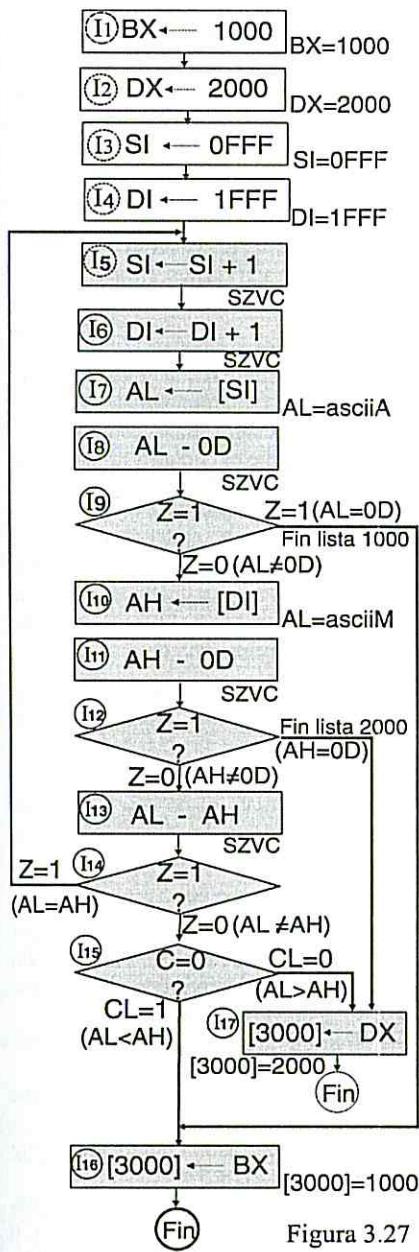


Figura 3.27

```

MOV BX, 1000
MOV DX, 2000
MOV SI, 0FFF
MOV DI, 1FFF
OTRO INC SI
INC DI
MOV AL, [SI]
CMP AL, 0D
JZ GAS1
MOV AH, [DI]
CMP AH, 0D
JZ GADI
CMP AL, AH
JZ OTRO
JA GADI
GASI MOV [3000],BX
INT 20
GADI MOV [3000],DX
INT 20

```

En el inicio del diagrama de la figura 3.27 en BX y DX quedan los números 1000 y 2000 que se usarán (uno u otro) para indicar al final del programa la dirección de la lista que es primera por orden alfabetico, como se pide.

Así en el caso de ANA y MARA deberá quedar en la dirección 3000 el valor 1000, indicando que la lista comenzada en 1000 es primera alfabeticamente. Luego (pasos 3 y 4), los punteros SI y DI de cada lista se inicializan con 0FFF y 1FFF, de modo que cuando en los pasos 5 y 6 se ordene SI ← SI + 1 y DI ← DI + 1, en la primer vuelta del procedimiento resulte:

SI = 0FFF + 1 = 1000 y DI = 1FFF + 1 = 2000. De este modo, en el caso a) de la figura 3.26 los registros SI y DI en la primera vuelta apuntarán a los códigos ASCII de las letras A y M que están en 1000 y 2000 en las listas dadas. Dichos códigos luego de los pasos 7 y 10 pasarán hacia AL y AH.

Para determinar si uno u otro de esos códigos corresponde o no al código ASCII 0D de la tecla Enter (↓), lo cual indicaría que es el fin de una u otra lista (o de ambas si tienen igual cantidad de letras), primero se ordena (pasos 8 y 11) las restas AL - 0D y AH - 0D, para luego (pasos 9 y 12), preguntar si Z=1, como en el ejercicio 9. En caso que sea Z=1 implica que AL = 0D, o que AH = 0D, o sea que una u otra lista ha terminado.

Si se asume que cada nombre tiene al menos una letra, en la primera vuelta dichas restas sólo pueden generar Z=0, pues las listas no pueden terminar con una sola vuelta.

Con la resta del paso 13 comienza el proceso para comparar dos letras que están en igual posición en ambas listas.

Si en el paso 14 se determina que Z=1 significa que esas dos letras comparadas son iguales (esto se supuso para la primera letra A en los casos b y c) de la figura 3.26, lo cual se detectaría en ambos casos en la primera vuelta. De ser Z=1 (figura 3.27) se ordena saltar al paso 5, con lo cual comienza una segunda vuelta donde nuevamente se comparan los dos códigos siguientes para detectar si alguno de ellos (o ambos) es de fin de lista, o bien si corresponden a letras distintas o iguales.

Si en la segunda vuelta las letras son iguales, como ocurre con ANA y ANAHI implica que hasta las segundas letras hay "empate" en el orden alfabetico pues dichas letras son N, por lo que se debe seguir con una tercer vuelta, en la que otra vez "empatarán". Enseguida se verá el paso siguiente. En la segunda vuelta de ANA y ALBA en el paso 13 la resta no dará cero, por lo que el paso 14 detectará Z=0.

Ello significa que las dos letras (N y L) no son iguales, lo cual implica que no será necesaria una tercera vuelta, y permite determinar en el paso 15, por ser C=0, que AL > AH (ver fig. 3.16), detectándose así que el código ASCII (de la N) que está en AL es mayor que el código (de la L) que está en AH.

Por lo tanto, si bien las dos primeras letras fueron iguales (A y A), las segundas no lo son, y por ser código N > código L es suficiente para determinar que ALBA va primero por orden alfabetico, por lo que en la dirección 3000 debe ir la dirección 2000 que aportará DX, donde comienza ALBA (paso 17); y así terminaría el programa para el caso b).

En el caso a) de ANA y MARA ya en el paso 14 de la primera vuelta resulta Z=0 (AL ≠ AH), y se sigue con el paso 15 que detectaría C=1 (AL < AH, o sea: código A < código B), por lo que ANA va primera por orden, y su dirección 1000 la provee BX para que se guarde en 3000 (paso 16).

Con ANAHI y ANA en el paso 14 de las tres primeras vueltas se detectará Z=1.

Pero en la cuarta vuelta, por ser 0D el cuarto código de la lista de ANA (figura 3.26.c), dicho código indica que ese nombre terminó, y en el paso 11 será: AH - 0D = 0D - 0D = 0, y por lo tanto Z=1. El paso 12 detectará Z=1 y ordenará saltar al paso 16, donde BX pasa a las celdas 3000/1 la dirección 1000 como indicación de que ANA va antes que ANAHI.

Si ambos nombres son idénticos, y por lo tanto de igual número de letras, en los pasos 8 y 9 de la vuelta en que primero se detecta el fin del nombre de la lista que empieza en 1000, se saltaría al paso 16 que ordena llevar 1000 (que está en BX) a la dirección 3000. Como las dos listas están en igual orden, no importa que la primera sea siempre la que está en 1000

Obsérvese que como inicialmente no se conoce el número n de caracteres que que componen cada lista, la secuencia no termina con DEC CL y JNZ.

EJERCICIO 15 Otro caso de una lista cuya cantidad n de caracteres que la componen no es un dato directo. Dado que al tipar caracteres puede haber errores, hacer un programa que valide (verifique) si los códigos ASCII de una lista que empieza en la dirección 1000 y termina con el código 0D de Enter, realmente corresponden a caracteres alfábéticos, ya sean mayúsculas o minúsculas indistintamente, y no a otros.

Asimismo contemplar la posibilidad que el carácter 0D (Enter) de fin de la lista esté errado o no esté, en cuyo caso controlar que un nombre pueda tener hasta 20d = 14h letras, pues caso contrario el programa podría no terminar nunca. Ya sea que se encuentre un código que no corresponda a una letra o que no esté el código 0D, escribir 0000h en direcciones 3000/I. Si la lista de letras se valida, escribir FFFF en esas direcciones. No se da la cantidad n de caracteres de la lista.

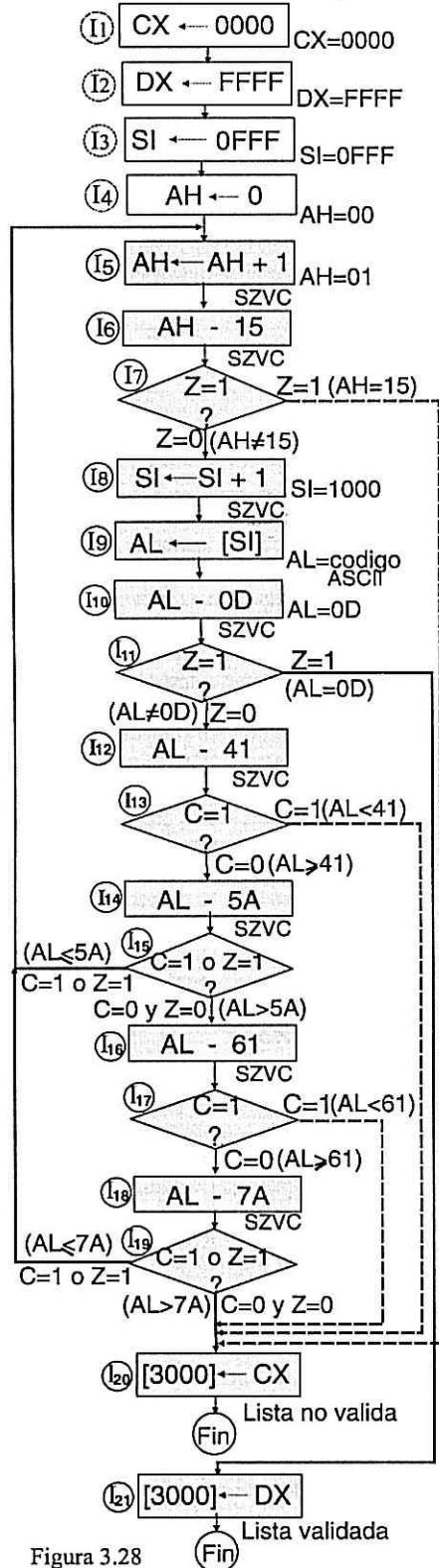


Figura 3.28

En el cuarto de los pasos de inicio (figura 3.28) se pone a cero el registro AH, que será usado para contar el número de códigos ASCII que han sido analizados, cuyo valor no puede superar 21d = 15h como se deducirá.

El procedimiento comienza (paso 5) incrementando dicho contador AH, pues en cada vuelta se empieza a analizar un nuevo código y se ordena la resta AH - 15 del paso 6 para constatar si AH = 15h = 21d, para el caso que ya se hayan analizado 20d códigos y no se detectó el fin de lista 0D.

Como en la primera vuelta es AH=1, en el paso 7 se detectará Z=0, o sea AH≠15, y mientras sea Z=0 se sigue en cada vuelta con SI ← SI+1 (paso 8) para que SI apunte al siguiente código que en esa vuelta corresponde analizar. Luego (paso 9) con AL ← [SI] se lleva hacia AL una copia del código ASCII apuntado por SI (SI=1000 en la primera vuelta).

Si a partir de la dirección 1000 se tiene 4C (L) 69 (i) 61 (a) 0D (J), o sea suponiendo el nombre Lia codificado en ASCII, en AL se tendría 4C en la primera vuelta.

En general, el código que está en AL puede ser:

- a) 0D de fin de nombre;
- b) de una mayúscula;
- c) de una minúscula;
- d) un código incorrecto, en cuyo caso se debe escribir 0000 en las direcciones de memoria 3000/I, según se pide en el enunciado.

Primero se verifica si el código que llegó a AL es 0D (Enter de final).

A tal fin el paso 10 ordena AL - 0D; y si en el paso 11 se detecta Z=1 por ser AL=0D, ello implica que en esta vuelta se detectó dicho código de fin de nombre, y que en las vueltas anteriores se validaron todos los códigos analizados como correspondientes a letras mayúsculas o minúsculas.

Por lo tanto si Z=1 se ordena saltar (figura 3.28) al paso 21 para escribir FFFF=DX en 3000/I y luego finalizar el programa.

En nuestro ejemplo por ser AL = 4C ≠ 0D se sigue con los pasos que determinan si el código es de una letra mayúscula o minúscula.

Si es una mayúscula, su codificación ASCII debe estar comprendida entre 41 (letra A) y 5A (letra Z); o sea debe ser 41 ≤ AL ≤ 5A.

Para saber si 41 ≤ AL el paso 12 ordena AL - 41, y en el paso 13 se pregunta si esa resta generó C=1, lo cual implicaría (fig 1.16) que AL < 41, o sea que es un código ASCII entre 00 y 40 que no es el de una mayúscula. Si bien entre 00 y 40 está 0D, este código no puede ser, pues para nuestro ejemplo antes se detectó AL ≠ 0D. Tampoco no es de una letra minúscula, dado que los códigos ASCII de éstas está comprendido entre 61 (letra a) y 7A (letra z), por lo que se trata de un código ASCII que no corresponde a ninguna letra. Entonces se ordena (figura 3.28) saltar al paso 20 para indicar en las direcciones 3000/I con 0000=CX que la lista no es válida.

Volviendo al paso 13, si en él se detecta que C=0 originado por la resta AL - 41, debe ser AL ≥ 41, (fig. 3.16) o sea 41 ≤ AL. Para determinar si el código que está en AL es el de una letra mayúscula (41 ≤ AL ≤ 5A) falta ver si AL ≤ 5A. Con este fin se ordena AL - 5A (paso 14) que en el ejemplo dado sería 4C - 5A. Si en el paso 15 se detecta que AL - 5A genera C=1 ó Z=1 implica AL ≤ 5A, determinándose que es una mayúscula. Esto sucede en el ejemplo dado, pues luego de hacer AL - 5A = 4C - 5A resultará C=1 y Z=0, siendo que 4C es el código de la letra mayúscula L. Por lo tanto se ordena saltar al paso 5 para analizar el código ASCII siguiente, que para el nombre Lia será 69 de la letra i. Así comenzará una segunda vuelta similar a la primera, salvo que en el paso 9 va hacia AL dicho código 69. Igualmente como en la primera vuelta, el procedimiento sigue por los pasos 10 a 15.

En el paso 14 será $AL - 5A = 69 - 5A$, por lo que en el paso 15 se detecta $C=0$ y $Z=0$, resultando $AL > 5A$, de donde se detectaría que en AL no se tiene el código de una mayúscula.

A fin de determinar si es una minúscula ($61 \leq AL \leq 7A$) se ordenan los pasos 16 al 19 semejantes a los pasos 12 al 15, salvo los valores límites. Si luego de hacer $AL - 61$ en el paso 17 se detecta que $C=1$, significa que $AL < 61$, o sea que el código que está en AL no es el de una minúscula. Como de los pasos 10 al 15 se determinó que AL no es 0D de fin de nombre, ni es el de una mayúscula, necesariamente es un código ASCII no permitido, por lo que se ordena saltar al paso 20 donde se escribe 0000 en 3000/1 (lista inválida).

Siendo en nuestro ejemplo $AL=69$ (letra i), al hacer $AL - 61 = 69 - 61$ resultará $C=0$ y $Z=0$, con lo cual $AL \geq 61$, (fig. 3.16) o sea $61 \leq AL$; y falta ver si $AL \leq 7A$. Con este fin se ordena $AL - 7A$ (paso 18) que en el ejemplo dado sería $69 - 7A$. Si el paso 19 detecta que $AL - 7A$ generó $C=1$ ó $Z=1$ implica $AL \leq 7A$, determinándose que es una minúscula. Esto sucede en el ejemplo dado, pues luego de hacer $AL - 7A = 69 - 7A$ resultará $C=1$ y $Z=0$, siendo que 69 es el código de la letra minúscula i, pudiéndose pasar a validar el código que sigue al 69 en la lista.

Por ello se ordena saltar al paso 5 para analizar el código ASCII siguiente, que para el nombre Lia será 61 de la letra a. Así comenzará una tercera vuelta muy similar a la segunda, salvo que en el paso 9 va hacia AL dicho código 61, que por ser el de una minúscula seguirá por los pasos 10 a 19, para volver saltar al paso 5, como sucedió en la vuelta anterior.

En la cuarta vuelta será $AH = 4$ (paso 5), y en el paso 9 hacia AL va el código 0D. En el paso 10 será $AL - 0D = 0D - 0D$ por lo que en el paso 11 se detectará $Z=1$, y se saltará al paso 21 para llevar FFFF = DX hacia 3000/1 (lista validada).

Los pasos 10 y 11 del inicio de la 4ta vuelta, se detectará que $AL = 0D$, y así se salta al paso 20 que valida la lista.

Si a partir de la cuarta vuelta y hasta la vuelta 20d = 14h no se encuentra el código 0D, al comenzar la vuelta 21d = 15h en el paso 7, por ser $Z=1$ (AH=21) se termina el programa invalidando la lista, aunque los códigos anteriores sean de letras.

	MOV CX, 0000 MOV DX, FFFF MOV AH, 00 MOV SI, 0FFF	Obsérvese nuevamente los beneficios de usar la instrucción CMP de resta sin asignación como $AL - XX$, que a lo largo de una secuencia, a partir del paso 9 permite restar a un mismo valor presente en AL distintos valores XX, sin que varíe el código ASCII que está en AL, para poder usar los flags con distintos fines, como indica la figura 3.28.
OTRO	INC AH CMP AH, 15 JZ ERRO INC SI MOV AL, [SI] CMP AL, 0D JZ VALE CMP AL, 41 JB ERRO CMP AL, 5A JBE OTRO CMP AL, 61 JB ERRO CMP AL, 7A JBE OTRO	En este ejercicio, como en el anterior, a diferencia de otros anteriores, por un lado no se conoce la cantidad de elementos que tienen las listas, por lo que no finalizan con DEC CL seguido de JNZ.
ERRO	MOV [3000],CX INT 20	En los ejercicios 14 y 15 el paso a una nueva vuelta depende de los valores de los elementos de una lista, como ser de AL, mientras que en los ejercicios anteriores este paso depende del número de elementos operados, o sea de CL. Asimismo en los ejercicios 14 y 15 los contadores y punteros se indexan al principio de cada vuelta, pues la ubicación de los pasos que ordenan indexar registros puede implicar variaciones en los valores de los flags, que afectarían a los flags resultantes de operaciones de resta sobre los códigos presentes por ejemplo en AL necesarios para el procedimiento que se está realizando.
	VALE MOV [3000],DX INT 20	

EJERCICIO 16 Intercambio de elementos entre dos listas, y variante con ahorro de un puntero Existen dos listas de igual cantidad de números enteros (2 bytes cada uno), que empiezan en las direcciones 2000 y 3000 respectivamente. La cantidad de números de ambas listas está en la dirección 1500. Comparar los números correspondientes, y dejar todos los elementos menores en una lista, y los mayores en otra.

Por ejemplo (situaciones inicial y final exemplificadas con números decimales):

2000 28 3000 12	2000 12 3000 28	(se intercambiaron los elementos correspondientes)
-5 10	-5 10	(no hubo que intercambiarlos)
35 -8	-8 35	(se intercambiaron los elementos correspondientes)

OTRO	MOV CL, [1500] MOV SI, 2000 MOV DI, 3000 MOV AX, [SI] MOV BX, [DI] CMP AX, BX JL DEJA MOV [DI], AX MOV [SI], BX	Carga en CL la cantidad de números 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 Resta los números correspondientes de ambas listas Si $AX < BX$ saltar a DEJA (los números donde están) Lleva a la lista apuntada por DI el número mayor Lleva a la lista apuntada por SI el número menor	OTRO	MOV CL, [1500] MOV SI, 2000 MOV AX, [SI] MOV BX, [SI + 1000] CMP AX, BX JL DEJA MOV [SI + 1000], AX MOV [SI], BX
DEJA	ADD DI,2 ADD SI,2 DEC CL JNZ OTRO INT 20	Incrementa DI en 2 Incrementa SI en 2 Decrementa CL Mientras $Z=0$, volver a OTRO Fin	DEJA	ADD SI,2 DEC CL JNZ OTRO INT 20

Este ahorro de un registro sólo puede ser hecho si las listas avanzan juntas.

EJERCICIO 17 Redondeo de números con parte entera (de longitud no limitada) y fraccionaria (un dígito)

A partir de la dirección 3000 están codificados en ASCII los dígitos $X_1 X_2 X_3 \dots X_n . Y$ de un número decimal con n dígitos X_i de parte entera y un solo dígito Y de parte fraccionaria, separadas por el código ASCII (2E) del punto (equivalente a una coma), como se exemplifica en la figura 3.29 para $N = 389,3$ y $N = 389,7$.

Desarrollar un programa de redondeo, para que el número cuyos dígitos conforman un vector en memoria, continúe guardado a partir de la dirección 3000, pero modificado por el redondeo, según se exemplifica, para que la parte fraccionaria sea cero, o sea para que el único dígito fraccionario Y valga cero.

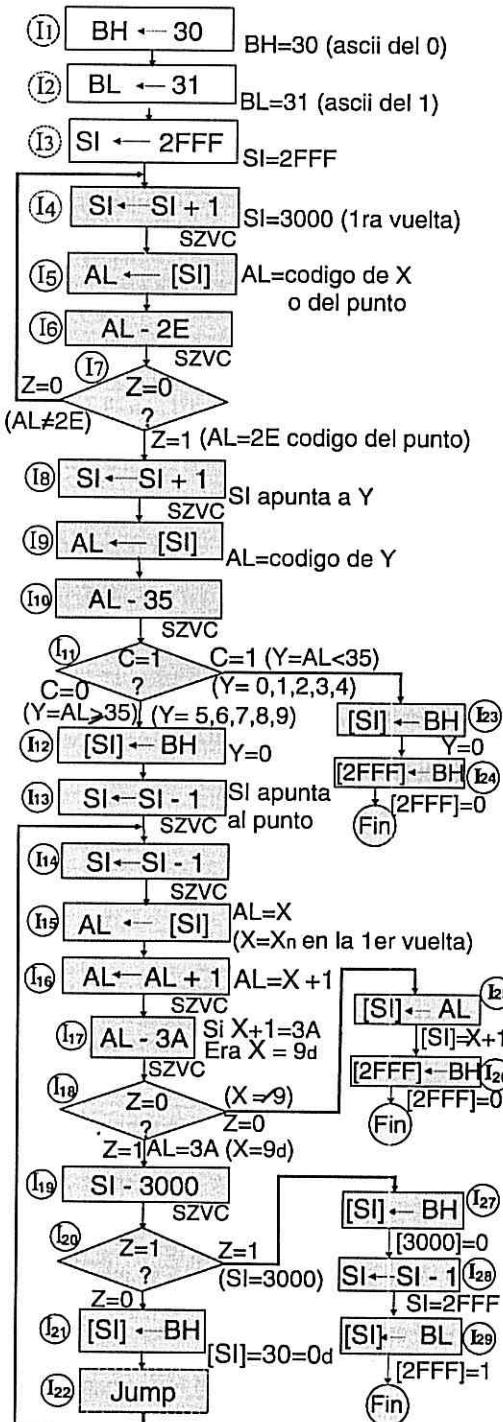


Figura 3.30

Para $N = 389,3$ (figura 3.29.a), por ser $Y=3$ (ASCII 33), por ser $Y < 5$. Entonces por ser $C=1$ el paso I11 ordenará saltar a I23 a fin de reemplazar (en la dirección 3004 apuntada por SI sin cambios desde I8) el código 33 de $Y = 3$ por el código 30 del dígito 0, quedando $Y = 0$.

Para finalizar, como pide el enunciado, en I24 se envía un cero a la dirección 3FFF, resultando $N = 0389,0$ (figura 3.29.a).

2FFF	XX	2FFF 00 (0)	2FFF XX	2FFF 00 (0)
3000	33 (3)	→ 3000 33 (3)	3000 33 (3)	→ 3000 33 (3)
3001	38 (8)	→ 3001 38 (8)	3001 38 (8)	→ 3001 38 (8)
3002	39 (9)	→ 3002 39 (9)	3002 32 (2)	→ 3002 33 (3)
3003	2E (.)	→ 3003 2E (.)	3003 2E (.)	→ 3003 2E (.)
3004	33 (3)	→ 3004 30 (0)	3004 37 (7)	→ 3004 30 (0)

Figura 3.29.a

Figuras 3.29.b

Si $0 \leq Y \leq 4$ para redondear no será necesario cambiar ningún dígito de la parte entera. Por ejemplo, todos los números entre 0,0 y 0,4 se redondean a 0,0; los que están entre 554329,0 y 554329,4 se redondean a 0554329,0 etc. con un 0 delante, por lo que se verá. Si $5 \leq Y \leq 9$ habrá que modificar alguno o todos dígitos de la parte entera a partir del X_n que está a la izquierda del punto.

Por ejemplo, todos los números entre 1849892,5 y 1849892,9 se redondean a 01849893,0; entre 0,5 y 0,9 se redondean a 01,0; entre 1849892,5 y 1849892,9 se redondean a 01849893,0; entre 554329,5 y 554329,9 se redondean a 0554330,0; etc.

En general si la parte entera del número termina en uno ó más 9s, los mismos deben convertirse en ceros y sumarle uno al siguiente dígito que está a la izquierda de ellos que sea distinto de 9, con lo cual terminaría el proceso de redondeo.

Continuando con el enunciado, si $5 \leq Y \leq 9$, y además si la parte entera son todos 9s, se deberá escribir un uno en la dirección 2FFF anterior a 3000, y un cero en los demás casos, como aparece en los ejemplos anteriores con cero adelante.

Por ejemplo los números entre 999,5 y 999,9 se redondearían a 1000,0 con el dígito uno adelante, en 2FFF. Como el primer 9 (X_1) de cada número a redondear de ese tipo estaría en la dirección 3000 (figuras 3.31.b y c), en esos casos y en todos los similares que se redondean a 1000 ... 00,0 el enunciado indica escribir en la dirección 2FFF el código ASCII 31 del dígito uno.

El diagrama lógico (figura 3.30) empieza inicializando BH y BL con el código ASCII 30 y 39 del 0 y 9 a utilizar; y a SI con 2FFF.

Como resultado del enunciado, no se indica el número n de dígitos enteros que conforman el número a redondear. Dado que el primer dígito entero (X_1) está en la dirección 3000, a partir del mismo primero se repite el lazo con los pasos I4 a I7, y en cada vuelta se detecta si el código que llega a AL mediante $AL \leftarrow [SI]$ (paso I5), es o no 2E (paso I6), correspondiente al punto.

Así, con el 389,Y (figura 3.29) se irán detectando sucesivamente los códigos 33, 38 y 39. Luego de n vueltas se habrán analizado los n dígitos $X_1 X_2 \dots X_n$ de la parte entera, y en la vuelta $n+1$ se detectará el código 2E del punto (en las figuras 3.29 con SI=3003). (Se supone que anteriormente se validaron los códigos ASCII del número a redondear, de una forma parecida a la del ejercicio 15).

Entonces, como luego del punto sigue (en 3004) el dígito Y fraccionario, con $SI \leftarrow SI + 1$ (paso I8) se apunta a este dígito, y otra vez de forma semejante a los pasos I4 a I7, con los pasos I8 a I11 ahora se detecta si $Y < 5$ (33) ó si $Y \geq 5$ usando el flag C (figura 3.16).

En el paso I10 la resta será $33 - 35$ generándose $C=1$, por lo cual en el paso I11 se saltará al paso I23 para reemplazar el código 33 de Y=3 por el código 30 del dígito 0, quedando Y=0.

Para $N = 382.7$ (figura 3.29.b), dado que $Y=7$ (ascii 37) el paso I10 de la resta será $37 - 35$ generando $C=0$, o sea $Y \geq 5$, por lo que el paso I11 ordenará seguir con $[SI] \leftarrow BH$ (I12), en el cual como en I23 se enviará 0 (30=BH) en la dirección (3004) de Y que sigue apuntada por SI, resultando $Y=0$; y hasta este punto del programa, pasaría a ser $N = 382.0$. Dado que $Y \geq 5$ para redondear $N = X_1X_2X_3 \dots X_n$. Y siempre se debe sumar uno al dígito entero X_n , que está a la izquierda del punto, cualquiera sea el valor de X_n ($0 \leq X_n \leq 9$), siendo $X_n = 2$ en 382.0 obtenido en I12. El dígito X_n (figuras 3.29 y 3.31) está siempre dos números hacia atrás de la dirección de Y , por lo que para localizar a X_n se debe restar 2 a SI que quedó apuntando a Y (SI en 3004 en este ejemplo). Esta resta se hará en dos pasos separados (I13 e I14) restando uno en cada paso, quedando en nuestro ejemplo SI=3002. Entonces, para $Y \geq 5$, luego de hacer $Y=0$ (en I12) siguen (I13 e I14) esas dos restas. Así SI apunta al dígito X_n , a fin de enviar otra vez una copia de X_n hacia AL (paso I15), para sumarle uno ($AL \leftarrow AL+1$) al código ascii del dígito X_n en I16. Para los dígitos X_n del 0 al 8 la suma $AL \leftarrow AL+1$ genera correctamente el código ascii del dígito decimal siguiente: $30 (0d) + 1 = 31 (1d)$; $31 (1d) + 1 = 32 (2d)$... $38 (8d) + 1 = 39 (9d)$. Volviendo a $N = 382.7$ que en el paso I12 pasó a ser 382.0, en I15 el código 32 del dígito $X_n = 2$ estaría en AL; y en I16 en el registro A se tendría $32 + 1 = 33$ (código ascii del 3d). Luego los pasos I17 e I18 (a tratar) permitirían determinar que $X_n \neq 9$, y por lo tanto I18 ordenaría saltar a I25, donde se ordena reemplazar (en la dirección 3002 apuntada por SI sin cambios desde I14) el código 32 de $X_n = 2$ por el código 33 que está en AL, quedando $X_n = 3$. Así 382.7 redondeado pasa a ser 383.0; y como en I26 se envía un cero a la dirección 3FFF, el redondeo finaliza con $N = 0383.0$ (figura 3.29.b).

Si hubiera sido $N = 0.5$ en I12 pasaría a ser 0.0, y en I16 por ser $X_n = 0$ (ascii 30) sería $30 + 1 = 31$. Como en I18 por ser $Z=0$ se detectará $X_n \neq 9$, se saltará a I25 a fin de que el contenido de $AL=31$ (ascii del 1) reemplace al código 30 (ascii del 0) en la dirección 3000 que sigue apuntada por SI desde I15. Así el 0.0 pasará a ser 1.0, y luego de I26 resultaría $N = 01.0$. Los mismos pasos sucederían con 3.9 ó con 394.6 que redondeados quedarían como 04.0 y 0395.0.

De ser $X_n = 9$, como ocurre con $N = 389.7$ que redondeado quedaría 0390.0, al sumarle uno al 9, el resultado 10 implica que en lugar del dígito $X_n = 9$ debe ir un cero, y que se debe sumar uno al dígito $X_{n-1} = 8$ que está a la izquierda de X_n . Si al ascii del 9, que es 39, se le suma uno en $AL \leftarrow AL+1$ (I16) resulta $39 + 1 = 3A$. Si bien 3A no es el ascii de ningún dígito, servirá para determinar si en AL (paso I15) estuvo el código 39 del 9, o no (códigos ascii del 0 al 8). Por tal motivo en I17 se ordena restar $AL - 3A$, que si genera $Z=1$ ($AL = 3A$) indirectamente implica que en el paso fue $AL = 39$, o sea $X_n = 9$, por lo que en el paso I18 se ordena seguir con I19.

Este paso (siempre para $Y \geq 5$) tiene en cuenta si la parte entera termina en uno o más 9s (como 389.7) o son todos nueve (9.6; 999....9.5 etc.), en cuyo caso el primero de la serie de nueve estaría en la dirección 3000.

	2FFF XX	2FFF 30 (0)	2FFF XX	2FFF 31 (1)	2FFF XX	2FFF 31 (1)
3000	33 (3) \rightarrow	3000 33 (3)	3000 39 (9) \rightarrow	3000 30 (0)	3000 39 (9) \rightarrow	3000 30 (0)
3001	38 (8) \rightarrow	3001 39 (9)	3001 2E (.)	3001 2E (.)	3001 39 (9) \rightarrow	3001 30 (0)
3002	39 (9) \rightarrow	3002 30 (0)	3002 36 (6) \rightarrow	3002 30 (0)	3002 2E (.)	3002 2E (.)
3003	2E (.)	3003 2E (.)			3003 35 (5) \rightarrow	3003 30 (0)
3004	37 (7) \rightarrow	3004 30 (0)				

Figura 3.31.a

Figura 3.31.b

Figura 3.31.c

Retornando a $N = 389.7$ (figura 3.31.a) el 9 estaría en la dirección 3002, y en el paso I12 se tendría 389.0. En I15 una copia del código 39 del 9 que está en 3002 pasa a AL; y en I16 se tendría $39 + 1 = 3A$. Luego los pasos I17 e I18 permitirían establecer que $X_n = 9$, y por lo tanto I18 ordenaría seguir con I19. En este paso se hace $SI - 3000$ para poder determinar si el 9 que se detectó mediante I17 e I18 es o no el primer dígito que está en 3000 (en 389.7 dicho dígito es 3). Despues del paso I15 sigue SI en 3002, por lo que en I19 es $SI - 3000 = 3002 - 3000 \neq 0$ generándose $Z=0$.

En consecuencia en I20 se ordenará seguir con I21 por que el 9 antes detectado (con $SI=3002$) no está en 3000, o sea que hay otro dígito a la izquierda de este 9, el cual se debe reemplazar por un cero, y sumarle uno al dígito X_{n-1} (en este caso 8) que está a su izquierda. En I21 se reemplaza en 3002 (apuntada por SI) el código 39 de este 9 por el código 30 = BH del dígito 0, resultando así 380.0 en reemplazo de 389.0 obtenido en el paso I12.

Sigue I22 para saltar incondicionalmente a I14 para que sea $SI = 3002 - 1 = 3001$ apuntando al código 38 del dígito 8; y luego de los pasos I15 e I16 resulta $AL = 38 + 1 = 39$ (ascii del 9). En I17 la resta $39 - 3A$ arrojará $Z = 0$ (lo cual implica que el dígito X (en este caso 8) apuntado por SI=3001 no es 9. Por ser $Z=0$ en I18 se saltará a I25 para reemplazar en 3001 (apuntada por SI) el código 38 del 8 por el código 39 del 9 que está en AL, resultando así 390.0 en reemplazo de 380.0 obtenido anteriormente en el paso I21. Para finalizar, como pide el enunciado, en I26 se envía un cero a la dirección 3FFF, finalizando el redondeo con $N = 0390.0$ (figura 3.31.a).

Suponiendo $N = 9.6$ (figura 3.31.b), el 9 estaría en la dirección 3000, y después del paso I15 el código 39 del 9 estaría en AL (siendo $SI=3000$); y en I16 se tendría $39 + 1 = 3A$. Luego los pasos I17 e I18 permitirían detectar que en I15 fue $AL=39$ (o sea $X_n = 9$ en 3000), por lo cual se debe seguir con I19 a fin de que se efectúe $SI - 3000 = 3000 - 3000$ generándose $Z=1$. Por consiguiente I20 ordenará saltar a I27 para hacer $[SI] \leftarrow BH = 30$, y así el código 30 del 0 reemplaza al código 39 del 9 en la dirección 3000 apuntada por SI, quedando 0.0 en vez de 9.0 (obtenido después del paso I12). En el paso I28 se hace $SI = SI - 1 = 3000 - 1 = 2FFF$; y se finaliza con I29 en el cual en la dirección 2FFF apuntada por SI se escribe $BL = 31$ (ascii del 1) en lugar del valor XX existente (figura 3.31.b), resultando en definitiva el número 10.0.

De haber sido $N = 99.5$ (figura 3.31.c), como con $N = 9.6$, los pasos I17 e I18 permitirían detectar que en I15 fue $AL=39$ (o sea $X_n = 9$ pero ahora en 3001), por lo cual se debe seguir con I19 a fin de que se efectúe $SI - 3000 = 3001 - 3000$ generándose $Z=0$. Por lo tanto I20 ordenará seguir con I21 para hacer $[SI] \leftarrow BH = 30$, y así el código 30 del 0 reemplaza

al código 39 del 9 en la dirección 3001 apuntada por SI, quedando 90.0 en vez de 99.0 (obtenido después del paso I12). El paso siguiente I22 ordena saltar incondicionalmente (jump) a I14 para volver a hacer $SI = SI - 1 = 3001 - 1$ resultando $SI=3000$ apuntando así (figura 3.31.c) al código 39 del 9 que está en 3000, y una copia del 39 irá hacia AL en I15. Al concretarse los pasos I16 a I18 se detectará que en el paso I15 estuvo el código 39 de un 9 en AL; y siendo ahora $SI=3000$ en el paso I19 la resta $SI - 3000$ será $3000 - 3000 = 0$ y $Z=1$. Como ocurrió para $N = 9.6$ con el 9 que estaba en 3000 en el paso I20 se ordenará saltar a la secuencia I27 a I29 antes vista, luego de la cual en lugar del 90.0 (obtenido después del paso I21) sucesivamente se formarán 00.0 y 100.0 que es el redondeo de 99.5.

Lo mismo puede generalizarse para cualquier número cuya parte entera está formada sólo por n nueves, siendo $Y \geq 5$. Del diagrama de la figura 3.30 resulta la siguiente codificación en Assembler:

MOV BH, 30	Queda en BH el código 30 del dígito 0
MOV BL, 31	Queda en BL el código 31 del dígito 1
MOV SI, 2FFF	Inicializa SI con la dirección 2FFF anterior a 3000
OTRO INC SI	Suma uno a SI ($SI=3000$ en la primer vuelta)
MOV AL, [SI]	El código del dígito X que apunta SI ($X1$ en la primera vuelta) pasa a AL
CMP AL, 2E	Resta $AL - 2E$ (código ascii del punto)
JNZ OTRO	Mientras código que está en AL $\neq 2E$ repetir desde OTRO hasta localizar el punto.
INC SI	Si en el paso anterior fue $Z=1$ ($AL=2E$) aumenta uno a SI para localizar dígito Y
MOV AL, [SI]	El código del dígito Y que apunta SI pasa a AL.
CMP AL, 35	Resta $AL - 35$ (código ascii de Y menos código ascii del 5)
JB RED1	Si ($C=1$) código ascii de Y es menor que 35 ($0 \leq Y \leq 5$) saltar a RED1
MOV [SI], BH	Si ($C=0$) código ascii de Y ≥ 35 ($Y \geq 5$) que quede $Y=0$ en lugar del valor de Y.
DEC SI	Resta uno a SI, así apunta a 2E, código ascii del punto que está a la izquierda de Y
DIZQ DEC SI	Otra vez resta uno a SI, así apunta al dígito Xn que está a la izquierda del punto.
MOV AL, [SI]	El código del dígito Xn pasa a AL en la 1er vuelta del loop que se inicia en DIZQ
INC AL	Suma uno al código ascii del dígito que está en AL
CMP AL, 3A	Resta $AL - 3A$ ($3A = 39 + 1$, siendo 39 el código ascii del 9)
JNZ RED2	Si $Z=0$ va a RED2 pues el código analizado no es el del 9
CMP SI, 3000	Si $Z=1$ el código analizado es del 9, y con $SI - 3000$ se detecta si el 9 está en 3000
JZ RED3	Si en $SI - 3000$ fue $Z=1$ saltar a RED3, pues está el ascii del 9 en 3000 ($X1=9$)
MOV [SI], BH	Si en $SI - 3000$ fue $Z=0$ va 0 en vez de ese 9, y hay otro dígito a su izquierda
JMP DIZQ	Saltar a DIZQ pues hay otro dígito a la izquierda del último 9 hallado.
RED1 MOV [SI], BH	Queda $Y=0$ cualquiera sea el valor anterior de Y, pues va en su lugar $BH=30$ (0)
MOV [2FFF], BH	En 2FFF va el ascii 30=BH del dígito 0, para que el redondeo quede $0X1X2...Xn.0$
INT 20	Fin
RED2 MOV [SI], AL	Queda en el lugar del dígito Xn el valor anterior que tenía su código ascii más uno
MOV [2FFF], BH	En 2FFF va el ascii 30=BH del dígito 0, para que el redondeo quede $0X1X2...Xn.0$
INT 20	Fin
RED3 MOV [SI], BH	Queda en la dirección 3000 el código 30=BH del 0 en lugar del código 39 del 9
DEC SI	Resta uno a SI
MOV [2FFF], BL	En 2FFF va el ascii 31=BL del dígito 1, para que el redondeo quede $1X1X2...Xn.0$
INT 20	Fin

EJERCICIO 18¹ Técnicas para repetir o no la ejecución de secuencias mediante indicadores de pasada

Se tiene una lista de números enteros cuya dirección inicial es 1000, siendo que la cantidad n de números que la conforman está en la dirección 1500. Copiar en otra lista que empieza en 2000 los números comprendidos entre 20 y 40; y para cada uno de éstos escribir debajo de su valor el número de orden que tienen en la lista originaria. Si en ésta aparecen repetidos consecutivamente números iguales que están entre 20h y 40h, escribirlos una sola vez en la nueva lista, con el número de orden del primero de los números repetidos.

En la inicialización se hace $DL=0$. De acá en adelante éste va a ser el valor de DL mientras no se encuentre un número N de la lista que esté comprendido entre 20 y 40 ($20 \leq N \leq 40$).

Con este fin, en el diagrama lógico de la figura 3.31 en el paso 5 se envía cada número N apuntado por SI hacia AX ($AX=N$); y en el paso 6 se aumenta uno el contador DH que va enumerando ordenadamente los n números de la lista. Su valor se usará cuando se escriba en la otra lista apuntada por DI el número de orden de un número N cuyo valor corresponde copiar en ella.

El paso 7 ordena la resta $AX - 20$ para que tomen valor los flags SZVC. Si en el paso 8 se detecta $S \neq V$, ($AX=N < 20$) hay que saltar al paso 18 donde se hace $DL=0$, esta vez para indicar que si por ejemplo es $N=15 < 20$ no estará entre 20 y 40. Por consiguiente el paso 19 ordena saltar incondicionalmente (jump) hacia el paso 15, que junto con el 16 y 17 permiten volver al paso 5 para pedir el número N que sigue en la lista, mientras sea $Z=0$ ($CL \neq 0$).

En caso que en el paso 8 se detecte $S=V$ ($AX=N \geq 20$), el número N puede llegar a estar entre 20 y 40.

¹ Este programa fue desarrollado por la Lic. Graciela Silvia D'Agostino, profesora de la UTN Regional Haedo

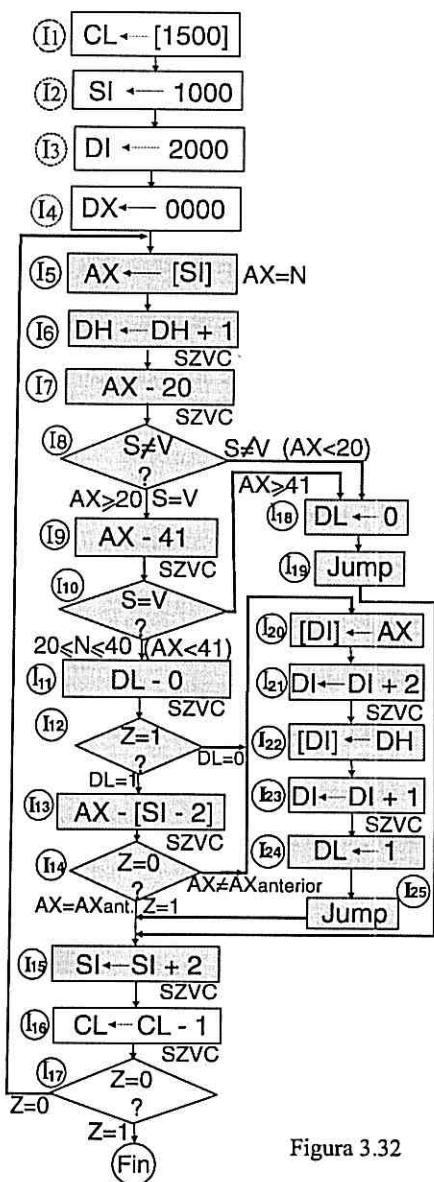


Figura 3.32

A la izquierda comienza la codificación en Assembler relativa a la figura 3.32

que idos lista 40h, nero AX valor : 20) 20 y rmi-	OTRO MOV CL, [1500] MOV SI, 1000 MOV DI, 2000 MOV DX, 0 OTRO MOV AX, [SI] INC DH CMP AX, 20 JL NO_ES CMP AX, 41 JG NO_ES CMP DL, 0 JZ ESCR CMP AX, [SI-2] JNZ ESCR ALFA ADD SI, 2 DEC CL JNZ OTRO INT 20
--	---

NO ES	MOV DL, 0 JMP ALFA
-------	-------------------------------------

que idos lista 40h, nero AX valor : 20) 20 y rmi-	Pone en cero a DL (indicador del paso por una secuencia) y DH (contador de orden) Pasa hacia AX una copia de un número N de la lista apuntada por SI Cada vez que se toma un número nuevo de la lista se suma uno al contador de orden DH Resta AX - 20 para que SZVC tomen valores Si S≠V resulta AX < 20 entonces saltar a NO_ES Resta AX - 40 para que SZVC tomen valores Si S=V resulta AX ≥ 41 (o sea AX > 40) entonces saltar a NO_ES Se hace DL = 0 si S≠V, pues AX < 41 (o sea AX ≤ 40, y hasta acá será 20 ≤ AX ≤ 40) Si Z=1 es DL=0 (siempre DL=0 en la 1er vuelta) se puede copiar N en la otra lista Si Z=0 es DL=1 (ya se escribió en la otra lista) hay que ver si AXactual = AXanterior Si Z=0 (AXactual ≠ AXanterior) no se repiten nros iguales. Copiar AXactual en otra lista Si Z=1 (AXactual = AXanterior) suma 2 a SI para apuntar siguiente número de la lista Resta uno a la cantidad de números que falta analizar en la lista apuntada por SI Mientras sea Z=0 saltar a OTRO para considerar el número siguiente de la lista
--	--

NO ES	Pone en 0 a DL pues N no está entre 20 y 40 Saltar incondicionalmente a ALFA
-------	---

(continúa en la siguiente página)

Para ver si $N \leq 40$ en el paso 9 se hace $AX - 41$ para que tomen valor los flags SZVC. Si en el paso 10 se detecta $S=V$ (por ejemplo si $N = 55$), es $AX=N \geq 41$, y por lo tanto $N > 40$, con lo cual también se ordena saltar al paso 18, que hace $DL=0$, y los pasos siguientes permiten llevar hacia AX el N siguiente.

Pero si en el paso 10 se detecta $S \neq V$ debe ser $AX=N < 41$, y por consiguiente $N \leq 40$ (por ejemplo si $N=35$); y de los pasos 8 al 10 resulta que $20 \leq N \leq 40$. Entonces sigue el paso 11 con la resta $DL - 0$ ($0 - 0 = 0$) para que tomen valor los flags SZVC. Si $Z=1$ implica que $DL=0$, lo cual significa que el N analizado antes de este N , no estuvo entre 20 y 40 (o se trata del primer N que está en la dirección 1000), y que en cambio este N (supuesto 35) efectivamente está entre 20 y 40.

Entonces, si $Z=1$ se ordena saltar al paso 20, a fin de escribir una copia de AX (con $N=35$) en la dirección de la otra lista apuntada por DI. Como cada N ocupa 2 bytes se le suma 2 a DI (paso 21), para poder escribir en esa lista, a continuación de N , el número de orden que tiene N , indicado por DH (paso 22); y como el valor de DH ocupa un byte, ahora se le suma uno a DI (paso 23), a fin de que quede apuntando a la siguiente celda libre, por si el N siguiente es distinto del actual (35) pero está entre 20 y 40. Luego (paso 24), se hace $DL=1$ para indicar que en la lista apuntada por DI se escribió el valor y el número de orden de un número (35) que estaba entre 20 y 40, a fin de que si el que le sigue en la lista tiene su mismo valor (35) no se escriba nada en dicha lista, pues el ejercicio requiere que si hay números consecutivos iguales que están entre 20 y 40, sólo se escriba el valor y orden del primero.

Habiendo realizado los pasos relativos al N analizado, el paso 25 ordena saltar incondicionalmente al paso 15, que junto con el 16 y 17 permiten volver al paso 5 para pedir el número N que sigue en la lista, mientras sea $Z=0$ ($CL \neq 0$).

Suponiendo que este N tenga igual valor (35) que el anterior, entonces otra vez se pasará por los pasos 5 al 12. Ahora el paso 11 será:

$DL - 0 = 1 - 0$ (pues en el paso 24 quedó $DL=1$) resultando $Z=0$. Por consiguiente ahora en el paso 12 se ordenará seguir con el paso 13: $AX - [SI-2]$ para que cambien los valores de SZVC a fin de determinar si el N actual apuntado por SI (que está en AX) es igual o no al N anterior, que sigue en la lista apuntada por SI dos direcciones atrás (SI-2).

Si en el paso 14 se detecta $Z=1$ (significa que ambos números N son iguales) se ordena seguir con el paso 15 y los que le siguen para pedir y analizar el nuevo N que sigue en la lista que apunta SI.

Por lo tanto, cuando $DL=1$ si se trata de un número igual al anterior que estuvo entre 20 y 40 no se salta al paso 20 para escribir su valor y número de orden. Esto mismo ocurrirá mientras se tenga k números consecutivos de ese valor (35) en la lista originaria. (Continúa luego del programa)

ESCR	MOV [DI], AX	Copia en la lista apuntada por DI el número N comprendido entre 20 y 40
	ADD DI, 2	Suma 2 a DI para preparar debajo escritura del número de orden de N debajo de su valor
	MOV [DI], DH	Escribe en la dirección apuntada por DI una copia del número de orden que indica DH
	INC DI	Se actualiza DI por si en AX llega otro N comprendido entre 20 y 40
	MOV DL, 1	DL=1 indica que ya se escribió en la lista apuntada por DI, por si el siguiente N es igual
	JMP ALFA	Salta incondicionalmente a ALFA

(Continuación de la explicación anterior) Si luego del primer N de valor 35, o luego de k repeticiones del mismo, el siguiente N apuntado por SI que llegó a AX es distinto (por ejemplo 29), pero está comprendido entre 20 y 40, se pasará nuevamente por los pasos 5 al 14 (y seguirá siendo DL=1); pero en el paso 14 resultará Z=0 por ser distintos el N actual y el anterior; y en este paso se ordenará saltar al paso 20 para escribir el valor y el número de orden del nuevo N que también está entre 20 y 40. Asimismo en el paso 24 se ratifica que seguirá siendo DL=1, hasta que en AX llegue un N que no esté comprendido entre esos valores, que hará DL=0.

En definitiva con DL=1 cuando en la lista apuntada por DI hay números consecutivos iguales que están entre 20 y 40, se garantiza que mientras ello suceda sólo se escribe el valor y número de orden del primero de ellos, o sea que se pasará una sola vez por la secuencia que permite dicha escritura.

EJERCICIO 19:

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 2000.

	MOV CL, [2000]	Carga en CL la cantidad de número inicial de la lista
	MOV BH, [1500]	Carga en BH el número en cuestión
	MOV SI, 2001	Inicializa SI para que apunte al comienzo de la lista
OTRO	CMP BH, [SI]	Compara el número que está en 1500 con uno de la lista
	JZ FINE	Si Z=1 (los números comparados son iguales), saltar a FINE
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z=0, volver a OTRO
	MOV [SI], BH	Si Z=1 el número no está en la lista, por lo que irá al final de ella
	MOV CL, [2000]	Carga nuevamente en CL la longitud de la lista
	INC CL	Incrementa la cantidad de números
	MOV [2000], CL	Guarda en 2000 la nueva longitud de la lista
	FINE INT 20	Fin

EJERCICIO 20:

Se tiene una lista de números de un byte que empieza en la dirección 2001, siendo que su cantidad está en la dirección 2000. Determinar si el número contenido en la dirección 1500 se halla en la lista que comienza en la dirección 2001, siendo que la cantidad de números de la lista está en la dirección 2000.

En caso que el número que está en 1500 sea uno 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.

	MOV CL, [2000]	Carga en CL la cantidad de números de la lista
	MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
	MOV AL, [1500]	Carga en AL el número a contrastar contenido en 1500
OTRO	CMP AL, [SI]	Resta al número que está en AL el número apuntado por SI
	JZ ALFA	Si son iguales (el número está en la lista) saltar a ALFA
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z=0, volver a OTRO
	INT 20	Fin
ALFA	MOV BL, [2000]	Carga en BL la cantidad de números de la lista
	DEC BL	Decrementa la cantidad de números de la lista
	MOV [2000], BL	Guarda la nueva cantidad de números de la lista
	DEC CL	Decrementar CL que lleva la cuenta de los números que faltan analizar
	JZ FINE	Si es el último elemento de la lista saltar a FINE
POST	MOV AL, [SI+1]	Si no es el último, cargar en AL el elemento siguiente
	MOV [SI], AL	Guarda en el lugar de un elemento, el que le sigue
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ POST	Mientras Z=0, volver a POST para leer el número siguiente
	FINE INT 20	Fin

EJERCICIO 21:

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 cantidad de números. 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.

	MOV CL, [2000]	Carga en CL la cantidad de números a analizar
	MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
	MOV AL, [1500]	Carga en AL el número a intercalar
OTRO	CMP AL, [SI]	Resta al contenido de AL, el número apuntado por SI
	JA PONE	Si el número de la lista es mayor hay que intercalar, saltar a PONE
	JZ FINE	Si son iguales (está en la lista) saltar a FINE
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ 0107	Mientras Z=0, volver a OTRO
FINE	INT 20	Fin
PONE	INC CL	Incrementa CL, pues hay un elemento más
ABAJ	MOV AH, [SI]	Salva el número que apuntaba SI en AH
	MOV [SI], AL	Guarda el número a intercalar (u otro a “bajar”) donde apunta SI
	MOV AL, AH	Lleva a AL el próximo número a “bajar”
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ ABAJ	Mientras Z=0, volver a ABAJ
	MOV BL, [2000]	Carga en BL la longitud de la lista
	INC BL	Incrementa BL
	MOV [2000], BL	Guarda en 2000 la cantidad de números incrementada
	INT 20	Fin

EJERCICIO 22

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.

	MOV SI, 4000	El registro SI apunta a la dirección 4000
	MOV DI, 2000	DI apunta al comienzo de la lista de direcciones
	MOV AL, FF	AL se carga con 8 “unos”
OTRA	MOV [SI], AL	El contenido de AL pasa a la posición apuntada por SI
	INC SI	Incrementa SI
	CMP SI, 5001	Resta 5001 (una dirección más que el límite 5000) al valor de SI
	JNZ OTRA	Mientras SI no sea 5001 volver a 0108
	MOV SI, 4000	El registro SI apunta otra vez a la dirección 4000 para verificar las escrituras
PROX	MOV AL, [SI]	El contenido de la posición apuntada por SI pasa a AL
	CMP AL, FF	Resta FF al valor de AL
	JZ BIEN	Si Z=1, ir a BIEN
	MOV [DI], SI	Guarda en otra lista la dirección (dada por SI) de una celda que no tiene 8 “unos”
BIEN	ADD DI, 2	Suma 2 a DI (las direcciones ocupan 2 bytes)
	INC SI	Incrementa SI
	CMP SI, 5001	Resta 5001 (una dirección más que el límite 5000) al valor de SI
	JNZ PROX	Mientras SI no sea 5001 volver a PROX
	INT 20	Fin

EJERCICIO 23

Se tiene una lista de números enteros de **un** byte cada uno, que empieza en la dirección 2000, siendo que la cantidad de los mismos 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).

	MOV CL, [1500]	Carga en CL la cantidad de números de la lista
	MOV SI, 2000	El registro SI apunta al comienzo de la lista de listas
	MOV DI, 3000	DI apunta al comienzo de la lista de resultados
OTRO	MOV AL, [SI]	Carga en AL un dato de la lista apuntada por SI
	CMP AL, 0	Compara el dato en AL contra 0
	JGE POSI	Si es mayor o igual que cero (positivo) salta a PTVO
	MOV AH, FF	Como es menor que cero (negativo), carga 8 unos en AH para propagar signo
ESCR	MOV [DI], AX	Carga en la lista apuntada por DI el nuevo número de 16 bits
	INC SI	Incrementa SI
	ADD DI, 2	Suma 2 a DI (los números resultantes ocupan 2 bytes)
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z=0, volver a OTRO
	INT 20	Fin
PTVO	MOV AH, 00	Carga 8 ceros en AH para propagar signo
	JMP ESCR	Salta a ESCR

EJERCICIO 24:

Se tiene una lista de caracteres ASCII que empieza en 2000, dentro de la cual se tiene un mensaje de **n** caracteres entre el carácter 02 = STX (start of text), y el carácter 03 = ETX (end of text) de dicha lista. Hallar el número **n** de caracteres del mensaje e indicarlo en la dirección 1500.

	MOV SI, 1FFF	Registro SI apunta al comienzo de la lista menos Uno
OTRO	INC SI	Incrementa SI
	MOV AL, 02	Carga 02 = STX en AL
	CMP AL, [SI]	Compara AL con el número apuntado por SI.
	JNZ OTRO	Si el carácter de la lista es distinto de STX, saltar a OTRO
	MOV CX, -1	Pone en -1 al contador
	MOV AL, 03	Carga 03 = ETX en AL
	INC CX	Incrementa CX
PROX	INC SI	Incrementa SI
	CMP AL, [SI]	Compara AL con el número apuntado por SI.
	JNZ PROX	Mientras Z=0, volver a PROX
	MOV [1500], CX	Guarda en 1500 el resultado
	INT 20	Fin

EJERCICIO 25

EJERCICIO 25
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	2000	05		2000	03	2000	03	
2001	34	(4)	2001	34	2001	39	(9)	2001	39
2002	33	(3)	2002	33	2002	36	(6) ==>	2002	36
2003	2E	(.) ==>	2003	2E	2003	38	(8)	2003	38
2004	35	(5)	2004	20	(SP)				
2005	30	(0)	2005	20	(SP)				
Antes		Después		Antes		Después			

MOV CL, [2000]	Carga en CL la cantidad de caracteres de la lista
MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
MOV AL, 2E	Carga 2E (ascii del punto) en AL
MOV BL, 20	Carga 20 (ascii del espacio en blanco) en BL
OTRO CMP AL, [SI]	Resta a AL el número apuntado por SI
JZ FRAC	Si son iguales (está el punto) saltar a FRAC
INC SI	Incrementa SI
DEC CL	Decrementa CL
JNZ OTRO	Mientras Z=0, volver a OTRO para ver si se detecta el código ascii del punto
INT 20	Finaliza por que el número es entero

(continúa en la hoja siguiente la secuencia para reemplazar los dos dígitos fraccionarios por espacios en blanco)

FRAC INC SI	Incrementa SI
MOV [SI], BL	El contenido de BL (20) se guarda donde apunta SI
DEC CL	Decrementa CL
JNZ FRAC	Mientras Z=0, volver a FRAC
INT 20	Finaliza luego de haber reemplazado los 2 dígitos fraccionarios con 20=SP

Los tres ejercicios siguientes tienen como objetivo relacionar la programación en alto nivel con su traducción en bajo nivel, en este caso con diagramas lógicos correspondientes a estructuras típicas en alto nivel; y también permiten sistematizar la correspondencia entre variables y direcciones de memoria

EJERCICIO 26:

Dada la siguiente codificación en un lenguaje de alto nivel:

INTEGERS

IF $20 \leq N \leq 40$ THEN $Q = N$
ELSE $Q = 0$

Suponiendo que sea traducida por un traductor tipo Compilador, que usa las direcciones 1000 y 2000 para el tipo de variables definidas:

- Indicar para el tipo de variables definidas, cómo aparecen en memoria
- Con vistas a traducir la codificación planteada en alto nivel a bajo nivel, en un diagrama lógico, del tipo usado en ejercicios anteriores, expresar lo que dicha codificación planteada ordena.
- Construir el diagrama lógico

- Ver figura 3.33.a
- En las direcciones 3000/1 de memoria se tiene un número entero N. Si $20 \leq N \leq 40$ escribir una copia de N en las direcciones 4000/1; caso contrario escribir un cero en estas direcciones
- Ver figura 3.33.c

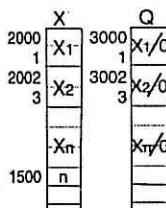


Figura 3.34.a

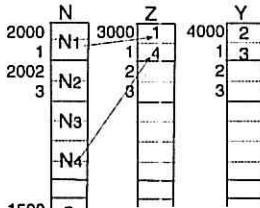


Figura 3.35.a

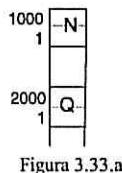


Figura 3.33.a

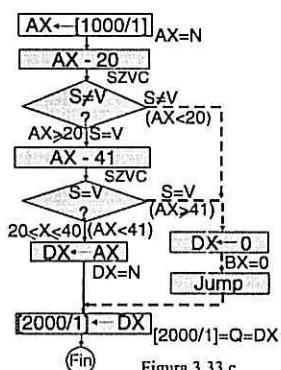


Figura 3.33.c

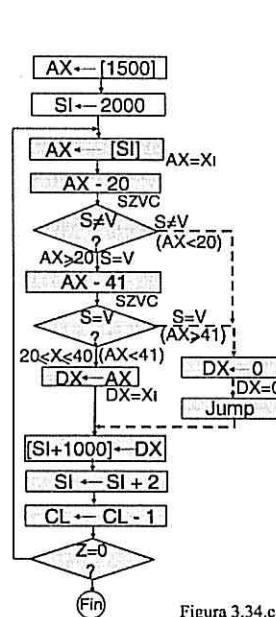


Figura 3.34.c

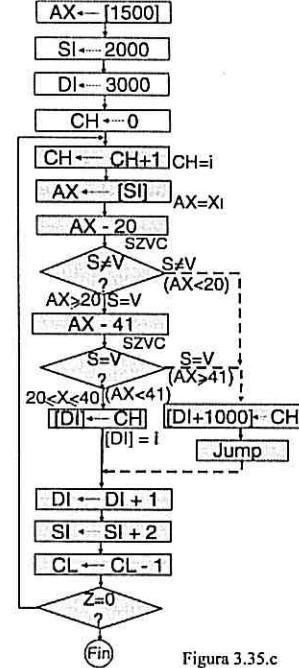


Figura 3.35.c

EJERCICIO 27:

Dada la siguiente codificación en un lenguaje de alto nivel:

FOR i = 1 to n

IF $20 \leq X \leq 40$

THEN $Q[i] = X[i]$

ELSE $Q[i] = 0$

Suponiendo que sea traducida por un traductor tipo Compilador, que usa las direcciones 1000 y 2000 donde empiezan el tipo de variables definidas, y la dirección 1500 para guardar la cantidad n de números:

- Indicar para el tipo de variables definidas, cómo quedan en memoria
 - Con vistas a traducir la codificación planteada en alto nivel a bajo nivel, en un diagrama lógico, del tipo usado en ejercicios anteriores, expresar lo que dicha codificación planteada ordena.
 - Construir el diagrama lógico.
- Ver figura 3.34.a
 - Se tiene una vector de enteros $X[i]$ que empieza en 2000 conformada por n números, siendo que n está en la dirección 1500. Analizar cada número X del vector: si $20 \leq X \leq 40$ copiar su valor X en otra lista que empieza en 3000, en el mismo orden que tenía en la lista que empieza en 1000; caso contrario, siguiendo dicho orden escribir 0000 en lugar de X .
 - Ver figura 3.34.c. En ella como las dos listas deben progresar juntas se usó SI + 1000 para apuntar la segunda

EJERCICIO 28:

Dada la siguiente codificación en un lenguaje de alto nivel:

```
m=1
n=1
FOR i = 1 to n {
    IF 20 ≤ N ≤ 40      THEN { Z[m] = i;
                                m = m+1}
    ELSE   {Y[n] = i;
            n = n+1 }}
```

Suponiendo que sea traducida por un traductor Compilador, que usa las direcciones 2000, 3000 y 4000 donde empiezan el tipo de variables definidas, y la dirección 1500 para guardar la cantidad n de números enteros a analizar.

- Indicar para el tipo de variables definidas, cómo se presentan en memoria
- Con vistas a traducir la codificación planteada en alto nivel a bajo nivel, en un diagrama lógico, del tipo usado en ejercicios anteriores, expresar lo que dicha codificación planteada ordena.
- Construir el diagrama lógico

- Ver figura 3.35.a
- Se tiene una lista (vector) de enteros $N[i]$ que empieza en 2000 conformada por n números, siendo que n está en la dirección 1500. Analizar cada número N del vector, y como indica la figura 3.35.a :

Si $20 \leq N \leq 40$ escribir su número de orden i en otra lista que empieza en 3000, debajo del número de orden del último número que verificó la condición anterior.

Si N no está entre 20 y 40 escribir su número de orden i en otra lista que empieza en 4000, debajo del número de orden del último número que tampoco verificó la condición anterior.

En la figura 3.35.a se supone que $N[1]$ y $N[4]$ están entre 20 y 40, y que no lo están $N[2]$ y $N[3]$.

- Ver figura 3.35.c. En ésta como las listas 2 y 3 deben progresar juntas se usó DI + 1000 para apuntar la lista 3.

CONSIDERACIONES GENERALES PARA PROGRAMAR EN ASSEMBLER

1. Tener bien en claro qué se quiere o se pide hacer.
2. Planificar **todos** los movimientos y operaciones a realizar mediante esquemas espaciales UCP-Memoria, como los indicados en los ejercicios 1 al 13.
3. Realizar un diagrama lógico que simbolice lo planificado en el paso 2.
4. Suponer valores a los datos y realizar una prueba de "escritorio" para verificar el correcto desarrollo del diagrama
5. Llevar a cabo la codificación Assembler del diagrama lógico obtenido en 3.

Cuando se efectúan las fases 2 y 3 se debe tener presente que los programas empiezan inicializando los registros elegidos y que en el caso de una lista o vector, si se conoce la cantidad **n** de números que la conforman siempre es posible terminarlos con las instrucciones DEC CL seguida de JNZ, a fin de volver a pedir el número siguiente de la lista mientras sea **CL ≠ 0** ó bien seguir con el almacenamiento de resultados en memoria, si el ejercicio así lo exige.

Si se conoce cómo finalizar los diagramas sobre listas, ello facilita su construcción, pues dado que la fase de inicialización de registros con valores es sencilla, la programación debe centrarse en el desarrollo del procedimiento que el lazo debe repetir (en grisado en los diagramas lógicos, y en negrita normal en la codificación en Assembler).

ELECCION DE LOS REGISTROS A UTILIZAR

En principio, como en los ejercicios desarrollados, usaremos el registro AX o AH ó AL para enviar datos desde memoria o para acumular resultados. También pueden usarse BX, CX o DX.

Los registros se eligen de uno o dos bytes acorde al tamaño de los datos (variables).

Si son variables que ocupan 2 bytes en memoria (por ej. enteros), se debe usar registros terminados en X; y si se trata de codificaciones que ocupen un byte en memoria (por ejemplo caracteres en ascii) usar AH, AL, BH, BL, CH, DH, DL.

El registro CL se usará como contador regresivo para guardar inicialmente la cantidad **n** de números o elementos que conforman una lista o vector (siempre que **n** sea un dato conocido)

Los registros SI y DI serán usados como **punteros**, para guardar los **16 bits** de las direcciones de las listas que ellos apuntan, y si se requiere una tercer lista se usará BX como puntero. Esto es, los únicos registros que en Assembler se pueden poner entre corchetes son SI, DI y BX. Por ejemplo:

MOV AX, [SI] ; ADD AX, [BX] ; SUB AL, [DI] ; MOV [BX], DL. No existen instrucciones como MOV DX, [AX].

En programas .COM (que son los que por defecto se usan con el Debug) también se puede emplear el registro BP (Base Pointer) de la UCP usado como puntero en llamadas a subrutinas. Por ejemplo podría ser ADD CX, [BP].

Los registros SI y DI no pueden partirse en dos registros de 8 bits (como ser DIH o DIL o SIH, etc.) lo cual sí puede hacerse con BH y BL de BX, cuando se usa a BX como acumulador (como ser en SUB BH, [SI] o en SUB BL, [SI]).

Las instrucciones del tipo MOV AX, [SI] ; ADD AX, [BX]; SUB AL, [DI]; MOV [BX], DL se usan para puentear elementos de una lista.

Cuando el corchete está a la derecha de la coma se están leyendo (copiando) dos celdas consecutivas de memoria que pasan a un registro como en MOV AX, [SI], o una sola como ocurre con MOV AL, [DI], pues AX es un registro para 2 bytes, mientras que AL es para un byte.

Si el corchete está a la izquierda implica que la operación ordenada dejará el resultado en una posición de memoria indicada por el registro puntero involucrado, y se escribirán uno o dos bytes según que el registro de origen sea para un byte (por ejemplo MOV [DI], BH) o de dos (como en MOV [DI], BX).

También son posibles instrucciones como MOV [DI], SI ó MOV [3000], SI en la que una copia del contenido del puntero SI se guarda en dos posiciones de memoria. En la primera de ellas la dirección está indicada por el puntero DI.

Si se tiene dos listas cuyos punteros deben progresar juntos siempre el mismo valor, puede usarse un solo puntero, según se ilustra en los ejercicios 16, 26 y 27. Esto también podría haberse hecho en el ejercicio 7.
Lo anterior puede extenderse a más de dos listas si se cumple para todas lo antedicho, a fin de ocupar menos registros.

DISTINTAS FORMAS DE REALIZAR UNA MISMA OPERACIÓN SEGÚN DONDE ESTÁ UNO DE LOS DATOS o DONDE VA EL RESULTADO

1) Casos de lectura o escritura de memoria, (instrucciones con un par de corchetes (**no** puede haber dos pares))

A continuación se indican cuatro formas para indicar *en memoria* donde está un dato o dónde se guarda un resultado, codificadas en Assembler para operaciones MOV, ADD, SUB, CMP, etc., con datos o resultados de **dos bytes**.

- Cuando se tiene en memoria un dato aislado, o sea que no forma parte de una lista (vector), suponiendo que el mismo está en las direcciones 2000/1 y que se lo quiere llevar hacia el registro AX, la instrucción será MOV AX, [2000]; y si se lo quiere sumar con el contenido de AX y dejar el resultado en AX, la instrucción será ADD AX, [2000].

- Si un dato o resultado está por ejemplo en AX, y una copia del mismo se lo quiere escribir en memoria, como ser en las direcciones 2000/1 que no forman parte de las direcciones de una lista, la instrucción será MOV [2000], AX.

- Suponiendo que en memoria se tiene un dato que forma parte de una lista (vector), cuya dirección está apuntada por ejemplo por el valor del registro SI, y una copia de este dato se quiere enviar hacia AX, la instrucción será MOV AX, [SI] y si se quiere sumar este dato con el valor de AX y dejar el resultado en AX, la instrucción será ADD AX, [SI].

Con SI=2000, dicho dato ocupará las direcciones 2000/1 de la lista de la que forma parte.

- Si un dato o resultado está en AX, y una copia del mismo se quiere escribir en dos celdas de memoria apuntadas por SI que forman parte de una lista (vector), la instrucción será MOV [SI], AX. De ser SI=2000, la escritura se hará en las direcciones 2000/1.

Lo anterior vale también si en vez de AX se usa BX, CX o DX, o si en vez de SI se usa DI o BX.

En cualquier caso se debe tener presente que luego de leer o escribir un elemento de una lista se debe actualizar (indexar) el valor del puntero, SI, DI o BX a fin de leer o escribir el elemento siguiente si corresponde. Si interviene un registro terminado en X, como AX, se debe sumar 2 al puntero, para localizar un elemento que está dos direcciones más abajo.

Para los casos anteriores si el dato o resultado ocupa un byte, y si usamos el registro AH, las instrucciones serán:

```
MOV AH, [2000]; ADD AH, [2000]
MOV [2000], AH
MOV AH, [SI] ; ADD AH, [SI]
MOV [SI], AH
```

Para todas las instrucciones anteriores si se supone 2000 la dirección involucrada, en la misma se lee o escribe un byte.

Los ejemplos dados también permiten afirmar que en las instrucciones donde intervienen AX, BX, CX, DX los operandos o resultados ocupan 2 bytes; mientras que si son registros terminados en H ó L, los mismos ocupan un byte.

NO es factible realizar **MOV [],I** como ser MOV [SI], [DI] ; ni **ADD [],I** como ser ADD [3000], [DI] CMP [SI], [3000], etc., o sea en una misma instrucción no puede haber dos pares de corchetes.
Esto es, no se puede pasar información de memoria a memoria, ni los dos datos a operar pueden estar en memoria.

Si se quiere efectuar esas operaciones con dos corchetes, se debe “triangular” a través de cualquier registro disponible:
Así, MOV [SI], [DI] puede hacerse como MOV DX, [DI] seguido de MOV [SI], DX si el dato es de dos bytes.
ADD [3000], [DI] puede hacerse como ADD DH, [DI] seguido de ADD [3000], DH si el dato es de un byte.

Las instrucciones que en el corchete tienen un número que es directamente la dirección a acceder se conocen como instrucciones en modo directo; y las que en el corchete un registro provee la dirección son en modo indirecto por registro.

2) Casos de operaciones entre registros (instrucciones en modo registro)

Ocurren cuando se copia información de un registro a otro de la UCP, o si un dato está en un registro y el resultado va a otro registro (que puede ser el mismo). Por ejemplo: MOV AX, SI ; ADD AL, BH ; SUB AX, AX etc.
Este tipo de instrucciones se ejecutan más rápidamente que las que tienen corchete, pues no se debe acceder a memoria.
NO deben realizarse operaciones entre registros de distinto tamaño, como ser MOV AL, SI ; ADD AL, CX

3) Casos de operaciones entre un registro y una constante (instrucciones en modo inmediato)

Típicamente son las instrucciones que inicializan registros (MOV SI, 2000), MOV AX, 0 ; o las que se usan para indexar punteros (ADD DI, 2), que ya se han generalizado luego de la figura 3.8.

En todas ellas la constante que interviene está inserta en la instrucción que llega a la UCP para ser ejecutada, o sea que las constantes que intervienen en un programa forman parte de instrucciones del mismo.

Como dicha constante está inmediatamente a continuación del código de operación, resulta la denominación “inmediato”. Dado que la constante a operar viene con la instrucción pedida de la memoria, no hace falta del tiempo para acceder a ésta para obtenerla, como en las instrucciones con corchete, por lo que también son instrucciones de rápida ejecución.

No son factibles instrucciones del tipo **MOV [], constante** por ej. MOV [3500], FF o SUB [2000], 4000 etc.
Estas operaciones necesitan “triangulación” a través de un registro: primero MOV CH, FF y luego MOV [3500], CH

4) Instrucciones de salto condicional (instrucciones en modo relativo)

Como aparece en distintos ejercicios y en las figuras 3.15 y 316, en cada rombo de un diagrama lógico, si se cumple la condición indicada en el mismo, se debe saltar a una instrucción (salida por el vértice derecho o izquierdo del rombo); y si no se cumple siempre se debe seguir con la instrucción que sigue en memoria a la instrucción de salto (salida por el vértice inferior del rombo, no pudiéndose saltar por este vértice hacia otra instrucción).

Se debe elegir para enteros (figura 3.15) o naturales (figura 3.16) en cada situación el rombo adecuado.

Cómo se codifican y ejecutan las instrucciones de salto condicional

Supongamos que en el Debug se codifique la instrucción de salto JZ 0126 seguida de INC DI:

011E JZ 0126 ↴

0120 INC DI

Codificación: cuando se pulse ↴ (Enter), el programa traductor Ensamblador reemplazará los códigos ascii de JZ 0126 por los códigos ascii de 7406. Esto es, 74 equivale JZ, y 06 = 0126 – 0120 es el valor que hay que sumarle a 0120 (dirección de la siguiente instrucción a ejecutar si no se cumple que Z=1, o sea si es Z=0) para que si se salta se obtenga 0120 + 6 = 0126.

Ejecución: en el momento de ejecutar JZ 0126 se pulsa el comando T del Debug. Entonces una copia de su código de máquina 7406 que está en la dirección 011E, pasará de memoria a la UCP.

Como esta instrucción al igual que todas las de salto condicional ocupa 2 bytes, la UC ordenará sumarle 2 al IP=011E, que así pasará de 011E a 0120, dirección de INC DI, con la que se debe seguir si no se cumple que Z=1, o sea si es Z=0. De esta forma, automáticamente primero el IP queda preparado con la dirección (0120) de la instrucción siguiente, por si no se cumple la condición Z=1, o sea si es Z=0.

En caso que sea Z=1 la UC ordenará sumarle al valor (0120) que quedó preparado el IP (para el caso que sea Z=0) el valor 06 que acompaña a J4: 0120 + 06 = 0126, obteniéndose la dirección (0126) de la instrucción hacia la que se ordenó saltar si fue Z=1, para que el programa prosiga su ejecución a partir de la misma.

Por lo tanto, el Ensamblador realiza en la codificación $0126 - 0120 = 06$ en lugar de $0126 - 011E = 08$, pues está pensado para proveer a la UCP el valor 06 que tiene en cuenta el hecho de que siempre se le suma 2 al IP, para que dirccione si Z=0 a la instrucción que sigue a la de salto condicional; y esto ocurre antes de que se determine durante la ejecución de la instrucción de salto si Z=0 ó Z=1,

Lo planteado para JZ vale para cualquiera de las instrucciones de salto de las **figuras 3.15 y 3.16 a las que siempre conviene consultar cuando se elige una instrucción de salto condicional**. Igualmente se deben tener presente las generalizaciones respecto de las mismas que siguen al ejercicio 5.

En relación con la elección del valor de los flags que intervienen en la condición de la instrucción de salto debe tenerse en cuenta los ejercicios 5, 6 y 9 donde puede verse la conveniencia de analizar atentamente la elección del valor del flag de la condición. Si se elige el valor contrario al apropiado trae aparejado una mala estructuración de un programa

El valor (06) que acompaña al código de operación de una instrucción de salto (como era en 7406), es un número entero de un byte. Este puede ser un número negativo que va de $80h = 10000000b = -128d$ hasta $11111111b = -1d$ (cuando se quiere sumar al IP un valor negativo para que se salte hacia atrás, como debe ser en un lazo); o un positivo que va de $00000000 = 0$, hasta un máximo $7Fh = 01111111b = 127d$ (cuando se quiere saltar hacia delante).

En definitiva en un salto condicional no se puede saltar más que 128d celdas hacia atrás de la dirección que sigue a la de salto, o hasta 127d celdas hacia delante de dicha dirección.

Los valores límites 80 ó 7F que calcula el Ensamblador no pueden ser superados, pues de ser así habrá indicación de error. Por ejemplo esto sucedería si se codifica 011E JZ 0200, dado que la resta $0200 - 0120$ superaría 7F.

5) Instrucciones de salto incondicional (JMP) y de fin de secuencia (INT 20)

Luego de una instrucción de fin de secuencia (INT 20) o de una de salto incondicional (JMP) no puede seguir otra que se puede ejecutar a continuación de ella.

Una instrucción JMP se usa en general para saltar de una secuencia a otra. También, si sigue a una instrucción de salto condicional permite, si no se cumple la condición planteada en ésta, saltar a otra secuencia.

6) Los dos tipos de Instrucciones de resta (SUB y CMP)

La orden **SUB** (Subtract) como en SUB AX, 3000 ordena hacer $AX \leftarrow AX - 3000$ y que los flags tomen el valor que corresponda. Si AX contenía 5000, después de la resta pasará a contener 2000 y se pierde el 5000.

En cambio con **CMP** (Compare) si se hace CMP AX, 3000 ésta ordena hacer $AX - 3000$ sin asignación del resultado, o sea sin destruir el valor (5000) que estaba en AX, y los flags tomarán los mismos valores que si se hiciera SUB AX, 3000. Es la típica resta que se usa para comparar, esto es con el fin de determinar cómo es un número (mayor, menor, igual) respecto de otro, sin que interese el resultado de la resta efectuada. Sólo importa los valores que tomarán los flags por los que se preguntará en la instrucción de salto condicional que sigue a CMP. Recién cuando se ejecuta dicha instrucción de salto, se determina cómo es el minuendo respecto del sustraendo en la comparación buscada, y no después de ejecutar CMP. Por ello, no es muy adecuada la denominación “comparar” para la instrucción en cuestión, siendo que sería más correcto designarla “resta sin asignación”.

Dado que no se destruye ningún operando, evita tener que gastar instrucciones para guardar uno de los operandos. La instrucción CMP se empezó a usar a partir del ejercicio 9.

6) Secuencia de instrucciones que salen de un rombo si se cumple la condición de salto

Cuando desde un rombo se salta a otra secuencia, que termina en un jump hacia ese mismo rombo, si se invierte el valor del flag(s) por el cual se pregunta en el rombo, dicha secuencia puede escribirse debajo de dicho rombo, para seguir con el paso a partir del cual comienza la secuencia común, y conformar así una sola secuencia en vez de dos. En ésta el rombo con la condición modificada ordena saltar a la instrucción a partir del cual empieza dicha secuencia común. Esto se exemplificó en los ejercicios 6 y 9.

7) Acerca de la cantidad n de números de una lista o vector

En la mayoría ejercicios anteriores que comienzan cargando en CL la cantidad de números de una lista o vector, se ha asumido que n ≠ 0 para no complicar el desarrollo. Siempre es factible mediante CMP CL, 0 y JZ determinar si n es cero. De ser así JZ ordenaría saltar al final del programa

EJERCICIOS PARA REALIZAR:

- Analizar cuál es el efecto de las siguientes instrucciones: MOV SI, [SI] MOV [DI], DI ADD SI, [SI]
- Discutir la siguiente aseveración: la instrucción de comparación es solo una resta para dar valores a SZVC, sin que interese el valor concreto del resultado, el cual no se asigna a ningún registro. Recién cuando se ejecuta la instrucción de salto que le sigue, la cual pregunta por el valor de los flags (según que los números comparados sean naturales o enteros), se sabe cómo es un número respecto del otro. O sea que la orden "comparación" no es muy adecuada.

A continuación se plantean los siguientes ejercicios a desarrollar, basados en los primeros 20 ejercicios resueltos; en todos los casos se supone que se conoce la dirección donde comienza la lista originaria y la cantidad n de números.

- Modificar el ejercicio 7 de manera de codificarlo usando solamente el puntero SI.
- Modificar el ejercicio 12 de modo de encontrar el menor de una lista de números naturales, y debajo de la celda dónde se guarda el mismo, escribir su dirección y su número de orden en la lista.
- Se tiene una lista de caracteres ASCII cuya dirección inicial y cantidad n se conoce. Formar otra lista en la que no haya elementos consecutivos iguales, y que vayan los comprendidos entre 30 y 60. Guardar en una tercer lista la dirección donde comienza cada sucesión de consecutivos iguales, y en una celda de memoria la cantidad de números de la 2da lista.
- Se tiene una lista de caracteres ASCII. Formar otra lista en la que vayan solamente elementos consecutivos iguales de la primera lista cuyo número de repeticiones esté comprendido entre 3 y 8 inclusive.
- Se tiene una lista de números enteros. Formar otra lista con los menores de 50 y los mayores de 80. Si entre estos números hay repetidos consecutivos iguales, pasar el primero de ellos a otra lista sólo si el número de repeticiones supera 3.
- Se tiene dos listas de igual longitud con números enteros. Considerar los elementos correspondientes. Determinar cuál es el mayor y restarlo del menor. En otra lista guardar esta diferencia y la dirección del mayor. Si los números son iguales seguir con los números siguientes.
- Encontrar la cantidad de máxima de números impares que hay en una lista de n números binarios enteros de dos bytes
- Se tiene 2 listas de igual cantidad de números enteros. Considerar los elementos correspondientes y determinar cuál es el mayor y restarlo del menor. En otra lista guardar la dirección del mayor y el valor de la diferencia hallada. Si los números correspondientes son iguales, seguir con los números siguientes.
- Se tiene una lista de n caracteres codificados en ascii. Formar otra lista en la que no haya caracteres consecutivos iguales que vayan los comprendidos entre 30 y 60. Guardar en una tercer lista la dirección donde comenzó en cada sucesión de consecutivos el primer carácter de los repetidos, y en una dirección de memoria la cantidad de caracteres de la segunda lista.
- Se tiene una lista de n caracteres codificados en ascii. Formar otra lista en la que vayan solamente el primero de cada grupo de caracteres consecutivos repetidos de la primera lista cuyo número de repetición esté entre 3 y 8.
- Recorrer 2 listas de igual número n de caracteres codificados en ascii. Si se encuentran dos pares de caracteres correspondientes iguales, guardarlos en dos posiciones consecutivas de memoria.
- Se tiene una lista de n números enteros. Sumarle 3 a cada uno y el resultado escribirlo en otra lista. Si el resultado genera overflow, escribir ABCD en lugar del resultado de la suma.
- Se tiene una lista de n caracteres codificado en ascii. Si alguno de ellos es mayor que 40 saltar al carácter siguiente; si es menor colocar 31 en su posición, y si es igual que 40 sumarle 3, pasando a otra lista el resultado.
- Se tiene dos listas con igual número n de caracteres codificados en ascii. Si elementos correspondientes de ambas listas valen 30, reemplazarlos en ambas listas por EE; y si valen 20 sumarle a cada uno el contenido de la posición anterior. En cualquier otro caso dejar las listas como están.
- Se tiene dos listas con igual número n de números naturales. Analizar cada par de elementos correspondientes. Si a un número N de la lista 1 le corresponde otro que sea N + 1 en la lista 2, guardar en otra lista 3 el N de la lista 1, y debajo del mismo la dirección donde se encontraba en esta lista. Asimismo contar la cantidad de estos sucesos, y si su número supera 10 detener el programa.

EJERCICIO 30: uso de JBE (Jump if below or equal)

A partir de 2001 se tiene una lista de combinaciones binarias de un byte que corresponde cada una a dígitos hexadecimales aislados (0, 1, 2, ..., F), siendo que el número **n** de combinaciones está en la dirección 2000. Convertir cada dígito hexadecimal a su correspondiente carácter ASCII según se ejemplifica a continuación. Los caracteres ASCII obtenidos guardarlos en otra lista que comienza en 3001.

Binario	Hexa		ASCII	
2001	00000000	(00)	3001	00110000 (30)
2002	00000100	(04)	3002	00110100 (34)
2003	00001001	(09) =>	3003	00111001 (39)
2004	00001010	(0A)	3004	01000001 (41)
2005	00001101	(0D)	3005	01000100 (44)
2006	00001111	(0F)	3006	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

$$\begin{array}{r} +7 = + \underline{\hspace{2cm}} 111 \\ 00010001 \end{array}$$

+30 = 00110000

01000001 = 41 en ASCII

OTRO	MOV CL, [2000]	Carga en CL el número n de combinaciones binarias de la lista
	MOV SI, 2001	SI apunta al comienzo de la lista de datos
	MOV DI, 3001	DI apunta al comienzo de la lista de resultados
OTRO	MOV AL, [SI]	Carga en AL un dato de la lista apuntada por SI
	CMP AL, 9	Compara el dato en AL contra 9
	JBE SUMA	Si es menor o igual que 9 salta a 0112
SUMA	ADD AL, 7	Suma 7 a AL
	ADD AL, 30	Suma 30 a AL
	MOV [DI], AL	Guarda un resultado en la lista apuntada por DI
	INC DI	Incrementa DI
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a 010A
	INT 20	Fin

La instrucción **JBE SUMA** que sigue a **CMP AL, 9**, ordena saltar a la instrucción que está en **SUMA** 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 SUMA**. O sea que en un diagrama lógico, en el rombo correspondiente a **JBE** habría que escribir **C=1** ó **Z=1**.

EJERCICIO 31:

Este ejercicio es contrario al anterior. Esto es, se tiene una lista de caracteres ASCII que empieza en 2001, y su número **n** 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.

OTRO	MOV CL, [2000]	Carga en CL el número n de caracteres de la lista
	MOV SI, 20001	SI apunta al comienzo de la lista de datos
	MOV DI, 3001	DI apunta al comienzo de la lista de resultados
OTRO	MOV AL, [SI]	Carga en AL un dato de la lista apuntada por SI
	SUB AL, 30	Resta 30 a AL
	CMP AL, 9	Compara el contenido de AL con 9
	JBE ALFA	Si es menor o igual salta a 0114 (Si AL < 9 es un símbolo del 0 al 9)
	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)
ALFA	MOV [DI], AL	Guarda el resultado en la lista apuntada por DI
	INC DI	Incrementa DI
	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a 010A
	INT 20	Fin

EJERCICIO 32: 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 número **n** 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_H en la dirección 3000.

	MOV CL, [2000]	Carga en CL el número n de caracteres de la lista
	MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
	MOV DL, 00	Carga 00 en DL suponiendo que todo está bien
OTRO	SUB AL, AL	Hace cero el registro AL
	ADD AL, [SI]	Suma contra AL para que pueda cambiar el indicador de paridad P
	JPE INCR	Si paridad en AL es par, saltar a INCR
	MOV DL, FF	Indicación de un elemento con paridad errada
INCR	INC SI	Incrementa SI
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z=0, volver a OTRO
	MOV [3000], DL	Guarda indicación de paridad
	INT 20	Fin

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

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.

EJERCICIO 33:

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

	MOV CL, [2000]	Carga en CL la cantidad de números de la lista
	MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
	SUB AX, AX	Pone AX en cero
	SUB DX, DX	Pone DX en cero
OTRO	ADD AX, [SI]	Suma contra AX un dato de la lista
	ADC DX, 00	Lleva en DX la suma de acarreos que ocurren cuando las sumas superan el formato de AX (Hace DX ← DX + 00 + Carry = DX + Carry)
	ADD SI, 2	Incrementa SI en 2, pues cada dato ocupa 2 bytes.
	DEC CL	Decrementa CL
	JNZ OTRO	Mientras Z sea 0, volver a OTRO
	MOV [1500], AX	Guarda en 1500 y 1501 los 16 bits inferiores de la suma
	MOV [1502], DX	Guarda en 1502 y 1503 los 16 bits superiores de la suma
	INT 20	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 34:

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 exemplifica a continuación

Los círculos indican comparaciones realizadas; y las flechas, los casos en que hubo intercambio de lugar entre 2 números.

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



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

EJERCICIO 35: uso del modo indirecto por registro con desplazamiento mediante valor de registro

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

MOV SI, 2000	SI apunta al comienzo de la lista de datos
SUB AX, AX	Pone AX en cero
SUB BX, BX	Pone BX en cero
SUB DX, DX	Pone DX en cero
MOV DL, [3000]	Carga N1 en DL
ADD SI, DX	Registro SI apunta a 2000 + N1
MOV AL, [SI]	(N1) ² se lleva a la mitad inferior de AX
SUB SI, DX	El puntero SI vuelve al valor 2000
MOV DL, [3001]	Carga N2 en DL
ADD SI, DX	Registro SI apunta a 2000 + N2
MOV BL, [SI]	(N2) ² se lleva a la mitad inferior de BX
ADD AX, BX	Suma los cuadrados, y reserva para el resultado un registro de 16 bits
MOV [3002], AX	Guarda el resultado en 3002 y 3003
INT 20	Fin

Otra forma más compacta de escribir la secuencia anterior es la siguiente:

MOV SI, 2000	SI apunta al comienzo de la lista de datos
SUB AX, AX	Pone AX en cero
SUB BX, BX	Pone BX en cero
SUB DX, DX	Pone DX en cero
MOV BL, [3000]	Carga N1 en BL (con BH = 0)
MOV AL, [SI + BX]	(N1) ² se lleva a la mitad inferior de AX
MOV BL, [3001]	Carga N2 en BL (con BH = 0)
MOV DL, [SI + BX]	(N2) ² se lleva a la mitad inferior de DX
ADD AX, DX	Suma los cuadrados, y reserva para el resultado un registro de 16 bits
MOV [3002], AX	Guarda el resultado en 3002 y 3003
INT 20	Fin

EJERCICIO 36: 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.

MOV AX, [2000]	Carga en AX los 16 bits inferiores del dividendo
MOV DX, [2002]	Carga en DX los 16 bits superiores del dividendo
MOV CX, [1000]	Carga en CX el divisor
DIV CX	Realiza la división
MOV [1500], AX	Guarda en 1500 el cociente
MOV [1502], DX	Guarda en 1502 el resto
INT 20	Fin

EJERCICIO 37

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 guardararlo en la dirección 1500

MOV CL, [2000]	Carga en CL la longitud de la lista
MOV SI, 2001	SI apunta al comienzo de la lista de datos
SUB AX, AX	Pone AX en cero
SUMA ADD AL, [SI]	Suma a AL un número de la lista apuntada por SI
ADC AH, 00	Lleva en AH la suma de acarreos que ocurren cuando las sumas superan el formato de AL (Hace AH \leftarrow AH + 00 + Carry = AH + Carry)
INC SI	Incrementa SI
DEC CL	Decrementa CL
JNZ SUMA	Mientras Z sea 0, volver a SUMA
MOV BL, [2000]	Carga en BL la cantidad de números de la lista
DIV BL	Hace AL / BL; el cociente va a AL, y el resto a AH
MOV [2000], AL	Guarda en 2000 el resultado de la división (el promedio)
INT 20	Fin

EJERCICIO 38: 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. Convertir 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.

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

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

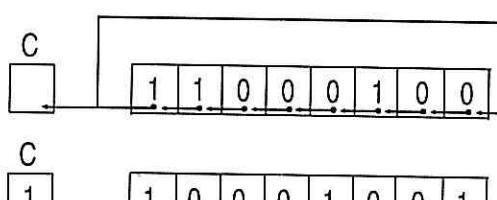
Suponiendo que en AL se tenga, como en este ejemplo 31, la operación será bit a bit: $00110001 = 31$

$$\begin{array}{r} \text{A } 00001111 = 0F \\ \hline 00000001 \end{array}$$

De esta manera, mediante la "máscara" OF 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** (ROtates 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 3.36.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 3.36.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).



Figuras 3.36.a y 3.36.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 39: 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:

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

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

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

MOV DL, [2000]	Carga en DL la longitud de la lista
MOV SI, 2001	Inicializa SI para que apunte al comienzo de la lista de datos
MOV DI, 3001	Inicializa DI para que apunte al comienzo de la lista de resultados
OTRO MOV AL, [SI]	Carga en AL los dos dígitos en BCD
MOV BL, AL	Copia en BL el contenido de AL
AND BL, 0F	Pone en 0 los 4 bits más altos de BL
AND AL, F0	Pone en 0 los 4 bits más bajos de AL
MOV CL, 4	Prepara CL para indicar 4 rotaciones
ROR AL, CL	Rota 4 posiciones a la derecha el contenido de AL
MOV BH, 0A	El multiplicador BH se hace de valor diez
MUL BH	Multiplica AL por BH y el resultado va a AL
ADD AL, BL	Suma las unidades al producto antes efectuado
MOV [DI], AL	Guarda el byte empaquetado en la otra lista
INC SI	El registro SI apunta al siguiente lugar
INC DI	DI apunta al siguiente lugar
DEC DL	Decrementa DL
JNZ OTRO	Mientras Z=0, volver a 010A
INT 20	Fin

EJERCICIO 40:

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

MOV CL, [2000]	Carga en CL la cantidad de dígitos de la lista
MOV SI, 2001	Registro SI apunta al comienzo de la lista de datos
OTRO SUB AL, AL	Hace cero el registro AL
ADD AL, [SI]	Suma un elemento de la lista contra AL
DAA	Instrucción de corrección para suma BCD
INC SI	Incrementa SI
DEC CL	Decrementa CL
JNZ OTRO	Mientras Z no sea 1, volver a OTRO
MOV [1500], AL	Guarda indicación de paridad
INT 20	Fin

EJERCICIO 41:

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

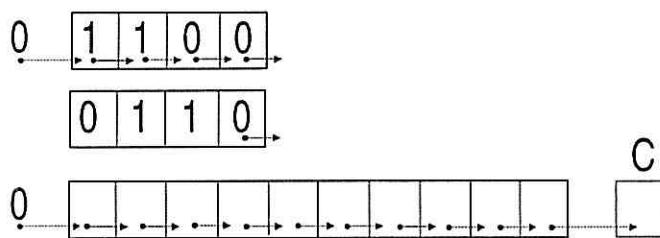
.....

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

El algoritmo (de Newton) se detendrá cuando la diferencia entre dos aproximaciones sucesivas sea cero, 1 ó -1

MOV SI, 2000	Inicializa SI
MOV AX, [SI]	Carga en AX los 16 bits inferiores de N
MOV DX, [SI+2]	Carga en DX los 16 bits superiores de N
MOV CX, C8	El divisor CX toma el valor $C8_{H} = 200_D$
DIV CX	Divide por 200, y el resultado queda en AX
ADD AX, 2	Suma 2 como pide el algoritmo
OTRA	
MOV CX, AX	Lleva A1 (cociente) que está en AX hacia CX (divisor)
MOV AX, [SI]	Vuelve a cargar en AX los 16 bits inferiores de N
MOV DX, [SI+2]	Carga en DX los 16 bits superiores de N
DIV CX	Divide por A1
ADD AX, CX	Suma N/A_1 que está en AX con A1 que está en CX
SHR AX, 1	Divide por 2 la suma antes hallada, para hallar A2
CMP AX, CX	Compara A2 con A1
JZ FINE	Si son iguales, terminar
SUB CX, AX	Resta A1 - A2
CMP CX, 1	Compara $(A_1 - A_2)$ contra 1
JZ FINE	Si es igual a 1, terminar
CMP CX, -1	Si no es igual a 1 determina si es $-1 = FFFF$
JNZ OTRA	Si no es igual a -1 se hace otra aproximación
MOV CX, AX	Guarda An válida en CX
FINE	Fin
INT 20	

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 42: 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 $-6_D = 11111010_B = FA_{H}$

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.

xxxx:0100 MOV CL, [2000]	Carga en CL uno de los números enteros de 8 bits
xxxx:0104 MOV AL, FA	Carga en AL el otro número entero de 8 bits
xxxx:0106 IMUL CL	Ordena multiplicar CL por AL, con resultado en AX
xxxx:010A INT 20	Fin

EJERCICIO 43: 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 $-6_D = FFFA_H$

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)

MOV BX, [2000]	Carga en BX el número entero de 16 bits que está en 2000 y 2001
MOV AX, FFFA	Carga en AX el otro número entero de 16 bits
IMUL BX	Ordena multiplicar BX por AX, con resultado en DX y AX
INT 20	Fin

EJERCICIO 44: 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 $-6_D = FA_H$

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.

MOV BX, [2000]	Carga en BX el número entero de 16 bits que está en 2000 y 2001
MOV AL, FA	Carga en AL el otro número entero de 8 bits
CBW	Convierte FA (que está en AL) en FFFA que ocupa AX
IMUL BX	Ordena multiplicar BX por AX, con resultado en DX y AX
INT 20	Fin

EJERCICIO 45: Suma en BCD

Sumar dos números BCD de 4 dígitos (2 bytes) que están a partir de las direcciones 1000 y 2000, y el resultado guardarlo a partir de la dirección 3000.

SUB AH, AH	Se pone AH = 00 para usarlo más adelante
MOV BX, [1000]	Carga en BX el número BCD de 2 bytes que está en 1000 y 1001
MOV DX, [2000]	Carga en DX el otro número BCD
MOV AL, BL	Pasa BL a AL, éste es el único registro para sumar en BCD y usar DAA
ADD AL, DL	Suma las mitades inferiores de los dos números en BCD
DAA	DAA corrige en bin sumas de dos dígitos BCD que superan 9 (U4 página 96)
MOV CL, AL	Pasa a CL la suma parcial correcta de las mitades inferiores de los nros BCD
MOV AL, BH	Pasa a AL la mitad superior del número BCD cargado en BH
ADC AL, DH	Suma mitades superiores de los dos nros BCD, más acarreo de mitades inferiores
DAA	Instrucción DAA corrige en binario sumas de 2 dígitos BCD que superan 9
MOV CH, AL	Pasa a CH la suma parcial correcta de mitades superiores de los nros BCD
MOV [3000], CX	Pasa de CX a 3000 y 3001 la suma de las mitades inferior y superior
ADC AH, 00	Si en suma de mitades sup. hubo acarreo hacer 01 a AH. Sino queda en 00
MOV [3002], AH	Pasa a 3002 (resultado) el valor 01 de AH si hubo carry; sino pasa el valor 00
INT 20	Fin

Si se tuviera una resta en BCD, en lugar de ADD AL, DL se tendría SUB AL, DL; en vez de DAA se usaría DAS, y en lugar de ADC se tendría SBB. Las dos anteúltimas instrucciones podrían no ser necesarias.

DIRECCIONES EFECTIVAS Y REGISTROS DE SEGMENTO EN MODO REAL¹

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 3.37 superior) cuyas direcciones iban de 0000 a FFFF = 65.535, o sea una memoria de 64 KB.

En “modo real”², 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:

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

Se observa que a 2B16 se le agregó un cero a la derecha (equivalente a cuatro ceros en binario).

Lo anterior implica (figura 3.37 inferior) que la UCP (80x86) en modo real direcciona una memoria cuyas direcciones van de 00000 a FFFFF = 1.048.575, 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 en modo real 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 3.38). 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 3.38 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 3.38). 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 ó DS:BX.
En la figura 3.38 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 la cima dentro de la pila (figura 3.38).
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 3.38). 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 3.38 se ha supuesto ES:DI = E230:5420

CS, DS, SS y ES permiten cambiar de lugar programas, con tan solo cambiar el valor contenido en ellos.

Si observamos el Debug vemos que normalmente los registros CS, DS, SS y ES tienen el mismo valor.

Ello implica que los 4 segmentos citados empiezan en la misma dirección (2B160), o sea que están superpuestos (figura 3.39). Esta es la forma en que los hemos estado usando, 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.

¹ Este tema es necesario cuando próximamente se traten las interrupciones.

² Cuando se enciende una PC, muchos procesadores actuales funcionan con la menor cantidad de recursos posibles, simulando un 8086 que utiliza sólo 1 MB de la memoria total, y con los registros que conocemos de Debug, lo cual se conoce como “modo real”, para luego pasar al “modo protegido” cuando el sistema operativo pasa a memoria, a fin de complementar sus funciones y poder realizar multitasking (multiprogramación).

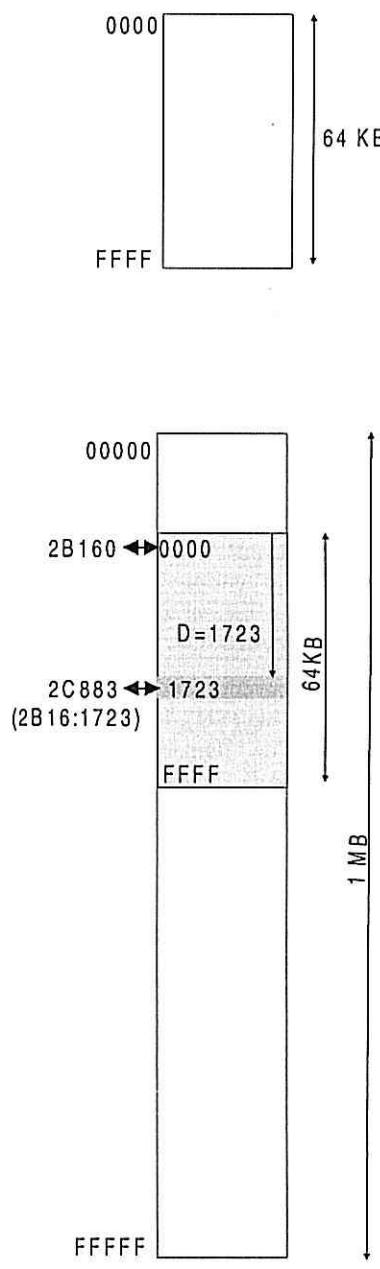


Figura 3.37

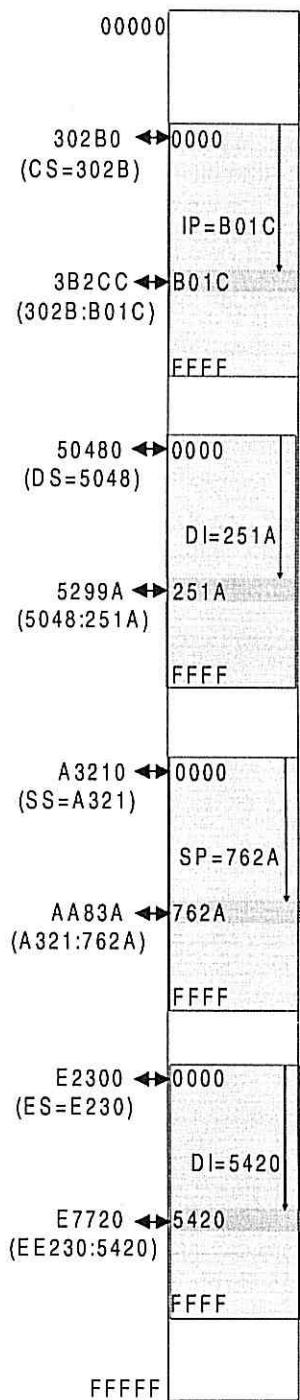


Figura 3.38

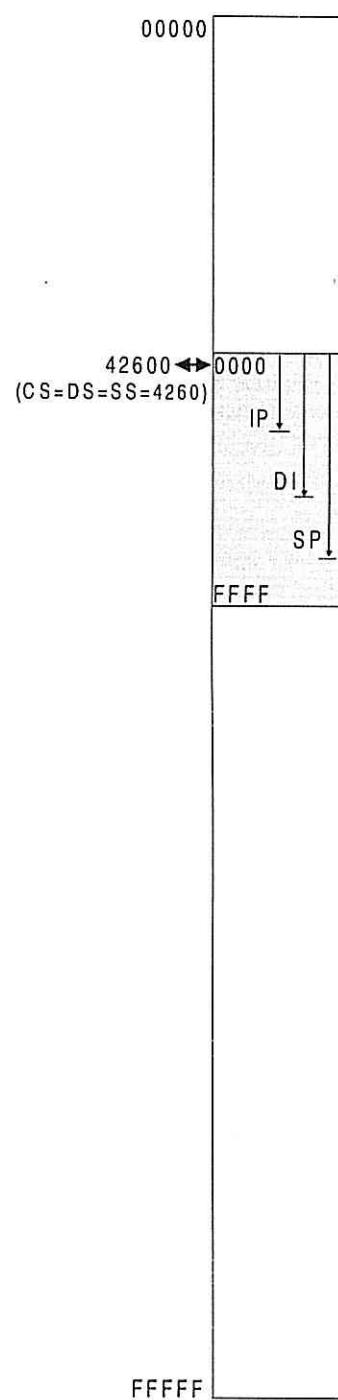


Figura 3.39

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 3.40)

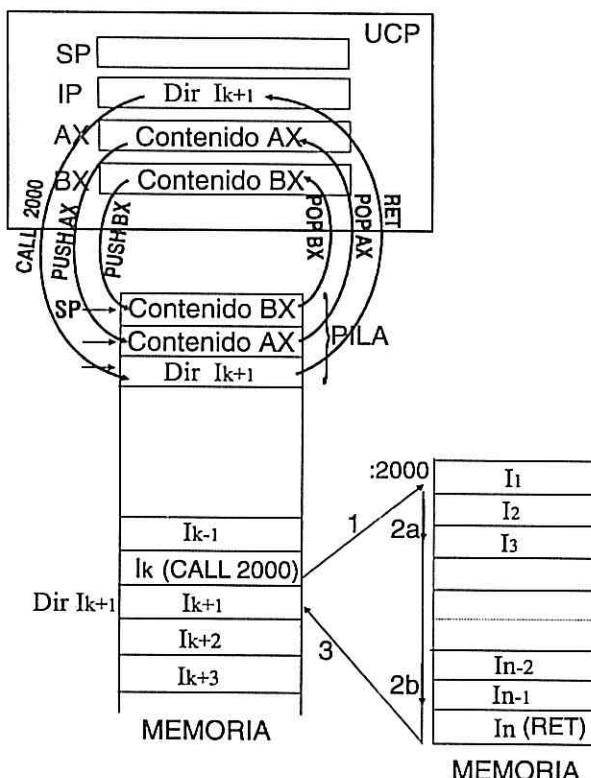


Figura 3.40

El nombre “pila” se debe a que en la zona de memoria que ella ocupa se van “apilando” –como una pila de platos– información relacionada con el PGM llamador, que es copia de contenidos de registros de la UCP, que la instrucción CALL y las primeras instrucciones de la SR (tramo 2a de la figura 3.40) ordenan apilar. 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.

1. Cuando desde un programa (PGM) se quiere pasar a ejecutar una subrutina (SR), una instrucción Ik del mismo denominada de “llamada a subrutina” (CALL en assembler de Intel), debe ordenar saltar a la primera instrucción (I1) de dicha SR. Previamente (paso 0) Ik ordena guardar en memoria principal la dirección de Ik+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 I1 y las siguientes instrucciones de la SR.
3. La última de estas instrucciones In (RET en assembler) ordena saltar a la dirección de Ik+1 del programa llamador de la SR, o sea retornar a este programa.

La dirección de Ik+1, es la “dirección de retorno” y es guardada –paso 0 de la ejecución de la instrucción Ik 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).

El registro **SP**, indicador de la dirección más alta (“cima”) que tiene la pila, es usado para apilar y desapilar en forma ordenada, los contenidos de la pila. El mismo es manejado *automáticamente* por la UC cuando ejecuta cada instrucción que ordena apilar o desapilar, siendo que la UC por medio del **SP** controla el orden de la pila.

A su vez la SR1 que llamó el PGM, puede llamar a otra SR2, y ésta a otra SR3, y así se pueden encadenar varias llamadas sucesivas a partir de la SR1 (“nested” o anidamiento de subrutinas), debiéndose volver siempre a la instrucción que sigue a la instrucción CALL que llamó a la SR1.

En esos casos la pila -que empezó a crecer a partir de la dirección provista por el SP con el valor que tenía cuando el PGM llamó a la SR1, o sea con un valor relacionado con dicho PGM- irá apilando la información contenida en los registros de la UCP, referente al valor que éstos tenían antes que desde cada SR se llamó a otra, como ocurrió cuando el PGM llamó a la SR1, constituyendo así una pila única para ese proceso.

Como cada SR termina con instrucciones que ordenan desapilar y volver a la instrucción siguiente a la que la llamó, todos los apilamientos realizados en la pila en cuestión se irán sucesivamente desapilando ordenadamente, hasta retornar al PGM que llamó a SR1. Entonces el SP también volverá al valor que tenía entonces. Más adelante al tratar el tema de las interrupciones por software y hardware, se verá que en un proceso de llamadas y retornos durante las cuales la pila tiene “vida”, almacenando direcciones y datos, también una única pila relacionada con un PGM, se usa tanto para las llamadas a SR como para las interrupciones, que también son llamadas pero a subrutinas del BIOS o del SO. Esto se ejemplifica en el ejercicio 47.

Para que un PGM pueda enviarle los datos que procesará la SR que llamó, y que ésta pueda dejarle al PGM los resultados que necesita (comunicación conocida como “pasaje de parámetros”), quién programó la SR debe dejar indicado los “buzones” necesarios, los cuales pueden ser:

- 1) Registros de la UCP.
- 2) Posiciones de la pila, en las que se apilarán datos para la SR y se desapilarán resultados para el PGM.

DESCRIPCION MAS DETALLADA DEL PROCESO MEDIANTE UN EJEMPLO Y EL DEBUG

EJERCICIO 46.a

Se tiene una lista de **n** números enteros cuya dirección inicial es 1000, siendo que dicha cantidad **n** de números está en la dirección 1500. Copiar en otra lista que empieza en 2000 sólo los números comprendidos entre 20 y 40; y para cada uno de éstos escribir debajo de su valor el número de orden que tiene en la lista originaria (ver esquema superior fig. 3.41). Suponer que existe una subrutina para determinar si $20h \leq N \leq 40h$.

Este ejercicio se basa en el nº 17 pero tiene menos exigencias. En la parte superior de la figura 3.41 se ejemplifica suponiendo que N1 no está entre 20 y 40, y que N2 sí lo está, con lo cual N2 se copia en 2000/1, y su subíndice **i = 2** en 2002. Encararemos el “pasaje de parámetros” de dos formas:

- 1) Usando un registro de la UCP como puente entre el programa (PGM) y la subrutina (SR).
- 2) Idem usando la pila que está en memoria.

1) Por pasaje de parámetros a través de registro:

El diagrama lógico de la figura 3.41 presenta las secuencias de pasos necesarios para esta alternativa, la mayoría de los cuales ya estaban en la figura 3.32.

Visión general: los pasos de la izquierda corresponden al PGM, y los de la derecha a la SR.

El proceso comienza con los pasos 1 al 9. Este último en Assembler es **CALL 3500** que ordena saltar a ejecutar la SR cuyo primer paso debe ser una instrucción que esté en la dirección de memoria 3500. La última instrucción de la ejecución de la SR, que es **RET**, contraria a **CALL**, ordena retornar a la instrucción del PGM que sigue a CALL, luego de lo cual el PGM sigue su curso, escribiendo o no (según corresponda) en la lista apuntada por DI, el número N seguido por su número de orden en la lista.

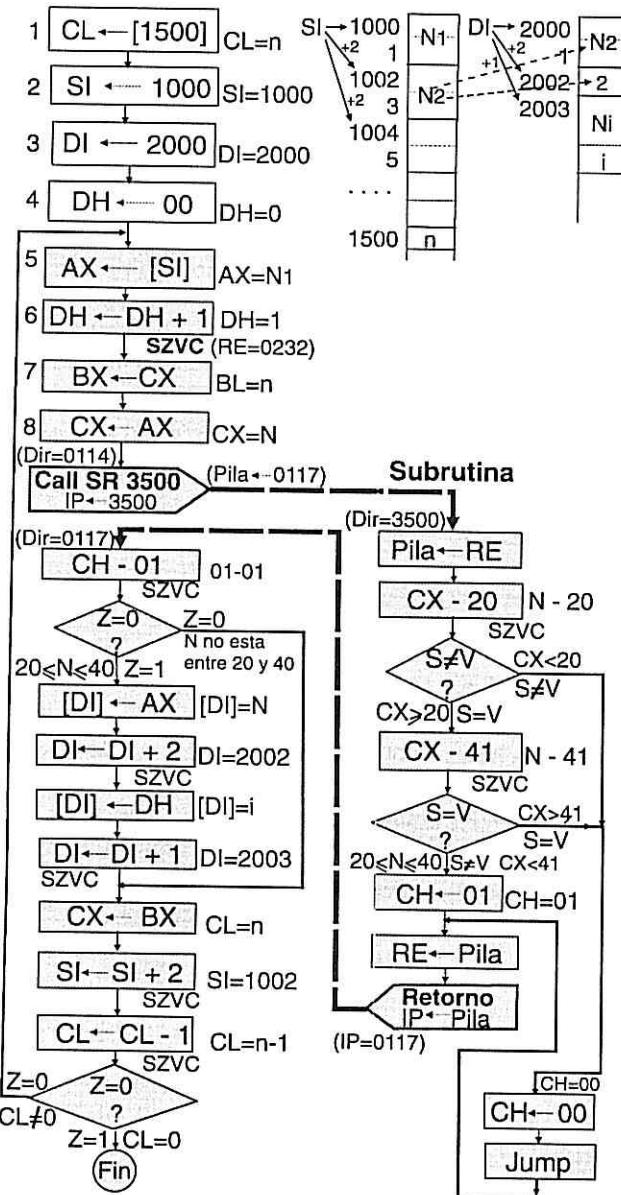
Después (mientras queden números sin analizar) en el PGM se volverá al paso 5 para obtener el siguiente N de la lista apuntada por SI, y en el paso 9 se volverá a llamar a la SR, y así de seguido hasta analizar los n números de la lista apuntada por SI.

En el PGM antes de que se llama a la SR mediante la instrucción **CALL**, y cada vez que ello sucede, se pasa al registro “puente” o “buzón” **CX** que es el que usa como acumulador la SR, una copia del número N de la lista apuntada por SI, a fin de determinar si es o no $20 \leq N \leq 40$.

Una subrutina, como la que calcula la raíz cuadrada de un número, puede ser llamada por distintos programas o desde diferentes puntos de un programa. La persona que desarrolla un programa llamador no tiene por qué ser la misma que generó la SR, ni tiene por qué conocer cómo ella funciona. Sólo se necesita conocer en qué registro dejar el dato(s), en este caso CX, y en qué registro quedará el resultado del procedimiento que realizó la SR (también en este ejemplo en la mitad CH de CX) para que el programa tome una copia del mismo.

Si se observa la SR se verifica que ella opera al registro CX (podría haber sido BX, CX ó DX) y que deja en CH la indicación acerca de si N está comprendido (indicación 01 h) o no (indicación 00 h) entre 20 y 40.

En el primer paso del diagrama de la figura 3.41 se inicializa el registro CL con una copia de la cantidad n de números a analizar, supuesta en la dirección 1500 (ver esquema superior derecho fig 3.41); y en los dos pasos siguientes los registros SI y DI se inicializan para que apunten a la dirección de comienzo de cada una de las listas. El 4to. paso ordena llevar a cero el registro DH que será usado para generar el número de orden que tiene en la lista cada número N que se analiza.



Cada vez que se ejecuta el paso 5 llega a AX una copia del número N cuya dirección indica SI; y cada vez que esto sucede se debe generar su número de orden, incrementando DH (paso 6: $DH \leftarrow DH + 1$).

Así, luego de pasar hacia AX el primer N (N1) que está en 1000, al sumar uno a DH resulta $0 + 1 = 1$ que es el número de orden de ese N (Al inicio era $DH = 0$).

En el paso 7 hay que pasar a BX una copia de CX, (en cuya mitad CL se guardó la cantidad n de números de la lista apuntada por SI), pues CX será modificado en el paso 8, dado que a CX irá una copia del número N que analizará la SR, la cual también modificará a CX. Con el paso CALL SR 3500 se ordena saltar a ejecutar la SR cuya primera instrucción está en 3500 (fig. 3.41)

Toda SR comienza guardando (apilando en la pila) mediante instrucciones PUSH¹ todos los registros que va a utilizar, menos el registro(s) elegido para hacer el pasaje de parámetros (CX en este tipo de pasaje).

Si bien la SR planteada sólo usa el registro CX, como durante su ejecución se realizan operaciones de resta, variarán los valores de los flags contenidos en el registro de estado (RE) como se indica con SZVC en la figura 3.41, en relación con los valores que tenían los flags, por ejemplo luego de hacer $DH \leftarrow DH + 1$.

O sea, que implícitamente en la SR además de CX se está usando RE, por lo que antes del retorno al PGM (paso CH - 01) los flags deben volver al valor que tenían antes de llamar a la SR.

Por ello la SR empieza con PUSHF (Flags) equivalente a PUSH RE, antes que la SR cambie el valor de los flags, guardando en la pila al RE como quedó en el PGM.

Igualmente, si por ejemplo la SR en vez de $CH \leftarrow 00$ hubiese hecho $BH \leftarrow 00$ y luego $CH \leftarrow BH$ habría que hacer PUSH BX después de PUSH F, para guardar BX en la pila, con el valor que tomó en BX ← CX. Al final de la SR, con POP BX dicho valor se restaura en BX. De no ser así, al retornar al PGM se pierde el valor que tenía BX en el mismo.

Por otra parte, la SR no guarda en la pila AX ni DX, por que no los emplea, y "no sabe" que el PGM los está usando.

Figura 3.41

En definitiva, en el primer paso de esta SR (Pila ← RE) el único registro cuyos contenidos se guardan en la pila en este ejemplo es RE mediante PUSH F, los cuales serán devueltos a RE mediante POP F, correspondiente al paso RE ← Pila. Los apilamientos que producen CALL 3500 y PUSH F serán analizados en detalle en la figura 3.42 y verificados cuando estas instrucciones se ejecuten mediante el Debug.

El paso que en la SR ordena la resta CX - 20 sirve para que tomen valor los flags SZVC. Si en el paso siguiente se detecta $S \neq V$, ($CX=N < 20$) hay que saltar al paso donde se hace $CH = 00$, para indicar que si por ejemplo es $N=15 < 20$ no estará entre 20 y 40. En caso que en el paso siguiente a la resta CX - 20 se detecte $S=V$ ($CX=N \geq 20$), el número N que está en CX puede llegar a estar entre 20 y 40.

Para ver si $N \leq 40$ luego se hace CX - 41 para que tomen valor los flags SZVC. Si en el paso que le sigue se detecta $S=V$ (por ej. si $N = 55$, es $CX=N \geq 41$, y por lo tanto $N > 40$, con lo cual también se ordena saltar al paso que hace $CH = 00$). Pero si en el rombo que sigue a CX - 41 se detecta $S \neq V$, debe ser $CX < 41$, y por consiguiente $N \leq 40$ (por ej. si $N=35$).

¹ "Push" proviene de "empujar" las monedas en un monedero del tipo cilindro con resorte, para ir apilándolas, siendo que la acción contraria de ir sacando cada moneda de la pila de monedas almacenadas, se denomina "Pop".

Como hasta este punto por los pasos anteriores era ya $20 \leq N$ ahora resulta $20 \leq N \leq 40$, por lo que en el paso siguiente se hace $CH = 01$ para indicar que N está entre 20 y 40. Con este paso termina el procedimiento que realiza la SR.

La SR mediante los pasos $RE \leftarrow$ Pila (POP F) e $IP \leftarrow$ Pila (RET), pasan de la pila hacia los registros RE e IP los contenidos que RE e IP tenían antes del llamado a la SR, desapilándose los mismos de la pila.

Con RET se retorna al PGM, restaurando en el IP desde la pila la dirección (0117) de la instrucción CMP CH, 01 correspondiente al paso CH – 01. El rombo simboliza que la instrucción de salto que sigue a CH – 01 permite determinar según el valor del flag Z que esta resta originó, si $CH = 01$ ó $CH = 00$.

Si $Z=1$ implica que la resta fue $01 - 01 = 0$, ó sea que la SR dejó $CH = 01$, indicación que $20 \leq N \leq 40$, por lo que se deben realizar los pasos para guardar una copia de N en la lista apuntada por DI, seguida de una copia de su número de orden i que contiene el registro DH.

Para ello en el paso $[DI] \leftarrow AX$ se escribe una copia del valor de N (que sigue guardado en AX) a partir de la dirección que indica DI (2000 para la primera vez que se encuentre un N entre 20 y 40). Luego mediante $DI \leftarrow DI + 2$ se ordena sumar 2 a DI para que la escritura siguiente siga debajo del número N recién escrito. Así, suponiendo que DI estaba en 2000, la escritura de N habrá ocupado las celdas 2000 y 2001; y si se le suma 2 a DI éste pasará a 2002, dirección a partir de la cual se puede escribir un nuevo valor sin “pisar” el valor de N antes escrito.

Esto es, si luego de cada escritura de una lista no se incrementara el registro que apunta a la última dirección donde se escribió un valor, el próximo número a escribir “pisaría” dicho valor que se había escrito.

Una vez incrementado DI, debajo de N se escribe su número de orden i en la lista indicado por DH, mediante $[DI] \leftarrow DH$, por lo que luego se incrementa DI nuevamente uno, pues el contenido de DH ocupa un byte. Siguiendo con la suposición anterior, el contenido de DH se hubiera escrito en 2002, y luego del incremento de DI (paso $DI \leftarrow DI + 1$) éste pasa a 2003 para apuntar a la dirección donde se escribirá (si aparece) otro N de la lista que está entre 20 y 40.

Con este incremento de DI terminan los pasos para el caso de que N esté entre 20 y 40, por lo que deben seguir los pasos para pedir el siguiente valor de N que corresponde analizar de la lista apuntada por SI.

De ser $Z=0$ la resta fue $00 - 01 \neq 0$ o sea que la SR dejó $CH = 00$, indicación que no es $20 \leq N \leq 40$, por lo que no se deben realizar los pasos recién descriptos para guardar $AX = N$ y $DX = i$. Si $Z=0$ estos pasos se “puentean” saltando al paso $CX \leftarrow BX$ que sigue a los mismos, a partir del cual comienzan los pasos para obtener el siguiente N a analizar.

El paso $CX \leftarrow BX$ es necesario para que CL vuelva a tener el valor n, pues la SR dejó CH con 01 ó 00, y CL no tiene más su valor inicial n luego que en el PGM se hizo $CX \leftarrow AX$.

Para obtener el próximo N se le suma 2 al valor que tenía SI, mediante el cual se localizó el último N analizado, para así poder pedir el N que sigue en la lista (paso $SI \leftarrow SI + 2$). De no incrementarse SI siempre se pediría de 2000 el mismo N . Pero antes de pedir un nuevo N hay que verificar si el N que se analizó antes no fue el último de la lista apuntada por SI. A tal fin (paso $CL \leftarrow CL - 1$), cada vez que se analizó un número de esta lista se le resta uno a CL (que en la primer vuelta tiene la cantidad n de números de la lista) de modo que CL actúe como contador regresivo. Cuando se detecta que CL vale cero implica que se analizaron todos los n números de la lista. Esta detección se realiza preguntando luego de dicha resta por el valor de Z que resultó de ella. Mientras sea $Z=0$ implica que el resultado de esa resta que quedó en CL no es cero, por lo que se debe saltar al paso 5 del diagrama donde se pide el siguiente N .

Cuando se analiza el último N de la lista en CL será $n = 1$, de modo que $CL - 1$ es $1 - 1 = 0$, resultando $Z=1$ (cero si) y por consiguiente en vez de volver a pedir el siguiente N de la lista, por ser $Z=1$ ($CL=0$) se termina la ejecución del PGM.

Si al inicio de la SR se hubiera guardado CX mediante PUSH CX, en la pila se tendría el último valor de CX antes de CALL 3500, o sea $CX = N$. Al finalizar el procedimiento de la SR resulta $CH=01$ ó $CH=00$, siendo que uno u otro valor de CH es determinado no bien se retorna al PGM. Pero si al inicio se hizo PUSH CX, antes de retornar se debe hacer POP CX, con lo cual volvería a CX el número N que se había guardado en la pila, pisando el valor de CH requerido.

Por lo tanto, no se puede guardar en la pila el registro o registros usado(s) usados para el pasaje de parámetros.

A continuación se codifica en Assembler el diagrama lógico de la figura 3.41

-A 0100		
0CC6:0100	MOVCL, [1500]	Carga en CL la cantidad n de números de la lista
0CC6:0104	MOV SI, 1000	Inicializa SI = 1000
0CC6:0107	MOV DI, 2000	Inicializa DI = 2000
0CC6:010A	MOV DH, 00	Inicializa DH = 00
0CC6:010C	MOV AX, [SI]	Lleva hacia AX una copia del número N apuntado por SI
0CC6:010E	INC DH	Cada vez que se toma un número nuevo de la lista se suma uno al contador de orden DH
0CC6:0110	MOV BX, CX	Guarda en BX una copia de CX, que luego será modificado para la SR
0CC6:0112	MOV CX, AX	Copia en CX una copia de AX, para que luego sea el dato que procesará la SR
0CC6:0114	CALL 3500	Pasa a ejecutar la SR que está en 3500
0CC6:0117	CMP CH, 01	Luego de que se ejecutó la SR, se vuelve al PGM para restar CH - 01
0CC6:011A	JNZ 0124	Si de CH - 01 fue Z=0 es que no es CH=01, o sea que no fue $20 \leq N \leq 40$, saltar a 0124
0CC6:011C	MOV [DI], AX	Copia en la lista apuntada por DI el número N comprendido entre 20 y 40
0CC6:011E	ADD DI, 2	Suma 2 a DI para preparar debajo la escritura del número de orden de N debajo de su valor
0CC6:0121	MOV [DI], DH	Escribe en la dirección apuntada por DI una copia del número de orden que indica DH
0CC6:0123	INC DI	Se actualiza DI por si en AX llega otro N comprendido entre 20 y 40
0CC6:0124	MOV CX, BX	Desde BX se restaura el valor que tenía CX antes de llamar a la SR
0CC6:0126	ADD SI, 2	Si Z=1 (AX = AX anterior) suma 2 a SI para apuntar siguiente número N de la lista
0CC6:0129	DEC CL	Resta uno a la cantidad de números que falta analizar en la lista apuntada por SI
0CC6:012B	JNZ 010C	Mientras sea Z=0 saltar a 010C para considerar el número siguiente de la lista
0CC6:012D	INT 20	Finalizar
-A 3500		
0CC6:3500	PUSHF	Guarda en la pila el valor de los Flags que usaba el programa, contenidos en el registro RE
0CC6:3501	CMP CX, 20	Resta CX - 20 para que SZVC tomen valores, siendo CX = N
0CC6:3504	JL 3520	Si S#V resulta CX < 20 entonces saltar a 3520
0CC6:3506	CMP CX, 41	Resta CX - 41 para que SZVC tomen valores
0CC6:3509	JG 3520	Si S=V resulta CX ≥ 41 (o sea CX > 40) entonces saltar a 3520
0CC6:350B	MOV CH, 01	Si S#V (CX < 41) se hace CH = 01, para indicar que $20 \leq CX \leq 40$
0CC6:350D	POPF	Pasa de la pila a RE el valor que tenían los flags en el programa antes de llamar a la SR
0CC6:350E	RET	Pasa de la pila al IP la dirección 0118 de la instrucción por la cual prosigue el programa
-A 3520		
0CC6:3520	MOV CH, 00	Si S=V (CX ≥ 41) (o sea CX > 40) se hace CH=00, para indicar que no es $20 \leq CX \leq 40$
0CC6:3522	JMP 350D	Saltar incondicionalmente a 350D

En lo que sigue, mediante el Debug se ejecuta la codificación Assembler correspondiente al diagrama de la figura 3.41 suponiendo que en la lista apuntada por SI están los siguientes n = 2 números: FFFFh = -1d y 0025h = 37d. Durante la ejecución aparecen comentarios, y resaltados los contenidos que se van apilando y quedan en la pila, así como dónde se originan y hacia dónde van los mismos u otros de interés. Para los valores apilados se indican a qué registros pertenecen, o sea también a los que deben retornar.

El último valor que queda en el SP luego de que se ejecutó la instrucción anterior indica siempre la dirección de la cima. Mediante el comando E se leen en memoria los contenidos que guarda la pila, pulsando la barra espaciadora del teclado a partir de la dirección que indica SP, a fin de lograr direcciones consecutivas crecientes desde la cima hacia abajo.

-E 1500
 0CC6:1500 00.02 (se escriben en 1500 la cantidad n = 2 de números de la lista)
 -E 1000
 0CC6:1000 FF. FF. 25. 00. (se escriben en memoria los números N1 = FFFF y N2 = 0025)
 -R IP
 IP 0100
 :0100 (se pone el registro IP en 0100 para que proporciones la dirección de la primera instrucción que está en 0100)

-R
 AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0100 NV UP EI PL NZ NA PO NC
 0CC6:0100 8A0E0015 MOV CL, [1500] DS:1500=02 (El valor 02 que está en 1500 se copia en CL)
 -T (El comando T ejecuta la instrucción que aparece arriba de T)
 AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0104 NV UP EI PL NZ NA PO NC
 0CC6:0104 BE0010 MOV SI, 1000 (El valor 1000 que llegó con la instrucción, pasó a SI)
 -T
 AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SH:1000 DI=0000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0107 NV UP EI PL NZ NA PO NC
 0CC6:0107 BF0020 MOV DI, 2000 (El valor 2000 que llegó con la instrucción, MOV DI, 2000 pasa a DI)
 AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=1000 DI:2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010A NV UP EI PL NZ NA PO NC
 0CC6:010A B600 MOV DI,00

-T (El valor 00 que llegó con la instrucción, MOV DI 00 pasa a DH que en este caso ya estaba en 00)
 AX=0000 BX=0000 CX=0002 DX=0040 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010C NV UP EI PL NZ NA PO NC
 0CC6:010C 8B04 MOV AX,[SI] DS:1000=EFF (En la dirección 1000 de memoria se tiene FF y en 1002 se tiene FF, cuyas copias pasarán hacia AX)

-T
 AX=FFFF BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010E NV UP EI PL NZ NA PO NC
 0CC6:010E FEC6 INC DH

-T (DH pasa de 00 a 01)
 AX=FFFF BX=0000 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0110 NV UP EI PL NZ NA PO NC (Estos valores de los flags de la ejecución de INC DH serán luego guardados en la pila)

-T (CX se copia en BX)
 AX=FFFF BX=0002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0112 NV UP EI PL NZ NA PO NC
 0CC6:0112 89C1 MOV CX, AX

-T (AX se copia en CX)
 AX=FFFF BX=0002 CX=FFFF DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0114 NV UP EI PL NZ NA PO NC
 0CC6:0114 E8E933 CALL 3500

-T (Se salta a la SR pues el IP pasa a 3500)
 AX=FFFF BX=0002 CX=FFFF DX=0100 SP=FFEC BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3500 NV UP EI PL NZ NA PO NC
 0CC6:3500 9C PUSHF Estos flags se guardan como RE=3202

-E FFEC (1) (Con CALL la cima pasa a FFEC y se guarda en la pila la dirección 0117 de retorno)
 0CC6:FFEC 17. 01. 00. (Con CALL la dirección de retorno 0117 de CMP CH, 01 se guarda en FFEC/D de la pila)

-T Con PUSHF la cima pasa de FFEC a FFEA

AX=FFFF BX=0002 CX=FFFF DX=0100 SP=FFEAE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3501 NV UP EI NG NZ NA PO NC
 0CC6:3501 83F920 CMP CX, 20 (Se resta FFFF – 0020)

-E FFEA (Se guardan en la pila los valores de los flags que están en RE=3202)
 0CC6:FFEA 02. 32. 17. 01. 00.

-T

AX=FFFF BX=0002 CX=FFFF DX=0100 SP=FFEA BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3504 NV UP EI NG NZ NA PO NC
 0CC6:3504 7C1A JL 3520 Con JL se salta si S≠V; como en CX – 20 fue NV (V=0) y NG (S=1) o sea S≠V se saltará a 3520

-T

AX=FFFF BX=0002 CX=FFFF DX=0100 SP=FFEA BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3520 NV UP EI NG NZ NA PO NC
 0CC6:3520 B500 MOV CH,00

-T (La mitad CH de CX queda en cero)
 AX=FFFF BX=0002 CX=00FF DX=0100 SP=FFEA BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3522 NV UP EI NG NZ NA PO NC
 0CC6:3522 EBE9 JMP 350D

-T (Salta incondicionalmente de IP=3522 a IP=350D)

AX=FFFF BX=0002 CX=00FF DX=0100 SP=FFEA BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350D NV UP EI NG NZ NA PO NC
 0CC6:350D 9D POPF Con POPF se restauran los valores de los flags que existían en el PGM luego de hacer INC DH (NG cambia a PL)

-T Con POPF la cima pasa de FFEA a FFEC

AX=FFFF BX=0002 CX=00FF DX=0100 SP=FFEC BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350E NV UP EI PL NZ NA PO NC

0CC6:350E C3 RET

-E FFEC

0CC6:FFEC 17. 01. 00.

-T (Con RET la cima pasa a FFEE y la dirección 0117 pasa de la pila al IP para retornar a la instrucción CMP CH, 01 del PGM)

AX=FFFF BX=0002 CX=00FF DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0117 NV UP EI PL NZ NA PO NC
 0CC6:0117 80FD01 CMP CH,01 (Ordena restar CH – 01 = 00 – 01 que arrojará resultado no cero)

-E FFEE .00

T

AX=FFFF BX=0002 CX=00FF DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011A NV UP EI NG NZ AC PE CY
 0CC6:011A 7508 JNZ 0124

-T Con JNZ se salta si Z=0; como luc NZ (Z=0 ó sea el resultado de CH – 01 no fue cero) se saltará a la dirección 0124

AX=FFFF BX=0002 CX=00FF DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0124 NV UP EI NG NZ AC PE CY
 0CC6:0124 89D9 MOV CX, BX

-T (CL de CX vuelve al valor 02 que tenía antes de CALL)

¹ Cuando se ejecuta PUSH F, el valor hallado que se guardó en la pila (en este ejemplo 3202 h) equivale a un número binario (00110010 00000010) que de acuerdo a la figura 3.49 por su ubicación en ésta se corresponde con los siguientes valores de los flags: C=0 (NC), Z=0 (NZ), S=0 (PL) y V=0 (NV), como puede verificarse luego de ejecutar INC DH y antes de ejecutar PUSH F.

AX=FFFF BX=0002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0126 NV UP EI NG NZ AC PE CY
 0CC6:0126 83C602 ADD SI,+02
 -T (Se suma 2 a SI para tomar el siguiente N de la lista)
 AX=FFFF BX=0002 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0129 NV UP EI PL NZ NA PO NC
 0CC6:0129 FEC9 DEC CL
 -T (Se resta uno a CL de CX pues ya se ha analizado un número de la lista y queda uno menos)
 AX=FFFF BX=0002 CX=0001 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012B NV UP EI PL NZ NA PO NC
 0CC6:012B 75E1 JNZ 010C Con JNZ se salta si Z=0; como fue NZ (Z=0 ó sea el resultado de CL – 1 no fue cero) se saltará
 -T
 AX=FFFF BX=0002 CX=0001 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010C NV UP EI PL NZ NA PO NC
 0CC6:010C 8B04 MOV AX,[SI] DS:1002=0025 (En 1002/3 de memoria se tiene 0025 que pasará a AX)
 -T
 AX=0025 BX=0002 CX=0001 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010E NV UP EI PL NZ NA PO NC
 0CC6:010E FEC6 INC DH
 -T (El número de orden que está en DH pasa de 1 a 2)
 AX=0025 BX=0002 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0110 NV UP EI PL NZ NA PO NC
 0CC6:0110 89CB MOV BX,CX Los valores de estos flags equivalen a RE=3202
 -T DE ACA EN MÁS SE REPITEN VARIOS PASOS ANTERIORES
 AX=0025 BX=0001 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0112 NV UP EI PL NZ NA PO NC
 0CC6:0112 89C1 MOV CX,AX
 -T
 AX=0025 BX=0001 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0114 NV UP EI PL NZ NA PO NC
 0CC6:0114 E8E933 CALL 3500
 -T
 AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEC BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3500 NV UP EI PL NZ NA PO NC
 0CC6:3500 9C PUSHF
 -E FFEC
 0CC6:FFEC 17. 01. 00. (Con CALL en la pila se volvió a guardar 0117, dirección de retorno al PGM)
 -T
 AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3501 NV UP EI PL NZ NA PO NC
 0CC6:3501 83F920 CMP CX,+20
 -E FFEA
 0CC6:FFEA 02. 32. 17. 00. (Con PUSHF en la pila se guarda RE=3202 con los valores de los flags de INC DH)
 -T
 AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3504 NV UP EI PL NZ NA PE NC
 0CC6:3504 7C1A JL 3520 Con JL se salta si S≠V; como en CX – 20 fue NV (V=0) y PL (S=0) o sea S=V no se salta
 -T
 AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3506 NV UP EI PL NZ NA PE NC
 0CC6:3506 83F941 CMP CX,+41
 -T
 AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3509 NV UP EI NG NZ NA PE CY
 0CC6:3509 7F15 JG 3520 Con JG se salta si S=V; como en CX – 41 fue NV (V=0) y NG (S=1) o sea S≠V no se salta
 -T
 AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350B NV UP EI NG NZ NA PE CY
 0CC6:350B B501 MOV CH,01
 -T (Se hace CH=01, indicando que 0025 está entre 20 y 40)
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350D NV UP EI NG NZ NA PE CY
 0CC6:350D 9D POPF
 -T
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEC BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350E NV UP EI PL NZ NA PO NC
 0CC6:350E C3 RET
 -E FFEC
 0CC6:FFEC 17. 01. 00. (El SP vuelve a su valor original)
 -T
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0117 NV UP EI PL NZ NA PO NC
 0CC6:0117 80FD01 CMP CH,01

-E FFEE
0CC6:FFEE 00.

-T (Con CMP CH, 01 se resta CH - 01, en este caso 01 - 01 = 0 debiendo resultar ZR (zero resultado, o sea Z=1)
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011A NV UP DI PL ZR NA PE NC
 0CC6:011A 7508 JNZ 0124

-T (Como fue ZR=resultado cero, o sea Z=1, no se saltará y se pasa a la instrucción siguiente)
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011C NV UP DI PL ZR NA PE NC
 0CC6:011C 8905 MOV [DI], AX DS:2000=6320 (valor "basura" anterior en 2000/1)

-T
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 RS=0CC6 SS=0CC6 CS=0CC6 IP=011E NV UP DI PL ZR NA PE NC
 0CC6:011E 83C702 ADD DI,+02

-E 2000 (Se verifica que una copia de 0025 paso de AX a la dirección 2000 de memoria apuntada por DI)
 0CC6:2000 25. 00.

-T (DI se le sumó 2 para no pisar en la próxima escritura lo que ya está en 2000/1)
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2002
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0121 NV UP DI PL NZ NA PO NC
 0CC6:0121 8835 MOV [DI], DH DS:2002=61

-T
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2002
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0123 NV UP DI PL NZ NA PO NC
 0CC6:0123 47 INC DI

-E 2002 (Se verifica que en 2002 pasó una copia de CH=2, número de orden del número que estaba entre 20 y 40)
 -T (DI se le sumó 1 para no pisar en la próxima escritura lo que ya está en 2002)
 AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2003
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0124 NV UP DI PL NZ NA PE NC
 0CC6:0124 89D9 MOV CX, BX

-T (CL de CX vuelve al valor 01 que tenía antes de CALL)
 AX=0025 BX=0001 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2003
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0126 NV UP DI PL NZ NA PE NC
 0CC6:0126 83C602 ADD SI,+02

-T (SI pasó a 1004, dirección que no se usará)
 AX=0025 BX=0001 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0129 NV UP DI PL NZ NA PO NC
 0CC6:0129 FEC9 DEC CL

-T (Se hizo 1 - 1 = 0 con lo cual CL=0 y Z=1=ZR, o sea que se tomaron los n números de la lista)
 AX=0025 BX=0001 CX=0000 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012B NV UP EI PL ZR NA PE NC
 0CC6:012B 75DF JNZ 010C Con JNZ se salta si Z=0; como fue ZR (Z=1 ó sea el resultado de CL - 1 fue cero) no se salta

-T
 AX=0025 BX=0001 CX=0000 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012D NV UP EI PL ZR NA PE NC
 0CC6:012D CD20 INT 20 (Finaliza el programa)

Habiendo ejecutado el programa con el Debug, sistematizaremos los resultados obtenidos luego de ejecutar CALL 3500, PUSHF, POPF y RET.

Como puede observarse, luego de la ejecución de CALL 3500 el SP pasó de FFEE a FFEC, y se apiló la dirección 0117 que es la dirección 0114 (que indicaba el IP donde está CALL 3500) más 3, que es la cantidad de celdas que ocupa el código E8E933 de CALL 3500 (siendo 0117h + 33E9h = 3500h).

Esto se esquematiza en la fig 3.42. Para hacerlo, cuando la UC ejecuta CALL realiza los siguientes 4 pasos:

1. Suma 3 al IP (IP = 0117 = 0114 + 3) antes de que pase a 3500 o sea se hace (IP ← IP + 3)
2. Resta 2 al SP (SP ← SP - 2) resultando FFEC = FFEE - 2 para que 0117 se empiece a escribir en la pila desde FFEC. Si se escribiera 0117 a partir de FFEE ocuparía FFEE y FFEF en vez de apilar hacia arriba.
3. Se escribe 0117 en memoria a partir de la dirección FFEC (cima de la pila) apuntada por SP, que conforme a la simbología usada en los diagramas lógicos simbolizaremos [SP] ← IP (IP = 0117)
4. Al IP pasa el valor 3500 = 0117 + 33E9 para que se pase a ejecutar la primera instrucción de la subrutina

La instrucción PUSHF al igual que PUSH AX, PUSH BX, etc., se ejecuta mediante los pasos 2 y 3 que implican el apilamiento de información (figura 3.42), por lo que adaptaremos los anteriores a esta instrucción.

1. Resta 2 al SP (SP ← SP - 2) resultando FFEA = FFEC - 2 para que 3202 se empiece a escribir en la pila desde FFEA. Si se escribiera 3202 a partir de FFEE "pisaría" el valor 0117 antes apilado.
2. Se escribe 3202 en memoria a partir de la dirección FFEA (cima de la pila) apuntada por SP, que conforme a la simbología usada en los diagramas lógicos simbolizaremos [SP] ← RE (RE = 3202)

Obsérvese que el valor de SP cambia *automáticamente* mientras se ejecutan CALL y PUSH, por lo que no es necesario insertar en el programa ninguna instrucción que ordene modificar el valor del SP.

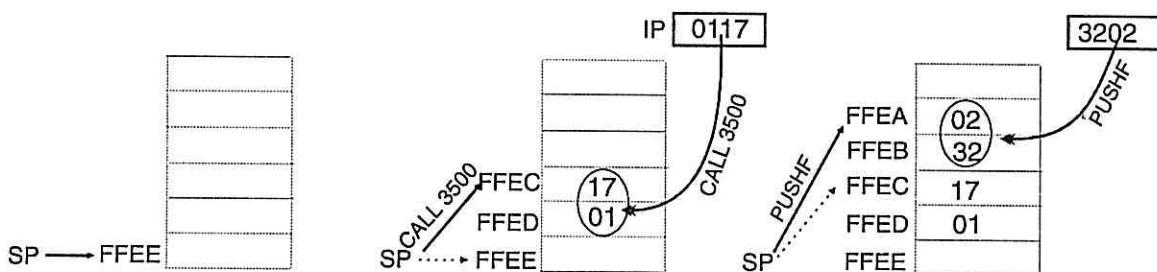


Figura 3.42

La ejecución de POPF (figura 3.43) al igual que cualquier otro POP, supone devolver o restaurar el valor que tenía RE (o el registro que indica POP) en relación con el PGM antes de la ejecución de CALL, a fin de seguir con la ejecución del PGM con los valores de los registros como los había dejado el PGM, salvo el registro "buzon" (CX en este caso) dada esta forma elegida de pasar parámetros.

POPF es contraria a PUSHF, o sea produce el desapilamiento de 3202 de RE que volverá a RE. Como aparece en el Debug, de NV UP EI NG NZ NA PO NC se pasa a NV UP EI PL NZ NA PO NC, valores que corresponden a RE=3202. Si bien 3202 seguirá en memoria hasta que no sea "pisado", pero que no formará más parte de la pila cuya nueva cima es FFEC, dado que SP con POPF pasó de FFEA a FFEC como indica el Debug. Los 2 pasos de la ejecución de POPF (válidos para cualquier otro POP salvo el registro involucrado) son:

1. RE ← [SP] en este caso RE ← [FFEA] puesto que antes de ejecutar POPF era SP=FFEAE, o sea que estaba apuntando a 0232, valor que debe ser leído a partir de FFEAE, por lo que primero debe hacerse esta lectura para que 3202 pase a RE (figura 3.43).
2. SP ← SP + 2 en este caso FFEAE + 2 = FFEC : a fin de desapilar hay que sumarle 2 a SP a fin de que la nueva cima sea otra vez FFEC, de modo que 3202 no forme parte de la pila. Si se hiciera SP ← SP + 2 antes de RE ← [SP] se hubiera pasado a RE en lugar de 3202 el valor 0117 que está a partir de FFEC.

Con RET el PGM debe reanudarse con la instrucción que sigue a CALL, cuya dirección (0117) se guardó en la pila con la ejecución de CALL, y los dos pasos para hacer este desapilamiento son como los de POPF:

1. IP ← [SP] en este caso IP ← [FFEC] puesto que antes de ejecutar RET era SP=FFEC, o sea que estaba apuntando a 0117, valor que debe ser leído a partir de FFEC, por lo que primero debe hacerse esta lectura para que 0117 pase a IP (figura 3.43).
2. SP ← SP + 2 en este caso FFEC + 2 = FFEE : a fin de desapilar hay que sumarle 2 a SP a fin de que la nueva cima sea otra vez FFEE, de modo que 0117 no forme parte de la pila. Si se hiciera SP ← SP + 2 antes de IP ← [SP] se hubiera pasado a IP en lugar de 0117 el valor "basura" que está a partir de FFEE.



Figura 3.43

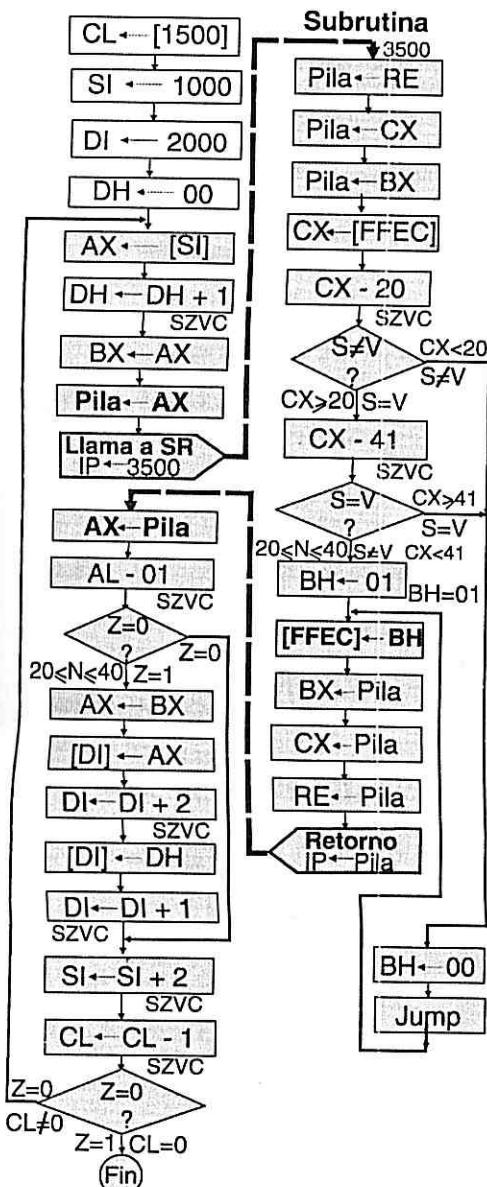
Sistematizando el llamado a una SR y el funcionamiento de la pila:

- La pila guarda temporariamente datos en forma ordenada, mientras se ejecuta una SR o un anidamiento de sucesivas llamadas a SR o interrupciones.
- Los datos tienen una restricción de acceso: sólo pueden agregarse o eliminarse por un extremo de la pila o "cima" cuya dirección la proporciona el registro SP del PGM que llamó a la SR1.
- 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.

- 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, siendo que los mismos dejar de formar parte de la pila, aunque queden transitoriamente en memoria. (En "modo real" se apilan o desapilan 2 bytes por vez).
- El SP controla el orden de la pila, y el SP es controlado *automáticamente* por la UC durante la ejecución de instrucciones que afectan a la pila (como CALL, PUSH, POP y RET).
- Cuando un PGM llama a una SR debe dejar, según se establezca, en registros o bien en la pila los datos que la SR procesará, y tomará de registros o bien de la pila, los resultados que necesita ("pasaje de parámetros").

EJERCICIO 46.b Por pasaje de parámetros a través de la pila

Con esta forma de pasar parámetros no es necesario que el PGM y la SR tengan registros compartidos: quien desarrolla el PGM puede usar los registros que quiera, sin preocuparse por los que están involucrados en la SR; y la persona que genera una SR no necesita imponer qué registros se deben usar como intermediarios en la comunicación con un PGM. En esta alternativa sólo es necesario que antes de llamar a la SR mediante CALL se deben insertar en el PGM instrucciones PUSH (usando el registro que quiera el programador) para que en un orden prefijado por quién desarrolló la SR se apilen en la pila los datos que la SR debe procesar. Asimismo, luego del CALL, para cuando se retorne al PGM, deben seguir instrucciones POP para tomar el o los resultados que obtuvo la SR de los mismos "buzones" de la pila donde se dejaron los datos a procesar para la SR.



Es como si se tratara de dos habitaciones separadas (una para el PGM y otra para la SR) que se comunican a través de agujeros o buzones en la pared que las separa para pasarse datos y resultados.

Dado que quien desarrolla la SR no sabe qué registros usará la persona que generará el PGM, en el inicio de la SR debe guardar en la pila mediante instrucciones PUSH todos los registros que usará la SR, incluido el RE, aunque algunos de ellos o todos no sean usados por el PGM llamador.

Para nuestro caso la SR además usar RE y CX, como se hizo antes, empleará a BX con el fin de dar más generalidad a lo exemplificado, pero no se usará CX como registro común. Por tal motivo la SR empieza con PUSH F, PUSH CX y PUSH BX. A medida que se ejecutan con el Debug estas instrucciones de la SR, se examinará la memoria para ver cómo se van apilando los valores que esos registros tenían antes de que comience el procedimiento que lleva a cabo la SR, o sea los valores con los que el PGM dejó a dichos registros antes de llamar mediante CALL a la SR.

Estos valores serán restaurados en esos registros cuando al final de la SR se ejecuten las correspondientes instrucciones POP en el orden adecuado. Así cuando se vuelva a la instrucción que en el PGM sigue a CALL, el PGM se reanudará como si la SR no hubiera existido.

Se supondrá (figura 3.44) que el número N a analizar deberá ir a la base de la pila (en este caso en las direcciones FFEC/D) mediante la instrucción PUSH AX del PGM que está antes de CALL 3500. La SR mediante MOV CX, [FFEC] copiará en CX este número N que dejó el PGM.

Una vez que la SR analizó el valor de N, dejará en la dirección FFEC mediante MOV [FFEC], BH el valor 01 para indicar que N está entre 20 y 40, y la indicación 00 en caso contrario.

El PGM empieza con las mismas 6 primeras instrucciones que se indican en las figuras 3.32 y 3.41.

Luego sigue MOV BX, AX que resguarda en BX una copia de AX, pues como la instrucción siguiente es PUSH AX recién citada para enviar a la base de la pila una copia de N, cuando después de ejecutar la SR se vuelve a la instrucción POP AX del PGM, una copia del contenido de las direcciones FFEC/D de la pila irá hacia AX, por lo cual en AX no estará más el valor N que se envió a la pila.

Si no se hubiera hecho antes MOV BX, AX no habría forma de recuperar dicho valor N, y por lo tanto por ejemplo no se podría guardar en la lista apuntada por DI si N está entre 20 y 40.

POP AX sirve para pasar desde la pila a la mitad AL de AX la indicación 01 ó 00 antes citada (que llegó a la pila con MOV [FFEC], BH de la SR) y también se cumple que no puede haber un PUSH sin el POP correspondiente, so pena de quebrar el orden LIFO con que se escriben y leen los contenidos de la pila.

Figura 3.44

Luego de POP AX el PGM determina con CMP AL, 01 y JNZ qué informó la SR: si N está o no entre 20 y 40. De ser $20 \leq N \leq 40$, con MOV AX, BX el registro BX devolverá a AX el valor N que tenía, y el PGM seguirá de forma parecida al ejercicio anterior.

Es importante notar que los PUSH con que empieza la SR sirven para guardar todos los registros que ella va a usar; mientras que los PUSH del PGM llamador se usan para dejar dato(s) en el “buzón” de la pila.

La codificación en Assembler de la figura 3.44 resulta:

-A 0100		
0CC6:0100	MOV CL,[1500]	Carga en CL la cantidad n de números de la lista
0CC6:0104	MOV SI, 1000	Inicializa SI=1000
0CC6:0107	MOV DI, 2000	Inicializa DI=2000
0CC6:010A	MOV DH, 00	Inicializa DH = 00
0CC6:010C	MOV AX, [SI]	Lleva hacia AX una copia del número N apuntado por SI
0CC6:010E	INC DH	Cada vez que se toma un número nuevo de la lista se suma uno al contador de orden DH
0CC6:0110	MOV BX, AX	Guarda en BX una copia de AX, que luego será modificado para la SR
0CC6:0112	PUSH AX	Pasa una copia de AX=N a la pila (direcciones FFEC/D)
0CC6:0113	CALL 3500	Pasa a ejecutar la SR que está en 3500 guardando también en la pila la dirección 0116
0CC6:0116	POP AX	Luego de ejecutar la SR, se vuelve al PGM para pasar de la pila a AX la indicación de la SR
0CC6:0117	CMP AL, 01	Restar AL - 01 (01 en AL indica que la SR detectó que $20 \leq N \leq 40$, caso contrario es AL=00)
0CC6:0119	JNZ 0125	Si de AL - 01 fue Z=0 implica que no fue AL=01, o sea que no fue $20 \leq N \leq 40$, saltar a 0150
0CC6:011B	MOV AX, BX	Desde BX se restaura el valor que tenía AX antes de llamar a la SR
0CC6:011D	MOV [DI], AX	Copia en la lista apuntada por DI el número N comprendido entre 20 y 40
0CC6:011F	ADD DI, 2	Suma 2 a DI para preparar debajo la escritura del número de orden de N debajo de su valor
0CC6:0122	MOV [DI], DH	Escribe en la dirección apuntada por DI una copia del número de orden que indica DH
0CC6:0124	INC DI	Se actualiza DI por si en AX llega otro N comprendido entre 20 y 40
0CC6:0125	ADD SI, 2	Suma 2 a SI para apuntar siguiente número N de la lista
0CC6:0128	DEC CL	Resta uno a la cantidad de números que falta analizar en la lista apuntada por SI
0CC6:012A	JNZ 010C	Mientras sea Z=0 saltar a 010D para considerar el número siguiente de la lista
0CC6:012C	INT 20	Fin
-A 3500		
0CC6:3500	PUSHF	Guarda en la pila el valor de los Flags que usaba el PGM, contenidos en el registro RE
0CC6:3501	PUSH CX	Guarda en la pila el contenido de CX que usaba el PGM pues será modificado por la SR
0CC6:3502	PUSH BX	Guarda en la pila el contenido de BX que usaba el PGM pues será modificado por la SR
0CC6:3503	MOV CX, [FFEC]	Pasa a CX una copia del número N que antes desde AX se pasó a la pila en FFEC/D
0CC6:3507	CMP CX, 20	Resta CX - 20 para que SZVC tomen valores, siendo CX = N
0CC6:350A	JL 3520	Si S=V resulta CX ≥ 41 (o sea CX > 40) entonces saltar a 3520
0CC6:350C	CMP CX, 41	Resta CX - 41 para que SZVC tomen valores
0CC6:350F	JG 3520	Si S=V resulta CX ≥ 41 (o sea CX > 40) entonces saltar a 3520
0CC6:3511	MOV BH, 01	Si S \neq V (CX<41) se hace BH=01, para indicar que $20 \leq CX \leq 40$
0CC6:3513	MOV [FFEC], BH	Envía a la pila (en FFEC/D) una copia de la indicación de BH acerca de N
0CC6:3517	POP BX	Pasa de la pila a BX el valor que tenía BX en el programa antes de llamar a la SR
0CC6:3518	POP CX	Pasa de la pila a CX el valor que tenía CX en el programa antes de llamar a la SR
0CC6:3519	POPF	Pasa de la pila a RE el valor que tenían los flags en el programa antes de llamar a la SR
0CC6:351A	RET	Pasa de la pila al IP la dirección 0117 de la instrucción por la cual prosigue el programa
-A 3520		
0CC6:3520	MOV BH, 00	Si S=V (CX ≥ 41) (o sea CX > 40) se hace BH=00, para indicar que no es $20 \leq CX \leq 40$
0CC6:3522	JMP 3513	Saltar incondicionalmente a 3513
-E 1500		
0CC6:1500	00.02	
-E 1000		
0CC6:1000	14. 22. 25. 00. (los 2 datos escritos son 2214 y 0025)	
-RIP		
IP 0100		
:0100		
-R		
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000		
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0100 NV UP EI PL NZ NA PO NC		
0CC6:0100 8A0E0015 MOV CL, [1500] DS:1500=		
-T		
AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000		
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0104 NV UP EI PL NZ NA PO NC		
0CC6:0104 BE0010 MOV SI, 1000		

```

-T
AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=1000 DI=0000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0107 NV UP EI PL NZ NA PO NC
0CC6:0107 BF0020 MOV DI, 2000

-T
AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010A NV UP EI PL ZR NA PE NC
0CC6:010A B600 MOV DH, 00

-T
AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010C NV UP EI PL ZR NA PE NC
0CC6:010C 8B04 MOV AX,[SI] DS:1000=2214

-T
AX=2214 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010E NV UP EI PL ZR NA PE NC
0CC6:010E FEC6 INC DH

-T
AX=2214 BX=0000 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0110 NV UP EI PL NZ NA PO NC
0CC6:0110 89C3 MOV BX, AX

-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0112 NV UP EI PL NZ NA PO NC
0CC6:0112 50 PUSH AX

-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFEC BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0113 NV UP EI PL NZ NA PO NC
0CC6:0113 E8EA33 CALL 3500

-E FFEC
0CC6:FFEC 14. 22. 00.

-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFEA BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3500 NV UP EI PL NZ NA PO NC
0CC6:3500 9C PUSHF

-E FFEA
0CC6:FFEA 16. 01. 14. 22. 00.

-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFE8 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3501 NV UP EI PL NZ NA PO NC
0CC6:3501 51 PUSH CX

-E FFE8
0CC6:FFE8 02. 32. 16. 01. 14. 22. 00.

-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFE6 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3502 NV UP EI PL NZ NA PO NC
0CC6:3502 53 PUSH BX

-E FFE8
0CC6:FFE8 02. 00. 02. 32. 16. 01. 14. 22. 00.

-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3503 NV UP EI PL NZ NA PO NC
0CC6:3503 8B0EECFF MOV CX,[FFEC] DS:FFEC=2214

-E FFE4
0CC6:FFE4 14. 22. 02. 00.
0CC6:FFE8 02. 32. 16. 01. 14. 22. 00.

-T
AX=2214 BX=2214 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3507 NV UP EI PL NZ NA PO NC
0CC6:3507 83F920 CMP CX,+20

-T
AX=2214 BX=2214 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350A NV UP EI PL NZ NA PO NC
0CC6:350A 7C14 JL 3520

-T
AX=2214 BX=2214 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350C NV UP EI PL NZ NA PO NC
0CC6:350C 83F941 CMP CX,+41

-T
AX=2214 BX=2214 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350F NV UP EI PL NZ NA PO NC
0CC6:350F 7F0F JG 3520

-T
AX=2214 BX=2214 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000

```

```

DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3520 NV UP EI PL NZ NA PO NC
0CC6:3520 B700 MOV BH, 00
-T
AX=2214 BX=0014 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3522 NV UP EI PL NZ NA PO NC
0CC6:3522 EBEF JMP 3513
-T
AX=2214 BX=0014 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3513 NV UP EI PL NZ NA PO NC
0CC6:3513 883EECFF MOV [FFEC], BH DS:FFEC=14
-E FFEC
0CC6:FFEC 14. 22.
-T
AX=2214 BX=0014 CX=2214 DX=0100 SP=FFE4 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3517 NV UP EI PL NZ NA PO NC
0CC6:3517 5B POP BX
-E FFEC
0CC6:FFEC 00. 22. (En lugar de N = 2214 se almacenó el resultado 2200 con CL = 00)
-T
AX=2214 BX=2214 CX=2214 DX=0100 SP=FFE6 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3518 NV UP EI PL NZ NA PO NC
0CC6:3518 59 POP CX
-E FFE6
0CC6:FFE6 02. 00.
0CC6:FFE8 02. 32. 16. 01. 00. 22. 00.
-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFE8 BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3519 NV UP EI PL NZ NA PO NC
0CC6:3519 9D POPF
-E FFE8
0CC6:FFE8 02. 32. 16. 01. 00. 22. 00.
-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFEA BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=351A NV UP EI PL NZ NA PO NC
0CC6:351A C3 RET
-E FFEA
0CC6:FFEA 16. 01. 00. 22. 00.
-T
AX=2214 BX=2214 CX=0002 DX=0100 SP=FFEC BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0116 NV UP EI PL NZ NA PO NC
0CC6:0116 58 POP AX
-E FFEC
0CC6:FFEC 00. 22. 00.
-T
AX=2200 BX=2214 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0117 NV UP EI PL NZ NA PO NC
0CC6:0117 3C01 CMP AL, 01
-T
AX=2200 BX=2214 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0119 NV UP EI NG NZ AC PE CY
0CC6:0119 750A JNZ 0125
-T
AX=2200 BX=2214 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1000 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0125 NV UP EI NG NZ AC PE CY
0CC6:0125 83C602 ADD SI,+02
-T
AX=2200 BX=2214 CX=0002 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0128 NV UP EI PL NZ NA PO NC
0CC6:0128 FEC9 DEC CL
-T
AX=2200 BX=2214 CX=0001 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012A NV UP EI PL NZ NA PO NC
0CC6:012A 75E0 JNZ 010C
-T
AX=2200 BX=2214 CX=0001 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010C NV UP EI PL NZ NA PO NC
0CC6:010C 8B04 MOV AX,[SI] DS:1002=0025
-T
AX=0025 BX=2214 CX=0001 DX=0100 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=010E NV UP EI PL NZ NA PO NC
0CC6:010E FEC6 INC DH
-T
AX=0025 BX=2214 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000

```

DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0110 NV UP EI PL NZ NA PO NC
 0CC6:0110 89C3 MOV BX, AX

-T

AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0112 NV UP EI PL NZ NA PO NC
 0CC6:0112 50 PUSH AX

-T

AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEC BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0113 NV UP EI PL NZ NA PO NC
 0CC6:0113 E8EA33 CALL 3500

-E FFEC

0CC6:FFEC 25. 00. 00.

-T

AX=0025 BX=0025 CX=0001 DX=0200 SP=FFE8 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3500 NV UP EI PL NZ NA PO NC
 0CC6:3500 9C PUSHF

-E FFEA

0CC6:FFE8 16. 01. 25. 00. 00.

-T

AX=0025 BX=0025 CX=0001 DX=0200 SP=FFE8 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3501 NV UP EI PL NZ NA PO NC
 0CC6:3501 51 PUSH CX

-E FFE8

0CC6:FFE8 02. 32. 16. 01. 25. 00. 00.

-T

AX=0025 BX=0025 CX=0001 DX=0200 SP=FFE6 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3502 NV UP EI PL NZ NA PO NC
 0CC6:3502 53 PUSH BX

-E FFE6

0CC6:FFE6 01. 00.

0CC6:FFE8 02. 32. 16. 01. 25. 00. 00.

-T

AX=0025 BX=0025 CX=0001 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3503 NV UP EI PL NZ NA PO NC
 0CC6:3503 8B0F EECFF MOV CX, [FFEC] DS:FFEC=0025

-E FFE4

0CC6:FFE4 25. 00. 01. 00.

0CC6:FFE8 02. 32. 16. 01. 25. 00. 00.

En la figura 3.45 aparece el apilamiento hasta la cima SP=FFE4 luego de ejecutar PUSH BX

-T

AX=0025 BX=0025 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3507 NV UP EI PL NZ NA PO NC
 0CC6:3507 83F920 CMP CX, +20

-T

AX=0025 BX=0025 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350A NV UP EI PL NZ NA PE NC
 0CC6:350A 7C14 JL 3520

-T

AX=0025 BX=0025 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350C NV UP EI PL NZ NA PE NC
 0CC6:350C 83F941 CMP CX,+41

-T

AX=0025 BX=0025 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350F NV UP EI NG NZ NA PE CY
 0CC6:350F 7F0F JG 3520

-T

AX=0025 BX=0025 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3511 NV UP EI NG NZ NA PE CY
 0CC6:3511 B701 MOV BH, 01

-T

AX=0025 BX=0125 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3513 NV UP EI NG NZ NA PE CY
 0CC6:3513 883 EECFF MOV [FFEC], BH DS:FFEC=25

-T

AX=0025 BX=0125 CX=0025 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
 DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3517 NV UP EI NG NZ NA PE CY
 0CC6:3517 5B POP BX

-E FFEC ▼ (Con MOV [FFEC], BH en FFEC de la pila, donde está AL, su valor 25 fue reemplazado por el resultado 01 de la SR)
 0CC6:FFEC 01. 00.

-E FFE4

0CC6:FFE4 25. 00. 01. 00.

Así estaba la pila antes de ejecutar POP BX

0CC6:FFE8 02. 32. 16. 01. 01. 00. 00.

-T
AX=0025 BX=0025 CX=0025 DX=0200 SP=FFE6 BP=0000 SI=1002 DI=2000 (A BX volvió desde la pila 0025)
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3518 NV UP EI NG NZ NA PE CY
0CC6:3518 59 POP CX
-E FFE6
0CC6:FFE6 01. 00.
0CC6:FFE8 02. 32. 16. 01. 01. 00. 00.

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFE8 BP=0000 SI=1002 DI=2000 (A CX volvió desde la pila 0001)
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3519 NV UP EI NG NZ NA PE CY
0CC6:3519 9D POPF
-E FFE8
0CC6:FFE8 02. 32. 16. 01. 01. 00. 00.

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEA BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=351A NV UP EI PL NZ NA PO NC
0CC6:351A C3 RET
-E FFEA
0CC6:FFEA 16. 01. 01. 00. 00.

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEC BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0116 NV UP EI PL NZ NA PO NC (A IP volvió desde la pila 0116)
0CC6:0116 58 POP AX
-E FFEC
0CC6:FFEC 01. 00. 00.

-T
AX=0001 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0117 NV UP EI PL NZ NA PO NC
0CC6:0117 3C01 CMP AL,01

-T
AX=0001 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0119 NV UP EI PL ZR NA PE NC
0CC6:0119 750A JNZ 0125

-T
AX=0001 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011B NV UP EI PL ZR NA PE NC
0CC6:011B 89D8 MOV AX, BX

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011D NV UP EI PL ZR NA PE NC
0CC6:011D 8905 MOV [DI], AX DS:2000=0025

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011F NV UP EI PL ZR NA PE NC
0CC6:011F 83C702 ADD DI, +02

-E 2000
0CC6:2000 25. 00. (En 2000/1 de la lista apuntada por DI esta 0025)

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2002
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0122 NV UP EI PL NZ NA PO NC
0CC6:0122 8835 MOV [DI], DH DS:2002=02

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2002
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0124 NV UP EI PL NZ NA PO NC
0CC6:0124 47 INC DI

-E 2002
0CC6:2002 02. (En 20002 de la lista apuntada por DI esta 2, número de orden de N2)

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0125 NV UP EI PL NZ NA PE NC
0CC6:0125 83C602 ADD SI, +02

-T
AX=0025 BX=0025 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0128 NV UP EI PL NZ NA PO NC
0CC6:0128 FEC9 DEC CL

-T
AX=0025 BX=0025 CX=0000 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012A NV UP EI PL ZR NA PE NC
0CC6:012A 75E0 JNZ 010C

-T
AX=0025 BX=0025 CX=0000 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012C NV UP EI PL ZR NA PE NC
0CC6:012C CD20 INT 20

MEJORAS EN LA SUBRUTINA DEL EJERCICIO ANTERIOR

La SR de la codificación anterior tiene la limitación que usa las direcciones fijas FFEC/D de la pila establecidas como "buzón". Esto implica que así la pila no se puede cambiar de lugar por el sistema operativo que administra la memoria. Para solucionar esto, se localizan los "buzones" a partir de la dirección que tiene la cima de la pila, en nuestro caso después de ejecutar las instrucciones PUSH AX, CALL 3500, PUSH F, PUSH CX y PUSH BX la cima llega a su altura máxima, cuya dirección indicada por el SP se mantiene durante todo el procedimiento de la SR.

En la SR anterior mediante MOV CX, [FFEC] se tomaba de la pila el valor N a analizar, y luego se enviaba al "buzón" el 01 ó 00 del resultado mediante MOV [FFEC], BH. Después la pila va cambiando con cada POP que se ejecuta.

Independientemente de las direcciones que ocupan en la pila los registros que ella guarda, el programador de la SR puede determinar la cantidad de celdas que existen entre la dirección de la cima y la del "buzón", luego que se apiló todo lo que había que apilar después del último PUSH de la SR (incluido lo que ordenó el PGM apilar en el "buzón" con PUSH AX). En nuestro ejemplo (figura 3.45) puede determinarse que: dirección buzón = FFEC = FFE4 + 8, y si la pila cambia de lugar siempre seguirá siendo: dirección buzón = dirección cima + 8

Como el registro SP siempre provee la dirección de la cima de la pila, en principio podría pensarse en reemplazar MOV CX, [FFEC] por MOV CX, [SP + 8], pero por seguridad esta instrucción no existe.

Para la pila existe un registro auxiliar BP ("Base Pointer", que se puede codificar en Assembler entre corchetes, al igual que SI, DI y BX, para apuntar a direcciones de la pila, como se requiere en este caso.

O sea que es posible codificar una instrucción como MOV CX, [BP + 8], pero previamente en BP hay que copiar el valor que tiene SP que está apuntando a la cima, mediante la instrucción MOV BP, SP como se indica más abajo.

A su vez la utilización del registro BP trae aparejada la necesidad de guardarlo en la pila mediante PUSH BP, pues se deben guardar todos los registros que va a usar la SR, dado que algunos o todos pueden estar siendo usados por el PGM. Este nuevo apilamiento (en punteado en fig. 3.45) hace que **durante el procedimiento de la SR sea siempre en nuestro caso: dirección buzón = dirección cima + 10 = dirección cima + A** (en el Debug los números, están en hexa: 10d = Ah). Con esta mejora la pila de un programa podrá ser ubicada por el sistema operativo en cualquier lugar de memoria, para lo cual el núcleo de la SR quedará en definitiva como sigue, con las nuevas instrucciones en negrita:

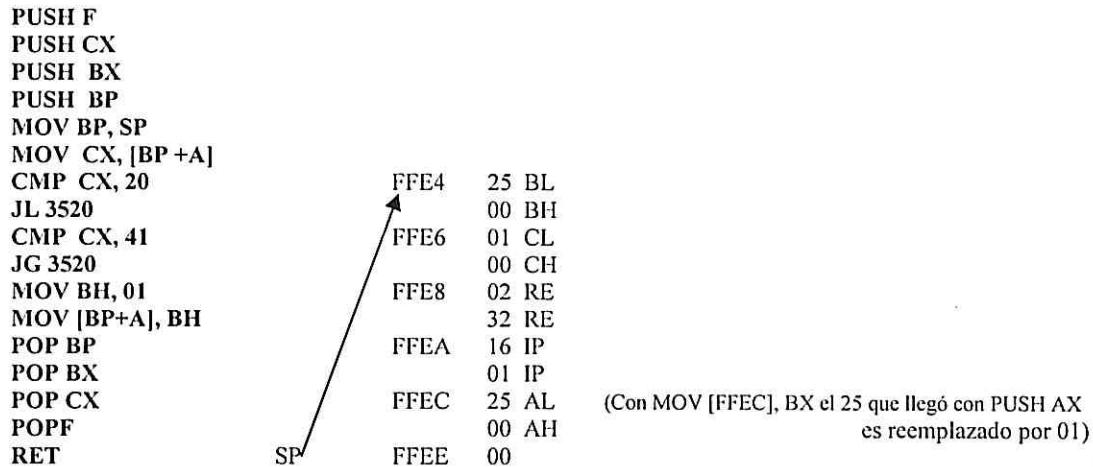


Figura 3.45

EJERCICIO 47:

A partir de la dirección 2000 existe una lista de números naturales de 2 bytes, cuya cantidad 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 41

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

A 0100		
xxxx:0100 MOV CL, [1500]		Carga en CL la cantidad n de números 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 010A		Mientras Z=0, volver a 010A
xxxx:0120 INT 20		Fin

A 5000		
xxxx:5000	PUSHF	Guarda los flags (Registro de Estado) en la pila ¹
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 $C8_{16} = 200_D$
xxxx:500B	DIV CX	Divide por 200, y el resultado queda en AX
xxxx:010D	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

A 0100		
xxxx:0100	MOV CL, [1500]	Carga en CL la cantidad n de números 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) ²
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) ³
xxxx:5001	PUSH AX	Guarda AX en la pila (designado por sus porciones AH y AL en la fig. 3.33) ³
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

¹ 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.

² Indicado como AH y AL en la base de la pila de la figura 3.46

³ 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
```

(Continuación de la codificación de la hoja anterior)

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 ¹
xxxx:500C MOV CX, C8	El divisor CX toma el valor $C8_{16} = 200_D$
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 ²
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

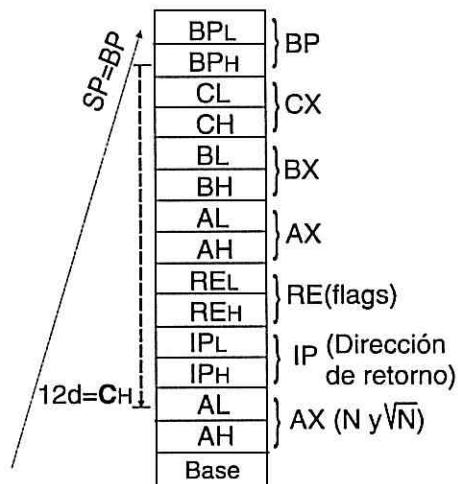


Figura 3.46

STACK OVERFLOW

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.

Si por algún error de programación este orden falla o no termina, la pila puede crecer de forma descontrolada, dando lugar a lo que se conoce como “stack overflow” (desborde de pila), situación que detecta el sistema operativo.

¹ Figura 3.46: 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).

² Serán los nuevos valores que toman los denominados AH y AL en la base de la pila de la figura 3.46

INTERRUPCIONES POR SOFTWARE

Al estudiar las llamadas a SR mediante la instrucción CALL se vió que las instrucciones del PGM que seguían a CALL debían esperar para ser ejecutadas que previamente se ejecuten las instrucciones de la SR. Todo sucede como si el PGM mediante CALL se hubiera autointerrumpido.

Asimismo, la SR que se ejecutaba no era del SO (sistema operativo) sino que proporcionaba una función (como ser para hallar raíces cuadradas de números) que ya estaba construida, evitando al programador tener que desarrollarla con el lenguaje de programación que utiliza.

Las interrupciones por software que se llevan a cabo mediante las instrucciones como "Supervisor Call" para UCPs de IBM, o "Trap" o INTxx para UCP de Intel (donde xx es un número en hexa) llaman a SR del SO o del BIOS ya construidas, muy complejas para desarrollar por usuarios. También el PGM luego de ellas queda autointerrumpido, esperando que se termine de ejecutar la SR del SO o del BIOS que da servicio.

Estas subrutinas son llamadas para que un PGM de usuario pueda usar recursos que el SO administra y protege, tales como archivos y periféricos (estos últimos mediante SR conocidas como "drivers" (manejadores). Por ejemplo, cuando a través de una interfaz gráfica de Windows con el mouse seleccionamos "abrir", "guardar" o "guardar como" o "imprimir", se pone en ejecución una SR que contiene una instrucción INTxx para llamar a otra SR que en cada caso da servicio a nuestra petición. Igualmente, si se desarrolla un PGM con sentencias tipo READ o WRITE (para leer o guardar un archivo) o WRITLN (para imprimir), cuando este PGM sea traducido a código de máquina aparecerá también en cada caso una instrucción INTxx.

En "Periféricos y Redes Locales" del autor, se detallan procesos de lectura y escritura de discos que se inicien con una instrucción INT.

El número xx citado sirve para localizar en memoria de manera indirecta, como se verá, la dirección donde empieza la primera instrucción de la SR. En cambio el número xxxx que acompaña a CALL xxxx provee directamente la dirección de dicha instrucción.

Otra diferencia es que mientras se ejecuta una SR del SO o del BIOS la UCP está en "modo kernel o supervisor" estado en el cual la ejecución dicha SR no puede a su vez ser interrumpida por interrupciones de hardware tipo IRQ (a tratar), mientras que la ejecución de una SR llamada por CALL se realiza con la UCP en "modo user", pudiendo ser demorada por esta clase de interrupciones.

Por otra parte en modo kernel la UCP puede ejecutar su repertorio completo de instrucciones, mientras que para seguridad del sistema, en modo usuario ciertas instrucciones no pueden ser ejecutadas para aplicaciones. Cada SO maneja su conjunto propio de llamadas al mismo, que constituyen una interfaz entre el SO y los PGMs de aplicación, y conforman un lenguaje que deben usar estos últimos para comunicarse con el SO.

De lo anterior se desprende que las interrupciones por software son una forma de llamar al SO para que preste servicio a aplicaciones, pues éste "por su cuenta" no pasa a ser ejecutado por la UCP sino mediante este tipo de interrupciones o cuando ocurre una interrupción por hardware (se tratan en detalle más adelante).

DETALLE DE LA EJECUCION DE INSTRUCCIONES INT xx (INTERRUPCION POR SOFTWARE)
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.

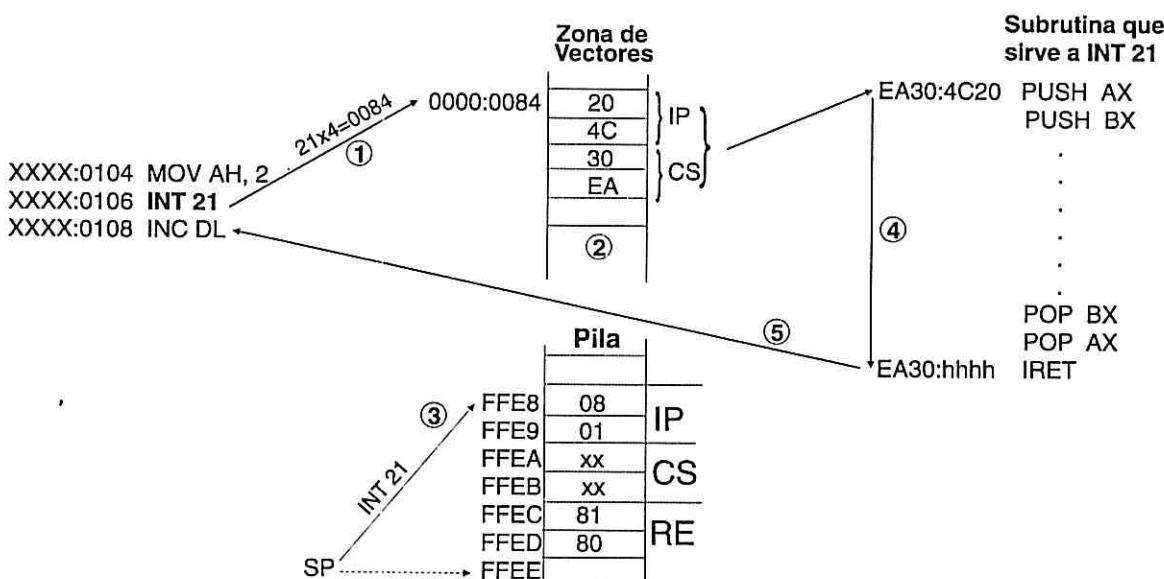


Figura 3.47

Dada la secuencia:

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

para ejecutar una instrucción INT xx, como ser 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 la componente derecha de una dirección cuya componente izquierda es siempre 0000, esto es 0000:0084, dirección que está dentro del primer segmento donde empieza en 00000 = 0000:0000 de la memoria principal (figura 3.38) denominada “zona de vectores interrupción”¹. Esta es la dirección donde a su vez se encuentra la dirección CS:IP de la SR que atiende a INT 21.² Como a partir de la dirección 0000:0084 está la dirección (CS:IP = EA30:4C20) de la SR que sirve a INT 21, o sea (figura 3.47), que el contenido de las 4 células que empiezan en 0000:0084 apunta al código de la SR, por lo cual éste se llama “vector interrupción”.
3. La UC guarda en la pila (figura 3.47) el registro de estado (RE) conteniendo los flags. Sobre éste apila la dirección de retorno (xxxx:0108) de la instrucción (INC DL) que sigue a INT 21. Por lo tanto, el SP pasa de FFEE a FFE8. El RE en el 80286 tiene 16 bits dispuestos como sigue (figura 3.48)

			V	D	I	T	S	Z	A	P	C
--	--	--	---	---	---	---	---	---	---	---	---

Figura 3.48

Suponiendo que los flags tengan los valores supuestos a continuación, el contenido del RE en hexa será:

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

 $= 0881$

4. Ejecuta la SR 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 SR ordenar restaurar desde la pila los contenidos de los registros guardados mediante las instrucciones PUSH.
5. La última instrucción de la SR, que necesariamente debe ser IRET ordena retornar a la instrucción que sigue a INT 21 en el PGM llamador (o sea a INC DL), cuya dirección xxxx:0108 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 3.47). 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 del CS:IP de la dirección de retorno, ordena restaurar los flags en RE.

EJERCICIO 47 LLAMADO A SUBRUTINAS INTEGRADO CON INTERRUPCIONES POR SOFTWARE

El presente ejercicio integra un llamado a SR con una interrupción por software mediante la instrucción INT 61.

Por ser conocido, para el llamado a SR se utiliza el ejercicio 46.a, con la diferencia que en la SR luego del paso CH \leftarrow 01 (instrucción MOV CH, 01) sigue INT 61 que ordena interrumpir en forma transitoria la secuencia que se venía ejecutando y se pase a ejecutar la SR del BIOS o del sistema operativo (SO) que atiende a INT 61.

Antes que se ejecute esta SR del SO en la pila, cualquier instrucción INT xx ordena que se guarden en la pila los registros IP, CS y RE con los valores que tenían en el momento en que se ejecuta INT xx, como se verificará con el Debug.

El registro RE además de los valores de los flags SZVC contiene el valor 0 (EI en el Debug) ó 1 (DI) de un flag designado I vinculado a las interrupciones, que hasta la ejecución de cualquier INT xx vale cero (EI = Enable I) indicando que habilita las interrupciones por hardware durante la ejecución de cualquier instrucción anterior a INT xx.

Cuando I = 0 se dice que la UCP está en “modo user”, admitiendo interrupciones por hardware.

Este flag I durante la ejecución de cualquier INT xx pasa a valer 1 (DI = Disable I), por lo que también cambia el contenido de RE, del cual el valor del flag I forma parte. Como luego de INT xx la UCP pasa a ejecutar una SR del BIOS o del SO, se dice que la UCP pasa de modo “user” a “modo kernel”.

¹ Esta zona, que cada sistema operativo escribe luego del booteo, consta de vectores que ocupan 4 posiciones de memoria, de donde se explica la multiplicación por 4 (en binario por 100). Como se verá al tratar interrupciones por hardware, también existe un vector para cada interrupción por IRQ, de modo que así se pueda localizar la subrutina que la atiende..

² Una aplicación, no sabe dónde está localizada la SR que le va a dar servicio. Si ésta tuviera dirección fija, cualquier cambio en el SO provocaría una reprogramación de la aplicación.

La SR que atiende a INT61 consta de 5 instrucciones representativas de una SR del BIOS, sin sentido real, y termina con IRET, que permite retornar a la instrucción que sigue a INT61 (en este caso POPF (RE ← Pila)).

Para retornar la instrucción IRET (opuesta a cualquier INT xx) devuelve desde la pila hacia los registros IP, CS y RE trae de la pila los valores que tenían antes de que se ejecute INT 61 (en este caso serán los que tenían mientras se ejecutaba la SR que permiten seguir con POPF y el flag I se pone en EI (modo "user") para que nuevamente la UC autorice interrupciones por hardware del tipo IRQ (ver más adelante).

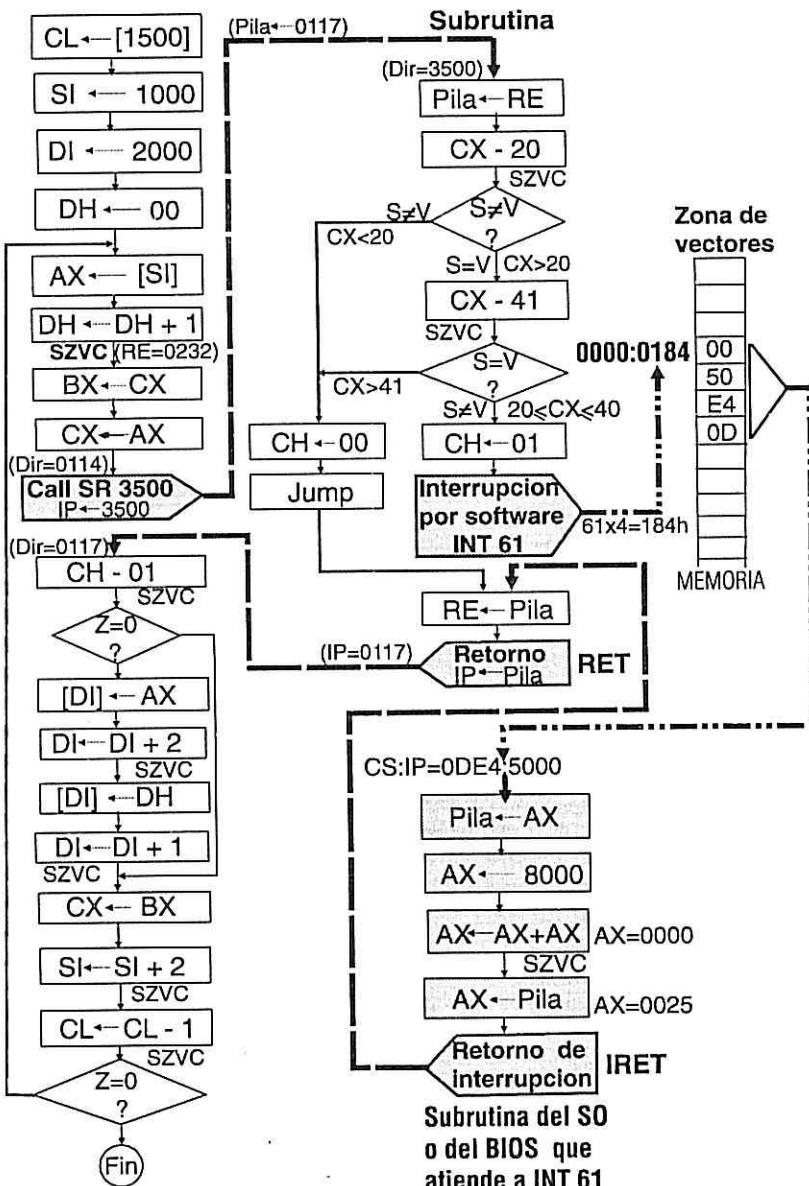


Figura 3.49

Después el diagrama lógico (figura 3.49) sigue el mismo curso que el de la figura 3.41, siendo que en grisado se han dibujado los pasos nuevos, y que los pasos anteriores CALL SR 3500 y RET también están en grisado.

Mediante el Debug se codificará en Assembler el diagrama de la figura 3.49, y se ejecutará para los mismos valores del ejercicio 46.a, como sigue a continuación.

```
-E 1000
0CC6:1000 00.FF 00.FF 00.25 00.00
-E 1500
0CC6:1500 00.02
```

Paso importante que no se puede obviar: escribir en el vector que empieza en la dirección 0000:0184 (siendo 184h = 61 x 4) la dirección 0DE4:5000 (ver figura 3.50) elegida *arbitrariamente* para la primera instrucción de las 5 de la SR que atienden a INT61, pues sino la UC no encontrará esta SR. Este vector sirve de enlace entre INT61 y la SR.

-E 0000:0184
0000:0184 00.00 00.50 00.E4 00.0D (Sin este paso no se podrá ejecutar la secuencia que sigue a continuación)

A continuación se codifica en Assembler el diagrama de la figura 3.49, y luego se ejecutan las instrucciones.

Es importante observar durante la ejecución, que antes de la ejecución de INT 61 el flag I del RE estaba en EI (Enable I), lo cual implica que están habilitadas las interrupciones por hardware que tengan lugar durante la ejecución de las instrucciones; y que luego de la ejecución de INT 61 el flag I pasa a DI (Disable I) inhabilitando dicho tipo de interrupciones.

-A 0100		
0CC6:0100	MOVCL, [1500]	Carga en CL la cantidad n de números de la lista
0CC6:0104	MOV SI, 1000	Inicializa SI = 1000
0CC6:0107	MOV DI, 2000	Inicializa DI = 2000
0CC6:010A	MOV DH, 00	Inicializa DH = 00
0CC6:010C	MOV AX, [SI]	Lleva hacia AX una copia del número N apuntado por SI
0CC6:010E	INC DH	Cada vez que se toma un número nuevo de la lista se suma uno al contador de orden DH
0CC6:0110	MOV BX, CX	Guarda en BX una copia de CX, que luego será modificado para la SR
0CC6:0112	MOV CX, AX	Copia en CX una copia de AX, para que luego sea el dato que procesará la SR
0CC6:0114	CALL 3500	Pasa a ejecutar la SR que está en 3500
0CC6:0117	CMP CH, 01	Luego de que se ejecutó la SR, se vuelve al PGM para restar CH - 01
0CC6:011A	JNZ 0124	Si de CH - 01 fue Z=0 es que no es CH=01, o sea que no fue $20 \leq N \leq 40$, saltar a 0124
0CC6:011C	MOV [DI], AX	Copia en la lista apuntada por DI el número N comprendido entre 20 y 40
0CC6:011E	ADD DI, 2	Suma 2 a DI para preparar debajo la escritura del número de orden de N debajo de su valor
0CC6:0121	MOV [DI], DH	Escribe en la dirección apuntada por DI una copia del número de orden que indica DH
0CC6:0123	INC DI	Se actualiza DI por si en AX llega otro N comprendido entre 20 y 40
0CC6:0124	MOV CX, BX	Desde BX se restaura el valor que tenía CX antes de llamar a la SR
0CC6:0126	ADD SI, 2	Si Z=1 (AX = AXanterior) suma 2 a SI para apuntar siguiente número N de la lista
0CC6:0129	DEC CL	Resta uno a la cantidad de números que falta analizar en la lista apuntada por SI
0CC6:012B	JNZ 010C	Mientras sea Z=0 saltar a 010C para considerar el número siguiente de la lista
0CC6:012D	INT 20	Finalizar
-A 3500		
0CC6:3500	PUSHF	Guarda en la pila el valor de los Flags que usaba el programa, contenidos en el registro RE
0CC6:3501	CMP CX, 20	Resta CX - 20 para que SZVC tomen valores, siendo CX = N
0CC6:3504	JL 3520	Si S≠V resulta CX < 20 entonces saltar a 3520
0CC6:3506	CMP CX, 41	Resta CX - 41 para que SZVC tomen valores
0CC6:3509	JG 3520	Si S=V resulta CX ≥ 41 (o sea CX > 40) entonces saltar a 3520
0CC6:350B	MOV CH, 01	Si S≠V (CX < 41) se hace CH = 01, para indicar que $20 \leq CX \leq 40$
00A7:350D	INT 61	
0CC6:350F	POPF	Pasa de la pila a RE el valor que tenían los flags en el programa antes de llamar a la SR
0CC6:3510	RET	Pasa de la pila al IP la dirección 0117 de la instrucción por la cual prosigue el programa
-A 3520		
0CC6:3520	MOV CH, 00	Si S=V (CX ≥ 41) (o sea CX > 40) se hace CH=00, para indicar que no es $20 \leq CX \leq 40$
0CC6:3522	JMP 350F	Saltar incondicionalmente a 350F

En la siguiente secuencia, las instrucciones MOV y ADD quieren ser representativas de una SR del sistema operativo o del BIOS. Se han elegido por que modifican el registro AX que en el PGM antes de CALL 3500 contenía el número N a analizar, y por que también modifican valores de flags que habían cambiado en la SR, siendo que PUSH AX y POP AX restauran el contenido con que AX quedó antes de llamar a la SR.

-A 0DE4:5000
0DE4:5000 PUSH AX
0DE4:5001 MOV AX, 8000
0DE4:5004 ADD AX, AX
0DE4:5006 ,POP AX
0DE4:5007 IRET

Figura 3.50

R

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0100 NV UP EI PL NZ NA PO NC
0CC6:0100 8A0E0015 MOV CL, [1500] DS:1500=02 (El valor 02 que está en 1500 se copia en CL)

-T
AX=0025 BX=0001 CX=0025 DX=0200 SP=FFEAE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350B NV UP EI NG NZ NA PE CY
0CC6:350B B501 MOV CH,01

-E FFEAE
0CC6:FFEAE 02. 30. 17. 01. 00.

-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEAE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350D NV UP EI NG NZ NA PE CY
0CC6:350D CD61 INT 61 (Con INT el flag I pasa de EI a DI)

-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0DE4 IP=5000 NV UP DI NG NZ NA PE CY
0DE4:5000 50 PUSH AX Estos flags de la SR corresponden a RE=3087 (ver pila siguiente)

-E FFE4
0CC6:FFE4 0F. 35. C6. 0C. 87. 30. (En la Pila se han apilado IP, CS y RE para cuando se retorne a la SR.)
IPret sr CSret sr REsr

0CC6:FFEAE 02. 30. 17. 01. 00.
REpgm IPret pgm La cima pasa de FFE4 a FFE2

-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFE2 BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0DE4 IP=5001 NV UP DI NG NZ NA PE CY
0DE4:5001 B80060 MOV AX,8000 (Los flags cambian de valor en relación con el que tenían en la SR)

-E FFE2 En FFE2 se apila AX=N, con lo cual la Pila queda como se indica en la pila siguiente
0CC6:FFE2 25. 00. 0F. 35. C6. 00. 87. 30.

0CC6:FFEAE 02. 30. 17. 01. 00.

-T
AX=8000 BX=0001 CX=0125 DX=0200 SP=FFE2 BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0DE4 IP=5004 NV UP DI NG NZ NA PE CY
0DE4:5004 01C0 ADD AX,AX

-T
AX=0000 BX=0001 CX=0125 DX=0200 SP=FFE2 BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0DE4 IP=5006 OV UP DI PL ZR NA PE CY
0DE4:5006 58 POP AX

-T (Con POP AX se desapila 0025 que pasa de la pila hacia AX)
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFE4 BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0DE4 IP=5007 OV UP DI PL ZR NA PE CY
0DE4:5007 CF IRET La cima vuelve a FFE4

-E FFE4
0CC6:FFE4 0F. 35. C6. 0C. 87. 30. (Con IRET el flag I pasa de DI a EI)
0CC6:FFEAE 02. 30. 17. 01. 00. La cima vuelve a FFEAE

-T (Con IRET se desapilan 350F, 0CC6 y 3087 que pasan de la pila hacia los registros indicados. Así IP vuelve a 350F, CS a 0CC6 y los flags al valor que tenían luego de CMP CX, 41)
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEAE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=350F NV UP EI NG NZ NA PE CY
0CC6:350F 9D POPF (Con CS:IP = 0CC6;350F se retorna a POPF de la SR)

-E FFEAE
0CC6:FFEAE 02. 30. 17. 01. 00. La cima vuelve a FFEC

-T (Con POPF se desapila 3002 que pasa a RE, por lo que los flags vuelven al valor que tenían luego de INC DH)
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEC BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=3510 NV UP EI PL NZ NA PO NC

0CC6:3510 C3 RET A partir de acá la ejecución del PGM sigue el mismo curso que en el ejercicio 3.46.a

-E FFEC (Con RET se desapila 0117 que pasa de la pila hacia IP, y así se retorna a CMP CH, 01 del PGM))
0CC6:FFEC 17. 01. 00.

-T La cima vuelve a su valor inicial FFEE
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0117 NV UP EI PL NZ NA PO NC

0CC6:0117 80FD01 CMP CH,01

-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011A NV UP EI PL ZR NA PE NC
0CC6:011A 7508 JNZ 0124

-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011C NV UP EI PL ZR NA PE NC
0CC6:011C 8905 MOV [DI],AX DS:2000=0025

-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2000
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=011E NV UP EI PL ZR NA PE NC
0CC6:011E 83C702 ADD DI,+02

-T

```

AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2002
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0121 NV UP EI PL NZ NA PO NC
0CC6:0121 8835    MOV    [DI],DH   DS:0202=02
-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2002
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0123 NV UP DI PL NZ NA PO NC
0CC6:0123 47      INC    DI
-T
AX=0025 BX=0001 CX=0125 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0124 NV UP EI PL NZ NA PE NC
0CC6:0124 89D9    MOV    CX,BX
-T
AX=0025 BX=0001 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1002 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0126 NV UP EI PL NZ NA PE NC
0CC6:0126 83C602  ADD    SI,+02
-T
AX=0025 BX=0001 CX=0001 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=0129 NV UP EI PL NZ NA PO NC
0CC6:0129 FEC9    DEC    CL
-T
AX=0025 BX=0001 CX=0000 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012B NV UP EI PL ZR NA PE NC
0CC6:012B 75DF    JNZ    010C
-T
AX=0025 BX=0001 CX=0000 DX=0200 SP=FFEE BP=0000 SI=1004 DI=2003
DS=0CC6 ES=0CC6 SS=0CC6 CS=0CC6 IP=012D NV UP EI PL ZR NA PE NC
0CC6:012D CD20    INT    20

```

EJERCICIOS BASICOS CON INSTRUCCIONES INT 21

EJERCICIO 48: Secuencia para visualizar en pantalla los $255 = FF_H$ caracteres que el código ASCII codifica, generando el código ASCII de cada carácter, a partir del correspondiente al código ASCII 00

xxxx:0100	MOV CL, FF	Carga en CL el total de caracteres a visualizar
xxxx:0102	MOV DL, 00	Carga en DL el código ASCII del primer carácter a visualizar
xxxx:0104	MOV AH, 2	Valor en AH para que INT 21 muestre en pantalla el carácter cuyo código ascii está en DL
xxxx:0106	INT 21	Llama a subrutina del BIOS para escribir en pantalla dicho carácter
xxxx:0108	INC DL	Se genera el código ASCII del siguiente carácter a visualizar
xxxx:010A	DEC CL	Descuenta uno al número de caracteres a visualizar
xxxx:010C	JNZ 0104	Mientras Z=0, volver a 0104
xxxx:010E	INT 20	Fin

Se comprende que se puede usar parte de esta secuencia para visualizar por pantalla cualquier mensaje, con tal que cada vez que se ejecuten MOV AH, 2 e INT 21 esté antes en el registro DL el código ASCII del próximo carácter a visualizar, en el orden que corresponda a las letras y espacios del mensaje a visualizar.

Las instrucciones de interrupción por software, tipo INT 21, no se pueden ejecutar en el Debug mediante el comando T, sino que debe usarse el comando P.
 La secuencia anterior puede ejecutarse de una vez con el comando G = 0100, habiendo puesto previamente IP=0100 mediante RIP

El lector interesado puede hacer la siguiente indagación para INT 21, usando el Debug.
 Siendo $21 \times 4 = 48$, mediante el comando E 0000:0048 ubicará los 4 bytes que constituyen el CS:IP de la primera instrucción de la subrutina que atiende a INT 21.
 Luego, mediante el comando U CS:IP podrá ver y seguir en Assembler dicha subrutina, pulsando nuevamente U cada vez que quiera proseguir viendo en pantalla las instrucciones siguientes de la subrutina.
 Otra prueba que puede realizarse mediante E 0000:0048 es intentar re escribir el vector interrupción de INT 21 (cosa que un virus puede hacer)

EJERCICIO 49: Secuencia para esperar el ingreso de 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 0100		
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	INT 21 con AH=1 permite entrar un carácter por teclado. El programa queda esperando hasta que se tipee el carácter, que luego aparece por pantalla
xxxx:0109	INT 21	Llama a subrutina del DOS que realiza lo indicado con AH=1
xxxx:010B	MOV [DI], AL	Guarda en la lista apuntada por DI el código ASCII carácter tipeado, que INT 21 deja en AL
xxxx:010D	INC DI	Incrementa DI
xxxx:010E	DEC CL	Decrementa la cantidad de repeticiones a realizar
xxxx:0110	JNZ 0109	Mientras Z=0, volver a 0109
xxxx:0112	INT 20	Fin

EJERCICIO 50: La siguiente secuencia, sobre la base de otra desarrollada por el Sr. Germán Apesteguía, espera el ingreso por teclado de 8 dígitos, como ser los de un DNI. Además, realiza una validación que rechaza el ingreso de caracteres no numéricos, produciéndose en tal caso un “beep” por el parlante de la computadora. Una vez finalizada la carga de los 8 dígitos, se visualizan por pantalla nuevamente, en una línea, todos los dígitos ingresados, mediante la función 9 de INT 21.

-A 0100		
xxxx:0100	MOV SI, 2000	Inicializa SI en 2000, comienzo de la lista a visualizar luego en pantalla
xxxx:0103	MOV CL, 8	Inicializa CL en 8, número de caracteres de un DNI
xxxx:0105	MOV AH, 1	Inicializa parámetros para esperar que se tipee un carácter, que se ve en pantalla
xxxx:0107	INT 21	Llama subrutina del BIOS. El carácter tipeado queda en AL
xxxx:0109	CMP AL, 30	Compara el carácter entrado por teclado con 30 (código ASCII del 0)
xxxx:010B	JB 0130	Si el código ASCII entrado es menor que el ASCII del 0, salta a 0130
xxxx:010D	CMP AL, 39	Compara el carácter entrado por teclado con 39 (código ASCII del 9)
xxxx:010F	JA 0130	Si el código ASCII entrado es mayor que el ASCII del 9, salta a 0130
xxxx:0111	MOV [SI], AL	Guarda en memoria el carácter dejado en AL al ejecutarse INT 21
xxxx:0113	INC SI	Se prepara para guardar otro carácter tipeado en memoria
xxxx:0114	DEC CL	Resta 1 a CL por haberse tipeado uno de los 8 caracteres
xxxx:0116	JNZ 0105	Mientras CL no sea cero, volver a 0105
xxxx:0118	MOV AL, 24	Carga 24 (código de \$) al registro AL
xxxx:011A	MOV [SI], AL	Escribe 24 al final de la lista de caracteres tipeados (Para la próx. INT 21)
xxxx:011C	MOV SI, 2000	Vuelve SI a su valor 2000
xxxx:011F	MOV DX, SI	Carga en DX el valor de SI, como lo exige la próxima INT 21
xxxx:0121	MOV AH, 9	Inicializa parámetros para que escriba en pantalla una lista de números que empieza en dirección indicada por DX y termina con el número 24
xxxx:0123	INT 21	Llama a subrutina del BIOS para que imprima la lista en memoria
xxxx:0125	INT 20	Fin
-A 0130		
xxxx:0130	MOV DL, 07	Deja en DL el ASCII 07 (bell) para que pueda escucharse un “beep”
xxxx:0132	MOV AH, 2	Inicializa parámetros para ver (en este caso escuchar) el ASCII 07
xxxx:0134	INT 21	Llama a subrutina del BIOS para visualizar caracteres por pantalla
xxxx:0136	JMP 0105	Vuelve a 0105

EJERCICIO 51: Secuencia para esperar el tipeo de cualquier número de caracteres por teclado, los cuales se visualizan en pantalla. Cuando se terminan de tipear se debe pulsar la tecla ESP. Tiene la limitación que no funciona correctamente la tecla Enter, pues no se puede saltar de renglón.

xxxx:0100	MOV AH, 6	Valor en AH para que INT 21 cargue cada carácter tipeado en AL, y que cada vez que ello ocurre sea Z=0, y Z=1 mientras no se tipee ninguno
xxxx:0102	MOV DL, FF	Se requiere cargar en DL el código FF (ASCII extendido)
xxxx:0104	INT 21	Llama a subrutina del BIOS para cargar en AL el carácter tipeado
xxxx:0106	JZ 0100	Mientras sea Z=1 (no hay tipeo) volver a 0100 (loop de espera)
xxxx:0108	CMP AL, 2B	Compara el código ASCII de la tecla tipeada con 2B (ASCII de ESC)
xxxx:010A	JZ 0130	Si es igual a 2B saltar a 0130 (Fin)
xxxx:010C	MOV DL, AL	Si no es ESC, pasar a DL el ASCII tipeado para preparar su visualización
xxxx:010E	INT 21	Llama a subrutina del BIOS para visualizar el carácter tipeado
xxxx:0110	JMP 0100	Vuelve al loop de espera de tecla tipeada
xxxx:0130	INT 20	Fin

EJERCICIO 52: 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
Ejemplo:

Suponiendo que el Debug en las posiciones 2000/1 nos muestra 41. 5A. en pantalla se debe ver 5A41
Si en DL cargamos 41 y luego ejecutamos MOV AH, 9 e INT 21 en pantalla aparecerá una A en vez de 41. Para que se visualice 5A41 hay que realizar lo que sigue.

Cargar 35 (código ASCII de 3) en DL, y luego ejecutar MOV AH, 9 e INT 21

Cargar 41 (código ASCII de A) en DL, y luego ejecutar MOV AH, 9 e INT 21

Cargar 34 (código ASCII de 4) en DL, y luego ejecutar MOV AH, 9 e INT 21

Cargar 31 (código ASCII de 1) en DL, y luego ejecutar MOV AH, 9 e INT 21

Por lo tanto, primero habrá que generar los códigos ASCII de cada uno de los 4 caracteres a visualizar, a partir de su valor (cuarteto binario que ocupa la mitad de una posición de memoria). Esto hace la secuencia abajo desarrollada.

Luego de MOV BX, [2000] resulta:

ROL BX, CL (con CL puesto antes en 4) efectúa 4 desplazamientos:
(La instrucción ROL ordena rotate left)

$$\begin{aligned} BX &= \underline{\underline{0}}\underline{1}\underline{0}\underline{1}\underline{1}\underline{0}\underline{1}\underline{0}\underline{0}\underline{1}\underline{0}\underline{0}\underline{0}\underline{0}\underline{1} = 5A41 \\ &\quad \underline{1}\underline{0}\underline{1}\underline{1}\underline{0}\underline{0}\underline{1}\underline{0}\underline{0}\underline{0}\underline{0}\underline{1} \\ &\quad \underline{0}\underline{1}\underline{1}\underline{0}\underline{1}\underline{0}\underline{1}\underline{0}\underline{0}\underline{0}\underline{1} \\ &\quad \underline{1}\underline{1}\underline{0}\underline{1}\underline{0}\underline{0}\underline{1}\underline{0}\underline{0}\underline{0}\underline{1} \\ &\quad \underline{1}\underline{0}\underline{1}\underline{0}\underline{0}\underline{1}\underline{0}\underline{0}\underline{0}\underline{0}\underline{1} = A415 \end{aligned}$$

En cada desplazamiento, los bits de BX se desplazan una posición a la izquierda, y el bit extremo izquierdo de BX (subrayado) pasa a ser el bit extremo derecho de BX. De esta forma, el 0101 (5) que estaba en la extrema izquierda de BX, luego de 4 desplazamientos pasa a la extrema derecha de BX.

Con MOV AL, BL resulta:

AND AL, 0F ordena hacer bit a bit una *And* ("y") entre el contenido de AL y 0F
(0 y 0 = 0; 0 y 1 = 0; 1 y 0 = 0; 1 y 1 = 1) para hacer 0 el cuarteto superior de AL:
ADD AL, 30 ordena sumarle 30 a AL

$$\begin{aligned} AL &= 00010101 = 15 \\ And \quad \underline{\underline{0}}\underline{0}\underline{0}\underline{0}\underline{1}\underline{1}\underline{1} &= 0F \\ &\quad \underline{0}\underline{0}\underline{0}\underline{0}\underline{1}\underline{0}\underline{1} = 05 \\ &\quad \underline{0}\underline{1}\underline{1}\underline{1}\underline{0}\underline{0}\underline{0} = 30 \\ &\quad \underline{0}\underline{0}\underline{1}\underline{1}\underline{0}\underline{1}\underline{0} = 35 \end{aligned}$$

Así, para los cuartetos del 0 al 9 se forma el código ASCII, en este caso 35

Para cuartetos de A hasta F (código ASCII > 39) se debe sumar luego 7 = 00000111. En este ejemplo, para la A, luego de ADD AL, 30 el resultado sería 00111010. Este sumado a 00000111 da 01000001 = 41 (código ASCII de A)

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 para no destruir luego contenidos de BX
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 (resultando de 30 a 3F)
xxxx:0110	CMP AL, 39	Compara con 39 (código ASCII del 9)
xxxx:0112	JBE 0116	Pregunta si es < 39 en cuyo caso en AL está el código ASCII (del 0 al 9)
xxxx:0114	ADD AL, 07	Si es mayor suma 7 (por ej. 3A + 7 = 41 = código ASCII de A)
xxxx:0116	MOV DL, AL	Para el BIOS (próxima INT 21) carga el número de AL en DL
xxxx:0118	MOV AH, 2	Inicializa parámetros para ver en pantalla el carácter ASCII que está en DL
xxxx:011A	INT 21	Llama a subrutina del BIOS
xxxx:011C	DEC CH	Decrementa cantidad de loops a realizar
xxxx:011E	JNZ 0106	Mientras Z=0, volver a 0106
xxxx:0120	INT 20	Fin

EJERCICIO 53: Secuencia para visualizar en pantalla como números decimales a números binarios naturales de 2 bytes que están en memoria. Para ello se requiere convertir estos números binarios a dígitos BCD y éstos a ASCII. Los números binarios forman una lista que está a partir de la dirección 1000, y su longitud está en 2500. Ellos deberán verse en pantalla en decimal uno debajo del otro.

Para convertir de binario a BCD se usa el algoritmo de los restos de las sucesivas divisiones por la base a la que se quiere pasar, en este caso diez. En el papel, cuando se pasa de base diez a base dos, las cuentas se hacen en base diez. En la máquina las divisiones se hacen en base dos, y el divisor es 1010 (base diez en binario) y terminan cuando el cociente es cero. Esto implica una división más, para detectar cuando el cociente es menor que el divisor, siendo que su resto es el primer dígito BCD. En base diez éste era el cociente de la última división.

La subrutina que pasa a BCD realiza las divisiones de la forma vista en el ejercicio 36, y para tomar los restos en orden inverso a su generación usa la pila.

A 0100		
xxxx:0100	MOV CX, [2500]	Carga en CX la longitud de la lista
xxxx:0104	MOV SI, 1000	El registro SI apunta al comienzo de la lista de datos
xxxx:0106	MOV AX, [SI]	Carga en AX un dato de la lista que apunta SI, a ser usado por subrutina
xxxx:0108	ADD SI, 2	Incrementa en 2 el puntero SI
xxxx:010B	CALL 5000	Llama a subrutina para conversión y visualización que está en 5000
xxxx:011E	DEC CX	Decrementa CX
xxxx:011F	JNZ 106	Mientras Z no sea 1, volver a 106
xxxx:0111	INT 20	Fin
A 5000		
xxxx:5000	PUSH BX	Guarda BX en la pila
xxxx:5001	PUSH CX	Guarda CX en la pila
xxxx:5002	PUSH DX	Guarda DX en la pila
xxxx:5003	MOV CX, 0	Lleva CX a cero, para ser usado como contador
xxxx:5006	MOV BX, A	Carga en BX el número divisor 1010 = A
xxxx:5009	MOV DX, 0	Lleva a cero el registro DX para no afectar cada división de nros de 16 bits
xxxx:500C	DIV BX	Divide AX por BX; el <u>resultado queda en AX y el resto en DX</u>
xxxx:500E	PUSH DX	Guarda en la pila el resto de cada división (dígito BCD)
xxxx:500F	INC CX	Incrementa el contador de divisiones efectuadas (dígitos del nro hallados)
xxxx:5010	CMP AX, 0	Compara AX con cero para determinar si es la última división
xxxx:5013	JNZ 5009	Si no es cero volver a 5009 .
xxxx:5015	POP DX	Toma un resto desde la pila (dígito del número a visualizar)
xxxx:5016	MOV AH, 6	Hace AH=6 para que INT 21 muestre en pantalla el dígito que está en DL
xxxx:5018	ADD DL, 30	Suma 30 a DL para que el dígito BCD esté en ASCII, como requiere INT 21
xxxx:500B	INT 21	Interrupción por software para sacar en pantalla el dígito que está en DL
xxxx:500D	DEC CX	Decrementa CX pues ya se visualizó un dígito decimal
xxxx:500E	JNZ 5015	Si no es cero ir a 5015, pues hay más dígitos decimales del nro a visualizar
xxxx:5020	MOV AH, 6	Hace AH=6 para que INT 21 entre en pantalla el ASCII que está en DL
xxxx:5022	MOV DL, 0D	Hace DL=0D (código ASCII de Enter) para comenzar otro renglón
xxxx:5024	INT 21	Interrupción por software para que en pantalla tenga lugar un Enter
xxxx:5026	MOV DL, 0A	Hace DL=0A (código ASCII de LF que sigue a Enter para saltar de renglón
xxxx:5028	INT 21	Interrupción por software para que en pantalla tenga lugar LF (line feed)
xxxx:502A	POP DX	Restaura DX desde la pila
xxxx:502B	POP CX	Restaura CX desde la pila
xxxx:502C	POP BX	Restaura BX desde la pila
xxxx:502D	RET	Retorna al programa para tomar otro número a ser visualizado en otro renglón

INTERRUPCIONES POR HARDWARE

Mientras que las interrupciones por software INTxx son instrucciones que forman parte de un PGM y el programador las ubica en puntos del programa que él decide, para solicitar servicios del SO (como ser el manejo de archivos, impresiones, etc.)¹, las interrupciones por hardware no se programan, pues por lo general no tienen mucho que ver con un programa en ejecución.

Pueden ocurrir o no mientras se ejecuta una instrucción del mismo, y suceden de manera inesperada.

Durante la ejecución de cada instrucción de un PGM puede suceder un evento en el que el hardware necesita que se interrumpe el PGM que se está ejecutando, y que la UCP pase a ejecutar una SR del BIOS o del SO que atiende ese tipo de evento, luego de lo cual se reanuda la ejecución del PGM interrumpido. Por ejemplo:

- en la placa madre hay una circuitería que a fracciones fijas de segundo solicita interrumpir el PGM que se está ejecutando para que se ejecute una SR del BIOS que actualiza la hora del reloj que está en pantalla.
- cada vez que se pulsa una tecla del teclado, se solicita interrumpir y se pase a ejecutar una SR del BIOS.
- cuando otro PGMx distinto al PGM que se está ejecutando solicitó antes mediante una INTxx la escritura de un sector de un disco, quedando así autointerrumpido a la espera de que la UCP reanude su ejecución; una vez que esa escritura se lleve a cabo una señal del hardware indicará que ello ocurrió, solicitando que se

¹ En el DOS las funciones se pedían con INTxx. Desde Windows 95 la mayor parte de las funciones del sistema están en la librería "Kernel32.DLL" que importan casi todos los programas. DLL son siglas de Dynamic Link Library (Biblioteca de Vínculos Dinámicos). Es un archivo ejecutable que permite compartir código y otros recursos para realizar ciertas tareas y funciones básicas que un PGM puede tomar. La DLL se carga en memoria y se usa una sola copia para todos los programas que soliciten servicios de ella.

interrumpa el PGM en ejecución para que una SR del BIOS verifique si dicha escritura fue exitosa. Se comprende que tales eventos pueden suceder en cualquier momento impredecible, y que no se puede realizar un PGM que además de su cometido, contemple si ocurrió una señal que avisa que hay que actualizar la hora, o que un PGM esté pensado para el caso que *otro* PGM esté esperando por el fin de una escritura, etc. Los eventos anteriores se originan en periféricos y dispositivos externos a la UCP que necesitan atención, por lo que el tipo de interrupciones que generan son las **interrupciones por hardware externas enmascarables**. También en la UCP hay situaciones de error al intentar ejecutar una instrucción de un PGM, como ser:

- cuando la instrucción tiene un código inexistente, u ordena hacer una división por un número muy pequeño y el resultado no entra en el formato establecido (división por cero).
 - si la instrucción no está permitida en modo usuario
 - si la dirección de algún operando corresponde a una zona prohibida o se intenta violar algún permiso de uso
- Estos eventos constituyen las “**interrupciones por hardware internas**” o “**excepciones**”, y dan lugar a que la SR del SO llame a la instrucción y en general el PGM que se está ejecutando.

DETALLE DE LAS INTERRUPCIONES POR HARDWARE ENMASCARABLES

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 una interrupción enmascarable:

1. Segundo se estableció (figura 3.50), las líneas IRQ_n de solicitud de interrupción (“interrupt request”), salen de las interfaces desde líneas de control del bus al cual esta unida cada interfaz y van a un chip donde 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 a una patilla del microprocesador, para ser sensada por la UC.
3. Como se especifica en el diagrama lógico de la figura 3.50, 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 flag I, contenido en el RE junto con los otros flags. En caso de que sea I = 0 (DI en el debug, por estar la UCP en modo kernel ejecutando una SR del SO) no da curso a la solicitud de interrupción indicada por INTR (y comenzada por la activación de la línea IRQ₂), pasando a ejecutar la instrucción I_{n+1}.
4. Si I = 1 (EI en el debug) por estar la UCP en modo user ejecutando software de usuario, se aceptará la solicitud de interrupción que llegó a la UC a través de la activación de INTR, y la UC activará la línea INTA (A de acknowledge = aceptación) que llega al “árbitro”.
5. El árbitro envía hacia la UC por el bus de datos, una combinación binaria, que es la componente derecha xxxx de la dirección (0000:xxxx) del vector que contiene la dirección donde se encuentra la subrutina preparada exclusivamente para atender la activación de esa línea IRQ₂.
6. La UC ordena apilar primero el valor de RE (con I = 1 como antes de la interrupción), luego el CS y después el IP (éste quedará en la cima de la pila) que conforman la dirección de la instrucción con la que se debe reanudar el programa (la pila queda como en las interrupciones por software, figura 3.47).
7. Con la dirección obtenida en el paso 5, la UC dirige a la primera instrucción de la subrutina que atiende la interrupción, con lo cual se ejecutará ésta y las siguientes.
8. Cuando ejecuta la última instrucción (IRET), la UC lee de la pila el valor de los registros IP y CS del programa interrumpido, con lo cual se reanuda la ejecución a éste, y luego restaura el valor de RE.

Este tipo de interrupción por hardware externa se denomina “**enmascarable**”, pues según el valor del flag I, a una interrupción se le puede dar curso o bien “enmascarar”. Esto último significa que la interrupción se tiene que quedar esperando hasta que sea I = 1, no pudiendo ser interrumpida la SR del BIOS o del SO.

De ser necesario el valor de I puede establecerse por software. Así, si un programa o secuencia no debe ser interrumpido, empezará con la instrucción CLI (Clear I), que hace I = 0, y al terminar debe hacerse I = 1 mediante la instrucción STI (S = 1).

Interrupciones por hardware externas no enmascarables: son sensadas por el microprocesador por medio de su patilla NMI (Not Maskable Interrupt), no pudiendo ser enmascaradas por el flag I. Un caso de activación de la línea que va a NMI ocurre cuando caen los 220v de la fuente de alimentación. Entonces la interrupción no enmascarable pondrá en juego una subrutina preparada para tal evento.

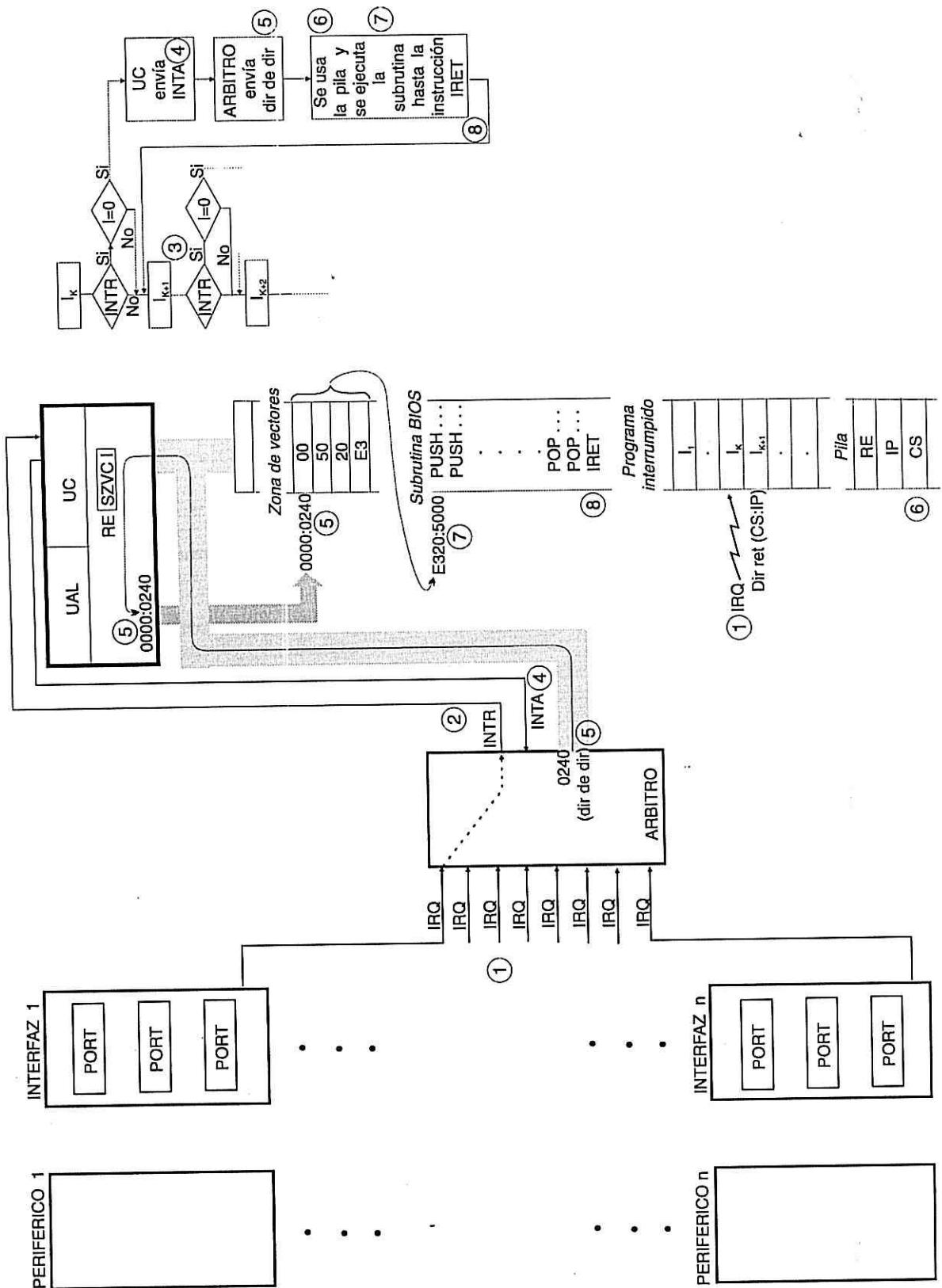


Figura 3.50

EJERCICIO INTEGRADOR DE SUBRUTINAS E INTERRUPCIONES

Dado el siguiente estado de la UCP (los registros que no aparecen no son usados por el PGM que se está ejecutando):
 AX=3325 SI=224A IP=2004 RE=2900 SP=FFEA CS=SS=DS=3A88

1. Si la siguiente instrucción a ejecutar es CALL 3500 que ocupa 3 bytes, indicar:
- Mediante las componentes XXXX : YYYY, en qué dirección de memoria está CALL 3500, y cuál es la dirección que aparecerá en el bus de direcciones.
 - Qué movimientos ocurren durante la ejecución de CALL 3500 y con qué valores quedan los registros involucrados.
 - Cómo queda la pila luego de la ejecución de CALL 3500.

- 1.a Las componentes son CS:IP = 3A88:2004 , y en el bus aparecerá $3A880 + 2004 = 3C884$
 1.b $IP = IP + 3 = 2004 + 3 = 2007$ (IP apunta brevemente a la instrucción In+1 que sigue a CALL (que ocupa 3 bytes)
 $SP \leftarrow SP - 2$ ($SP = FFEA - 2 = FFE8$). O sea que la nueva cima de la pila está en FFE8.
 $[SP] \leftarrow IP$ $[FFE8/9] = 2007$ En las direcciones FFE8/9 de la pila se guardó el valor de IP.
 $IP \leftarrow 3500$ IP = 3500 apunta a la dirección 3500 donde comienza la subrutina
 1.c Del paso anterior resulta:
 SS:SP = 3A88:FFE8 07 (IP)
 FFE9 20 (IP)

2. Si la subrutina empieza con PUSHF (Push flags) y sigue con PUSH AX (ver esquema final), indicar:
- Mediante las componentes XXXX : YYYY, dónde comienza PUSHF
 - Qué movimientos ocurren durante la ejecución de PUSH F y con qué valores quedan los registros involucrados.
 - Cómo queda la pila luego de la ejecución de PUSHF.

- 2.a CS:IP = 3A88:3500
 2.b $SP \leftarrow SP - 2$ ($SP = FFE8 - 2 = FFE6$) La nueva cima de la pila está en FFE6.
 $[SP] \leftarrow RE$ $[FFE6/7] = 2900$
 En las direcciones FFE6/7 de la pila se guardó el valor de RE que contiene los valores de los flags.
 2.c SS:SP = 3A88:FFE6 00 (RE)
 FFE7 29 (RE)
 FFE8 07 (IP)
 FFE9 20 (IP)

- 3.a Qué movimientos ocurren durante la ejecución de PUSH AX y con qué valores quedan los registros involucrados.
 3.b Cómo queda la pila luego de la ejecución de PUSHF.

- 3.a $SP \leftarrow SP - 2$ ($SP = FFE6 - 2 = FFE4$) La nueva cima de la pila está en FFE4
 $[SP] \leftarrow AX$ $[FFE4/5] = 3325$ En las direcciones FFE4/5 de la pila se guardó el valor de AX
 3.b SS:SP = 3A88:FFE4 25 (AX)
 FFE5 33 (AX)
 FFE6 00 (RE)
 FFE7 29 (RE)
 FFE8 07 (IP)
 FFE9 20 (IP)

4. Si a PUSH AX sigue la instrucción INT 21, cuyo vector interrupción contiene las componentes CS = 8942 e IP=3320. Indicar:
- Mediante las componentes XXXX:YYYY, dónde comienza INT21, y qué movimientos durante su ejecución permiten escribir la pila.
 - Cómo queda la pila luego de los movimientos de la ejecución de INT 21 que permitieron escribirla
 - Cuáles son las acciones que faltan realizar para concluir con la ejecución de INT 21 en relación con el flag I y la localización de la subrutina del SO que atiende a INT21.
 - Con qué instrucción termina esta subrutina, y qué movimientos ocurren durante su ejecución
 - Cómo queda la pila cuando finaliza la ejecución de esta subrutina del SO
 - Si este tipo de interrupciones se puede enmascarar

4.a Todas las instrucciones PUSH e INTxx ocupan 2 bytes en memoria (ver esquema final). Puesto que PUSHF comenzaba en CS:IP = 3A88:3500, PUSH AX comenzará en 3A88:3502 e INT 21 comenzará en 3A88:3504. Los movimientos que ocurren durante la ejecución de INT21 para apilar son los siguientes:

- $SP \leftarrow SP - 2$ ($SP = FFE4 - 2 = FFE2$) La nueva cima de la pila está en FFE2.
 $[SP] \leftarrow RE$ $[FFE2/3] = 0029$ (Como hasta ahora la UAL no hizo ninguna operación, el valor de RE se mantiene)
 $SP \leftarrow SP - 2$ ($SP = FFE2 - 2 = FFE0$) La nueva cima de la pila está en FFE0.
 $[SP] \leftarrow CS$ $[FFE0/1] = 3A88$

IP \leftarrow IP + 2 (IP = 3504 + 2 = 3506)¹
 SP \leftarrow SP - 2 (SP = FFE0 - 2 = FFDE) La nueva cima de la pila está en FFDE
 [SP] \leftarrow IP [FFDE/F] = 3506

4.b Hasta este momento de la ejecución de INT21, conforme a los pasos de 4.a la pila quedará:

```

SS:SP = 3A88: FFDE 06 (IP)
      FFDF 35 (IP)
      FFE0 88 (CS)
      FFE1 3A (CS)
      FFE2 00 (RE)
      FFE3 29 (RE)
      FFE4 25 (AX)
      FFE5 33 (AX)
      FFE6 00 (RE)
      FFE7 29 (RE)
      FFE8 07 (IP)
      FFE9 20 (IP)
  
```

4.c Mientras que en 4.a se guardó en la pila el contenido del registro RE que existía antes de la ejecución de INT21, durante la ejecución RE va a cambiar, pues ahora el flag I que forma parte de RE debe pasar de I = 1 a I = 0, deshabilitando (enmascarando) cualquier interrupción por hardware que ocurra mientras se ejecuta la subrutina que atiende a INT 21. Asimismo, puesto que se ejecuta una subrutina del SO, por ser el flag I = 0 la CPU está en modo "kernell", mientras que si I = 1 está en modo "user".

A los efectos de localizar la subrutina que atiende a INT 21, la componente izquierda de la dirección de su vector es 0000 (al igual que para cualquier interrupción); y la componente derecha vale $21 \times 4 = 84h$. O sea que la dirección del vector será 0000:0084. Y de acuerdo con los datos del ejercicio, en la zona de vectores se tendrá:

```

0000:0084 20 (IP)
0000:0085 33 (IP)
0000:0086 42 (CS)
0000:0087 89 (CS)
  
```

Entonces CS:IP = 8942: B320 es la dirección donde empieza la subrutina del SO que atiende a INT21.

4.d La subrutina del SO debe terminar con IRET. Su ejecución comprende los siguientes movimientos en relación con la pila de la respuesta 4.b

```

IP  $\leftarrow$  [SP] IP = 3506
SP  $\leftarrow$  SP + 2 (SP = FFDE + 2 = FFE0) La nueva cima de la pila está en FFE0
CS  $\leftarrow$  [SP] CS = 3A88
SP  $\leftarrow$  SP + 2 (SP = FFE0 + 2 = FFE2) La nueva cima de la pila está en FFE2
RE  $\leftarrow$  [SP] CS = 2900
SP  $\leftarrow$  SP + 2 (SP=FFE2 + 2 = FFE4; así SP apunta al contenido de AX guardado en la pila)
  
```

Como ahora los valores de CS:IP son 3A88:3506, la próxima instrucción a ejecutar será la I_k+1 que sigue a INT 21. Luego de la ejecución de IRET debe ser I = 1, para que la UCP vuelva a modo "user" y las interrupciones por hardware sigan su curso. Esto de hecho ocurre al ser otra vez RE=2900 contenido para el cual es I = 1.

4.e Conforme a los movimientos del paso 4.e la pila queda como en 3.b

4.f INT21 como cualquier otra interrupción por software no puede ser enmascarada, o sea demorada, pues se trata de una instrucción que ejecuta la UC, y no hay forma de demorarla.

5. Puesto que la subrutina empezó con PUSHF, PUSH AX e INT21, indicar:

5.a con qué instrucciones debe terminar

5.b qué movimientos ocurren cuando se ejecuta cada una, y cómo queda la pila luego de cada ejecución

5.a Las instrucciones deben ser

```

POP AX
POP F
RET
  
```

5.b Ejecución de POP AX

AX \leftarrow [SP] (AX = 3325)

SP \leftarrow SP + 2 (SP = FFE4 + 2 = FFE6) La nueva cima de la pila está en FFE6, y la pila queda como en el paso 2.c

Ejecución de POP F

RE \leftarrow [SP] (RE = 2900)

SP \leftarrow SP + 2 (SP = FFE6 + 2 = FFE8) La nueva cima de la pila está en FFE8 y la pila queda como en el paso 1.c

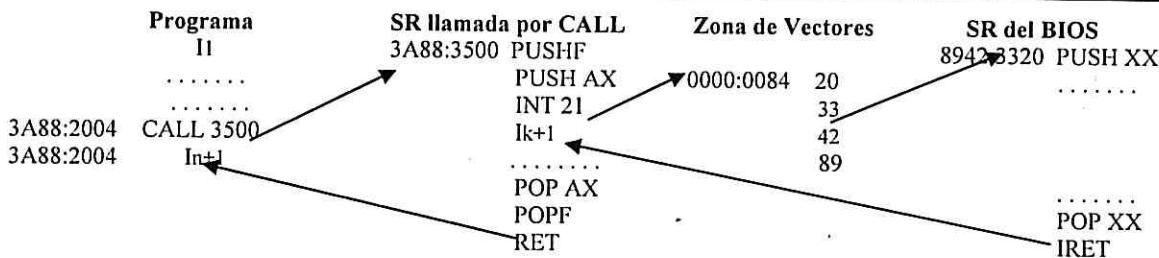
Ejecución de RET

¹ INT21 ocupa 2 bytes, y cuando se pidió INT21, el IP estaba en 3504, ahora IP apunta transitoriamente a la instrucción I_k+1 que sigue a INT21, a la que se debe retornar cuando finalice la subrutina del SO llamada por INT21

IP ← [SP] (IP = 2407)
 SP ← SP + 2 (SP = FFE8 + 2 = FFEA)

La próxima instrucción a ejecutar es In+1 que sigue a CALL 3500
 La nueva cima está en FFEA desde donde comenzó este ejercicio

6. Realizar un esquema del proceso llevado a cabo



- 7. Si en lugar de INT21 durante la ejecución de POP AX se activa una línea IRQxx siendo la dirección de su vector interrupción 0000:0032 , y el contenido de éste 255B 3429, qué cambios ocurrirían en relación con el punto 4.**

En este caso, como no existe la instrucción INT21, en su dirección (3504) existirá otra instrucción cualquiera Ix. Luego de la instrucción POP AX la subrutina quedará interrumpida, y se pasará a ejecutar otra subrutina del SO, que finalizará con IRET al igual que la llamada por INT21.

Ahora en la cima de la pila sólo se modificará el valor del IP que en lugar de ser 3506 sería 3504:

SS:SP = 3A88: FFDE 04 (IP)
 FFDF 35 (IP)
 FFE0 88 (CS)
 FFE1 3A (CS)
 FFE2 00 (RE)
 FFE3 29 (RE)
 FFE4 25 (AX)
 FFE5 33 (AX)
 FFE6 00 (RE)
 FFE7 29 (RE)
 FFE8 07 (IP)
 FFE9 20 (IP)

Igualmente como en INT21, el registro RE va a cambiar durante la ejecución, pues ahora el flag I que forma parte del mismo debe cambiar de 1 al 0, deshabilitando (enmascarando) cualquier otra interrupción por hardware ocurra mientras se ejecuta la subrutina del SO que atiende a IRQxx. Asimismo, por ser flag I = 0 la CPU está en modo "kernel".

A los efectos de localizar la subrutina que atiende a IRQxx, de acuerdo con los datos del ejercicio, en la zona de vectores se tendrá:

0000:0032 20 (IP)
 0000:0033 34 (IP)
 0000:0034 5B (CS)
 0000:0035 25 (CS)

Entonces CS:IP = 255B : 3420 es la dirección dónde empieza la subrutina del SO que atiende a INTxx.
 El resto es semejante a lo tratado para INT21.

COMPLEMENTO

MANEJO DEL PORT PARALELO

(De un trabajo del Ing. Rubén López)

Cuando se arranca una PC, el BIOS barre las direcciones de los puertos en un determinado orden, el primer dispositivo encontrado recibe la denominación (denominada lógica) LPT1 y así sucesivamente, guardando los valores asignados en una lista de dispositivos que corresponderá a la implementación particular del hardware de la PC y del BIOS en cuestión, no habiendo una garantía cierta de que dicho nombre lógico en distintas máquinas pueda corresponder a la misma dirección física.

La secuencia típica con que los dispositivos físicos son detectados corresponde a la secuencia de direcciones hexa 3B8, 378 y 278, recibiendo la denominación lógica LPT1, LPT2 y LPT3 respectivamente, si bien también es muy común que para el mismo orden de barrido se asigne los nombres LPT3, LPT1 y LPT2.

Esto nos plantea la necesidad de verificar en cada máquina, cuál es la dirección física del puerto lógico que se deseé utilizar, ya que desde assembler siempre accederemos a él no por su denominación lógica sino por su dirección física.

También es conveniente recordar que los dispositivos lógicos LPT1 y LPT2, en su configuración estándar, están asociados a IRQ 5 e IRQ 7 respectivamente: pero estos valores pueden haber sido modificados mediante "Jumpers" configurables en la placa del port. o mediante el Setup de configuración de la PC para los ports implementados "on board".

1) Dirección del port de datos de la interfaz "Port Paralelo" considerada como dirección base

Una interfaz "Port Paralelo", como se estudió (Unidad 1), se encuentra conformada a su vez por tres registros o ports, cada uno de los cuales tiene su propia dirección física, la cual se determina a partir de la dirección perteneciente al port de datos (también llamada "dirección base" o "base"). A continuación se describen las características más importantes de cada uno de estos registros y su dirección:

Port de datos: Se trata de un registro de sólo salida correspondiente a las 8 líneas de datos, cuya dirección Base según lo expresado podrá ser 3B8, 378 ó 278.

Port de status: Es un registro de sólo entrada utilizado para las 5 líneas de estado (Status), cuya dirección se calcula como Base + 1

Port de control: Registro de tipo bidireccional utilizado para el manejo de 4 líneas de CONTROL, su dirección se obtiene como Base + 2

Si por ejemplo el port LPT1 se encuentra en la dirección 278, resulta que el port de datos tiene la dirección física 278h, el port de status le corresponde la dirección 279h y el port de control se encontrará en 27Ah.

2) Consideraciones acerca del Port de Datos:

Es muy importante tener en cuenta para el manejo del port de datos, las siguientes consideraciones:

- Si se escribe en el port de datos, la escritura se realiza en dicho registro, que guarda el valor enviado, permitiendo a la CPU proseguir con otra tarea.

- Si se realiza una lectura del port de datos, se lee el valor de las líneas D0 a D7, correspondientes al último valor enviado al port de Datos.

3) Disposición (PinOut) de contactos ("pines") del conector DB25 en relación con los registros ports de la interfaz "Port Paralelo"

La siguiente tabla describe la correspondencia que existe entre el conector DB25 y cada bit de los registros que componen la interfaz "Port Paralelo"

PORT DATOS	PIN	PORT CONTROL	PIN	PORT STATUS	PIN
D ₀ (Out)	2	/C ₀ (I/O inv)	1	---	
D ₁ (Out)	3	/C ₁ (I/O inv)	16	---	
D ₂ (Out)	4	C ₂ (I/O)	17	---	
D ₃ (Out)	5	/C ₃ (I/O inv)	14	S ₃ (IN)	15
D ₄ (Out)	6	C ₄ (FF)	--	S ₄ (IN)	12
D ₅ (Out)	7	---	--	S ₅ (IN)	13
D ₆ (Out)	8	---	--	S ₆ (IN)	10
D ₇ (Out)	9	---	--	/S ₇ (IN inv)	11

Importante: Se debe notar que el bit 7 del Port de Status y que los bits 0, 1 y 3 del Port de Control sus salidas están invertidas (indicación mediante /), por lo que su valor lógico es 1 cuando su señal eléctrica está con un valor bajo de tensión (0 volts).

4) Secuencia para generar una interrupción mediante una señal del exterior:

Mediante el bit 6 (S₆) del Port de STATUS (pin 10) (denominado generalmente Acknowledge) se podrá generar una interrupción desde el exterior siempre que la señal perteneciente al bit 4 (C₄) se encuentre asignada por jumper o setup, de modo de poder activar a la IRQ 5 ó 7 (típicamente usados por los LPT), y se cumpla que "en el bit 4 del Port de CONTROL (C₄) por software se escriba un 1, y luego el bit S₆ (pin 10) desde el exterior pase de bajo a alto (proceso que se logra cuando en esa posición si hay un cero se le escribe un uno)"

5) Instrucciones de E/S en Assembler:

Para manejar el port paralelo desde assembler en modo real (compatible XT) contamos con las siguientes instrucciones:

Instrucciones disponibles: <i>in AL/AX, dir</i> <i>out dir, AL/AX</i>	y sus similes: <i>in AL/AX, DX</i> <i>out DX, AL/AX</i>
---	---

Ejemplo 1:

Para poder saber si en una determinada dirección física tenemos implementado un port lógico, la forma más sencilla será realizar un programa en assembler que escriba y luego lea una determinada dirección, si el valor leído es igual al que previamente hemos escrito, en principio podríamos pensar que dicho port en nuestra PC se encuentra implementado. El siguiente programa nos brinda una 1ra aproximación para detectar la existencia del port en la dirección física 378_h:

MOV AL, 55	carga AL con 55 _h (número que alterna unos y ceros)
MOV DX, 378	carga DX con 378 _h
OUT DX, AL	copia el valor de AL en port 378 _h
MOV AL, 0	limpia AL si es que por casualidad guarda 55 _h
IN AL, DX	carga AL con el valor del port 378 _h

A primera vista este programa de manera sencilla permite grabar en el port 378_h el valor contenido en el registro AL (o sea 55_h); y luego leer el valor que tiene dicho port, copiándolo nuevamente al registro AL (a fin de verificar que sea 55_h).

Pero pueden darse posibles problemas de lectura debido a la característica "read/back" del port (posibilidad de error cuando luego de una instrucción OUT sobre un port sigue una IN para leerlo, siendo que aún el hardware no terminó de escribirlo), siendo además dependiente de la implementación particular del hardware de la PC que sea factible o no dicho problema. Esto se manifiesta por el error producido cuando después de haber escrito un valor en un port, al realizar la lectura del mismo no devuelve dicho contenido, sino cualquier otro valor.

Para poder solucionar este inconveniente que nos impide determinar la existencia o no de un port en una de las direcciones citadas, se utiliza entre la escritura y la lectura del mismo la instrucción **JMP dir** (siendo dir la dirección de la instrucción siguiente). Esta instrucción por un lado genera una demora por su propio tiempo de ejecución, mientras que por otro lado vacía la cola de instrucciones del pipeline de la CPU (Unidad 1), garantizando de esta forma demoras efectivas. La secuencia anterior ahora quedaría:

	MOV AL, 55	carga en AL la cte 55 (1 y 0 alternos)
	MOV DX, 378	carga en DX dirección port de datos
	OUT DX, AL	copia valor de AL en port 378 _h
	JMP SGTE1	demora + vacía la cola del pipeline
SGTE1	JMP SGTE2	demora + vacía la cola del pipeline
SGTE2	MOV AL, 0	limpia AL
	IN AL, DX	carga AL con el valor del port 378 _h
	JMP SGTE3	demora + vacía la cola del pipeline
SGTE3	JMP SGTE4	demora + vacía la cola del pipeline
SGTE4	

Finalmente podremos concluir, que si el valor devuelto a AL no es el mismo que se mandó al port de datos, es muy probable que a partir de esa dirección no exista la interfaz "Port paralelo", o esté interfiriendo la plataforma gráfica de 32 bits W2000/XP, razón por la cual no se recomienda trabajar para esta actividad en modo real con ella, mientras que con las plataformas W9x/NT no se manifiesta esta interferencia.

APENDICE: Circuito simple con leds para visualizar desde el exterior el port de datos (port de salida)

Mediante el sencillo circuito de leds y resistencias que se indica a continuación, que puede armarse directamente sobre un conector DB25 enchufable en la entrada del port paralelo de una PC, podremos visualizar el efecto de distintos programas manejando dicho port

- (18) ---Ⓐ---|←-- (2)
- (19) ---Ⓐ---|←-- (3)
- (20) ---Ⓐ---|←-- (4)
- (21) ---Ⓐ---|←-- (5)
- (22) ---Ⓐ---|←-- (6)
- (23) ---Ⓐ---|←-- (7)
- (24) ---Ⓐ---|←-- (8)
- (25) ---Ⓐ---|←-- (9)

Simbología:

- (xx) = nro del pin del conector DB25
- Ⓐ-- = Resistencia 1K - 1/8 W
- ←-- = LED de 3 mm. de diámetro

Ejemplo: El siguiente programa realiza el encendido de los ocho leds en forma sucesiva, de a uno por vez, mediante un barrido desde el bit de menor peso hacia el bit de mayor peso; concluido el barrido el programa se detiene.

	MOV DX, 378	carga dirección port de datos en DX
	MOV AL, 1	carga 00000001 para encender el diodo extremo
UNO	OUT DX, AL	conectado al pin 2 del conector DB25
CDEM	MOV AH, 20	escribe el dato en el port
		se prepara para repetir 20h veces la demora que
DEM	MOV BX, FFFF	sigue, para hacer visible el encendido del led
SGTE	JMP SGTE	inicializa contador de demora con FFFF = 65.535
	DEC BX	vacia la cola de instrucciones del pipeline de la UCP
	JNZ DEM	decrementa el contador BX
	DEC AH	repite núcleo rutina de demora hasta que BX=0
	JNZ CDEM	repite rutina de demora hasta que AH = 0
	SHL AL, 1	shift de 1 bit para encender próximo led
	CMP AL, 0	verificación si encendieron todos (inst. no necesaria)
	JNZ UNO	
	INT 20	fin

Nota: con el valor de AH se ajusta la duración de la demora de 65.536 loops en BX

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 = 00000000000011100 (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 = 00111000 = 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 2 números de varios dígitos son restados, el flag C es tenido en cuenta para los dígitos siguientes, usando instr. SBB. AAS opera sólo sobre el registro AL. Puede cambiar los flags AC y C.

ADC Instrucción exemplificada en el ejercicio 37 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 38 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 46.

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 decremente SP en dos y se copia en la pila el valor de CS. Luego otra vez se decremente SP en dos, y se copia en la pila el IP con el offset de la instrucción que sigue a CALL. Después se carga 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 44, 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; ver ejemplo en instruc CMPB)

CLI (Clear Interrupt flag). Ordena poner a cero el flag I, para que la UCP no responda (enmascare) 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 instr.)

CMP Instrucción exemplificada en el ejercicio 9, 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 comparar 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 instr 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 vez de usar CLD se usa 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 45

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 2

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 ejercicios 36 y 37

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 43

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 Ordena sumar solamente uno. Instrucción exemplificada en el ejercicio 2.

INTxx Instrucción exemplificada en el ejercicio 47

IRET Instrucción exemplificada en el ejercicio 47

JA = JNBE Instrucción exemplificada en el ejercicio 12

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

JB = JC = JNAE Instrucción exemplificada en el ejercicio 15

JBE = JNA Instrucción exemplificada en el ejercicio 30

JCXZ (Jump if the CX register is Zero)

JE = JZ Instrucción exemplificada en el ejercicio 14

JG = JNLE Instrucción exemplificada en el ejercicio 18

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

JL = JNGE Instrucción exemplificada en el ejercicio 18

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

JMP Instrucción exemplificada en el ejercicio 3.a

JNE = JNZ Instrucción exemplificada en el ejercicio 2

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 3

JPE = JP Instrucción exemplificada en el ejercicio 32

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

LEA Transfiere a un registro el valor de la dirección del operando fuente.

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 valor de CX. Cada vez que LOOP se ejecuta CX es automáticamente decrementado en 1. Mientras CX≠0 salta a ejecutar la instrucción indicada por la dirección que acompaña a LOOP. (Esta no puede superar el rango de +127 a -128 bytes desde la instr. 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 42

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 38

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 (apuntada por SP), y los transfiere al registro o destino indicado en la instrucción. Luego se incrementa en dos el SP. Ejemplificada en el ejercicio 46

POPF Instrucción exemplificada en el ejercicio 46. 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 46

PUSHF Instrucción exemplificada en el ejercicio 46

RCL Ordena la rotación a izquierda dibujada un cierto número de veces, en la que se incluye al flag C (comparar con ROL) $C \leftarrow MSB \leftarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB$

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) $C \rightarrow MSB \rightarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB$

$C \rightarrow MSB \rightarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB$

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.

Ej: 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 46

ROL (Rotate Left) exemplificada en el ejercicio 38.

Ordena la rotación a izquierda dibuja, un número de veces

$C \leftarrow MSB \leftarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB$

ROR (Rotate Right)

Ordena la rotación a derecha dibujada, un cierto número de veces

$MSB \rightarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB \rightarrow C$

SHL = SAL (Shift Left)

$C \leftarrow MSB \leftarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB \leftarrow 0$

Como indica el dibujo, ordena desplazar hacia la izquierda cada bit del operando un cierto número de posiciones (indicado por CL¹). 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

$MSB \rightarrow MSB \rightarrow \underbrace{\hspace{1cm}}_{\downarrow} \underbrace{\hspace{1cm}}_{\uparrow} LSB \rightarrow C$

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 (Subtract 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 I el flag D, de dirección.

STI (Set Interrupt flag):

Ordena hacer I 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 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.

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. Usada en el ejercicio 34

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.

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