DULLIN-STRASSENBURG

EL LENGUAJE MAQUINA PARA CPC

464, 664 & 6128

UN LIBRO DATA BECKER
EDITADO POR FERRE MORET, S.A.

DULLIN-STRASSENBURG

EL LENGUAJE MAQUINA PARA CPC

464, 664 & 6128

UN LIBRO DATA BECKER

EDITADO POR FERRE MORET, S.A.

Este libro ha sido traducido por Don Joaquin Hommen, ingeniero electronico y experto conocedor del Commodore

Imprime: APSSA, ROCA UMBERT, 26 - L'HOSPITALET DE Ll. (Barcelona)

ISBN 84-86437-35-0

Depósito legal B-10.515/86

Advertencia importante

Los circuitos, procedimientos y programas reproducidos en este libro son divulgados sin tener en cuenta el estado de las patentes. Están destinados exclusivamente al uso amateur o docente, y no pueden ser utilizados para fines comerciales. Todos los circuitos, datos técnicos y programas de este libro, han sido elaborados o recopilados con el mayor cuidado por los autores y reproducidos utilizando medidas de control eficaces. No obstante, es posible que exista algún error. FERRE MORET, S.A. se ve por tanto obligada a advertirles, que no puede asumir ninguna garantía, ni responsabilidad jurídica, ni cualquier otra responsabilidad sobre las consecuencias atribuibles a datos erróneos. Los autores les agradecerán en todo momento la comunicación de posibles fallos.

A G R A D E C I M I E N T O

La confección de un libro de este tipo en la mayoría de los casos sólo produce satisfacciones al autor. Para los amigos y amigas sólo queda la frustración, dado que nosotros los autores hemos de encontrarnos inmersos plenamente en el trabajo del libro (como en una "permanente borrachera informática") para poder acabarlo. Queremos agradecer a todos aquellos que han hecho posible este libro mediante su paciencia y ayuda. Damos las gracias especialmente a Birgitt Mikutta, Kristin Grünewald, Andreas Bethmann por su invalorable ayuda en la corrección del manuscrito, y a Ralph Dullin por la confección de los dibujos y tablas.

Agradecemos asimismo a la empresa Zilog Inc, USA que ha puesto a nuestra disposición los listados de comandos del Z80A.

P R O L O G O

Desde el momento en que pudimos disponer de uno de los tan prometidos ordenadores CPC, nos quedamos maravillados con esta prodigiosa máquina. El BASIC del CPC es realmente extraordinario. En el momento de ocuparnos de su estructura interna y de la programación máquina, notamos que lamentablemente todavía no están disponibles informaciones suficientes sobre el tema por el momento. Así se nos ocurrió la idea de confeccionar este libro.
La programación en lenguaje máquina ofrece algunas ventajas decisivas frente al lenguaje BASIC, en cuanto a velocidad y requerimientos de memoria para almacenamiento de datos. El objeto de este libro, consiste en facilitar al usuario del CPC el acceso al lenguaje máquina y de este modo permitirle el aprovechamiento de las ventajas antes mencionadas para sus programas.
Sin embargo el aprendizaje del lenguaje máquina no es tan sencillo porque ¿quién es capaz de comprender la siguiente línea sin ninguna explicación?

```
21,00,C0,36,CC,23,BC,20,FA,C9
```

Pero no se desanime tan pronto. Le resultará sencillo trabajar en lenguaje máquina si maneja el libro de la siguiente manera:

- Trabaje el libro a fondo capítulo a capítulo.
- Intente resolver los ejercicios
- Si ello le resulta difícil, revise nuevamente el capítulo.

Pero basta ya de buenos consejos. Desde ahora le invitamos a entrar en la aventura del LENGUAJE MAQUINA.

Los autores.

INDICE

CAPITULO III : LA SENTENCIA DE COMANDO DEL Z80

CAPITULO IV : LOS COMANDOS

ANEXO

CAPITULO I : INTRODUCCION

1.1 ¿Qué es el LENGUAJE MAQUINA?

El Lenguaje Máquina es el lenguaje de programación que el ordenador es capaz de tratar directamente. ¿Qué se entiende por ésto?. Como usted seguramente sabrá, cada ordenador posee un microprocesador que podemos considerar como el "cerebro" del ordenador. Este IC (Circuito Integrado) se denomina CPU (Central Processing Unit) o Unidad Central de Procesamiento (UCP). La CPU ejecuta las órdenes de la máquina, dirige el funcionamiento del ordenador y el de todos los periféricos. La Unidad Central es la pieza clave de todo el ordenador. Cuando programamos en lenguaje máquina, utilizamos comandos dirigidos directamente a la CPU, la cual puede ejecutarlos inmediatamente. Con ello, el lenguaje máquina depende de cada uno de los diferentes tipos de procesador. Los ordenadores CPC poseen un procesador Z80, que también halla aplicación en muchos otros microordenadores. El Z80 es una unidad central de gran rendimiento; entiende más de 600 comandos que trata a gran velocidad.

¿Por qué en realidad Lenguaje Máquina?

La mayoría de los ordenadores domésticos están equipados con BASIC. Como seguramente ya habrá observado, el aprendizaje de este lenguaje no presenta gran dificultad. El BASIC CPC llama la atención especialmente por su variedad de comandos. Da la impresión como si con este BASIC no quedaran nunca deseos sin cumplir, pudiendo quedar siempre bien resueltos todos los problemas de programación.

Imagínese usted:

> El Ministro de Asuntos Exteriores, Sr. BASIC, negocia con su colega, el Sr. CPU, en el país del Lenguaje Máquina. Desgraciadamente, sus conocimientos de ese idioma son muy escasos, de modo que solicita los servicios de la intérprete, la Srta. Interpreter, quien traduce sus frases al lenguaje máquina. Como es de suponer, la Srta. Interpreter, a pesar de ser una excelente traductora, siempre es un poco más lenta en la traducción que el político en su discurso, y debido a ello, se prolonga innecesariamente el diálogo de la negociación.

Al programar en BASIC encontramos exactamente con el mismo problema. A través del interpretador, el ordenador ha de interpretar primero el BASIC escrito por el programador. El interpretador de BASIC es una parte de los programas ya incorporados, que interpreta el programa siguiendo uno a uno cada comando dado. Acto seguido, provoca la inmediata ejecución de los mismos. Para ser más exactos: el interpreter reconoce el comando BASIC y luego, resuelve la ejecución de dicho comando BASIC a través de la llamada a las rutinas en Lenguaje Máquina de cada comando concreto.

Por ejemplo:

```
MODE 2
```

En primer lugar, el interpretador lee el comando carácter a carácter, donde por ejemplo, los espacios, dos puntos, paréntesis y comas le indican cuando ha finalizado una palabra. La palabra (MODE) la compara con las entradas de la tabla de comandos en BASIC del ROM. Si no la encuentra, intenta interpretar la palabra como variable. Si ello tampoco funciona, saca un mensaje de error. Si el interpretador encuentra la palabra, bifurca hacia la

dirección de salto correspondiente a la palabra. Allí se lee el siguiente valor, en nuestro ejemplo: 2; se verifica la admisibilidad de ese argumento y se ejecuta el comando. Luego, se vuelve de nuevo al interpretador y el proceso descrito arriba comienza nuevamente. La labor que, en nuestro ejemplo desempeña la Srta. Interpreter, requiere algo de tiempo. Este tiempo se ahorra si programamos directamente en lenguaje máquina.

Desgraciadamente el lenguaje máquina presenta el inconveniente de ser muy abstracto. Básicamente, al hombre le resulta difícil imaginarse cifras. Esta dificultad es la causa de la creación de los denominados "Lenguajes Superiores de Programación", tales como el LOGO, BASIC, etc., que operan con conceptos y no con cifras. Estos lenguajes presentan un compromiso en la comunicación entre el hombre y la máquina. Sin embargo, desgraciadamente ello trae consigo importantes inconvenientes en cuanto a la velocidad, a la necesidad de espacio en la memoria y, a menudo también, en cuanto a la posibilidad de programación.

Todos los lenguajes de programación, tales como el Cobol, Pascal, Fortran, etc, han de ser traducidos antes de ser ejecutados por el ordenador. Aquí se distingue entre interpretador y compilador:

Un interpretador, como por ejemplo el del ordenador CPC, traduce paso a paso todos los comandos del programa, ejecutándolos simultáneamente. El interpretador es, por consiguiente, un traductor simultáneo; es decir, que durante el desarrollo del programa, cada comando se interpreta nuevamente. Por ello, las modificaciones en BASIC no plantean problemas.

Contrariamente a ello, un compilador traduce cada programa una única vez, creando al mismo tiempo otro programa equivalente en lenguaje máquina. Sólo entonces puede ejecutarse el programa en lenguaje máquina. Por lo general, el proceso del compilador dura bastante, pero una vez realizado, el programa en lenguaje máquina

producido se ejecuta a gran velocidad. Si se desea modificar el programa, deberá compilarse nuevamente la versión modificada del mismo. Esto hace que las modificaciones en tales programas sean de larga duración. En este libro le presentamos un compilador que traduce de lenguaje ensamblador a Código Máquina o Lenguaje Máquina. A este compilador se le denomina ENSAMBLADOR.

Aquí puede reconocerse una ventaja fundamental del lenguaje máquina: los programas en lenguaje máquina alcanzan una velocidad de ejecución hasta 1000 veces superior que los programas en BASIC.

Asimismo los programas en lenguaje máquina escritos a medida para resolver un problema especial, son más rápidos que los programas en lenguaje máquina confeccionados mediante compilador. El comando RETURN en BASIC tiene un tiempo de ejecución de aprox. 0.6 milisegundos; el comando correspondiente RET en lenguaje máquina necesita sólo 2.5 microsegundos. Ello hace que la orden RET en lenguaje máquina sea casi 240 veces más rápida, y en su equivalente para el comando POKE en lenguaje máquina, hasta casi 1000 veces. Tales diferencias son importantes a la hora de, por ejemplo, ordenar y buscar en grandes cantidades de datos, desplazar contenidos en la memoria, como es necesario para el Scrolling o también para programas de textos. Además, la programación de gráficas complejas en BASIC es demasiado lenta; es decir, el lenguaje máquina se hace indispensable para programas de juegos y gráficas profesionales.

Existen también otras ventajas:

Por regla general, los programas en lenguaje máquina son más cortos que los programas en BASIC, con lo cual se ahorra un importante espacio en la memoria. Tan pronto como usted haya escrito sus primeros programas en lenguaje máquina comprobará que un programa en lenguaje máquina de más de 500 bytes ya es muy largo y que pueden hacerse muchas cosas con él. Por el contrario, para un programa en BASIC de similares características, se necesitaría mucho más espacio de memoria.

NOTA: La longitud de un programa BASIC en BYTES puede calcularse con el CPC con "PRINT HIMEM-FRE(0)-370".

Otra ventaja del lenguaje máquina estriba en que únicamente con él pueden aprovecharse al máximo todas las posibilidades del ordenador. En primer lugar, con lenguaje máquina es más rápido programar, por ejemplo datos de entrada o salida. Asimismo, con ayuda de programas propios pueden controlarse periféricos de entrada/salida, y recibir datos desde ellos. También sólo en lenguaje máquina es posible desarrollar estructuras de datos propias, las cuales, generalmente, ahorran mucho más espacio que las dadas en BASIC. Grandes cantidades de datos como las que aparecen - entre otras- en el tratamiento de textos, pueden de este modo almacenarse mejor en la memoria disponible.

Estos ejemplos deberían ser suficientes para poner de manifiesto la necesidad del lenguaje máquina incluso en el caso de un ordenador CPC con muy buen lenguaje BASIC. Sin embargo hay que decir que la programación en lenguaje máquina presenta un gran inconveniente.

El lenguaje máquina es el lenguaje de la CPU del ordenador y por lo tanto, el lenguaje más orientado hacia la máquina. Ello supone que el programador deberá pensar de un modo muy abstracto para llegar a entender este lenguaje. El ser humano tiende a pensar en palabras y asociaciones; es decir, que un lenguaje orientado hacia el hombre utiliza estructuras y conceptos claros. En el caso del lenguaje máquina esto no es así. Principalmente, la CPU sólo entiende cifras; es decir que un programa para la máquina se reduce simplemente a una serie de números y no a una secuencia de conceptos. En tal forma, la programación en lenguaje máquina para programas extensos sería casi imposible. Por ello los "pioneros de la informática" ya desarrollaron una especie de lenguaje intermedio, capaz de hacer más inteligibles y claros los programas en lenguaje máquina. A este lenguaje se le denominó ENSAMBLADOR. El lenguaje ensamblador ordena a cada código de la máquina (o sea, a una cifra) una serie de símbolos. Tales símbolos se componen de:

1. Las palabras del comando: consisten generalmente en una abreviación de la palabra inglesa de comando, denominada también mnemónico.

2. El operando que, por ejemplo, especifica las direcciones, constantes o similares (concernientes al comando).

De este modo, la confección de un programa en lenguaje máquina se simplifica al escribirlo en ensamblador. Luego, este lenguaje ensamblador es traducido automáticamente al lenguaje máquina por el denominado "Programa Ensamblador". Un ensamblador de este tipo (un compilador para lenguaje ensamblador) es el que presentamos en este libro, para que puedan los lectores programar en ensamblador (nos referimos aquí al lenguaje ensamblador o Assembler).

1.2 EL PRIMER PROGRAMA MAQUINA

Para enseñarles que es muy útil el aprendizaje del lenguaje máquina, ofrecemos la comparación entre un programa BASIC y su primer programa en lenguaje máquina.
Entre las siguientes líneas BASIC

```
10 HL=&C000
20 POKE HL,&CC
30 HL=HL+1
40 IF HL <=&FFFF THEN 20
50 RETURN
```

A continuación entre en modalidad directa "MODE 2" tecleando seguidamente "GOSUB 10" y observe el resultado.

El siguiente programa carga el programa máquina con el mismo propósito que el anterior en BASIC:

```
10 MEMORY &9FFF
20 FOR I=&A000 TO &A009
30 READ a
40 POKE i,a
50 NEXT I
60 END
70 DATA &21,&00,&C0,&36,&CC,&23,&BC,&20,&FA,&C9
```

Ahora introduzca nuevamente en modalidad directa "MODE 2", cargue el programa máquina con "RUN" y llame al programa máquina cargado con CALL &A000, quedará sorprendido.
Como habrá podido comprobar, los tiempos son:

- Programa BASIC: Aprox. 1 minuto
- Programa máquina: Aprox. 1/10 segundo.

El tiempo de ejecución puede calcularse teóricamente.
Para nuestro programa ejemplo sería por ejemplo 0,1106 segundos.

La longitud comparativa de los programas:

- Programa BASIC: 88 bytes
- Programa máquina: 10 bytes

Desde &A000 hasta &A009.

Esperamos que no haya sufrido un schock demasiado fuerte con todas estas novedades. En los capítulos siguientes se lo explicaremos todo paso por paso.

Para la analogía de los programas:

```
BASIC                              ENSAMBLADOR

10 HL=&C000                        LD HL,C000
20 POKE HL,&CC                     LD (HL),&CC
30 HL=HL+1                         INC HL
                                   CP H
40 IF HL < =&FFFF THEN 20            JR NZ,$-6 > A006
50 RETURN                          RET
```

EXPLICACION:

Línea 10: Aquí se posiciona el valor para la variable HL, es decir el registro HL en el inicio de la memoria de pantalla (LD = inglés Load = Carga).

Línea 20: En esta línea se almacena el valor &CC en la dirección HL. Dado que la memoria de pantalla está posicionada desde &C000 hasta &FFFF este comando provoca una modificación de pantalla.

Prueben con diferentes valores para la dirección HL en la memoria de pantalla en modalidad directa (HL puede encontrarse entre &C000 y &FFFF) así como para el argumento (en nuestro ejemplo &CC) que puede adoptar valores entre &00 y &FF (p. ej. POKE &C100,&AA).

Línea 30: Incrementa la variable HL es decir el registro HL, en 1. (INC=ingl. Increase = incrementar).

Línea 40: Comprobación si HL es mayor que &FFFF es decir si se ha llegado al final de la memoria de pantalla. Esto en lenguaje máquina tiene que realizarse con 2 comandos: CP (ingl. = Compare = comparar); JR (Jump relative = bifurcación relativa); NZ (ingl. not zero = no cero). Puede decirse: "bifurca, si se verifica condición no cero (NZ).".
Esta explicación no es completamente exacta, más adelante en este libro haremos la aclaración correspondiente.

En las siguientes líneas presentamos el listing de ensamblador del ejemplo:

LISTADO ENSAMBLADOR para el programa en lenguaje máquina

Dir.	Código	N.BASIC	Comando	Ensamblador	Comentario
A000	2100C0	10	LD	HL,C000	Inicio memoria de pantalla
A003	36CC	20	LD	(HL),&CC	&CC valor a grabar en memoria pantalla
A005	23	30	INC	HL	HL = HL + 1
A006	BC	40	CP	H	Comparación vs 0
A007	20FA	50	JR	NZ,$-6>A006	Si no cero (NZ<>0), vuelve al paso 6 del programa; si 0 al siguiente comando
A009	C9	60	RET		Retorno al BASIC

Esperamos haber despertado su curiosidad, ya que a partir de ahora, pasaremos al tratamiento sistemático del lenguaje máquina y explicaremos los ejemplos dados arriba.

1.3 SISTEMAS NUMERICOS.

En el capítulo anterior se ha utilizado el símbolo & como indicador para un número del sistema hexadecimal (hexadecimal - 16). ¿Qué significa todo esto?.
En la realización de sistemas de cálculo electrónico hay 2 formas de representación numérica.

Analógica: En un ordenador analógico se representa un número mediante un voltaje, por ejemplo 1=1 voltio y 100=100 voltios. Por ello un reloj con agujas es un reloj analógico. El aumento del tiempo corresponde (de forma análoga) al número de vueltas de la aguja.

Digital: Un ordenador digital se basa en la idea de no utilizar la medida del voltaje sino únicamente 2 estados: hay corriente o no hay corriente. Digital significa la representación de cantidades mediante números. Los estados ACTIVO y DESACTIVO corresponden a los números 0 y 1.

De esta forma un ordenador digital solamente tiene 2 dígitos disponibles. Con ayuda de este concepto se realiza la representación numérica en el ordenador.
Para tareas fijas puede ser más conveniente trabajar con un ordenador analógico (por ejemplo control de maquinaria). Si se quieren resolver diferentes problemas en un ordenador, el ordenador digital supera al analógico porque la programación del ordenador analógico en la forma que conocemos es imposible. Esto quiere decir que todos los ordenadores personales y domésticos son ordenadores digitales que trabajan con el sistema dual (con las cifras 0 y 1).

Para el programador son significativos los siguientes sistemas de numeración:

1. Sistema decimal
2. Sistema binario
3. Sistema hexadecimal

Los sistemas numéricos son esquemas de cifras ordenadas según un determinado principio. Cada dígito puede convertirse a otro sistema numérico. En todos los sistemas numéricos, el valor posicional de un dígito aumenta de derecha a izquierda.
Para explicar los restantes sistemas numéricos, partiremos del conocido sistema decimal.

SISTEMA DECIMAL

```
 Millar     Centena     Decena     Unidad          - Valor posicional
   7          3           5          6             - Dígitos
<-------------------------------------------------------------------
```

El valor posicional aumenta de derecha a izquierda

Potencia	Número	Denominación
10^0	1	U-nidad
10^1	10	D-ecena
10^2	100	C-entena
10^3	1000	M-illar
10^4	10000	D-ecena de mil
10^6	1000000	Millón

El número decimal 1335 puede representarse como sigue:

1335 significa: 1M + 3C + 3D + 5U

435 significa: 4C + 3D + 5U

1335 es : 1*1000 + 3*100 + 3*10 + 5*1

$$1335 \text{ es también: } 1*10^3 + 3*10^2 + 3*10^1 + 5*10^0$$

Se define una potencia de exponente 0 con valor 1.

Ejemplo: $10^0 = 1$, $2^0 = 1$, $x^0 = 1$

El Sistema DUAL

El sistema Dual, se basa en el mismo principio. La diferencia fundamental en este caso es que la cifras no se forman a través de potencias de 10 sino por medio de potencias de 2.
La base del sistema Dual es entonces 2.

Binario 10101101 = Decimal 173

```
 7  6  5  4  3  2  1  0
2  2  2  2  2  2  2  2        Valor de posición

1  0  1  0  1  1  0  1        Cifra
```

```
        7      6      5      4      3    2      1      0
173 = 1*2  + 0*2  + 1*2  + 0*2  + 1*2  1*2  + 0*2  + 1*2
```

173 = 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1

De momento ha aprendido ya a tranformar números binarios a sistema decimal. Naturalmente, este proceso también se puede hacer a la inversa. Para aclarar el proceso inverso, observemos la cifra decimal 173 antes calculada.

Pensemos qué potencia de dos hay aún en esta cifra. Una ayuda: en principio el sistema binario puede aplicarse a números de n-dígitos. Sin embargo, en el ámbito del ordenador sólo se utilizan números binarios de 8-posiciones. Pueden aparecer las siguientes potencias de dos.

	7	6	5	4	3	2	1	0
Potencias de dos	2	2	2	2	2	2	2	2
valor decimal	128	64	32	16	8	4	2	1

En este caso 2^7=128 es la potencia de dos más alta. Ahora calculamos la diferencia entre 173 y 128. El resultado es 45. Con este resto se procederá igual que anteriormente. Volvemos a buscar la potencia de dos más alta contenida en ese valor. Utilizando la tabla, se encuentra fácilmente y resulta 2^5=32. Finalmente, volvemos a calcular la diferencia: (45-32=13).

El proceso descrito se sigue aplicando hasta que el resto sea cero.

```
2^3=8      (13-8=5)
2^2=4      ( 5-4=1)
2^0=1      ( 1-1=0)
```

Así hemos obtenido las siguientes potencias de dos:
2^7, 2^5, 2^3, 2^2, 2^0.
Debajo de cada potencia obtenida escribiremos un uno y debajo de la potencia inexistente un cero:

```
 7   6   5   4   3   2   1   0
2   2   2   2   2   2   2   2

1   0   1   0   1   1   0   1  =  173
```

La cifra decimal 173 viene expresada por lo tanto en el sistema binario por 10101101. En lo sucesivo, representaremos las cifras binarias anteponiendo el signo &X.

p. ej. 173 = &X 10101101

BIT Y BYTE

Un bit es la unidad de información más pequeña a partir de la cual se componen todas las demás informaciones. BIT es la abreviación

de "binary digit", lo que equivale a decir: cifra binaria. Se habla de un BIT activado cuando el BIT tiene el estado 1, o de un BIT desactivado cuando tiene el estado 0.

Los ordenadores CPC tienen un procesador de 8-BITS; es decir, puede tratar largas cifras binarias de 8-BITS, que comprende los valores decimales 0-255.

Número binario:

```
1 0 1 1 0 1 1 1

a d a a d a a a          a = BIT activado;   d = BIT desactivado

7 6 5 4 3 2 1 0          número de BITS
```

A cada BIT y a cada cifra se le asigna un número de BIT. El BIT con el valor posicional más bajo, es decir, el que está más a la derecha, tiene el número 0. La numeración aumenta de derecha a izquierda. El número de BIT corresponde al exponenete de la potencia de 2, que determina el correspondiente valor posicional.

En el ordenador, tiene sentido algunas veces imaginarse los estados del BIT como un interruptor.

```
INTERRUPTOR ABIERTO = 1
INTERRUPTOR CERRADO = 0
```

En un número de 8 interruptores se presentan valores de 0-255, es decir, 256 estados diferentes del interruptor.

El conjunto de 8 interruptores (BITS) recibe el nombre de BYTE. Un BYTE puede colocarse en una determinada posición de memoria del ordenador. ¿Pero cómo se memorizan o graban números superiores a 255?.

Para ello se divide el número en 2 mitades: el LOW Byte (inglés low=bajo; byte de valor bajo) y el HIGH byte (inglés high=alto; byte de valor alto). Estos bytes son entonces colocados en 2 posiciones contiguas de la memoria.

El HIGH byte y el LOW byte se calculan de la siguiente forma:

Número dividido por 256 = (High byte) + resto
El resto de la división será el LOW byte.

Recordemos: El número 255 es el valor máximo representable con un byte compuesto de 8 bits consecutivos.

Ejemplo: el número 34065 debe descomponerse en un LOW y en un HIGH byte.

```
34065 / 256 = 133   resto 17
34065       = 133 * 256 + 17
```

133 = HIGH byte
17 = LOW byte

La generalización codificada en BASIC sería:

```
1. HB = INT(Número / 256)      HB = High byte
   LB = Número - HB * 256      LB = Low byte
```

Esta forma se aplica en números de tamaño variable.

```
2. HB = Número < 256           HB = High Byte
   LB = Número MOD 256         LB = Low Byte
```

Esta segunda fórmula se utiliza en números del rango comprendido entre 32768 y 32767.

Con ello, un número comprendido entre 256 y 65535 en la memoria necesita 2 bytes.

Para simplificar la representación de números, colocados de esta forma en la memoria, es mejor adoptar otro sistema de numeración.

El sistema Hexadecimal.

La base del sistema hexadecimal es 16.
Recordemos:
La base del sistema decimal es 10.
La base del sistema binario es 2.

Para representar números cuyo valor sea superior a 10, se utilizan en el sistema hexadecimal las letras A hasta F.

Sistema decimal:
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...

Sistema hexadecimal:
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,...

En primer lugar convertiremos números hexadecimales a decimales:

Potencia	Valor
16^0	1
16^1	16
16^2	256
16^3	4096

```
&3ABF=3*16^3 + 10*16^2 + 11*16^1 + 15*16^0
&3ABF=3*4096 + 10*256  + 11*16   + 15* 1
&3ABF= 12288 +   2560  +   176   + 15
&3ABF= 15039
```

Otro ejemplo:

```
&1A3E = 1*16^3 + 10*16^2 + 3*16^1 + 14*16^0
&1A3E = 1*4096 + 10*256  + 3*16   + 14*1
&1A3E =   4096 +   2560  +   48   +   14
&1A3E =   6718
```

Ahora convertimos números decimales en números hexadecimales.

El procedimiento de cálculo es similar al descrito en páginas anteriores. Supongamos que el número decimal 45380 debe representarse en el sistema hexadecimal:

1. Paso: Pensemos qué potencia máxima de 16 puede contener este número (pueden utilizarse para ello las tablas de conversión).

2. Paso: Dividimos nuestro número (45380) entre el valor obtenido (4096) y convertimos el número decimal así obtenido en número hexadecimal.

   ```
   45380/4096 = 11 Resto 324
   Decimal = 11 => Hexadecimal B
   ```

3. Paso: Ahora seguimos el mismo procedimiento con el resto (324). Dividimos ahora entre el número siguiente de la tabla (que es 256).

   ```
   324/256 = 1  Resto 68
   Decimal=1 => Hexadecimal = 1
   ```

El cálculo anterior debe continuarse hasta el momento en que se obtiene 0 como resto de la división.

```
68/16    = 4    Resto 4
Decimal = 4 => Hexadecimal 4

4/1      = 4    Resto 0
Decimal = 4 => Hexadecimal 4
```

El número de nuestro ejemplo es &B144

La ventaja del sistema hexadecimal estriba en que el ser humano puede leer directamente el Low y High byte.
Para &3ABF por ejemplo:

- Pueden unirse las cifras correspondientes al High-Byte (3 y A). Tienen en este caso el valor decimal (3*16^1+10*16^0) = 58.

- Unimos ahora las cifras del Low-Byte (B y F) convirtiéndolas conjuntamente y tenemos en decimal (11*16^1 + 15*16^0)=191.

Entren entonces lo siguiente:

```
PRINT  PEEK(9),PEEK(10)
```

En ambas direcciones 9 y 10 está la dirección de salto, a la que bifurca el sistema operativo, cuando una rutina, situada por ejemplo en un módulo conectable, ha de ser llamada. Para una dirección de salto es posible un valor entre 0 y 65535 (es decir hasta &FFFF). Esta cantidad puede almacenarse mediante la ayuda del sistema High-Byte y Low-Byte. Ahora queremos calcular la dirección de SALTO. Con el comando anterior de BASIC, obtenemos de la dirección 9 el valor 130 (CPC 664 Valor=138 / CPC 6128 Valor=138) y de la dirección 10 el valor 185 (CPC 664 Valor=185 / CPC 6128 Valor=185). En sistema decimal la dirección de salto resulta entonces 185*256+130=47490 (CPC 664 185*256+138=47498 / CPC 6128 185*256+138=47498).

Realizaremos a continuación el mismo cálculo en el sistema hexadecimal:
130=&82 (CPC 664 138 = &8A / CPC 6128 138 = &8A) y
185=&B9 (CPC 664 185 = &B9 / CPC 6128 185 = &B9), como podrá comprobar con facilidad. El valor de la dirección de salto lo obtenemos escribiendo consecutivamente el High-Byte y el Low-Byte: 47490=&B982 (CPC 664 47498 = &B98A / CPC 6128 47498 = &B98A)

Así pues resulta igual de fácil separar una cifra hexadecimal High y Low- Byte, así como componerla a partir de Bytes High y Low. Por regla general el Low-Byte de una cifra se halla en la dirección más baja de la memoria, y a continuación le sigue el High-byte.

Con ésto, usted ha aprendido ya la primera ventaja del sistema hexadecimal. Además, es asimismo muy sencillo pasar del sistema decimal al sistema hexadecimal; para ello se subdivide una cifra binaria en 2 bloques, cada uno de 4 bits. El bloque desde el bit 0 al bit 3 se denomina Low-Nibble y el otro bloque, desde el bit 4 al bit 7, High-Nibble. Cada Nibble corresponde exactamente a un dígito hexadecimal. Ello es fácil de comprender, pues una cifra binaria de 4 bits puede contener como máximo el valor 15 (15=8+4+2+1). Todos los valores desde 0 al 15 pueden representarse asimismo por una cifra hexadecimal (0,1,...9,A,B,C,D,E,F). Observemos un ejemplo:

1 1 0 1	1 0 0 1
High Nibble	Low Nibble
8+4+1	8+1
13	9
&D	&9

Así pues: &X11011001=&D9

Con un poco de práctica, usted podrá leer directamente de una cifra de 4 bits su correspondiente valor hexadecimal y viceversa, valiéndose para ello de la tabla siguiente:

Sistema Binario	Sistema Hexadecimal	Sistema Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

De igual modo funciona la conversión del sistema hexadecimal al sistema binario. Cada número hexadecimal se sustituye por la combinación de 4 bits, p. ej. &C7=&X1100 0111.

La comprensión de la conversión de los diferentes sistemas numéricos es una base para la programación en lenguaje máquina.

Ejercicio:

1. Complete la siguiente tabla:

Decimal	Binario	Hexadecimal
130	?	?
?	&X1001001	?
57312	-----	?
?	-----	&C0B6
?	?	&37

2. El valor 37315 debe almacenarse a partir de la posición de memoria &A000. Calcule el High-byte y el Low-byte proporcionando los correspondientes comandos BASIC para almacenar el número.

3. A partir de la posición de memoria &0006 hay una importante dirección de bifurcación del sistema operativo. ¿Qué valor tiene?

Soluciones:

1.

Decimal	Binario	Hexadecimal
130	&X10000010	&82
147	&X10010011	&93
57312	-----	&DFE0
49334	-----	&C0B6
55	&X00110111	&37

2. High-Byte=145=&91; Low-Byte=195=&C3
```
POKE &A000,&C3:POKE &A001,&91
```

3. Low-Byte=PEEK(&0006), High-Byte=PEEK(&0007)
   Dirección de bifurcación=&0580

En el epílogo del libro encontrará una tabla de equivalencias con valores desde 0-255 (1-Byte) en los 3 sistemas de numeración.

1.4 ARQUITECTURA DEL ORDENADOR

Si queremos dedicarnos a la programación en lenguaje máquina, antes deberemos hacernos una idea de la estructura y organización

interna del ordenador. En el presente capítulo intentaremos desarrollar una imagen que se adapte a nuestras necesidades.

Usted ya sabe que posee un ordenador de 64K (K-Kilobyte=1024 Bytes). Ello significa que la capacidad de memoria de su ordenador es de 64*1024=65536 bytes. Como un byte se compone de 8 bits unidos con los que se representan los datos de la memoria interna, resulta que su ordenador se compone de 64*1024*8 bits, es decir aproximadamente medio millón de interruptores que pueden adoptar los estados encendido o apagado. Sin embargo, tal expresión no tiene sentido para el trabajo concreto del ordenador, por lo que se han unido 8 bits en un byte. Estos 64*1024 bytes se encuentran en la memoria RAM del ordenador. RAM significa: Random Access Memory, en castellano, memoria de lectura y escritura o también, memoria de trabajo. Los 65536 bytes de RAM están numerados desde &0000 hasta &FFFF. El número correspondiente al byte es su dirección. Esta dirección viene dada normalmente en una cifra hexadecimal. Podemos acceder directamente al RAM desde el BASIC, utilizando para ello los comandos PEEK y POKE. *PEEK (dirección)* lee el valor de los bytes situados en la dirección dada y *POKE dirección,valor* altera el contenido de la dirección dada con el valor indicado. A cada dirección le corresponde un byte compuesto de 8 bits, es decir entre 0 y 255 (&00-&FF), el valor a almacenar deberá estar por ello dentro de este ámbito. Naturalmente la dirección debe hallarse entre &0000 y &FFFF.

La memoria RAM se utiliza para almacenar los programas introducidos en el ordenador. El contenido codificado de pantalla se almacena a partir de la dirección &C000, donde en MODE 2 un punto corresponde a un bit activado y viceversa. El RAM contiene asimismo diversas importantes rutinas del sistema operativo e informaciones sobre colores actuales, ocupación de teclado, caracteres definidos, etc. Esta información puede alterarse con instrucciones descontroladas que, modificando el contenido del RAM, pueden colapsar el ordenador. Por ejemplo, jamás intente POKE &8,0.

La distribución del RAM es la siguiente:

&0000 - &0170	utilizado por el sistema
&0171 - &AB7F	para programas BASIC
&AB80 - &BFFF	sistema
&C000 - &FFFF	memoria de pantalla

Las direcciones comprendidas entre &AB80 y &BFFF corresponden al sistema en los modelos de CPC 464 sin Floppy. Si se trata en cambio de un CPC 464 con Floppy, corresponde modificar la tabla anterior con &A6FB. En el caso del CPC 664 este valor es &A67B y en el CPC 6128 es &A67B.

A través del comando *MEMORY dirección* podemos limitar el espacio reservado para programas BASIC. De esta manera disponemos del área que comienza con la dirección indicada en el comando MEMORY hasta &AB7F para almacenar nuestros programas máquina. En nuestro ejemplo hemos reservado el área &A000 hasta &AB7F para el programa máquina, mediante *MEMORY &9FFF*, almacenándolo a continuación a partir de &A000 con la ayuda de comandos POKE.

Quedarán sorprendidos al ver que se utilizan solamente un poco más que 5 K de RAM para rutinas del sistema.

¿Dónde están el interpretador y el sistema operativo, que nos hacen posible el programar en BASIC?
Supone bien:

Existe además otra memoria importante, el ROM (Read Only Memory= memoria de lectura o memoria de valor fijo). En el ROM se encuentran todos los datos y programas que nos permiten programar en BASIC sin dificultad. Puesto que la ROM es una memoria de valor fijo, se graba con datos y programas (en lenguaje máquina) y se instala en el ordenador antes de salir de fábrica. Lamentablemente no es posible acceder al contenido de esta memoria ROM desde el BASIC. Cuando hayamos confeccionado un programa máquina para tal fin objeto, resulta lo siguiente:

El CPC incorpora 2 ROMS de 16K (más 16 K de ROM para manejo del Diskette) cuyas direcciones se superponen con las del RAM. Esto es necesario dado que el procesador Z80 posee únicamente 16 líneas de direccionamiento, es decir que la dirección de un byte no puede superar los 16 bits previstos.

Con 16 bits queda perfectamente cubierto el espacio desde &0000 hasta &FFFF. Para poder leer del ROM debe indicarse en primer lugar a la CPU que se desea leer el ROM y a continuación pueden utilizarse las mismas direcciones que las del RAM. Los ROM ocupan las siguientes áreas:

1. ROM &0000 - &3FFF	sistema operativo
2. ROM &C000 - &FFFF	BASIC

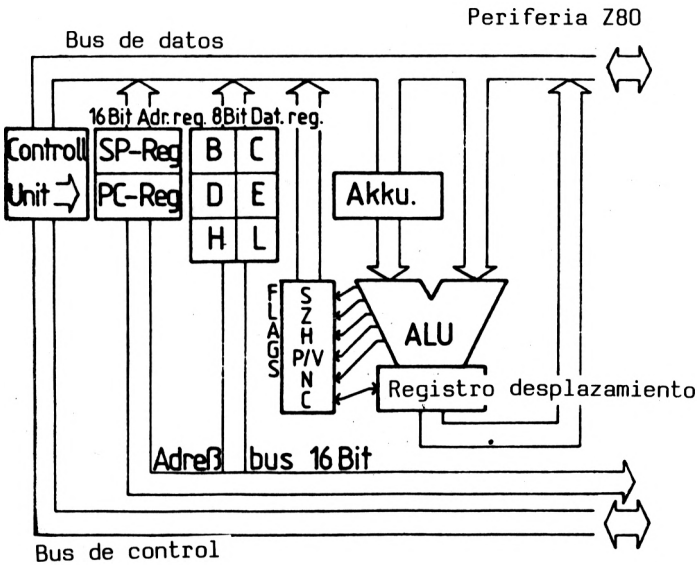
El sistema operativo contiene, como su nombre indica, las rutinas básicamente necesarias para que el ordenador trabaje. Su función es dirigir los periféricos, administrar los datos, mover datos, etc. En el área del ROM se encuentran asimismo las copias de las rutinas del sistema existentes en el RAM. Al conectar o hacer un Reset del ordenador, tales rutinas son copiadas del ROM al RAM. Además, en el ROM se encuentra la memoria de signos o caracteres (&3800-&3FFF), donde cada carácter del ordenador se representa en una matriz de bits (es decir, 0-ningún punto, 1-punto).

Los comandos BASIC programados por el usuario se ejecutan en el ROM de BASIC. La tabla de palabras de comando está, por ejemplo a partir de la dirección &E388. Esta es la forma de almacenamiento del CPC.

Naturalmente, nuestro ordenador posee además otros componentes, como por ejemplo el procesador Z80 o el Sound-Chip. En el próximo capítulo describiremos el procesador Z80. Si quieren ampliar la información presente sobre la arquitectura interna de su ordenador, les remitimos a los libros "CPC 464 INTERNO" y "CPC 664 & 6128 INTERNO".

CAPITULO II: EL PROCESADOR Z80

2.1 ESTRUCTURA DE LA CPU



El ordenador CPC posee una CPU (Unidad Central de Procesamiento) basada en el procesador Z80. Recordemos que podemos calificar a la CPU como "el cerebro" del ordenador. De este modo, es correcta la denominación de MPU (MPU: del inglés Micro Processing Unit es decir Microprocesador).

En el presente capítulo nos ocuparemos de la estructura y de la función de cada uno de los elementos que componen la CPU. El gráfico de esta página nos será de ayuda para comprender la vida interna de la Unidad Central. Observemos el diagrama de izquierda a derecha:

1. CU (CU:ingl. Control Unit = Unidad de Control).
   Todos los movimientos del ordenador son controlados y dirigidos por esta unidad.

2. Bus de Control.
   El Bus de Control es el "brazo largo" del CU. A través de él se dirigen y supervisan los elementos fuera de la CPU.

3. Indicador de la pila SP (SP:ingl. Stack Pointer).
   Con ayuda del SP pueden guardarse en el RAM datos y direcciones de retorno de los subprogramas.
   Puesto que en el SP se almacenan direcciones, es un registro de 16 bits.

4. Contador de Programa PC (PC: ingl. Program Counter)
   El PC indica la dirección de memoria en la que se halla cada comando que ha de ejecutarse.

5. Registros B a L
   La CPU posee varios registros, donde se almacenan los datos.

6. Flags (Flag: ingl. flag = bandera; en este caso significa indicador característico). Los flags sirven como indicadores de determinados incidentes que acontecen en la CPU durante las operaciones del ordenador. Los flags pueden estar activados (flecha arriba), o desactivados (flacha abajo).

7. Bus de direcciones (se encuentra fuera de la CPU)
   El Bus de direcciones realiza el enlace con otros MPU del ordenador. Indica el lugar de memoria en el ROM o en RAM, cuyo contenido ha de leerse o grabarse. El Bus de direcciones tiene una extensión de 16-bits, necesarios para direccionar el espacio de memoria de 64K.

8. Bus de Datos (se encuentra fuera de la CPU)
   Los buses de datos "trasladan" los datos que han de ser leídos o grabados. El bus de datos apunta así hacia la dirección de los datos. El bus de datos tiene una extensión de 8-bits.

9. Acumulador (lat. Akkumulator)
   El acumulador (ACU) es el registro más importante de la CPU. También se le puede calificar como el registro de cálculo.

10. ALU (ALU: ingl. Arithmetical Logical Unit - Unidad aritmética lógica, Unidad de cálculo).
    La ALU realiza diversas operaciones aritméticas y lógicas. Los flags se activan/desactivan en función del resultado de las operaciones.

11. Cursor
    El cursor dirige las rutinas de rotación y traslación.

Como ya hemos mencionado en el punto 5., la CPU contiene varios registros. Para comprender mejor su funcionamiento los hemos dividido en 5 grupos.

1. El Acumulador
2. Los Flags
3. Los "seis asociables" registros de 8 bits
4. Los "cuatro inseparables" registros de 16 bits
5. Registro Interrupt/Refresh

2:2 EL ACUMULADOR

El ACU, o bien el registro A, es el registro más importante del Z80. La mayoría de los comandos aritméticos y lógicos utilizan este registro. Cuando se ejecuta un comando de comparación, se efectúa fundamentalmente con el contenido del ACU. Al igual que todos los registros, excepto SP, PC, IX e IY, el registro A es un registro de 8-bits.

2.3 LOS FLAGS

El registro Flag o registro F, tiene una extensión de 8-bits (como A, B, C, D, E, H y L). Sin embargo, también tiene otras funciones. En el registro Flag se utilizan los distintos bits como indicadores para determinados sucesos originados durante las operaciones de la ALU (unidad de cálculo). Cada uno de los bits del registro F tiene el significado siguiente:

S	Z	H	P/V	N	C	- Denominación del Flag.
7	6 5	4 3	2	1	0	- Número del bit

C - Carry (acarreo)
N - Sustracción
P/V - Paridad/Desboradmiento
H - Media Transferencia
Z - Zero (cero)
S - Sign (signo)

Flag C (Bit 0)
Si en el transcurso de una suma o resta se produce un acarreo, se activa este bit, de lo contrario se desactiva.

Flag N y H (Bit 1 Bit 4)
Estos flags los utiliza el Z80 internamente. Para nuestros fines, carecen de significado.

Flag P/V (Bit 2)
Este flag tiene una doble función:
Se activa cuando tiene lugar un desbordamiento (V Ingl. overflow), de lo contrario se desactiva. Además indica la paridad (P) de un byte.

Flag Z (Bit 6)
Este flag se activa cuando el resultado de una resta es cero, de lo contrario se desactiva. Se activa asimismo este flag como resultado de una comparación, en caso de igualdad.

Flag S (Bit 7)
Este flag se activa cuando el resultado de una suma o resta es superior a 127. Como veremos más adelante, en la aritmética de la CPU, los bytes superiores a 127 representan valores negativos.

Los bits 3 y 5 del registro Flag no se utilizan.

2.4 LOS "SEIS ASOCIABLES" REGISTROS DE 8-BITS

Este grupo se compone de 6 registros de 8-bits, los registros B, C, D, E, H y L.
Estos registros tienen la facultad de formar pares de registros, con el fin de crear un registro de 16-bits. En los registros C, E y L se almacena el Low-Byte, en los registros B, D y H el High-Byte.

B/C (Byte Counter)
El registro B, o mejor dicho el par de registros BC, se utilizan frecuentemente como contadores, por ejemplo en los bucles.

H/L (High/Low)
El par de registros HL se utilizan en general para almacenar direcciones.

Es conveniente habituarse a denominar los registros de esta forma, ya que algunos comandos utilizan los registros siguiendo esta nomenclatura. En principio pueden utilizarse también naturalmente el registro L o E como contador.
Una particularidad del Z80 radica en que todos los registros antes citados se presentan nuevamente con la misma función. Este doble juego de registros está disponible, aunque sólo puede utilizarse un juego de registros cada vez.

2.5 LOS "CUATRO INSEPARABLES" REGISTROS DE 16 BITS

Este grupo está formado por 4 registros de 16 bits: SP, PC, IX e IY.
El registro SP es un registro fijo; es decir, no puede descomponerse en 2 registros de 8 bits. El puntero de pila (SP) indica cada una de las direcciones de la memoria, que contienen direcciones de retorno o datos de memoria intermedia. La dirección se refiere a una posición de memoria situada en un espacio del RAM, denominado Stack o Pila. La utilización de los Stacks para el almacenamiento de datos se lleva a cabo de la siguiente manera:

Al conectar el ordenador se coloca el SP en la dirección más elevada de la pila ($C000). Si hay que colocar un Byte en la pila, entonces el SP se reduce automáticamente en 1, almacenándose este Byte en la dirección indicada por el SP. Así pues, el SP apunta siempre al último elemento de la pila. Al "recoger de la pila", el proceso es inverso. En primer lugar, se lee el Byte en la dirección indicada por el SP, a continuación se incrementa el SP en 1. De este modo, es posible encadenar llamadas a los subprogramas.

El PC es un registro especial. Desde un programa no puede grabarse ni modificarse. El PC se administra internamente e indica siempre la dirección del comando actual.

Los registros IX/IY se utilizan fundamentalmente para almacenar

direcciones o direcciones relativas. Al igual que todos los registros descritos en 2.5, estos 2 registros pertenecen también a los registros de 16-bits. En éstos no es posible acceder por separado al High-Byte y al Low-Byte (como ocurre con los registros BC, DE y HL). El uso del Registro índice, se asemeja al par de registros HL. Veremos la diferencia cuando hablemos del direccionamiento indexado.

2.6 REGISTROS INTERRUPT/REFRESH

Estos 2 registros corresponden a la CU.

I - o bien registro Interrupt (ingl. interrupt: interrupción).
Si tiene lugar una interrupción; es decir, una interrupción del programa, este registro de 8 bits contiene la parte superior de la dirección a la cual ha de realizarse la bifurcación. La parte inferior viene suministrada por el elemento del ordenador que ha ocasionado la interrupción.

R - o bien registro Refresh (ingl. refresh: refresco)
Este registro es utilizado por el Hardware como contador, con el fin de refrescar, a intervalos regulares, el contenido de la memoria dinámica. Con ello se impide que se pierdan las informaciones almacenadas. La pérdida de datos se impide mediante la continua recarga del mismo contenido de la memoria.

La CPU ejecuta el comando de la siguiente manera:

En primer lugar lee el Byte en la dirección indicada por el PC, aumentando el apuntador del mismo en 1; es decir, señalando ahora el byte siguiente. El Byte leído se interpreta como comando. A continuación, se procede a la lectura eventual de los datos

CAPITULO III: LA SENTENCIA DE COMANDOS DEL Z80

3.1 INTRODUCCION: ENTRADA DE PROGRAMAS EN LENGUAJE MAQUINA

Para poder comenzar a probar comandos con el Z80, deberemos en primer lugar, saber de qué manera se introduce y almacena un programa en lenguaje máquina, partiendo del BASIC.De la misma manera que a cada número de línea le corresponde un comando BASIC; en lenguaje máquina, a cada comando le corresponde una dirección.

BASIC		LENGUAJE MAQUINA		
N. Línea	Comando	Dirección	Comando	Código
9	HL=HL+1	&A009	INC HL	&23
10	RETURN	&A00A	RET	&C9

- En BASIC, a cada número de línea le corresponde un comando.

- En CODIGO MAQUINA, a cada comando le corresponde una dirección.

Así pues, un programa máquina es una sucesión de códigos de comando, situados en la memoria en direcciones sucesivas.

Mediante el BASIC, y con ayuda de los comandos *POKE*, podemos almacenar los códigos en las correspondientes direcciones. De este modo, los programas son llamados con el comando *CALL Dirección*, donde la dirección se refiere a la dirección inicial del programa máquina, que corresponde al espacio de memoria que contiene el primer código máquina. Para evitar equivocaciones, debemos en primer lugar reservar el espacio necesario para nuestro programa, con el comando MEMORY. A través de *MEMORY &9FFF* reservamos el ámbito de direcciones de &A000 hasta &AB7F, con un tamaño de &B80 Bytes (aproximadamente 3K), suficiente para nuestro programa en lenguaje máquina.

Un ejemplo de programa BASIC para cargar programas en lenguaje máquina es el siguiente:

```
10 MEMORY &9FFF
20 FOR I=Dirección inicial TO Dirección Final
30 READ A
40 POKE I,A
50 NEXT I
60 DATA ......
70 DATA ......
    .
    .
    .
```

Las líneas DATA contienen los códigos que componen el programa máquina. La dirección final (V=Variable; en lo sucesivo utilizaremos siempre esta abreviatura detrás de aquellas expresiones que representen variables), será naturalmente superior a &9FFF, y la dirección inicial (V), será menor que &AB80.
La llamada del programa se efectúa mediante *CALL Dirección inicial*.

En general utilizaremos &A000 como dirección inicial. La dirección final (V) resulta de sumar la longitud total del programa a la dirección inicial, menos 1.
La longitud del programa corresponde al número de entradas en las líneas DATA.

Para la entrada de programas pequeños es práctico el siguiente programa BASIC:

```
10 CLS
20 MEMORY &9FFF
30 LOCATE 10,10: INPUT "dirección inicial"; ADR
40 IF ADR < &A000 OR ADR > &ABFF THEN 30
50 PRINT
60 PRINT HEX$(ADR,4);":";
70 INPUT VALOR$
```

```
80 IF VALOR$="" THEN END
90 VALOR=VAL("&"+VALOR$)
100 ADR=ADR+1
110 IF ADR > &AB7F THEN PRINT "Memoria llena": END
120 GOTO 60
```

Introduzca los códigos hexadecimales, y el programa resolverá el *POKE*. No será necesario entrar el signo hexadecimal (&) para la dirección inicial. Si quiere terminar el programa , pulse ENTER.

Ahora que hemos aprendido a entrar programas en lenguaje máquina, veamos los comandos del Z80.

Observación: En las aclaraciones de comandos trabajaremos a menudo con analogías de los comandos en BASIC. Para ello, nos imaginaremos un registro en BASIC como una variable con el mismo nombre (por ejemplo el registro HL del lenguaje máquina, corresponde a la variable HL en BASIC).

Los comandos del Z80 se subdividen en 5 grupos:

1. Transferencia de datos
2. Tratamiento de datos y comparaciones
3. Saltos
4. Comandos de control
5. Entrada y salida.

3.2 TRANSFERENCIA DE DATOS

Estos comandos se utilizan para transferir o enviar datos. La transferencia puede realizarse de:

a) Registro a registro.

Ello supone una designación en BASIC; como por ejemplo, A=B, o SP=HL. El comando en lenguaje máquina tiene el formato: LD A,B

(LD ingl. load: carga).

b) Registro a posición de memoria

En la transferencia de datos del registro a la posición de memoria, el comando en BASIC *POKE Dirección de memoria, Variable*, por ejemplo *POKE &A000,HL* corresponde al comando máquina LD (&A000),HL.

c) Posición de memoria a registro.

La transferencia de datos desde la memoria a un registro, por ejemplo LD H,(&A005), corresponde al comando BASIC *H=PEEK (&A005)*.

3.3 TRATAMIENTO DE DATOS Y COMPARACIONES

Los comandos para el tratamiento de datos podemos subdividirlos nuevamente en 5 grupos:

- Operaciones aritméticas (por ejemplo ADición, SUBstracción)
- Operaciones lógicas (por ejemplo AND, OR)
- Comandos contadores (INCrementar, DECrementar)
- Manipulación de Bits (SET, RESet)
- Intercambios y desplazamientos de Bits (Rotate = rotar, Shift = desplazar).

Durante la ejecución de estos comandos se modifican los contenidos de los registros o de la memoria (en RAM). Muchos de estos comandos son similares a los correspondientes comandos BASIC:

Ensamblador	BASIC
SUB A,B (SUBstracción)	A=A-B
ADD HL,BC (ADDición)	HL=HL+BC
AND C	A=A AND C
OR &HL	A=A OR PEEK(HL)

Las comparaciones se realizan, o bien de bits individuales en el registro, o en las posiciones de memoria (comando BIT), o bien se comparan los contenidos de los registros o de la memoria con el ACU (comando CP=compare). Según los resultados de estas comparaciones, la ALU activará o desactivará cada uno de los flags en el registro F.

3.4 BIFURCACIONES

Con ayuda de estos comandos es posible incluir bifurcaciones en programas máquina.
Se distinguen tres tipos de bifurcación:

- bifurcación directa a una dirección de 16-bits (JP=Jump)
- bifurcación relativa a la dirección actual (JR=Jump relativ)
- bifurcaciones a subprogramas (CALL con retorno por RET)

Una bifurcación es condicional cuando la decisión de si debe efectuarse la misma depende del estado de un Flag.
Un salto condicional, es decir, aquel que depende del estado de un Flag, sería por ejemplo JR NZ,$-6>A000.

Analogías:

Ensamblador	BASIC
JP	GOTO
CALL	GOSUB
RET	RETURN
JR	-------

3.5 COMANDOS DE CONTROL

Con estos comandos puede interrumpirse un programa. El control con interrupciones también es posible con estos comandos.

3.6 COMANDOS DE ENTRADA/SALIDA (Input/Output)

Los comandos I/O sirven para la comunicación con los periféricos de entrada y salida. Dependiendo de la dirección del puerto I/O seleccionado se realizan diferentes tareas mediante estos comandos. Los circuitos integrados que se seleccionan frecuentemente a través de comandos I/O son p.ej.:

PPI (Programmable Peripheral Interface)
PSG (Programmable Sound Generator)
CRTC (Cathode Ray Tube Controller) y así los dispositivos periféricos: teclado, altavoz, monitor, impresora y dispositivo de cassette.

pertenecientes al comando (el PC se incrementa nuevamente). Acto seguido se ejecuta el comando y el proceso comienza nuevamente.

Ahora que ya conocemos la CPU del Z80, pasaremos a ocuparnos de los verdaderos comandos de la máquina.

CAPITULO IV: LOS COMANDOS

4.1 COMANDOS DE TRANSFERENCIA DE 8-BITS

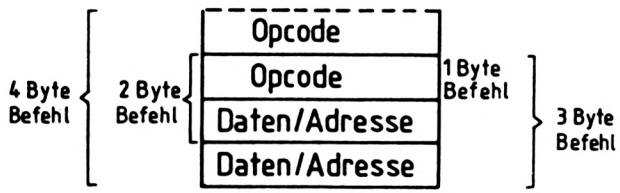
Todos los comandos de transferencia de este tipo se representan por el comando de carga LD (load).
Un comando de carga, presenta el formato siguiente:

LD Destino,Origen

En los comandos de transferencia de 8-bits, se mueven 8-bits desde el origen al destino. Como ejemplo de estos comandos le presentaremos los tipos de direccionamiento del Z80.

Cada comando máquina se compone fundamentalmente de un código de (Opcode), que puede continuarse con un campo de operando o de dirección. El Opcode determina qué operación debe ser efectuada. Frecuentemente, un Opcode contiene Bits que son utilizados como indicadores de un registro. El realidad, estos Bits no pertenecen al Opcode. Para simplificar el ejemplo, contaremos estos indicadores eventuales como pertenecientes al Opcode. En algunos comandos, el Opcode viene seguido de Bytes de datos y de dirección. Además existen asimismo comandos cuyo Opcode tiene 2 bytes. De este modo, un comando puede llegar a tener una longitud de 1 a 4 bytes.

DIBUJO PAGINA 49



Befehl = Comando
Daten = Datos
Adresse = Dirección

Para interpretar los datos o direcciones que siguen a un comando, es necesario conocer los diferentes tipos de direccionamiento.

Direccionamiento inmediato (Inmediatly Adressing)

(Ingl. inmediatly: inmediato). Este es el direccionamiento más sencillo.

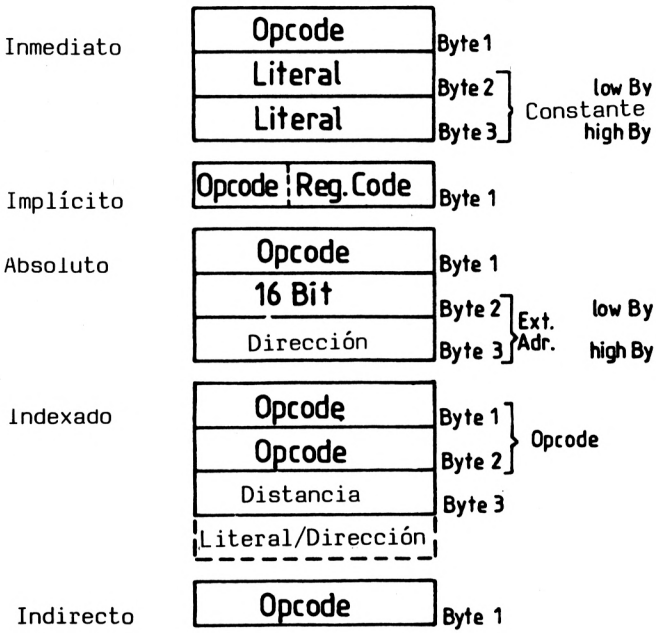
Formato:

LD registro,datos

En este comando "registro" representa un registro (A, B, C, D, E, H o L) y "datos" corresponde a un número de 8-bits; es decir, el registro dado se carga con la constante "inmediata" que le sigue. Una constante de este tipo también se denomina Literal. El direccionamiento Inmediato se representa en el esquema 3. Al Opcode de 8-bits le sigue un literal (constante) de 8 - ó 16-bits.

Ejemplo:

```
LD C,&7F        BASIC: C=&7F
                (significa: cargar registro C con &7F)
```



Dib.3 4.1

Direccionamiento implícito y de registro (ingl.: Implied Register Adressing)

Los comandos que trabajan exclusivamente con registros utilizan el direccionamiento implícito (ingl.: Implied: implícito).

Formato:

LD Reg,Reg

Traslada el contenido del registro origen al registro destino. Pueden utilizarse los registros A, B, C, D, E, H o L.

La denominación de este tipo de direccionamiento resulta del hecho de que el operando (es decir ambos registros afectados) no se indica expresamente. En realidad el Opcode de la orden contiene los registros afectados, (los implica).
El código de operación de este comando en formato binario es:

01DDDOOO

Cada una de las letras D y O corresponden a un bit. Las tres "D" representan el registro destino y las tres "O", el registro origen. El código para los registros es:

A-111 E-011
B-000 H-100
C-001 L-101
D-101

Ejemplo: LD B,C = 01 000 001 = &41
LD B C

Con ello es posible representar los comandos direccionados implícitamente como un código de operación de 1 byte. Por este motivo, es muy reducida la duración de su ejecución.

Ejemplo:

LD A,B BASIC: A=B

Significado: transferencia del contenido de B a A, es decir carga el registro A con el contenido de B.

Zilog Inc. (el inventor del Z80) califica el anterior tipo de direccionamiento como direccionamiento de registro y discrepa en la denominación del direccionamiento implícito. Según esto,

solamente serían direccionados implícitamente los comandos LD I,A; LD R,A; LD A,R y LD A,I. Nosotros no haremos no obstante tal diferencia, y utilizaremos ambos conceptos.: direccionamiento implícito y direccionamiento de registro.

Direccionamiento absoluto o "externo" (External Adressing)

(Ingl.:external=externo)
Como direccionamiento absoluto se describe el proceso de extraer datos de la memoria, o de almacenarlos en ella. Mediante este procedimiento, la dirección de 16-bits de la posición de memoria se indica por completo (la dirección "absoluta").

Formato:

LD (Dir),reg o bien LD Reg,(Dir)
(Dir: es la dirección de la posición de memoria)

El registro dado se carga con el contenido de la posición de memoria indicada y viceversa. Del esquema 3, puede deducirse que la dirección sigue al código de operación. El direccionamiento absoluto requiere 3 bytes, por lo que la ejecución de los comandos de esta clase resulta relativamente larga.

Ejemplo:

```
LD A,(&BF93)          BASIC: A=PEEK(&BF93)
LD (&A001),A          BASIC: POKE &A001,A
```

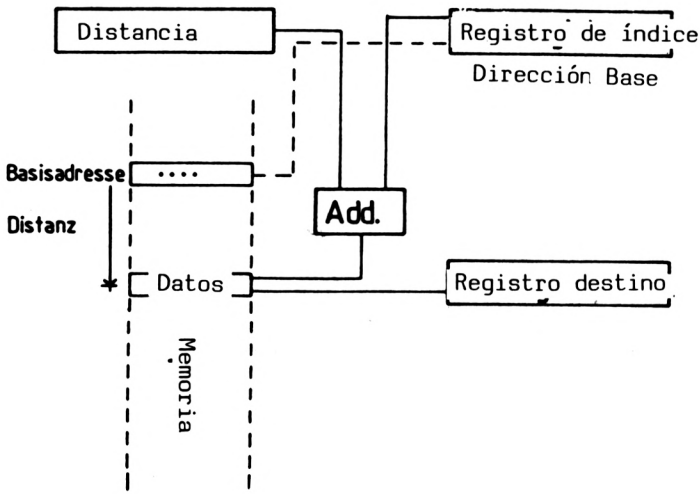
Direccionamiento Indexado (Indexed Adressing)

(Ingl.:index:índice)
En el direccionamiento indexado no queda indicada la dirección absoluta de la posición de memoria, sino que se calcula a partir del contenido de un registro índice y del desplazamiento indicado.

Formato:

```
LD Reg,(IX+des)      o bien      LD (IX+des),Reg
LD Reg,(IY+des)      o bien      LD (IY+des),Reg
(des=desplazamiento, IX ó IY son registros de índice)
```

Carga del registro definido con la posición de memoria que posee la siguiente dirección (y viceversa): La dirección resulta del contenido del registro índice y del desplazamiento indicados.



Dib. 4

Direccionamiento indexado

Los comandos indexados poseen un código de operación de 2-bytes, al que sigue el desplazamiento indicado. El primer byte del código de operación es:

&DD - Si se refiere al registro IX
&FD - Si se refiere al registro IY

Los bytes restantes del código son idénticos, independientemente de si se refiere al registro IX o al registro IY. La técnica del direccionamiento indexado se utiliza para acceder consecutivamente a los elementos de un bloque de datos. El desplazamiento puede ser positivo o negativo; es decir, el byte del desplazamiento se indica como complemento de dos.

Ejemplo:

```
LD E,(IX+&32)          BASIC: E=PEEK (IX+&32)
LD (IY+&12),A          BASIC: POKE IY+&12,A
```

Direccionamiento Indirecto (Registro indirecto)

Este tipo de direccionamiento se asemeja al direccionamiento indexado y al absoluto, la única diferencia es que en este caso la posición de memoria se direcciona mediante el contenido de uno de los pares de registros HL, BC o DE.

Formato:

```
LD reg,(prs)        o bien    LD (prs),reg
(prs es uno de los pares de registros HL, BC o DE)
```

Carga el registro con el contenido de la posición de memoria direccionada mediante el par de registros HL, BC o DE.
Esta técnica de direccionamiento presenta, frente al direccionamiento indexado y al absoluto, la ventaja de que sólo necesita comandos de 1 byte de longitud; es decir, que el registro indicado y el par de registros HL, BC o DE están contenidos en el

código de operación, no necesitando indicación expresa. De este modo el comando es más rápido, ofreciendo asimismo la posibilidad de acceder a las 64 K.

Ejemplo:

```
LD B,(HL)          BASIC: B=PEEK (HL)
LD (BC),A          BASIC: POKE BC,A
```

Con esto hemos estudiado ya todos los tipos de direccionamiento que aparecen en los comandos de transferencia de 8-bits. En el curso de este capítulo aún daremos a conocer algunos otros tipos de direccionamiento, así como también aplicaremos los ya conocidos a otros comandos. En el anexo encontrará unas tablas, donde verá todos los comandos, clasificados por funciones (transferencia, bifurcación, etc.), y por tipos de direccionamiento. En estas tablas podrá consultar los códigos de operación de todos los comandos. A continuación queremos unir los comandos de carga de 8-bits nuevamente. En el anexo encontrarán una tabla de las abreviaciones utilizadas.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	OP-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments r, r'	Reg.
LD r, r'	r ← r'	•	•	•	•	•	•	01 r r'	1	1	4	r, r'	Reg.
LD r, n	r ← n	•	•	•	•	•	•	00 r 110	2	2	7	000	B
								← n →				001	C
LD r, (HL)	r ← (HL)	•	•	•	•	•	•	01 r 110	1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	•	•	•	•	11 011 101	3	5	19	011	E
								01 r 110				100	H
								← d →				101	L
LD r, (IY+d)	r ← (IY+d)	•	•	•	•	•	•	11 111 101	3	5	19	111	A
								01 r 110					
								← d →					
LD (HL), r	(HL) ← r	•	•	•	•	•	•	01 110 r	1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	•	•	•	•	11 011 101	3	5	19		
								01 110 r					
								← d →					
LD (IY+d), r	(IY+d) ← r	•	•	•	•	•	•	11 111 101	3	5	19		
								01 110 r					
								← d →					
LD (HL), n	(HL) ← n	•	•	•	•	•	•	00 110 110	2	3	10		
								← n →					
LD (IX+d), n	(IX+d) ← n	•	•	•	•	•	•	11 011 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD (IY+d), n	(IY+d) ← n	•	•	•	•	•	•	11 111 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD A, (BC)	A ← (BC)	•	•	•	•	•	•	00 001 010	1	2	7		
LD A, (DE)	A ← (DE)	•	•	•	•	•	•	00 011 010	1	2	7		
LD A, (nn)	A ← (nn)	•	•	•	•	•	•	00 111 010	3	4	13		
								← n →					
								← n →					
LD (BC), A	(BC) ← A	•	•	•	•	•	•	00 000 010	1	2	7		
LD (DE), A	(DE) ← A	•	•	•	•	•	•	00 010 010	1	2	7		
LD (nn), A	(nn) ← A	•	•	•	•	•	•	00 110 010	3	4	13		
								← n →					
								← n →					
LD A, I	A ← I	•	‡	IFF	‡	0	0	11 101 101	2	2	9		
								01 010 111					
LD A, R	A ← R	•	‡	IFF	‡	0	0	11 101 101	2	2	9		
								01 011 111					
LD I, A	I ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 000 111					
LD R, A	R ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 001 111					

Notes: r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

‡ = flag is affected according to the result of the operation.

Ejercicio:

Acaban de aprender los comandos de carga de 8 bits. Como aplicación modificaremos algunas posiciones de memoria del sistema operativo con la ayuda de estos comandos.

Veamos en primer lugar la posición de memoria &B289 (el 664: &B72A / 6128: &B72A). En esta dirección se almacena la primera columna izquierda de la ventana actual de la pantalla, la cual puede ser confeccionada desde el BASIC con el comando *WINDOW 10,80,1,25*. Queremos inicializar el ancho de la pantalla en su porción izquierda en 10. Desde el BASIC lo logramos con *WINDOW 10,80,1,25*. También existe la posibilidad de grabar directamente el valor deseado en la posición de memoria &B289 (664: &B72A / 6128: &B72A) mediante el comando *POKE*.

Probemos entonces con *POKE &B289,10* (664: &B72A,10 / 6128: &B72A) y la columna de pantalla de la izquierda es la décima.
Intenten confeccionar el comando BASIC

POKE &B289,10 (664: &B72A,10 / 6128: &B72A,10).

en lenguaje máquina. Finalicen su programa con el comando RET (código &C9).

Discusión de la solución

Analicemos en primer lugar el ejercicio:
queremos cargar una posición de memoria (dirección &B289 (664: &B72A/ 6128: &B72A)) con un valor de un byte, es decir un número entre 0 y 255.
Dado que el valor es un número de 1 byte, utilizamos un comando de carga de 8 bits.
Ahora existen diferentes soluciones, que dependen de la modalidad de direccionamiento.
Para cargar una posición de memoria con un byte se debe determinar naturalmente previamente la dirección de la posición de memoria deseada. La posibilidad más simple es el direccionamiento absoluto

en el cual se indica la dirección completa:

```
LD (&B289),A  (Direccionamiento absoluto) para CPC 464
LD (&B72A),A  (Direccionamiento absoluto) para CPC 664
LD (&B72A),A  (Direccionamiento absoluto) para CPC 6128
```

Es decir: carga la posición de memoria &B289 (664: &B72A/ 6128: &B72A) con el valor contenido en el ACU. Esto significa que previamente el ACU debe cargarse con el valor deseado.

```
LD A,10  (Direccionamiento inmediato)
```

Ambos comandos ejecutados consecutivamente llevan al resultado deseado.

```
LD A,10
LD (&B289),A  (664: LD (&B72A),A /
               6128: LD (&B72A),A)
RET
```

El comando RET debe ejecutarse al final de cada programa máquina, que provoca un retorno al BASIC.

Hasta ahora hemos presentado el programa máquina únicamente en ensamblador, es decir en forma simbólica. Para que el procesador pueda ejecutar nuestros comandos, deben traducirse los mismos previamente a números. De las listas de comando presentadas o bien de las tablas contenidas en el final de este libro obtenemos para el comando de carga de 8 bits, LD A,10 con direccionamiento inmediato el código &3E,10, donde el segundo código representa el valor deseado. Para el comando con direccionamiento absoluto LD (&B289),A (664: LD (&B72A),A / 6128: LD (&B72A),A) obtenemos &32,&89,&B2 (664: &32,&2A,&B7 / 6128: &32,&2A,&B7). También aquí los 2 últimos códigos representan la dirección indicada. Tengan en

cuenta que en primer lugar se indica el Low Byte y a continuación el High Byte.
El código para RET es, como ya hemos indicado, &C9.

De estos códigos se compone la línea DATA del cargador BASIC en el capítulo 3:

```
DATA &3E,10,&32,&89,&B2,&C9 para CPC 464
DATA &3E,10,&32,&2A,&B7,&C9 para CPC 664
DATA &3E,10,&32,&2A,&B7,&C9 para CPC 6128
```

Nuestro programa tiene una longitud de 6 bytes, es decir el bucle FOR-NEXT tiene que ejecutarse desde &A000 hasta &A005. Después de ejecutar con *RUN* el cargador BASIC, nuestro primer programa máquina se encuentra en la memoria. Puede arrancar su primer programa introduciendo *CALL &A000*. Si ha realizado todo correctamente, el cursor se debería desplazar inmediatamente a la décima columna de la pantalla. Si se quiere alterar el valor de la columna izquierda, solamente debe modificarse el segundo número de la línea DATA (el "10") grabando el programa nuevamente en la memoria con *RUN* y llamarlo mediante *CALL &A000*.

Prueben asimismo las siguientes direcciones:

```
  464   664  6128

&B288 &B729 &B729   línea superior
&B28A &B72B &B72B   columna derecha
&B28B &B72C &B72C   línea inferior
```

4.2 COMANDOS DE TRANSFERENCIA DE 16 BITS

Los comandos de carga de 16 bits presentan el formato general:

LD Destino,Origen

Sin embargo, aquí se transfieren 16 bits. Mediante estos comandos se activan los pares de registros BC, DE, HL, SP, IX e IY.

Direccionamiento inmediato

Como aquí solamente se cargan registros de 16 bits, la constante que sigue al código de operación deberá tener una longitud de 16 bits. De este modo, los 2 bytes siguientes al código de operación, contienen el Low-byte y el High-byte de la constante (¡en este orden!). En contraposición al direccionamiento inmediato con constantes de 1 byte, esta técnica recibe el nombre de direccionamiento inmediato extendido (ingl. Inmediatly extended).

Formato:

LD x,datos-16
(x: uno de los registros de 16 bits SP,BC,DE,HL,IX,IY)
(datos-16: constante de 16 bits)

Mediante este comando, el registro x se carga con la constante datos-16.

Ejemplo:

```
LD HL,&C000          BASIC: HL=&C000
```

Direccionamiento implícito

Para los comandos de carga de 16 bits sólo existen 3 comandos de este tipo, y todos ellos afectan al registro SP:

LD SP,HL LD SP,IX LD SP,IY

Estos comandos significan:
Carga del apuntador de la pila con el contenido del registro HL, IX ó IY.

Analogía en BASIC:
SP=HL SP=IX SP=IY

Direccionamiento Absoluto

El direccionamiento absoluto en los comandos de 16 bits debemos explicarlo con mayor detalle:

Formato:

LD prs,(dir) o bien LD (dir),prs

(prs: par de registros BL, DE, HL, SP, IX ó IY)

Dado que dir señala una dirección, es decir direcciona un único byte y puesto que sin embargo prs representa un registro de 16 bits, se utiliza la convención siguiente: se carga en primer lugar en el registro el Low-byte de la dirección Dir, y a continuación el High-byte de la dirección dir+1.

Por ejemplo: LD HL,(&AB80) significa:
Registro L = Low-byte de la dirección &AB80
Registro H = High-byte de la dirección &AB81

En el comando inverso de la forma LD (dir),x; el low-byte se almacena en la correspondiente dirección (dir) y el High-byte en la dirección Dir+1.

Por ejemplo: LD (&CB00),IX
Dirección &CB00 = Low-byte de IX
Dirección &CB01 = High-byte de IX

Un comando de este tipo corresponde por lo tanto a 2 comandos de carga de 8-bits.

Comando de 16 bits		Comando de 8-bits
LD BC,(&FC05)	corresponde	LD C,(&FC05) (Low-byte)
		LD B,(&FC06) (High-byte)

Como ya sabemos, un número de 16 bits puede representarse a partir del Low-byte y del High-byte, de la siguiente forma:

```
Número=256*(High-Byte)+(Low-Byte)
```

Por esto, la equivalencia en BASIC:

```
Lenguaje máquina          BASIC
LD DE,(&4000)             DE=256*PEEK(&4001)+PEEK(&4000)
```

Utilizando el sistema hexadecimal, puede representarse:

```
DE=VAL("&"+HEX$(PEEK(&4001))+HEX$(PEEK(&4000)))
```

Para codificar en BASIC el comando inverso; por ejemplo LD (&6800),IY se necesitan 2 comandos:

```
POKE &6800, IY-INT (IY/256)*256     (Low-byte)
POKE &6801,    INT(IY/256)          (High-byte)
```

En caso de que estas analogías no le sean suficientemente claras, repase el capítulo relativo a la representación de números. Coloque entonces números para cada DE e IY, y luego ¡realice el cálculo real!.
Las listas de comandos de este libro han sido proporcionadas por la empresa Zilog.
Dan una vista general sobre todos los comandos disponibles. Para cada grupo de comandos imprimimos la tabla correspondiente. Utilicen las tablas para buscar los Op-Code necesarios para los ejercicios.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	•	•	•	•	00 dd0 001 ← n → ← n →	3	3	10	dd Pair 00 BC 01 DE
LD IX, nn	IX ← nn	•	•	•	•	•	•	11 011 101 00 100 001 ← n → ← n →	4	4	14	10 HL 11 SP
LD IY, nn	IY ← nn	•	•	•	•	•	•	11 111 101 00 100 001 ← n → ← n →	4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	•	•	•	•	00 101 010 ← n → ← n →	3	5	16	
LD dd, (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	•	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	•	•	•	•	11 011 101 00 101 010 ← n → ← n →	4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	•	•	•	•	11 111 101 00 101 010 ← n → ← n →	4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	•	•	•	•	00 100 010 ← n → ← n →	3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	•	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	4	6	20	
LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	•	•	•	•	11 011 101 00 100 010 ← n → ← n →	4	6	20	
LD (nn), IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	•	•	•	•	11 111 101 00 100 010 ← n → ← n →	4	6	20	
LD SP, HL	SP ← HL	•	•	•	•	•	•	11 111 001	1	1	6	
LD SP, IX	SP ← IX	•	•	•	•	•	•	11 011 101 11 111 001	2	2	10	
LD SP, IY	SP ← IY	•	•	•	•	•	•	11 111 101 11 111 001	2	2	10	qq Pair
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H	•	•	•	•	•	•	11 qq0 101	1	3	11	00 BC 01 DE
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H	•	•	•	•	•	•	11 011 101 11 100 101	2	4	15	10 HL 11 AF
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H	•	•	•	•	•	•	11 111 101 11 100 101	2	4	15	
POP qq	qq_H ← (SP+1) qq_L ← (SP)	•	•	•	•	•	•	11 qq0 001	1	3	10	
POP IX	IX_H ← (SP+1) IX_L ← (SP)	•	•	•	•	•	•	11 011 101 11 100 001	2	4	14	
POP IY	IY_H ← (SP+1) IY_L ← (SP)	•	•	•	•	•	•	11 111 101 11 100 001	2	4	14	

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
E.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

Ejercicio:

Antes de continuar con la explicación de comandos queremos aplicar los aprendidos hasta ahora. Como saben, la memoria de pantalla del CPC comienza en la dirección &C000. En esta área, cada 8 bits (1 byte) corresponden a 8 puntos consecutivos (en MODE 2). La dirección &C000 está asignada a los primeros 8 puntos, comenzando en la esquina superior izquierda de la pantalla. Los 8 puntos inferiores (1 byte) están almacenados en dirección &C800, los siguientes de la línea inferior en dirección &D000 etc. (en intervalos de &800). Introduzca:

```
10 POKE &C000,&FF
20 POKE &C800,&FF
30 POKE &D000,&FF
40 POKE &D800,&FF
50 POKE &E000,&FF
60 POKE &E800,&FF
70 POKE &F000,&FF
80 POKE &F800,&FF
MODE 2
RUN
```

Como pueden observar, el cuadrado superior izquierdo se ha rellenado con el color actualmente definido.

(&FF=&X11111111=8 puntos activados)

Ahora debe traducir este programa a lenguaje máquina mediante los comandos recién estudiados. Finalice su programa máquina con RET (&C9).

Discusión de la solución para
el programa máquina a confeccionar.

En principio necesitamos un comando que cargue una posición de memoria con un valor (=POKE). Para ello podemos utilizar los

comandos con direccionamiento indirecto, indexado y absoluto (vean definición). Para traducir exactamente nuestro ejemplo BASIC, elegimos el direccionamiento absoluto, es decir indicamos la dirección completa, como en el programa BASIC. Naturalmente también es posible almacenar la dirección en un registro utilizando a continuación el direccionamiento indirecto o indexado.

Ejemplo:

```
          BASIC: HL=&C000: POKE HL,&FF
Lenguaje máquina: LD HL,&C000     LD (HL),&FF
```

Dado que con los comandos de 16 bits se almacenan cada vez 2 posiciones consecutivas de memoria, utilizamos el comando de 8 bits.

```
LD (dir),A
```

Antes de la ejecución de este comando debemos almacenar el valor &FF en el ACU, utilizando para ello el direccionamiento inmediato:

```
LD A,&FF
```

Nuestro programa tendrá entonces el siguiente aspecto:

```
LD A,&FF
LD (&C000),A
LD (&C800),A
LD (&D000),A
LD (&D800),A
LD (&E000),A
LD (&E800),A
LD (&F000),A
LD (&F800),A
RET
```

Ahora buscamos los códigos para los correspondientes comandos:

```
LD A,datos:   &3E,co
LD (dir),A: &32,dl,dh   :Low, High
RET       : &C9
```

Así se convierten las líneas DATA de nuestro cargador de BASIC del capítulo 3.1 de la siguiente forma:

```
10 MEMORY &9FFF
20 FOR i=&A000 TO &A01A
30 READ a
40 POKE i,a
50 NEXT i
60 END

65 DATA &3E,&FF,&32,&00,&C0,&32,&00,&C8
70 DATA &32,&00,&D0,&32,&00,&D8,&32,&00,&E8
80 DATA &32,&00,&E8,&32,&00,&F0,&32,&00,&F8
90 DATA &C9
```

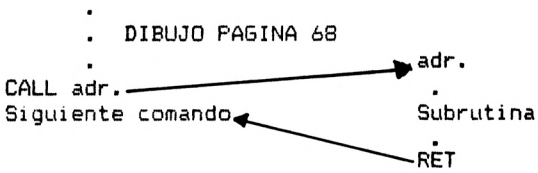
Queremos almacenar el programa en dirección &A000 (=dirección inicial (V)). El programa tiene una longitud de 27 bytes. La dirección final será entonces &A000 + 27 - 1 = &A01A. De aquí obtenemos la línea 20:

```
20 FOR i=&A000 TO &A01A
```

Después de almacenar a través de RUN el programa máquina, será "pokeado" automáticamente. Podemos entrar ahora *MODE 2* y *CALL &A000* comenzando la ejecución del programa. Como podrán observar, el color puede verse en el campo superior izquierdo de la pantalla. El programa pueden asimismo cargarlo directamente. Para ello, comiencen la carga directa con la dirección inicial &A000, seguido por los códigos correspondientes (p.ej. &3E, &FF, etc.). Este ha sido nuestro primer programa máquina. Ustedes pueden ahora modificarlo o mejorarlo con los nuevos comandos que se expliquen.

4.3 COMANDOS DEL STACK POINTER (PILA)

Para comprender mejor el modo de funcionamiento de los apuntadores, es necesario saber qué ocurre en el interior del Z80 cuando se bifurca a un subprograma. El comando Ensamblador necesario para ello es: CALL dirección. El problema básico consiste en que la CPU ha de "memorizar" la dirección de los comandos siguientes a la llamada, ya que cuando retorne del subprograma (RET), debe continuar en este punto la ejecución del programa principal.



Dib. 5 Llamada Subrutina

Dado que los registros son importantes para otras operaciones importantes, las direcciones de retorno han de memorizarse fuera de la CPU, es decir, en la RAM. Siguiendo este procedimiento, sin embargo, sólo podría memorizarse una única dirección de retorno. Esto significa que no sería posible un anidamiento de subprogramas. Por este motivo, se ha reservado un espacio de la RAM para esta operación, que se denomina STACK o PILA. Podemos imaginarnos esta pila, como un apilamiento de platos.

Una dirección de retorno se registrará siendo anotada en un plato. El plato así "direccionado", se coloca encima de la pila de platos. De este modo pueden tener lugar muchas llamadas a

subprogramas, aumentando simplemente el apilamiento. Cuando se realiza un retorno, se quita el plato superior de la pila y se bifurca en la dirección que lleva escrita. De este modo se suceden, uno a continuación del otro, los saltos hacia atrás, hasta que la pila de platos queda deshecha; es decir hasta que se llega nuevamente al programa principal. Es importante remarcar que el último plato colocado en la pila será siempre el primero que se extraiga (de lo contrario, toda la pila se vendría abajo).

Puesto que en el ordenador no son platos los que se apilan, deberá utilizarse un registro del Z80 como "medidor de la cima" de la pila. Este registro guarda siempre en memoria la dirección de la cima de la pila. Se denomina Stack Pointer (SP). De todos modos, nuestra pila en el ordenador "cuelga" del techo; es decir, el primer plato es colocado en la dirección más alta de la pila y el último en la dirección más baja. El Stack Pointer (la pila) se encuentra en la dirección &BFFF en el CPC.
Con todo lo expuesto, la ejecución de comandos CALL sería:

Sección del Stack Pointer:

```
Posición inicial:                    ...
                  Pila &BFF4 : (entrada anterior)
                       &BFF3 : (entrada anterior)
                       &BFF2 : (entrada anterior)
                       &BFF1 : (última entrada)
                       &BFF0 : (lugar para nueva entrada)
                       &BFEF : (lugar para nueva entrada)
                         .           ...
                         .           ...
Stack Pointer  SP:   &BFF1
```

El registro SP indica la última entrada en el Stack. Supongamos que en la ejecución de un programa, el procesador encuentra el comando CALL &B267 en la dirección &780.

```
&780   CALL  &B267
&783   siguiente comando
```

Después de leer el comando el PC (Program Counter) se encuentra en &783. Esta es la dirección de retorno que ha de registrarse. La dirección se coloca en el Stack en la forma de Low-byte High-byte. El registro SP se decrementa, se almacena el High-byte, vuelve a decrementar el SP y se almacena el Low-byte. A continuación se carga el registro PC con la dirección inicial del subprograma (&B267), donde se continúa la ejecución del programa.
Resulta entonces la serie siguiente:

```
        .....    ...
        .....    ...
        .....    ...
Pila:   &BFF0    &07
        &BFEF    &83 : (última entrada)
        .....    ...
        .....    ...
SP: &BFEF
```

Como pueden observar el apuntador SP indica la última entrada.
Con el comando RET, el proceso se invierte:
El byte de la posición de memoria indicada por el SP, se carga como Low-byte en el PC. Se incrementa en 1 el SP y se carga el High-byte de la dirección de retorno en el PC. A continuación se incrementa nuevamente en 1 el SP, indicando entonces la dirección de retorno actual en la pila. La ejecución del programa continúa en la posición indicada por el PC, es decir la dirección de retorno correcta.

```
        .....    ...
        .....    ...
Pila:   &BFF1    ...    SP: &BFF1
        &BFF0    &07
        &BFEF    &83
        .....    ...
```

Los procesos descritos tienen lugar automáticamente en el Z80, siempre que se utiliza un comando CALL o RET. Esto determina que

la sucesión en la pila sea siempre correcta y que el SP indique siempre la posición correcta. Si se modifica el SP directamente a través del programa, puede producirse un desorden en la serie con el consiguiente bloqueo del ordenador, por esto es muy importante utilizar con sumo cuidado los comandos LD SP,x.
Existe también la posibilidad de colocar datos en la pila y llamarlos desde la misma. Para ello se utilizan los comandos:

PUSH (colocar en el Stack)
y
POP (extraer del Stack)

El comando PUSH tiene un funcionamiento análogo al del comando CALL. Los datos que han de almacenarse se graban en la pila una vez decrementado el SP. Con POP se procede a la lectura de los datos y el SP se incrementa automáticamente. La CPU se hace cargo en este caso de varias operaciones. Con PUSH y POP pueden "apilarse" varios pares de registros de 16-bits, a excepción del propio SP.

Formato:

PUSH x POP x
(x:AF, BC, DE, HL, IX, IY)

Dado que el acumulador es un registro de 8 bits así como el flag, cuando estos registros quieren almacenarse en el Stack se tratan conjuntamente.
La técnica del almacenamiento intermedio en el Stack sólo tiene sentido si los registros apropiados no son suficientes.

Ejemplo:

HL contiene el primer sumando
BC contiene el segundo sumando

Ahora llamamos a una subrutina para efectuar la operación de sumar HL y BC. El resultado de la suma se almacena en HL. Si después de la operación fuera necesario saber el contenido original del primer sumando, éste debería guardarse previamente en el Stack.

```
LD HL,sumando-1
LD BC,sumando-2
PUSH HL
CALL SUMAR
...
...
...
POP HL
...
...
```

Si se necesita este sumando puede extraerse de la pila con el comando POP HL.

Ha de tenerse en cuenta que el comando POP correspondiente a un comando PUSH, debe estar siempre en el mismo subprograma, de lo contrario los datos almacenados mediante PUSH se interpretan como dirección de retorno para el comando RET, lo que probablemente ocasionaría el bloqueo del ordenador. Los comandos PUSH y POP no poseen ningún comando BASIC análogo. En BASIC estos comandos deberían codificarse:

Ejemplo BASIC:

```
PUSH AF          BASIC:  POKE SP-1,A : (High-byte)
                         POKE SP-2,F
                         SP=SP-2

POP BC           BASIC:  BC=PEEK(SP)+256*PEEK(SP+1)
                         SP=SP+2
```

Dado que ambos comandos (PUSH y POP) utilizan el registro SP como

indicador de dirección, ambos se cuentan como direccionamiento indirecto.

Ejemplo:

```
PUSH HL                    SP=&BE05
                           HL=&1234

Después de la ejecución:       dirección &BE04:&12
                               dirección &BE03:&34

SP = &BE03
HL = &1234
```

Ejemplo:

```
POP HL                 SP=&BE03
                       HL=&FFFF
```

Después de la ejecución:

SP = &BE05
HL = &1234

La lista de comandos correspondientes la encontrará al final del capítulo 4.2: Comandos de carga de 16 bits.

4.4 COMANDOS DE INTERCAMBIO

Además de los comandos sencillos de transferencia de datos (LD), en el Z80 existe otro tipo de comando que intercambia el contenido de dos áreas. Estos comandos se representan por EX (ingl. exchange:intercambiar).

El comando de este tipo, EX DE,HL intercambia por ejemplo el contenido del registro DE con el del registro HL. El comando EX

con direccionamiento indirecto, intercambia el contenido de los registros HL, IX e IY con el elemento superior de la pila (sin alterar el SP).

Formato:

EX SP,x
x: HL, IX o bien IY

Existen asimismo otros comandos que intercambian los contenidos de las series de pares de registros. Como ya hemos mencionado, cada registro A, BC, DE, HL y F tiene su correspondiente registro A', BC', DE', y F'. Sin embargo, sólo se trabaja a la vez con un par de registros. Si es necesario, no obstante pueden intercambiarse los contenidos de ambos pares de registros.

El comando EX AF,AF' intercambia el contenido del acumulador y del registro flag con los correspondientes A' y F'. El comando EXX intercambia los restantes pares de registros BC, DE y HL con BC', DE' y HL'.
Estos comandos utilizan direccionamiento implícito.

Ejemplo:

```
EX DE,HL        BASIC: ZWI=HL: HL=DE: DE=ZWI

EX (SP),HL      BASIC: ZWI=HL: HL=256*PEEK(SP+1)+PEEK(SP):
                       POKE SP+1,INT(ZWI/256):
                       POKE SP,ZWI-INT(ZWI/256)*256
```

LISTA DE COMANDOS

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	1	1	4	
EXX	(BC DE HL) ↔ (BC' DE' HL')	•	•	•	•	•	•	11 011 001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	1	5	19	
EX (SP), IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	•	•	•	•	11 011 101 11 100 011	2	6	23	
EX (SP), IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	•	•	•	•	11 111 101 11 100 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 100 000	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	① 0	•	0	0	11 101 101 10 110 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 101 000	2	4	16	
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	① 0	•	0	0	11 101 101 10 111 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A - (HL) HL ← HL+1 BC ← BC - 1	•	↕	② ① ↕	↕	1	↕	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	↕	② ① ↕	↕	1	↕	11 101 101 10 110 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A - (HL) HL ← HL-1 BC ← BC-1	•	↕	② ① ↕	↕	1	↕	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	↕	② ① ↕	↕	1	↕	11 101 101 10 111 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1

② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.

↕ = flag is affected according to the result of the operation.

4.5 COMANDOS DE TRANSFERENCIA Y BUSQUEDA DE BLOQUES

A diferencia del comando LD que transfiere únicamente 1 ó 2 bytes, los comandos de transferencia de bloques transfieren todo un bloque de datos. Este tipo de comandos es una particularidad del Z80. Por lo general no es posible disponer de estos comandos en los microprocesadores, pues son muy complejos de realizar para los constructores de ordenadores, pero son de gran utilidad para el programador, ya que aumentan la capacidad de rendimiento de un programa.

Un bloque de datos se caracteriza por lo siguiente:

- La dirección inicial y final del bloque se almacena en el registro HL.
- La longitud del bloque en bytes se almacena en el registro BC (Byte Counter).

Con estos 2 valores es posible definir bloques de hasta 64K de longitud, que comienzan en cualquier posición de memoria. Puesto que el bloque así definido ha de transferirse, se deberá dar previamente la dirección inicial o la dirección final del bloque a transferir, que se almacenará en DE. Una vez colocados estos datos en los registros, puede efectuarse el verdadero comando de transferencia de bloques.

Existen cuatro comandos de transferencia de bloques:

LDD, LDDR, LDI, LDIR

Cada comando de transferencia de bloques decrementa el contador BC después de cada transferencia de 1 byte. Dos de los comandos, LDI y LDIR, incrementan los indicadores HL y DE, señalando entonces la dirección de origen y la dirección de destino del siguiente byte a transferir.

Contrariamente a ello, LDD y LDDR decrementan los contadores; es decir el bloque se transfiere por así decirlo "comenzando por

arriba". Para estos comandos, HL y DE también deberán estar cargados al principio con la dirección de inicio, o bien, la dirección de destino del bloque.

La R que aparece al final del comando corresponde a Repeat (ingl.:repetir). Estos comandos se repiten de forma automática hasta que BC=0; es decir, hasta que queda transferido todo el bloque. Para cada comando en particular rige lo siguiente:

LDI : Carga e (I)ncrementa

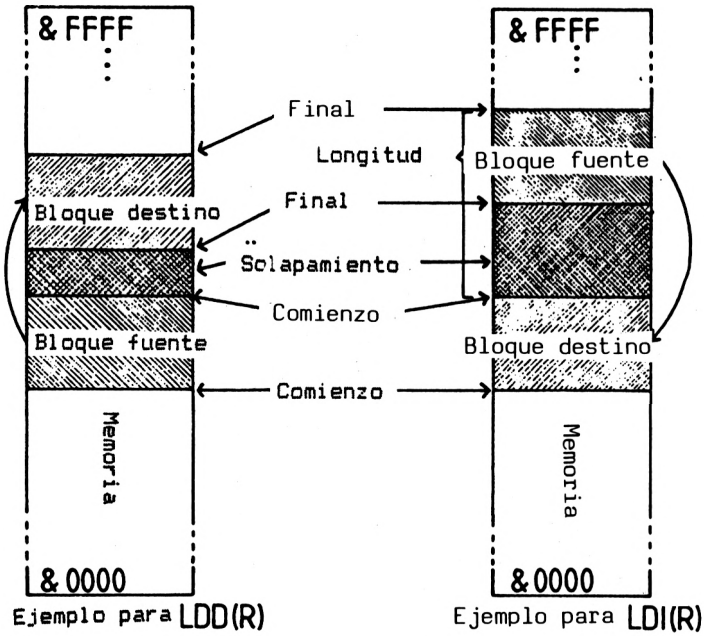
Este comando transfiere un byte de la dirección HL a la dirección DE. A continuación, se decrementa BC. Los indicadores de dirección HL y DE se incrementan, de modo que todo queda dispuesto para una eventual continuación de la transferencia. Para ello, se deberá saltar nuevamente a este comando.

LDIR : Carga, incrementa y repite

El proceso de transferencia se realiza de igual modo que en LDI. A continuación el PC apunta nuevamente de forma automática a este comando. Luego, se vuelve a ejecutar hasta que BC=0. Finalmente, continúa el programa con el comando siguiente.

LDD : Carga y (D)ecrementa

Similar a LDI, sólo que aquí el bloque se transfiere comenzando por la dirección final; es decir, HL y DE se decrementan. Esta diferencia es importante cuando se entrecruzan el bloque de destino y el bloque de origen. Si se utilizase aquí un comando erróneo, en según que condiciones, se modificarían los datos del bloque original antes de haber sido transferidos.



Comandos de transferencia de bloque

LDDR : Carga, decrementa y repite

Similar al comando LDD, sólo que aquí, al igual que en LDIR, se repite el comando, hasta que todo el bloque queda transferido.

Ejemplo:

```
LDIR        BASIC: 10 POKE DE,PEEK(HL)
                   20 HL=HL+1
                   30 DE=DE+1
                   40 BC=BC-1
                   50 IF BC <> 0 THEN 10

LDD         BASIC: POKE DE,PEEK(HL)
                   DE=DE-1: HL=HL-1: BC=BC-1
```

Piense por sí mismo cómo sería la analogía en BASIC para LDDR y LDI.

En el tamaño del programa BASIC, puede observarse que se trata de un comando de gran potencia.
Influencia de los Flags: Cuando tras la ejecución, BC=0, entonces P/V=0.
Los comandos Repeat LDDR y LDIR siempre activan los Flags P/V en 1.

COMANDOS DE BUSQUEDA DE BLOQUE

Con la ayuda de los comandos de búsqueda de bloque, un bloque de datos puede buscarse por el contenido de un determinado byte. El contenido buscado se almacena antes en el acumulador. Si durante la búsqueda, el comando encuentra un byte cuyo contenido es igual al contenido del acumulador, se activa el flag Z y se detiene o bloquea la repetición de los comandos Repeat. Los registros se utilizan de la misma forma que en los comandos de transferencia de bloques.

HL - Dirección inicial o final del bloque
BC - Byte Counter: Longitud del bloque
DE - No tiene función
A - El acumulador contiene el byte a buscar.

CPIR compara en cada transferencia el contenido de la posición de memoria HL con el contenido del acumulador. A continuación incrementa HL y decrementa BC. Si B=0, el flag P/V se desactiva con 0, de lo contrario toma el valor 1. Si en la comparación entre A y HL se presenta igualdad, se activa el flag Z, de lo contrario se desactiva el mismo flag.

El flag S corresponde, como en CP, al séptimo bit del resultado de la resta A-(HL). El carry no se influye.
Existen 4 comandos de búsqueda de bloques:

CPI, CPIR, CPD, CPR

Su modo de funcionamiento corresponde al de cada comando de transferencia de bloque.
Todos los comandos de bloque son comandos de 2 bytes y su primer byte de Opcode es &ED. De igual modo que ocurre con los comandos de transferencia de bloques, con los comandos de búsqueda la programación se simplifica y se hace más rápida en muchos casos.

La lista de comandos correspondiente a la transferencia de bloques y a la búsqueda de bloques se encuentra al final del presente capítulo.

APLICACION

Para comprender plenamente el comando LDDR, lo probaremos a continuación. Queremos desplazar el contenido de la pantalla en un carácter hacia la derecha. Dado que un byte corresponde exactamente al ancho de un carácter, hemos de desplazar el bloque desde &C000 hasta &FFFF por un byte hacia arriba.
Confeccione para ello un programa máquina con la ayuda de los comandos de transferencia de bloques.

Solución:

Analicemos en primer lugar nuestro problema:
El bloque fuente se encuentra en el área &C000 - &FFFE.

Este bloque debe desplazarse en un byte hacia arriba, es decir al área &C001 - &FFFF. Ambos bloques al parecer se superponen. Dado que la dirección final del bloque fuente &FFFE está superpuesta hemos de optar por el comando LDDR.
Calculemos ahora los contenidos de registros HL, DE, BC. Hl contendrá la dirección final del bloque fuente, en este ejemplo &FFFE. BC contiene la cantidad de bytes a desplazar que es &4000-1 (el área de pantalla desde &C000 hasta &FFFF son &4000 bytes) entonces: BC=&3FFF. DE contiene la dirección final del bloque destino &FFFF.
De esta manera resulta el siguiente programa Assembler:

```
LD    HL,&FFFE
LD    DE,&FFFF
LD    BC,&3FFF
LDDR
RET
```

Después de la traducción del programa resultan las siguientes líneas DATA para el programa cargador BASIC:

```
DATA &21,&FE,&FF,&11,&FF,&FF
DATA &01,&FF,&3F,&ED,&B8
DATA &C9
```

(Dirección inicial es &A000 y dirección final &A00B).

Introduzcan ahora *MODE 2* y carguen el programa máquina con *RUN* arrancándolo con *CALL dirección*.

Nuestro programa tiene un pequeño fallo estético:
en el recuadro superior izquierdo permanece un punto.
Para hacerlo desaparecer cargamos la posición de memoria correspondiente &C000 con 0.

```
LD A,00
LD(&C000),A
Code: &3E,&00,&32,&00,&C0
```

Estos comandos los introducimos después del comando LDDR. La última línea DATA es la siguiente:

```
DATA &3E,&00,&32,&00,&C0,&C9
```

(La dirección final se modifica a &A010)
Después de comprobar el programa introduzca lo siguiente:

```
FOR I=1 TO 80: CALL &A000: NEXT
```

El resultado de esta instrucción es que la pantalla se desplaza una línea hacia abajo. Para ello se necesita bastante tiempo porque las 16 Ks de la pantalla deben desplazarse 80 veces. En BASIC este desplazamiento tardaría aproximadamente 1 hora. Si se desplaza el bloque de pantalla en 80 caracteres de una vez, el tiempo de ejecución sería 80 veces menor. Para ello debemos modificar los contenidos de registro en nuestro programa máquina:

HL debe contener &FFFF-80 en lugar de &FFFF-1, siendo &FFAF.
DE queda en &FFFF.

La cantidad de bytes a desplazar es &4000-80=&3FB0.

Modifiquemos ahora las líneas DATA debidamente, y nuestro programa desplaza la pantalla en una línea hacia abajo. Lamentablemente los primeros 80 bytes de la memoria de pantalla quedan en su estado primitivo por lo cual debemos borrarlos. También utilizaremos para ello el comando de transferencia de bloques. Para poder borrar un área mediante el mismo hemos de utilizarlo malintencionadamente:

En primer lugar almacenamos el Null-byte en posición &C000 (LD(&C000),0). A continuación desplazamos el bloque de &C000 hasta &C000+80=&C050 hacia &C001. Dado que las áreas en la dirección final del bloque fuente se superponen, deberíamos utilizar en realidad LDDR.

Utilizando en cambio LDIR, HL=&C000, DE=&C001, BC=&4F, se sobrescribe en todos los casos la posición de memoria que se

trasmite a continuación, con el valor recién insertado. Dado que &C000 tiene el valor 0, todos los bytes del bloque tendrán el valor 0.

El programa completo tiene el aspecto siguiente:

Dirección/código		N. Línea BASIC	Comando ensamblador
A000	21AFFF	10	LD HL,&FFAF
A003	11FFFF	20	LD DE,&FFFF
A006	01B03F	30	LD BC,&EFB0
A009	EDB8	40	LDDR
A00B	3200C0	50	LD (&C000),A
A00E	2100C0	60	LD HL,&C000
A011	1101C0	70	LD DE,&C001
A014	014F00	80	LD BC,&4F
A017	EDB0	90	LDIR
A019	C9	100	RET

Explicación del listado de Assembler:

La dirección se numera consecutivamente según la cantidad de bytes del código. Dado que 1 byte se indica siempre con 2 números HEX resulta el intervalo inexplicable a simple vista de A000 a A003.

El código se compone de 3 bytes: &21, &00, &C0. Como cada byte incrementa la dirección por el valor 1, la dirección inicial del siguiente comando será A003 (A000+3=A003). Fácilmente se puede obtener la longitud de comandos mediante la cantidad de códigos. Los comandos Assembler se posicionan a continuación de los códigos. Su función la explicaremos más adelante.

Si al "trabajar" con el ordenador se produce un Scroll, resultarán irregularidades en el procedimiento del programa máquina. Este fenómeno solamente se produce si no ha borrado la pantalla con el comando MODE 2 antes de llamar el programa.
Prueben además lo siguiente:

```
FOR I=1 TO 26: CALL &A000: NEXT
```

Este comando debería borrar toda la pantalla (25 líneas). Sin embargo las líneas que desaparecen en el borde inferior de la pantalla aparecen nuevamente en el borde superior, en el centro de la línea.

Este hecho se explica por un lado con la estructura de la memoria de pantalla y por otro lado por el hecho de que el scrolling incorporado funciona de diferente forma. Nos ocuparemos de este problema cuando hayamos aprendido unos cuantos nuevos comandos.

Haga unas cuantas pruebas con los comandos de transferencia de bloques:
Utilice diferentes valores para HL, DE y BC.
Tengan en cuenta en todos los casos que el bloque de destino no exceda el área de &C000-&FFFF. Esto provocaría un "colapso" en el ordenador, porque se sobrescribirán rutinas del sistema.
Lo siguiente vale la pena probarlo:

```
HL=&C000,     DE=&FFFF,     BC=&3FFE
```

4.6 COMANDOS ARITMETICOS

Los primeros ordenadores digitales que aparecieron en los años 50 eran considerados fundamentalmente como máquinas calculadoras. Aunque aquellas primeras computadoras poco tienen que ver con los ordenadores actuales, los comandos aritméticos son similares. Existen 2 operaciones aritméticas básicas, la adición y la sustracción, que corresponden a los comandos en lenguaje máquina ADD y SUB. Como el ordenador calcula en el sistema binario, observemos de momento, cómo se realizan estas operaciones en el sistema numérico.

Adición:

En el sistema decimal se suman 2 cifras colocadas una sobre la otra. La posición de la unidad se anota y las eventuales posiciones de las decenas (el resto) se memorizan para la adición de las siguientes cifras.

Ejemplo:

```
   3573
+  7154      (* Aquí debería usted memorizar un 1
-------         al acarreo. Esta cifra corresponde al
  10727         desbordamiento)
  *  *
```

Se produce acarreo siempre que la suma de 2 cifras es mayor que 9 (10-1). En el sistema binario se produce un acarreo cuando la suma de 2 cifras es mayor que 1 (2-1).

Reglas:

```
0 + 1 = 1
1 + 0 = 1
0 + 0 = 1
1 + 1 = 0  <---- (en la última suma debe acarrear 1)
```

Aplicación:

```
  1 0 0 1 0 1 1 0 = &96 = 150
+ 0 0 1 1 1 0 0 1 = &39 =  57
-----------------   ---   ---
  1 1 0 0 1 1 1 1 = &CF = 207
    *   *   *
```

(* significa: 1 de acarreo)

En el sistema hexadecimal se obtiene el mismo resultado.

Se produce un acarreo cuando el resultado es mayor que 15.

```
  0 0 1 0 1 1 1 0 = &2E =  46
+ 0 0 0 1 0 1 1 1 = &17 =  23
-----------------   ---   ---
  0 1 0 0 0 1 0 1 = &45 =  69
```

&E+&7 = 14+7 = 21 = &15

Es decir: anotar 5, acarrear 1.

Con el ejemplo anterior, aún se produce otro caso en la adición binaria:

```
  11
+ 11
----
 110
```

En la segunda posición es válida la siguiente regla:

1 + 1 + 1 = 1, y 1 de acarreo.

Aplicación:

```
1)      1 0 1 0 1 1 1 0  =  &  ?  =   ?
      +  0 0 1 0 1 1 1 1  =  &  ?  =   ?
        ------------------     ----   ----
                ?                ?      ?
```

```
2)      0 0 1 1 1 1 1 1  =  &  ?  =   ?
      + 0 0 1 0 1 1 1 1  =  &  ?  =   ?
      ------------------     ----    ---
              ?          =  &  ?  =   ?

3)      1 1 1 1 1 1 1 1  =  &  ?  =   ?
      + 1 1 0 0 1 0 1 0  =  &  ?  =   ?
      ------------------     ----    ---
              ?          =  &  ?  =   ?
```

Solución:

```
1)      1 0 1 0 1 1 1 0  =  &AE   = 174
      + 0 0 1 0 1 1 1 1  =  &2F   =  47
      ------------------     ----   ----
        1 1 0 1 1 1 0 1     &DD     221

2)      0 0 1 1 1 1 1 1  =  &3F   =  63
      + 0 0 1 0 1 1 1 1  =  &2F   = 157
      ------------------     ----   ---
        1 1 0 1 1 1 0 0  =  &DC   = 220

3)      1 1 1 1 1 1 1 1  =  &FF   = 255
      + 1 1 0 0 1 0 1 0  =  &CA   = 202
      ------------------     ----   ---
       1 1 1 0 0 1 0 1 1  =  &1C9  = 457
```

Referencia a 3). En esta adición se produce un acarreo de la posición 8 (bit 7) a la posición 9 (bit 8). Sin embargo un byte tiene únicamente 8 bits, por esta razón el "bit de acarreo", el Carry, se almacena en el bit 0 del registro flag. En principio

pueden sumarse también números de varias cifras, pero para ello debe procederse de otra forma.

Sustracción

La substracción en el sistema binario es análoga a la del sistema decimal.
Existen las reglas siguientes:

```
0 - 1 = 1      memorizar 1
1 - 0 = 1
0 - 0 = 0
1 - 1 = 0
```

Veamos un ejemplo:

```
  0 1 1 0 1 1 1 0 = &6E = 110
- 0 0 1 1 0 1 0 1 = &35 =  53
-----------------   ---   ---
  0 0 1 1 1 0 0 1   &39    57
      * *       *
```

Reconoceremos las reglas especiales para la continuación del cálculo con el acarreo:

```
1 - (1+1) = 1      acarrear 1
0 - (1+!) = 0      acarrear 1
```

Ejercicios:

Realice los mismos ejercicios planteados para la adición, ahora con la substracción. Compruebe usted mismo los resultados convirtiendo al sistema decimal.

Referente a 2). Tras finalizar el cálculo, se obtiene un resultado negativo. El resultado correcto sería 63-157=-84. En sistema binario:

```
  0 0 1 1 1 1 1 1
- 1 0 0 1 1 1 0 1
-----------------
1 1 0 1 0 0 0 1 0 = &1A2
```

Evidentemente, este resultado es erróneo. En la sustracción binaria con el ordenador, se presenta el problema de que aparecen números negativos. Para ello se ha establecido la convención siguiente:

El bit 7 de un número binario se utiliza como bit de signo. 0 indica cantidades positivas, 1 indica valores negativos. Con esto se limita, de -128 hasta +127, el margen de números codificados por un byte. La sustracción de números binarios lleva de esta forma a la adición algebraica (con signo 5-2=5+(-2)). La representación con signo que se utiliza en la sustracción se denomina complemento a 2.

¿Qué es el COMPLEMENTO A DOS?

En la representación del complemento a dos, los números positivos se representan al igual que hasta ahora hemos visto, p.ejemplo 5=&B00000101, 126=&B01111110.

Un número negativo se representa calculando en primer lugar su complemento. El complemento es el número binario, en el que todos los bits tienen el valor inverso, 0 se convierte en 1 y viceversa. El número binario así obtenido de denomina "complemento a uno" o simplemente complemento.

Ejemplo:

```
Número      : 7 = &x00000111
Complemento :     &x11111000
```

Para calcular el complemento a dos de un número se debe sumar 1 al mismo.

Ejemplo:

```
Complemento       :      &x11111000
más 1               +           1
                      ------------
Complemento a 2   :      &x11111001
```

Esta es la representación de -7 en complemento a 2.

Así pues, el complemento a dos se forma de la siguiente manera:

- un número positivo permanece invariable
- en los números negativos se calcula el complemento y se le suma 1

Representación del complemento a 2.

Decimal	Complemento a dos
+127	&X01111111
+126	&X01111110
+125	&X01111101
.	...
.	...
.	...
+ 2	&X00000010
+ 1	&X00000001
0	&X00000000
- 1	&X11111111
- 2	&X11111110
- 3	&X11111101
.	...
.	...
-126	&X10000010
-127	&X10000000
-128	&X10000000

Para conservar el valor de un número negativo representado en complemento a dos, se construye nuevamente a partir del número el complemento a dos.

Ejemplo:

```
&x00000111      Complemento
+         1     más 1
----------
&X00001000
```

&X00001000 = 8

¡Esto significa que el valor de &X11111000 es -8!
Una doble aplicación de complemento a dos nos lleva nuevamente al número de partida.

El Z80 dispone de comandos para la conversión del contenido del acumulador en el complemento (CPL) y en el complemento a dos (NEG). A continuación veremos la función de estos comandos en sus análogos de BASIC:

Observemos en primer lugar la formación del complemento:
'A' contiene un número entre 0 y 255 (1 Byte). El comando BIN$ convierte un número en un string, que corresponde al valor en binario del número. Complementaremos esta cadena de caracteres trabajando bit a bit.

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Nro. Binario :";abin$
40 FOR i=0 TO 7
50 bit$=MID$(abin,8-i,1): REM Bit Nro. 1
60 IF bit$="1" THEN bit$="0" ELSE bit$="1"
70 akpl$=bit$+akpl$ : REM akpl$ es complemento $ de a
80 NEXT
90 PRINT "Complemento: "; akpl$
100 A=VAL ("&X"+akpl$)
```

La línea 50 extrae, si corresponde, el Bit-i de abin$. En la línea 60 se construye el complemento del bit; es decir, si contiene un 0 se inserta un 1 y viceversa. En la línea 70 se reúnen los bits complementados en akpl$. El presente programa es lógicamente bastante lento. El comando XOR puede calcular el complemento en BASIC de forma más rápida. En este punto sólo nos interesa presentar el programa; la modalidad de funcionamiento de este comando lógico la expondremos en el siguiente capítulo.

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Nro. Binario:"; abin$
40 a=a XOR 255
50 akpl$=BIN$(a,8)
90 PRINT "Complemento: ";akpl$
```

La línea 40 realiza la verdadera formación del complemento.

El comando NEG convierte un número positivo en uno negativo codificado en complemento a dos. En BASIC obtenemos idéntico resultado agregando la línea 45:

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Nro. Binario:"; abin$
40 a=a XOR 255
45 a=a+1
50 akpl$=BIN$(a,8)
60 PRINT "Complemento: ";akpl$
```

Inserte también la línea siguiente:

```
100 GOTO 40
```

Tras la interrupción de este programa sin fin, comprobará que la realización de una doble formación del complemento a dos lleva nuevamente al punto de partida.

Con la representación del complemento a dos, se considera una sustracción de dos números como adición de uno de ellos con el complemento a dos del otro. Por otro lado, cuando se coloca el Bit 7, el resultado de una substracción se considera como número negativo (representado en complemento a dos).

Ejemplo:

```
120 - 63 = 57
120 = &X01111000
 63 = &X00111111
```

El complemento a dos de 63 es &X11000001
Ahora efectuamos la adición:

```
   01111000  =  120
+  11000001  = Complemento a dos de 63
-----------
  100111001
```

No tengamos en cuenta de momento el desbordamiento del bit 7 al bit 8 (Carry). Nuestro resultado es correcto: &X00111001 = 57.

El bit 7 no se ha activado, lo que quiere decir que el resultado es positivo, según lo cual no debería activarse tampoco el Carry.

Dado que estamos calculando con el complemento a dos, el resultado del acarreo queda también complementado. En este caso no es necesario tenerlo en cuenta. Nuestro resultado es correcto de todos modos.

La observación exacta de la aritmética trabajando con números con signo, demuestra que deben tenerse en cuenta algunos casos especiales, donde es muy importante el efecto conjunto de los flags.

Ejercicio:

Calcule el complemento a dos de:

1) -60
2) -120
3) +5
4) -6

Soluciones:

1) &X11000100 (=196)
2) &X10001000 (=136)
3) &X00000101 (=5)
4) &X11111010 (=250).

COMANDOS ARITMETICOS Y DE CALCULO DE 8-BITS

Existen 2 comandos de este tipo para la adición y la substracción:

ADD;ADC y SUB;SBC

Para los comandos que utilizan la letra C (-Carry), debe tenerse en cuenta en la operación el flag de acarreo en su forma correspondiente. Al utilizar uno de estos dos comandos, se sumará o restará el bit 0 del registro F (¡¡el carry!!).
Los operandos de estos comandos tienen el formato siguiente:

A,x donde x corresponde a reg, datos, (HL), (IX+d) o (IY+d)

De esta forma se obtienen los siguientes tipos de instrucciones.

```
A,reg        - implícito
A,datos      - inmediato
A,(HL)       - indirecto
A,(XY+des)   - indexado
```

Con el comando SUB, sólo se indican como operandos: reg, datos (HL), (IX+d) o (IY+d). "A" se excluye debido a que todos los comandos de este tipo se refieren al Acumulador.
Estos comandos son operaciones de 8 bits. El Z80 contiene además operaciones de 16 bits.

INFLUENCIA DE FLAGS

Al ejecutar comandos que manipulan datos, se influyen los siguientes flags:

Flag de acarreo (Carry)

El carry se activa cuando tiene lugar un acarreo del bit 7 al bit 8. Puesto que un byte se compone sólo de los bits 0 a 7, el acarreo se almacena en el flag C. En caso contrario, el flag de acarreo se desactiva.

Flags N y H

Estos flags se influencian en la aritmética BCD, pero carecen de importancia para el programador.

Flags P/V - Overflow

Un desbordamiento se define de la siguiente forma:

- Cuando hay un acarreo interno del bit 6 al bit 7, y no hay acarreo del bit 7 al bit 8 (el acarreo que se indica mediante el Carry, se denomina acarreo externo).

- Cuando no hay ningún acarreo interno, pero sí uno externo.

No precisaremos aquí cómo han sido formuladas estas definiciones. Lo importante es saber que este flag es activado, cuando en una operación aritmética, el signo del resultado (Bit 7) se modifica con error. El flag V se activa cuando tiene lugar un desbordamiento, de lo contrario se desactiva.

Zero-Flag

Este flag se activa si el resultado de la operación es 0, de lo contrario se desactiva.

Flag de signo

Este flag corresponde al bit 7 del resultado. En la representación numérica con signo, éste corresponde al signo, de aquí le viene su nombre de flag de signo.

Encontrará una tabla de la influencia de los flags en el Anexo del presente libro.

En lo sucesivo, para la explicación de los comandos escribiremos lo siguiente para el estado de un flag después de una operación:

1 - El flag se activa después de la operación.
0 - El flag se desactiva después de la operación
X - El flag es desconocido después de la operación
↕ - El flag se activa o desactiva como resultado de la operación
P - El flag P/V indica la paridad
  - Carácter en blanco: no hay influencia

```
Ejemplo:      Flags S Z P/V C
                    X | 1
              significado:
                           S   - desconocido
                           Z   - si 0 entonces 1 y viceversa
                           P/V - 1
                           C   - ninguna influencia
```

Analogías BASIC para los comandos:

```
ADD A,H          BASIC:  A=A+H

ADC A,&A9        BASIC:  A=A+&A9+CF
```

CF es el Carry-flag, su valor se suma adicionalmente.

```
SUB A,(HL)       BASIC:  A=A-PEEK(HL)
SBC A,L          BASIC:  A=A-L-CF
```

Ejemplos:

```
ADD A,(HL)        A  = &1F
                  HL = &B1C9
```

Posición de memoria &B1C9: &43

```
      &1F = 0 0 0 1 1 1 1 1
    + &43 = 0 1 0 0 0 0 1 1
            ---------------
            0 1 1 0 0 0 1 0
          8 7 6 5 4 3 2 1 0  - Números de bit
```

Bit-8 = 0 => Carry flag = 0
Bit-7 = 0 => Sign-flag = 0
Resultado <>0 => Zero-flag = 0
Desbordamiento externo = 0 e interno = 0 => overflow (P/V)-flag

Contenido del acumulador después de la operación: &X011000110
&62

```
ADD A,D      A contiene &E1
             D contiene &A2
```

```
  &E1 =   1 1 1 0 0 0 0 1
+ &A2 =   1 0 1 0 0 0 1 0
-----   -----------------
 &183 = 1 1 0 0 0 0 0 1 1
        8 7 6 5 4 3 2 1 0   - Nro. bit
```

```
Bit 8 = 1 => Carry-flag = 1
Bit 7 = 1 => Sign-flag  = 1
Resultado no nulo       => Zero-flag = 0
Desbordamiento externo e interno  => overflow (P/V) flag = 0

Contenido del ACU tras la ejecución: &X10000011=&83
```

Como pueden observar, el acumulador no contiene el resultado correcto; sólo obtenemos el resultado correcto agregando el flag de acarreo como octavo bit. Por este motivo es importante comprobar el estado de los flags tras las operaciones aritméticas y corregir así los eventuales resultados incorrectos.

Observe además que en una operación cuyo resultado es exactamente 256, se activa el flag de cero, aunque el resultado no sea nulo.

```
ADC  A,&19          A=&5A
                    Carry-flag = 1 (activado)

  &5A = 0 1 0 1 1 0 1 0
+ &19 = 0 0 0 1 1 0 0 1
       ---------------
  &74 = 0 1 1 1 0 1 0 0

Flags: S Z V C      ACU = &X01110100 = &74
       0 0 0 0
```

Recuerde: si se desactiva el carry-flag antes de un comando ADC, éste corresponderá exactamente al comando ADD.

```
SUB A,(HL)

      A contiene &3C
     HL contiene &BC19
  &BC19 contiene &15

      0 0 1 1 0 1 1 0          &36
      1 1 1 0 1 0 1 1          complemento a dos de &15
    -----------------
    1 0 0 1 0 0 0 0 1

Bit 7=0 => el Sign-flag = 0
Bit-8=1 => Carry-flag=0

Observen  que  en nuestro ejemplo el complemento se ha tomado  del
verdadero Carry (¡caso especial!).

No hay desbordamiento V=0
Resultado <> 0 => Z=0

Contenido del ACU después de la ejecución &X00100001=&21

SBC A,B
    A=&57
    B=&73

    CF=1

      0 1 0 1 0 1 1 1  =  &57
   +  1 0 0 0 1 1 0 1  = Complemento a dos de &73
   +  1 1 1 1 1 1 1 1  = Complemento a dos de &1(CF)
   -------------------
    1 1 1 1 0 0 0 1 1


Flags:  S  Z  V  C
        1  0  0  1
```

Contenido del ACU: &X11100100 = &E4
complemento a dos de 29
es decir, el resultado es -29 (87-115-1=-29)

Además de la aritmética binaria, existe otra modalidad de efectuar operaciones en el ordenador.

Aritmética BCD

En esta modalidad, cada cifra del sistema decimal se representa por un bloque de 4 bits. Esta aplicación es importante para el tratamiento de problemas comerciales, que exigen normalmente mantener una perfecta exactitud de los dígitos decimales. Para las operaciones BCD existe el comando especial DAA, que prepara el contenido del acumulador para tales operaciones.

Existen también los comandos especiales CPL y NEG, ya descritos. CPL complementa el contenido del ACU y NEG lo convierte en complemento a dos.

Esta aritmética convierte asimismo algunos comandos "normales" en especiales; p. ej. para borrar el ACU, puede utilizarse SUB A. Este comando efectúa la operación en la mitad de tiempo que el comando LD A,0.

A este tipo de comandos corresponden asimismo los comandos de contador, que incrementan o decrementan el valor de una posición de memoria. Para los comandos de contado se dispone de los direccionamientos implícito, indexado y de registro. Los comandos de este tipo se utilizan frecuentemente para la programación de bucles. Su modalidad de funcionamiento es sencilla:

```
INC  x      incrementa x y
DEC  x      decrementa x, donde x puede ser:
```

reg, (HL), (XY+des)

```
INC reg      BASIC: reg=reg+1
DEC (HL)     BASIC: POKE HL,PEEK (HL)-1
```

Cada uno de los flags de signo, de cero y V se activan o desactivan, según el resultado de la operación. El carry permanece inalterado. Es importante remarcar el hecho de que sólo los comandos de contador de 8 bits influyen sobre los flags; por el contrario en los comandos de contador de 16 bits, debe efectuarse una comparación adicional.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	V	↕	0	↕	10 [000] r	1	1	4	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
ADD A, n	A ← A + n	↕	↕	V	↕	0	↕	11 [000] 110 ← n →	2	2	7	
ADD A, (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 [000] 110	1	2	7	
ADD A, (IX+d)	A←A + (IX+d)	↕	↕	V	↕	0	↕	11 011 101 10 [000] 110 ← d →	3	5	19	
ADD A, (IY+d)	A←A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101 10 [000] 110 ← d →	3	5	19	
ADC A, s	A ← A + s + CY	↕	↕	V	↕	0	↕	[001]				s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction
SUB s	A ← A - s	↕	↕	V	↕	1	↕	[010]				
SBC A, s	A ← A - s - CY	↕	↕	V	↕	1	↕	[011]				
AND s	A ← A ∧ s	0	↕	P	↕	0	1	[100]				
OR s	A ← A ∨ s	0	↕	P	↕	0	0	[110]				The indicated bits replace the 000 in the ADD set above.
XOR s	A ← A ∘ s	0	↕	P	↕	0	0	[101]				
CP s	A - s	↕	↕	V	↕	1	↕	[111]				
INC r	r ← r + 1	•	↕	V	↕	0	↕	00 r [100]	1	1	4	
INC (HL)	(HL) ← (HL)+1	•	↕	V	↕	0	↕	00 110 [100]	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	•	↕	V	↕	0	↕	11 011 101 00 110 [100] ← d →	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d) + 1	•	↕	V	↕	0	↕	11 111 101 00 110 [100] ← d →	3	6	23	
DEC m	m ← m-1	•	↕	V	↕	1	↕	[101]				m is any of r, (HL), (IX+d), (IY+d) as shown for INC. Same format and states as INC. Replace 100 with 101 in OP code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

COMANDOS ARITMETICOS DE 16 BITS

En principio, los comandos aritméticos de 16 bits se asemejan a los de 8 bits, aunque tienen mayores limitaciones. Para algunos pares de registros, sólo se presentan los comandos ADD, ADC y SUB. El resultado de una operación se coloca básicamente en el registro HL (no en el acumulador, como en los comandos de 8 bits). Con el comando ADD existe la posibilidad de almacenar resultados en los registros índices.

Los comandos de 16 bits representan ejecuciones sucesivas de comandos de 8 bits. Como la asociación de estos comandos se produce de forma automática, los comandos de 16 bits son más rápidos y más cortos.

16-BITS	8-BITS
ADD HL,BC	LD A,L
	ADD A,C
	LD L,A
	LD A,H
	ADC A,B
	LD H,A

Algunos comandos aritméticos de 16 bits utilizan el direccionamiento implícito. La influencia de flags en ADC y SBC es análoga a la de los comandos de 8 bits. Con ADD sólo se influencia en flag carry, ¡y en los comandos INC y DEC, de 16-bits, no se tienen en cuenta los flags!.

```
ADD IX,DE          BASIC: IX=IX+DE
ADC HL,BC          BASIC: HL=HL+BC+CF
SBC HL,SP          BASIC: HL=HL-SP-CF
```

Ejemplo:

```
      HL=&C000
      DE=&0800

ADD HL,DE

      &C000 = 1100 0000 0000 0000
    + &0800 = 0000 1000 0000 0000
    ----------------------------
      &C800 = 1100 1000 0000 0000

Flags: S  Z  V  C
               0
```

Los flags S, Z y V no quedan afectados.

```
HL=&F800
DE=&0800

ADC HL,DE

      &F800 =   1111 1000 0000 0000
    + &0800 =   0000 1000 0000 0000
    -------   --------------------
     &10000 = 1 0000 0000 0000 0000

Flags: S  Z  V  C
       0  0  1  1
```

En este caso el contenido de HL no representa el resultado correcto &10000, sino que tiene 0. El flag carry señala el error. En las operaciones con 16 bits, es el bit 16 el que pone de manifiesto este error.
Todos los comandos de cálculo de 16 bits son de direccionamiento implícito. Pueden referirse a los registros de 16 bits: BC, DE, HL, SP, IX e IY. Contrariamente a los comandos de cálculo de 8 bits, estos comandos no influyen sobre los flags (!).

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD HL, ss	HL ← HL+ss	‡	•	•	•	0	X	00 ss1 001	1	3	11	ss Reg. 00 BC 01 DE 10 HL 11 SP
ADC HL, ss	HL←HL+ss+CY	‡	‡	V	‡	0	X	11 101 101 01 ss1 010	2	4	15	
SBC HL, ss	HL←HL-ss-CY	‡	‡	V	‡	1	X	11 101 101 01 ss0 010	2	4	15	
ADD IX, pp	IX ← IX + pp	‡	•	•	•	0	X	11 011 101 00 pp1 001	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY←IY+rr	‡	•	•	•	0	X	11 111 101 00 rr1 001	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	•	•	•	•	•	•	00 ss0 011	1	1	6	
INC IX	IX ← IX + 1	•	•	•	•	•	•	11 011 101 00 100 011	2	2	10	
INC IY	IY ← IY + 1	•	•	•	•	•	•	11 111 101 00 100 011	2	2	10	
DEC ss	ss ← ss - 1	•	•	•	•	•	•	00 ss1 011	1	1	6	
DEC IX	IX ← IX - 1	•	•	•	•	•	•	11 011 101 00 101 011	2	2	10	
DEC IY	IY ← IY - 1	•	•	•	•	•	•	11 111 101 00 101 011	2	2	10	

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
‡ = flag is affected according to the result of the operation.

Aplicación:

Una vez dejado atrás el tramo más árido, vamos por fin a aplicar por vez primera los nuevos comandos. Escriba un pequeño programa para la suma de 2 cifras de 8 bits. Las cifras se almacenarán en la memoria RAM mediante comandos POKE de BASIC. El resultado de la suma debe almacenarse nuevamente en el RAM. Después de volver al BASIC, podremos leer el resultado utilizando PEEK.

Solución:

Dado que las sumas de 8 bits utilizan básicamente el ACU, el primer sumando deberemos almacenarlo en el ACU.

```
LD A,sumando
```

El segundo sumando lo almacenaremos en un registro de 8 bits.

```
LD H,sumando
```

Ahora efectuaremos la suma:

```
ADD A,H
```

El resultado debemos almacenarlo en &A100:

```
LD (&A100),A
```

Si seleccionamos como dirección inicial &A000, obtenemos el cuadro siguiente:

```
A000 3E10      10      LD   A,&10
A002 2620      20      LD   H,&20
A004 87        30      ADD  A,H
A005 3200A1    40      LDD  (&A100),A
A008 C9        50      RET
```

La línea DATA del programa cargador será:

```
60 DATA &38,&10,&26,&20,&84,&32,&00,&A1,&C9
```

Del listado ensamblador se deduce que el primer sumando es almacenado en la dirección &A001, y el segundo en la dirección &A003. En nuestro ejemplo hemos elegido &10 y &20. El programa BASIC, que establece estos valores, ejecuta el programa máquina y extrae los resultados, tiene el siguiente aspecto:

```
10 POKE &A001,1er. Sumando
20 POKE &A003,2do. Sumando
30 CALL &A000
40 PRINT PEEK (&A000)
```

4.7 COMANDOS LOGICOS Y CP (COMPARE)

En los comandos para el tratamiento de datos se incluyen también los comandos lógicos.

El Z80 posee los comandos lógicos AND, OR y XOR (OR exclusivo), y el comando de comparación CP. Todos estos comandos trabajan con datos de 8 bits. El ACU es siempre el registro con el que se realiza la operación lógica. Por esta razón, el ACU no es necesario indicarlo (como por ej. en ADD A,B) en el operando del comando ensamblador (como p.ej. AND B).
Los cuatro comandos AND, OR, XOR y CP pueden codificarse con los tipos siguientes de direccionamiento:

-Implícito (registro A, B, C, D, E, H, L)
-Indirecto : registro (HL)
-Indexado
-Inmediato

Observemos las funciones de los comandos lógicos. Cualquier persona es capaz de comprender la siguiente sentencia lógica:

"Cuando llueva (entonces) la calle se mojará"

Esta expresión es una deducción de la forma
(cuando ...., entonces ....).
Observemos ahora la sentencia siguiente:

"Cuando llueve, y yo estoy en la calle, (entonces) me mojo".

En este caso hay 2 expresiones unidas por la conjunción Y. El Y lógico (ingl.:AND) expresa que ambas expresiones, es decir "llueve" (1ra. expresión) y "yo estoy en la calle" (2da. expresión), han de tener lugar para que se produzca el resultado. Si no llueve (no se cumple la 1ra. expresión), yo no me mojo. Si no estoy en la calle (no se cumple la 2da. expresión), tampoco me mojo. Para que la consecuencia se produzca (sea verdadera), han de cumplirse ambas expresiones. Esta es la particularidad de la conexión AND (Y). Ya que el ordenador trabaja con 0 y 1, se adopta la convención siguiente:

1 - corresponde a "expresión verdadera"
0 - corresponde a "expresión falsa"

Con ello se obtiene:

```
1 AND 1 = 1 ambas expres. son verdaderas => resultado verdadero
1 AND 0 = 0 una expresión es falsa       => resultado falso
0 AND 1 = 0 una expresión es falsa       => resultado falso
0 AND 0 = 0 ambas expresiones son falsas => resultado falso
```

El BASIC del CPC contiene los comandos lógicos. Pruébelos usted mismo:

```
PRINT 1 AND 1
PRINT 1 AND 0     etc...
```

Las operaciones lógicas son de máxima importancia para la técnica de ordenadores y son fáciles de realizar electrónicamente. Para ello hay 2 direcciones de entrada que llevan corriente (=1), o que no la llevan (=0), conectadas a un circuito eléctrico, cuya dirección de salida conduce o no corriente (es 1 ó 0)según sean las necesidades de la entrada. Tales circuitos se comprenden con ayuda del Algebra de Boole. Un microprocesador está compuesto por múltiples puertas lógicas cerradas, situadas una después de la otra. La suma en el MPU se forma p.ej., a partir de diversas operaciones lógicas.

Nosotros como programadores que somos, no entramos en contacto con estas estructuras. Aplicamos las operaciones lógicas a datos (8 bits). Para ello se unen en operaciones lógicas cada uno de los correspondientes bits de ambos bytes.

```
      11111000
AND   01010011
      --------
      01010000

Bit 0: 0 AND 1 = 0
Bit 1: 0 AND 1 = 0
Bit 3: 0 AND 0 = 0
Bit 4: 1 AND 0 = 0
Bit 5: 1 AND 1 = 1
      .....
      .....
```

Una de las aplicaciones más útiles del comando AND consiste en la utilización de este comando para borrar o anular determinados bits.

```
      A=&X10111001
```

Supongamos que queremos únicamente observar los bits 0 a 3, es decir, han de anularse desde el bit 4 hasta el bit 7. Para conseguirlo unimos (con Y) A con &X00001111.

```
        10111001        :A
AND     00001111        :Máscara
        --------
        00001001
```

La máscara utilizada para anular los bits, contiene un '0' para un bit que debe anularse, y un '1' para conservar el valor de un bit significativo.

Formulemos en BASIC:

```
A = &X10111001
A = A AND &X00001111
```

En lenguaje máquina obtenemos:

```
LD  A,&X10111001
AND &X00001111
```

Observe las siguientes expresiones:

"Cuando llueve, O me baño, (entonces) me mojo"

El resultado será cierto cuando, como mínimo, una de las dos expresiones sea verdadera. Con ello, obtenemos el cuadro siguiente para la conjunción 'O' (OR).

```
0  OR  0 = 0
0  OR  1 = 1
1  OR  0 = 1
1  OR  1 = 1
```

Con la conjunción OR es posible activar determinados bits de un byte.
A contiene &X10001011
Ahora han de colocarse los 3 primeros bits (5, 6, 7) en '1':

```
      10001011        :A
  OR  11100000        :Máscara
      --------
      11101011
```

Para cada bit que ha de colocarse a 1, la máscara comtiene un 1 y para los bits que no han de modificarse, un 0.

```
LD A,&X10001011       BASIC: A=&X10001011
OR &X11100000         BASIC: A=A  OR  &X11100000
```

El comando XOR o "OR exclusivo", sólo se diferencia en un punto del OR normal. Si ambos bits contienen un 1, la salida será 0. El comando OR exclusivo suministra un 1 para entradas diferentes y un 0 para entradas iguales:

```
0  XOR  0 = 0
0  XOR  1 = 1
1  XOR  0 = 1
1  XOR  1 = 0
```

Existen 2 aplicaciones típicas para la relación XOR, la comparación y la complementación. Los bytes que han de compararse se unen mediante XOR. Si el resultado es 0, esto significa que todos los bytes comparados han resultado iguales; si se produce en cambio desigualdad, quedan activados los bits desiguales en el resultado.

```
      10101010
 XOR  10101010     ¡Comparación!
      --------
      00000000

      10101010
 XOR  10101100     ¡Comparación!
      --------
      00000110
```

=> Bit 1 y Bit 2 son diferentes.

Para la complementación se unen nuevamente los bits con una máscara. Esta contiene un 1 para un bit que ha de complementarse y un 0 para un bit que permanece invariable.

Bit 4-7 deben complementarse.

```
      10101111        :A
XOR   11110000        :Máscara
      --------
      01011111
```

Analogías:

Lenguaje máquina	BASIC
AND H	A=A AND H
OR (HL)	A=A OR PEEK (HL)
XOR &FF	A=A XOR &FF

En los comandos lógicos, el carry siempre se coloca en 0. Los flags Z y S, como es habitual, se influyen. El flag P/V indica en estos comandos la paridad del resultado. La paridad será 1 cuando la cantidad de '1' en el byte sea par, y será cero cuando el número de '1' del byte sea impar.

Aplicación:

1. ¿Qué produce un:
   - OR con &FF ?
   - OR con &0 ?
   - AND con &FF ?
   - AND con &0 ?
   - XOR con &FF ?
   - XOR con &0 ?

2. En Basic existe el comando NOT. Convierta este comando a lenguaje máquina siguiendo dos modalidades diferentes (respecto al ACU).

Soluciones:

Para 1.
OR &FF => &FF, es decir, todos los bits están activados
OR &0 => no hay ninguna modificación
AND &FF => ninguna modificación
AND &0 => &0 es decir, todos los bits has sido desactivados
XOR &FF => todos los bits se han complementado
XOR &0 => ninguna alteración

Para 2.

Comando XOR: XOR &FF

Comando CPL: CPL

EL COMANDO DE COMPARACION CP (INFLUENCIA DE FLAGS)

El comando CP se utiliza para comparar el contenido del ACU con un byte, que puede estar direccionado de las formas siguientes:

- direccionamiento implícito: registros A, B, C, D, E, H, L
- direccionamiento indirecto: par de registros (HL)
- direccionamiento indexado
- direccionamiento inmediato

Mediante el comando CP se sustrae al acumulador el byte direccionado y se influyen cada uno de los flags según el resultado de la operación. Contrariamente al comando SUB, el resultado no se almacena en el acumulador, es decir, el contenido del ACU no queda afectado por el comando. Dependiendo del estado de los flags, puede efectuarse un salto condicional después de

este comando.
Observemos los posibles casos que se presentan en la comparación:

El contenido del acumulador es mayor:
- En este caso, el carry siempre es 0, pues el resultado puede ser mayor que 255.

El contenido del acumulador es igual:
- En este caso, Z=1, debido a que el resultado de la sustracción es 0. También aquí C=0 debido a que no se realiza ningún acarreo.

El contenido del acumulador es menor:
- En este caso, el flag de acarreo siempre estará activado, pues se realiza un acarreo negativo.

REGLAS:

C=0	significa	>=
Z=0	significa	=
C=1	significa	<

Se obtiene además:

Z=1	significa	<>
C=0 y Z=1	significa	>
C=1 o bien Z=0	significa	=<

Estas reglas serán válidas sólo cuando los bytes que han de compararse se consideran como números sin signo, situados entre 0 y 255.

Si ambos bytes presentan números con signo en complemento a dos, adquieren validez unas reglas más complejas que resultan de las reglas a las que están sujetos los flags para la aritmética con

signos. En la mayoría de los casos no es necesaria esta aplicación.
Para la decisión de la comparación de igualdad se utiliza el flag Z. Según el estado de los flags S y V se decidirá si es mayor o menor. Los flags S y V se enlazan lógicamente mediante el comando XOR; es decir, si se activa V (tiene lugar un desbordamiento), se complementa S, de lo contrario S permanece en su estado anterior.

```
S XOR V = 0          significa    >=
S XOR V = 1          significa    <
```

En lo sucesivo, partiremos de la premisa de que los bytes se interpretarán como números carentes de signo.

Ejemplo:

```
      A = &35
      B = &21

      CP B

      suministra a S  Z  V  C
                   0  0  0  0  debido a:

      00110101  :A
    - 00100001  :B (ningún (!) complemento a dos)
      --------
      00010100

      No hay acarreo                 =>  C=0
      Bit=0                          =>  S=0
       <> 0                          =>  Z=0
      No hay desbordamiento          =>  V=0
```

El Carry-flag es igual a cero, lo que significa que el contenido del acumulador es mayor que el del byte comparado (contenido del registro B).

```
      C = &81

      CP C   proporciona

Flags: S  Z  V  C
       1  0  1  1    debido a:

      00000001        : Registro A
    - 10000001        : Registro C
      --------
     110000000

Acarreo de bit 7 a bit 8        => C=1
Bit 7 = 1                       => S=1
   <>                           => Z=0
acarreo de bit 7 a bit 8
ningún acarreo de bit 6 a bit 7 => V=1
```

En consecuencia C=1, de donde se deduce que el valor con el que se ha efectuado la comparación (contenido en el registro C), es mayor que el contenido del acumulador.

Encontrará una lista de comandos lógicos junto a los comandos aritméticos de 8 bits (Capítulo 4.6).

PROGRAMA DEMO

```
A000 06FF       10        LD   B,&FF
A002 2100C0     20        LD   HL,&C000
A005 7E         30        LD   A,(HL)
A006 A8         40        XOR  B
A007 77         50        LD   (HL),A
A008 23         60        INC  HL
A009 3E00       70        LD   A,0
A00B BC         80        CP   H
A00C 20F7       90        JR   NZ,&A005
A00E C9        100        RET
```

Este programa invierte la pantalla completa en MODE 2.
LD B,&FF es la máscara con la cual se invierte el contenido del ACU, mediante el comando XOR B.

HL se carga con la dirección inicial de la pantalla, &C000 (LD HL,&C000). Comienza entonces el bucle de programa. LD A,(HL) lee un byte de la memoria de pantalla. Este valor se invierte a través de XOR B, grabándose a continuación nuevamente en la memoria de pantalla mediante LD (HL),A. Se incrementa entonces HL (INC HL) y se comprueba si HL se encuentra aún en el área de la memoria de pantalla.

El contenido de HL está comprendido entre &C000 y &FFFF. Si HL se incrementa nuevamente (&FFFF + 1), obtenemos el valor 0 para HL. En realidad el resultado sería &10000, pero dado que HL sólo puede almacenar números de 16 bits, se ignora el bit sobrante: HL=0.

Con el comando CP debe comprobarse si L ya contiene 0. Dado que CP siempre compara el contenido del ACU, el mismo debe cargarse previamente con 0 mediante LD A,0.

La comparación debe realizarse ahora con el High-byte de HL, si H=0 resulta entonces HL=0. A través del comando CP el flag Z se ve influído de la forma correspondiente.

El siguiente comando de bifurcación JR NZ,&A005 significa:
"Bifurca a dirección &A005, si Z diferente de 0 (non zero), de lo contrario ejecuta el comando siguiente".

Si HL=0 el programa finaliza con RET.
Las líneas DATA del cargador BASIC son las siguientes:

```
DATA &06,&FF,&21,&00,&C0,&7E,&A8,&77
DATA &23,&3E,&00,&BC,&20,&F7,&C9
```

Seleccione &A000 como dirección inicial, &A000+15-1=&A00E como dirección final y arranque con *CALL &A000* (en MODE 2).

Podemos también utilizar el comando CPL (Complementar ACU) en lugar de XOR B.
Conmute ahora a MODE 1 y pruebe la rutina. El resultado deseado no se obtiene. La razón la encuentran en la arquitectura de la memoria de pantalla. Como ya saben en MODE 2 bits y puntos activados se corresponden directamente. Por ello no pueden elegirse diferentes colores de letra en modalidad 2. En MODE 1 tenemos 4 colores disponibles. Dado que disponemos únicamente del área &C000-&FFFF y adicionalmente debe almacenarse la información correspondiente al color, en MODE 1 los cuatro bits superiores de cada byte se responsabilizan de la activación de un punto de doble espesor. Los bits inferiores determinan el color. Dado que invertimos los puntos y no los colores, debemos modificar la máscara de inversión. LD B,&FF (&FF=&X11111111) significa que se invierten todos los bits mediante XOR B. A través de &X11110000=&F0, se invierten los 4 bits superiores.

Para utilizar el programa también en MODE 1 debemos modificar el segundo valor de la línea 60 de DATAS, de &FF a &F0.
En MODE 0 únicamente los bits 6 y 7 se responsabilizan de los puntos. Realice con este objeto la necesaria modificación del programa.

4.8 COMANDOS DE ROTACION Y DESPLAZAMIENTO

¿Qué se entiende por desplazamiento de las cifras de un número?

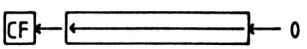
```
  4   3   2   1   0
10  10  10  10  10


    3   7   3   0

3   7   3   0   0    : Desplazamiento hacia la izquierda!

        3   7   3    : Desplazamiento hacia la derecha!
```

En el sistema decimal, un desplazamiento hacia la izquierda ocasiona una multiplicación por 10 (base del sistema decimal), y un desplazamiento hacia la derecha ocasiona una división entre 10. (Un desplazamiento de las cifras hacia hacia la izquierda significa un desplazamiento de la coma una posición hacia la derecha).

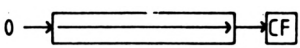
En el sistema binario, un desplazamiento hacia la izquierda o derecha significará respectivamente una multiplicación por 2 o una división entre 2. En BASIC no existe ningún equivalente directo para estos comandos (será pues la multiplicación o la división por o entre 2).

El Z80 posee 76 comandos de este tipo, de los cuales, la mayoría utilizan el direccionamiento implícito, indirecto o indexado. Existen varias formas de rotación y de desplazamiento. En primer lugar distinguiremos entre las operaciones rotar y desplazar.

Desplazar: Al correr hacia derecha e izquierda, el contenido del registro se mueve bit a bit en la dirección correspondiente. El bit que sobra al realizarse el desplazamiento es recogido en el flag de acarreo. El espacio que queda libre al otro lado del byte se rellena con cero.

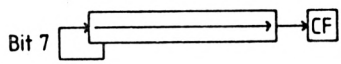


Desplazamiento aritmético a la izquierda



Desplazamiento lógico a la derecha

Al aplicar el comando SRL a números con signo se origina un error. El bit 7 del signo, se desplaza al lugar del bit 6. En la posición del bit 7 se inserta un 0. Por ello, un número negativo (bit 7=1) se convierte en positivo (bit 7=0). El comando SRA evita este error. En este comando, al bit incluído a la izquierda es idéntico al bit de signo. Es decir, será 0 cuando el bit izquierdo también lo sea (+) y 1 cuando el bit izquierdo sea 1 (-). Dado que este comando tiene en cuenta el significado aritmético del séptimo bit, se le denomina comando de desplazamiento aritmético, y no lógico.



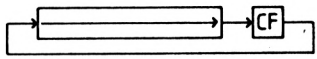
Desplazamiento aritmético a la derecha

Dib. 8

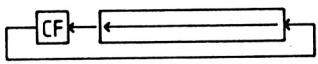
Rotar: Contrariamente a la operación de desplazamiento, en la rotación, el bit que se inserta es, o bien el bit de acarreo o el que se desplaza fuera del byte por el extremo opuesto.
En el Z80 existen 2 tipos de rotación:

Rotación de 8 bits (sin carry)
Rotación de 9 bits (con carry)

En una rotación de 9 bits hacia la derecha, se desplazan los 8 bits una posición hacia la derecha, el bit del extremo derecho se traslada al carry, y el bit que se inserta por el extremo izquierdo corresponde al antiguo contenido del carry (antes de la inserción del nuevo bit del extremo derecho). Dado que en este caso rotan los 8 bits del byte y el carry (el bit 9), a este tipo de rotación se la denomina: rotación de 9 bits.



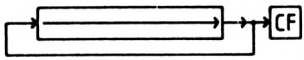
Rotación derecha a través del Carry



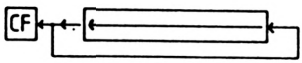
Rotación izquierda a través del Carry

Dib. 9 Rotación 9 bits

En la rotación de 8 bits, rotan únicamente los 8 bits del registro. En el carry sólo se almacena el bit desplazado. Sin embargo, el anterior contenido del carry no se desplaza con los bits restantes. El bit desplazado fuera del byte, se recoge nuevamente por el extremo opuesto del registro.



Rotación derecha



Rotación izquierda

Dib. 10 Rotación de 8 bits

Por otro lado existen asimismo 2 comandos especiales para la rotación de cifras (bloques de 4 bits) en formato BCD.

RLD y RRD (D: Digit: dígito). Estos comandos rotan 2 dígitos de la posición de memoria indicada por HL, y el dígito contenido en la mitad inferior del acumulador.

En general los comandos de rotación y desplazamiento tiene un código de operación de 2 bytes. El primer byte del Opcode es siempre &CB (en los comandos de direccionamiento indexado &CB corresponde al 2do. byte del opcode, ya que el primero es siempre &DD o bien &FD. Excepción: RRD/RLD comienzan con ED). Ya que los comandos de rotación son útiles en operaciones aritméticas, se establecieron otros 4 comandos, que operan por medio del ACU y poseen un Opcode de 1 byte. Estos comandos resultan por ello la mitad de largos y se ejecutan en la mitad de tiempo que los comandos estándar.

```
"Normal"            "Especial-ACU"

 RLC  A              RLCA
 RRC  A              RRCA
 RL   A              RLA
 RR   A              RRA
```

Los flags S y Z se influyen en la forma habitual mediante los comandos normales de rotación y de desplazamiento. El flag P/V indica la paridad. El contenido del carry es el correspondiente bit, que se desplaza fuera del byte. Los comandos especiales para el acumulador sólo modifican el flag C, mientras que los flags S, Z y P permanecen inalterados. Los comandos de rotación BCD RLD/RRD sólo influencian los flags S, Z y P en la forma antes citada, pero no influyen sobre el carry.

Ejemplo:

```
            SRL C          C: &36

        00110110            :&36
    0--> 0011011-->0 al Carry
        00011011            :registro C después de la ejecución
                  0         :carry después de la ejecución
```

SRL ocasiona una división entre 2: &36^2=&1B

```
    SRA (HL)        HL:&B100   posición de memoria &B100:&C2

        11000010            :&C2
        *1100001-->0        :Carry
        11000010   0        :CF:(HL) después de la ejecución=&E1
```

(* Bit 7 permanece en esta posición)

Como complemento a dos, significa:

&C2 = -62
&E1 = -31

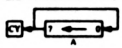
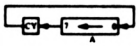
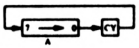
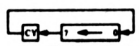
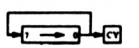
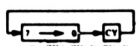
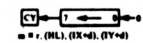
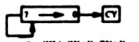
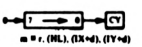
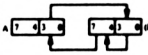
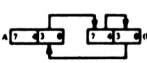
El comando SRA efectúa correctamente la división de los números con signo, de lo contrario SRL (HL) habría contenido &61 = 97 como resultado, que no es la mitad de -62 sino de 194, que corresponde a &C2 como número carente de signo.

```
RLC D
    D: &E4                  Carry=1

       &E4 =  &X11100100
Nuevo Carry <-- 11100100  <-- 1=Antiguo Carry
                11001001

Contenido de D después de la ejecución: &C5
Carry-F = 1
```

Como vemos &C5 no es el doble de &E4. El motivo del error está en que un bit ha rotado hacia el carry. Así pues, &1C5 debiera representar el doble de &E4, lo que no es correcto dado que el contenido antiguo del carry (=1) entró por el extremo opuesto. Entonces el doble de &E4 es &1C9-1=&1C8.

Si se han de rotar números compuestos de varios bytes, el bit desplazado del byte que se acaba de rotar, se introduce por RLC o RRC mediante el carry, hacia el interior del siguiente byte. (ver programa al final del capítulo)

```
RRA
    ACU: &76

       &76 =  &X011101110
              &X*01110110 -> carry
(* en este punto rota "hacia dentro" el anterior bit 0)

ACU:  &X00111011     CF=0

Contenido del acumulador : &3B
                           &3B*2 = &76
```

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA		‡	•	•	•	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		‡	•	•	•	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		‡	•	•	•	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		‡	•	•	•	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r		‡	‡	P	‡	0	0	11 001 011 00 000 r	2	2	8	Rotate left circular register r
RLC (HL)		‡	‡	P	‡	0	0	11 001 011 00 000 110	2	4	15	
RLC (IX+d)		‡	‡	P	‡	0	0	11 011 101 11 001 011 ← d → 00 000 110	4	6	23	
RLC (IY+d)		‡	‡	P	‡	0	0	11 111 101 11 001 011 ← d → 00 000 110	4	6	23	
RL m		‡	‡	P	‡	0	0	010				Instruction format and states are as shown for RLC,m. To form new OP-code replace 000 of RLC,m with shown code
RRC m		‡	‡	P	‡	0	0	001				
RR m		‡	‡	P	‡	0	0	011				
SLA m		‡	‡	P	‡	0	0	100				
SRA m		‡	‡	P	‡	0	0	101				
SRL m		‡	‡	P	‡	0	0	111				
RLD		•	‡	P	‡	0	0	11 101 101 01 101 111	2	5	18	Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD		•	‡	P	‡	0	0	11 101 101 01 100 111	2	5	18	

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ‡ = flag is affected according to the result of the operation.

APLICACION

La aplicación estándar para los comandos de rotación y desplazamiento se utiliza al realizar cálculos. En nuestro ejemplo queremos utilizarlos para el desplazamiento de la pantalla.
Con ayuda de los comandos de transferencia de bloques es posible desplazar la pantalla en dirección horizontal, carácter por carácter. Con los nuevos comandos podemos realizar un desplazamiento bit por bit.

El listado de ensamblador:

```
A000 97       10   SUB  A
A001 2100C0   20   LD   HL,&C000
A004 CB3E     30   SRL  (HL)
A006 23       40   INC  HL
A007 BC       50   CP   H
A008 20FA     60   JR   NZ,&A004
A00A C9       70   RET
```

Reconocerán la estructura básica del bucle, con el cual se incrementa HL desde &C000 hasta &FFFF.
El primer comando es nuevo.

SUB A,A se utiliza en lugar del comando LD A,0. SUB A,A borra el ACU. Este comando es más rápido porque está direccionado de forma implícita.

LD A,0 se direcciona de forma inmediata, es decir que los datos (0!) deben leerse adicionalmente. Ahora lo fundamental del programa:

SRL (HL)

Dado que HL debe recorrer el área completa de direcciones, hemos elegido el direccionamiento indirecto. SRL desplaza los 8 bits de cada byte de la pantalla en una posición hacia la derecha.

Modifique el programa en líneas DATA con ayuda del listado de ensamblador y cárguelo con el cargador BASIC a partir de dirección &A000. El programa desplaza hacia la derecha cada carácter de la pantalla. Dado que ignoramos el bit desbordado por la derecha los caracteres tienen un bit truncado por la derecha.
Introduzca lo siguiente:

```
FOR I=1 TO 8: CALL &A000: NEXT
```

Con este comando se borra la pantalla. Los caracteres desaparecen bit por bit por la derecha, porque en el comando SRL el bit que entra por la izquierda tiene valor cero (ningún punto).
Reemplazamos SRL (HL) por SLA (HL).

El código para este comando es: &CB,&25. Introduzca las líneas DATA para el quinto elemento &25 (en lugar de &3E) y cargue el programa nuevamente con RUN. Este programa tiene efecto similar, pero el desplazamiento se realiza hacia la izquierda.
Intente asimismo SRA (HL), código &CB,&2E. El quinto byte en las líneas DATA es entonces &2E. Después de que el bucle FOR-NEXT se ejecuta 8 veces, se forma en la pantalla una configuración extraña. Esto se produce porque el comando SRA, deja el bit 7 en su posición. Después de la ejecución múltiple del comando todos los bits obtienen el antiguo valor del bit 7.

De la letra R del mensaje 'READY' obtenemos 2 líneas horizontales (después de ejecutar 8 veces). La razón para ello es la configuración de bits de esta letra.

```
          76543210      Número de Bit
         1******
         2 **  **
         3 **  **
Línea    4 *****
         5 ** **
         6 **  **
         7***   *
         8
```

Cada carácter se representa de esta manera en una matriz de 8X8. En el caso de R, el bit 7 solamente está activado en línea 1 y línea 7a. Si ejecuta las ocho llamadas de programa máquina consecutivamente podrá observar que la R se desplaza dejando una raya en líneas 1 y 7.

La e del mensaje Ready, desaparece totalmente porque en este carácter no hay ningún bit 7 activado. De la a queda una raya en línea 6, de la 'd' en líneas 4, 5 y 6 y de la 'y' ninguna raya.

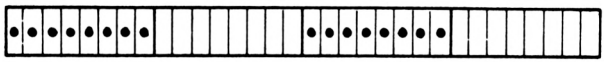
A través de estos resultados esperamos se aclare la razón por la cual el comando SRA se denomina como aritmético y el comando SRL en cambio como lógico. Intenten introducir asimismo los restantes comandos en su programa.

RRC tiene el código &CB,&DE; RLC tiene el código &CB,&06. Modifique el cargador y ejecute el programa 8 veces con el bucle FOR-NEXT. Podemos observar claramente por qué estos comandos se denominan comandos de rotación. Cada carácter rota, es decir, los bits que se desbordan por la derecha/izquierda (para RRC/RLC) se integran por el lado opuesto. Después de la octava ejecución, la pantalla se encuentra nuevamente en el estado inicial.

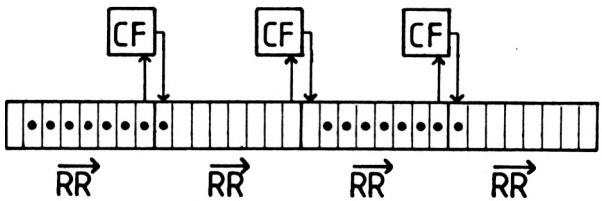
Nos quedan aún los comandos de rotación de 9 bits, RL (Código &CB,&16) y RR (código &CD,&1E).

Con la llamada del programa a través de RR, la pantalla obtiene la muestra de rayas. Después de cada llamada consecutiva, se observa que las líneas se vuelven más gruesas, hasta que finalmente la pantalla queda totalmente en blanco después de 8 llamadas. Esto en ningún caso es el resultado esperado. Mediante la rotación de 8 bits, el contenido de la pantalla debería haberse desplazado en un bit en la dirección correspondiente.

CUATRO BYTES DE PANTALLA ANTES DE LA EJECUCION



DESPUES DE LA EJECUCION REPETIDA DE RR



Dib. 11 APLICACION ROTACION 9 BITS

El contenido debería desplazarse en un bit hacia la derecha, porque el bit rotado se almacena en el carry y a continuación vuelve a rotarse al bit siguiente.
Dado que no hemos obtenido el resultado esperado, por lo visto hay un error en el programa.
Intenten encontrar este error y busquen una solución.
(Consejo: observen la influencia de los flags!)

Como siempre está activado el primer bit de un carácter después de cada ejecución (= las "rayas" de la pantalla) y este bit se extrae del carry flag, el carry siempre ha contenido 1. De esta manera no correspondía al último bit del byte anterior. ¿Cómo es posible?

Veamos los restantes comandos del programa. Después de la rotación tenemos el comando INC HL. Los comandos de contador de 16 bits, no influencian los flags. Ahora tenemos CP H. ¡Aquí tenemos el error buscado!.

El objeto del comando CP es activar flags, influyendo el carry en cada paso del bucle. Dado que H es mayor que A (A=0), el carry se activa cada vez (excepto en el primer paso). El carry flag activado se rota mediante RR en la pantalla y ésta se queda en blanco.

Como solución a este problema se nos presentan 2 posibilidades:

1. - Almacenar temporalmente los flags antes de cada comando CP
2. - Evitar la alteración de flags

Con referencia al punto 1: con los comandos de pila es posible salvar el registro F en la pila (inmediatamente a continuación del comando de rotación), extrayéndolo nuevamente de la pila (directamente antes del comando de rotación). Después de RR debe introducirse PUSH AF (almacenar en la pila) y delante de RR el comando POP AF (extraer de la pila). Hemos de tener en cuenta además no desordenar la pila.

El primer comando de pila en nuestro programa sería como se ha descrito arriba POP AF, pero esto sería falso, porque mediante el mismo se extraen datos de la pila que no se encuentran aún en la misma. De esta forma extraíamos la dirección de retorno.

El programa bifurcaría a una dirección errónea, al intentar un retorno. Por ello debemos insertar PUSH AF una vez antes del bucle y después del bucle (delante de RET) POP AF.

Tengan en cuenta el orden correcto al utilizar PUSH y POP. Después de estas mejoras, nuestro programa se representa de la siguiente forma:

```
A000 97          10   SUB  A
A001 F5          15   PUSH AF
A002 2100C0      20   LD   HL,&C000
A005 F1          25   POP  AF
A006 CB1E        30   RR   HL
A008 F5          35   PUSH AF
A009 23          40   INC  HL
A00A BC          50   CP   H
A00B 20FB        60   JR   NZ,&A005
A00D F1          65   POP  AF
A00E C9          70   RET
```

Si un programa BASIC necesita un minuto para este desplazamiento de un bit, este programa máquina es casi demasiado lento. Debido a los 2 comandos de pila en el bucle que se ejecuta 16000 veces, el programa se prolonga innecesariamente. Para evitar esta desventaja referente a la velocidad de ejecución, les indicamos ahora la segunda posibilidad.

Con referencia al punto 2: para que funcione el comando JR NZ dejando el carry inalterado, necesitamos un comando que influya el flag Z pero no el flag C. Esto nos ofrecen los comandos de contador de 8 bits. Para incrementar la pareja de registros HL, son necesarios 2 comandos de contador de 8 bits. En primer lugar incrementamos el Low-byte. Si L después del incremento no es 0, el bucle se repite.
Si L en cambio es 0 debe incrementarse H en 1.
Ejemplo:

	H=&C0	L=&FE	HL=&C0FE
después del incremento:	H=&C0	L=&FF	HL=&C0FF
después del incremento:	H=&C1	L=&0	HL=&C100

La nueva porción de programa:

```
INC  L
JR   NZ,Dirección
INC  H
JR   NZ,Dirección
RET
```

Además podemos omitir el comando SUB A, porque ya no utilizamos el ACU.
Listado de Assembler:

```
A000 97       10   SUB  A
A001 F5       15   PUSH AF
A002 2100C0   20   LD   HL,&C000
A005 F1       25   POP  AF
A006 CB1E     30   RR   HL
A008 F5       35   PUSH AF
A009 23       40   INC  HL
A00A BC       50   CP   H
A00B 20FB     60   JR   NZ,&A005
A00D F1       65   POP  AF
A00E C9       70   RET
```

Active el programa en líneas DATA con:

```
60 DATA &21,&00,&C0,&CB,&1E,&26,&20,&FB
70 DATA &24,&20,&F8,&C9
```

Cargue el programa con el cargador BASIC y pruébelo con RUN.
El comando RRD: Altera &CB (Byte 4) en &ED y &1E en &67. Después de cargado, ejecute el programa con diferentes 4 bits (1 cifra BCD).

Compruébelo son el siguiente programa BASIC:

```
10 FOR K=1 TO 4
20 FOR I=0 TO 11
30 LOCATE (K-1)*8+1,12-I:PRINT "HOLA"
35 LOCATE (K-1)*8+1,12+I:PRINT "HOLA"
40 FOR J=0 TO K
50 CALL &A000
60 NEXT J
70 NEXT I
80 NEXT K
```

4.9 COMANDOS DE MANIPULACION DE BITS

En el capítulo 4.7 se mostró cómo pueden utilizarse las operaciones lógicas para activar o desactivar bits individuales o grupos de bits en el ACU. Sin embargo, es útil poder activar o desactivar un bit individual en un registro o posición de memoria deseados. Debido a que ello supone necesariamente la ejecución de un elevado número de comandos, la mayoría de las CPUs disponen para ello de muy pocos o ningún comando. A este respecto, el Z80 posee un "surtido" de 120 comandos de manipulación de bits que incluye comandos de comprobación de bits.

Los comandos de comprobación de bits verifican si un bit determinado se encuentra en estado de "activado" o "desactivado" en un registro o en una posición de memoria. Según el resultado de la comprobación se activa o desactiva a su vez el Zero-Flag. El carry no se influye, y los flags S y P/V quedan indeterminados después de la ejecución (!). Los dos comandos para activar (SET) o desactivar (RES) bits, no ejercen ninguna influencia sobre los flags.

Todos los comandos de manipulación de bits comienzan con el código de operación &CB (excepto los comandos de direccionamiento indexado). El segundo código de operación resulta del número del bit y del código del registro.
Para acceder al Byte afectado se dispone de los siguientes direccionamientos:

- Implícito : registros A, B, C, D, E, H, L
- Indirecto : (HL)
- Indexado : (IX+d) y (IY+d)

Formato:

```
BIT b,reg       BIT b,(HL)      BIT b,(XY+des)
RES b,reg       RES b,(HL)      RES b,(XY+des)
SET b,reg       SET b,(HL)      SET b,(XY+des)
```

b = número de bit

El número de bit b, se codifica de la siguiente forma:

```
0 - 000        4 - 100
1 - 001        5 - 101
2 - 010        6 - 110
3 - 011        7 - 111
```

Todos estos comandos también se consideran como comandos de direccionamiento de bits, debido a que el correspondiente bit se indica en el código de operación.

Ejemplos:

```
BIT 6,B          B:&33

      &X00110011  = &33
         *
        76543210  - número de bits
```

*: el bit número 6 es 0.
Dado que el bit 6=0, el flag Z se activa a 1, después de la ejecución:

```
 B=&33           Flags: S  Z  V  C
                        U  1  U      U= Flags S y V desconocidos

RES  1,(HL)      HL:&A975
                    &1975 = &23

                &X00100011 = &23
                       *
                  76543210 - número de bit
```

*: El bit número 1 se desactiva

```
                &X00100001 = &21
```

Posición de memoria &A975 después de la ejecución: &21

Flags: S Z V C
- ninguno afectado

SET 7,C C: &7F

&X01111111 = &7F
*
76543210 - número de bit

*: El bit número 7 se activa

&X11111111 = &FF

El flag-C después de la ejecución es &FF

Flags: S Z V C
- ninguno afectado

Analogías en BASIC

Intentemos ejecutar el comando SET b,A en BASIC:
El bit b debe activarse. Con el comando OR tenemos la posibilidad de activar determinados bits. El bit b tiene el valor de posición 2^b. Obtenemos entonces:

SET b,A BASIC: A=A OR (2^b)

Con el comando RES ocurre algo similar:

RES b,A BASIC: A=A AND (255-2^b)

Los comandos especiales SCF y CCF:

Dado que el bit 0 se utiliza con bastante frecuencia en el registro F (el carry), existen para ello 2 comandos especiales.

SCF inserta en el carry el valor 1.
CCF complementa el valor del carry-F, es decir, de C=0 lo convierte en C=1 y viceversa.
Estos son los únicos comandos con los que podemos afectar directamente los flags.
El carry puede borrarse también con los comandos lógicos.
Los comandos CCF y SCF se encuentran en la lista correspondiente de comandos al final del capítulo 4.11.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
BIT b, r	$Z \leftarrow \overline{r_b}$	•	‡	X	X	0	1	11 001 011 01 b r	2	2	8	
BIT b, (HL)	$Z \leftarrow \overline{(HL)_b}$	•	‡	X	X	0	1	11 001 011 01 b 110	2	3	12	
BIT b, (IX+d)	$Z \leftarrow \overline{(IX+d)_b}$	•	‡	X	X	0	1	11 011 101 11 001 011 ← d → 01 b 110	4	5	20	
BIT b, (IY+d)	$Z \leftarrow \overline{(IY+d)_b}$	•	‡	X	X	0	1	11 111 101 11 001 011 ← d → 01 b 110	4	5	20	
SET b, r	$r_b \leftarrow 1$	•	•	•	•	•	•	11 001 011 [11] b r	2	2	8	
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	•	•	•	•	11 001 011 [11] b 110	2	4	15	
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	•	•	•	•	11 011 101 11 001 011 ← d → [11] b 110	4	6	23	
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	•	•	•	•	11 111 101 11 001 011 ← d → [11] b 110	4	6	23	
RES b, m	$s_b \leftarrow 0$ m≡r, (HL), (IX+d), (IY+d)							[10]				To form new OP-code replace [11] of SET b,m with [10]. Flags and time states for SET instruction

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

b	Bit Tested
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

PROGRAMA PARA LA MANIPULACION DE BITS

Confeccione un programa que llene la pantalla de rayas de un punto de ancho y posicionadas en el centro de un carácter, en modalidad 2. Utilice nuevamente el bucle del programa anterior.

Listado de ensamblador

```
A000 2100C0     10   LD  HL,&C000
A003 CBDE       20   SET 3,(HL)
A005 2C         30   INC L
A006 20FB       40   JR  NZ,&A003
A008 24         50   INC H
A009 20F8       60   JR  NZ,&A003
A00B C9         70   RET
```

Programa BASIC:

```
5 MEMORY &9FFF
10 FOR i=&A000 TO &A00B
20 READ a
30 POKE i,a
40 NEXT i
50 MODE 2
60 CALL &A000
70 END
100 DATA &21,&00,&C0,&CB,&DE,&2C,&20,&FB
110 DATA &24,&20,&FB,&C9
```

En lugar de SET 3,(HL) puede utilizarse SET 4,(HL) código: &CB,&E6. En las líneas DATA ha de reemplazar el &DE por &E6.

4.10 BIFURCACIONES

Gran parte de las bifurcaciones son condicionadas, es decir que dependen del estado adoptado por los flags; por ello describiremos en este apartado nuevamente, de forma resumida, el papel que juega cada flag.
Los flags H y N se utilizan en la aritmética BCD, no pueden comprobarse. Los demás flags (C, P/V, Z y S) pueden comprobarse.

Carry-Flag (flag de acarreo)

El flag de acarreo tiene 2 funciones:

- Indica si se ha producido un acarreo en una adición o sustracción.
- Los comandos SRL, SRA, SLA, RR, RL, RRC, RLC, RRA, RLA, RRCA, y RLCA utilizan el carry como noveno bit.

Una excepción la constituyen los comandos de rotación RLD, RRD en BCD. Estos comandos no afectan al carry.
Los comandos lógicos AND, OR y XOR ponen el carry siempre a ceros, y pueden utilizarse para desactivar el carry. Los comandos siguientes producen una alteración del carry:

NEG: Se activa el flag C si antes de la ejecución del comando, A tenía el valor 0.

DAA: Este comando afecta al carry de forma compleja. Se utiliza para la realización de cálculos en formato BCD.

SCF: Set Carry Flag.
Este comando activa el carry en 1.

CCF: Complement Carry Flag
Este comando complementa el carry.

¡Los restantes comandos no afectan al carry!

Parity/Overflow (Paridad/Desbordamiento -P/V)

Este flag tiene varias funciones, dependiendo del comando ejecutado.

- Desbordamiento

En las operaciones aritméticas ADD de 8 bits, ADC, SUB, SBC, INC de 8 bits, DEC de 8 bits, NEG y CP, se indica un desbordamiento. Esto significa que el signo de un número ha sido modificado con error.
Excepciones: ADD de 16 bits, INC de 16 bits, DEC de 16 bits.
Estos comandos no afectan sobre el flag V.

- Paridad

En los comandos INPUT (IN), en los de rotación y en los de desplazamiento RR, RL, RRC, RLC, RLD, RRD, SLA, SRA y SRL, en los comandos lógicos AND, OR, XOR y en DAA, este flag se utiliza como flag de paridad P. P tiene valor 1 cuando la cantidad de '1' de un byte es par, y tiene valor 0 cuando el número de bits activados es impar.
Excepción: ¡RLA, RRA, RLCA, RRCA no afectan P!

- En los comandos de bloque LLD, LDI, CPD, CPI, CPDR y CPIR, se desactiva P/V, en caso de que BC=0 (BC representa el registro contador), en caso contrario, se activa.
  Por esta razón, P/V siempre se desactiva con LDDR y LDIR.

- Flag de interrupción.

Con LD A,I y LD A,R el flag P/V toma el valor del Interrupt-Enable-Flip-Flops (IFF), activándose a 1 cuando admite interrupciones enmascarables y desactivándose con 0 cuando no acepta las mismas interrupciones.

ATENCION: El comando BIT y todos los comandos de Entrada/Salida de bloques, activan este flag modificando su contenido en algunos casos. Otros comandos no influyen sobre el mismo.

Zero-Flag (Cero,Z)

El flag Z indica si el valor de un byte es nulo, es decir si el valor es 0, al flag Z se activa, de lo contrario se desactiva.
En los comandos de comparación, Z se activa (1) cuando existe una igualdad, de lo contrario se desactiva (0).
En el comando BIT, el Flag Z se activa cuando el bit comprobado es 0, de lo contrario se desactiva.
Los siguientes comandos influyen sobre el flag Z:

```
Aritméticos             : ADD,ADC,SUB,SBC,INC,DEC,NEG,DAA
            ATENCION    : ADD de 16 bits no influye el flag!
Comparación             : CP: Z=1 si igualdad, Z=0 no hay igualdad
Bit                     : BIT
Rotación/desplazamiento : RR,RL,RRC,RLC,SRL,SRA,SLA,RLD,RRD
              ATENCION  : RRA,RLA,RRCA,RLCA no influyen el flag!
Búsqueda/Bloque         : CPI,CPIR,CPD,CPDR: Z=1 por igualdad
Entrada                 : IN
Comandos de carga       : LD A,I o bien LD A,R
```

Entrada/Salida de bloques: INI, IND, OUTI, OUTD activan a Z (1) si tras su ejecución B=0; y INIR, INDR, OTIR, OTDR activan Z en todos los casos.
¡Todos los restantes comandos no afectan sobre Z!

SIGN-Flag (Signo, S)

El flag de signo contiene el valor del bit más alto de un byte. Este bit, corresponde al signo en la aritmética con signo.

Los siguientes comandos afectan el flag S:

Todos los comandos aritméticos o lógicos:
ADD,ADC,SUB,SBC,INC,DEC,NEG,DAA,AND,OR,XOR,CP

Los comandos de rotación y desplazamiento:
RL,RR,RLC,RRC,SRL,SRA,SLA,RLD,RRD.

Los comandos de búsqueda de bloque: CPD,CPI,CPDR,CPIR.

Los comandos de entrada IN y los comandos de carga LD A,I y LD A,R.

ATENCION: los comandos: ADD de 16 bits, INC de 16 bits, DEC de 16-bits, RLA,RRA,RLCA,RRCA no afectan este flag!!.

El comando BIT y los comandos de entrada/salida de bloques, INI, IND,OUTI,OUTD,INIR,INDR,OTIR,OTDR alteran el estado del flag S, dejándolo en un estado indeterminado.
En el anexo encontrará información sobre la influencia de los flags para cada comando.

En el Z80 existen 5 modalidades diferentes de bifurcación.

- Bifurcaciones dentro del programa principal (JUMP) que corresponden al comando BASIC 'GOTO'.
- Bifurcaciones a subprogramas (CALL y RET), que corresponden a los comandos BASIC 'GOSUB' y 'RETURN'.
- Bifurcaciones relativas (JUMP RELATIVE) semejantes al comando BASIC 'FOR-NEXT'.
- Comandos RESTART (RST), que realizan una bifurcación a una dirección previamente determinada. El comando RST no posee análogo BASIC.
- Bifurcaciones de interrupción (ver comandos de control).

Los tres primeros tipos de bifurcación existen como incondicionales y condicionales en el Z80, dependiendo del estado de un flag.
En los comandos de bifurcación condicional, se realiza la bifurcación según el estado de los flags Z, C, P/V y S. Cada flag puede comprobarse por el valor 0 ó 1.
En lenguaje ensamblador, se utilizan las abreviaciones siguientes:

Z	= Bifurcación si valor es nulo	(Z=1)
NZ	= Bifurcación si valor es NO nulo	(Z=0)
C	= Bifurcación si existe acarreo	(C=1)
NC	= Bifurcación si NO existe acarreo	(C=0)
PO	= Bifurcación si paridad es IMPAR	(P/V=0)
PE	= Bifurcación si paridad es PAR	(P/V=1)
P	= Bifurcación si POSITIVO (+)	(S=0)
M	= Bifurcación si NEGATIVO (-)	(S=1)

El Z80 reconoce también un comando de control de bucles, que decrementa el registro B y ejecuta luego una bifurcación relativa, toda vez que el registro B<>0. Este comando se codifica DJNZ (Decrement Jump Non Zero).

JUMP

Las bifurcaciones dentro del programa principal se ejecutan con el comando JP. La dirección de bifurcación puede indicarse por medio de 2 modalidades:

Direccionamiento absoluto:

Formato:

JP dirección o bien JP cond,dir

(cond representa una condición (Condition), es decir, para Z,NZ,C,NC,PO,PE,P ó M)

JP dirección - Bifurcación incondicional a la dirección dada.
JP Cond,dirección - Si la condición se cumple, bifurca a la dirección dada. Si la condición no se cumple, se ejecuta el siguiente comando.

Analogías BASIC:

```
JP dir          BASIC: GOTO Número de línea
JP NZ,dir       BASIC: IF Z=0 THEN GOTO número de línea
JP Z,dir        BASIC: IF Z=1 THEN GOTO Número de línea
```

El procesador realiza una bifurcación leyendo la dirección dada en el PC. Entonces, se lee y ejecuta en esta posición el siguiente código de comando.
En el direccionamiento absoluto, al código de operaciones de 1-byte le sigue la correspondiente dirección de bifurcación en la forma Low-byte High-byte. Dado que todos los comandos de 3 bytes son relativamente lentos, se crearon los saltos relativos debido a que éstos son comandos de únicamente 2 bytes. Las bifurcaciones de direccionamiento indirecto tienen un código de operaciones de 1 byte.

Direccionamiento indirecto.

Formato:

JP (X)

X: HL, IX o IY

JP (X) bifurca a la dirección indicada por el registro X.

CALL/RET

Ya hemos descrito cómo se almacenan o leen las direcciones de retorno CALL y RET. Una llamada a un subprograma puede ser condicional o incondicional. La dirección de bifurcación

(dirección inicial del subprograma), se indica por direccionamiento absoluto.

Formato:

CALL dir o bien CALL cond,dir

En la ejecución del comando, todas las operaciones necesarias se realizan en el stack, en SP y en PC. El funcionamiento de este comando es el siguiente:
Una vez completada la lectura del comando, PC indica el siguiente comando a ejecutar. A continuación se realizan las operaciones:

```
(SP-1)=PC    - (High byte)
(SP-2)=PC    - (Low byte)
SP=SP-2
PC=dir
```

El siguiente comando se lee de la dirección que indica el PC. Para finalizar un subprograma se coloca un comando RET. El RETURN puede ser asimismo condicional o incondicional.

Formato:

RET o bien RET cond

Al ejecutar el comando RET, ocurre lo siguiente:

```
PC - (Low byte) = (SP)
PC - (High byte) = (SP+1)
SP=SP+2
```

La ejecución del programa se continúa en la dirección recogida del stack.
El comando RET, contrariamente al comando CALL, sólo tiene 1 byte de longitud. En el comando CALL debe indicarse la dirección de 16 bits, es decir que este comando tiene una longitud de 3 bytes.

Existen 2 bifurcaciones especiales de retorno: RETI y RETN, que se describen en el capítulo referente a los comandos de control.

RESTART

Este tipo de comando de salto tiene una longitud de 1 byte y por esta razón se ejecuta a mayor velocidad que los restantes comandos de bifurcación (sin tener en cuenta el comando RET). El comando RST, que en lo sucesivo denominaremos Restart, ocasiona un salto de un subprograma a una dirección localizada en la parte inferior de la memoria. Existen 8 comandos Restart. Las direcciones de bifurcación del restart son 0, &8, &10, &18, &20, &28, &30 y &38.

Formato:

RST dir
dir: una de las direcciones de 8 bits antes mencionadas.

Dado que restart es el comando de bifurcación más rápido, en la porción inferior de la memoria (0-&40) se localizan importantes rutinas o bifurcaciones a estas rutinas, que se utilizan frecuentemente. Más adelante estudiaremos la función exacta de cada comando restart.

JUMP RELATIVE

Las bifurcaciones relativas, realizan el salto en relación a la dirección actual. La distancia de salto debe por tanto indicarse. El primer byte es el código de operaciones, el segundo indica la distancia con signo (en complemento a dos). Este proceso se caracteriza como direccionamiento relativo. En este caso, la distancia recibe la denominación de offset de desplazamiento.

Formato:

JR e o bien JR cond,e
e: Offset
cond: Z,NC,C,NZ

Las bifurcaciones relativas condicionales sólo son posibles mediante los flags C y F. ¿Cómo se calcula el offset?

Observemos el último programa del capítulo 4.9. En la dirección &A006 se encuentra el comando JR. La dirección de destino es el comando SET 3,(HL) en dirección &A003. De esta forma, la diferencia es entonces &A006 hasta &A003 = 3. Como se trata de una bifurcación hacia atrás (la dirección de destino es menor que la dirección de partida), el offset es -3. Para conservar el segundo byte del comando deberemos restar 2 del offset.

¿Porqué es necesaria esta resta?

El procesador siempre lee en primer lugar el comando completo, en este caso el código de operaciones (byte 1) y el offset (byte 2). Tras cada lectura, PC se incrementa en 1. Una vez leído el comando por completo, el PC se encontrará en la dirección de inicio del siguiente comando. Como consecuencia de ello, el indicador de programa será mayor en 2 que la dirección del comando de salto. El Z80 ejecuta el salto sumando la distancia al PC. Por este motivo deberemos tener en cuenta el aumento del PC en 2. En un "Salto hacia atrás" será necesario saltar por encima de estos 2 bytes. La distancia que ha de memorizarse se calcula a partir de:

-3-2=-5 = &FB en complemento a 2.

Este byte se indica en el listado ensamblador en la dirección &A007, a continuación del Opcode en la dirección &A006. En lenguaje ensamblador no se indica esta diferencia de 2. El comando es JR $-3 ($ para la dirección actual del comando). El Assembler calcula automáticamente la diferencia de salto y realiza la resta de 2 y la conversión al complemento a 2. Aunque en el comando ensamblador se indica el comando en 16 bits, aquí se trata de un salto relativo. Teniendo en cuenta la resta, es posible efectuar bifurcaciones relativas a la dirección actual desde +129 hasta -126 bytes.

Resumamos el método de cálculo del byte de offset:

El comando de salto se encuentra en la dirección DIR.
La dirección del salto se encuentra en la dirección DIRZ
Offset = DIRZ - DIR
Byte a almacenar: (Offset - 2) en complemento a 2.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	•	•	•	•	11 000 011 ← n → ← n →	3	3	10	
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	•	•	•	•	11 cc 010 ← n → ← n →	3	3	10	
JR e	PC ← PC + e	•	•	•	•	•	•	00 011 000 ← e-2 →	2	3	12	
JR C, e	If C = 0, continue	•	•	•	•	•	•	00 111 000 ← e-2 →	2	2	7	If condition not met
	If C = 1, PC ← PC+e								2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	•	•	•	•	00 110 000 ← e-2 →	2	2	7	If condition not met
	If C = 0, PC ← PC + e								2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	•	•	•	•	00 101 000 ← e-2 →	2	2	7	If condition not met
	If Z = 1, PC ← PC + e								2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	•	•	•	•	00 100 000 ← e-2 →	2	2	7	If condition not met
	If Z = 0, PC ← PC + e								2	3	12	If condition met
JP (HL)	PC ← HL	•	•	•	•	•	•	11 101 001	1	1	4	
JP (IX)	PC ← IX	•	•	•	•	•	•	11 011 101 11 101 001	2	2	8	
JP (IY)	PC ← IY	•	•	•	•	•	•	11 111 101 11 101 001	2	2	8	
DJNZ,e	B ← B-1 If B = 0, continue	•	•	•	•	•	•	00 010 000 ← e-2 →	2	2	8	If B = 0
	If B ≠ 0, PC ← PC + e								2	3	13	If B ≠ 0

cc	Condition
000	NZ non zero
001	Z zero
010	NC non carry
011	C carry
100	PO parity odd
101	PE parity even
110	P sign positive
111	M sign negative

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <-126, 129>

e-2 in the op-code provides an effective address of pc +e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	$(SP-1)\leftarrow PC_H$ $(SP-2)\leftarrow PC_L$ $PC\leftarrow nn$	•	•	•	•	•	•	11 001 101 ← n → ← n →	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	•	•	•	•	11 cc 100 ← n → ← n →	3 3	3 5	10 17	If cc is false If cc is true
RET	$PC_L\leftarrow(SP)$ $PC_H\leftarrow(SP+1)$	•	•	•	•	•	•	11 001 001	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	•	•	•	•	11 cc 000	1 1	1 3	5 11	If cc is false If cc is true
RETI	Return from interrupt	•	•	•	•	•	•	11 101 101 01 001 101	2	4	14	
RETN	Return from non maskable interrupt	•	•	•	•	•	•	11 101 101 01 000 101	2	4	14	
RST p	$(SP-1)\leftarrow PC_H$ $(SP-2)\leftarrow PC_L$ $PC_H\leftarrow 0$ $PC_L\leftarrow P$	•	•	•	•	•	•	11 t 111	1	3	11	

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
‡ = flag is affected according to the result of the operation.

Aplicación:

En el listado de ensamblador (Capítulo 4.9) existe un salto relativo en la dirección &A009. El destino de la bifurcación es &A003. Calcule el offset-byte y compare su resultado con el listado de ensamblador.

Ahora hemos tratado los comandos de mayor relevancia. Hagamos referencia a un programa del capítulo sobre comandos de carga.
La tarea del programa era rellenar el cuadrado superior izquierdo de la pantalla. Esto puede realizarse mejor con un bucle.

En BASIC obtenemos:

```
10 FOR I=&C000 TO &FFFF STEP &800
20 POKE I,&FF
30 NEXT
```

Para formular nuevamente el programa en lenguaje máquina, cargamos la pareja de registros HL con la dirección inicial &C000. Para traducir la instrucción STEP &800 se carga DE con &800 realizando a continuación una adición de 16 bits. Si después de la adición queda activado el carry, hemos llegado al final del programa. Confeccione con la ayuda de estas instrucciones el programa máquina correspondiente.

Solución:

```
A000 2100C0      10    LD  HL,&C000
A003 110008      20    LD  DE,&800
A006 36FF        30    LD  (HL),&FF
A008 19          40    ADD HL,DE
A009 30FB        50    JR  NC,&A006
A00B C9          60    RET
```

Modifique ahora este programa de manera que no se rellene el cuadrado sino que se represente el carácter de forma inversa.

Solución:

```
A000 2100C0      10    LD  HL,&C000
A003 110008      20    LD  DE,&800
A006 7E          30    LD  A,(HL)
A007 2F          40    CPL
A008 77          50    LD  (HL),A
A009 19          60    ADD HL,DE
A00A 30FA        70    JR  NC,&A006
A00C C9          80    RET
```

En lugar de la inversión del byte con CPL podemos utilizar naturalmente el comando XOR &FF. Este en cambio es más largo (2 bytes) y por lo mismo más lento.

El comando DJNZ facilita una programación de bucle más adecuada. El offset se indica como en el comando JR, en el segundo byte. El registro B se utiliza como contador. Para realizar 8 repeticiones de bucle debe cargarse el registro B con 8, porque si B=0 ya no se efectúa la bifurcación. El comando JR es reemplazado por DJNZ y al comienzo B se carga con el valor 8.

Listado ensamblador:

```
A000 0608        10    LD  B,8
A002 2100C0      20    LD  HL,&C000
A005 110008      30    LD  DE,&800
A008 7E          40    LD  A,(HL)
A009 2F          50    CPL
A00A 77          60    LD  (HL),A
A00B 19          70    ADD HL,DE
A00C 10FA        80    DJNZ &A008
A00E C9          90    RET
```

4.11 COMANDOS DE CONTROL

Los comandos de control modifican o afectan sobre el modo operativo o el funcionamiento de la CPU.

El comando NOP

NOP significa NO Operation. Así pues, el comando NOP, carece de función. Aunque parezca paradójico, tiene su justificación. Por un lado, el comando NOP puede utilizarse para un retardo intencional; en el ordenador CPC un comando NOP necesita aproximadamente un microsegundo (10^-6 seg.) y por otra parte este comando puede ser utilizado como reserva de espacio en determinados programas. De este modo resulta más sencilla la búsqueda o rectificación de errores. El código de operación de este comando es &00, es decir que si el programa se ejecuta por descuido en un área borrada, no corre el riesgo de resultar dañado o modificado, ya que NOP no ocasiona modificaciones.

El comando STOP

Este comando interrumpe las operaciones de la CPU durante el tiempo necesario hasta que se ejecute un Reset o un Interrupt.

Comandos de control Interrupt.

Una interrupción sirve principalmente para la ejecución de procesos importantes en el ordenador. Una interrupción es el aviso que un componente del mismo emite sobre la entrada a un estado determinado, p. ej. la espera de los periféricos de entrada/salida a la entrada de datos. La CPU se encarga de tratar esos avisos según su importancia. Un programa que se ejecute normalmente puede ser interrumpido por un Interrupt. Estas interrupciones juegan un importante papel en la entrada y salida de datos. Los ordenadores CPC ofrecen la posibilidad de programar interrupciones a partir

del BASIC (EVERY-AFTER). En estos comandos, el interrupt es activado por el reloj interno del procesador. Si se requiere una interrupción, el programa bifurca a la dirección inicial de un subprograma que ejecuta las acciones correspondientes de la respectiva interrupción. De este programa servicio interrupción puede bifurcarse nuevamente al programa principal mediante el comando RETI (Return Interrupt).
Se distingue entre interrupciones con máscara e interrupciones sin máscara. Las últimas se ejecutan bajo cualquier condición y tienen máxima prioridad. El comando RETN permite retornar desde un Non-Mascarable-Interrupt (NMI).

DI Disable Interrupt y EI Enable Interrupt

El comando DI provoca un bloqueo temporal de las interrupciones con máscara, desde el momento en que este comando se ejecuta. Las interrupciones permanecen bloqueadas hasta que el comando EI (Enable Interrupt) las vuelve a activar.
El procesador Z80 posee 3 modalidades de interrupción: IM 0, IM 1, IM 2.

IM 0 (Interrupt Modus 0)

Con IM 0 puede conmutarse del modo 1 al modo 0.
Después de una interrupción en esta modalidad, el procesador espera un comando de entrada/salida de un periférico.

IM 1

Esta modalidad representa el estándar que se presenta al conectar el ordenador.
Al producirse una interrupción en esta modalidad, el procesador bifurca automáticamente a una dirección predeterminada.

IM 2 (Vector-Interrupt)

Con esta modalidad de interrupción, el procesador bifurca a una dirección existente en una tabla de funciones.

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	‡	‡	P	‡	•	‡	00 100 111	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	•	•	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	$A \leftarrow 0 - A$	‡	‡	V	‡	1	‡	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	$CY \leftarrow \overline{CY}$	‡	•	•	•	0	X	00 111 111	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	1	•	•	•	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	•	•	•	•	•	•	00 000 000	1	1	4	
HALT	CPU halted	•	•	•	•	•	•	01 110 110	1	1	4	
DI	$IFF \leftarrow 0$	•	•	•	•	•	•	11 110 011	1	1	4	
EI	$IFF \leftarrow 1$	•	•	•	•	•	•	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	•	•	•	•	•	•	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	•	•	•	•	•	•	11 101 101 01 010 110	2	2	8	
IM2	Set interrupt mode 2	•	•	•	•	•	•	11 101 101 01 011 110	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

4.12 COMANDOS DE ENTRADA-SALIDA

Los comandos I/O forman un grupo de comandos del Z80 al que normalmente damos poca importancia. La razón de ello es que el resultado y la aplicación de estos comandos depende en gran medida del Hardware. Cada ordenador utiliza para la comunicación con sus dispositivos periféricos por lo menos un circuito integrado preparado especialmente para el tratamiento I/O. Su ordenador posee un componente llamado PPI (Programmable Peripheral Interface (8255)), que se conoce asimismo como PIO o PIA.

Una parte de los comandos I/O se refieren a este componente que realiza automáticamente la comunicación con el teclado, el cassette y el control de memoria. A través de las líneas I/O, se encuentran asimismo conectados al procesador el PSG (Programmable Sound Generator) responsable de la generación de sonidos que controla además el Joystick, y el CRTC (Cathode Ray Tube Controller) que genera la imagen del monitor. Como pueden observar, todos los componentes seleccionables a través de comandos I/O forman parte importante del ordenador. La conexión entre estos componentes autónomos (es decir por ejemplo el CRTC genera constantemente e independientemente de la CPU la señal de salida de vídeo) y el Z80 se realiza a través de dichos comandos I/O. Con ayuda de los mismos se transmiten o reciben comandos y datos de/a estos componentes. Los componentes I/O se ocupan de la comunicación con los periféricos.

Para estas conexiones se utiliza la denominación "Interfase". Como ya hemos explicado, existen interfases internas (procesador-componente I/O) e interfases externas (componente I/O-dispositivo).

Un comando I/O envía (o graba) un valor a la interfase. Si hay un dispositivo conectado, el mismo recibe el valor y realiza la acción correspondiente. Existen un total de 256 direcciones I/O diferentes. La dirección (también: dirección del puerto I/O)

determina a qué dispositivo deben "transmitirse" los datos.

De lo explicado anteriormente pueden ver que un comando completo necesita informaciones:

- La dirección del puerto I/O.
- El valor de los datos, que qieren "transmitirse" o bien el registro en el cual se deben almacenar los datos recibidos.

La dirección del puerto siempre figura entre paréntesis. En el caso del comando IN los datos se leen del puerto grabándolos en el registro indicado. El comando OUT "transmite" los datos indicados al puerto seleccionado.
Los comandos I/O se utilizan en dos modalidades de direccionamiento:

Direccionamiento inmediato

Formato:

OUT (n),A IN A,(n)

Direccionamiento indirecto

Formato:

OUT (C),r IN r,(C)

El BASIC del CPC ofrece los comandos correspondientes que son *INP* y *OUT*. El resultado es el mismo que en los comandos de lenguaje máquina pero muchas aplicaciones pueden realizarse únicamente desde el lenguaje máquina.

Existen además 4 comandos para entrada de bloques y 4 para salida de bloques, que se utilizan de manera similar a los comandos de transferencia de bloques en los cuales HL apunta a la dirección de memoria, C representa la dirección del puerto y B la longitud del bloque.

Los comandos I/O comienzan con el código &ED y poseen un código de 2 bytes. Solamente los comandos con direccionamiento inmediato IN A,(n) y OUT (n),A utilizan un código de 1 byte.

Al utilizar los comandos I/O en los ordenadores CPC, existen algunas restricciones del set de comandos del Z80A estándard. La dirección de puerto se indica normalmente en el direccionamiento inmediato por ejemplo como valor de un byte. Este byte se envía a las líneas de dirección &A0 a &A7.

Por el contrario en el CPC están conectadas las líneas &A8 hasta &A15 con los componentes I/O. Esto significa que debe indicarse una dirección de puerto de 2 bytes, donde solamente el High byte determina la selección del puerto. Esta posibilidad la tenemos únicamente operando con direccionamiento indirecto, cargando la dirección del puerto en el registro B.

Aunque el comando es:

OUT (C),r o bien IN r,(C)

internamente se posiciona el contenido del registro B en las líneas de dirección &A8 hasta &A15. La asignación de las direcciones de puerto a los componentes correspondientes, es relativamente complicada, debido a que en muchos casos únicamente determinadas combinaciones de bits pueden seleccionar un componente.

Debido a este método de direccionamiento de puerto, los comandos I/O del direccionamiento inmediato y los comandos I/O de bloque no pueden utilizarse en los ordenadores CPC.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, (n)	A ← (n)	•	•	•	•	•	•	11 011 011 ← n →	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	•	‡	P	‡	0	‡	11 101 101 01 r 000	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	X	‡ ①	X	X	1	X	11 101 101 10 100 010	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 110 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X	‡ ①	X	X	1	X	11 101 101 10 101 010	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL -1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 111 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUT (n), A	(n) ← A	•	•	•	•	•	•	11 010 011 ← n →	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	•	•	•	•	11 101 101 01 r 001	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X	‡ ①	X	X	1	X	11 101 101 10 100 011	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 110 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	‡ ①	X	X	1	X	11 101 101 10 101 011	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL -1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 111 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

CAPITULO V : PROGRAMACION DEL Z80

5.1 EL ASSEMBLER (ENSAMBLADOR)

Hemos escrito un lenguaje assembler, para evitarnos en los próximos programas máquina el efectuar la traducción manual.

El ensamblador convierte un programa escrito en lenguaje Assembler (programa fuente o Source) en código máquina (produce el programa objeto Object-Program). Para ello, calcula por ejemplo de forma automática los desplazamientos en las bifurcaciones relativas. Reducimos por ello al mínimo el fatigoso trabajo de los programas máquina, de traducción manual de los Opcode etc.

Para el programa ensamblador del Z80, se han establecido las siguientes convenciones:

Una línea de ensamblador se forma de la siguiente manera:

```
Label   Comando   Operando  ;Comentario
```

Dado que queremos utilizar el editor BASIC para la entrada de los programas, cada instrucción assembler se asigna a un número de línea.

Definimos a continuación el formato de entrada de ensamblador. Para evitar errores al utilizar el ensamblador, es muy importante el siguiente apartado. Estúdienlo con profundidad.

Label:

Al principio de una línea puede encontrarse un Label (denominación). Un Label es una variable. La longitud del nombre

de variable (el nombre del Label no puede exceder los 6 caracteres). Los nombres de Label han de comenzarse con una letra. Los comandos Assembler no deben utilizarse como nombres Label. Utilizando los Labels nos simplifica la programación de bifurcaciones:

```
        .
ANF  Comando            ANF: Label
        .
        .
        JR  ANF             : Bifurca hacia ANF
        .
        .
```

El ensamblador calcula automáticamente la distancia correcta.

Comando (Mnemonic):

Después del eventualmente presente Label, ha de continuar el comando. Label y comando han de separarse por un espacio en blanco. El mnemónico ha de ser una palabra de comando assembler válida. Palabras de comando válidas hemos utilizado constantemente en las listas de comandos p. ej.: LD, ADD, INC, etc.

Operando:

Después del comando, se continúa con el operando, separado por un espacio en blanco. En las bifurcaciones la dirección de bifurcación puede indicarse como Label (ver anterior). La existencia de este Label es naturalmente importante.
En lugar de constantes o distancias se pueden utilizar nombres de variables o Labels.
Los operandos no deben contener nunca espacios en blanco.

Comentario:

Al final de una línea puede continuar un comentario, separado por un espacio en blanco y un punto y coma. Todas las líneas que

siguen al punto y coma, no se tienen en cuenta en la traducción. Los comentarios son una ayuda útil para la posterior comprensión del programa.

Durante la traducción, el assembler produce un listado de assembler que puede visualizarse a través de la impresora o la pantalla. El código producido puede además almacenarse en el cassette/diskette.

El listado de assembler tiene el siguiente aspecto:

```
&Dir   &Code   BASIC      Label   Comando   Operando    : Comentario
                N.Línea

A003   36CC     50        SEGUIR  LD        (HL),Bitma ; Matriz bit
A005   23       60                INC       HL         ; incr. HL
```

Adicionalmente a los comandos Z-80 el ensamblador reconoce un gran número de pseudocomandos. Estos son en realidad instrucciones para el ensamblador p.ej. END, que indica al ensamblador la finalización de la búsqueda y traducción de comandos.
Otra instrucción de importancia es EQU (inglés equal: igual). Con EQU se define el valor de una variable.

Nombre variable EQU valor

La instrucción ORG (organization) indica la dirección a partir de la cual debe almacenarse el programa. En la mayoría de los casos utilizamos como dirección inicial &A000.
En la definición numérica, se han determinado las siguientes convenciones.
Los números hexadecimales se representan posicionando un "&" delante del número.
Los números binarios se representan con "&X" delante del número.
Si un número se indica sin ninguno de estos símbolos se interpreta como número decimal.

Las convenciones estándard para el ensamblador Z80 son una H al final de un número en hexadecimal y una B al final de un número binario.
No obstante utilizaremos las convenciones arriba mencionadas "&" y "&X".

Prueben el assembler introduciendo un pequeño programa.

El programa fuente de ensamblador puede introducirse independientemente del Assembler. El primer programa del capítulo 1.2 se representaría de la siguiente manera:

```
10 'org &a000
20 'bildad equ &c000 ; inicio memoria pantalla
30 'bitmat equ &cc ;  matriz puntos de pantalla
40 'ld hl,bildad
50 'seguir ld (hl),bitmat
60 'inc hl
70 'cp h ; comparación con cero
80 'jr nz,seguir
90 'ret .
100 'end
```

Al introducir un programa pueden utilizarse letras minúsculas o mayúsculas indistintamente.

Tengan en cuenta que después de cada número de línea con una separación de 1 espacio en blanco, debe introducirse Shift 7 (').
Si omite este carácter la línea correspondiente no puede traducirla el ensamblador y visualiza el mensaje de error:

"error ' missing in ....."

La línea 10 determina el área de almacenamiento de programa a partir de &A000 en sentido ascendente.

La línea 20 asigna el valor &C000 a la variable Bildad (V).

A continuación puede utilizarse siempre "Bildad" (V) en lugar de &C000. Utilizando adecuadamente las variables, el programa resultará más depurado y más claro. En el bucle de programa hemos utilizado el label "seguir" como destino de bifurcación. Por lo demás utilizamos los comandos assembler normales.

Si en una línea se utiliza un comentario, debe separarse del resto del comando por un punto y coma. Es importante que delante del punto y coma haya un espacio en blanco. Los espacios en blanco significan para el ensamblador que finaliza un label y a continuación sigue el comando. Por ello entre label, comando, operando y comentario siempre (!!) debe utilizar un espacio en blanco y por ejemplo dentro de un operando nunca (!!) deben mediar espacios en blanco.

Ejemplo:

```
          ( HL )          ERRONEO !!!
          (HL)            CORRECTO!!!
```

Al final del programa debe codificarse el pseudocomando END, que indica al ensamblador que puede finalizar la traducción.

Grabe el programa introducido con *SAVE "nombre"* y cargue el ensamblador mediante *MERGE*. El assembler ocupa el número de líneas a partir de 10.000 y la línea 1. Dichos números de línea no deben por ello utilizarse en el programa fuente.

OBSERVACION para usuarios de diskette:

Un programa que quiere cargarse mediante *MERGE* desde el diskette ha de ser un fichero ASCII sin cabeceras. Esto se logra mediante una "A" separada por una coma después del comando *SAVE*. Dado que la carga de este tipo de fichero necesita más tiempo se debe cargar siempre en primer lugar el assembler (como programa normal BASIC) y a continuación el programa fuente (fichero ASCII sin

cabeceras). Dado que el AMSDOS ocupa aproximadamente 500 bytes de memoria RAM que se asignan dinámicamente, los usuarios de diskette deben almacenar los programas máquina en el ámbito de direcciones hasta el máximo de &A600, para evitar complicaciones.

En el CPC 664 y 6128 se puede realizar la programación normal BASIC con *MERGE*. La grabación lenta como fichero ASCII no es necesaria.

Ahora arrancamos el programa con *RUN*.

El assembler pregunta por el nombre del programa, si se desea un listado de ensamblador de la traducción y si se quiere imprimir el mismo. Las respuestas predefinidas (S para listado y N para impresora) pueden introducirse mediante *ENTER*. Utilice en primer lugar los valores estándard.

Ahora comienza la verdadera traducción.
El listado de ensamblador conocido, se visualiza en la pantalla. Al producirse un error se visualizan los pertinentes mensajes con la línea correspondiente. Al final del listado se indican, si están presentes, las etiquetas indefinidas y las variables. A continuación viene el nombre de programa, la dirección inicial, la dirección final, longitud del programa y cantidad de errores. Si se han producido errores, los mismos pueden corregirse en la correspondiente línea BASIC.

Al final del listado se visualiza una tabla de todos los labels y variables con sus valores en el orden de su aparición. Finalmente se pregunta si se quiere grabar (almacenar) el código máquina producido.

Al introducir "S" el código producido se almacena como fichero binario bajo el nombre indicado con la extensión ".OBJ" (OBJ: Object Code"). Después del ensamblaje, el programa máquina se encuentra en la posición indicada en la memoria y puede llamarse mediante *CALL*.

Si desea traducir otros programas en lenguaje máquina, puede borrar el programa fuente anterior mediante *DELETE 2-9999*. El nuevo programa puede cargarse mediante *MERGE*. Como ejemplo les indicamos el listado ensamblador completo de nuestro procedimiento de programa.

```
A000             10        ORG &a000
A000             20 BILDAD EQU &c000 ; Inicio memoria pantalla
A000             30 BITMAT EQU &cc ; Matriz puntos de pantalla
A000 2100C0      40        LD  hl,bildad
A003 36CC        50 SEGUIR LD  (hl),bitmat
A005 23          60        INC hl
A006 BC          70        CP  h ; comparación con cero
A007 20FA        80        JR  nz,seguir
A009 C9          90        RET


Programa : Bild
Inicio   : &A000      Fin : &A009
Longitud : 000A

0 errores

Tabla de variables:

BILDAD C000    BITMAT 00CC    SEGUIR A003
```

Intente comprender al introducir el listado, la estructura básica del ensamblador mediante las explicaciones que siguen al listado.

ADVERTENCIA: Nunca modifique la línea 1. Tenga en cuenta que no sobre ningún espacio en blanco o carácter similar, tampoco al final de la línea. Además no debe modificarse ni introducirse nada hasta la línea 10010 incluída, ni en el comienzo de la porción de inicialización, líneas 14160-14180. El valor inicial de bpc (V) y de vapt (V) pueden contener entonces valores erróneos, por lo cual el programa no funcionaría más.

```
1 MEMORY &9EFF:GOTO 10000
10000 REM ********** Z80 Assembler c 1984 by Holger Dullin *
*********
10010 GOTO 14160
10020 LOCATE 20,4:PRINT"Z 8 0 - A s s e m b l e r"
10030 LOCATE 5,8:INPUT "Nombre Programa :",nombre$
10040 LOCATE 19,11:PRINT "S"
10050 LOCATE 5,11:INPUT"Listado (s/n):",t$
10060 IF t$="n" THEN listflag=0:GOTO 10100 ELSE listflag=-1
10070 LOCATE 19,13:PRINT"n";
10080 LOCATE 5,13:INPUT "Impresora (s/n): ",t$
10090 IF t$="s" THEN aus=8:PRINT#aus ELSE aus=0
10100 REM "INICIO DEL ENSAMBLAJE"**
10110 MODE 2
10120 REM COMPROBACION INICIO DE LINEA
10130 laze=FNdeek(bpc)
10140 bpc=bpc+2
10150 zenr=FNdeek(bpc)
10160 IF zenr>9999 THEN PRINT#aus,"Fin asumido":GOTO 13400
10170 bpc=bpc+2
10180 IF FNdeek(bpc)<>49153 THEN PRINT#aus,"Error ' missing
in ";zenr:bpc=bpc+laze-4:feza=feza+1:GOTO 10130
10190 bpc=bpc+2
10200 REM lectura linea ---------------
10210 POKE vapt,laze-7
10220 POKE vapt+1,bpc-256*INT(bpc/256)
10230 POKE vapt+2,INT(bpc/256)
10240 REM Descomponer linea-------------
10250 zeia$=zei$
10260 bpc=bpc+laze-6
10270 FOR i=0 TO 3:a$(i)="":NEXT
10280 bepo=INSTR(zei$,";")
10290 IF bepo=0 THEN bemer$="":GOTO 10320
```

```
10300 bemer$=RIGHT$(zei$,LEN(zei$)-bepo+1)
10310 zei$=LEFT$(zei$,bepo-1)
10320 j=0
10330 IF LEFT$(zei$,1)=" " THEN zei$=RIGHT$(zei$,LEN(zei$)-1
):GOTO 10330
10340 sppo=INSTR(zei$," ")
10350 IF zei$="" THEN j=j-1:GOTO 10420
10360 IF sppo=0 THEN 10410
10370 a$(j)=LEFT$(zei$,sppo-1):zei$=RIGHT$(zei$,LEN(zei$)-sp
po)
10380 IF zei$="" THEN j=j-1:GOTO 10420
10390 IF j>3 THEN 10420
10400 j=j+1:GOTO 10330
10410 a$(j)=zei$
10420 IF j>2 THEN 13250
10430 REM interpretacion--------------
10440 j=0
10450 bef$=LEFT$(UPPER$(a$(j))+"      ",4)
10460 po=INSTR(teadr$,bef$)
10470 IF po<>0 THEN lp=0:GOTO 11190
10480 po=INSTR(teb1$,bef$)
10490 IF po<>0 THEN lp=1:GOTO 10810
10500 po=INSTR(teed$,bef$)
10510 IF po<>0 THEN lp=2:pw(1)=&ED:GOTO 10850
10520 REM comprobacion pseudocomando
10530 po=INSTR(teps$,bef$)
10540 IF po<>0 THEN 10890
10550 REM a$(j)=label ?  --------------
10560 IF j>0 THEN 13250
10570 IF a$(0)="" THEN 13100
10580 a$=a$(0)
10590 GOSUB 13630
10600 IF nolafl THEN 13280
```

```
10610 label$=UPPER$(lab$)
10620 wert=mpc
10630 lata$(ltp)=label$:wlta(ltp)=mpc:ltp=ltp+1
10640 FOR i=0 TO ultp:IF label$=ulata$(i) THEN 10670
10650 NEXT i
10660 j=j+1:GOTO 10450
10670 ON udata(i,2) GOTO 10680,10700
10680 adr=udata(i,1)-1:ziel=wert:GOSUB 14100
10690 pw(1)=of:GOTO 10720
10700 pw(2)=INT(wert/256)
10710 pw(1)=wert-pw(2)*256
10720 PRINT#aus,"**** Linea "udata(i,0);" : ";ulata$(i);"=&"
;HEX$(wert,4)
10730 FOR k=1 TO udata(i,2)
10740 POKE udata(i,1)+k-1,pw(k)
10750 NEXT k
10760 FOR k=i TO ultp-1
10770 ulata$(k)=ulata$(k+1)
10780 FOR c=0 TO 2:udata(k,c)=udata(k+1,c):NEXT c:NEXT k
10790 ultp=ultp-1:i=i-1
10800 GOTO 10650
10810 REM  com1 /1 Byte sin Operando
10820 IF a$(j+1)<>"" THEN 13270
10830 pw(1)=wb1((po-1)/4)
10840 GOTO 13100
10850 REM ed / 2bytes sin operando comenzando con ed
10860 IF a$(j+1)<>"" THEN 13270
10870 pw(2)=wed((po-1)/4)
10880 GOTO 13100
10890 REM pseudocomandos----------
10900 j=j+1
10910 ope$=a$(j):op$=UPPER$(ope$)
10920 ON (po-1)/4 GOTO 10980,11040,11060,11080,11100,11160
10930 REM EQU
10940 IF label$="" THEN 13280
```

```
10950 a$=op$:GOSUB 13790
10960 wlta(ltp-1)=wert
10970 GOTO 13100
10980 REM ORG
10990 IF op$="" THEN 13290
11000 a$=op$:GOSUB 13790
11010 lp=0
11020 mpc=wert:mpstart=mpc
11030 GOTO 13100
11040 REM END
11050 GOTO 13400
11060 REM DB
11070 a$=op$:GOSUB 14050:GOTO 13100
11080 REM DW
11090 a$=op$:GOSUB 13860:GOTO 13100
11100 REM DM
11110 IF LEFT$(op$,1)<>CHR$(34) OR RIGHT$(op$,1)<>CHR$(34) T
HEN 13260
11120 zwi$=MID$(ope$,2,LEN(ope$)-2)
11130 lp=LEN(zwi$)
11140 FOR i=1 TO lp:pw(i)=ASC(MID$(zwi$,i,1)):NEXT
11150 GOTO 13100
11160 REM DS
11170 a$=op$:GOSUB 13860
11180 ds=wert:lp=0:GOTO 13100
11190 REM con valoracion---------------
11200 j=j+1:ope$=a$(j)
11210 op$=UPPER$(ope$)
11220 IF op$="" AND bef$<>"RET " THEN 13290
11230 GOSUB 11240:GOTO 11340
11240 poko=INSTR(op$,",")
11250 IF poko=0 THEN o1$=op$:koflag=0:GOTO 11280
```

```
11260 koflag=-1
11270 o1$=LEFT$(op$,poko-1):o2$=RIGHT$(op$,LEN(op$)-poko)
11280 pokla=INSTR(op$,"("):poklz=INSTR(op$,")")
11290 IF pokla=0 THEN klaflag=0:klin$="":GOTO 11330
11300 IF pokla>poklz THEN GOTO 13260
11310 klaflag=-1
11320 klin$=MID$(op$,pokla+1,poklz-pokla-1)
11330 RETURN
11340 REM
11350 ipo=INSTR(op$,"IX")
11360 IF ipo<>0 THEN pwi=&DD:ireg$="IX":GOTO 11450
11370 ipo=INSTR(op$,"IY")
11380 IF ipo<>0 THEN pwi=&FD:ireg$="IY":GOTO 11450
11390 zwi=(po+3)/4
11400 ON zwi GOTO 12630,11920,11900,12040,12040,12080,12220,
12240,12340,12320,12380,12430,12430,12520,12560
11410  REM ld0 bifurcacion relativa (2),  bifurcacion (3),  numer
,(2), pila (2), rst, i/o, im
11420 IF zwi<24 THEN 11590
11430 IF zwi<32 THEN 11760
11440 GOTO 11830
11450 REM comandos indexados ----------
11460 iflag=-1
11470 IF (NOT klaflag) OR (ipo-pokla<>1) THEN op$=LEFT$(op$,
ipo-1)+"HL"+RIGHT$(op$,LEN(op$)-ipo-1):GOTO 11550
11480 zwi$=MID$(klin$,3,1):IF zwi$<>"+" AND zwi$<>"-" THEN I
F bef$="JP  " THEN 11540 ELSE GOTO 13250
11490 a$=RIGHT$(klin$,LEN(klin$)-3)
11500 GOSUB 14050:lp=lp-1
11510 IF fe$<>"" THEN GOTO 13260
11520 disflag=-1
11530 disw=wert:IF zwi$="-" THEN disw=(disw XOR 255) +1
11540 op$=LEFT$(op$,pokla)+"HL"+RIGHT$(op$,LEN(op$)-poklz+1)
```

```
11550 IF (INSTR(op$,"IX")=0)AND(INSTR(op$,"IY")=0)THEN 11570
11560 IF (op$=("HL,"+ireg$))AND(bef$="ADD ") THEN op$="HL,HL
" ELSE GOTO 13260
11570 GOSUB 11240
11580 GOTO 11390
11590 REM aritmetico-logicos ----------
11600 IF NOT koflag THEN a$=o1$:GOTO 11620
11610 IF o1$<>"A" THEN 11670 ELSE a$=o2$
11620 lp=1:code=zwi-16
11630 GOSUB 13680
11640 IF rflag THEN pw(1)=128 OR(code*8) OR rrr:GOTO 13100
11650 pw(1)=&X11000110 OR (code*8)
11660 GOSUB 14050:GOTO 13100
11670 IF o1$<>"HL" THEN 13260
11680 a$=o2$
11690 GOSUB 13730
11700 IF NOT rflag THEN 13260
11710 IF bef$="ADD " THEN code=&X1001:lp=1:GOTO 11750
11720 pw(1)=&ED:lp=2
11730 IF bef$="ADC " THEN code=&X1001010 :GOTO 11750
11740 IF bef$="SBC " THEN code=&X1000010 ELSE GOTO 13250
11750 pw(lp)=code OR (dd*16):GOTO 13100
11760 REM rotacion-desplazamiento------
11770 lp=2:pw(1)=&CB
11780 IF koflag THEN 13260
11790 a$=op$:GOSUB 13680
11800 IF NOT rflag THEN 13260
11810 pw(2)=(8*(zwi-24)) OR rrr
11820 GOTO 13100
11830 REM bitti ------------------------
11840 lp=2:pw(1)=&CB
11850 a$=o2$:GOSUB 13680
11860 IF NOT rflag THEN 13260
11870 bbb=ASC(o1$)-48
11880 IF (0>bbb) OR (7<bbb) OR (LEN(o1$)<>1) THEN 13260
```

```
11890 pw(2)=(64*(zwi-31))OR(bbb*8)OR rrr:GOTO 13100
11900 REM bifurcaciones relativas------
11910 lp=1:pw(1)=&10:a$=op$:GOTO 11990
11920 lp=1
11930 IF NOT koflag THEN ccc=&X11:a$=op$:GOTO 11980
11940 a$=o1$:GOSUB 13760
11950 IF (NOT cflag) OR (ccc>3) THEN 13260
11960 ccc=ccc OR 4
11970 a$=o2$
11980 pw(1)=ccc*8
11990 IF LEFT$(a$,1)<>"$" THEN GOSUB 13860:lp=lp-2:IF i>ltp
THEN wert=mpc  :GOTO 12010:ELSE 12010
12000 wert=mpc+VAL(RIGHT$(a$,LEN(a$)-1))
12010 lp=lp+1:adr=mpc:ziel=wert
12020 GOSUB 14100
12030 pw(2)=of:GOTO 13100
12040 REM Spruenge --------------------
12050 zwi=1:lp=1
12060 IF bef$="RET " THEN code=0 ELSE code=&X100
12070 GOTO 12110
12080 IF op$="(HL)" THEN lp=1:pw(1)=&E9:GOTO 13100
12090 code=&X10
12100 zwi=0:lp=1
12110 IF bef$="RET " THEN IF op$="" THEN 12130 ELSE 12160 EL
SE
12120 IF koflag THEN 12160
12130 pw(1)=192 OR code OR 1 OR (zwi*8)
12140 a$=op$
12150 GOTO 12200
12160 a$=o1$:GOSUB 13760
12170 IF NOT cflag THEN 13260
12180 pw(1)=192 OR code OR(ccc*8)
```

```
12190 a$=o2$
12200 IF bef$="RET " THEN 13100
12210 GOSUB 13860:GOTO 13100
12220 REM comandos contador------------
12230 zwi=0:GOTO 12250
12240 zwi=1
12250 IF koflag THEN 13260
12260 lp=1:a$=op$:GOSUB 13680
12270 IF rflag THEN pw(1)=&X100 OR (rrr*8) OR zwi:GOTO 13100
12280 GOSUB 13730
12290 IF NOT rflag THEN 13260
12300 pw(1)=&X11 OR (dd*16) OR (zwi*8)
12310 GOTO 13100
12320 REM comandos de pila-------------
12330 code=&X11000001:GOTO 12350
12340 code=&X11000101
12350 a$=op$:dreg$(3)="AF":GOSUB 13730:dreg$(3)="SP"
12360 IF NOT rflag THEN 13260
12370 lp=1:pw(1)=code OR (dd*16):GOTO 13100
12380 REM restart ---------------------
12390 a$=op$:GOSUB 14050
12400 zwi=wert1/8
12410 IF zwi<>INT(zwi) OR zwi>7 THEN 13260
12420 lp=1:pw(1)=&X11000111 OR (zwi*8):GOTO 13100
12430 REM comandos I/O-----------------
12440 IF NOT(koflag AND klaflag) THEN 13260
12450 IF bef$="IN  " THEN zwi=0 ELSE zwi=1:zwi$=o2$:o2$=o1$:
o1$=zwi$
12460 IF klin$<>"C" THEN 12500
12470 a$=o1$:GOSUB 13680
12480 IF (NOT rflag) OR (klin$<>"C") THEN 13260
12490 lp=2:pw(1)=&ED:pw(2)=64 OR (rrr*8) OR zwi:GOTO 13100
12500 lp=1:a$=klin$:GOSUB 14050
12510 pw(1)=&X11011011 XOR (zwi*8):GOTO 13100
```

```
12520 REM  modificacion interrupciones
12530 IF op$<>"0" AND op$<>"1" AND op$<>"2" THEN 13260
12540 lp=2:pw(1)=&ED
12550 pw(2)=&X10001110 OR ((VAL(op$)-(op$<>"0"))*8):GOTO 1310
0
12560 REM EX ---------------------------
12570 lp=1
12580 IF op$="(SP),HL" THEN pw(1)=&E3:GOTO 13100
12590 IF op$="DE,HL" THEN pw(1)=&EB:GOTO 12620
12600 IF op$="AF,AF'" THEN pw(1)=&8:GOTO 13100
12610 GOTO 13260
12620 IF iflag THEN 13260 ELSE 13100
12630 REM ld----------------------------
12640 IF NOT koflag THEN 13260
12650 a$=o1$:GOSUB 13680
12660 IF rflag THEN 12860
12670 GOSUB 13730
12680 IF rflag THEN 12760
12690 a$=o2$:GOSUB 13730
12700 IF rflag THEN 12740
12710 zwi$=o2$:o2$=o1$:o1$=zwi$
12720 a=0:GOSUB 12940
12730 IF nflag THEN 13260 ELSE GOTO 13100
12740 IF NOT klaflag THEN 13260
12750 zwi$=o2$:zwiflag=1:GOTO 12800
12760 IF op$="SP,HL" THEN lp=1:pw(1)=&F9:GOTO 13100
12770 IF klaflag THEN zwi$=o1$:zwiflag=0:GOTO 12800
12780 a$=o2$
12790 lp=1:code=1:GOTO 12830
12800 a$=klin$
12810 IF zwi$="HL" THEN lp=1:code=&A:GOTO 12830
12820 lp=2:pw(1)=&ED:code=&X1001011
12830 code=code AND NOT (zwiflag*8)
12840 pw(lp)=code OR (dd*16)
```

```
12850 GOSUB 13860:GOTO 13100
12860 zzz=rrr:a$=o2$:GOSUB 13680
12870 IF NOT rflag THEN 12900
12880 lp=1:pw(1)=64 OR (zzz*8) OR rrr
12890 IF pw(1)=&76 THEN 13260 ELSE 13100
12900 a=1:GOSUB 12940
12910 IF NOT nflag THEN 13100
12920 lp=1:pw(1)=&X110 OR (rrr*8)
12930 a$=o2$ : GOSUB 14050:GOTO 13100
12940 REMcarga especial 8-bit----------
12950 nflag=0
12960 IF o1$<>"A" THEN nflag=-1:RETURN
12970 IF klaflag THEN 13030
12980 IF o2$="I"THEN zwi=0:GOTO 13010
12990 IF o2$="R" THEN zwi=1:GOTO 13010
13000 nflag=-1:RETURN
13010 code =&X1000111:lp=2:pw(1)=&ED
13020 pp=(a*2) OR zwi:GOTO 13080
13030 IF klin$="BC" THEN zwi=0:GOTO 13070
13040 IF klin$="DE" THEN zwi=1:GOTO 13070
13050 lp=1:pw(1)=&X110010 OR (a*8)
13060 a$=klin$:GOSUB 13860:RETURN
13070 code=&X10:lp=1:pp=(zwi*2)OR a
13080 pw(lp)=code OR (8*pp):RETURN
13090 REM
13100 REM salida **********
13110 IF iflag THEN 13310
13120 IF fe$<>"" THEN feza=feza+1
13130 IF NOT listflag THEN LOCATE 5,3:PRINT zenr:GOTO 13200
13140 IF fe$<>"" THEN PRINT CHR$(7);:PRINT#aus,fe$,TAB(30);z
enr;zeia$:GOTO 13210
13150 PRINT#aus,HEX$(mpc,4);" ";
13160 FOR i=1 TO lp:PRINT#aus,HEX$(pw(i),2);:POKE mpc+i-1,pw
(1):NEXT i
```

```
13170 PRINT#aus,TAB(14);USING"####";zenr;
13180 PRINT#aus,TAB(20);label$;TAB(27);bef$;TAB(32);ope$;" "
;bemer$;
13190 PRINT#aus
13200 mpc=mpc+lp+ds
13210 lp=0:ds=0
13220 label$="":bef$="":ope$="":bemer$="":fe$=""
13230 GOTO 10130
13240 REM Mensajes Error  -------------
13250 fe$="Syntax Error":GOTO 13100
13260 fe$="Syntax Error en Operando " GOTO 13100
13270 fe$="Demasiados Operandos":GOTO13100
13280 fe$="Falta Label" GOTO 13100
13290 fe$="Falta Operando"GOTO 13100
13300 fe$="Cantidad ilegal" :GOTO 13100
13310 REM Indexado  --------------------
13320 FOR j=lp TO 1 STEP -1
13330 pw(j+1)=pw(j):NEXT
13340 pw(1)=pwi:lp=lp+1
13350 IF NOT disflag THEN 13380
13360 IF lp=3 THEN pw(4)=pw(3)
13370 pw(3)=disw:lp=lp+1
13380 iflag=0:disflag=0
13390 GOTO 13120
13400 REM Fin Programa  **********
13410 PRINT#aus
13420 IF ultp=0 THEN 13470
13430 FOR i=0 TO ultp-1
13440 PRINT#aus,"Label indefinido    ";ulata$(i);"En ";udata
(i,0);"/direccion &";HEX$(udata(i,1),4)
13450 feza=feza+1:NEXT i
13460 PRINT#aus
13470 PRINT#aus,"Programa :";name$
13480 PRINT#aus,"Inicio : &";HEX$(mpstart,4);"   Final : &";H
EX$(mpc-1,4)
13490 PRINT#aus, "Longitud: ";HEX$(mpc-mpstart,4)
13500 PRINT#aus,feza;"Error "'
13510 IF ltp=0 THEN 13560
13520 PRINT#aus,"Tabla de variables :"
```

```
13530 FOR i=0 TO ltp-1
13540 PRINT#aus,LEFT$(lata$(i)+"          ",7);HEX$(wlta(i),4)
,
13550 NEXT i
13560 PRINT#aus
13570 input "Grabacion (S/N) :",t$
13580 IF t$<> "S" THEN 13600
13590 SAVE name$+".obj",B,mpstart,mpc-mpstart
13600 END
13610 REM Subrutinas     **********
13620 REM label test ------------------
13630 laas=ASC(UPPER$(LEFT$(a$,1)))
13640 IF laas<65 OR laas>90 THEN nolafl=-1:RETURN
13650 IF LEN(a$)>6 THEN PRINT "Label  largo":a$=LEFT$(a$,6)
13660 lab$=a$:nolafl=0
13670 RETURN
13680 REM r test ----------------------
13690 FOR i=0 TO 7
13700 IF reg$(i)=a$ THEN rflag=-1:rrr=i:RETURN
13710 NEXT
13720 rflag=0:RETURN
13730 REM rps test --------------------
13740 FOR i=0 TO 3:IF dreg$(i)=a$ THEN rflag=-1:dd=i:RETURN
ELSE NEXT
13750 rflag=0:RETURN
13760 REM cond test -------------------
13770 FOR i=0 TO 7:IF a$=cond$(i) THEN cflag=-1:ccc=i:RETURN
 ELSE NEXT
13780 cflag=0:RETURN
13790 REM Test numero-----------------
13800 wert=VAL(a$)
13810 laas=ASC(LEFT$(a$,1))
13820 IF wert=0 AND laas<>38 AND (laas>57 OR laas<48) THEN f
e$="CARACTER ILEGAL":wert=0:RETURN
```

```
13830 IF wert>=0 THEN 13850
13840 IF LEFT$(a$,1)="&" THEN wert=wert+2^16 ELSE fe$="illeg
al Quantity":wert=0
13850 RETURN
13860 REM Valoracion--------------------
13870 GOSUB 13620
13880 IF nolafl THEN GOSUB 13790:GOTO 13920
13890 FOR i=0 TO ltp:IF lata$(i)<>lab$ THEN NEXT
13900 IF i>ltp THEN 13980
13910 wert=wlta(i)
13920 werth=INT(wert/256)
13930 wertl=wert-256*werth
13940 lp=lp+2
13950 pw(lp-1)=wertl
13960 pw(lp)=werth
13970 RETURN
13980 ulata$(ultp)=a$
13990 udata(ultp,0)=zenr
14000 udata(ultp,1)=mpc+lp-iflag-disflag
14010 udata(ultp,2)=2+(bef$="DJNZ" OR bef$="JR  ")
14020 ultp=ultp+1
14030 wert=0
14040 GOTO 13920
14050 REM Valoracion LOW---------------
14060 GOSUB 13860
14070 lp=lp-1
14080 IF werth<>0 THEN fe$="CANTIDAD ILEGAL":wert=0
14090 RETURN
14100 REM Calcular Offset ------------
14110 of =ziel-adr
14120 of=of-2
14130 IF of>129 OR of<-126 THEN fe$="illegal Quantity":of=0
14140 IF of<0 THEN of=256+of
14150 RETURN
```

```
14160 REM Inicializacion *********
14170 zei$="test"
14180 vapt=@zei$
14190 DEF FNdeek(x)=PEEK(x)+256*PEEK(x+1)
14200 MODE 2
14210 teadr$="LD  JR  DJNZCALLRET JP  INC DEC PUSHPOP RST IN
  OUT IM  EX  ADD ADC SUB SBC AND XOR OR  CP  RLC RRC RL  RR
  SLA SRA *** SRL BIT RES SET "
14220 teed$="CPD CPDRCPI CPIRIND INDRINI INIRLDD LDDRLDI LDI
RNEG OTDROTIROUTDOUTIRETIRETNRLD RRD "
14230 DATA A9,B9,A1,B1,AA,BA,A2,B2,A8,B8,A0,B0,44,BB,B3,AB,A
3,4D,45,6F,67
14240 teb1$="CCF CPL DAA DI  EI  EXX HALTNOP RLA RLCARRA RRC
ASCF "
14250 DATA 3F,2F,27,F3,FB,D9,76,00,17,07,1F,0F,37
14260 teps$="EQU ORG END DB  DW  DM  DS  "
14270 DIM lata$(70),wlta(70),ulata$(50),udata(50,2)
14280 DIM wb1(12),wed(20)
14290 FOR i=0 TO 20:READ a$:wed(i)=VAL("&"+a$):NEXT
14300 FOR i=0 TO 12:READ a$:wb1(i)=VAL("&"+a$):NEXT
14310 bpc=FNdeek(&170)+&170:mpc=40960:mpstart=mpc
14320 DIM reg$(7),cond$(7),dreg$(3)
14330 FOR i=0 TO 7:READ reg$(i):NEXT
14340 FOR i=0 TO 7:READ cond$(i):NEXT
14350 FOR i=0 TO 3:READ dreg$(i):NEXT
14360 DATA B,C,D,E,H,L,(HL),A
14370 DATA NZ,Z,NC,C,PO,PE,P,M
14380 DATA BC,DE,HL,SP
14390 GOTO 10020
```

Descripción del programa

Linea 1:
Se reserva espacio de memoria RAM de &A000 hasta &AB7F para el programa máquina y a continuación se salta el programa fuente entre línea 2 y 9999.

Línea 10010:
Bifurcación a la porción de programa de inicialización, es decir confección de la tabla de comandos, etc. (vean línea 14160).

Línea 10020-10090:
Menú, se determinan Listflag (V) y canal de salida aus (V).

Línea 10100-10190:
BPC indica la dirección actual en el programa fuente BASIC (BPC: Basic Program Counter). Al comienzo de una línea se encuentra la longitud de la misma como Low y High bytes. FNdeek (BPC) lee el valor de 16 bits en dirección BPC y BPC+1. El valor corresponde a la longitud de línea laze (V). BPC se incrementa en 2 y el número de línea zenr (V) se lee. Si es mayor que 9999 se finaliza la traducción. En línea 10180 se comprueba si el carácter (') se encuentra al comienzo de la línea, de lo contrario se visualiza un mensaje de error y se lee la línea siguiente.

Línea 10200-10240:
Mediante esta porción del programa se llena zei$(V) con la línea actual. Para mantener una alta velocidad del ensamblador, esto se realiza modificando el apuntador de string de zei$(V) en la tabla de variables internas.

Línea 10240-10420:
Se almacena en primer lugar un eventual comentario en

bemer$(V), a continuación se corta la línea restante con cada espacio en blanco y las porciones cortadas se almacenan en A$ (j) (v). Si la línea puede dividirse en más de 3 partes (label, comando, operando), es decir j)2, se visualiza un Syntax Error.

Línea 10430-10540:
Aquí se comprueba si en A$ (j) (v) se trata de un comando válido. En caso afirmativo se bifurca a la posición en la que se traducen estos comandos.

Línea 10540-10550:
Si no se ha confirmado un comando, se comprueba por un pseudocomando bifurcando a continuación.

Línea 10560-10800:
Si se trata de un label, éste se introduce en una tabla de labels y el MPC (Machin Program Counter) se asigna como valor al label (línea 10610-10630). En las líneas siguientes hasta 10800 se comprueba si este label ha sido ya utilizado con anterioridad, sin haberse definido. En caso afirmativo se pokea el valor correspondiente y el label se borra de la tabla de los indefinidos ulata$ (i) (v). Si no se trata de un label válido (es decir si no comienza con una letra), se visualiza el mensaje de error "Falta label" (línea 10600).

Línea 10810-10840:
Aquí se valoran los comandos con Opcode de un byte que no tienen operando. El código se compone de la posición Tebl$ (v) y el correspondiente wvl(i) (v).

Línea 10850 - 10880:

Aquí se tratan los comandos de 2 bytes sin operando, el primer Opcode es siempre &ed (pw (1) = &ed). El segundo byte del Opcode se compone de la posición del comando en teed$ (v) y wed (i) (v).

Línea 10890 - 11180:
Aquí se traducen todos los pseudo-comandos.

Línea 11190 - 13080:
Si el comando no pertenece a ninguno de los grupos mencionados, se valora en esta porción del programa. Se comprueba en primer lugar si el operando op$ (v) contiene una coma. En caso afirmativo se descompone en o1$ (v) (porción delante de la coma) y o2$ (v) (porción posterior a la coma) y koflag (v) se activa en -1 (=verdadero).A continuación se comprueba la existencia de paréntesis. En caso afirmativo se almacena el contenido de los paréntesis en Klin$ (v) y Klflag (v) se activa (= -1).

Línea 11280 - 11330:
Si se trata de un comando direccionado de forma indexada se bifurca a línea 11450.

Línea 11400 - 11440:
Según la posición en teadr$ (V), se bifurca a la rutina de tratamiento de comandos.

Línea 11450 - 11580:
En los comandos indexados (XY) y (XY + dis) se reemplazan con HL y los iflags(V), disflag(V) se activan correspondientemente. Continúa entonces la valoración normal de comandos a partir de línea 11390. Después de la interpretación, se altera nuevamente la modificación y se visualiza el código del comando indexado (el análogo al HL).

Línea 11590 - 11750:
Aquí se interpretan los comandos aritméticos (8 y 16 bits).

Línea 11760 - 11820:
Comandos de rotación y desplazamiento.

Línea 11830 - 11890:
Comandos de manipulación de bits.

Línea 11900 - 12030:
Bifurcaciones relativas (JR y DJNZ).

Línea 12050 - 12210:
Otras bifurcaciones (JP, RET, CALL).

Línea 12220 - 12310:
Comandos de contador (INC, DEC).

Línea 12320 - 12620:
(Ver líneas REM)

Línea 12630 - 13080:
Comandos de carga.

La explicación detallada de todas estas rutinas, excedería los límites del presente libro. Seleccionamos como ejemplo la rutina para los comandos de manipulación de bits:

Línea 11840:
lp (V)(longitud del comando, es decir número de valores a 'pokear') es 2. El primer valor pw(1) (V) es &CB. Estos valores corresponden a todos los comandos bit.

Línea 11850:
o2$(V) (porción del operando posterior a la coma) se almacena en a$(V) para el traspaso a la subrutina a partir de línea 13680. La subrutina comprueba si se trata de uno de los registros A, B, C, D, E, H, L o bien (HL).

Línea 11860:
Si no se encuentra coincidencia (Rflag = 0) se visualiza el mensaje de error "Syntax Error en operando". De lo contrario se devuelve el código del registro en rrr y se activa el Rflag.

Línea 11870:
Se asigna el valor del número posicionado delante de la coma (del número de bit) a bbb.

Línea 11880:
Se comprueba si el número se encuentra en el rango de 0 a 7. En caso negativo se visualiza nuevamente "Syntax Error en operando".

Línea 11890:
Finalmente se almacena el Opcode en pw(2)(V). El Opcode se compone de la siguiente forma:

01 bbb rrr - para comandos BIT
10 bbb rrr - para comandos RES
11 bbb rrr - para comandos SET

De (zwi - 31) * 64 resultan los bits 7 y 6 del Opcode. zwi (V) es la posición del comando en teadr$ (V). bbb*8 representan los bits 5-3 y rrr los bits 2-0. rrr se determina en la subrutina y corresponde al código de registro. Si el Opcode está calculado se bifurca a la salida (línea 13100). De forma similar funcionan asimismo las restantes rutinas.

Línea 13100 - 13230:
Salida: si el iflag (V) (Flag para comandos indexados) está activado, se bifurca previamente a la rutina de línea 13310.

De lo contrario se visualiza la línea completa de ensamblador. Si se detectan errores se visualizan los mensajes correspondientes y se incrementa feza(V), el contador de errores.

Antes de bifurcar al comienzo para traducir la línea siguiente se inicializan las variables más importantes y el mpc (machine program counter) se incrementa en la longitud de comando lp(V).

Línea 13240 - 13300:
Si se detecta un error se bifurca a una de estas rutinas, que rellenan el string de errores fe$(V) con el mensaje bifurcando a continuación a la salida.

Línea 13310 - 13390:
En este punto se preparan los códigos para los comandos indexados.

Línea 13400 - 13600:
Al final del programa se visualizan los labels indefinidos, el nombre del programa, dirección inicial y final, longitud, cantidad de errores y la tabla de variables.
A partir de la línea 13570 comienza la grabación del código objeto producido.

Línea 13610 - 14150:
Contiene subrutinas de utilización muy frecuente.

Linea 13620 - 13670:
Se comprueba si a$(V) contiene un label válido.

Linea 13680 - 13720:
Estas líneas comprueban si a$(V) contiene un registro (A,B,C,D,E,H,L,(HL)).

Línea 13730 - 13750:
Se comprueba si a$(V) contiene una pareja de registros (BC, DE, HL, o SP).

Línea 13760 - 13780:
Comprobación de que a$(V) contenga una condición (C, NC, Z, NZ, PO, PE, P o M).

Línea 13790 - 13850:
Comprueba si a$(V) es un número y retorna su valor.

Línea 13860 - 14040:
Estas líneas determinan el valor de 2 bytes de a$(V). a$(V) puede representar un número, una variable o un label.

Línea 14050 - 14090:
Se determina aquí el valor de un byte (Low-byte) de a$.

Línea 14100 - 14150:
Calcula el offset para bifurcaciones relativas.

Linea 14160 - 14390:

Inicialización: se generan los campos de datos y los strings para la comparación. vapt(V) apunta a la dirección en la cual se encuentra almacenada la longitud de string de zei$(V). FNdeek(X) indica el valor de 16 bits de 2 posiciones consecutivas de memoria.
bpc se posiciona inicialmente en la siguiente linea de linea 1 (=384).
mpc se carga con &A000.

Lista de variables:
(SUB significa: SUBRUTINA)

```
        a - Traspaso a SUB "carga especial 8 bits"
       a$ - Traspaso a diferentes subrutinas
      adr - Traspaso a SUB "calcular offset": dirección
            bifurcación
      aus - Canal dispositivo salida (0 ó 8)
      bbb - Código número bit en comandos manipulación bit
     def$ - Palabra reservada de ensamblador
   bemer$ - Comentario en línea ensamblador
     bepo - Posición inicial de comentario en línea
      bpc - Apuntador programa BASIC
      ccc - Código condición de bifurcación
    cflag - Activado    (=-1), si condición encontrada.
            Desactivado (=9) si condición no encontrada, retorna
            SUB "COND TEST".
     code - Se utiliza  para generar el Opcode del comando de
            referencia.
     dis$ - Contiene la distancia en comandos indexados
  disflag - Activado  en comando indexado con indicación
            distancia, de lo contrario desactivado.
     disw - Valor de la distancia indicada (complemento a 2).
       ds - Contiene  la cantidad de posiciones de memoria
            reservadas por un comando ds.
      fe$ - Mensaje de error
     feza - Número de errores.
    i,j,k - Contadores para bucles
    iflag - Activado  en comandos indexados, de lo contrario
            desactivado
      ipo - Posición del  registro de indice (IX o IY) en el
            operando
    ireg$ - Si se  trata de direccionamiento indexado ireg$
            contiene IX o bien IY
  klaflag - Activado  si el operando contiene paréntesis, de lo
            contrario desactivado.
     klin$ - Contenido del paréntesis del operando (si existe)
   koflag - Activado  si el operando contiene coma, de lo
```

contrario desactivado.

laas - Código ASCII del primer carácter de un label a comprobar (SUB "Label-test")

lab$ - Retorno del nombre del label desde SUB "Label-test"

label$ - Nombre del label actual

laze - Longitud de la línea actual de programa fuente traducido

listflag - Activado si se desea listado, de lo contrario desactivado

lp - Longitud de comando (longitud código objeto)

ltp - Apuntador a posición libre en tabla de label (lata$) (label apuntador tabla)

mpc - Machine Program Counter: Apunta la posición de memoria en la cual se almacena el siguiente código máquina

mpstart - Dirección inicial del programa máquina

name$ - Nombre del programa

nflag - Activado si en un comando de carga se trata de direccionamiento inmediato, de lo contrario desactivado (SUB "carga-especial-8-bit")

nolafl - No Label Flag: Activado si la comprobación de label tiene resultado negativo, de lo contrario desactivado; retorno del SUB "Label-test"

o1$ - Porción de operando delante de la coma

o2$ - Porción de operando posterior a la coma

of - Offset calculado de SUB Offset

op$ - Operando para el tratamiento

ope$ - Operando, original para salida

po - Posición de la palabra de comando en el test-string

pokla - Posición de "Abrir paréntesis" en operando

poklz - Posición de "Cerrar paréntesis" en operando

poko - Posición de la coma en el operando

pwi - Primer byte del Opcode en direccionamiento indexado, es decir &FF o &DF

rflag - Activado si SUB "Rtest" ha detectado uno de los registros A,B,C,D,E,H,L, (HL) de lo contrario desactivado

rrr - Código del registro; retorno de SUB "Rtest"

sppo - Space Position, posición de espacio en blanco en la

línea
t$ - String de entrada (Menú)
teadr$ - Test direccionamiento: contiene todos las palabras de comando que aparecen con un operando
tebi$ - Test comandos de un byte: contiene todas las palabras de comandos que no tienen operando y que poseen un Opcode de 1 byte
teed$ - Test &ED: contiene todas las palabras de comando que aparecen únicamente sin operando, con un Opcode de 2 bytes y cuyo primer byte es &ED
teps$ - Test pseudo: contiene todos los pseudo-comandos
ultp - Apuntador tabla de labels indefinidos: apunta al siguiente espacio libre en la tabla ulata$ y udata
vapt - Apuntador variables: apunta a la dirección de zei$ en la tabla interna de variables
wert - Valor de una expresión, retorno de SUB "Valoración" o bien SUB "Test-número"
werth - High-byte del valor
wertl - Low-byte del valor
zei$ - Contiene la línea actual a tratar
zeia$ - Contiene la línea actual (original)
zenr - Número actual de línea
ziel - Traspaso de SUB "Calcular Offset" a dirección destino
zwi - Diversas tareas de almacenamiento temporal
zwi$ - Diversas tareas de almacenamiento temporal

Tablas:

lata$ (50) - Tabla de labels
wlta (50) - Valor de labels en tabla de valores
ulata$ (50) - Tabla de labels indefinidos
udata$ (50,2) - Datos de labels indefinidos
(i,0) : Número de línea cuando aparece

```
      (i,1) : Dirección del valor a pokear posteriormente
      (i,2) : Tipo, es decir 16 bit (=2) u Offset (=1)
   wb1 (12) - Opcode de comandos 1 byte (teb1$)
   wed (20) - Opcode de comandos de 2 bytes (teed$)
  reg$ (7) - Tabla de registros: B,C,D,E,H,L, (HL), A
 cond$ (7) - Tabla de condiciones: NZ, Z, NC, C, PO, PE, a, M
dreg$ (3) - Tabla de registros dobles: BC, DE, HL, SP
```

5.2 PROGRAMACION

Como primer proyecto de programación nos ocuparemos nuevamente de la pantalla.

Suponemos que al programar los ejemplos, no han entrado en modalidad 2 antes de arrancar el programa. El resultado es algo imprevisible, que les explicaremos a continuación:

Una vez entrado en modalidad 2, el primer byte de la pantalla, de la esquina superior izquierda corresponde a la dirección &C000:

Mediante *POKE &C000,255* obtenemos una línea en esta posición.

Movemos ahora el cursor al borde inferior de la pantalla y efectuamos un scrolling moviendo el cursor una posición hacia abajo. A continuación movemos el cursor al comienzo de la línea media de la pantalla introduciendo nuevamente *POKE &C000,255*, veremos entonces dibujarse una línea en el área inferior de la pantalla. Si introducimos en cambio *POKE &C050,255* la línea aparece en la posición primitiva. La diferencia entre &C000 y &C050 es &50, es decir 80 en decimal. Esta diferencia corresponde a los 80 caracteres de la línea que al producirse el scrolling ha "desaparecido" por la parte superior de la pantalla. Si realizamos un nuevo scrolling obtenemos la línea mediante *POKE &C0A0,255* (&C050 + &50 = &C0A0).

La diferencia entre &C000 y la dirección real del byte superior izquierdo de la pantalla se almacena internamente en las direcciones &B1C9 (Low) y &B1CA (High). Leamos el valor de esta posición de memoria de 16 bits. Si no se produce ningún nuevo scrolling, obtenemos

```
*PRINT HEX$(PEEK(&B1C9)+PEEK(&BACA)*256)*  el valor A0

*PRINT HEX$(PEEK(&B7C4)+PEEK(&B7C5)*256)*  el valor A0

*PRINT HEX$(PEEK(&B7C6)+PEEK(&B7C7)*256)*  el valor A0
```

Esta es la diferencia exacta entre &C000 y &C0A0. A través de una modificación de los contenidos de &B1C9 y &B1CA tenemos la posibilidad de obtener interesantes efectos en la pantalla.

Por ejemplo:

```
FOR I=0 TO 255:POKE &B1C9,I:PRINT"*":NEXT :CPC 464
FOR I=0 TO 255:POKE &B7C4,I:PRINT"*":NEXT :CPC 664
FOR I=0 TO 255:POKE &B7C4,I:PRINT"*":NEXT :CPC 6128
```

En todas las operaciones relacionadas con la pantalla hemos de tener en cuenta esta diferencia.

Considerando la diferencia producida por el scrolling, queremos ahora modificar el programa para invertir el carácter superior izquierdo de la pantalla.

En primer lugar cargamos HL nuevamente con &C000. Dado que trabajamos en ensamblador, almacenamos &C000 en una variable.
El programa se arranca como de costumbre a partir de la dirección &A000. Las primeras líneas se representan de la siguiente manera:

```
10 'Bildad EQU &C000 ; Dirección base pantalla
20 'ORG    &A000
30 'LD     HL,Bildad
```

Ahora debemos sumar la diferencia a la dirección base.

```
40 'LD      DE,(&B1C9) ; "diferencia de scrolling"
50 'ADD     HL,DE ; Calcular dirección inicial
```

El comando de carga de 16 bits LD DE,(&B1C9) en línea 50 carga Low y High bytes en la pareja de registros DE. Procedemos ahora de igual forma que en el programa del capítulo 4.10.

```
60 'LD      DE,&800 ; Diferencia
70 'LD      B,8  ; Contador bucle
80 '    LD A,(HL) ; Matriz de bits actual
90 'CPL ; Invertir
100 'LD (HL),A ; Almacenar nuevamente
110 'ADD HL,DE ; Sumar diferencia
```

ADD HL,DE puede provocar que HL exceda el valor &FFFF.

Ejemplo:

HL=&F9A0 DE=&800

Después ADD HL,DE :

HL=&01A0 Carry=1

Seguramente ésta no será la dirección correcta en la memoria de pantalla, ya que en esta dirección se encuentran nuestros programas BASIC. Los puntos almacenados en las direcciones superiores a &FFFF se encuentran en dirección &C000.
Si se produce un desbordamiento (CF=1), hemos de sumar &C000 a HL. Intenten ahora completar ustedes este programa en lenguaje máquina.

Solución:

```
120 'CALL C,DIFADD ; Subrutina para corrección
130 'DJNZ wieder ; Repetir 8 veces
140 'RET ; Retorno Subrutina
150 'DIFADD PUSH DE ; Subrutina arranque, salvar DE
160 'LD DE,bildat ; Bildat=&C000
170 'ADD HL,DE ; Sumar a HL
180 'POP DE ; Extraer DE
190 'RET ; Retorno subrutina
200 'END
```

Descripción:

Línea 120:
Si se ha producido un desbordamiento se bifurca a la rutina de corrección

Líneas 150 y 190:
Todas las parejas de registros han sido utilizadas. La adición de 16 bits es posible únicamente mediante el direccionamiento implícito. Por ello se almacena temporalmente el contenido de DE mediante PUSH DE en el stack, extrayéndolo nuevamente después de la adición, mediante POP DE.

Ensamblemos el programa y observemos el listado:

```
A000             10   BILDAD EQU &C000 ; direccion pantalla
A000             20          ORG &A000
A000 2100C0      30          LD  HL,BILDAD
A003 ED5B5BC9B1  40          LD  DE,(&B1C9) ; "diferencia scolling"
A007 19          50          ADD HL,DE ; nueva direccion inicial
A008 110008      60          LD  DE,&800 ; diferencia
A00B 0608        70          LD  B,8 ; contador bucle
A00D 7E          80   WIEDER LD  A,(HL) ; matriz actual de bits
A00E 2F          90          CPL ; invertir
```

```
A00F 77          100          LD  (HL),A ; almacenar nuevamente
A010 19          110          ADD HL,DE ; sumar diferencia
A011 DC0000      120          CALL C,DIFADD ; subrutina correccion
A014 10F7        130          DJNZ WIEDER
A016 C9          140          RET
**** linea 120  : DIFADD = A017
A017 D5          150  DIFADD PUSH DE ; salvar DE
A018 1100C0      160          LD  DE,BILDAD ; =&C000
A01B 19          170          ADD HL,DE ; sumar a HL
A01C D1          180          POP DE
A01D C9          190          RET

Programa: Invertir
Inicio: &A000    Final: &A01D
Longitud: 001E
0 Errores
Tabla de variables:
BILDAD C000    WIEDER A00D    DIFADD A017
```

En línea 130 se bifurca al label DIFADD, aunque el mismo se define en línea 150. Por ello se almacena momentáneamente DC0000 como código. Al traducir la línea 150 el ensamblador encuentra el label DIFADD e indica que el mismo aparece en línea 120. El código DC0000 se transforma automáticamente, lo mismo puede ocurrir con comandos JR y JP. Este problema se produce al realizar una bifurcación hacia adelante en el programa.

Esta forma de tratamiento descrita para las bifurcaciones hacia adelante es necesaria dado que se trata de un ensamblador de un paso, lo que significa que el ensamblador repasa el programa fuente una única vez.

Un ensamblador de 2 pasos, en cambio, en el primer paso busca únicamente todas las variables y labels, asignando sus valores correspondientes. En el segundo paso se realiza la traducción. Ensambladores profesionales realizan varios pasos (PASSES). En nuestro caso, el ensamblador de un paso es más adecuado porque se ejecuta en la mitad de tiempo que el ensamblador de 2 pasos.

Volvamos al programa:

Existen naturalmente otras soluciones para este programa. En primer lugar es importante que el programa obtenga los resultados deseados. Es aconsejable buscar la versión más breve y más rápida.

En los programas siguientes no nos fijaremos tanto en la velocidad de ejecución y en la ocupación de memoria, sino en la facilidad de comprensión de las soluciones.

HL no puede ser en ningún caso menor que &C000. H puede adoptar valores entre &C0 y &FF. En todos los valores los bits superiores (número 7 y 6) están activados. Para prevenir posibles errores podemos activar estos bits en cada paso del bucle. En este caso podemos olvidar la subrutina de línea 160 y escribir en línea 130:

```
130 'SET 6,H
135 'SET 7,H
```

Con la interrelación OR podemos resolver con mayor rapidez esta tarea (OR permite activar bits individualmente).

```
85  'LD C,&X11000000
130 'LD A,H
133 'OR C
135 'LD H,A
```

Los programas de manipulación de pantalla realizados hasta el momento, pueden asimismo adaptarse a una formulación estándard teniendo en cuenta la diferencia del scrolling. Las modificaciones pertinentes quedan a su elección.

Rutinas BASIC del monitor

Hemos visto hasta ahora el procedimiento del ensamblador pero quedan aún por comentar algunas utilidades que optimizan el

trabajo en lenguaje máquina, entre ellas el llamado 'MONITOR'.

No se trata del monitor (pantalla) de su ordenador, sino de un programa que permite por ejemplo comprobar el contenido de la memoria, visualizarlo y modificarlo (inglés to monitor: comprobar). Un monitor ofrece asimismo la posibilidad de almacenar, cargar y arrancar programas máquina. A continuación programaremos algunas funciones de un monitor de este tipo en lenguaje máquina.

De esta manera "matamos 2 pájaros de un tiro":
se exponen aquí técnicas fundamentales de programación obteniendo como resultado un programa monitor.

Como ya hemos mencionado, la tarea fundamental de un programa monitor es la de visualizar el contenido de la memoria. Ello puede realizarse en BASIC mediante comandos PEEK.

Confeccione un programa que al introducir la dirección inicial (V) y la dirección final (V) le proporcione los contenidos intermedios de memoria. Utilice para la salida el formato usual para un HEX DUMP (salida de contenidos de registros en formato hexadecimal), de la siguiente manera:

Dump hexadecimal desde dirección &10 hasta &27:

```
0010 C3 16 BA C3 10 BA D5 C9 C.:C.:UI
0018 C3 BF B9 C3 B1 B9 E9 00 C?9C19i.
0020 C3 CB BA C3 B9 B9 00 00 CK:C99..
```

En los equipos CPC 664 y CPC 6128 obtendremos lamentablemente otros valores en el Hex Dump

Su programa debe generar la misma imagen que el nuestro. La secuencia de códigos ha de ser necesariamente exacta.
Debe tenerse en cuenta, que en la porción derecha y a continuación de una línea de Hex-dump se visualiza la representación ASCII de

los códigos correspondientes. Los códigos de valor superior a 127 se decrementan previamente en 128 y los códigos no representables (0-31) se visualizan como puntos.

Solución:

```
10 REM Rutinas de Monitor BASIC
20 MODE 1
30 INPUT inicio
40 INPUT final
50 FOR i=inicio TO final STEP 8
60 ascii$=""
70 PRINT HEX$(i,4);" ";
80 FOR j=0 TO 7
90 w=PEEK(i+j)
100 PRINT HEX$(w,2);" ";
110 IF w > 127 THEN w=w-128
120 IF w<32 THEN w=46
130 ascii$=ascii$+CHR$(w)
140 NEXT j
150 PRINT" ";ascii$;
160 NEXT i
170 END
```

Con este programa pueden visualizar el contenido completo de la memoria RAM. Introduzca en su programa monitor la línea siguiente:

```
1 REM Esta es la primera línea
```

Veamos el contenido de memoria desde &170 hasta &200. En la representación ASCII de los contenidos de memoria vemos la primera línea, es decir el comentario "Esta es la primera línea". A partir de &170 se almacenan los programas BASIC en la memoria. A continuación del programa BASIC hay una página de todas las variables del programa, administrada internamente, donde para

variables numéricas se almacena directamente el valor numérico y para variables de string se almacena la dirección y la longitud de la cadena de caracteres. Las variables se almacenan en la secuencia de su aparición en el programa.
Los programas profesionales monitor, ofrecen la posibilidad de modificar directamente el contenido de memoria a través de la pantalla. Hasta aquí el comando M (Monitor) del programa.

Rutina Fill

Veamos ahora la rutina "Fill", que se utiliza para rellenar un área de memoria individualizada, con un valor fijo. De esta manera por ejemplo, puede borrarse la memoria de pantalla completa, es decir, rellenarla con ceros. El comando F (Fill) se utiliza por ejemplo para obtener condiciones determinadas en el contenido de la memoria antes de la ejecución del programa. Tenemos el siguiente ejemplo:

El programa BASIC requiere la entrada de la dirección inicial y final del área a rellenar y el valor con el cual queremos inicializar la misma. En el programa BASIC, debe comprobarse si la dirección inicial (V) es menor que la dirección final (V) y si se trata de números de 2 bytes, es decir números entre 0 y 2^16-1. Debe comprobarse también el valor (V) por el área 0-255 (1 byte). Estos 3 valores (5 bytes) se "POKEAN" en posiciones fijas de memoria, para tenerlos disponibles después de la llamada a la rutina FILL de lenguaje máquina. El programa máquina debe realizar el "relleno", retornando a continuación al BASIC.

Presentamos seguidamente el programa BASIC que realiza la entrada de esta forma y que comprueba los criterios mencionados.

```
10 MEMORY &9FFF
90 MODE 2
100 LOCATE 10,5: PRINT"PROGRAMA MONITOR"
110 LOCATE 5,8: PRINT"?"
120 LOCATE 7,10: INPUT "DIRECCION INICIAL:",START
130 IF START < 0  OR START <= 2^16 THEN 120
140 IF START  <>  INT(START) THEN 120
150 LOCATE 7,11: INPUT "DIRECCION FINAL:",ENDE
160 IF ENDE <= START  OR ENDE  > = 2^16 THEN 150
170 IF ENDE  <>  INT(ENDE) THEN 150
180 LOCATE 7,12: INPUT "VALOR:",VALOR
190 IF VALOR < 0 OR VALOR  >  255 OR (VALOR  <> INT(VALOR))
    THEN 180
200 POKE &A000,VALOR
210 POKE &A002,INT(START/256): POKE &A001,START-INT
    (START/256)*256
220 POKE &A004,INT(ENDE/256): POKE &A003,ENDE-INT
    (ENDE/256)*256
230 CALL &A005
240 END
```

Para este programa máquina el valor (V) se encuentra en la dirección &A000, la dirección inicial a partir de &A001 (Low-High) y la dirección final a partir de &A003 (Low-high). Dado que las primeras posiciones de memoria a partir de &A000 están ocupadas, arrancamos el programa máquina a partir de dirección &A005.

La primera porción del programa fuente:

```
10 '       ORG   &A005
20 'START EQU &A001
30 'ENDE  EQU &A003
40 'VALOR EQU &A000
50 '       LD  A,(VALOR)
60 '       LD  DE,(START) ; Apuntador bloque
```

Descripción del programa:

Línea 10:
Inicia el programa en &A005

Línea 20-40:
Para mayor claridad se definen las direcciones de los datos traspasados (direcciones de transferencia) como variables. Para cualquier modificación es necesario únicamente alterar el valor en la definición de la variable.

Línea 50-60:
El valor se carga en el ACU (1 byte), la dirección final en la pareja de registros HL (2 bytes) y la dirección inicial en la pareja de registros DE (2 bytes).

De esta manera llegamos finalmente a la rutina FILL.
Presentamos en primer lugar la solución más aproximada:

```
70  'BUCLE LD  (DE),A    ; Escribir valor
80  '      INC DE        ; Incrementar apuntador
90  '      LD  HL,(ENDE) ; Calcular
100 '      SBC HL,DE     ; si ya llegado
110 '      JR  NZ,BUCLE  ; al final?
120 '      LD  (DE),A    ; Rellenar ultimo elemento
130 '      RET
140 '      END
```

Descripción del programa:

Línea 50:
Cargar HL con dirección final (V)

Línea 70:
Inicio del bucle. Almacenar en HL el valor (A).

Línea 80:
Incrementar apuntador dirección (DE).

Línea 100:
Sustracción 16 bits de dirección actual de la dirección final (HL-DE)

Línea 110:
Si el apuntador DE es menor que la dirección final en HL, el flag Z no queda activado, dado que HL - DE es diferente de cero. En este caso (NZ) se bifurca al inicio del bucle (BUCLE). Si HL es igual a DE, entonces Z=1 y se ejecuta el siguiente comando (Línea 120).

Línea 120:
Aquí se almacena el mismo valor A (= contenido ACU) en la dirección final del área a rellenar. Esto hasta el momento no se había realizado (¿Por qué?)

Línea 130:
Retorno al BASIC.

Si traduce este programa mediante el ensamblador obtendrá el listado siguiente:

```
A000             10   START  EQU  &A001
A000             20   ENDE   EQU  &A003
A000             30   VALOR  EQU  &A000
A005             40          ORG  &A005
A005 3A00A0      50          LD   A,(VALOR)
A008 ED5B01A0    60          LD   DE,(START) ; Apuntador bloque
A00C 12          70   BUCLE  LD   (DE),A     ; Almacenar valor
A00D 13          80          INC  DE         ; Incr. apuntador
```

```
A00E 2A03A0     90          LD   HL,(ENDE)
A011 ED52       100         SBC  HL,DE       ; Final?
A013 20F7       110         JR   NZ,BUCLE    ; No, seguir
A015 12         120         LD   (DE),A      ; Rell. ultimo ele.
A016 C9         130         RET
```

```
Programa: FILL
Start: &A005    Ende: &A016
Longitud: 0012
0 Errores
Tabla de variables:
START A001    ENDE A003    VALOR A000    BUCLE A00C
```

Confeccione un cargador BASIC para este programa integrando el programa FILL.

```
20 FOR I=&A000 TO &A016: READ A$: A=VAL("&"+A$): POKE I,A: NEXT
25 DATA FF,00,C0,FF,FF
30 DATA 3A,00,A0,ED,5B,01,A0,12
40 DATA 13,2A,03,A0,ED,52,20,F7
50 DATA 12,C9
```

Como ya hemos mencionado, el presente programa es la posibilidad más simple para realizar la rutina FILL. Pero resulta excesivamente largo y lento.
La solución más rápida la tendremos utilizando comandos de carga de bloques. Para rellenar un área hemos de utilizarlos concientemente de manera incorrecta (vean capítulo 4.3).

Programa fuente:

```
      (Líneas 10-70 como el anterior)
80  'SBC HL,DE        ; Longitud del bloque
90  'LD  B,H          ; Contador de bytes
100 'LD  C,L          ; Cargado con longitud bloque
110 'LD  H,D          ; Bloque fuente inicial (HL)
```

```
120 'LD  L,E          ; Cargar con direccion inicial
130 'INC DE           ; Direccion destino = direccion inicial + 1
140 'LD  (HL),A       ; Cargar primer byte fuente con valor
150 'LDIR
160 'RET
170 'END
```

Traduzca el programa máquina para un cargador BASIC. Arranque el programa BASIC seleccionando la dirección inicial &C000, la dirección final &CFFF y el valor &FF.
El bloque se encuentra en el área de pantalla. Valor = &FF = &X1111 1111 corresponde a 8 puntos activados. Como resultado debemos obtener líneas horizontales de un punto de espesor.

Rutina TRANSFER

A continuación utilizaremos "correctamente" los comandos de carga de bloques, para confeccionar una rutina de transferencia. Este programa debe transferir un área de memoria a otra posición. Con la ayuda de un programa BASIC debe proporcionarse la dirección inicial y final del bloque fuente así como la dirección inicial del bloque destino, comprobando que son correctas. Para la transferencia utilizamos las direcciones siguientes:

Inicio bloque fuente: &A020/&A021
Final bloque fuente: &A022/&A023
Inicio bloque destino: &A024/&A025

La dirección inicial del programa máquina es entonces &A026.

El bloque fuente y el bloque destino no deben solaparse, de esta forma el bloque destino siempre tendrá los valores correctos aunque el bloque fuente sea sobreescrito.

Programa Fuente:

```
5  'RUTINA DESPLAZAMIENTO BLOQUE
10 'QANF    EQU  &A020          ; Direccion inicial bloque fuente
20 'QEND    EQU  &A022          ; Direccion final bloque fuente
30 'ZANF    EQU  &A024          ; Direccion inicial bloque destino
40 'ORG     EQU  &A026          ; Inicio programa
45 ' ; INICIO PROGRAMA, OBTENER LONGITUD BLOQUE
50 '        LD   HL,(QEND)
60 '        LD   DE,(QANF)
70 '        OR   A              ; Borrar carry para SBC
80 '        SBC  HL,DE          ; = Longitud bloque - 1
90 '        INC  HL             ; + 1 = longitud bloque
100 '       LD   B,H            ; Longitud bloque
110 '       LD   C,L            ; Almacenado en BC
115 ' ; Decision por incrementar o decrementar
120 '       LD   HL,(QANF)
130 '       SBC  HL,DE          ; ZANF menor que
140 '       JR   C,LADINC       ; QANF, entonces LADINC
150 '       SBC  HL,BC          ; Diferencia menor que
160 '       JR   NC,LADINC      ; Longitud bloque, entonces LADINC
170 '       LD   HL,(ZANF)
180 '       ADD  HL,BC          ; ZANF mas longitud
190 '       DEC  HL             ; - 1 = Final bloque destino
200 '       EX   DE,HL          ; Cargado de HL hacia DE
210 '       LD   HL,(QEND)      ; Final bloque fuente
220 '       LDDR
230 '       RET
240 ' ; CARGA BLOQUE INCREMENTAR
250 'LADINC EX   DE,HL          ; Inicio fuente de DE hacia HL
260 '       LD   DE,(ZANF)
270 '       LDIR
280 '       RET
290 '       END
```

El inicio y final del programa no necesitan mayores explicaciones. La porción intermedia es algo más difícil dado que es aquí donde se decide si debe utilizarse el comando LDDR o bien LDIR (líneas 115-160). Observemos la necesidad de esta decisión (capítulo 2.3). En el caso normal, es decir si no hay solapamiento de los bloques utilizamos el comando LDIR. Si la dirección inicial de destino es menor que la dirección inicial del bloque fuente, puede también utilizarse el comando LDIR. Mediante la sustracción de la línea 130 y la bifurcación de la línea 140, se dirige a carga bloque incrementar en el caso de Zanf menor que Qanf. Si Zanf es mayor o igual que Qanf, ha de determinarse si Zanf es menor o igual que Qend.

```
                    Zanf <= Qend
                    Zanf <= Qanf + Longitud - 1
Zanf - Qanf - Longitud <= -1
             HL - BC <= -1
```

Si después de HL - BC queda el carry activado (resultado menor o igual que -1), debe entonces utilizarse el comando LDDR. Si el carry es igual 0, HL - BC tuvo un resultado mayor o igual a 0, por lo tanto Zanf no se encuentra en el bloque fuente, bifurcando seguidamente a LDIR.
Para incorporar este programa en el monitor debemos almacenarlo er líneas DATA. En un programa de esta longitud se producen frecuentemente errores; para evitarlos tenemos 2 posibilidades. Durante la lectura de las líneas DATA se suman todos los valores leídos y la suma final se comprueba con una suma de comprobación. Si la suma final no coincide con la suma de comprobación, se detecta un error. En nuestro caso ello se representa de la siguiente manera:

```
10 FOR I=&A020 TO &A051
20 READ A$: A=VAL("&"+A$): POKE I,A: S=S+A: NEXT
30 DATA 00,80,FF,BF,00,C0
40 DATA 2A,22,A0,ED,5B,20,A0,B7
50 DATA ED,52,23,44,4D,2A,24,A0
```

```
60 DATA ED,52,38,10,ED,42,30,0C
70 DATA 2A,24,A0,09,2B,EB,2A,22
80 DATA A0,ED,B8,C9,EB,ED,5B,24
90 DATA A0,ED,B0,C9
100 IF S <> 5186 THEN PRINT "ERROR EN DATAS" ELSE PRINT "OK!"
```

Para nosotros es más simple la segunda posibilidad:
Después del listado de ensamblador del programa, seguramente habrá grabado el código objeto generado en el cassette (diskette). Con *LOAD "Nombre programa"* puede cargarse este programa desde un programa BASIC.
Un programa monitor debería ofrecer asimismo la posibilidad de cargar y almacenar programas en lenguaje máquina.

Con la ayuda de LOAD "Nombre",Dirección y
SAVE "Nombre",B,Dirección inicial,Longitud

Podemos realizarlo fácilmente.
Si agrupamos todas las funciones su monitor "conoce" los comandos siguientes:

- M - (inglés Monitor) - Visualizar área memoria
- F - (inglés Fill) - Rellenar área con valor (V)
- T - (inglés Transfer) - Desplazar áreas
- L - (inglés Load) - Cargar programas máquina
- S - (inglés Save) - Grabar programas máquina

Rutina COMPARE

Nos ocuparemos a continuación de la rutina COMPARE, que se utiliza para la comparación de 2 áreas de memoria. La abreviación del comando es C. Como entradas desde un programa BASIC, la rutina necesita la dirección inicial y final del bloque fuente y la dirección inicial del bloque a comparar. Todas las direcciones del bloque a comparar en las cuales los valores almacenados no coincidan con los correpondientes del bloque fuente, deben visualizarse.

Programa Fuente

```
10 '         ORG  &A060
20 'FLAG     DB   1
30 'ANF      DS   2
40 'ENDE     DS   2
50 'ANFVER   DS   2              ; Inicio Bloque comparacion
60 '         LD   DE,(ANF)
70 '         LD   HL,(ENDE)
80 '         OR   A
90 '         SBC  HL,DE          ; = Longitud bloque
100 '        INC  HL             ; + 1
110 '        LD   B,H
120 '        LD   C,L            ; Almacenado en BC
130 '        EX   DE,HL          ; ANF hacia HL
140 '        LD   DE,(ANFVER)    ; Apuntador bloque
150 'WEITER  LD   A,(DE)         ; Elemento de comparacion
160 '        INC  DE
170 '        CPI                 ; Comparacion (HL) con A
180 '        JR   NZ,AUSGA       ; Desigual entonces salida
190 '        JP   PE,WEITER      ; Siguiente elemento
200 '        LD   A,B
210 '        LD   (FLAG),A       ; Final, flag = 0
220 '        RET
230 'AUSGA   LD   (ANF),HL
240 '        LD   (ANFVER),DE
250 '        RET  PE             ; Aun no final bloque
260 '        DEC  B              ; B=255
270 '        LD   A,B
280 '        LD   (FLAG),A       ; Flag=255
290 '        RET                 ; Final bloque
300 '        END
```

En las líneas 20-50 se reserva espacio de memoria para los datos a transferir. Para ello se utilizan los pseudo-comandos. El comando DB (Define Bytes) almacena el valor indicado como operando en la dirección actual. En nuestro caso se almacena el valor 1 en

dirección &A060. Esta posición de memoria sirve como flag para la comunicación con el programa BASIC. El flag puede obtener los valores siguientes:

1 - Se ha detectado desigualdad en la comparación, el bloque no se ha comparado aún en su totalidad.
0 - El bloque se ha comparado íntegramente
255 - El bloque se ha comparado íntegramente y en el último elemento del mismo se ha detectado desigualdad.

En las líneas 30 y 50 se encuentran pseudo-comandos DS (Define Storage): reserva espacio de memoria. El pseudo-comando DS indica al ensamblador que referencie el mpc para incrementar el número determinado de posiciones de memoria. De esta manera se reserva este espacio y podemos almacenar en el mismo las variables de transferencia. En nuestro caso necesitamos para el almacenamiento de ANF, ENDE y ANFVER 2 bytes para cada una (Low y High bytes de la dirección), y por ello hemos utilizado DS 2.

En las líneas 60-120 cargamos el byte counter BC con la longitud del bloque fuente. El comando INC HL de línea 100 es necesario, ya que de lo contrario no se compararía el último elemento.

En línea 130 se carga HL con la dirección inicial del bloque fuente y en línea 140 se carga DE con la dirección inicial del bloque a comparar.

A partir de línea 150 comienza el bucle principal del programa. En primer lugar cargamos el ACU con el valor del bloque de comparación (150), y el apuntador en el bloque de comparación se incrementa (160). CPI tiene varias funciones. Compara el contenido del ACU (= valor de un elemento del bloque de comparación) con el valor en dirección HL (= valor de un elemento del bloque a comparar). Según el resultado de la comparación se altera el flag Z. Seguidamente se incrementa HL y se decrementa BC. Si BC resulta igual 0 se reinicializa el P/V (PO), de lo contrario se activa (PE).

En línea 180 se bifurca a la salida si los valores comparados han resultado diferentes. De lo contrario se repite el bucle descrito mediante línea 190, si P/V = 0, es decir valor PE. Si por el contrario P/V es 1, el flag es 0 según línea 200 resultando un retorno al BASIC.

A partir de línea 220 comienza la porción de programa para la salida.
En primer lugar se almacenan los apuntadores actuales de bloque. DE contiene la dirección de la posición de memoria incrementada en 1, que no representa igualdad. Después de la salida de esta dirección mediante el programa BASIC, se llama nuevamente a la rutina continuando en la posición correcta, dado que ANF y ANFVER se han actualizado antes de bifurcar al BASIC mediante las líneas 230 y 240. Si el bloque no ha sido comparado totalmente, es decir BC <> 0 y P/V = 1 o bien PE, se ejecuta el comando RET. Si por el contrario se ha comparado ya el último elemento (no hay igualdad) entonces a través de las líneas 200-280 el flag V obtendrá el valor 255 diferenciando de esta forma este caso del de comparación completa con resultado de igualdad.

```
A060             10          ORG   &A060
A060 01          20  FLAG    DB    1
A061             30  ANF     DS    2
A063             40  ENDE    DS    2
A065             50  ANFVER  DS    2            ; Inicio bloque comp.
A067 ED5B61A0    60          LD    DE,(ANF)
A06B 2A63A0      70          LD    HL,(ENDE)
A06E B7          80          OR    A
A06F ED52        90          SBC   HL,DE        ; Longitud bloque
A071 23          100         INC   HL           ; HL + 1
A072 44          110         LD    B,H
A073 4D          120         LD    C,L          ; Cargar hacia BC
A074 EB          130         EX    DE,HL        ; ANF hacia HL
A075 ED5B65A0    140         LD    DE,(ANFVER)  ; Apuntador bloque
A079 1A          150 WEITER  LD    A,(DE)       ; Ele. comparacion
A07A 13          160         INC   DE
```

```
A07B EDA1       170         CPI              ; Comp. (HL) con A
A07D 20FE       180         JR   NZ,AUSGA    ; Salida por desigual
A07F EA79A0     190         JP   PE,WEITER   ; Siguiente ele.
A082 78         200         LD   A,B
A083 2260A0     210         LD   (FLAG),A    ; Final FLAG=0
A086 C9         220         RET
**** LINEA 180: AUSGA = A087
A087 2261A0     230 AUSGA   LD   (ANF),HL
A08A ED5365A0   240         LD   (ANFVER),DE
A08E E8         250         RET  PE          ; Aun no es fin blq.
A08F 05         260         DEC  B           ; B=255
A090 78         270         LD   A,B
A091 3260A0     280         LD   (FLAG),A
A094 C9         290         RET              ; Fin bloque


Programa: COMPARE
Start: &A060    Ende: &A094
Longitud: 0035
0 Errores
Tabla de variables:
FLAG   A060   ANF    A061    ENDE  A063    ANFVER A065
WEITER A079   AUSGA A087
```

El programa BASIC para la llamada de esta rutina es el siguiente:

```
10 REM COMPARE
20 MEMORY &9FFF
30 MODE 2
40 POKE &A060,1
50 INPUT "INICIO BLOQUE :&",A$
60 ADR=&A061: GOSUB 170
70 INPUT "FINAL BLOQUE   :&",A$
80 ADR=&A063: GOSUB 170
90 INPUT "INICIO BLOQUE COMPARACION :&",A$
100 ADR=&A065: GOSUB 170
110 CALL &A067
```

```
120 W=PEEK(&A060)
130 IF W=0 THEN END
140 PRINT HEX$(PEEK(&A061)+256*PEEK(&A062)-1,4)
150 IF W=1 THEN 110
160 END
170 A=VAL("&"+A$)
180 IF A<0 THEN A=A+2^16
190 AH = INT(A/256)
200 POKE ADR,A-AH*256
210 POKE ADR+1,AH
220 RETURN
```

El comando GO (G), la llamada de un programa máquina desde el programa monitor (por ejemplo para test), puede programarse fácilmente con el comando BASIC *CALL Dirección* y la correspondiente rutina de entrada para la dirección (V).

En el programa COMPARE, las conmutaciones entre BASIC y lenguaje máquina son bastante laboriosas y poco comprensibles. La unión entre lenguaje máquina y BASIC es necesaria porque no podemos aún programar entradas y salidas (es decir Inputs/Print) en lenguaje máquina. Estas rutinas son relativamente complejas. Para visualizar por ejemplo un carácter en la pantalla debe calcularse la posición correcta del carácter teniendo en cuenta la diferencia por scrolling. A continuación deben extraerse los 8 bytes necesarios para la representación del carácter desde la memoria de caracteres (ROM &3800 hasta &3FFF) y almacenar todo ello en la memoria de pantalla. Dado que la salida de un carácter en la pantalla funciona al conectar el ordenador, la rutina correspondiente ha de estar incorporada previamente al ROM. Conociendo esta rutina o por lo menos su dirección inicial, podríamos llamarla directamente desde nuestro programa en lenguaje máquina. Esta posibilidad de llamar rutinas del sistema con la ayuda del lenguaje máquina, es muy útil e interesante.

CAPITULO VI: UTILIZACION DE RUTINAS DEL SISTEMA

6.1 EL DISASSEMBLER EL SIMULADOR PASO A PASO

El CPC posee 32k de ROM. Estos 32 kilobytes contienen rutinas del sistema. Los 16 kbytes superiores del ROM (&C000 hasta &FFFF) contienen el BASIC, los 16 kbytes inferiores (&0 hasta &3FFF) contienen el sistema operativo del ordenador. El sistema operativo incluye muchas rutinas que son de interés para el programador de lenguaje máquina.

Para analizar estas rutinas necesitamos otra "herramienta" adicional, el disassembler.

Un disassembler interpreta los bytes de un área determinada de código máquina y traduce los códigos a los correspondientes comandos de ensamblador. Así el disassembler realiza la tarea inversa que el ensamblador. Con el disassembler podemos traducir programas máquina cargados en el ordenador, a sentencias de ensamblador. De igual modo pueden traducirse rutinas internas del sistema. De estos programas confeccionados por profesionales pueden copiarse algunas cosas y además podemos utilizar las rutinas en nuestros propios programas.

El programa siguiente "Disassi+Simula" contiene entre otras cosas un disassembler. Después de entrar el programa arránquelo con *RUN*; el asterisco que aparece en el borde izquierdo indica que se espera una entrada. En las primeras líneas de la pantalla se visualiza además "Modalidad de Entrada". Elija entonces en primer lugar el disassembler mediante la letra "d".

La "D" que se visualiza indica que ha sido reconocida su inserción. Introduzca a continuación del carácter & la dirección del desensamblaje. Pruebe con 20 y pulse *RETURN* o *ENTER*.

Un segundo carácter & indica que además debe introducir la dirección final. Introduzca 22 para esta primera prueba. A continuación comienza el disassembler.

```
0020 C3C6BA     JP &BACB
(En el 664  JP &BAC6 / en el 6128 JP &BAC6)
```

Continúe el desensamblaje en la dirección de bifurcación indicada. Para ello debe llamar nuevamente el disassembler mediante "D". Existe además una posibilidad más cómoda:

Si se encuentra en la modalidad de entrada, puede llamar nuevamente la función recién ejecutada pulsando *RETURN* o *ENTER*. Introduzca ahora la dirección de bifurcación que acaba de traducir y concluya con *RETURN* o *ENTER*.

La introducción de la dirección final puede resolverla simplemente pulsando *RETURN* o *ENTER*. A continuación se suma &18 a la dirección inicial introducida, que tiene como consecuencia el desensamblaje en la mayoría de los casos de "una pantalla completa".

Obtendrán lo siguiente:

```
BACB F3          DI
BACC D9          EXX
BACD 59          LD   E,C
BACE CB D3       SET  2,E
BAD0 CB DB       SET  3,E
BAD2 ED 59       OUT  (C),E
BAD4 D9          EXX
BAD5 7E          LD   A,(HL)
BAD6 D9          EXX
BAD7 ED 49       OUT  (C),C
BAD9 D9          EXX
BADA FB          EI
BADB C9          RET
BADC D9          EXX
```

```
BADD 79                    LD  A,C
BADE F6 0C                 OR  &0C
BAE0 ED 79                 OUT (C),A
BAE2 DD 7E 00              LD  A,(IX+0)
```

Las direcciones del principio de cada linea en su caso serán diferentes, si no utilizan el CPC 464. Los comandos traducidos son iguales en todos los ordenadores CPC.

Esta rutina del sistema que acabamos de traducir, se utiliza para la lectura del RAM. El valor de la dirección HL del RAM se carga en el ACU independientemente del estado ROM/RAM. La rutina se llama a través del comando RST &20.

Si quiere continuar desensamblando introduzca *RETURN* (para desensamblador), *RETURN* para dirección inicial, lo que significa que se continúa a partir de la última dirección traducida, y *RETURN* para la dirección final, según el significado ya expuesto.

De esta manera puede desensamblar fácilmente programas continuos.

Veamos ahora otras funciones del programa:

Al teclear "H" (HELP) obtendrán la lista de todos los comandos que ofrece este programa.

Con "P" (PRINT) pueden activar/desactivar la impresora. El estado actual se señala en la esquina superior derecha.

Con "B" se finaliza el programa.

Otra función importante del desensamblador es "R" que representa el estado ROM/RAM. Introduzca ahora "R":

El "RAM" después de "lo:" se representa en inverso. "lo:" significa que nos referimos al estado del Lower ROM, es decir las direcciones &0 a &3FFF. Pulsando la barra de espacio podemos llegar del estado RAM al ROM y viceversa.
Si ha seleccionado el Lower ROM llegará pulsando *RETURN* al Hi-RAM (&C000 hasta &FFFF). La barra de espacio tiene en este caso la misma función explicada anteriormente.

También existe la posibilidad de activar alguno de los 252 ROM de expansión. Entre estos se encuentra por ejemplo el ROM de diskette que tiene el número 7. Para seleccionar un ROM de expansión se debe introducir *CTRL+E*.

A continuación puede introducirse el número deseado después de "hi:" por ejemplo 7 para el ROM de diskette. Queda entonces seleccionado el ROM deseado para las direcciones &C000 a &FFFF. Todas las restantes direcciones acceden al RAM. La ampliación del RSX, RDEEK que también puede "PEEKear" los ROMS se explica en el capitulo RSX.

Otra función relacionada directamente con el desensamblador es el generador de fuentes.

El generador de fuentes, después de ser llamado con "S" genera un listado fuente de las direcciones indicadas, que se almacena a partir de línea 10000. Este programa fuente puede modificarse directamente a través del ensamblador. De esta manera tenemos un método simple para por ejemplo utilizar rutinas del sistema para programas propios sin necesidad de introducirlas nuevamente. Los programas fuente generados pueden modificarse más adelante parà adaptarlos a las necesidades individuales.

Hagamos una prueba:

Seleccione con "R" etc. el Lower ROM. Saltee la pregunta "Definir nombre de label?" pulsando *,RETURN*. A continuación introduzca 1D2 como dirección inicial y 1E1 como dirección final.

Finalice la ejecución con "B" y liste a continuación las líneas a partir de 10000. El programa fuente generado puede almacenarse separadamente y utilizarse en el momento oportuno.

Si la porción a desensamblar contiene direcciones conocidas, es decir a las cuales puede asignarse un label, puede hacerlo después de la pregunta "Definir nombre del label:".

La rutina &BCCB se denomina frecuentemente como "ROMWALK" dado que comprueba la distribución actual de memoria. Asignaremos ahora a la dirección &BCCB el label ROMWALK.

Seleccione el hi-ROM y llame al generador de fuente. A continuación introduzca:

ROMWALK,&BCCB

Continúen con este procedimiento para definir otros labels, finalizando la entrada con "," y *RETURN*.

Introduzca ahora como dirección &C006 a &C00C y compruebe el resultado.
La rutina que genera líneas BASIC desde el BASIC, como sucede con el generador de fuentes, se explica más adelante en este libro.

Las funciones restantes del programa, se refieren todas al simulador paso a paso.

EXPLICACION DEL SIMULADOR PASO A PASO

Una de las dificultades en la programación con ensamblador consiste en que no existe una forma simple de comprobación de los programas confeccionados. Un error que en BASIC podría llevar a un "Syntax error", en ensamblador lleva en la mayoría de los casos a la caída del sistema. Por ello el tiempo de desarrollo de un programa es mucho mayor y la búsqueda de errores más difícil.

Para detectar un error en el programa sería de gran utilidad poder ejecutar el programa máquina, paso a paso comprobando cada vez que el contenido de los registros sea correcto. Sin este simulador deben realizarse estas comprobaciones manualmente.

Las funciones de simulación del programa realizan esta pesada tarea automáticamente. Seleccione los ROMS, llame el programa simulador con "Z" e introduzca 1D7.

```
SZ H PNC A  B C  D E  H L   IX   IY
00000000 00 0000 0000 0000 0000 0000
SZ H PNC' A' B'C' D'E' H'L' SP
00000000  00 0000  0000  0000 9FF0

01D7  23           INC   HL
```

En esta posición el "*" indica que se espera una entrada. Están previstas las entradas siguientes:

L espacio en blanco
A
E

Pulsando la barra de espacio o la tecla RETURN, continúa la simulación. Se visualizan los contenidos de registros eventualmente modificados y el comando siguiente.

```
SZ H PNC A  B C  D E  H L   IX   IY
00000000 00 0000 0000 0000 0000 0000
SZ H PNC' A' B'C' D'E' H'L' SP
00000000  00 0000  0000  0000 9FF0

01D8  70           LD    (HL),B
```

La tecla A tiene una función importante:

Pulsando la tecla A, el programa de simulación se altera a modalidad editor. De esta manera tiene la posibilidad de modificar los contenidos de registros y la dirección del programa.

Proceda entonces en el orden exacto que a continuación detallamos:

Después de pulsar la tecla A, el cursor se encuentra debajo de la S (Sign Flag) de la última salida de registro. Los contenidos indicados en esta línea corresponden a los registros señalados. En esta línea pueden realizar modificaciones introduciendo contenidos de registro en formato hexadecimal (indistintamente en mayúscula o minúscula) en las posiciones determinadas. La entrada para el registro flag, ha de ser naturalmente binaria, es decir con ceros y unos. Contenidos actuales pueden aceptarse con copy. Si copy no se desplaza introduzca previamente "Shift+Cursor derecha" y "Shift+Cursor izquierda". Un comantario para la visualización:

Al principio de la línea se encuentra el registro flag como número binario. Sobre los bits individuales puede ver su significado:

S - Sign Flag
Z - Zero Flag
H - Half overflow Flag
P - P/V Parity/Overflow Flag
N - BCD Subtraction Flag
C - Carry Flag

A continuación viene el acumulador A (8 bits) y los registros universales B hasta L, conjuntamente con registros de 16 bits, seguidos por los registros IX e IY.
Se repite aquí el orden secuencial de los registros para el segundo set de registros. Al final de la línea se encuentra el apuntador del stack.

El apuntador del stack o pila (SP) nunca debe contener direcciones menores a &.... Su valor máximo es la dirección inicial de un programa máquina eventualmente cargado en el ordenador, es decir en la mayoría de los casos &A000. Con valores menores que &.... el programa máquina, que realiza la simulación se sobrescribé, lo que lleva infaliblemente a la caída del sistema. Eventualmente debe modificarse el SP, lo que debe efectuarse únicamente al comienzo de una simulación. porque de lo contrario se altera la secuencia

de las direcciones de retorno. Se debe repasar la línea integra mediante COPY o introducirla directamente. Si ha finalizado una línea pulse *RETURN* para llegar a la siguiente línea. No utilice las teclas Cursor u otras teclas que pueden cargar en los registros valores no deseados.

Después de terminadas las modificaciones de la línea anterior con *RETURN*, el cursor se posiciona al comienzo de la línea que contiene el comando a traducir. En esta línea puede modificar ahora el PC, es decir los primeros cuatro números hexadecimales de la línea. La dirección introducida es la dirección de continuación de la simulación una vez pulsado *RETURN*. En caso de no desear la modificación de esta dirección, debe de todas formas aceptarse la misma mediante *COPY*.

De esta forma finaliza la modalidad de edición iniciada mediante la tecla A y la simulación continúa.

La última posibilidad de operación es la tecla E.

La tecla E conmuta entre simulación real o pseudosimulación. Simulación real significa que todos los comandos se ejecutan realmente. En el caso de algunos comandos, este tipo de ejecución puede tener fatales consecuencias. Entre estos comandos "peligrosos" se encuentran todos aquellos que alteran realmente la configuración del sistema. Por ejemplo, los comandos que modifican los contenidos de memoria como LD (&B1DB),A. Los contenidos de memoria así modificados pueden alterar informaciones importantes utilizadas por el interpretador BASIC o por el sistema operativo, provocando errores insalvables. También los comandos I/O, que sirven entre otras cosas para la selección de ROM/RAM pueden llevar inmediatamente a la caída del sistema si se simulan en forma real. Atención asimismo con los comandos que se refieren directamente al SP.

Por este motivo se proporciona la opción mediante la tecla E, de seleccionar la modalidad de simulación antes de ejecutar el comando.

En el extremo superior derecho de la pantalla se indica si la simulación es real o pseudosimulación.

Veamos el ejemplo de la simulación de la rutina TEST HL-DE a partir de la dirección ROM &FF. Seleccione el HI-ROM y active la simulación real. LLame el simulador mediante 7 e introduzca &D232 (664:&D282/6128:&D282) como dirección inicial. Compare su resultado con el listado que ofrecemos a continuación. Excepto las direcciones de bifurcación y las direcciones de los comandos, su pantalla debería coincidir con lo expuesto abajo (en el caso del 464 la coincidencia debe ser total!).

```
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 0000 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
D232  11 4F 00      LD    DE,&004F
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
D235  CD B8 FF      CALL  &FFB8
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFB8  7C            LD    A,H
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFB9  92            SUB   D
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
01000010  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBA  C0            RET   NZ
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
01000010  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBB  7D            LD    A,L
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
01000010  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBC  93            SUB   E
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
10110011  B1 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBD  C9            RET
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
10110011  B1 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
D238  D2 AB CE      JP    NC,&CEAB
```

En primer lugar DE se carga con &004F. Observe la modificación de lo expuesto debajo de DE. A continuación verán la llamada de la rutina TEST HL-DE. Después de este comando se decrementa SP, porque se ha almacenado la dirección de retorno en la pila. La rutina realiza la sustracción HL-DE. Como era de esperar se activa el carry flag (=1) en el momento del retorno, indicando que DE es mayor que HL.

Pruebe usted la misma simulación a partir de la dirección inicial. Cargue previamente HL y DE con valores diferentes (&H101, &H1000, &H80) y observe la influencia en los flags. El comando RET o RETNZ finaliza la rutina. La dirección de retorno se extrae de la pila y continúa la ejecución del programa con el comando siguiente después de RST &H20.

Si tiene dudas en programas o comandos utilice el simulador para comprobar su procedimiento.

DESCRIPCION DEL PROGRAMA

Antes de la explicación sistemática del programa, aclararemos algunas cosas aun del funcionamiento del simulador.

El núcleo del simulador consiste en un programa máquina que se carga al final del programa. Se puede decir que POKEAMOS el comando a ejecutar al centro de este programa. Previamente se cargan los registros con los valores correspondientes. Después de la ejecución del comando, los contenidos de registros eventualmente alterados se almacenan en la memoria para traspasarlos al BASIC. El simulador llama a este programa, mediante CALL &9700 en la línea 1140. Para que se comprenda mejor, ofrecemos el listado en ensamblador del programa máquina.

Reflexione sobre el funcionamiento del programa observando este

listado. Si desea simularlo ha de desactivar la simulación real de los comandos que alteran el SP y realizar las modificaciones mediante "A". Para mantener breve el programa, no almacenamos los contenidos de registros en la memoria mediante comandos LD, los grabamos y leemos mediante comandos PUSH y POP. Para ello inicializamos el SP con el inicio del área reservada para la transferencia (SPUELO-SP traspaso dirección Low).

Con la ayuda de este método pueden ejecutarse todos los comandos excepto los comandos de bifurcación, que nos llevan fuera del programa máquina haciendo imposible el retorno al BASIC.

Dado que el segundo set de registros del Z80-A del CPC se reserva para rutinas del sistema, almacenamos el contenido del mismo antes del programa mediante PUSH y lo extraemos al final mediante POP.

La modificación del segundo set original de registros mediante programas propios puede llevar a la caida del sistema. El siguiente listado se ha confeccionado con el CPC 6128 y las modificaciones para el 464 y el 664 se hallan indicadas en el programa.

```
A000              10                    ; Simulador
9F70              20          ORG   &9F70
9F70 F3           30          DI        ; Desactivar interrupt
9F71 D9           40          EXX       ; Uso del 2do set registros
9F72 08           50          EX    af,af'
9F73 F5           60          PUSH  af  ; Salvar segundo set registros
9F74 C5           70          PUSH  bc  ; en la pila (STACK)
9F75 D5           80          PUSH  de  ; antes de la utilizacion
9F76 E5           90          PUSH  hl
9F77 08          100          EX    af,af'
9F78 D9          110          EXX
9F79 ED730000    120          LD    (Spreal),sp ; Salvar SP real
9F7D 310000      130          LD    sp,spuelo   ; SP trasp. reg.
9F80 F1          140          POP   af
9F81 C1          150          POP   bc  ; Leer registro
9F82 D1          160          POP   de
9F83 E1          170          POP   hl
9F84 D9          180          EXX
9F85 08          190          EX    af,af'
9F86 DDE1        200          POP   ix
9F88 FDE1        210          POP   iy
9F8A F1          220          POP   af
9F8B C1          230          POP   bc
9F8C D1          240          POP   de
9F8D E1          250          POP   hl
9F8E ED7B0000    260          LD    sp,(spsimu) ; Carga SP simul.
```

```
9F92 0000       270 BEFEHL DW    0  ; Reservar espacio
9F94 0000       280        DW    0  ; para comando
9F96 ED730000   290        LD    (spsimu),sp  ; Almac. Spsimu
9F9A 310000     300        LD    sp,spsimu    ; para trasp. reg.
9F9D E5         310        PUSH  hl
9F9E D5         320        PUSH  de ; Regraba registro
9F9F C5         330        PUSH  bc
9FA0 F5         340        PUSH  af
9FA1 FDE5       350        PUSH  iy
9FA3 DDE5       360        PUSH  ix
9FA5 08         370        EX    af,af'
9FA6 D9         380        EXX
9FA7 E5         390        PUSH  hl
9FA8 D5         400        PUSH  de
9FA9 C5         410        PUSH  bc
9FAA F5         420        PUSH  af
9FAB ED7B0000   430        LD    sp,(spreal) ; Recup. SP real
9FAF D9         440        EXX
9FB0 08         450        EX    af,af' ; Cargar segundo set
9FB1 E1         460        POP   hl     ; de registros con
9FB2 D1         470        POP   de     ; valores iniciales
9FB3 C1         480        POP   bc
9FB4 F1         490        POP   af
9FB5 08         500        EX    af,af'
9FB6 D9         510        EXX
9FB7 FB         520        EI       ; Liberar interrupt
9FB8 C9         530        RET
```

```
**** Linea     120   : SPREAL=&9FB9
**** Linea     430   : SPREAL=&9FB9; Liberar interrupt
9FB9           540     SPREAL DS 2
**** Linea     130   : SPUELO=&9FBB
9FBB           55      SPUELO DS 20
; Espacio para transferencia de registros
**** Linea     260   : SPSIMU=&9FCF
**** Linea     290   : SPSIMU=&9FCF
**** Linea     300   : SPSIMU=&9FCF
9FCF 0000      560     SPSIMU DW pila
; Espacio para SP de simulacion
9FD1           570             DS 4
; Espacio para desbordamiento pila
**** Linea     560   : PILA=&9FD5
9FD5           580     PILA DS 50
; Espacio para pila de simulacion

Programa : Simula
Start    : &9F70    End : &A006
Longitud : 0097
0 Errores
Tabla de variables:
BEFEHL   9F92   SPREAL 9FB9    SPUELO 9FBB    SPSIMU 9FCF
PILA     9FD5
```

```
10 'Z80 - Desensamblador, Simulador y Generador de fuente
20 ' de H.D. 02/10/1985
30 MEMORY &9EFF
40 MODE 2
50 GOSUB 2760:REM Init
60 REM ENTRADA
70 PRINT pri$:LOCATE #3,1,VPOS(#0):PRINT#3,"*";
80 CLS#1:PRINT#1,"Modalidad de ENTRADA"
90 a$=INKEY$:IF a$="" THEN 90
100 IF a$=CHR$(13) THEN 120
110 FOR me=0 TO bean:IF(ASC(a$)OR &20)<>(ASC(menu$(me))OR &2
0) THEN NEXT:GOTO 70
120 CLS#1:PRINT#1,menu$(me);:CLS#3
130 PRINT LEFT$(menu$(me),1);" ";
140 ON me+1 GOSUB 380,1600,1510,510,630,580,1260,210,250,190
150 IF NOT zflag THEN zflag=0 ELSE zflag=1
160 GOTO 70
170 ' ****************************************
180 REM end
190 MODE 2:END
200 ' ***************************************
210 REM Impresora activada/desactivada
220 IF aus=0 THEN PEN #2,0:PAPER #2,1:aus=8:PRINT#2,"an "; E
LSE PEN #2,1:PAPER #2,0:aus=0:PRINT#2,"aus";
230 GOSUB 460:RETURN
240 ' ***************************************
250 REM rom/ram Status
260 GOSUB 460:PEN #4,0:PAPER #4,1
270 LOCATE #4,1,1:PRINT #4,romstat$(lostat);
280 a$=INKEY$:IF a$="" THEN 280
290 IF a$=CHR$(13) THEN PEN #4,1:PAPER #4,0:LOCATE #4,1,1:PR
INT #4,romstat$(lostat); ELSE lostat=1-lostat:GOTO 270
300 PEN #5,0:PAPER #5,1:IF histat=2 THEN histat=1
310 LOCATE #5,1,1:PRINT #5,romstat$(histat);
```

```
320 a$=INKEY$:IF a$="" THEN 320
330 IF a$=CHR$(13) THEN PEN #5,1:PAPER #5,0:LOCATE #4,1,1:PR
INT #5,romstat$(histat);:GOTO 350
340 IF a$<>CHR$(5) THEN histat=1-histat:GOTO 310 ELSE PEN #5
,1:PAPER #5,0:CLS #5:INPUT #5,;"",exrom:histat=2
350 IF histat=2 THEN status=exrom ELSE status=252+lostat+his
tat*2
360 RETURN
370 ' ************************************
380 REM Help
390 CLS
400 LOCATE 10,2:PRINT"Z 8 0  -  D I S A S S E M B L E R  +
S I M U L A T O R"
410 LOCATE 1,5:FOR i=0 TO bean:LOCATE 15,VPOS(#0)
420 PRINT LEFT$(menu$(i),1);" / ";CHR$(ASC(menu$(i)) OR &20)
;" - ";menu$(i)
430 NEXT
440 RETURN
450 ' ++++++++++++++++
460 REM sub zflag
470 LOCATE 1,VPOS(#0):PRINT " ";CHR$(8);
480 IF zflag=1 THEN zflag=-1
490 RETURN
500 ' ************************************
510 REM Simulator
520 GOSUB 1790:IF noflag THEN pc=pc ELSE pc=a
530 zflag=-1:PRINT:GOTO 1140
540 FOR i=adrbef TO adrbef+3:POKE i,0:NEXT:qq=0
550 GOSUB 1880:REM Desensamblar
560 RETURN
570 ' ************************************
580 REM Simulador Real ein/aus
590 echtsim=-1-echtsim
600 IF echtsim THEN PEN#6,0:PAPER #6,1:PRINT#6,"an "; ELSE P
```

```
EN #6,1:PAPER #6,0:PRINT#6,"aus";
610 GOSUB 460:RETURN
620 ' ************************************
630 REM Espacio en blanco = Simular
640 IF ABS(zflag)<>1 THEN PRINT"Llamar primero simulador
  con Z":RETURN
650 GOSUB 460
660 IF NOT echtsim THEN 1140
670 IF po=0 THEN bef$=pr$:GOTO 690
680 bef$=LEFT$(LEFT$(pr$,po-1)+"    ",4)
690 REM Tratar de forma especial los comandos de bifurcacion
700 IF bef$="JP  " OR bef$="JR  " THEN GOSUB 970:GOTO 1140
710 IF bef$<>"RST " THEN 730
720 spradr=pc-dwflag:pc=VAL(MID$(pr$,5,4)):GOTO 790
730 IF bef$<>"DJNZ" THEN 770
740 b=PEEK(adrreg+15):b=b-1-256*(b=0):POKE adrreg+15,b
750 IF b<>0 THEN GOSUB 1050
760 GOTO 1140
770 IF bef$<>"CALL" THEN 860
780 spradr=pc:GOSUB 970
790 REM Almacenar direccion retorno en Stack (pila)
800 sp=PEEK(stapel)+256*PEEK(stapel+1)
810 sp=sp-1:POKE sp,INT(spradr/256)-256*(spradr<0)
820 sp=sp-1:POKE sp,spradr-256*INT(spradr/256)
830 POKE stapel,sp-INT(sp/256)*256
840 POKE stapel+1,INT(sp/256)-256*(sp<0)
850 GOTO 1140
860 IF LEFT$(bef$,3)<>"RET" THEN 1130
870 GOSUB 1060
880 IF bedflg=0 THEN 1140
890 REM Extraer direccion retorno del Stack (pila)
900 sp=PEEK(stapel)+256*PEEK(stapel+1)
910 pc=PEEK(sp+1)*256+PEEK(sp)
920 sp=sp+2
```

```
930 POKE stapel,sp-INT(sp/256)*256
940 POKE stapel+1,INT(sp/256)-256*(sp<0)
950 GOTO 1140
960   ++++++++++++++++
970 REM Sub pc activo en bifurcaciones
980 GOSUB 1060
990 IF bedflg THEN 1050
1000 IF bedflg=0 THEN RETURN
1010 a$=MID$(pr$,po+2,2)
1020 IF a$="HL" THEN pc=PEEK(adrreg+18)+256*PEEK(adrreg+19):
RETURN
1030 IF a$="IX" THEN pc=PEEK(adrreg+8)+256*PEEK(adrreg+9):RE
TURN
1040 IF a$="IY" THEN pc=PEEK(adrreg+10)+256*PEEK(adrreg+11):
RETURN
1050 pc=VAL(RIGHT$(pr$,5)):RETURN
1060 REM Sub test condicion
1070 a$=MID$(pr$,po+1,2)
1080 FOR i=0 TO 7:IF a$<>cond$(i) THEN NEXT
1090 IF i=8 THEN bedflg=1:RETURN
1100 bit=((i\2)*2-(i>3)-((i=4)OR(i=5)))
1110 IF (PEEK(adrreg+12)AND 2^bit)/2^bit=-(i MOD 2 =1) THEN
bedflg=-1 ELSE bedflg=0
1120 RETURN
1130 IF echtsim THEN CALL &9F70 : REM Ejecutar comando,
1140 REM Visualizar registros
1150 PRINT#aus,"SZ H PNC  A  B C  D E  H L   IX   IY  SZ H P
NC' A' B'C' D'E' H'L'  SP "
1160 FOR q=1 TO 0 STEP -1
1170 PRINT#aus,BIN$(PEEK(adrreg+q*12),8);"  ";HEX$(PEEK(adrr
eg+1+q*12),2);" ";
1180 FOR k=2 TO 6 STEP 2:pore=q*12+k:GOSUB 1220:NEXT
1190 IF q=1 THEN pore=8:GOSUB 1220:pore=10:GOSUB 1220 ELSE p
ore=20:GOSUB 1220
```

```
1200 NEXT:PRINT#aus
1210 GOTO 540
1220 REM Salida registro 16-bits
1230 PRINT#aus,HEX$(256*PEEK(adrreg+pore+1)+PEEK(adrreg+pore
),4);" ";
1240 RETURN
1250 ' ***********************************
1260 REM Modificar registros
1270 IF ABS(zflag)<>1 THEN PRINT"¡Llamar primero al simulador
  con Z":RETURN
1280 GOSUB 460
1290 LOCATE 1,VPOS(#0)-2
1300 LINE INPUT "",z$
1310 q=12:GOSUB 1400 : REM Primer set de registros
1320 q=0.1:po=29:GOSUB 1450 : REM IX
1330 po=34:GOSUB 1450 : REM IY
1340 q=0:GOSUB 1400 : REM Segundo set de registros
1350 q=-4:po=67:GOSUB 1450
1360 LINE INPUT "",z$
1370 pc=VAL("&"+LEFT$(z$,4))
1380 RETURN
1390 ' +++++++++++++++
1400 REM Sub transferencia registros modificados al Mapro
1410 POKE adrreg+1+q,VAL("&"+MID$(z$,11-(q=0)*38,2))
1420 POKE adrreg+q,VAL("&X"+MID$(z$,1-(q=0)*38,8))
1430 FOR po=14 TO 24 STEP 5:GOSUB 1450:NEXT
1440 RETURN
1450 REM Sub determinar valores de registros
1460 wert=VAL("&"+MID$(z$,po-(q=0)*38,4))
1470 POKE adrreg+q+(po\5)*2-1,INT(wert/256)-256*(wert<0)
1480 POKE adrreg+q+(po\5)*2-2,wert-INT(wert/256)*256
1490 RETURN
1500 ' ***********************************
```

```
1510 REM Generador fuentes Z80
1520 souflag=-1
1530 PRINT
1540 PRINT "Definicion nombre label:"
1550 FOR i=0 TO 50:INPUT la$(i),wa$(i):IF la$(i)="" THEN PRI
NT:GOTO 1580
1560 z$=STR$(zeino)+CHR$(32)+CHR$(39)+CHR$(32)+la$(i)+" EQU
"+wa$(i)+CHR$(0)+CHR$(0)
1570 CALL mst,@z$:zeino=zeino+10:NEXT
1580 maxlab=i-1
1590 ' ************************************
1600 REM Desensamblador
1610 IF me= 1 THEN souflag=0:maxlab=0
1620 GOSUB 1790:IF noflag THEN pc=pc ELSE pc=a
1630 GOSUB 1790:ende=a:IF noflag THEN ende=pc+&18
1640 PRINT
1650 IF NOT souflag THEN 1720
1660 z$=STR$(zeino)+CHR$(32)+CHR$(39)+CHR$(32)+";"+CHR$(0)+C
HR$(0)
1670 CALL mst,@z$
1680 zeino=zeino+10
1690 z$=""
1700 z$=STR$(zeino)+CHR$(32)+CHR$(39)+CHR$(32)
1710 FOR i=0 TO maxlab:IF pc=VAL(wa$(i)) THEN z$=z$+la$(i)+"
 " ELSE NEXT
1720 GOSUB 1880
1730 IF NOT souflag THEN 1760
1740 z$=z$+pr$+CHR$(0)+CHR$(0)
1750 CALL mst,@z$
1760 IF pc>ende THEN RETURN
1770 IF souflag THEN 1680 ELSE 1720
1780 ' +++++++++++++++
1790 REM Sub entrada direccion hexadecimal
1800 x=POS(#0)+1:y=VPOS(#0)
```

```
1810 LOCATE x,y:INPUT"&",a$
1820 IF a$="" THEN noflag=-1:GOTO 1850
1830 noflag=0
1840 a=VAL("&"+a$)
1850 LOCATE x+5,y
1860 RETURN
1870 '++++++++++++++++++++++++++++++++++++++++
1880 REM sub disassi
1890 adr=pc
1900 PRINT#aus,HEX$(adr,4);"  ";
1910 iflag=0
1920 GOSUB 2690
1930 GOSUB 2040
1940 IF iflag THEN 2350
1950 IF w=&CF OR w=&D7 OR w=&DF OR w=&EF THEN pr$=pr$+" /DW:
nn":dwflag=2 ELSE dwflag=0
1960 IF INSTR(pr$,"n")<>0 THEN 2460
1970 IF INSTR(pr$,"e")<>0 THEN 2600
1980 po=INSTR(pr$," ")
1990 IF PR$="" THEN PR$="???"
2000 IF po=0 THEN PRINT#aus,TAB(21);pr$;:GOTO 2020
2010 PRINT#aus,TAB(21);LEFT$(pr$,po-1);TAB(27);RIGHT$(pr$,LE
N(pr$)-po);
2020 PRINT#aus
2030 RETURN
2040 REM  Interpretar
2050 IF (w=&DD OR w=&FD) AND NOT iflag THEN 2240
2060 IF w=&ED THEN 2210
2070 IF w=&CB THEN 2150
2080 GOSUB 2290
2090 ON co1 GOTO 2110,2130,2100
```

```
2100 pr$=bef$(w):RETURN
2110 IF w=&76 THEN pr$="HALT":RETURN
2120 pr$="LD "+regtab$(co2)+","+reg$:RETURN
2130 IF co2=0 OR co2=1 OR co2=3 THEN a$=" A," ELSE a$=" "
2140 pr$=arilog$(co2)+a$+reg$:RETURN
2150 REM cb
2160 GOSUB 2690
2170 IF iflag THEN dis=w:GOSUB 2690
2180 GOSUB 2290
2190 IF co1=0 THEN pr$=rotschi$(co2)+" "+reg$ ELSE pr$=bitti
$(co1)+STR$(co2)+","+reg$
2200 RETURN
2210 REM ed
2220 GOSUB 2690
2230 IF w<&40 OR w>&BF THEN pr$="???":RETURN ELSE GOTO 2100
2240 REM xy
2250 iflag=-1
2260 IF w=&DD THEN i$="IX" ELSE i$="IY"
2270 GOSUB 2690
2280 GOTO 2040
2290 REM descomponer codigo
2300 co1=(w AND &X11000000)/64
2310 co2=(w AND &X111000)/8
2320 co3=w AND &X111
2330 reg$=regtab$(co3)
2340 RETURN
2350 REM indexado
2360 po=INSTR(pr$,"HL")
2370 IF po=0 THEN pr$="???":GOTO 1980
2380 IF INSTR(pr$,"(HL)")<>0 THEN 2420
2390 IF pr$="EX DE,HL" THEN pr$="???":GOTO 1980
2400 IF pr$="ADD HL,HL" THEN pr$="ADD "+i$+","+i$:GOTO 1980
2410 pr$=LEFT$(pr$,po-1)+i$+RIGHT$(pr$,LEN(pr$)-po-1):GOTO 1
 950
```

```
2420 IF LEFT$(pr$,2)="JP" THEN 2410
2430 IF pc-adr<3 THEN GOSUB 2690:dis=w
2440 IF dis>127 THEN dis$=STR$(dis-256) ELSE dis$="+"+RIGHT$
(STR$(dis),LEN(STR$(dis))-1)
2450 i$=i$+dis$:GOTO 2410
2460 REM Reemplazar n
2470 po=INSTR(pr$,"nn")
2480 IF po<>0 THEN 2530
2490 po=INSTR(pr$,"n")
2500 GOSUB 2690
2510 pr$=LEFT$(pr$,po-1)+"&"+HEX$(w,2)+RIGHT$(pr$,LEN(pr$)-p
o)
2520 GOTO 1980
2530 GOSUB 2690:lb=w
2540 GOSUB 2690
2550 wert=w*256+lb
2560 FOR i=0 TO maxlab:IF UNT(wert)=VAL(wa$(i)) THEN pr$=LEF
T$(pr$,po-1)+la$(i)+RIGHT$(pr$,LEN(pr$)-po-1):GOTO 1980
2570 NEXT
2580 pr$=LEFT$(pr$,po-1)+"&"+HEX$(wert,4)+RIGHT$(pr$,LEN(pr$
)-po-1)
2590 GOTO 1980
2600 REM Reemplazar e
2610 po=INSTR(pr$,"e")
2620 GOSUB 2690
2630 IF w>127 THEN w=w-256:REM 2er-Komp.
2640 w=w+2
2650 a$="$"+STR$(w)+" >"+"&"+HEX$(pc+w-2,4)
2660 FOR i=0 TO maxlab:IF UNT(pc+w-2)=VAL(wa$(i)) THEN a$=la
$(i) ELSE NEXT
2670 pr$=LEFT$(pr$,po-1)+a$+RIGHT$(pr$,LEN(pr$)-po)
2680 GOTO 1980
2690 REM Leer byte
2700 !RPEEK,@w%,pc,status:w=w%
```

```
2710 pc=pc+1
2720 IF ABS(zflag)=1 THEN POKE adrbef+qq,w:qq=qq+1
2730 PRINT#aus,HEX$(w,2);" ";
2740 RETURN
2750 ' ***********************************
2760 REM init
2770 romstat$(1)="RAM":romstat$(0)="ROM"
2780 status=255:w%=0
2790 adrreg=&9FBB:adrbef=&9F92:stapel=&9FCF:' s2=PRINT PRINT
 PRINT
2800 POKE stapel,&F0:POKE stapel+1,&9F
2810 DIM la$(50),wa$(50)
2820 zeino=10000
2830 LOCATE 1,1:PRINT"lo : RAM   hi : RAM";:lostat=1:histat=
1
2840 LOCATE 68,1:PRINT"Impresora desactivada"
2850 LOCATE 61,2:PRINT"Simulacion real desactivada"
2860 WINDOW #0,3,80,3,22
2870 WINDOW #1,30,66,1,1
2880 WINDOW #2,76,78,1,1
2890 WINDOW #3,1,2,3,22
2900 WINDOW #4,6,9,1,1:WINDOW #5,17,20,1,1
2910 WINDOW #6,76,78,2,2
2920 pri$=CHR$(10)+CHR$(11)+CHR$(11)
2930 aus=0
2940 READ bean:FOR i=0 TO bean:READ menu$(i):NEXT
2950 GOSUB 380
2960 DIM cond$(7),regtab$(7),rotschi$(8),bitti$(3),arilog$(7
),bef$(255)
2970 FOR i=0 TO 7:READ cond$(i):NEXT
2980 FOR i=0 TO 7:READ regtab$(i):NEXT
2990 FOR i=0 TO 7:READ rotschi$(i):NEXT
3000 FOR i=1 TO 3:READ bitti$(i):NEXT
```

```
3010 FOR i=0 TO 7:READ arilog$(i):NEXT
3020 FOR i=0 TO &7F:READ bef$(i):NEXT
3030 FOR i=&80 TO &9F:bef$(i)="":NEXT
3040 FOR i=&A0 TO &FF:READ bef$(i):NEXT
3050 mst=&9F00
3060 FOR i=mst TO mst+&19:READ a$:w=VAL("&"+a$)
3070 s=s+w:POKE i,w:NEXT
3080 IF s<> 4273 THEN PRINT"Error en DATAS 1" :END ELSE s=0
3081 ' 464: if s<> 4121 then ....
3082 ' 664: if s<> 4288 then ....
3090 FOR i=&9F20 TO &9F6C
3100 READ a$:w=VAL("&H"+a$)
3110 s=s+w:POKE i,w:NEXT
3120 IF s<> 9278 THEN PRINT"Error en DATAS 2" :END ELSE s=0
3121 ' 464: if s<> 9405 then ....
3122 ' 664: if s<> 9341 then ....
3130 CALL &9F20: REM Incorporar RSX
3140 FOR i=&9F70 TO &9FB8
3150 READ a$:w=VAL("&H"+a$)
3160 s=s+w:POKE i,w:NEXT
3170 IF s<> 12811 THEN PRINT"Error en DATAS 3" :END
3180 RETURN
3190 REM DATAS ++++++++++++++++++++
3200 DATA 9
3210 DATA Help, Disassembler,Generador fuente,Simulador Z-80
3220  DATA " (Barra espacios)=Comando simular", Simulación
real, Alteracion contenido registros
3230 DATA Impresora activada/desactivada,ROM/RAM Status, Hasta
pronto!!
3240 DATA NC,"C,",PO,PE,NZ,"Z,","P,","M,"
3250 DATA B,C,D,E,H,L,(HL),A
3260 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
3270 DATA BIT,RES,SET
3280 DATA ADD,ADC,SUB,SBC,AND,XOR,OR,CP
3290 DATA NOP,"LD BC,nn","LD (BC),A",INC BC,INC B,DEC B,"LD
B,n",RLCA
```

```
3300 DATA "EX AF,AF'","ADD HL,BC","LD A,(BC)",DEC BC,INC C,D
EC C,"LD C,n",RRCA
3310 DATA DJNZ e,"LD DE,nn","LD (DE),A",INC DE,INC D,DEC D,"
LD D,n",RLA
3320 DATA JR e,"ADD HL,DE","LD A,(DE)",DEC DE,INC E,DEC E,"L
D E,n",RRA
3330 DATA "JR NZ,e","LD HL,nn","LD (nn),HL",INC HL,INC H,DEC
 H,"LD H,n",DAA
3340 DATA "JR Z,e","ADD HL,HL","LD HL,(nn)",DEC HL,INC H,DEC
 H,"LD L,n",CPL
3350 DATA "JR NC,e","LD SP,nn","LD (nn),A",INC SP,INC (HL),D
EC (HL),"LD (HL),n",SCF
3360 DATA "JR C,e","ADD HL,SP","LD A,(nn)",DEC SP,INC A,DEC
A,"LD A,n",CCF
3370 DATA "IN B,(C)","OUT (C),B","SBC HL,BC","LD (nn),BC",NE
G,RETN,IM 0,"LD I,A"
3380 DATA "IN C,(C)","OUT (C),C","ADC HL,BC","LD BC,(nn)",,R
ETI,,"LD R,A"
3390 DATA "IN D,(C)","OUT (C),D","SBC HL,DE","LD (nn),DE",,,
IM 1,"LD A,I"
3400 DATA "IN E,(C)","OUT (C),E","ADC HL,DE","LD DE,(nn)",,,
IM 2,"LD A,R"
3410 DATA "IN H,(C)","OUT (C),H","SBC HL,HL","LD (nn),HL",,,
,RRD
3420 DATA "IN L,(C)","OUT (C),L","ADC HL,HL","LD HL,(nn)",,,
,RLD
3430 DATA ,,"SBC HL,SP","LD (nn),SP",,,,
3440 DATA "IN A,(C)","OUT (C),A","ADC HL,SP","LD SP,(nn)",,,
,
3450 DATA LDI,CPI,INI,OUTI,,,,,LDD,CPD,IND,OUTD,,,,
3460 DATA LDIR,CPIR,INIR,OTIR,,,,,LDDR,CPDR,INDR,OTDR,,,,
3470 DATA RET NZ,POP BC,"JP NZ,nn",JP nn,"CALL NZ,nn",PUSH B
C,"ADD A,n",RST &00
```

```
3480 DATA RET Z,RET,"JP Z,nn",->,"CALL Z,nn",CALL nn,"ADC A,
n",RST &08
3490 DATA RET NC,POP DE,"JP NC,nn","OUT (n),A","CALL NC,nn",
PUSH DE,"SUB n",RST &10
3500 DATA RET C,EXX,"JP C,nn","IN A,(n)","CALL C,nn",->,"SBC
 A,n",RST &18
3510 DATA RET PO,POP HL,"JP PO,nn","EX (SP),HL","CALL PO,nn"
,PUSH HL,"AND n",RST &20
3520 DATA RET PE,JP (HL),"JP PE,nn","EX DE,HL","CALL PE,nn",
->,"XOR n",RST &28
3530 DATA RET P,POP AF,"JP P,nn",DI,"CALL P,nn",PUSH AF,"OR
n",RST &30
3540 DATA RET M,"LD SP,HL","JP M,nn",EI,"CALL M,nn",->,"CP n
",RST &38
3550 DATA DF,04,9f,C9,07,9f,FD,EB
3560 DATA 23,5E,23,56,EB,CD,4d,de
3561 '464:..,..,..,..,..,..,61,dd
3562 '664:..,..,..,..,..,..,52,de
3570 DATA B7,C8,CD,cf,EE,D0,CD,a5
3571 '464:..,..,..,04,ee,..,..,c6
3572 '664:..,..,..,d4,ee,..,..,aa
3580 DATA E7,C9
3581 '464:e6,..
3582 '664:e7,.
3590 DATA 01,2F,9F,21,3A,9F,CD,D1
3600 DATA BC,3E,C9,32,20,9F,C9,34
3610 DATA 9F,C3,3E,9F,52,50,45,45
3620 DATA CB,00,00,00,00,00,DF,42
3630 DATA 9F,C9,45,9F,FD,FE,03,C2
3640 DATA 55,D0,7A,FE,00,C2,1D,C2
3641 '464:ed,cf,..,..,..,..,1d,c2
3642 '664:58,d0,..,..,..,..,50,cb
3650 DATA 7B,32,6A,9F,DD,6E,02,DD
3660 DATA 66,03,DF,68,9F,DD,6E,04
3670 DATA DD,66,05,77,97,23,77,C9
3680 DATA 6B,9F,FD,7E,C9
```

```
3690 DATA F3,D9,08,F5,C5,D5,E5,08
3700 DATA D9,ED,73,B9,9F,31,BB,9F
3710 DATA F1,C1,D1,E1,D9,08,DD,E1
3720 DATA FD,E1,F1,C1,D1,E1,ED,7B
3730 DATA CF,9F,00,00,00,00,ED,73
3740 DATA CF,9F,31,CF,9F,E5,D5,C5
3750 DATA F5,FD,E5,DD,E5,08,D9,E5
3760 DATA D5,C5,F5,ED,7B,B9,9F,D9
3770 DATA 08,E1,D1,C1,F1,08,D9,FB
3780 DATA C9
ARILOG $ 2140 2960 3010
ADR ! 1890 1900 2430
ADRREG ! 740 740 1020 1020 1030 1030 1040 1040 1110 1170 117
0 1230 1230 1410 1420 1470 1480 2790
ADRBEF ! 540 540 2720 2790
A ! 520 1620 1630 1840
AUS ! 220 220 220 1150 1170 1200 1230 1900 2000 2010 2020 27
30 2930
A $ 90 90 100 110 280 280 290 320 320 330 340 1010 1020 1030
 1040 1070 1080 1810 1820 1840 2130 2130 2140 2650 2660 2670
 3060 3060 3100 3100 3150 3150
BITTI $ 2190 2960 3000
BIT ! 1100 1110 1110
BEDFLG ! 880 990 1000 1090 1110 1110
B ! 740 740 740 740 740 750
BEF $ 670 680 700 700 710 730 770 860 2100 2960 3020 3030 30
40
BEAN ! 110 410 2940 2940
CO3 ! 2320 2330
CO2 ! 2120 2130 2130 2130 2140 2190 2190 2310
CO1 ! 2090 2190 2190 2300
COND $ 1080 2960 2970
DIS $ 2440 2440 2450
DIS ! 2170 2430 2440 2440 2440 2440
```

```
DWFLAG ! 720 1950 1950
ENDE ! 1630 1630 1760
ECHTSIM ! 590 590 600 660 1130
EXROM ! 340 350
HISTAT ! 300 300 310 330 340 340 340 350 350 2830
I $ 2260 2260 2400 2400 2410 2450 2450
IFLAG ! 1910 1940 2050 2170 2250
I ! 410 420 420 420 540 540 1080 1080 1090 1100 1100 1100 11
00 1110 1550 1550 1550 1550 1560 1560 1580 1710 1710 1710 25
60 2560 2560 2660 2660 2660 2940 2940 2970 2970 2980 2980 29
90 2990 3000 3000 3010 3010 3020 3020 3030 3030 3040 3040 30
60 3070 3090 3110 3140 3160
K ! 1180 1180
LB ! 2530 2550
LA $ 1550 1550 1560 1710 2560 2660 2810
LOSTAT ! 270 290 290 290 350 2830
MAXLAB ! 1580 1610 1710 2560 2660
MST ! 1570 1670 1750 3050 3060 3060
MENU $ 110 120 130 420 420 420 2940
ME ! 110 110 120 130 140 1610
NOFLAG ! 520 1620 1630 1820 1830
PORE ! 1180 1190 1190 1190 1230 1230
PR $ 670 680 720 1010 1050 1070 1740 1950 1950 1960 1970 198
0 1990 1990 2000 2010 2010 2010 2100 2110 2120 2140 2190 219
0 2230 2360 2370 2380 2390 2390 2400 2400 2410 2410 2410 241
0 2420 2470 2490 2510 2510 2510 2510 2560 2560 2560 2560 258
0 2580 2580 2580 2610 2670 2670 2670 2670
PO ! 670 680 1010 1070 1320 1330 1350 1430 1460 1470 1480 19
80 2000 2010 2010 2360 2370 2410 2410 2470 2480 2490 2510 25
10 2560 2560 2580 2580 2610 2670 2670
PC ! 520 520 520 720 720 780 910 1020 1030 1040 1050 1370 16
20 1620 1620 1630 1710 1760 1890 2430 2650 2660 2700 2710 27
10
PRI $ 70 2920
```

```
Q ! 1160 1170 1170 1180 1190 1310 1320 1340 1350 1410 1410 1
420 1420 1460 1470 1480
QQ ! 540 2720 2720 2720
ROTSCHI $ 2190 2960 2990
REG $ 2120 2140 2190 2190 2330
REGTAB $ 2120 2330 2960 2980
ROMSTAT $ 270 290 310 330 2770 2770
S ! 3070 3070 3080 3080 3110 3110 3120 3120 3160 3160 3170
SOUFLAG ! 1520 1610 1650 1730 1770
STAPEL ! 800 800 830 840 900 900 930 940 2790 2800 2800
SP ! 800 810 810 810 820 820 820 830 830 840 840 900 910 910
 920 920 930 930 940 940
SPRADR ! 720 780 810 810 820 820
STATUS ! 350 350 2700 2780
W % 2700 2700 2780
W ! 1950 1950 1950 1950 2050 2050 2060 2070 2100 2110 2170 2
230 2230 2260 2300 2310 2320 2430 2510 2530 2550 2630 2630 2
630 2640 2640 2650 2650 2660 2700 2720 2730 3060 3070 3070 3
100 3110 3110 3150 3160 3160
WA $ 1550 1560 1710 2560 2660 2810
WERT ! 1460 1470 1470 1480 1480 2550 2560 2580
X ! 1800 1810 1850
Y ! 1800 1810 1850
ZEINO ! 1560 1570 1570 1660 1680 1680 1700 2820
Z $ 1300 1360 1370 1410 1420 1460 1560 1570 1660 1670 1690 1
700 1710 1710 1740 1740 1750
ZFLAG ! 150 150 150 480 480 530 640 1270 2720
```

DESCRIPCION DEL PROGRAMA

Línea 10 - 60:

Inicialización

Línea 70 - 160:

Bucle principal

Línea 70:

Si se detecta el final de pantalla se realiza un scrolling (pri$) visualizándose "*".

Línea 100:

Al pulsar *RETURN* repetir última entrada.

Línea 110:

Comprobar la entrada de los comandos por medio de las letras iniciales.

Línea 120 - 140:

Visualizar letra y comando y bifurcar a la rutina correspondiente.

Línea 150:

Flag Z determina si puede simularse mediante la barra de espacio.

Línea 180 - 190:

Punto final del menú, por ejemplo "Hasta pronto!".

Línea 210 - 230:

Punto del menú "Impresora Activada/Desactivada".

Línea 230:

Subrutina Flag Z activo/desactivo.

Línea 250 - 360:

Punto del menú "Selección estado ROM/RAM".

Línea 260 - 290:

Entrada para LOW-ROM

Línea 300 - 360:

Entrada para HIGH-ROM

Línea 340:

Tratamiento de entrada del número del ROM de expansión.

Línea 380 - 440:

Las funciones P, E y R no deben influir en el flag Z ni a la posición del cursor. Esto se realiza mediante la presente subrutina.

Línea 510 - 560:

Punto del menú simulador Z80.

Línea 520:

Extraer dirección inicial.

Línea 530:

Visualizar registros.

Línea 540:

Borrar área de transferencia de comandos.

Línea 550:

Desensamblar comandos.

Línea 580 - 610:

Punto del menú simulación real activada/desactivada.

Línea 630 - 1130:

Punto del menú barra de espacio=simular.

Línea 640:

Seguir únicamente si Flag Z activado.

Línea 650:

Mantener estado Flag Z.

Línea 660:

Si simulación real activada ejecutar comando de lo contrario visualizar registros sin modificación.

Línea 670 - 680:

Código de comando hacia bef$.

Línea 690 - 1120:

Simular comandos de bifurcación desde el BASIC.

Línea 700:

Comando JP/JR: cargar PC con dirección de destino indicada.

Línea 710 - 720:

Comandos RST: PC a dirección de destino; dirección RET a pila.

Línea 730 - 760:

Incrementar con B el comando DJNZ.

Línea 770 - 780:

Comando CALL y rutina que almacena dirección de retorno en la pila.

Línea 790 - 850:

En este punto se corrigen la pila, SP y PC para bifurcaciones a subrutinas (CALL y RST).

Línea 860 - 880:

Comando RET bedflg=0 significa RET incondicional.

Línea 890 - 950:

Corrección de la pila, SP y PC en caso de retornos.

Línea 970 - 1050:

Aquí se inicializa el PC dependiendo de FL. Las bifurcaciones indirectas se tratan en las líneas 830-860.

Línea 1050:

Inicializa PC con dirección indicada.

Línea 1060 - 1120:

Sub (subrutina) inicializa FL con:
1-,si es incondicional
0-,si no se cumple la condición
-1-,si se cumple la condición

Línea 1100:

Calcula número de bit del flag actual en el registro de flags.

Línea 1130:

Simulación mediante programa máquina, si simulación real activada y no se simulan comandos de bifurcación.

Línea 1140 - 1210:

Salida formateada de los contenidos de registros.

Línea 1220 - 1240:

Visualiza un registro de 16 bits. La posición de los registros se da según la secuencia de los comandos PUSH/POP del programa máquina.

Línea 1260 - 1380:

Rutina de entrada, si se modifican registros.

Línea 1400 - 1440:

Aquí se graban los registros estándard después de su modificación, en las posiciones correspondientes de la memoria.

Línea 1450 - 1490:

Después de una modificación puede inicializarse nuevamente cualquier registro mediante esta rutina.

Línea 1500 - 1580:

Punto del menú "Generador fuente".

Línea 1540 - 1580:

Define labels introducidos y genera las correspondientes "Líneas EQU" en el listado fuente.

Línea 1600 - 1770:

Punto del menú "Desensamblador" (también generador fuente).

Línea 1610:

Si se ha seleccionado el desensamblador se desactivan "Souflag" y "Maxlab".

Línea 1620:

Entrada dirección inicial (con opción *RETURN*).

Línea 1630:

Entrada dirección final (con opción *RETURN*).

Línea 1650:

Saltea generador fuente en función desensamblador.

Línea 1660:

Genera línea en blanco

Línea 1670:

Llama programa máquina que convierte Z$ en línea.

Línea 1680:

Incrementar "Zeiso".

Línea 1700:

Genera comienzo de la siguiente línea.

Línea 1710:

Comprueba que label introducido es correcto.

Línea 1720:

Llama desensamblador.

Línea 1740/1750:

Genera línea recién desensamblada.

Línea 1790 - 1860:

Subrutina para entrada direcciones HEX reconociendo si se introduce únicamente *RETURN*.

Línea 1880 - 1900:

Aquí se visualiza dirección actual.

Línea 1920:

Leer y visualizar siguiente byte.

Linea 1930:

Bifurcación a la subrutina que ejecuta la interpretación.

Linea 1940:

Bifurcar al tratamiento de comandos indexados si iflag activado (=-1).

Linea 1950:

Tratamiento de los comandos RST, que utilizan la palabra DATA siguiente.

Linea 1960:

Bifurcación si el comando contiene números.

Linea 1970:

Bifurcación si el comando contiene distancias relativas.

Linea 1980 - 2020:

Salida formateada.

Linea 2040 - 2280:

Subrutinas para la interpretación.

Linea 2050:

Bifurcación al tratamiento de comandos indexados.

Linea 2060:

Bifurcación al tratamiento de los comandos que comienzan con &ED.

Línea 2070:

Bifurcación al tratamiento de los comandos que comienzan con &CD.

Línea 2080:

Bifurcación a la subrutina que descompone W en co1(bit 6,7), co2(bit 5-3) y co3(bit 2-0).

Línea 2090:

Si co1=0 o co1=3 a línea 360, es decir leer comandos de la tabla, de lo contrario a línea 370, comandos de forma LD REG,REG. Línea 390 comandos aritmético-lógicos de 8 bits.

Línea 2100:

Determinar PR$ según tabla.

Línea 2120 - 2130:

Comandos de la forma LD r,r y HALT.

Línea 2130 - 2140:

Comandos aritmético-lógicos.

Línea 2150 - 2200:

Tratamiento de los comandos que comienzan con el código &CB.

Línea 2160:

Leer siguiente byte.

Línea 2180:

Descomponer en co1, co2, co3.

Línea 2190:

Generar pr$ para comandos rotación/desplazamiento (cc1=0) y comandos de manipulación de bits.

Línea 2210 - 2230:

Tratamiento de los comandos que comienzan con el código &ED.

Línea 2220:

Leer siguiente byte.

Línea 2230:

Si código es válido determinar el mismo según tabla (línea 2100).

Línea 2240 - 2280:

Primer tratamiento de los comandos indexados.

Línea 2250:

Activar flag.

Línea 2260:

Almacenar en i$ el registro (IX ó IY).

Línea 2270:

Leer siguiente byte.

Línea 2280:

Comenzar interpretación nuevamente (línea 300).

Línea 2290 - 2330:

SUB Descomponer código, w se descompone en co1 (Bit 7,6), co2 (Bit 5-3) y co3 (Bit 2-0). Reg$ contiene el registro correspondiente a co3.

Línea 2350 - 2450:

Comandos indexados "segundo tratamiento". Comprobar si está permitido el comando indexado; en caso afirmativo reemplazar HL con i$. Si existe un desplazamiento las líneas 2430, 2440 indican esta distancia.

Línea 2460 - 2590:

Si pr$ contiene una "n" la misma se reemplaza por un número.

Línea 2490 - 2520:

Números de un byte (n).

Línea 2530 - 2590:

Reemplazar números de 2 bytes (nn). Si se ejecuta el generador fuente se comprueba si dirección = Label.

Línea 2600 - 2680:

Reemplazar desplazamiento (e).

Línea 2630 - 2640:

Calcular desplazamiento.

Línea 2650 - 2670:

Aplicar desplazamiento y comprobar si dirección destino = Label.

Línea 2690 - 2740:

SUB leer y visualizar siguiente byte, si el simulador está desactivado "pokear" código de comando al programa máquina.

Línea 2750 - 3180:

Inicialización: confección de las tablas.

Línea 3190 - 3780:

Líneas DATA.

LISTA DE VARIABLES

A - Retorno de SUB "Hex.-Dec." valor de A$ interpretado como número hex.
A$ - Transferencia de subrutinas de diversas tareas.
ADR - Dirección primer código del comando actual.
ADRREG - Dirección registro de transferencia.
ADRBEF - Dirección transferencia del comando.
AUS - Canal dispositivo salida
B - Valor registro B.
BEAN - Cantidad de comandos.
BEDFLG - Flag de condiciones. -1,0,1
BEF$ - Código de comando.
BIT - Número de bit.
CO1 - Bit 7 y 6
CO2 - Bit 5 a 3
CO3 - Bit 2 a 0
DIS - Distancia en comandos indexados
DIS$ - Distancia string para salida
ENDE - Ultima dirección para desensamblaje.
ECHTSIM - Flag simulación real.
EXROM - Número ROM de expansión.
HISTAT - Estado Upper ROM 0, 1 ROM/RAM.
IFLAG - Activado, si direccionamiento indexado, de lo contrario desactivado.
I - Contador.
K - Contador.
LB - Almacenamiento temporal del Low-byte de números de 2 bytes
LOSTAT - Estado Lower ROM.
MST - Dirección inicial programa máquina, generador líneas.
MAXLAB - Cantidad de labels introducidos.
ME - Número del punto del menú.
NOFLAG - "Si *RETURN* al introducir dirección" flag.
PC - Apuntador de programa indica la dirección de la posición actual.
PO - Posición de n, nn, e, HL... en PR$

PORE - Posición registro
PRI$ - String para scrolling.
PR$ - PR$ contiene comando interpretado de ensamblador
Q - Contador.
QQ - Contador transferencia de comandos al simulador.
REG$ - Registro transferencia de comandos prog. máquina.
ROMSTAT - Contiene ROM/RAM activo/desactivo.
S - Suma de comprobación.
SOUFLAG - Flag generador fuente.
STAPEL - Dirección de SP
SP - SP
SPRADR - Dirección de bifurcación.
STATUS - Estado ROM/RAM para Far Call.
W% - Valor leído de RDEEK.
WERT - Valor de un número de 2 bytes (nn)
X - Entrada posición X.
Y - Entrada posición Y.
ZEINO - Número de línea para fuente.
Z$ - Línea para fuente y entrada línea modificación de registro.
ZFLAG - Flag de simulación.

TABLAS

regtab$ (7) - Registros
rotschie$ (7) - Comandos rotación y desplazamiento
bitti$ (3) - Comandos manipulación bits
arilog$ (7) - Comandos aritméticos/lógicos
bef$ (255) - 0 hasta &3F: comandos que comienzan con &ED y que tienen como byte 1 un número
&40 - &BF: comandos que comienzan con &ED y que tienen como byte 2 un número
&BF - &FF: comandos que tienen como byte 1 un número
menu$ - Códigos de comando extendidos (Bean).
LA$ - Label$
WA$ - Dirección correspondiente a los labels.
COND$ - Condiciones (7).

RUTINAS DEL SISTEMA

INTRODUCCION

La utilización efectiva de rutinas del sistema es el ABC de la programación en lenguaje máquina. En el caso de los ordenadores CPC nos hemos de enfrentar en primer lugar a algunos problemas.

Al conectar el ordenador, las 64K de memoria se encuentran íntegramente ocupadas por el RAM, que contiene únicamente una pequeña porción de las rutinas del sistema. El sistema operativo, como ya hemos explicado, se encuentra "por debajo" del RAM en las direcciones &0 hasta &3FFF. Las direcciones intermedias, &4000 a &BFFF generalmente se encuentran inicializadas con RAM. En el área superior de direcciones &C000 hasta &FFFF se solapan el RAM de vídeo, el ROM de BASIC, el ROM del sistema operativo de diskette y eventualmente 252 ROM de expansión. En el caso del CPC 6128 ocurre además que el área de direcciones se encuentra íntegramente solapada con las segundas 64Kbytes de RAM.

Para poder manejar todos estos ROMs y RAMs, hace falta un procedimiento que se conoce como "Bankswitching". Esto significa que la memoria deseada puede conectarse según las necesidades para el área actual de direcciones. Para lograr esto se utilizan los comandos RESTAT. Estos comandos acceden a las direcciones &0 hasta &38. En estas posiciones, los contenidos del ROM y del RAM son iguales. Ello puede comprobarse fácilmente con el desensamblador. De esta forma queda asegurado que los comandos RESTART pueden utilizarse siempre de la misma forma independientemente de la actual configuración de memoria.

Casi todos los comandos RESTART bifurcan al área de direcciones desde &AB80 hasta &BFFF. Allí generalmente está seleccionado el RAM, es decir, que las rutinas seleccionadas siempre están presentes. Estas rutinas se ocupan entonces según la necesidad de la conmutación entre los diferentes ROMs y RAMs.

En el área RAM mencionada se encuentran también los vectores de bifurcación hacia todas las rutinas importantes del sistema operativo y hacia las rutinas aritméticas. A través de estos vectores de bifurcación se realiza todo tipo de comunicación entre el BASIC y el sistema operativo.

Un vector de bifurcación es de gran utilidad. En lugar de bifurcar directamente a una rutina determinada, se bifurca a una dirección previamente determinada (!) en la cual se encuentra el comando de bifurcación para esta rutina. Este procedimiento ofrece varias ventajas.

1. En el caso de la familia CPC, donde ya existen tres versiones de ROM diferentes, el programador en lenguaje máquina puede confeccionar programas utilizando los vectores descritos, de tal forma que los programas realizados con esta técnica pueden aplicarse en cualquiera de los tres ordenadores.

2. También es posible modificar (parchear) los vectores para que apunten a rutinas propias. De esta forma nos aseguramos que el sistema utilice estas rutinas propias de forma automática. Este procedimiento se utiliza asimismo si se conecta un dispositivo de diskette al CPC.

Aparte de los vectores, el tercer RAM (&8000 hasta &BFFF) contiene gran número de posiciones de memoria en las cuales el sistema almacena importantes informaciones, los llamados PEEKS y POKES.

Resumimos:

En las direcciones &0 hasta &40 se encuentran las rutinas de los comandos RST en el ROM y en el RAM, las cuales posibilitan la comunicación entre los diferentes componentes de memoria. Dado que para cada RESTART en un vector de bifurcación sólo se dispone de 8 bytes para longitud de programa, estas rutinas RESTART bifurcan a las rutinas reales de ejecución en el tercer RAM activado permanentemente.

Por ello algunos comandos RESTART de los ordenadores CPC pueden denominarse como comandos ampliados de JUMP o CALL, con los cuales es posible indicar adicionalmente el estado ROM/RAM para la dirección de destino de bifurcación. La función exacta de los comandos RST se detalla más adelante.

En el tercer RAM se encuentran vectores de bifurcación, que en general son comandos RST así como importantes PEEKs y POKEs.

Una de las rutinas del sistema de mayor importancia es la que se utiliza para la visualización de un carácter en la pantalla, que puede llamarse mediante CALL &BB5A. Esta rutina visualiza un carácter que corresponde al valor contenido en el ACU.
Confeccione un programa para la salida del set de caracteres (códigos 32 a 255) del CPC.

Solución:

```
10 '        ORG  &A000
20 'PRINT   EQU  &BB5A
30 '        LD   B,223          ; Contador = 255 - 32
40 '        LD   A,32
50 'SCHLEI  CALL PRINT          ; Visualizar CHR$(A)
60 '        INC  A
70 '        DJNZ SCHLEI
80 '        RET
90 '        END
```

En relación con la salida de caracteres el ensamblador posee el pseudo-comando DM. Al comando DM sigue como operando una palabra entre comillas. Los códigos ASCII de las letras de la palabra se almacenan mediante DM a partir de la dirección actual. Vean el siguiente programa:

```
10 '          ORG  &A000
20 'PRINT     EQU  &BB5A
30 '          LD   HL,WORT       ; Direcc. palabra a visualizar
40 'SCHLEI    LD   A,(HL)        ; Cargar ACU cod. ASCII letra
50 '          INC  HL            ; Pos. apunt. siguiente letra
60 '          CALL PRINT
70 '          OR   A             ; Activar flags
80 '          JR   NZ,SCHLEI     ; No cero, letra siguiente
90 '          RET
100 'WORT     DM   "CPC"
110 '         DB   0
120 '         END
```

El Null-byte generado por DB 0 al final de la palabra (línea 110) sirve para reconocer el final de la palabra a visualizar.

Ejecutando disassembler a partir de la dirección de entrada de la rutina (&BB5A) obtenemos lo siguiente:

```
BB5A CF 00 94       RST   &08/DW: &9400
```

En la dirección &BB5A se encuentra un comando Restart hacia dirección &0008. Si aquí continuamos la traducción, obtenemos:

```
0008 C3 82 B9       JP    &B982
```

Esta rutina (a partir de &B982) se denomina como "Rutina RST &08" y provoca un tratamiento especial de los 2 bytes (Low-High) siguientes al comando RST &08. Por esta razón, el disassembler visualiza los bytes siguientes a RST &08 con la marca DW (Data Word, es decir low y high-byte). DW representa también un pseudo-comando, que provoca que el Data-word siguiente al comando, un número de 2 bytes, se almacene en la posición correspondiente de la memoria. Los bits 0-13 se interpretan como dirección de destino de bifurcación (con 14 bits pueden representarse direcciones en el área &0 hasta &3FFF). Bit 14 y 15 se utilizan para la selección de ROM/RAM.

Bit 14 determina el estado del área de &0-&3FFF. Bit 15 determina el estado del área de &C000 hasta &FFFF (RAM pantalla o ROM BASIC). Un bit activado selecciona el RAM y el bit desactivado el ROM.
¿Qué estado y qué destino de bifurcación tiene el siguiente comando?

```
RST &08
DW  &9400
```

Descomponemos:
&9400=&8000+&1400=&X01 01 0100 0000 0000

Bit 15 = 1 ==> RAM de pantalla
Bit 14 = 0 ==> ROM sistema operativo
Dirección: &1400

RST &08/DW &9400 provoca una bifurcación a la rutina del sistema operativo en dirección &1400. Se ha seleccionado la pantalla (RAM).
Aunque los comandos RST constituyen principalmente bifurcaciones a subrutinas, es decir la dirección de retorno se almacena en el stack, el comando RST &08 no es una subrutina sino una bifurcación normal. Esto se realiza mediante la manipulación de la pila en la rutina a partir de &B982.
También los restantes comandos RST tienen tareas especiales, que les explicaremos a lo largo del presente capítulo.

Con la ayuda de la rutina PRINT confeccionaremos ahora un programa MONITOR.

EL MONITOR

Dado que visualizamos los contenidos de memoria en números hexadecimales, necesitamos en primer lugar una subrutina que visualice un byte como número hexadecimal. El byte a visualizar se transfiere a través del ACU.

Ejemplo: A = 63 = &3F = &X 0011 1111

F corresponde a los 4 bits inferiores (High-nibble).
3 corresponde a los 4 bits superiores (Low-nibble).

En primer lugar se visualiza el High-nibble. Para ello desplazamos 4 veces el contenido del ACU hacia la derecha (rotación de 8 bits). El resultado de este desplazamiento es &X 1111 0011. A continuación se borran los 4 bits superiores mediante AND. Seguidamente el ACU contiene el valor &X 0000 0011 = 3. Este valor (3) debe visualizarse.
El código ASCII de 3 es 51. Para obtener el valor 51 en el ACU hemos de sumar 48 al contenido del mismo (=3). A continuación llamamos a la rutina PRINT. Para visualizar el Low-nibble borramos los 4 bits superiores del contenido antiguo del ACU. Después de la adición de 48 obtenemos 63. Como salida queremos obtener F (&F=15). El valor ASCII de F es 70, lo que significa que si el número hexadecimal a visualizar es mayor que 9 (por lo tanto representable con una letra) hemos de sumar adicionalmente 7 antes de llamar la rutina.
Intenten confeccionar la rutina para la salida de un byte en formato hexadecimal.

```
A000             10        ORG  &A000
A000             20 PRINT  EQU  &BB5A
A000             30 ; SALHEX
A000             40 ; Visualiza ACU en hexadecimal
A000             50 ; Registro-E se modifica
A000 5F          60 AUSHEX LD   E,A      ; Almacenar ACU temp.
A001 0F          70        RRCA          ; Rotar ACU
A002 0F          80        RRCA          ; por 4 bits
A003 0F          90        RRCA          ; hacia la
```

```
A004 0F          100        RRCA           ; Derecha
A005 E60F        110        AND  &X1111    ; Borrar bit 4-7
A007 CD0000      120        CALL conv      ; Visualizar high-nibble
A00A 7B          130        LD   A,E       ; Contenido anterior ACU
A00B E60F        140        AND  &X1111    ; Borrar bits 4-7
A00D CD0000      150        CALL conv      ; Visualizar Low-nibble
A010 C9          160        RET            ; Final AUSHEX
A011             170  ; Rutina conv
A011             180  ; visualiza cont. ACU en Hex
****       Línea 120: CONV = &A011
****       Línea 150: CONV = &A011
A011 FE0A        190 CONV   CP   &A        ; Valor numero < 10
A013 38FE        200        JR   C,ZAHL    ; Si, entonces hacia nro.
A015 C607        210        ADD  A,7       ; Sumar 7 para letra
****       Línea 200: zahl = &A017
A017 C630        220 ZAHL   ADD  A,48       ; = codigo ASCII del
                                              Numero Hexadecimal
A019 CD5ABB      230        CALL PRINT
A01C C9          240        RET            ; Final CONV

Programa : SALHEX
Inicio   : &A000      Final : &A01C
Longitud : 001B
0 Errores
Tabla de variables:
PRINT  BB5A     AUXHEX  A000     CONV A011     ZAHL A017
```

Con este programa podemos confeccionar en lenguaje máquina la salida para el programa COMPARE del último capítulo. Unan ambos programas de manera que la rutina anterior se ocupe de la salida de las direcciones.

```
A000             10 ; COMPARE
A000             20         ORG  &A000
A000             30 PRINT   EQU  &BB5A
A000             40 ANF     DS   2
A002             50 ENDE    DS   2
A004             60 ANFVER  DS   2           ; Inicio bloque comp.
A006 ED5B00A0    70         LD   DE,(ANF)
A00A 2A02A0      80         LD   HL,(ENDE)
A00D B7          90         OR   A
A00E ED52        100        SBC  HL,DE       ; Longitud bloque
A010 23          110        INC  HL          ; HL + 1
A011 44          120        LD   B,H
A012 4D          130        LD   C,L         ; Cargar hacia BC
A013 EB          140        EX   DE,HL       ; ANF hacia HL
A014 ED5B04A0    150        LD   DE,(ANFVER) ; Apuntador bloque
A018 1A          160 WEITER LD   A,(DE)      ; Ele. comparacion
A019 13          170        INC  DE
A01A EDA1        180        CPI              ; Comp. (HL) con A
A01C C40000      190        CALL NZ,AUSGA    ; Salida por desigual
A01F EA18A0      200        JP   PE,WEITER   ; Siguiente ele.
A022 C9          210        RET              ; Final COMPARE
A023             220                         ;
**** LINEA 190: AUSGA = &A023
A023 D5          230 AUSGA  PUSH DE          ; Salvar apunt. blq.
A024 F5          240        PUSH AF          ; Salvar flags
A025 2B          250        DEC  HL          ; Decr. HL salida
A026 7C          260        LD   A,H
A027 CD0000      270        CALL AUSHEX      ; Visualizar Lowbyte
A02A 7D          280        LD   A,L
A02B CD0000      290        CALL AUSHEX      ; Visualizar Highbyte
A02E 23          300        INC  HL          ; Reconstruir HL
A02F 3E20        310        LD   A,&20       ; Espacio blanco
A031 CD5ABB      320        CALL PRINT
A034 F1          330        POP  AF
A035 D1          340        POP  DE
A036 C9          350        RET              ; Final AUSGA
A037             360                         ;
```

```
A037              370                                ; SALHEX
A037              380                                ; Visualiza ACU HEX
A037              390                                ; Modifica reg E
****   Linea 270: AUSHEX = &A037
****   Linea 290: AUSHEX = &A037
A037 5F           400 AUSHEX  LD   E,A               ; Almacenar ACU temp.
A038 0F           410         RRCA                   ; Rotar ACU
A039 0F           420         RRCA                   ; por 4 bits
A03A 0F           430         RRCA                   ; hacia la
A03B 0F           440         RRCA                   ; derecha
A03C E60F         450         AND  &X1111            ; Borrar bit 4-7
A03E CD0000       460         CALL conv              ; Vis. high-nibble
A041 7B           470         LD   A,E               ; Cont. anterior ACU
A042 E60F         480         AND  &X1111            ; Borrar bits 4-7
A044 CD0000       490         CALL conv              ; Vis. Low-nibble
A047 C9           500         RET                    ; Final AUSHEX
A048              510  ; Rutina conv
A048              520  ; visualiza cont. ACU en Hex
****       Linea 460: CONV = &A048
****       Linea 490: CONV = &A048
A048 FE0A         530 CONV    CP   &A                ; Valor numero < 10
A04A 38FE         540         JR   C,ZAHL            ; Si, hacia nro.
A04C 38FE         550         JR   C,ZAHL
A04E C607         560         ADD  A,7               ; + 7 para letra
****       Linea 540: ZAHL = &A050
****       Linea 550: ZAHL = &A050
A050 C630         570 ZAHL    ADD  A,48              ; = codigo ASCII del
                                                      Numero Hexadecimal
A052 CD5ABB       580         CALL PRINT
A055 C9           590         RET                    ; Final CONV

Programa: COMPHEX
Start: &A000    Ende: &A055
Longitud: 0056
0 Errores
Tabla de variables:
PRINT  BB5A   ANF    A000    ENDE   A002    ANFVER A004
WEITER A018   AUSGA A023    AUSHEX A037    CONV   A048
ZAHL   A050
```

Continuaremos con la rutina MONITOR. Desde el BASIC se transfieren las direcciones inicial y final.

```
10 '          ORG  &A000
20 'PRINT     EQU  &BB5A
30 'START     DS   2
40 'ENDE      DS   2
```

Carguemos en primer lugar HL con la dirección inicial, visualizándola a continuación.

```
50 '          LD   HL,(START) ; Apuntador
60 'WEI16     LD   A,H        ; Visualizar highbyte
70 '          CALL AUSHEX
80 '          LD   A,L        ; Visualizar lowbyte
90 '          CALL AUSHEX
```

A continuación se visualiza un espacio en blanco.

```
100 '         LD   A,&20      ; ASCII del espacio en blanco
110 '         CALL PRINT
```

Seguidamente visualizaremos los valores de las 16 posiciones siguientes de memoria (8 en MODE 1):

```
120 '         LD   B,16       ; Contador
130 'WEI      LD   A,(HL)     ; Cargar byte al ACU
140 '         CALL AUSHEX     ; y visualizarlo
150 '         LD   A,&20      ; espacio en blanco
160 '         CALL PRINT      ; Visualizar
170 '         INC  HL
180 '         DJNZ WEI
```

Lo siguiente es visualizar un espacio en blanco y los últimos 16 (8) bytes como caracteres ASCII. De los códigos mayores que 127 se restan 128 (se desactiva bit 7). Para códigos menores que 32 (caracteres de control) se visualiza un punto (ASCII=46).

```
190 '          LD   A,&20
200 '          CALL PRINT          ; Espacio en blanco
210 '          LD   DE,16
220 '          OR   A              ; Carry = 0
230 '          SBC  HL,DE          ; Decrementar apuntador en 16
240 '          LD   B,16
250 'WEIAS     LD   A,(HL)         ; Cargar ACU con byte
260 '          INC  HL             ; Incrementar apuntador
270 '          RES  7,A            ; Convertir car. grafico a ASCII
280 '          CP   &20            ; Mayor igual 32?
290 '          JR   NC,PR          ; Si, entonces salida
300 '          LD   A,46           ; ASCII para punto
310 'PR        CALL PRINT          ; Visualizar caracter
320 '          DJNZ WEIAS
```

Para llegar al inicio de la línea siguiente transferimos código 13 y código 10:
(CHR$(13) = Carriage Return = Enter)
(CHR$(10) = Line Feed = Avance de línea)

```
330 '          LD   A,13           ; Carriage Return
340 '          CALL PRINT          ; Salida
350 '          LD   A,10           ; Avance linea
360 '          CALL PRINT          ; Salida
```

Ahora se comprueba si hemos llegado al final:

```
370 '          PUSH HL             ; Salvar apuntador
380 '          LD   DE,(ENDE)
390 '          OR   A              ; Carry = 0
400 '          SBC  HL,DE          ; Apuntador final  < = 0
410 '          POP  HL             ; Extraer apuntador
420 '          JR   C,WEI16        ; HL - DE  <  0 entonces seguir
430 '          JR   Z,WEI16        ; HL - DE = 0 entonces seguir
440 '          RET                 ; HL - DE  >  0 entonces final
450 '                              ; Final MONITOR
```

Queda por agregar la rutina AUSHEX y nuestro programa puede funcionar:

```
460 ' ; SALHEX
470 ' ; Visualiza ACU en HEX
480 ' ; Modifica el registro E
490 'AUSHEX  LD   E,A          ; Almacenar ACU temp.
500 '        RRCA              ; Rotar ACU
510 '        RRCA              ; por 4 bits
520 '        RRCA              ; hacia la
530 '        RRCA              ; derecha
540 '        AND  &X1111       ; Borrar bit 4-7
550 '        CALL conv         ; Vis. high-nibble
560 '        LD   A,E          ; Cont. anterior ACU
570 '        AND  &X1111       ; Borrar bits 4-7
580 '        CALL conv         ; Vis. Low-nibble
590 '        RET               ; Final AUSHEX
600 '                          ; Rutina conv
610 '                          ; visualiza cont. ACU en Hex
620 'CONV    CP   &A           ; Valor numero < 10
630 '        JR   C,ZAHL       ; Si, hacia nro.
640 '        ADD  A,7          ; + 7 para letra
650 'ZAHL    ADD  A,48         ; = codigo ASCII del
                                 Numero Hexadecimal
660 '        CALL PRINT
670 '        RET               ; Final CONV
680 '        END
```

Con esta rutina solamente podemos leer el RAM. Para acceder al ROM utilizamos el comando RST &18. Este comando provoca en el CPC un llamado FAR CALL. Los 2 bytes siguientes a RST &18 representan un apuntador a la dirección de un vector de bifurcación. En la dirección de vector indicada se encuentran 3 bytes. Los 2 primeros apuntan a la dirección real de bifurcación y el tercero determina el estado ROM/RAM.

Ejemplo:

```
            .
            .
&A000       RST  &18
&A001       DW   Vekaddr
&A003       RET
            .
Vekaddr  DW Zielad
         DB Status
```

Mediante el comando RST &18 en &A000 se realiza una bifurcación de subrutina hacia Zielad (V) donde el Status (V) determina si se selecciona ROM o RAM.
Para Status (V) se utilizan los valores siguientes:

Status	&0-&3FFF (Sistema operativo)	&C000-&FFFF (BASIC)
&FC=252:	ROM	ROM
&FD=253:	RAM	ROM
&FE=254:	ROM	RAM
&FF=255:	RAM	RAM

Todos los restantes valores del estado seleccionan un ROM de expansión.
El área comprendida entre &4000 y &BFFF se utiliza sin excepción como área de direcciones RAM.

El nombre "FAR CALL" (significa "llamada remota"), expresa que a través del comando RST &18 son posibles bifurcaciones a todos los RAM y ROM del ordenador. El FAR CALL funciona como un CALL, es decir después de un RST &18, se produce la bifurcación a la subrutina ejecutándose la misma hasta encontrar el correspondiente comando RET; la ejecución del programa se continúa entonces en el comando siguiente al RST que ha provocado la bifurcación.

Si queremos leer el ROM mediante la rutina MONITOR, llamamos a la misma a través del comando RST &18. La dirección que indica el vector de bifurcación es entonces la dirección inicial de la rutina MONITOR. Para seleccionar ambos ROMS el estado es 252.

La ampliación del programa es la siguiente:

```
10  '          ORG &A000
20  '          RST &18
30  '          DW  Vektor
40  '          RET          ; Retorno al BASIC
50  'VEKTOR    DW  MONITO   ; Direccion vector bifurcacion
60  'STATUS    DB  252      ; Estado ROM/ROM
70  'PRINT     EQU &BB5A
80  'START     DS  2
90  'ENDE      DS  2
100 'MONITO    LD  HL,(START) ; Apuntador
 ..........
 ..........
```

El listado completo de ensamblador:

```
A000             10           ORG  &A000
A000 DF          20           RST  &18
A001 0000        30           DW   vektor
A003 C9          40           RET                ; Retorno al BASIC
**** Linea 30: VEKTOR = &A004
A004 0000        50  VEKTOR   DW   MONITO        ; Dir. vector bifurc.
A006 FC          60  STATUS   DB   252           ; Estado ROM-ROM
A007             70  PRINT    EQU  &BB5A
A007             80  START    DS   2
A009             90  ENDE     DS   2
**** Linea 50: MONITO = &A00B
A00B 2A07A0      100 MONITO   LD   HL,(START)    ; Apuntador
A00E 7C          110 WEI16    LD   A,H           ; Visualizar highbyte
A00F C00000      120          CALL AUSHEX
A012 7D          130          LD   A,L           ; Visualizar Lowbyte
A013 CD0000      140          CALL AUSHEX
A016 3E20        150          LD   A,&20         ; ASCII esp. blanco
A018 CD5ABB      160          CALL PRINT
A01B 0610        170          LD   B,16          ; Contador
A01D 7E          180 WEI      LD   A,(HL)        ; Cargar byte en ACU
A01E CD0000      190          CALL AUSHEX        ; y visualizar
```

```
A021 3E20       200          LD   A,&20       ; Espacio en blanco
A023 CD5ABB     210          CALL PRINT      ; Visualizar
A026 23         220          INC  HL
A027 10F4       230          DJNZ WEI
A029 3E20       240          LD   A,&20
A02B CD5ABB     250          CALL PRINT      ; Espacio en blanco
A02E 111000     260          LD   DE,16
A031 B7         270          OR   A          ; Carry = 0
A032 ED52       280          SBC  HL,DE      ; Decr. ap. en 16
A034 0610       290          LD   B,16
A036 7E         300 WEIAS    LD   A,(HL)     ; Cargar ACU con byte
A037 23         310          INC  HL         ; Incrementar apunt.
A038 CBBF       320          RES  7,A        ; Convertir caracter
                                               grafico en ASCII
A03A FE20       330          CP   &20        ; Mayor igual 32
A03C 30FE       340          JR   NC,PR      ; Si, salida
A03E 3E2E       350          LD   A,46       ; ASCII para punto
**** Linea 340: PR = &A040
A040 CD5ABB     360 PR       CALL PRINT      ; Visualizar caracter
A043 10F1       370          DJNZ WEIAS
A045 3E0D       380          LD   A,13       ; Carriage Return
A047 CD5ABB     390          CALL PRINT      ; Salida
A04A 3E0A .     400          LD   A,10       ; Avance linea
A04C CD5ABB     410          CALL PRINT      ; Salida
A04F E5         420          PUSH HL         ; Salvar apuntador
A050 ED5B09A0   430          LD   DE,(ENDE)
A054 B7         440          OR   A          ; Carry = 0
A055 ED52       450          SBC  HL,DE      ; Apunt.  > =0
A057 E1         460          POP  HL         ; Extraer apuntador
                                           (sin influencia flags)
A058 38B4       470          JR   C,WEI16    ; HL-DE  <  0, seguir
A05A 28B2       480          JR   Z,WEI16    ; HL-DE=0, seguir
A05C C9         490          RET             ; HL-DE  >  0, final
A05D            500                          ; Final MONITOR
A05D            510                          ; SALHEX
A05D            520                          ; Visualiza ACU HEX
A05D            530                          ; Modifica reg E
****   Linea 120: AUSHEX = &A05D
```

```
****   Linea 140: AUSHEX = &A05D
****   Linea 190: AUSHEX = &A05D
A05D 5F          540 AUSHEX  LD   E,A          ; Almacenar ACU temp.
A05E 0F          550         RRCA              ; Rotar ACU
A05F 0F          560         RRCA              ; por 4 bits
A060 0F          570         RRCA              ; hacia la
A061 0F          580         RRCA              ; derecha
A062 E60F        590         AND  &X1111       ; Borrar bit 4-7
A064 CD0000      600         CALL conv         ; Vis. high-nibble
A067 7B          610         LD   A,E          ; Cont. anterior ACU
A068 E60F        620         AND  &X1111       ; Borrar bits 4-7
A06A CD0000      630         CALL conv         ; Vis. Low-nibble
A06D C9          640         RET               ; Final AUSHEX
A06E             650  ; Rutina conv
A06E             660  ; visualiza cont. ACU en Hex
****       Linea 600: CONV = &A06E
****       Linea 630: CONV = &A06E
A06E FE0A        670 CONV    CP   &A           ; Valor numero < 10
A070 38FE        680         JR   C,ZAHL       ; Si, hacia nro.
A072 C607        690         ADD  A,7          ; + 7 para letra
****       Linea 680: ZAHL = &A074
A074 C630        700 ZAHL    ADD  A,48         ; = codigo ASCII del
                                                  numero hexadecimal
A076 CD5ABB      710         CALL PRINT
A079 C9          720         RET               ; Final CONV

Programa: MONITOR
Start: &A000    Ende: &A079
Longitud: 007A
0 Errores
Tabla de variables:
VEKTOR A004   STATUS A006   PRINT  BB5A   START  A007
ENDE   A009   MONITO A00B   WEI16  A00E   WEI    A01D
WEIAS  A036   PR     A040   AUSHEX A05D   CONV   A06E
ZAHL   A074
```

El programa de manejo BASIC:

```
10 REM Programa manejo MONITOR
20 MEMORY &9FFF
30 LOAD "monitor.obj"
40 r$(0)="ROM":r$(1)="RAM"
50 MODE 2
60 PRINT "M O N I T O R   L E C T U R A   R O M / R A M "
70 INPUT "Direccion inicial : &",a$
80 adr=&A007: GOSUB 220
90 INPUT "Direccion final   : &",a$
100 ADR=&A009: GOSUB 220
110 p=VPOS(<>0)
120 PRINT "Sistema operativo :";r$(betrsta);CHR$(13)
130 a$=INKEY$: IF a$="" THEN 130
140 IF a$ <> CHR$(13) THEN betrsta=betrsta XOR 1: GOTO 120
    ELSE PRINT
150 PRINT "BASIC             :";r$(basista);CHR$(13);
160 a$=INKEY$: IF a$="" THEN 160
170 IF a$ <> CHR$(13) THEN basista=basista XOR 1: GOTO 150
    ELSE PRINT
180 status=&X11111100 OR basista*2 OR betrsta
190 POKE &A006,status
200 CALL &A000
210 GOTO 70
220 a=VAL("&"+a$)
230 IF a < 0 THEN a=a+2^16
240 ah=INT(a/256): POKE adr+1,ah
250 POKE adr,a-ah*256
260 RETURN
```

Al seleccionar el estado se mantiene el estado indicado pulsando la tecla ENTER. Pulsando cualquier otra tecla, el estado se altera. Ahora podemos comenzar "el viaje por el Firmware".

Arranque el programa introduciendo lo siguiente:

```
Dirección inicial: &CC00        *ENTER*
Dirección final  : &CE00        *ENTER*
Sistema operativo   : ROM          *ENTER*
BASIC               : ROM          *ENTER*
```

Introduzcan las mismas direcciones alterando el estado del área BASIC para el RAM y obtendrán en lugar de los mensajes de error del ordenador (que se encuentran en el ROM), solamente el Hex-Dump del área de pantalla (RAM).
A partir de &660 (ROM) se encuentra el mensaje de puesta en marcha del ordenador. En el BASIC ROM se encuentra a partir de &E380 la lista de los comandos BASIC utilizados por el intérprete.
Observen los contenidos de ROM y RAM para tener una idea de la distribución de memoria.
La gran mayoría del contenido del ROM pertenece a programas. Confeccione ahora una rutina con el comando RST &18 que transfiera el contenido de una posición de memoria ROM a un programa BASIC e incorpórela luego en el disassembler en la subrutina a partir de línea 940. Tendrá entonces la posibilidad de traducir los programas del ROM. Para comprender mejor las rutinas internas resulta muy útil un listado ROM comentado.

EL BREAKPOINT

Explicamos a continuación la forma de confeccionar un programa de comprobación de programas en lenguaje máquina. Alguna vez seguramente ha sufrido una "caída" del sistema, sin saber la razón de la misma. Un programa máquina no visualiza ningún mensaje de error sino que simplemente paraliza el ordenador por condiciones de error. No es posible la reconstrucción del procedimiento del

error y sería de gran utilidad poder interrumpir un programa en cualquier posición para visualizar los contenidos de registros. Mediante esta información pueden detectarse los errores. Para ello utilizamos el comando RST &30. Este RESTART no lo utiliza el sistema operativo, quedando a nuestra disposición. Los restantes comandos RESTART no deben en ningún caso utilizarse porque realizan tareas propias del sistema operativo. El comando RST &30 tiene el código &F7. En la posición donde se desea interrumpir el programa se introduce el código &F7 mediante POKE. A través del código de comando &F7 el programa se interrumpe, por ello el nombre 'Breakpoint'. A continuación se arranca el programa a comprobar y si encuentra el comando RST &30 se produce una bifurcación de subrutina a dirección &30. En esta dirección introducimos un comando JP que bifurca a la rutina de salida de registros. El siguiente listado de ensamblador se autodocumenta.

Listado de ensamblador:

```
A000             10        ORG  &A500
A500             20                          ; Activar Breakpoint
A500 3EC3        30        LD   A,&C3        ; Codigo para JP
A502 323000      40        LD   (&0030),A    ; Despues &30 (RST) 1
A505 210000      50        LD   HL,REDUMP    ; Dir. inicial reg-Dump
A508 223100      60        LD   (&0031),HL   ; Despues &31/&32
A50B C9          70        RET               ; Activar final
A50C             80                          ; Reg-Dump
**** Linea 50 : REDUMP=&A50C
A50C F5          90 REDUMP PUSH AF           ; Registro
A50D C5         100        PUSH BC           ; almacenar en Stack
A50E D5         110        PUSH DE
A50F E5         120        PUSH HL
A510 210000     130        LD   HL,0         ; calcular contenido
A513 39         140        ADD  HL,SP        ; original SP
A514 110A00     150        LD   DE,10        ; y almacenarlo
A517 19         160        ADD  HL,DE        ; en el Stack
A518 E5         170        PUSH HL           ;
A519 060C       180        LD   B,12         ; Nros. bytes a vis.
```

```
A51B            190                             ; Salida en el orden
A51B            200                             ; PC AF BC DE HL SP
A51B 2B         210 PRNT    DEC   HL
A51C 7E         220         LD    A,(HL)        ; Extraer byte de Stack
A51D CD0000     230         CALL  AUSHEX        ; y visualizarlo
A520 10F9       240         DJNZ  PRNT
A522 E1         250         POP   HL            ; Extr. SP anterior
A523 E1         260         POP   HL            ; extraer
A524 D1         270         POP   DE            ; restantes
A525 C1         280         POP   BC            ; registros
A526 F1         290         POP   AF            ; del Stack
A527 DDE1       300         POP   IX            ; Extraer direccion ret.
A529 C9         310         RET                 ; Retorno al BASIC

......            RUTINA SALHEX

Programa :BREAKPOI
Start    : &A500        Ende : &A529
Longitud : 002A
0 Error


Tabla de variables :
REDUMP A50C     PRNT    A51B
```

Naturalmente debe agregarse la rutina SALHEX.

Mediante CALL &A500 se almacena la dirección inicial de la rutina Dump-Registros a través del comando JP a partir de dirección &30. De esta manera se dispone del comando RST &30 para comprobar programas. Prueben el siguiente programa después de CALL &500:

```
10 ' ORG &A000
20 ' LD  A,1
30 ' LD  BC,&0203
40 ' LD  DE,&0405
50 ' LD  HL,&0607
60 ' RET
70 ' END
```

Después de la traducción arranque el programa con CALL &A000. Los registros deberían cargarse con los valores 1 a 7. Para comprobarlo utilizamos en lugar del comando RET el comando RST &F7:

```
POKE &A00B,&F7
```

Introduzcan ahora CALL &A000 y obtendrán el resultado siguiente:

```
A00C 0168 0203 0405 0607 BFF8
```

Los primeros 2 bytes representan el contenido de PC, después de la interrupción (la interrupción se produjo en dirección &A00B). A continuación tenemos el ACU (=1). El siguiente es el registro de flags:

```
&68=&X 0 1 1 0 1 0  0 0
       S Z   H  P/V N C
```

A continuación los registros B, C, D, E, H y L.
Las últimas 4 cifras representan el contenido de SP antes de la interrupción.
Tengan en cuenta que el Breakpoint (el código &F7, es decir RST &30) con esta rutina debe encontrarse únicamente en el nivel superior del programa. Si se encontrara en una subrutina no se produciría el retorno correcto al BASIC, porque se extrae únicamente una dirección de retorno del stack mediante POP IX.
Elaborando el principio de esta rutina es posible confeccionar utilidades de programación como por ejemplo un simulador de paso a paso. Buenos paquetes de programas contienen tales programas de comprobación.

RUTINA BUSQUEDA

Para completar la colección de rutinas de MONITOR, continuamos con una rutina que busca en la memoria por una secuencia determinada de caracteres. Si se quiere asimismo buscar en el ROM, debe hacerse como en la rutina MONITOR utilizando un comando FAR CALL &18. La rutina SALHEX también debe integrarse.

```
A000             10                              ; BUSQUEDA
A000             20         ORG   &A000
A000             30 PRINT   EQU   &BB5A
A000             40 START   DS    2
A002             50 ENDE    DS    2
A004             60 LONG    DS    1              ; Long. sec. caracteres
A005             70 TAB1    DS    1              ; Inicio secuencia car.
A006             80 TAB2    DS    19             ; Reserva max. 20 car.
A019             90                              ; Inicio busqueda
A019 3A05A0     100         LD    A,(TAB1)       ; Primer elemento
A01C ED5B00A0   110         LD    DE,(START)     ; Inicio bloque
A020 2A02A0     120         LD    HL,(ENDE)      ; Final bloque
A023 B7         130         OR    A              ; Carry=0
A024 ED52       140         SBC   HL,DE          ; Long. bloque
A026 23         150         INC   HL             ; Despues BC
A027 44         160         LD    B,H            ; Cargar
A028 4D         170         LD    C,L
A029 EB         180         EX    DE,HL          ; Inicio despues HL
A02A EDB1       190 COMP    CPIR                 ; Busca hasta
A02C CC0000     200         CALL  Z,FOUND        ; igualdad = > FOUND
A02F E0         210         RET   PO             ; RET por fin busca
A030 18F8       220         JR    COMP
**** Linea 200 : FOUND = &A032
A032 F5         230 FOUND   PUSH  AF
A033 C5         240         PUSH  BC
A034 E5         250         PUSH  HL
A035 3A04A0     260         LD    A,(LONG)
A038 4F         270         LD    C,A            ; Almacena longitud
```

```
A039 0600      280         LD   B,0        ; en BC
A03B 0D        290         DEC  C          ; compara desde 2do ele.
A03C 28FE      300         JR   Z,OK
A03E 1106A0    310         LD   DE,(TAB2) ; Dir. 2do elemento
A041 1A        320 COMP1   LD   A,(DE)     ; siguiente elemento
A042 EDA1      330         CPI             ; Comparar
A044 13        340         INC  DE         ; Incrementa apuntador
A045 20FE      350         JR   NZ,RUECK  ; Desigual = > CPIR
A047 EA42A0    360         JP   PE,COMP1  ; Seguir si BC no 0
**** Linea 300 : OK=&A04B
A04A E1        370 OK      POP  HL         ; Dir. sec. encontrada+1
A04B 2B        380         DEC  HL
A04C 7C        390         LD   A,H        ; Highbyte
A04D CD0000    400         CALL SALHEX     ; Salida
A04E 7D        410         LD   A,1        ; Low-byte
A051 CD0000    420         CALL SALHEX     ; Salida
A054 3E20      430         LD   A,32       ; Espacio en blanco
A056 CD5ABB    440         CALL PRINT      ; Salida
A059 23        450         INC  HL         ; Restaurar valor anter.
A05A C1        460         POP  BC
A05B F1        470         POP  AF
A05C C9        480         RET             ; Seguir busqueda
**** Linea 350 : RUECK = &A05E
A05D E1        490 RUECK   POP  HL         ; No igual
A05E C1        500         POP  BC
A05F F1        510         POP  AF
A060 C9        520         RET             ; continuar busqueda
.......          .           .
.......          .           .
```

RUTINA SALHEX

```
Programa : BUSCAR
Start:     &A000        Ende :  &A061
Longitud : 0062
0 Error
```

```
Tabla de variables:
PRINT    BB5A      START   A000      ENDE    A002
LONG     A004      TAB1    A005      TAB2    A006
COMP     A02A      FOUND   A032      COMP1   A042
OK       A04B      RUECK   A05E
```

La secuencia a buscar debe almacenarse antes de la llamada a la rutina en la dirección &A005 mediante CALL &A019. Esto y el pokear la longitud de la dirección inicial y final se realiza a través de un programa BASIC.

ENTRADA DE DATOS

Hasta ahora hemos conocido rutinas del sistema que permiten la salida de lenguaje máquina. Ahora nos ocuparemos de la entrada de datos. Datos variables, como dirección inicial y final hemos transferido al programa máquina de forma laboriosa mediante comandos POKE desde el BASIC.
El BASIC del CPC nos ofrece la posibilidad de transferir datos mediante un comando CALL. Con este comando es posible la transferencia de hasta 32 números de 2 bytes. El comando CALL ampliado tiene el siguiente formato:

```
CALL Dirección,Expresión,Expresión,....
```

Expresión puede representar cualquier número de 16 bits, una función o una variable cuyo valor corresponda a un número de 16 bits. Dado que pueden transferirse hasta 32 números, no existe la posibilidad de almacenarlos todos en los registros. Los números transferidos se almacenan en el stack. El ACU contiene la cantidad de expresiones transferidas. El registro DE contiene el último valor indicado. La dirección del stack en la cual se encuentra la última entrada de los números transferidos, se transfiere a través del registro IX. El registro C contiene el estado ROM/RAM (ver FAR CALL RST &18), el mismo en la llamada estándard siempre es &FF

(RAM seleccionados). HL siempre apunta a la dirección en la cual termina el comando CALL. Resumiendo:

Registro	No hay transferencia de números	Transferencia de N números
A	0	n (Cantidad)
F	F=&68 (Z=1)	F=&28 (Z=0!)
B	&20	&20-n
C	&FF (estado)	&FF
DE	Dirección destino bifurcación	último nro. transferido
HL	Dirección final comando CALL	
IX	Dirección Stack &BFFE	Dirección stack del último elemento =&BFFE-2*n

Utilicemos esta manera de entrada para alimentar el programa MONITOR con los valores correspondientes. Se deben transferir:
La dirección inicial
La dirección final
El estado RAM/ROM (FAR CALL)

La llamada tiene el formato siguiente:

```
CALL &A000, Dirección inicial, Dir. final, estado
```

Las modificaciones de programa son las siguientes:

```
10 ' ORG &A000
15 ' CP  3        ; 3 Parametros
20 ' RET NZ       ; No, final
```

En primer lugar se comprueba si se han entrado 3 valores (A=3). De lo contrario se produce un retorno al BASIC.

```
25 '  LD  A,D
30 '  OR  A
35 '  RET NZ
40 '  LD  A,E
45 '  LD  (STATUS),A
```

En las líneas 25, 30 y 35 se comprueba si D=0. Si D es diferente de 0 finaliza el programa. El estado es un número de 1 byte. Pueden entrarse asimismo números de 2 bytes. Por esta razón ha de comprobarse si el segundo byte, el high byte es igual a cero. En las líneas 40 y 45 se almacena el estado introducido en la posición correcta para el comando RST &18.

```
50 '  LD   E,(IX+2)
55 '  LD   D,(IX+3)
60 '  LD   L,(IX+4)
65 '  LD   H,(IX+5)
```

En las líneas 50 y 55 se carga la dirección final transferida en DE. En líneas 60 y 65 se almacena la dirección inicial en HL.

```
70 '  RST   &18
75 '        DW   Vektor
80 '        RET              ; Retorno al BASIC
85  'VEKTOR DW   MONITO      ; Direccion vector bifurcacion
90  'STATUS DS   1           ; Estado ROM/RAM
95  'ENDE   DS   2
100 'MONITO LD   (ENDE),DE

 ..............  el resto ya lo conocen
```

Después de la traducción del programa completo puede lograr por ejemplo con

```
CALL &A000,&CC50,&CE60,252
```

que se visualice el mensaje de error del ROM BASIC.

Otra rutina importante es la utilizada para la entrada de una tecla. Después de la llamada de &PB06 el ordenador espera hasta que se pulse una tecla. El valor de la tecla pulsada se refleja en el ACU.
Con la siguiente rutina simple podemos realizar una entrada simple a través del teclado.

```
A000             10        ORG   &A000
A000             20 GET    EQU   &BB06
A000             30 PRINT  EQU   &BB5A
A000 CD06BB      40 EIN    CALL  GET
A003 CD5ABB      50        CALL  PRINT
A006 FE0D        60        CP    13              ; Enter ???
A008 20F6        70        JR    NZ,EIN
A00A 3E0A        80        LD    A,10
A00C CD5ABB      90        CALL  PRINT           : Avance linea
A00F C9         100        RET

Programa: entrada
Start:    &A000          Ende:    &A00F
Longitud: 0010
0 Errores
Tabla de variables
 GET    BB06    PRINT  BB5A    EIN    A000
```

Nota: En este tipo de entrada funcionan todos los caracteres de control CTRL, por ejemplo CTRL L para borrar la pantalla o CTRL G para que se escuche el timbre.

AMPLIACION DE COMANDOS CON RSX

La ampliación del conjunto de comandos del BASIC del CPC puede ampliarse de diferentes formas. Solamente en el 464 pueden ampliarse las rutinas presentes en el ROM del BASIC mediante "áreas de parcheo" en el RAM. Esta posibilidad tiene sus

restricciones dado que solamente pueden efectuarse 9 parches. En el caso del CPC 6128/664 ello no es posible por falta de espacio. Sin embargo en todos los CPC existe el método estándard que facilita la ampliación con algunos comandos, y que se utiliza en todos los comandos de diskette AMSDOS.

Seguramente alguna vez ha llamado la modalidad CPM mediante |CPM si posee un dispositivo de diskette. El comando CPM comienza con ("ñ"). Al introducir "CPM" (sin raya) se obtiene un "Syntax Error". Si trabaja sin floppy (posibilidad para 464) se obtiene después de |ñCPM un "Unknown Command Error".

Se trata de un comando que solamente es válido si hay un floppy conectado y por ello representa una ampliación.

El símbolo "ñ" indica al sistema que a continuación hay un comando ampliado. En los ordenadores CPC existe un método especial que permite reconocer y ejecutar de entrada posibles comandos ampliados. Este método de incorporación de comandos se llama RSX. RSX ("Resident System Extention") significa ampliación fija del sistema, es decir que el RSX ha sido pensado para la incorporación de nuevos comandos residentes en ROMs adicionales (ROMs de expansión). Para el trabajo con el floppy existe un ROM de expansión que incorpora asimismo los comandos ñCPM, ñERA, etc.

El método RSX puede asimismo utilizarse para incorporar rutinas propias como verdaderos comandos y utilizarlos con comodidad.

Explicaremos a continuación el método RSX con el siguiente ejemplo del comando DOKE.

DOKE es casi un "Doble comando POKE". Con |POKE puede almacenarse un valor entre 0 y 255 en una posición de memoria. DOKE almacena un valor entre 0 y 65500 en 2 posiciones consecutivas de memoria. Para ello el valor se descompone en Low-byte y High-byte, donde el Low-byte se almacena en la posición inferior de memoria y el High-byte en la posición de memoria con

dirección superior. Con DOKE se simplifica la modificación de muchos parámetros del sistema almacenados en el RAM (usualmente como PEEKs y POKEs).

La transferencia de parámetros en el caso de los comandos RSX funciona de forma análoga a la del comando CALL.

El comando DOKE necesita 2 parámetros:
La dirección a partir de la cual se desea almacenar el valor y el valor que se desea almacenar a partir de esta dirección. El comando tiene el siguiente formato:

|DOKE, Dirección,Valor

Los parámetros se transfieren de la siguiente forma:

A: Contiene la cantidad de los parámetros transferidos.

Flags: Zero Flag = 1 si no se transfieren parámetros, de lo contrario cero.

B: Contiene 32-cantidad de parámetros.

DE: Contiene último parámetro transferido.

IX: Dirección del último elemento. El penúltimo parámetro se encuentra entonces en dirección IX+2, el siguiente en dirección IX+4, etc.

La rutina DOKE es la siguiente:

```
100 ' LD  L,(IX+2) ; Dirección  transferida
110 ' LD  H,(IX+3) ; cargada en registro HL
120 ' LD  (HL),E   ; Almacenar Low-byte
130 ' INC HL       ; Incrementar direccion a High-byte
140 ' LD  (HL),D   ; Almacenar High-byte
150 ' RET          ; Final
```

Con CALL &A000, dirección, valor podemos llamar el comando DOKE después de ensamblar la rutina, suponiendo que éste se ha almacenado previamente a partir de &A000. Pruebe usted por ejemplo:

```
CALL &A000,..,..
```

De esta forma tenemos una rutina DOKE. Nos molesta únicamente la llamada mediante CALL. DOKE debería llamarse mediante DOKE y para ello debemos confeccionar una pequeña rutina de adaptación. Utilizamos para ello la rutina "Logext" del sistema operativo que se ocupa de la expansión. Solamente hemos de transferir algunos valores al Logext.
Para incorporar el DOKE, el sistema necesita "saber" lo siguiente:

1) Nombre de la ampliación.
2) Dirección de la rutina
3) Dirección de 4 bytes no utilizados que se utilizan internamente para la administración de la rutina.

La dirección de los 4 bytes del sistema se cargará en el registro HL antes de la llamada de Logext. Se carga también BC con la dirección inicial de una tabla en la cual se encuentran más informaciones.

Esta tabla contiene en primer lugar la dirección de una segunda tabla que contiene el nombre o varios nombres de diferentes rutinas que deben incorporarse. El segundo elemento de la primera tabla contiene un comando de bifurcación a la rutina o las rutinas a incorporar. Si se incorporan varias rutinas, los comandos de bifurcación han de corresponderse con los nombres de las rutinas.

Como ya hemos dicho la segunda tabla contiene los nombres de las rutinas. Para poder diferenciar los diversos nombres, el bit 7 de la última letra del nombre debe estar activado permanentemente.

Veamos nuestro ejemplo:

```
10 ' LD   BC,RSXTAB ; Direccion 1ra. tabla
20 ' LD   HL,SYSBYT ; Direccion byte del sistema
30 ' CALL &BCD1     ; Llamar rutina Logext
40 ' RET            ; Incorporacion finalizada
50 ' RSXTAB DW NAMTAB ;  Tabla 1 1ra dir. = dir. inicial tabla 2
60 ' JP   START     ; Comandos bifurcacion a la rutina
70 ' NAMTAB DM "DOK" ; Tabla de nombres
80 ' DB &25         ; =ASC("E")+128 activa bit 7
90 ' DB 0           ;  Null Byte marca final tabla nombres
95 ' SYSBYT DS 4    ;  Reserva 4 bytes internos
100 ' START LD ..... ; Aquí comienza la rutina

              ........
```

Una vez ensamblado el programa (o el cargador BASIC) se incorpora el comando DOKE con CALL &A000. A partir de este momento el DOKE es una ampliación de comandos (RSX), es decir el comando puede utilizarse como cualquier otro comando en modalidad directa o en un programa.
El nuevo comando tiene el siguiente formato:

DOKE Dirección, Valor

Observen que esta rutina sólo es necesario llamarla una única vez. Los 4 bytes del sistema se utilizan entre otras cosas para apuntar eventualmente a otras tablas RSX. Si la misma rutina se inicializa por segunda vez, el apuntador que debe indicar la siguiente tabla RSX señalará lógicamente a la misma tabla.

Por ello puede ocurrir que el reconocimiento del comando quede clavado en un bucle sin fin, cosa muy inoportuna. El siguiente programa evita que se produzca una segunda llamada grabando el comando RET en el comienzo de la rutina INIT.

```
A000 010000      10 INIT   LD     BC,RSXTAB
A003 210000      20        LD     HL,SYSBYT
A006 CDD1BC      30        CALL   &BCD1
A009 3EC9        40        LD     A,&C9
A00B 3200A0      50        LD     (INIT),A
A00E C9          60        RET
**** Linea 10 : RSXTAB=&A00F
A00F 0000        70 RSXTAB DW     NAMTAB
A011 C30000      80        JP     START
**** Linea 70 : NAMTAB=&A014
A014 444F4B      90 NAMTAB DM     "DOK"
A017 C5         100        DB     &05
A018 00         110        DB     0
**** Linea 20 : SYSBYT=&A019
A019            120 SYSBYT DS     4
**** Linea 80 : START=&A01D
A01D DD6E02     130 START  LD     L,(IX+2)
A020 DD6603     140        LD     H,(IX+3)
A023 73         150        LD     (HL),E
A024 23         160        INC    HL
A025 72         170        LD     (HL),D
A026 C9         180        RET


Programa: DOKE
Start:    &A000          Ende:    &A026
Longitud: 0027
0 Errores

Tabla de variables
 INIT    A000    RSXTAB   A00F    NAMTAB   A014    SYSBYT   A019
 START   A01D
```

Otro ejemplo para una rutina RSX incorporada ya la hemos visto en relación con el desensamblador. El siguiente listado ensamblador refleja exactamente este comando incorporado en el programa mencionado.

Una nueva posibilidad de este programa consiste en que pueden asimismo traspasarse valores desde el programa máquina al BASIC. Para ello se utiliza la función de apuntador de variables C".

Este símbolo "@" (A de arrobas), representa una función que hasta el momento no se ha mencionado en ningún manual del CPC. Para poder comprender y aplicar correctamente esta función, hemos de ocuparnos un poco del almacenamiento interno de las variables del BASIC.

Existen tres modalidades diferentes de la representación de datos en el ordenador:

1. Integer INT - Número entero
2. Real REL - Número decimal con coma
3. Strings STR - Alfanumérico

La primera modalidad, la representación de números enteros ya la hemos visto. Un número entero se descompone en Low-byte y High-byte. El bit 7 del High-byte se utiliza como identificación de signo (+/-). Cero significa número positivo y uno número negativo. Los números negativos se representan en el complemento de 2, es decir el valor del número se complementa y se le suma 1 (vean capítulo 4.6, comandos aritméticos). De esta forma las variables enteras o constantes enteras pueden obtener valores enteros entre -32768 y +32767.

Decimal	Binario	Hexadecimal
-32768	1 000 0000 0000 0000	80 00
-32767	1 000 0000 0000 0001	80 01
-32766	1 000 0000 0000 0010	80 02
-32765	1 000 0000 0000 0011	80 03
	.....	
-2	1 111 1111 1111 1110	FF FE
-1	1 111 1111 1111 1111	FF FF
0	0 000 0000 0000 0000	00 00
1	0 000 0000 0000 0001	00 01
2	0 000 0000 0000 0010	00 02
	.....	
32766	0 111 1111 1111 1110	7F FE
32767	0 111 1111 1111 1111	7F FF

La función @ indica la dirección en la cual se almacena el valor de una variable. En el caso de variables enteras es la dirección del Low-byte. Prueben ahora lo siguiente:

* W%=&1234: POKE @W%,&56: PRINT HEX$(W%) * nos da &1256.

De esta manera pueden comunicarse valores desde el programa máquina al BASIC, transfiriendo los apuntadores de la variable en el momento de la llamada de la rutina. El programa máquina almacena el valor a comunicar, en esta dirección. Después del retorno al BASIC el valor queda disponible en la variable correspondiente. Ofrecemos a continuación el listado de ensamblador del comando RPEEK que realiza un retorno del valor de la forma expuesta.

```
A000              10                ; RPEEK
A000              20                ; Comando RSX
A000              30                ; Formato:   "|RPEEK, variable
                        transferencia(INT),direccion,estado ROM/RAM"
9F20              40          ORG   &9F20
9F20              50                ; CPC 6128 ; 464,    664
9F20              60 OPMIS    EQU   &D055  ;      &CFED, &D058
9F20              70 IMPARG   EQU   &C21D  ;      &C205, &CB50
9F20 010000       80 INIT     LD    BC,RSXTAB
9F23 210000       90          LD    HL,SYSBYT
9F26 CDD1DC      100          CALL  &BCD1 ; Logext
9F29 3EC9        110          LD    A,&C9
9F2B 32209F      120          LD    (INIT),A
9F2E C9          130          RET
**** Linea 80 : RSXTAB &9F2F
9F2F 0000        140 RSXTAB  DW    NAMTAB
9F31 C30000      150          JP    START
**** Linea 140 : NAMTAB = &9F34
9F34 52504545    160 NAMTAB  DM    "RPEE"
9F38 CB          170          DB    &CB
9F39 00          180          DB    0
**** Linea  90 : SYSBYT = &9F3A
9F3A             190 SYSBYT  DS    4
**** Linea 150 : START = &9F3E
9F3E DF          200 START   RST   &18     ; Por las rutinas de error
9F3F 0000        210          DW    VEKTO1 ; Bifurcacion por Far Call
9F41 C9          220          RET          ; Con Upper ROM activado
**** Linea 210 : VEKTO1 = &9F42
9F42 0000        230 VEKTO1  DW    ANFANG
9F44 FD          240          DB    253
**** Linea 230 : ANFANG = &9F45
9F45 FE03        250 ANFANG  CP    3         ; 2 Parametros?
9F47 C255D0      260          JP    NZ,OPMIS ; No, falta operando
9F4A 7A          270          LD    A,D
9F4B FE00        280          CP    0
9F4D C21DC2      290          JP    NZ,IMPARG ; Estado sup.255, error
9F50 7B          300          LD    A,E
```

```
9F51 320000     310         LD    (STATUS),A  ; Estado Far Call
9F54 DD6E02     320         LD    L,(IX+2)    ; Direccion Low
9F57 DD6603     330         LD    H,(IX+3)    ; Direccion High
9F5A DF         340         RST   &18         ; Bifurcar rutina
9F5B 0000       350         DW    VEKTO2      ; Leer byte
9F5D DD6E04     360         LD    L,(IX+4)    ; Leer apuntador
9F60 DD6605     370         LD    H,(IX+5)    ; variable INT
9F63 77         380         LD    (HL),A      ; Trasp. val. a var.
9F64 97         390         SUB   A
9F65 23         400         INC   HL          ; High byte var.=0
9F66 77         410         LD    (HL),A
9F67 C9         420         RET
**** Linea 350 : VEKTO2 = &9F68
9F68 0000       430 VEKTO2  DW    REDBYT
**** Linea 310 : STATUS = &9F6A
9F6A FD         440 STATUS  DB    253
**** Linea 430 : REDBYT = &9F6B
9F6B 7E         450 REDBYT  LD    A,(HL)      ; Leer rutina
9F6C C9         460         RET

End Assumed


Program  : RPEEK
Start    : &9F20        End    : &9F6C
Longitud : 004D
0 Errores


Tabla de Variables

 OPMIS   D055    IMPARG   C21D    INIT     9F20    RSXTAB   9F2F
 NAMTAB  9F34    SYSBYT   9F3A    START    9F3E    VEKTO1   9F42
 ANFANG  9F45    VEKTO2   9F68    STATUS   9F6A    REDBYT   9F6B
```

Después de haber explicado la transferencia numérica nos dedicaremos ahora a otro tipo de variables, la variable de STRING.

En un string se almacenan datos alfanuméricos, es decir caracteres como letras o cifras. Cada carácter tiene un código asignado. Los códigos de caracteres para los valores de 0 a 127 son los correspondientes al llamado American Standard Code for Information Interchange (ASCII).

Para almacenar un carácter se necesita exactamente un byte. Un "String" (inglés string: cadena), es una secuencia de códigos de caracteres consecutivos. Dado que un carácter corresponde a un byte, el string se almacena en secuencia consecutiva de posiciones de memoria en el RAM. Para asignar un string determinado a una determinada variable son necesarias 2 informaciones que componen el llamado "String Descriptor".

1. La dirección de la primera posición de memoria que contiene el primer código de carácter del string.

2. La longitud del string, es decir la cantidad de bytes que forma la cadena de caracteres.

Estos 2 datos se almacenan junto con el nombre de la variable en el área de variables del BASIC. La cadena de caracteres en sí se encuentra en otra posición, que puede ser el mismo programa o un área de memoria reservada especialmente para strings.

En BASIC podemos utilizar nuevamente la función "@" para leer la dirección del string descriptor de una variable, que se compone de 3 bytes:

1. Byte : Longitud del string.
2. y 3. Byte : Dirección inicial del string.

La función "@" proporciona la dirección del primer byte del string descriptor.

Prueben el programa siguiente:

```
10 X$ = "ZZZeichenkeTTTTe"
20 AD = @(X$)
30 LA = PEEK(AD)
40 ST = PEEK(AD+1) + 256 * PEEK(AD+2)
50 FOR I=ST TO ST+LA-1
60 PRINT CHR$(PEEK(I));
70 NEXT
```

El símbolo "@" en relación con el tratamiento de strings ya ha sido utilizado en el programa generador de fuentes. El programa máquina utilizado era capaz de convertir una línea BASIC entrada como string en una línea BASIC real. De esta manera se simula la entrada manual del string en modalidad directa. A continuación este string se traduce a una línea como en modalidad directa.

La única restricción del siguiente programa es que la línea a generar debe tener un número de línea superior al actual, en caso contrario el programa no encontraría la posición correcta después del RET.

```
A000              10                ; LINER
A000              20                ; Genera lineas BASIC
A000              30                ; 1. en modalidad directa
A000              40                ; 2. en programa, si numero linea
A000              50                ; mayor que actual
A000              60                ; A$ contenido de linea en ASCII
A000              70                ; finalizada con Null Byte
A000              80                ;
A000              90                ; Formato: CALL &A000,@A$
A000             100                ;
9F00             110          ORG   &9F00
9F00             120                ; CPC 6128; 464,   664
9F00             130 CHRSKP   EQU   &DE4D      ; &DD61, &DE52
9F00             140 TESTER   EQU   &EECF      ; &EE04, &EED4
9F00             150 ASSEMB   EQU   &E7A5      ; &E6C6, &E7AA
9F00 DF          160          RST   &18     ; Far Call
9F01 0000        170          DW    VEKTOR ; Dado que BASIC ROM
9F03 C9          180          RET          ; ha de estar activado
**** Linea 170 : VEKTOR = &9F04
9F04             190 VEKTOR   DW    START
9F06 FD          200          DB    253     ; Low ROM OFF Up ROM ON
**** Linea 190 : START = &9F07
9F07 EB          210 START    EX    DE,HL   ; Direccion transf. a HL
9F08 23          220          INC   HL      ; Sobreleer Long. string
9F09 5E          230          LD    E,(HL) ; Low byte Direcc. string
9F0A 23          240          INC   HL
9F0B 56          250          LD    D,(HL)    ; High byte dir. string
9F0C EB          260          EX    DE,HL     ; Direccion string a HL
9F0D CD4DDE      270          CALL  CHRSKP
9F10 B7          280          OR    A
9F11 CB          290          RET   Z
```

```
9F12 CDCFEE     300         CALL  TESTER
9F15 D0         310         RET   NC
9F16 CDA5E7     320         CALL  ASSEMB  ; Traducir e insertar lin.
                                          ; HL debe apuntar a primer
                                          ; byte de la linea ASCII
9F19 C9         330         RET           ; Final


Program  : LINER
Start    : &9F00        End    : &9F19
Longitud : 001A
0 Errores


Tabla de Variables

 CHRSKP  DE4D    TESTER  EECF    ASSEMB  E7A5    VEKTOR  9F04
 START   9F07
```

A continuación ofrecemos el programa máquina incorporado en un programa, que genera un cargador BASIC de los programas residentes en la memoria en código objeto, y codificados en lenguaje máquina.

Es muy útil para transferir programas máquina al sistema, incluso para quienes no posean un compilador Assembler.

```
10 MEMORY &9EFF
20 LIN=&9F00
30 FOR I=LIN TO LIN+&19 : REED A$ : W=VAL("&"+A$)
40 S=S+W : POKE I,W : NEXT
50 IF S <> 4273 THEN PRINT "ERROR EN DATAS" : END
51 ' 464 : IF S <> 4121 THEN .......
52 ' 664 : IF S <> ???? THEN .......
60 PRINT "OK!" : S=0
70 DATA   DF,04,9F,C9,07,9F,FD,EB
80 DATA   23,5E,23,56,EB,CD,4D,DE
81 '464 :   ,  ,  ,  ,  ,  ,61,DD
82 '664 :   ,  ,  ,  ,  ,  ,52,DE
90 DATA   B7,C8,CD,CF,EE,D0,CD,A5
91 '464 :   ,  ,  ,04,  ,  ,  ,C6
92 '664 :   ,  ,  ,9B,EF,  ,  ,AA
100 DATA  E7,C9
101 '464: E6,..
102 '664: E7,..
110 INPUT "DIRECCION INICIAL",ST
120 INPUT "DIRECCION FINAL",EN
130 AZ$=CHR$(34) : REM Comillas
140 NU$=CHR$(0)+CHR$(0) : REM Marca final
150 A$="1000 FOR I=&"+HEX$(ST,4)+" TO &"+HEX$(EN,4)+NU$
160 CALL LIN,@A$
170 A$="1010 READ A$ : W=VAL("+AZ$+"&H"+AZ$+"+A$)"+NU$
180 CALL LIN,@A$
190 A$="1020 S=S+W : POKE I,W : NEXT"+NU$
200 CALL LIN,@A$
210 Z=1050
220 I=ST
230 A$=""
240 J=0
250 W=PEEK(I+J) : S=S+W
260 A$=A$+HEX$(W,2)+","
270 IF I+J < EN THEN J=J+1 : IF J < 8 THEN 250 ELSE EF=-1
280 A$=LEFT$(A$,LEN(A$)-1)
290 A$=STR$(Z)+" DATA "+A$+NU$
300 CALL LIN,@A$
310 Z=Z+10
```

```
320 IF NOT EF THEN I=I+8 : GOTO 230
330 A$="1030 IF S <> "+STR$(S)+" THEN PRINT "+AZ$+"ERROR EN DATAS"
    + AZ$ + ": END" + NU$
340 CALL LIN,@A$
350 A$="1040 PRINT "+AZ$ + "OK!" + AZ$ +" : END " + NU$
360 CALL LIN,@A$
370 DELETE 10-380
380 END
```

El programa de la página anterior ha sido el último del presente libro. Naturalmente hay muchas cosas más para explicar referente a la programación práctica, pero es de gran importancia que usted mismo realice abundantes ejercicios. Aconsejamos la lectura del libro "Consejos y trucos del CPC" tomo 2, donde se ofrecen aplicaciones del lenguaje máquina, como por ejemplo el programa XREF con el cual se realiza la revisión general de las variables del Desensamblador/Simulador.

Esperamos encuentren entretenida la programación en lenguaje máquina.

CAPITULO VII: PERSPECTIVAS

Acaban de aprender las técnicas fundamentales de programación, así como los programas de ayuda para la confección de programas en lenguaje máquina.

La programación en ensamblador es inevitable para resolver algunos problemas de programación. Los tiempos del desarrolo del software son sin duda mucho más largos que los de los programas confeccionados con lenguajes de alto nivel. Por esta razón son necesarios buenos programas de desarrollo para la programación efectiva.

Las propiedades de tales programas los discutiremos brevemente. Un paquete de programas para desarrollo de programas máquina debe incorporar por lo menos un programa assembler y un amplio programa monitor.

El ensamblador es necesario para el desarrollo de programas voluminosos. Adicionalmente a los pseudo-comandos conocidos por ustedes, muchos ensambladores ofrecen la posibilidad de simplificar aún más el desarrollo de programas. Por ejemplo entre ellos cuenta la definición de macro-instrucciones, el ensamblaje condicionado y el acceso a programas o variables externas.

MACROS:

Frecuentemente ocurre que una secuencia determinada de comandos se repite varias veces en un programa. Utilizando macros se evita la codificación repetida de la secuencia completa de instrucciones. Con la ayuda de una definición macro, puede darse un nombre a una secuencia utilizándolo en el programa fuente en lugar de la secuencia de comando.

El ensamblador reemplaza el nombre de macro por la secuencia correspondiente. Los programas fuente resultan más cortos y más claros con la utilización de macros.

ENSAMBLAJE CONDICIONADO:

Con el ensamblaje condicionado es posible traducir determinadas porciones del programa dependiendo de una condición. Es posible por ejemplo que un programa fuente general pueda confeccionarse como una administración de ficheros, utilizándole a continuación para confeccionar aplicaciones diversas.

PROGRAMAS Y VARIABLES EXTERNAS

Al programar en ensamblador es de gran utilidad la técnica de programación estructurada, es decir que problemas grandes se subdividen en porciones reducidas, confeccionando cada porción de programa por sí mismo. Muchas veces se utilizan las mismas subrutinas, por ejemplo nosotros hemos utilizado la rutina para la salida hexadecimal de un carácter en programas diversos. Rutinas de este tipo que se utilizan con cierta frecuencia y variables que se requieren repetidas veces componen una bibioteca de programas/variables, en el caso de un ensamblador profesional. Las rutinas se reconocen por su nombre en el programa fuente y pueden cargarse automáticamente desde el cassette/diskette, para incorporarlos en el programa objeto.

El programa que realiza la unión de diferentes programas máquina se conoce como "LINKER" (link: inglés: une). Con el mismo se relaciona también un programa llamado "RELOCATOR" (reubicador), que se utiliza para corregir las direcciones que se modifican al introducir y desplazar módulos. Los programas que contienen también esta posibilidad son muy amplios pero relativamente caros. La programación en cambio se agiliza y se hace mucho más confortable. Muchos ensambladores poseen un editor propio, es decir, la entrada de comandos de ensamblador no necesita la numeración de línea, siendo ésta únicamente orientativa.

Existen algunos programas adicionales de utilidad para el ensamblador. La mayoría de ellos se agrupan en un MONITOR. Las rutinas estándard de un MONITOR ya las hemos revisado. El disassembler en la mayoría de los casos se encuentra incorporado

en el programa MONITOR.

Una particularidad importante de un programa MONITOR es la facilidad que ofrece de comprobar programas. La posibilidad de incorporar un Breakpoint es el método más sencillo de comprobación. Rutinas ampliadas de comprobación se agrupan en un programa denominado DEBUGGER (depurador). El programa más importante es el simulador de paso a paso, similar a la función TRON del BASIC del CPC.

El poseer buenos programas de utilidad para el desarrollo de software no lo es todo. Es mucho más importante el paso a la práctica de la programación. Este libro les ha facilitado las técnicas fundamentales necesarias para la programación del Z80. Sólo practicando podrán realmente aprender el lenguaje máquina. Les deseamos mucha suerte en la confección de sus propios programas !!!

ANEXO

RUTINAS DE UTILIDAD DEL SISTEMA

Dirección &0000: RST 0: RESET

Llamada con CALL 0. Actúa como Conectar/Desconectar (RESET).

Dirección &0008 RST &08- Low-Jump

Bifurca a una dirección en el ROM del sistema operativo o al RAM solapado. Los bits 14 y 15 determinan la selección ROM/RAM. Un bit activado significa RAM y un bit desactivado ROM. bit 14 determina el área inferior de direcciones (&0-&3FFF) y bit 15 determina el área superior (&C000-&FFFF).

Dirección &000C: JP (HL) con selección ROM/RAM.

Los bits 14 y 15 de HL tienen la misma función que en RST &08.

Dirección &0010: RST &10: Side Call

Utilizado para la llamada de una rutina en el ROM de expansión.

Dirección &0018: RST &18: Far Call

Utilizado para la llamada de una rutina del ROM o del RAM. (Ver capítulo 6.2).

Dirección &0020: RST &20: RAM-Lam

El ACU se carga con el valor de la dirección a la cual apunta HL. En este caso queda siempre seleccionado el RAM.

Dirección &0028: RST &28: Firm Jump

Utilizado para la llamada de una rutina en el sistema operativo (Firmware). La dirección se indica directamente a continuación del comando.

Dirección &0030: RST &30: User Restart

Reservado para programas del usuario.

Dirección &BB06: KM(Key Manager)- Wait Char

El código ASCII de la tecla pulsada se transfiere al ACU.

Dirección &BB24: KM - Get Joystick

El registro H contiene el estado del joystick 1, el registro L el del joystick 2.

Dirección &BB5A: TXT Output

Visualiza en la pantalla el valor del carácter contenido en el ACU.

Dirección &BB6C: TXT Clear Window

Borra la ventana actual de la pantalla.

Dirección &BB75: TXT Set Cursor

H/L corresponde a línea/columna.

Dirección &BB78: TXT Get Cursor

Dirección &BB81: TXT Cursor On

Dirección &BB84: TXT Cursor Off

Dirección &BBC0: GRA Move Absolut

El registro DE es la coordenada X, HL la coordenada Y.

Dirección &BBC6: GRA Ask Cursor

Ocupación de registro como en Move Absolut

Dirección &BBEA: GRA Plot Absolut

Dirección &BC9B: CAS (/Disk) Cas Catalog

Dirección &BCD1: KL (Kernel) Log Ext

Incorpora ampliaciones RSX (ver capítulo 6.2).

Dirección &BD0D: KL Time Please

Retorna el valor del Timer como valor de 4 bytes en DE y HL.

Dirección &BD10: KL Time Set

Dirección &BD1C: MC (Machine) Set Screen Mode

Inicializa la modalidad de pantalla con el valor del ACU.

Dirección &BD2B: MC Print Char

Retorna el valor del ACU a través de la impresora.

Dirección &BD37: Jump Restore : "Freno de emergencia"

Inicializa todos los vectores de bifurcación eventualmente "dañados" con sus valores iniciales.

TABLA DE CONVERSION

decimal	hex	binario	decimal	hex	binario
0	&00	&X00000000	26	&1A	&X00011010
1	&01	&X00000001	27	&1B	&X00011011
2	&02	&X00000010	28	&1C	&X00011100
3	&03	&X00000011	29	&1D	&X00011101
4	&04	&X00000100	30	&1E	&X00011110
5	&05	&X00000101	31	&1F	&X00011111
6	&06	&X00000110	32	&20	&X00100000
7	&07	&X00000111	33	&21	&X00100001
8	&08	&X00001000	34	&22	&X00100010
9	&09	&X00001001	35	&23	&X00100011
10	&0A	&X00001010	36	&24	&X00100100
11	&0B	&X00001011	37	&25	&X00100101
12	&0C	&X00001100	38	&26	&X00100110
13	&0D	&X00001101	39	&27	&X00100111
14	&0E	&X00001110	40	&28	&X00101000
15	&0F	&X00001111	41	&29	&X00101001
16	&10	&X00010000	42	&2A	&X00101010
17	&11	&X00010001	43	&2B	&X00101011
18	&12	&X00010010	44	&2C	&X00101100
19	&13	&X00010011	45	&2D	&X00101101
20	&14	&X00010100	46	&2E	&X00101110
21	&15	&X00010101	47	&2F	&X00101111
22	&16	&X00010110	48	&30	&X00110000
23	&17	&X00010111	49	&31	&X00110001
24	&18	&X00011000	50	&32	&X00110010
25	&19	&X00011001	51	&33	&X00110011

decimal	hex	binario
52	&34	&X00110100
53	&35	&X00110101
54	&36	&X00110110
55	&37	&X00110111
56	&38	&X00111000
57	&39	&X00111001
58	&3A	&X00111010
59	&3B	&X00111011
60	&3C	&X00111100
61	&3D	&X00111101
62	&3E	&X00111110
63	&3F	&X00111111
64	&40	&X01000000
65	&41	&X01000001
66	&42	&X01000010
67	&43	&X01000011
68	&44	&X01000100
69	&45	&X01000101
70	&46	&X01000110
71	&47	&X01000111
72	&48	&X01001000
73	&49	&X01001001
74	&4A	&X01001010
75	&4B	&X01001011
76	&4C	&X01001100
77	&4D	&X01001101
78	&4E	&X01001110
79	&4F	&X01001111
80	&50	&X01010000
81	&51	&X01010001
82	&52	&X01010010
83	&53	&X01010011
84	&54	&X01010100
85	&55	&X01010101
86	&56	&X01010110
87	&57	&X01010111
88	&58	&X01011000
89	&59	&X01011001
90	&5A	&X01011010
91	&5B	&X01011011
92	&5C	&X01011100
93	&5D	&X01011101
94	&5E	&X01011110
95	&5F	&X01011111
96	&60	&X01100000
97	&61	&X01100001
98	&62	&X01100010
99	&63	&X01100011
100	&64	&X01100100
101	&65	&X01100101
102	&66	&X01100110
103	&67	&X01100111

decimal	hex	binario	decimal	hex	binario
104	&68	&X01101000	130	&82	&X10000010
105	&69	&X01101001	131	&83	&X10000011
106	&6A	&X01101010	132	&84	&X10000100
107	&6B	&X01101011	133	&85	&X10000101
108	&6C	&X01101100	134	&86	&X10000110
109	&6D	&X01101101	135	&87	&X10000111
110	&6E	&X01101110	136	&88	&X10001000
111	&6F	&X01101111	137	&89	&X10001001
112	&70	&X01110000	138	&8A	&X10001010
113	&71	&X01110001	139	&8B	&X10001011
114	&72	&X01110010	140	&8C	&X10001100
115	&73	&X01110011	141	&8D	&X10001101
116	&74	&X01110100	142	&8E	&X10001110
117	&75	&X01110101	143	&8F	&X10001111
118	&76	&X01110110	144	&90	&X10010000
119	&77	&X01110111	145	&91	&X10010001
120	&78	&X01111000	146	&92	&X10010010
121	&79	&X01111001	147	&93	&X10010011
122	&7A	&X01111010	148	&94	&X10010100
123	&7B	&X01111011	149	&95	&X10010101
124	&7C	&X01111100	150	&96	&X10010110
125	&7D	&X01111101	151	&97	&X10010111
126	&7E	&X01111110	152	&98	&X10011000
127	&7F	&X01111111	153	&99	&X10011001
128	&80	&X10000000	154	&9A	&X10011010
129	&81	&X10000001	155	&9B	&X10011011

decimal	hex	binario
156	&9C	&X10011100
157	&9D	&X10011101
158	&9E	&X10011110
159	&9F	&X10011111
160	&A0	&X10100000
161	&A1	&X10100001
162	&A2	&X10100010
163	&A3	&X10100011
164	&A4	&X10100100
165	&A5	&X10100101
166	&A6	&X10100110
167	&A7	&X10100111
168	&A8	&X10101000
169	&A9	&X10101001
170	&AA	&X10101010
171	&AB	&X10101011
172	&AC	&X10101100
173	&AD	&X10101101
174	&AE	&X10101110
175	&AF	&X10101111
176	&B0	&X10110000
177	&B1	&X10110001
178	&B2	&X10110010
179	&B3	&X10110011
180	&B4	&X10110100
181	&B5	&X10110101

decimal	hex	binario
182	&B6	&X10110110
183	&B7	&X10110111
184	&B8	&X10111000
185	&B9	&X10111001
186	&BA	&X10111010
187	&BB	&X10111011
188	&BC	&X10111100
189	&BD	&X10111101
190	&BE	&X10111110
191	&BF	&X10111111
192	&C0	&X11000000
193	&C1	&X11000001
194	&C2	&X11000010
195	&C3	&X11000011
196	&C4	&X11000100
197	&C5	&X11000101
198	&C6	&X11000110
199	&C7	&X11000111
200	&C8	&X11001000
201	&C9	&X11001001
202	&CA	&X11001010
203	&CB	&X11001011
204	&CC	&X11001100
205	&CD	&X11001101
206	&CE	&X11001110
207	&CF	&X11001111

decimal	hex	binario
208	&D0	&X11010000
209	&D1	&X11010001
210	&D2	&X11010010
211	&D3	&X11010011
212	&D4	&X11010100
213	&D5	&X11010101
214	&D6	&X11010110
215	&D7	&X11010111
216	&D8	&X11011000
217	&D9	&X11011001
218	&DA	&X11011010
219	&DB	&X11011011
220	&DC	&X11011100
221	&DD	&X11011101
222	&DE	&X11011110
223	&DF	&X11011111
224	&E0	&X11100000
225	&E1	&X11100001
226	&E2	&X11100010
227	&E3	&X11100011
228	&E4	&X11100100
229	&E5	&X11100101
230	&E6	&X11100110
231	&E7	&X11100111
232	&E8	&X11101000
233	&E9	&X11101001

decimal	hex	binario
234	&EA	&X11101010
235	&EB	&X11101011
236	&EC	&X11101100
237	&ED	&X11101101
238	&EE	&X11101110
239	&EF	&X11101111
240	&F0	&X11110000
241	&F1	&X11110001
242	&F2	&X11110010
243	&F3	&X11110011
244	&F4	&X11110100
245	&F5	&X11110101
246	&F6	&X11110110
247	&F7	&X11110111
248	&F8	&X11111000
249	&F9	&X11111001
250	&FA	&X11111010
251	&FB	&X11111011
252	&FC	&X11111100
253	&FD	&X11111101
254	&FE	&X11111110
255	&FF	&X11111111

	0	1	2	3	4	5	6	7
0	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLCA
1	DJNZ of	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RLA
2	JR NZ,of	LD HL,nn	LD (nn),HL	INC HL	INC H	DEC H	LD H,n	DAA
3	JR NC,of	LD SP,nn	LD (nn),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A
9	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A
A	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A
B	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A
C	RET NZ	POP BC	JP NZ,nn	JP nn	CALL NZ,nn	PUSH BC	ADD A,n	RST & 00
D	RET NC	POP DE	JP NC,nn	OUT (n),A	CALL NC,nn	PUSH DE	SUB n	RST & 10
E	RET PO	POP HL	JP PO,nn	EX (SP),HL	CALL PO,nn	PUSH HL	AND n	RST & 20
F	RET P	POP AF	JP P,nn	DI	CALL P,nn	PUSH AF	OR n	RST & 30

	8	9	A	B	C	D	E	F
0	EX AF,AF	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRCA
1	JR of	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2	JR Z,of	ADD HL,HL	LD HL,(nn)	DEC HL	INC L	DEC L	LD L,n	CPL
3	JR C,of	ADD HL,SP	LD A,(nn)	DEC SP	INC A	DEC A	LD A,n	CCF
4	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
B	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
C	RET Z	RET	JP Z,nn	⟶	CALL Z,nn	CALL nn	ADC A,n	RST & 08
D	RET C	EXX	JP C,nn	IN A,(n)	CALL C,nn	⟶	SBC A,n	RST & 18
E	RET PE	JP (HL)	JP PE,nn	EX DE,HL	CALL PE,nn	⟶	XOR n	RST & 28
F	RET M	LD SP,HL	JP M,nn	EI	CALL M,nn	⟶	CP n	RST & 38

	0	1	2	3	4	5	6	7
0	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A
3								
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A

	8	9	A	B	C	D	E	F
0	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

	0	1	2	3	4	5	6	7
4	IN B,(C)	OUT (C),B	SBC HL,BC	LD (nn),BC	NEG	RETN	IM 0	LD I,A
5	IN D,(C)	OUT (C),P	SBC HL,DE	LD (nn),DE			IM 1	LD A,I
6	IN H,(C)	OUT (C),H	SBC HL,HL	LD (nn),HL				RRD
7			SBC HL,SP	LD (nn),SP				
8								
9								
A	LDI	CPI	INI	OUTI				
	LDIR	CPIR	INIR	OTIR				

	8	9	A	B	C	D	E	F
4	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(nn)		RETI		LD R,A
5	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(nn)			IM 2	LD A,R
6	IN L,(C)	OUT (C),L	ADC HL,HL	LD HL,(nn)				RLD
7	IN A,(C)	OUT (C),A	ADC HL,SP	LD SP,(nn)				
8								
9								
A	LDD	CPD	IND	OUTD				
B	LDDR	CPDR	INDR	OTDR				

EXPLICACION PARA LAS SIGUIENTES TABLAS:

En la primera tabla los códigos &CB, &ED, &DD y &FD están representados por flechas. Esto significa:

&CB: si el código a traducir es &CB debe buscarse el segundo código en la segunda tabla. Estos comandos son los comandos de rotación y desplazamiento.

&ED: si el primer código a traducir es &ED debe buscarse el segundo código en la tercera tabla.

&DD y &FD: si el primer código es &DD o &FD se trata de comandos de dirección indexada. &DD se refiere al registro IX y &FD al registro IY. Los comandos de dirección indexada no se reflejan en otra tabla. Pueden obtenerse de las tablas presentes de la siguiente forma: el segundo código se busca como de costumbre en las tablas, el comando obtenido ha de contener el registro HL. Si el registro HL no aparece en el operando o si se ha obtenido el comando EX DE,HL se trata de un comando inválido (visualizado por el disassembler como !!!). Si se trata de un comando válido debe reemplazarse el registro HL por IX/IY.

HL se convierte en IX/IY.
HL se convierte en (IX + dis)/(IY + dis), donde 'dis' se determina por el tercer código.

Estas reglas son válidas excepto en el comando JP (HL) para todos los comandos que contienen HL. De JP (HL) de todas formas, aunque HL se representa entre paréntesis, se obtiene después de la inserción de los registros índices JP (IX)/JP(IY).

			SOURCE																	
			IMPLIED		REGISTER							REG INDIRECT			INDEXED		EXT ADDR	IMME.
			I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
DESTINATION	REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
		B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
		C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
		D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
		E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
		H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
		L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
	REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
		(BC)			02													
		(DE)			12													
	INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
		(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
	EXT ADDR	(nn)			32 n n													
	IMPLIED	I			ED 47													
		R			ED 4F													

SOURCE

DESTINATION		REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
REGISTER	AF										**F1**
	BC								**01 n n**	ED 4B n n	**C1**
	DE								**11 n n**	ED 5B n n	**D1**
	HL								**21 n n**	**2A n n**	**E1**
	SP				**F9**		DD F9	FD F9	**31 n n**	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	**22 n n**	ED 73 n n	DD 22 n n	FD 22 n n			
REG. IND.	(SP)	**F5**	**C5**	**D5**	**E5**		DD E5	FD E5			

PUSH INSTRUCTIONS →

POP INSTRUCTIONS ↑

NOTE: The Push & Pop Instructions adjust the SP after every execution

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			**EB**		
REG. INDIR.	(SP)			**E3**	DD E3	FD E3

			SOURCE: REG. INDIR. (HL)	
DESTINATION: REG. INDIR. (DE)			ED A0	'LDI' – Load (DE)◄—(HL) Inc HL & DE, Dec BC
			ED B0	'LDIR,' – Load (DE)◄—(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED A8	'LDD' – Load (DE)◄—(HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Load (DE)◄—(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg HL points to source
Reg DE points to destination
Reg BC is byte counter

SEARCH LOCATION: REG. INDIR. (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory
to be compared with accumulator
contents
BC is byte counter

SOURCE

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

SOURCE

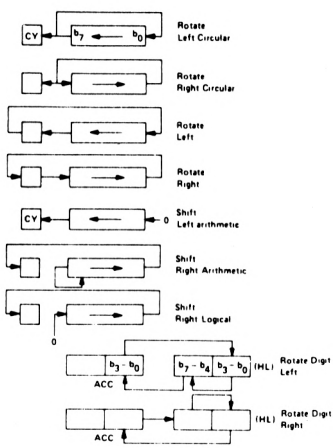
DESTINATION			BC	DE	HL	SP	IX	IY
	'ADD'	HL	09	19	29	39		
		IX	DD 09	DD 19		DD 39	DD 29	
		IY	FD 09	FD 19		FD 39		FD 29
	ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
	SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
	INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
	DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Source and Destination

TYPE OF ROTATE OR SHIFT

	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
'RLD'								ED 6F		
'RRD'								ED 67		

	A
RLCA	07
RRCA	0F
RLA	17
RRA	1F



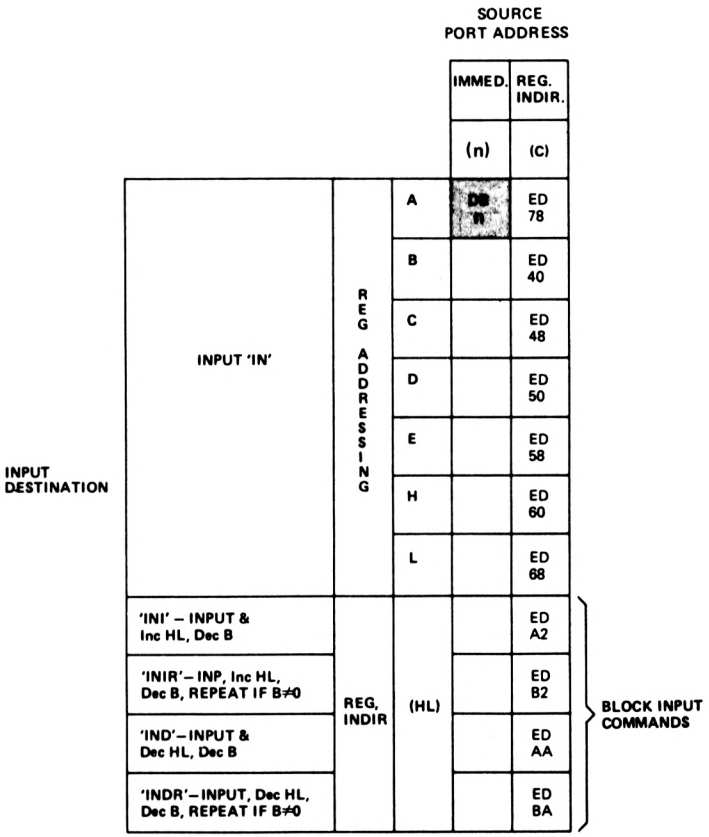
		REGISTER ADDRESSING							REG. INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB. 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6.	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

CONDITION

			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

NOTE—CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

		OP CODE	
CALL ADDRESS	0000_H	C7	'RST 0'
	0008_H	CF	'RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'


SOURCE PORT ADDRESS
IMMED.
REG. INDIR.
(n)
(C)
INPUT DESTINATION
INPUT 'IN'
REG ADDRESSING
A
B
C
D
E
H
L
DB n
ED 78
ED 40
ED 48
ED 50
ED 58
ED 60
ED 68
'INI' – INPUT & Inc HL, Dec B
'INIR'– INP, Inc HL, Dec B, REPEAT IF B≠0
'IND'– INPUT & Dec HL, Dec B
'INDR'– INPUT, Dec HL, Dec B, REPEAT IF B≠0
REG, INDIR
(HL)
ED A2
ED B2
ED AA
ED BA
BLOCK INPUT COMMANDS

			SOURCE REGISTER A	B	C	D	E	H	L	REG. IND. (HL)
'OUT'	IMMED.	(n)	D3 n							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB

BLOCK OUTPUT COMMANDS

PORT DESTINATION ADDRESS

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

Instruction	C	Z	P/V	S	N	H	Comments
ADD A, s; ADC A,s	↕	↕	V	↕	0	↕	8-bit add or add with carry
SUB s; SBC A, s; CP s; NEG	↕	↕	V	↕	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	0	↕	P	↕	0	1	Logical operations
OR s; XOR s	0	↕	P	↕	0	0	And set's different flags
INC s	•	↕	V	↕	0	↕	8-bit increment
DEC m	•	↕	V	↕	1	↕	8-bit decrement
ADD DD, ss	↕	•	•	•	0	X	16-bit add
ADC HL, ss	↕	↕	V	↕	0	X	16-bit add with carry
SBC HL, ss	↕	↕	V	↕	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	↕	•	•	•	0	0	Rotate accumulator
RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	↕	↕	P	↕	0	0	Rotate and shift location m
RLD, RRD	•	↕	P	↕	0	0	Rotate digit left and right
DAA	↕	↕	P	↕	•	↕	Decimal adjust accumulator
CPL	•	•	•	•	1	1	Complement accumulator
SCF	1	•	•	•	0	0	Set carry
CCF	↕	•	•	•	0	X	Complement carry
IN r, (C)	•	↕	P	↕	0	0	Input register indirect
INI; IND; OUTI; OUTD	•	↕	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	•	X	↕	X	0	0	Block transfer instructions
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	•	↕	↕	↕	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	•	↕	IFF	↕	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	•	↕	X	X	0	1	The state of bit b of location s is copied into the Z flag
NEG	↕	↕	V	↕	1	↕	Negate accumulator

The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from into bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
↕	The flag is affected according to the result of the operation.
•	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction. .
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.

COMMODORE

Ofrece un campo fascinante y amplio de problemáticas científicas. Para esto el libro contiene muchos listados interesantes: Análisis de Fornier y síntesis, análisis de redes, exactitud de cálculo, formateado de números, cálculo del valor PH, sistemas de ecuaciones diferenciales, modelo ladrón presa, cálculo de probabilidad, medición de tiempo, integración, etc.

64 en el campo de la Técnica y la Ciencia. 361 págs. P.V.P. 2.800,- ptas.

La obra Standard del floppy 1541, todo sobre la programación en disquettes desde los principiantes a los profesionales, además de las informaciones fundamentales para el DOS, los comandos de sistema y mensajes de error, hay varios capítulos para la administración práctica de ficheros con el FLOPPY, amplio y documentado Listado del Dos. Además un filón de los más diversos programas y rutinas auxiliares, que hacen del libro una lectura obligada para los usuarios del Floppy.

Todo sobre el Floppy 1541. Precio venta 3.200 ptas.

Un excelente libro, que le mostrará todas las posibilidades que le ofrece su grabadora de casettes. Describe detalladamente, y de forma comprensible, todo sobre el Datassette y la grabación en cassette. Con verdaderos programas fuera de serie: Autostart, Catálogo (¡busca y carga automáticamente!), backup de y a disco, SAVE de áreas de memoria, y lo más sorprendente: un nuevo sistema operativo de cassette con el 10-20 veces más rápido Fast Tape. Además otras indicaciones y programas de utilidad (ajuste de cabezales, altavoz de control).

El Manual del Cassette. 190 pág. P.V.P. 1.600,- ptas.

¡Por fin una introducción al código máquina fácilmente comprensible! Estructura y funcionamiento del procesador 6510, introducción y ejecución de programas en lenguaje máquina, manejo del ensamblador, y un simulador de paso a paso escrito en BASIC.

Lenguaje máquina para Commodore 64. 1984, 201 pág. P.V.P. 2.200,- ptas.

CONSEJOS Y TRUCOS, con más de 70.000 ejemplares vendidos en Alemania, es uno de los libros más vendidos de DATA BECKER. Es una colección muy interesante de ideas para la programación del Commodore 64, de POKEs y útiles rutinas e interesantes programas. Todos los programas en lenguaje máquina con programas cargadores en Basic.

64 Consejos y Trucos. 1984, 364 pág. P.V.P. 2.800,- ptas.

Este libro, contiene muchos interesantes programas de aprendizaje para solucionar problemas, descritos detalladamente y de manera fácilmente comprensible. Temas: progresiones geométricas, palanca mecánica, crecimiento exponencial, verbos irregulares, ecuaciones de segundo grado, movimientos de péndulo, formación de moléculas, aprendizaje de vocablos, cálculo de interés y su capitalización.

Manual escolar para su Commodore 64. 389 págs. P.V.P. 2.800,- ptas.

En el libro de los robots se muestran las asombrosas posibilidades que ofrece el CBM 64, para el control y la programación, presentadas con numerosas ilustraciones e intuitivos ejemplos. El punto principal: Cómo puede construirse uno mismo un robot sin grandes gastos. Además, un resumen del desarrollo histórico del robot y una amplia introducción a los fundamentos cibernéticos. Gobierno del motor, el modelo de simulación, interruptor de pantalla, el Port-Usuario cómodo del modelo de simulación, Sensor de infrarrojos, concepto básico de un robot, realimentación unidad cibernética, Brazo prensor, Oír y ver.

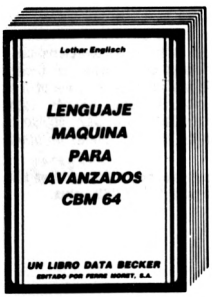
Robótica para su Commodore 64. 340 págs. P.V.P. 2.800 ptas.

Saberse apañar uno mismo, ahorra tiempo, molestias y dinero, precisamente problemas como el ajuste del floppy o reparaciones de la platina se pueden arreglar a menudo con medios sencillos. Instrucciones para eliminar la mayoría de perturbaciones, listas de piezas de recambio y una introducción a la mecánica y a la electrónica de la unidad de disco, hay también indicaciones exactas sobre herramientas y material de trabajo. Este libro hay que considerarlo en todos sus aspectos como efectivo y barato.

Mantenimiento y reparación del Floppy 1541. 325 págs. P.V.P. 2.800,- ptas.

Este es el libro que buscaba: un diccionario general de micros que contiene toda la terminología informática de la A a la Z y un diccionario técnico con traducciones de los términos ingleses de más importancia - los DICCIONARIOS DATA BECKER prácticamente son tres libros en uno. La increíble cantidad de información que contienen, no sólo los convierte en enciclopedias altamente competente, sino también en herramientas indispensables para el trabajo. El DICCIONARIO DATA BECKER se edita en versión especial para APPLE II, COMMODORE 64 e IBM PC.
El diccionario para su Commodore 64. 350 pág. P.V.P. 2.800,- ptas.

Casi todo lo que se puede hacer con el Commodore 64, está descrito detalladamente en este libro. Su lectura no es tan sólo tan apasionante como la de una novela, sino que contiene, además de listados de útiles programas, sobre todo muchas, muchas aplicaciones realizables en el C64. En parte hay listados de programas listos para ser tecleados, siempre que ha sido posible condensar «recetas» en una o dos páginas. Si hasta el momento no sabía que hacer con su Commodore 64, ¡después de leer este libro lo sabrá seguro!
El libro de ideas del Commodore 64. 1984, más de 200 pàginas, P.V.P. 1.600,- ptas.

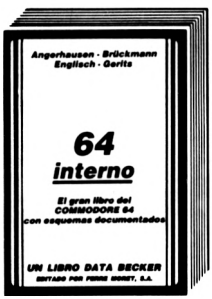


¿Ud. ha logrado iniciarse en código máquina? Entonces el «nuevo English» le enseñará cómo convertirse en un profesional. Naturalmente con muchos programas ejemplo, rutinas completas en código máquina e importantes consejos y trucos para la programación en lenguaje máquina y para el trabajo con el sistema operativo.
Lenguaje máquina para avanzados CBM 64. 1984, 206 pàg. P.V.P. 2.200 ptas.

Este libro ofrece una amplia práctica introducción en el importante tema de la gestión de ficheros y bancos de datos, especialmente para los usuarios del Commodore 64. Con muchas interesantes rutinas y una confortable gestión de ficheros.
Todo sobre bases de datos y gestión de ficheros para Commodore-64. 221 págs. P.V.P. 2.200,- ptas.

Gráficos para el Commodore 64 es un libro para todos los que quieren hacer algo creativo con su ordenador. El contenido abarca desde los fundamentos de la programación de gráficos hasta el diseño asistido por ordenador (CAD).
Gráficos para el Commodore 64. 295 pags. P.V.P. 2.200,- ptas.

Para los usuarios que posean un VIC-20, C-64 o PC-128 este libro contiene gran cantidad de consejos, trucos, listados de programas, así como información sobre Hardware, tanto si usted dispone de una impresora de margarita o de matriz, como si tiene un Plotter VC-1520, el GRAN LIBRO DE IMPRESORAS constituye una inestimable fuente de información.
Todo sobre impresoras. 361 págs. P.V.P. 2.800,- ptas.



Con más de 60.000 ejemplares vendidos, ésta es la obra estándar para el COMMODORE 64. Todo sobre la tecnología, el sistema operativo y la programación avanzada del C-64. Con listado completo y exhaustivo de la ROM, circuitos originales documentados y muchos programas. ¡Conozca su C-64 a fondo!
64 Interno. 1984, 352 pág. P.V.P. 3.800,- ptas.

Con importantes comandos PEEK y POKE se pueden hacer también desde el Basic muchas cosas, para las que se necesitarían normalmente complejas rutinas en lenguaje máquina. Con una enorme cantidad de POKEs importantes y su posible aplicación. Para ello se explica perfectamente la estructura del Commodore 64: Sistema operativo, interpretador, página cero, apuntadores y stacks, generador de caracteres, registros de sprites, programación de interfaces, desactivación de interrupt. Además una introducción al lenguaje máquina. Muchos programas ejemplo.
PEEKS y POKES. 177 pág. P.V.P. 1.600,- ptas.

Este libro presenta una detallada e interesante introducción a la teoria, conceptos básicos y posibilidades de uso de la inteligencia artificial (1A). Desde un resumen histórico sobre las máquinas «pensantes» y «vivientes» hasta programas de aplicación para el Commodore 64.

Inteligencia artificial. 395 págs. 2.800,- ptas.

64, Consejos y Trucos vol. 2 contiene una gran profusión de programas, estímulos y muchas rutinas útiles. Un libro que constituye una ayuda imprescindible para todo aquél que quiera escribir programas propios con el COMMODORE.

Consejos y Trucos, Commodore 64. Vol. 2. 259 págs. 2.200,- ptas.

Este libro ofrece al programador interesado una introducción fácilmente comprensible para los tan extendidos Assembler PROFI-ASS, SM MAE y T.E-X.ASS. con consejos y trucos de gran utilidad, indicaciones y programas adicionales. Al mismo tiempo sirve de manual orientado a la práctica, con aclaraciones de conceptos importantes e instrucciones.

El Ensamblador. 250 páginas. 2.200,- ptas.

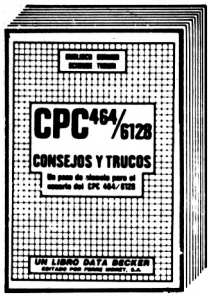
El libro de Primicias del Commodore 128 no ofrece solamente un resumen completo de todas las características y rendimientos del sucesor del C-64 y con ello una importante ayuda para su adquisición. Muestra, además, todas las posibilidades del nuevo equipo en función de sus tres modos de operación.

Todo sobre el nuevo Commodore 128. 250 págs. P.V.P. 2.200,- ptas.

El libro Commodore 128-Consejos y Trucos es un filón para cualquier poseedor del C-128 que desee sacar más partido a su ordenador. Este libro no sólo contiene gran cantidad de programas-ejemplo, sino que además explica de un modo sencillo y fácil la configuración del ordenador y de su programación.

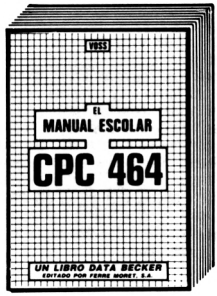
Commodore 128-Consejos y Trucos. 327 págs. 2.800,- ptas.

AMSTRAD



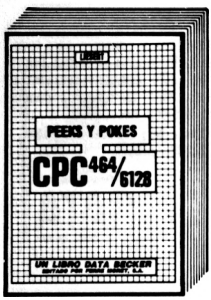
Ofrece una colección muy interesante de sugerencias, ideas y soluciones para la programación y utilización de su CPC-464: Desde la estructura del hardware, sistema de funcionamiento - Tokens Basic, dibujos con el joystick, aplicaciones de ventanas en pantalla y otros muchos interesantes programas como el procesamiento de datos, editor de sonidos, generador de caracteres, monitor de código máquina hasta listados de interesantes juegos.

CPC-464 Consejos y Trucos. 263 págs. P.V.P. 2.200,- ptas.



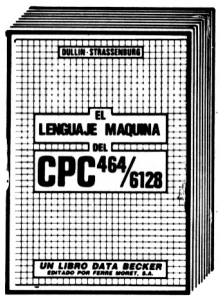
Escrito para alumnos de los últimos cursos de EGB y de BUP, este libro contiene muchos programas para resolver problemas y de aprendizaje, descritos de una forma muy compleja y fácil de comprender. Teorema de Pitágoras, progresiones geométricas, escritura cifrada, crecimiento exponencial, verbos irregulares, igualdades cuadráticas, movimiento pendular, estructura de moléculas, cálculo de interés y muchas cosas más.

CPC-464 El libro del colegio. 380 págs. P.V.P. 2.200,- ptas.



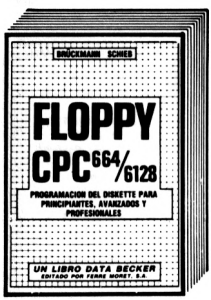
PEEKS, POKES y CALLS se utilizan para introducir al lector de una forma fácilmente accesible al sistema operativo y al lenguaje máquina del CPC. Proporciona además muchas e interesantes posibilidades de aplicación y programación de su CPC.

PEEKS y POKES del CPC 464/6128. 180 pág. P.V.P. 1.600,- ptas.



El libro del lenguaje máquina para el CPC 464/6128 está pensado para todos aquellos a quienes no les resulta suficiente con las posibilidades y rapidez del BASIC. Se explican aquí detalladamente las bases de la programación en lenguaje máquina, el funcionamiento del procesador Z-80 con sus respectivos comandos así como la utilización de las rutinas del sistema con abundantes ejemplos. El libro contiene programas completos de aplicación tales como Ensamblador. Desensamblador y Monitor, facilitando de esta manera la introducción del lector en el lenguaje máquina.

El Lenguaje Máquina del CPC 464/ 6128. 330 pág. P.V.P. 2.200,- ptas.



El LIBRO DEL FLOPPY del CPC lo explica todo sobre la programación con discos y la gestión relativa de ficheros mediante el floppy DDI-1 y la unidad de discos incorporada del CPC 664/6128. La presente obra, un auténtico stándart, representa una ayuda incomparable tanto para el que desee iniciarse en la programación con discos cómo para el más curtido programador de ensamblados. Especialmente interesante resulta el listado exhaustivamente comentado del DOS y los muchos programas de ejemplo, entre los que se incluye un completo paquete de gestión de ficheros.

El Libro del Floppy del CPC. 353 pág. P.V.P. 2.800,- ptas.

¡Dominar CP/M por fin! Desde explicaciones básicas para almacenar números, la protección contra la escritura, o ASCII, hasta la aplicación de programas auxiliares de CP/M, así como «CP/M interno» para avanzados, cada usuario del CPC rápidamente encontrará las ayudas e informaciones necesarias, para el trabajo con CP/M. Este libro tiene en cuenta las versiones CP/M 2.2, así como CP/M Plus (3.0), para el AMSTRAD CPC 464, CPC 664 y CPC 6128.

CP/M. El libro de ejercicios para CPC. 260 pág. P.V.P. 2.800,- ptas.

MSX

Escrito para alumnos de los últimos cursos de EGB y de BUP, este libro contiene muchos programas para resolver problemas y de aprendizaje, descritos de una forma muy completa y fácil de comprender. Teorema de Pitágoras, progresiones geométricas, escritura cifrada, crecimiento exponencial, verbos irregulares, igualdades cuadráticas, movimiento pendular, estructura de moléculas, cálculo de interés y muchas cosas más.

MSX el Manual Escolar. 389 págs. P.V.P. 2.800,- ptas.

El libro contiene una amplia colección de importantes programas que abarcan, desde un desensamblador hasta un programa de clasificaciones deportivas. Juegos superemocionantes y aplicaciones completas. Los programas muestran además importantes consejos y trucos para la programación. Estos programas funcionan en todos los ordenadores MSX, así como en el SPECTROVIDEO 318 328.

MSX Programas y Utilidades, 1985, 194 pág. P.V.P. 2.200,- ptas.

Las computadoras MSX no sólo ofrecen una relación precio/rendimiento sobresaliente, sino que también poseen unas cualidades gráficas y de sonido excepcionales. Este libro expone las posibilidades de los MSX de forma completa y fácil. El texto se completa con numerosos y útiles programas ejemplo.

MSX Gráficos y Sonidos, 250 págs. P.V.P. 2.800,- ptas.

Este libro contine una colección sin igual de trucos y consejos para todos los ordenadores con la nueva norma MSX. No sólo contiene las recetas completas, sino también los conocimientos básicos necesarios.

MSX - Consejos y Trucos. 288 págs. P.V.P. 2.200,- ptas.

El libro del Lenguaje Máquina para el MSX está creado para todos aquellos a quienes el BASIC se les ha quedado pequeño en cuanto a rendimiento y velocidad. Desde las bases para la programación en Lenguaje Máquina, pasando por el método de trabajo del Procesador Z-80 y una exacta descripción de sus órdenes, hasta la utilización de rutinas del sistema todo ello ha sido explicado en detalle e ilustrado con múltiples ejemplos en este libro.
El libro contiene, además, como programas de aplicación, un ensamblador un desensamblador y un monitor.

MSX Lenguaje Máquina. 306 págs. 2.200,- ptas.

ZX SPECTRUM

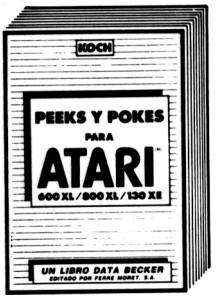
Una interesante colección de sugestivas ideas y soluciones para la programación y utilización de su ZX ESPECTRUM. Aparte de muchos peeks, pokes y USRs hay también capítulos completos para, entre otros, entrada de datos asegurado sin bloqueo de ordenador, posibilidades de conexión y utilización de microdrives y lápices ópticos, programas para la representación de diagramas de barra y de tarta, el modo de utilizar óptimamente ROM y RAM.

ZX Spectrum Consejos y Trucos, 211 pág. P.V.P. 2.200,- ptas.

Escrito para alumnos de los últimos cursos de EGB y de BUP, este libro contiene muchos programas para resolver problemas y de aprendizaje, descritos de una forma muy completa y fácil de comprender. Teorema de Pitágoras, progresiones geométricas, escritura cifrada, crecimiento exponencial, verbos irregulares, igualdades cuadráticas, movimiento pendular, estructura de moléculas, cálculo de interés y muchas cosas más.

ZX Spectrum el Manual Escolar. 389 págs. P.V.P. 2.200,- ptas.

ATARI



Tan interesante como el tema, es el libro que explica de forma fácilmente comprensible el manejo de Peeks y Pokes importantes, y representa un gran número de Pokes con sus posibilidades de aplicación, incluyendo además programas ejemplo. Al lado de temas como lo son la memoria de la pantalla, los bits y los bytes, el mapa de la memoria, la tabla de modos gráficos o el sonido, también se detalla de forma magnífica la estructura del ATARI 600XL/800XL/130XE.

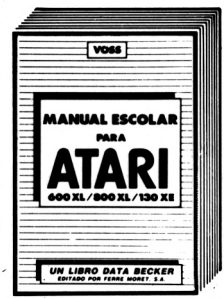
Peeks y Pokes para ATARI 600XL/800XL/130XE. 251 pág. P.V.P. 2.200, ptas.

Una lograda introducción al sugestivo tema de los «juegos estratégicos» Desde juegos sencillos con estrategia fija a juegos complejos con procedimientos de búsqueda hasta programas con capacidad de aprendizaje —muchos ejemplos interesantes, escritos por supuesto de forma fácilmente comprensible. Con programas de juegos ampliamente detallados: NIM con un montón, bloqueo, hexapawn, mini-damas y muchos más.

Juegos estratégicos y cómo programarlos en el ATARI 600XL/800XL/130XE. 181 pág. P.V.P. 1.600,- ptas.

Jugar a aventuras con éxito y programarlas uno mismo - todo lo verdaderamente importante sobre el tema, lo contiene este guía fascinante que te lleva a través del mundo fantástico de las aventuras. El libro abarca todo el espectro, hasta las más sofisticadas aventuras gráficas llenas de trucos, acompañándolas siempre de numerosos programas ejemplo. Sin embargo la clave —al margen de muchas aventuras para teclear— es un generador de aventuras completo, mediante el cual la programación de aventuras se convierte en un juego de niños.

Aventuras - y cómo programarlas en el ATARI 600XL/800XL/130XE. 284 pág. P.V.P. 2.200,- ptas.



Muchos programas interesantes de soluciones de problemas y de aprendizaje, descritos de forma amplia y comprensible, y adecuados sobre todo para escolares. ¡Aquí el aprendizaje intensivo se convierte de una tarea divertida! Al margen de temas como los verbos irregulares, o las ecuaciones de segundo grado, un resumen corto de las bases del tratamiento electrónico de datos, y una introducción a los principios del análisis de problemas, completan este libro que debería obrar en posesión de cualquier escolar.

El libro escolar para ATARI 600XL/800XL/130XE. 389 pág. P.V.P. 2.800,- ptas.

OTROS TITULOS

El primer libro recomendado para escuelas de enseñanza de informática y para aquellas personas que quieren aprender la programación. Cubre las especificaciones del Ministerio de Educación y Ciencia para Estudios de Informática. Es el primer libro que introduce a la lógica del ordenador. Es un elemento de base que sirve como introducción para la programación en cualquier otro lenguaje. No se requieren conocimientos de programación ni siquiera de informática. Abarca desde los métodos de programación clásicos a los más modernos.

Metodología de la Programación. 250 págs. P.V.P. 2.200,- ptas.

La técnica y programación del Procesador Z80 son los temas de este libro. Es un libro de estudio y de consulta imprescindible para todos aquellos que poseen un Commodore 128, CPC, MSX u otros ordenadores que trabajan con el Procesador Z80 y desean programar en lenguaje máquina.

El Procesador Z80. 560 pág. P.V.P. 3.800,- ptas.

El tema de este libro es la técnica y programación de los procesadores de la familia 68000. Es una obra de consulta indispensable, un manual para todo programador que quiera utilizar las ventajas del 68000.

Técnica y programación para el procesador 68000. 516 págs. P.V.P. 3.800,- ptas.

EL CONTENIDO:

¡Así es como se facilita el acceso al lenguaje máquina! El libro de lenguaje máquina para CPC 464, 664 y 6128 es importante para cualquier usuario, que realmente desee convertirse en un experto.

Extracto del contenido:

- Qué es el lenguaje máquina
- Los sistemas numéricos
- Estructura del ordenador
- El procesador Z80
- Estructura de la CPU
- El conjunto de comandos del Z80 con explicaciones detalladas
- Programación del procesador Z80
- Ensamblador completo para copiar con descripción del programa
- Desensamblador y simulador paso a paso
- Importantes rutinas de sistema para la creación de programas
- Monitor de lenguaje máquina con listado

ESTE LIBRO HA SIDO ESCRITO POR:

Holger Dullin y Hardy Strassenburg, estudiantes de Biología y de Física y expertos programadores, que se procuraron uno de los primeros CPC y se lanzaron inmediatamente a la programación del procesador Z80.
Además son autores de los siguientes libros de DATA BECKER: MSX Lenguaje Máquina, MSX Consejos y Trucos y Todo sobre Impresoras EPSON.

ISBN 84-86437-35-0

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/